

Einsatz von CORBA und Java für verteilte Client/Server-Anwendungen im Internet

Studienarbeit

von
Matthias Wipf

Betreuer:

Dipl.-Inform. Arne Koschel
Dipl.-Inform. Ralf Nikolai

verantwortlicher Betreuer:

Prof. Dr. P.C. Lockemann

**Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe**

Juli 1997

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufgabenstellung	2
1.3	Übersicht über die Kapitel	2
2	Grundlagen	3
2.1	CORBA	3
2.2	Java	5
2.2.1	Historie	5
2.2.2	Überblick	5
2.2.3	Ausblick	7
2.3	CORBA & Java	8
2.4	Vergleich von CORBA mit RMI	9
3	Produktübersicht	13
3.1	VisiBroker for Java (Visigenic Software Inc.)	13
3.1.1	Informationen	13
3.1.2	Verfügbarkeit	13
3.1.3	Funktionalität	14
3.1.4	Besonderheiten	16
3.1.5	Gesamteindruck	17
3.2	OrbixWeb (IONA Technologies Ltd.)	17
3.2.1	Informationen	17
3.2.2	Verfügbarkeit	18
3.2.3	Funktionalität	18
3.2.4	Besonderheiten	19
3.2.5	Gesamteindruck	20
3.3	Joe (SunSoft)	21
3.3.1	Informationen	21
3.3.2	Verfügbarkeit	21
3.3.3	Funktionalität	21
3.3.4	Gesamteindruck	23
3.4	PowerBroker CORBAplus Java Edition (Expersoft)	23

3.4.1	Informationen	23
3.4.2	Verfügbarkeit	23
3.4.3	Funktionalität	24
3.4.4	Gesamteindruck	24
3.5	JacORB (Freie Universität Berlin)	24
3.5.1	Informationen	24
3.5.2	Verfügbarkeit	25
3.5.3	Funktionalität	25
3.5.4	Gesamteindruck	25
3.6	JIDL (Sandia National Labs)	25
3.6.1	Informationen	25
3.6.2	Verfügbarkeit	26
3.6.3	Funktionalität	26
3.6.4	Gesamteindruck	26
3.7	Spring Java IDL (Sun)	26
3.7.1	Informationen	26
3.7.2	Verfügbarkeit	26
3.7.3	Funktionalität	27
3.7.4	Gesamteindruck	27
3.8	Java IDL / Remote Objects for Java (JavaSoft)	27
3.8.1	Informationen	27
3.8.2	Verfügbarkeit	27
3.8.3	Funktionalität	28
3.8.4	Gesamteindruck	29
3.9	HORB (Electrotechnical Lab. / Dr. Hirano Satoshi)	29
3.9.1	Informationen	29
3.9.2	Verfügbarkeit	29
3.9.3	Funktionalität	29
3.9.4	Vergleich mit CORBA 2.0	31
3.9.5	Gesamteindruck	31
3.10	JYLU (Stanford Digital Library Testbed Development)	31
3.10.1	Informationen	31
3.10.2	Verfügbarkeit	31
3.10.3	Funktionalität	32
3.10.4	Gesamteindruck	32
3.11	Jade (Architecture Projects Management Ltd.)	32
3.11.1	Informationen	32
3.11.2	Verfügbarkeit	33

3.11.3	Funktionalität	33
3.11.4	Gesamteindruck	33
3.12	Zusammenfassung	33
4	Kommunikation zwischen ORBs unterschiedlicher Hersteller	37
4.1	Die ORB Interoperabilitäts-Architektur	37
4.1.1	Inter-ORB Bridges	37
4.1.2	Das Interoperabilitäts-Protokoll	38
4.2	Ein Beispiel	39
5	Einsatz im Rahmen von WWW-UIS	43
5.1	Datenbankzugriff mit OrbixWeb 1.0	43
5.2	Ereignis-Monitor mit VisiBroker for Java 1.2	43
6	Erfahrungen bei der Realisierung	49
6.1	Erfahrungen mit Java	49
6.1.1	Die Umgebungsvariable CLASSPATH	49
6.1.2	Applet Viewer contra Netscape Navigator	49
6.1.3	Debugging von Applets im Netscape Navigator	50
6.2	Erfahrungen mit Java ORBs	51
6.2.1	Bereitstellen der benötigten Java Klassen	51
6.2.2	Beschleunigung des Ladevorgangs der ORB Klassen	51
6.2.3	VisiBroker for Java im Netscape Navigator 4 (Communicator)	52
7	Zusammenfassung und Ausblick	53
8	Literaturverzeichnis	55
Anhang A	Hinweise zur Installation	59
A.1	VisiBroker for Java 1.2	59
A.2	OrbixWeb 1.0	60
A.3	OrbixWeb 2.0 beta2	60
Anhang B	Erstellen einer einfachen Client/Server Anwendung mit Java	63
B.1	VisiBroker for Java 1.2	63
B.2	OrbixWeb 2.0	67
B.3	Java IDL	69
B.4	RMI	71
B.5	Caffeine	74
Anhang C	Quelltexte der Beispiel-Anwendung „Ereignis-Monitor“	75
C.1	IDL Beschreibung	75
C.2	Client Applet (VisiBroker for Java)	76
C.3	Server (Orbix)	79

1 Einleitung

1.1 Motivation

Das Internet und speziell das World Wide Web [WWW] erfreuen sich immer größerer Beliebtheit, da sie dem Benutzer einen einfachen Zugriff auf große Mengen von Informationen und Dienstleistungen ermöglichen. Aufgrund des exponentiellen Wachstums wird das WWW von vielen Experten als großer Zukunftsmarkt angesehen, was sich auch an der wachsenden Anzahl der sich beteiligenden Institutionen und Unternehmen zeigt. Es gibt inzwischen kaum ein Unternehmen, das seine Dienste dem Kunden nicht über einen eigenen Web-Server anbietet.

Bisher hat der Benutzer die Möglichkeit mit Hilfe eines WWW-Browsers auf statische Text- und Grafikseiten zugreifen, die mit der Hypertext Markup Language (HTML) geschrieben sind und von jedem HTML-fähigen Web-Browser dargestellt werden können. Der wesentliche Nachteil von HTML ist das Fehlen interaktiver Komponenten, also Elementen, die dem Benutzer das aktive Eingreifen ermöglichen. Deshalb wurden auch schon früh HTML-Erweiterungen wie Formelemente hinzugefügt. Sie erlauben es anhand von Benutzereingaben mit Hilfe von CGI-Scripts (Common Gateway Interface) dynamische HTML-Seiten zu erstellen. Das CGI ermöglicht es Clients, Programme auf WWW-Servern auszuführen, die häufig in einer Skriptsprache wie z.B. Perl implementiert sind. Als Parameter können allerdings nur Strings übergeben werden, weshalb komplexere Datentypen erst codiert und vom aufgerufenen Programm wieder decodiert werden müssen. Eine Typüberprüfung der Parameter ist deshalb nicht möglich. Zudem erzeugt das CGI einen nicht unerheblichen Kommunikationsaufwand, da das Ergebnis eines Aufrufs eine komplette HTML-Seite und nicht nur einzelne Rückgabeparameter sind.

Eine mögliche Lösung für diese Problematik stellt das von der Firma Sun Microsystems entwickelte Java dar. Java ist eine objektorientierte Programmiersprache mit der sogenannte Applets erstellt werden können, die in HTML-Seiten eingebunden und von Java-fähigen Browsern ausgeführt werden können. Darüber hinaus lassen sich mit Java aber auch eigenständige Applikationen erstellen, wie es mit anderen Sprachen wie z.B. C und C++ möglich ist. Die wesentlichen Stärken von Java sind seine Portabilität und die zugehörigen, umfangreichen Klassenbibliotheken, die z.B. die Konstruktion von graphischen Benutzeroberflächen unterstützen und die Entwicklung komplexer Anwendungen ermöglichen. Die Funktionalität, die bisher nur von der Serverseite erbracht wurde, kann nun auch teilweise in den Client verlagert werden. Durch die Nutzung der client-seitigen Rechenleistung wird die Generierung von dynamischen HTML-Seiten überflüssig und der Server entlastet. Eine Schwäche von Java besteht zumindest bis zur Version 1.0.2 darin, daß es kein richtiges Sprachkonzept zum Erstellen von verteilten Client/Server-Applikationen gibt.

Auf diesem Gebiet hat sich die CORBA-Spezifikation der OMG (Object Management Group) als herstellerübergreifender Standard etabliert, die inzwischen in der Revision 2.0 vorliegt. Mittels CORBA (Common Object Request Broker Architecture) lassen sich verteilte Anwendungen in heterogenen Umgebungen realisieren, wobei auch eine Integration bereits bestehender Applikationen mit Hilfe von Objekt-Adaptern möglich ist.

Die Kombination von Java und CORBA bietet nun die Möglichkeit, skalierbare und leistungsfähige Client/Server-Anwendungen für Intra- und Internet zu entwickeln. In Java implementierte ORBs lassen sich in den Web-Browser laden und ermöglichen den Zugriff auf CORBA Dienste und verteilte Objekte. Viele Hersteller von CORBA Implementierungen haben das

Potential dieses Konzeptes bereits erkannt und bieten inzwischen eine Java-Anbindung für ihre Produkte an.

1.2 Aufgabenstellung

Im Rahmen dieser Studienarbeit soll eine Übersicht über alle derzeit verfügbaren Produkte gewonnen werden, die den CORBA Standard mit der Programmiersprache Java verbinden. Ziel ist es, die gebotenen Möglichkeiten und Grenzen der Produkte zu ermitteln, sowie Konzepte und Entwicklungsstand darzustellen und Stärken und Schwächen zu bewerten.

Zur Untersuchung der praktischen Verwendbarkeit im Rahmen des WWW-UIS Projektes sollen mit den am FZI zur Verfügung stehenden Produkten kleine Beispielanwendungen implementiert werden. Inhalte dieser Anwendungen sind u.a. die Kombination bestehender CORBA-Objekte zu Datenbankzugriffen und ereignisbasierter (aktiver) Mechanismen, die hier mit einer einfachen Java-Oberfläche zu versehen sind.

Als weiterer Punkt soll untersucht werden, wie CORBA Implementierungen verschiedener Hersteller mittels des Internet Inter-ORB Protokolls kommunizieren. Die sich dabei ergebenden Probleme sollen erörtert und mögliche Lösungswege gefunden werden.

1.3 Übersicht über die Kapitel

Diese Arbeit ist folgendermaßen gegliedert:

Kapitel 2 gibt eine kurze Einführung in die grundlegenden Konzepte von CORBA und Java und zeigt die Möglichkeiten auf, die sich durch Kombination der beiden Konzepte ergeben. Ein Vergleich der konkurrierenden Konzepte zur Erstellung von Client/Server-Anwendungen mit Java schließt sich an.

Das dritte Kapitel befaßt sich mit den derzeit verfügbaren Produkten, die einen Zugriff auf verteilte Objekte mittels CORBA und Java ermöglichen. Untersucht werden dabei die Funktionalität, Entwicklungsstand, Konzepte und Verfügbarkeit. An Hand dieser Kriterien wird - soweit möglich - eine Bewertung vorgenommen.

Im vierten Kapitel wird auf die Möglichkeit der Kommunikation zwischen ORBs unterschiedlicher Hersteller über das im CORBA 2.0 Standard definierte Internet Inter-ORB Protokoll (IIOP) eingegangen.

In Kapitel 5 werden die Beispiel-Szenarien für das WWW-UIS beschrieben, die mit Hilfe der verfügbaren CORBA/Java Produkte programmiert wurden. Details der Implementierung werden beschrieben.

Die bei der Entwicklung der Beispielprogramme aufgetretenen Probleme und die allgemeinen Erfahrungen im praktischen Umgang mit Java ORBs werden in Kapitel 6 beschrieben.

Kapitel 7 schließt diese Arbeit mit einer Zusammenfassung und einem Ausblick ab.

2 Grundlagen

Dieses Kapitel soll dem Leser einen kurzen Überblick über die Konzepte von CORBA und Java verschaffen und die grundsätzlichen Möglichkeiten aufzeigen, die sich durch eine Kombination der beiden Technologien bieten. Anschließend erfolgt ein Vergleich mit Java RMI, dem konkurrierenden Konzept zur Erstellung von verteilten Anwendungen mit Java.

2.1 CORBA

Durch die zunehmende Vernetzung von EDV-Systemen und dem daraus resultierenden Kommunikationsaufwand werden Konzepte benötigt, die das Zusammenspiel von verteilten Objekten regeln und die Heterogenität der bestehenden Hard- und Software verdecken. Die von der Object Management Group (OMG) spezifizierte Common Object Request Broker Architecture (CORBA) hat sich inzwischen als herstellerübergreifender Standard etabliert, der eine Architektur für die Verteilung und Zusammenarbeit von objektorientierten Softwarebausteinen in heterogenen und vernetzten Systemen vorsieht. Die OMG wurde 1990 gegründet und ist ein herstellerübergreifendes Konsortium mit mehr als 700 Mitgliedern. Ziele der OMG sind die Förderung objektorientierter Softwareansätze und die Schaffung einer Architektur zur Entwicklung verteilter objektorientierter Anwendungen. Die CORBA Spezifikation liegt derzeit in der Revision 2.0 vor [OMG95a], in der vor allem Vorgaben zur Interoperabilität von Produkten unterschiedlicher Hersteller gemacht werden. Darauf wird in Kapitel 4 gesondert eingegangen.

Object Request Broker

Die zentrale Komponente dieser Architektur ist der Object Request Broker (ORB), der für die Kommunikation von Client- und Serverobjekten zuständig ist. Möchte ein Client eine Methode eines entfernten Objektes aufrufen, das sich nicht im eigenen Adressraum befindet, so wird dieser Aufruf an den ORB weitergeleitet. Dieser ist für die Lokalisierung des Zielobjektes zuständig und übermittelt anschließend den eigentlichen Aufruf. Gegebenenfalls sorgt er auch dafür, daß der angeforderte Server gestartet wird. Die resultierenden Ergebniswerte werden ebenso vom ORB an den Client zurückgeleitet. Somit ermöglicht der ORB die Interoperabilität von Anwendungen auf verschiedenen Rechnern in heterogenen verteilten Umgebungen, in dem der Aufenthaltsort von Objekten gekapselt wird.

Interface Definition Language (IDL)

Ein weiteres wesentliches Entwurfsziel der CORBA-Architektur ist die Programmiersprachenunabhängigkeit von CORBA-Objekten und deren Clients [Stal95]. Für die Definition von Schnittstellen wurde eine eigene Sprache entwickelt, die den Namen Interface Definition Language (IDL) trägt. Die IDL ist eine reine Beschreibungssprache und definiert die Methoden, Parameter und Rückgabewerte von Objekten unabhängig von deren Implementierungssprache. In ihrer Syntax ist sie stark an C++ angelehnt, zusätzlich gibt es noch einige CORBA spezifische Erweiterungen, z.B. in Bezug auf die Definition von Datentypen. Um die in IDL beschriebenen Objekte in einer konkreten Programmiersprache implementieren zu können, wird ein IDL-Compiler benötigt, der eine Abbildung der Schnittstellenbeschreibung auf die jeweilige Implementierungssprache vornimmt. Von der OMG wurden bisher Sprachanbindungen für C, C++, Smalltalk, Ada95 und Cobol standardisiert. Ende Juni 1997 wurde auch die Abbildung für Java verabschiedet.

Schnittstellen des ORB

Auf Client-Seite hat der Benutzer eines Objektes zwei Möglichkeiten, einen Methodenaufruf zu initiieren:

- Eine Operation kann über die sogenannten IDL-Stubs angefordert werden. Dies sind Code-schablonen, die vom IDL Compiler für die verwendete Implementierungssprache erzeugt und zum Clientcode hinzugebunden werden.
- Alternativ können Operations-Anforderungen zur Laufzeit erzeugt werden. Informationen über vorhandene Schnittstellen und Objekte lassen sich aus dem Interface Repository abfragen und ermöglichen es, dynamisch einen Methodenaufruf zusammenzubauen und auszuführen. Für dynamische Operations-Aufrufe wird die Dynamische Aufrufschnittstelle (**D**ynamic **I**nvocation **I**nterface) benutzt.

Zusätzlich gibt es noch eine ORB-Schnittstelle, die einige Basisfunktionen, z.B. zur Erstellung dynamischer Aufrufe, zur Verfügung stellt.

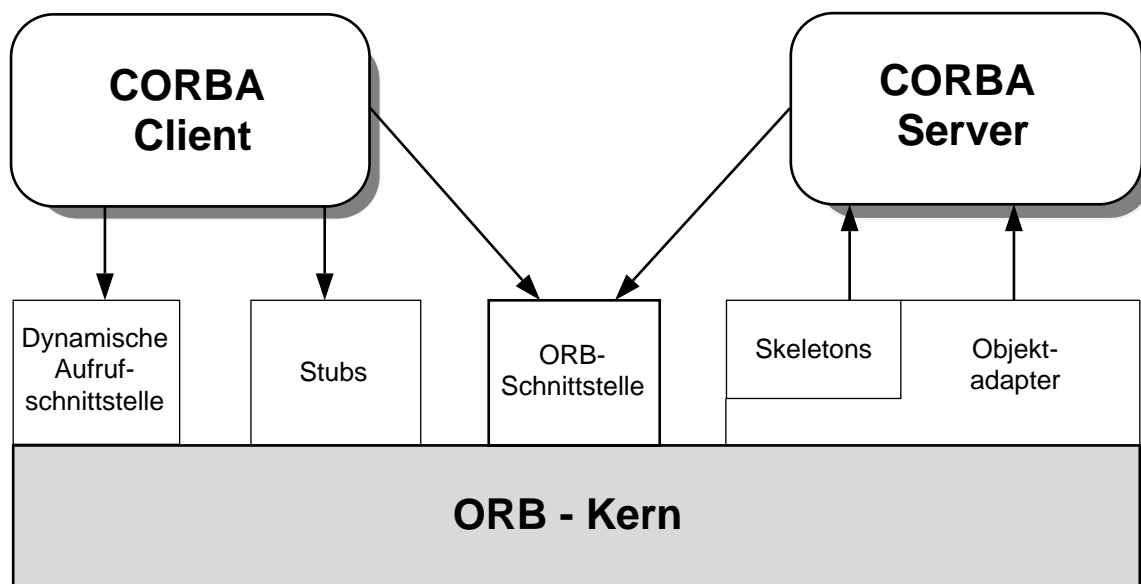


Abb. 2.1 CORBA Architekturschema

Auf Server-Seite leitet der ORB eintreffende Methodenaufrufe an einen für das Zielobjekt zuständigen Objekt-Adapter (OA) weiter. Dessen Aufgabe besteht darin, das Zielobjekt zu lokalisieren und die Übergabeparameter aufzulösen. Der Aufruf einer Operation erfolgt dann über die sogenannten IDL Skeletons, welche vom IDL Compiler generiert werden und die Schnittstelle zur eigentlichen Objekt-Implementierung darstellen.

Damit der ORB in der Lage ist, zur Laufzeit für jede Implementierung den korrekten Objekt-Adapter zu identifizieren, werden zur Aktivierung von Objektimplementierungen benötigte Information im Implementation Repository abgelegt.

Spezifikation weiterer Dienste

Über die Übertragung von Methodenaufrufen hinaus bietet die OMG-Referenzarchitektur

weitere Basisdienste, Schnittstellen und Komponenten an, die für den Programmierer beim Erstellen verteilter Anwendungen nützlich sein können.

Die *CORBA services* definieren eine Sammlung von Diensten, Mechanismen und Schnittstellen, die in verteilten objektorientierten Umgebungen häufig benötigt werden [OMG95b]. Dazu zählen Dienste zur Abbildung von Namen auf Objektreferenzen (Naming Service), Dienste zum dauerhaften Speichern von Objekten (Persistence Service), Dienste zur asynchronen Meldung von Ereignissen (Event Service), Dienste zur Durchführung von transaktionsgesteuerten Abläufen (Transaction Service) und andere mehr.

In den *CORBA facilities* spezifiziert die OMG Schnittstellen und Objekte, die allgemein nützliche Funktionen in Bereichen wie z.B. Benutzerschnittstellen offerieren [OMG95c].

Die *CORBADomains* decken ein spezielles Anwendungsgebiet ab, wie z.B. CAD oder Finanzbuchhaltung.

Mit den *Application Objects* werden Objekte und Schnittstellen definiert, die spezifisch für Anwendungen des Endbenutzers sind, wie z.B. die Ablaufsteuerung eines Fertigungsprozesses.

2.2 Java

Im folgenden wird kurz auf die Entstehung von Java eingegangen. Die Konzepte dieser Programmiersprache und ihre Rolle für das Internet werden beschrieben, es wird aber darauf verzichtet, tiefer auf Details der Programmierung einzugehen. Hier wird auf die inzwischen reichlich verfügbare Fachliteratur verwiesen [MSSt96] [Flan96]. Eine sehr ausführliche Liste nahezu aller in diversen Sprachen publizierter Java Bücher findet man unter [Piet97].

2.2.1 Historie

Die Geschichte von Java begann im Jahre 1990 in den Entwicklungslabors der Firma Sun Microsystems. Die Visionäre von Sun gingen davon aus, daß intelligente, elektronische Haushaltsgeräte in den kommenden Jahren eine wichtige Rolle spielen würden. Im Rahmen des Green Projektes sollte eine geeignete Programmiersprache entwickelt werden, die unabhängig von der jeweiligen Plattform arbeiten kann - ob im Telefon, der Waschmaschine oder PC. Als Name wurde *Oak* gewählt, dem Projekt war allerdings kein Erfolg beschieden, da Gespräche mit führenden Elektronikherstellern scheiterten.

Durch die zunehmende Popularisierung des Internets und speziell des World Wide Webs ergab sich nun aber ein ideales Einsatzgebiet für eine plattformunabhängige Programmiersprache. Das Konzept von *Oak* wurde wieder aufgegriffen, nur der Name wurde, da er sich nicht schützen ließ, in *Java* umbenannt, welches in den Vereinigten Staaten ein Synonym für Kaffee ist.

2.2.2 Überblick

Java ist eine objektorientierte Sprache, deren Konzepte aus verschiedenen anderen objektorientierten Programmiersprachen entnommen wurden und die stark dem weitverbreiteten C++ ähnelt. Im Unterschied zu C++ wurde Java allerdings von Anfang an objektorientiert konzipiert, weshalb es mit Ausnahme der elementaren Datentypen nur Objekte gibt. C Elemente wie *struct*, *union*, *enum* und *typedef* gibt es nicht, das Operator Overloading und die Mehrfachvererbung aus C++ wurden ebenfalls gestrichen. Ein weiterer Aspekt bei der Konzipierung von Java bestand darin, die Sprache möglichst einfach und übersichtlich zu gestalten, weshalb auf Zeiger verzichtet wurde. Dadurch konnte eine potentielle Fehlerquelle ausge-

geschlossen werden, die Programme sind übersichtlicher und leichter zu warten. Um die Speicherfreigabe braucht sich der Programmierer nicht zu kümmern, dies übernimmt die automatische Speicherverwaltung (Automatic Garbage Collection). Weitere wichtige Details der Sprache sind das Konzept zur Behandlung von Ausnahmen und die Integration von Multi-Threading.

Hauptbestandteile

Java besitzt drei wesentliche Komponenten:

- **Schnittstellen** (Interface): Schnittstellen definieren Methoden und Variablen ohne jedoch eine Implementierung vorzunehmen. Eine Mehrfachvererbung von Schnittstellen ist erlaubt. Java Schnittstellen sind den CORBA IDL Schnittstellen sehr ähnlich.
- **Klassen** (Class): Klassen implementieren entweder die in einer Schnittstelle definierten Methoden oder eigene Methoden. Für Klassen ist nur eine einfache Vererbung zulässig.
- **Objekte** (Object): Objekte sind Laufzeit-Instanzen von Klassen und immer mit einer virtuellen Java-Maschine verknüpft. Der Zustand eines Objekts kann durch Modifikation öffentlicher Variablen oder Methodenaufrufe verändert werden.

Die Zusammenhänge von Schnittstellen, Klassen und Objekten werden in Abb. 2.2 verdeutlicht.

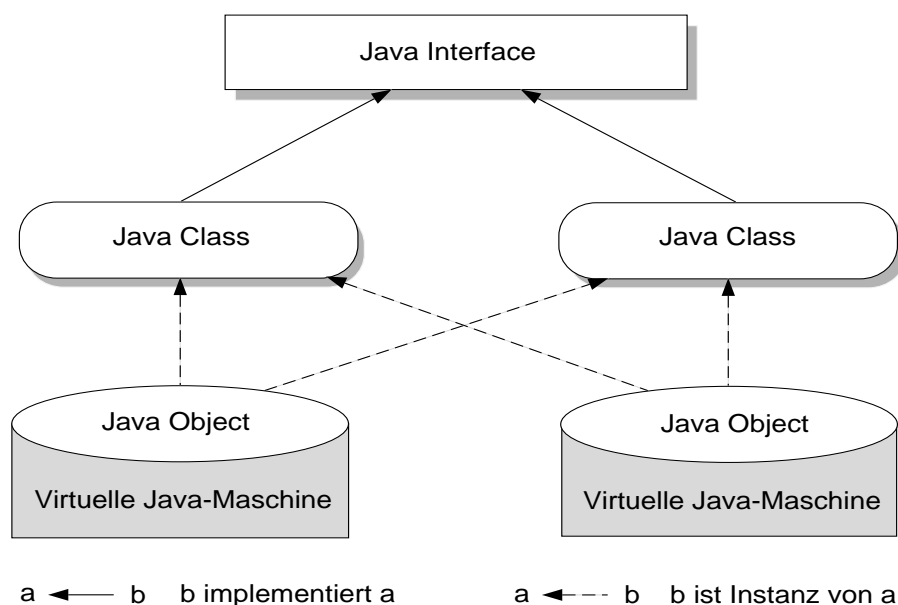


Abb. 2.2 Beziehungen zwischen Schnittstellen, Klassen und Objekten

Plattformunabhängigkeit

Java ist eine interpretierte Sprache und somit von der verwendeten Plattform unabhängig. Zur Ausführung wird lediglich eine virtuelle Java-Maschine benötigt, die den vom Compiler erzeugten maschinenneutralen Bytecode übersetzt und ausführt. Die Java Runtime ist maschinenspezifisch und muß auf jedes Betriebssystem portiert werden. Sun bietet das JDK (Java Development Kit) für Sparc und X86 Solaris, Windows NT/95 und MacOS an. Java Portierung-

gen für Windows 3.1, OS/2 und AIX wurden von IBM entwickelt, das auch an Ports für MVS und OS/400 arbeitet. Weitere Portierungen gibt es für Linux, IRIX, HP-UX und weitere UNIX Derivate.

Applikationen und Applets

Grundsätzlich gibt es zwei Anwendungsmöglichkeiten für Java. Zum einen kann man Java als gewöhnliche Programmiersprache wie z.B. C++ einsetzen, um eigenständige Programme, sogenannte Java-Applikationen, zu erstellen. Diese Programme werden zwar als eigenständig bezeichnet, zur Ausführung ist aber eine virtuelle Java-Maschine nötig. Zum anderen lassen sich Java Programme in Web-Seiten integrieren, man spricht dann von Java-Applets. Diese sind nur in einem Java-fähigen Web-Browser lauffähig. Eine Java-Unterstützung bieten derzeit der Netscape Navigator ab der Version 2.0, Microsoft Internet Explorer ab 3.0, Oracle Power Browser ab 1.5 und der HotJava Browser von Sun. Diese Browser verstehen einen speziellen HTML-Tag, den <APPLET>-Tag, der auf ein Java-Applet verweist, das vom Browser geladen und ausgeführt wird.

Sicherheit

Aus Sicherheitsgründen unterliegen Applets - im Gegensatz zu Java-Applikationen - einigen Einschränkungen. Ausführbare Inhalte, die in Form von Applets von einem fremden Rechner geladen werden, stellen für das lokale System eine potentielle Gefahr dar, da es für die Seriosität des heruntergeladenen Programmcodes keine Garantie gibt. Einem Applet ist es deshalb nicht erlaubt, auf Client-Seite Dateien zu schreiben oder zu lesen, da sonst das lokale Dateisystem ausspioniert oder Daten gelöscht werden könnten. Aus dem gleichen Grunde dürfen auf dem lokalen Rechner auch keine Programme gestartet werden. Auf Serverseite darf der Inhalt von Dateien gelesen werden, ein Schreibzugriff ist allerdings nicht erlaubt. Die Java-Browser erlauben Applets nur zu dem Rechner eine Verbindung aufzunehmen, von dem sie geladen wurden. Der Compiler fängt sprachliche Sicherheitslöcher wie Bereichsüberschreitungen ab, zusätzlich wird der Bytecode eines über das Netz geladenen Java Programmes noch einmal verifiziert. Der Verzicht auf Zeiger trägt wesentlich zur Sicherheit von Java bei, da eine Manipulation von Speicherstellen nicht mehr möglich ist. In frühen Versionen der Java-Browser wurden dennoch einige Sicherheitsmängel entdeckt, die inzwischen aber behoben sind. Eine vollständige Sicherheit kann sicher nicht garantiert werden, das Sicherheitskonzept ist aber umfassend und allgemein anerkannt.

2.2.3 Ausblick

Java erfreut sich großer Beliebtheit, bietet es doch völlig neue Möglichkeiten für Softwareerstellung, -vertrieb und -administration. Viele namhaften Hersteller haben es inzwischen von Sun lizenziert, selbst die Microsoft Corporation. In der Version 1.0.2 des JDK sind aber noch sehr viele Fehler anhalten, speziell im AWT (Abstract Windows Toolkit). Diese sollen in der Version 1.1 des JDKs beseitigt werden, das bereits in einer fehlerbereinigten Version 1.1.3 vorliegt. Ein Überblick über die Neuerungen im JDK 1.1 ist erhältlich bei JavaSoft [Jav97b]. Da Java immer noch sehr neu ist, gibt es bisher wenig richtige Anwendungen. Für das erste Halbjahr 1997 haben aber die Firmen Corel und Star Division Java-Implementierungen ihrer Officepakete angekündigt, Demoversionen können bereits ausprobiert werden. Viele andere Firmen arbeiten ebenfalls an Java-Versionen ihrer Produkte. Eine weitere Popularisierung von Java ist durch die Integration in Betriebssysteme (JavaOS), die Geschwindigkeitssteigerung durch JIT (Just In Time) Compiler und die Verfügbarkeit von Netzcomputern (NC) zu erwarten. Durch das große Interesse an Java zeichnet sich auch eine Verwendung im Electronic

Consumer Markt ab, der ursprünglich ja auch als Einsatzort von Java gedacht war.

2.3 CORBA & Java

Eine Kombination von CORBA und Java bietet sich an, da sich die Stärken der beiden Technologien ideal ergänzen und die jeweiligen Schwächen beheben. Java ORBs sind komplett in Java implementiert und profitieren von der Portabilität dieser Programmiersprache, die auch eine Verwendung in einem Java-fähigen Web-Browser ermöglicht. Client-seitige Java Applets können bei Bedarf die Klassen eines Java ORBs nachladen und dadurch auf CORBA Server zugreifen. Das client-seitige Laufzeitsystem eines Java ORBs wird auch als ORBlet bezeichnet.

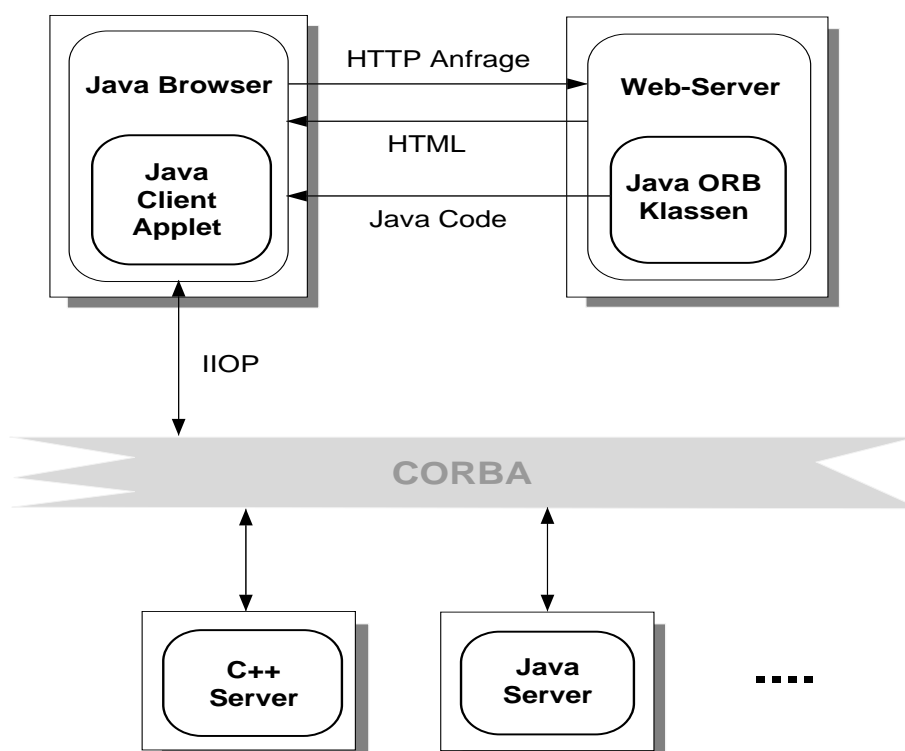


Abb. 2.3 Schematik der Integration von CORBA und Java

Für den CORBA Programmierer bietet eine Sprachanbindung für Java eine Reihe von Vorteilen. Durch die Portabilität von Java und der großen Anzahl unterstützter Plattformen werden speziell client-seitige Komponenten sehr universell verwendbar. Clients können als Applets programmiert und im Internet oder firmeninternen Intranet eingesetzt werden. In Java programmierte Server bieten den Vorteil, daß aufwendige Portierungen entfallen und Updates durch Nachladen neuer Klassen möglich sind, ohne daß eine Neuübersetzung notwendig ist. Auch Applets lassen sich als Server einsetzen. Sie unterliegen allerdings den durch die Java-Browser auferlegten Sicherheitsbestimmungen, die keinen Zugriff auf lokale Ressourcen erlauben. Ist aber zumindest möglich, Callback Objekte in Applets zu integrieren.

Durch sein objektorientiertes Sprachkonzept ist die Integration von Java und CORBA relativ problemlos. Java verfügt mit der Behandlung von Ausnahmen und dem Multi-Threading über Konzepte, die zur Erstellung verteilter Anwendungen sehr hilfreich sind. Wie bereits erwähnt

ist die Sprachanbindung (IDL-Java-Mapping) erst seit kurzem standardisiert. Die meisten Hersteller verwenden deshalb noch ihre eigenen Mappings und werden ihre Produkte in nächster Zeit soweit anpassen, daß sie mit dem Standard konform sind.

Von der Kombination mit CORBA profitiert Java dadurch, daß nun ein Konzept zur Verfügung gestellt wird, um auf entfernte und verteilte Objekte ortstransparent zugreifen zu können. Die Verbindungsaufnahme eines Applets ist nicht mehr nur auf den Server beschränkt, von dem es geladen wurde, da dieser weitere Server aufrufen kann. Durch die Trennung von Schnittstelle und Implementierung in CORBA lassen sich auf Server-Seite verschiedene Programmiersprachen einsetzen, zeitkritische Komponenten können z.B. in C++ implementiert werden. Dem Programmierer bieten sich auf Server-Seite vielfältige Möglichkeiten für den Einsatz von unterschiedlichen Programmiersprachen, Hardwareplattformen und Betriebssystemen, deren Heterogenität von CORBA vollständig verdeckt wird. Für den Programmierer der Client-Seite ist dies alles transparent, d.h. er braucht über sämtliche Implementierungsdetails nichts zu wissen. Der Aufruf einer Methode eines entfernten Objektes erfordert keinerlei Kenntnis über den Aufenthaltsort dieses Objekts, da der entsprechende Server vom ORB lokalisiert und gegebenenfalls gestartet wird. Die Angabe von Rechnernamen und Ports ist nicht nötig.

Java ist bestens dazu geeignet, um interaktive WWW-Anwendungen zu erstellen. Auf Server-Seite können diese Anwendungen mit Hilfe von CORBA skalierbar gemacht und auf mehrere Rechner verteilt werden. Durch die Nutzung von client-seitigen Ressourcen und die Möglichkeit, Dienste auf verschiedenen Rechnern parallel auszuführen, lassen sich leistungsfähige Anwendungen mit hoher Verfügbarkeit erstellen. Bereits bestehende Anwendungen können weiterverwendet und mittels eines Objekt-Adapters an einen ORB angekoppelt werden. Dadurch ergeben sich enorme Kostenvorteile, da Anwendungen für den Einsatz im Internet oder Intranet nicht neu geschrieben werden müssen.

Zusammengefaßt bietet die Integration von CORBA und Java folgende Vorteile:

- Erstellung von interaktiven, skalierbaren Internet/Intranet-Anwendungen
- Weiterverwendung von bestehenden Anwendungen
- Integration von verschiedenen Hardwareplattformen, Betriebssystemen und Programmiersprachen
- Verringerter Aufwand in der Administration

Weiterführende Literatur zum Thema CORBA und Java findet man unter [VoDu97] und [OrHa97]. Diverse Artikel wurden in verschiedenen Fachzeitschriften veröffentlicht und sind auch im WWW abrufbar [EFVo96, Vog96a, Vog96b].

2.4 Vergleich von CORBA mit RMI

Das Fehlen eines konkreten Sprachkonzeptes für Client/Server-Anwendungen wurde den Entwicklern von Java bereits frühzeitig bewußt, weshalb unter dem Namen „Remote Objects for Java“ zwei Projekte ins Leben gerufen wurden, die dem Programmierer die Erstellung von verteilten Anwendungen mit Java erleichtern sollen.

Java IDL ermöglicht es, Java Clients mit Netzwerk-Servern transparent zu verbinden, wobei der von der OMG spezifizierte Industriestandard IDL (Interface Definition Language) zur

Beschreibung von Schnittstellen verwendet wird. Eine ausführliche Beschreibung dieses Produkts erfolgt in Kapitel 3.8.

Das zweite Projekt trägt den Namen Java RMI (Remote Method Invocation) und ist Bestandteil des JDK 1.1 [Hran97]. Das Konzept von RMI beruht auf dem Umstand, daß für reine Java Anwendungen Schnittstellen nicht unbedingt in IDL definiert werden müssen. Schließlich verfügt Java bereits über ein eigenes `interface`-Konzept. Dies macht sich RMI zu Nutze und realisiert eine verteilte Objektarchitektur für eine homogene „Java-Welt“, d.h. Clients und Server müssen vollständig in Java geschrieben werden.

Die Konzepte von RMI und CORBA sind sich sehr ähnlich. Der Aufruf von entfernten Methoden erfolgt bei beiden Technologien über ein Proxy-Objekt. Intern nimmt das Proxy-Objekt die Aufrufe entgegen und leitet sie an das RMI-Laufzeitsystem bzw. den ORB weiter. Stellvertreterobjekte, die sogenannten Stubs und Skeletons, werden von einem Compiler erzeugt. Bei RMI wird dem Compiler allerdings nicht wie bei CORBA die Schnittstellebeschreibung (IDL), sondern die Implementierungsklasse der als Java `interface` definierten Schnittstelle übergeben.

Beide Technologien profitieren von den besonderen Eigenschaften von Java, z.B. der automatischen Speicherbereinigung und der Möglichkeit, Klassen bei Bedarf nachladen zu können.

Die RMI-Technologie bietet gegenüber den Java ORBs folgende Vorteile:

- RMI kann nahtlos in das Java-Sprachsystem integriert werden. Bei CORBA ist dies wegen des Anspruchs auf Sprachunabhängigkeit nicht möglich, d.h. hier wird eine Abbildung von IDL nach Java benötigt.
- Eine zusätzliche Schnittstellen-Beschreibungs-Sprache ist nicht notwendig, da Schnittstellen als Java `interface` definiert werden.
- RMI erlaubt es Parameter nicht nur per Referenz, sondern auch per Wert zu übergeben.

Dem gegenüber sprechen für eine Lösung mit CORBA und Java folgende Argumente:

- Durch die Unterstützung von mehreren Programmiersprachen wird es mit CORBA möglich, bestehende (nicht in Java geschriebene) Anwendungen weiterzuverwenden. Die RMI Architektur unterstützt ausschließlich Java, was die Anbindung von nicht in Java programmierten Applikationen sehr erschwert.
- CORBA Objekte sind völlig ortstransparent, d.h. über den Aufenthaltsort eines Servers muß der Client nichts wissen. Bei RMI muß dem Client explizit mitgeteilt werden, auf welchem Rechner sich sein Server-Objekt befindet.
- Im Gegensatz zu RMI können Server bei Bedarf gestartet werden.
- Die Interoperabilität von CORBA Objekten, die von ORBs unterschiedlicher Hersteller verwaltet werden, ist durch das IIOP (Internet Inter-ORB Protokoll) gewährleistet (siehe hierzu Kapitel 4).
- CORBA bietet mit den CORBAservices, Common Facilities und Domain Objects weitreichende Funktionen und Hilfsmittel an, die für den Programmierer sehr nützlich sein können.

Zum gegenwärtigen Zeitpunkt müssen sich die Entwickler von verteilten Java-Anwendungen für eine von beiden Technologien entscheiden. Die beiden Konzepte lassen sich noch nicht miteinander verbinden. Der Großteil der Argumente spricht für die CORBA Lösung, die sich spe-

ziell für komplexere Architekturen anbietet. RMI eignet sich dagegen vor allem für reine Java Anwendungen kleinerer bis mittlerer Größe [VoDu97]. Benchmark-Test haben gezeigt, daß Java ORBs deutlich schneller sind als RMI [OrHa97]. Dies zeigt sich besonders deutlich, wenn die Umwandlung mehrerer Übergabeparamter notwendig ist.

In Zukunft werden beide Welten mehr und mehr zusammenwachsen. JavaSoft will zum einen Java IDL als Kern-Bestandteil in die Java Plattform aufnehmen. Zum anderen hat JavaSoft angekündigt, die Entwicklung von RMI weiter fortzusetzen und darin die Unterstützung für CORBA IIOP zu integrieren. Mit Hilfe des IIOP wird es möglich, von RMI Anwendungen aus auf CORBA Dienste zuzugreifen. Umgekehrt sind aber auch Bestrebungen im Gange, CORBA so zu erweitern, daß ein Zusammenspiel mit RMI möglich ist [Jav97c]. Durch die Zusammenarbeit mit der OMG sollen die Vorzüge von RMI, wie z.B. die Übergabe von Paramtern per Wert, auch in die CORBA Architektur integriert werden.

3 Produktübersicht

Die Anzahl verfügbarer CORBA/Java Produkte steigt seit der Auslieferung des ersten Java Development Kits ständig an. Von allen führenden ORB Herstellern gibt es inzwischen entsprechende Produkte oder liegen Produktankündigungen vor. Zudem gibt es auch eine Reihe von Forschungsvorhaben und nicht kommerziellen Produkten.

Im folgenden werden alle zum gegenwärtigen Zeitpunkt verfügbaren Produkte untersucht, die zur Entwicklung verteilter Client/Server-Anwendungen mittels CORBA und Java verwendet werden können. Dabei wird auf den jeweiligen Entwicklungsstand, die Verfügbarkeit und die vorhandene Funktionalität eingegangen. Soweit möglich wird eine Bewertung vorgenommen.

3.1 VisiBroker for Java (Visigenic Software Inc.)

VisiBroker for Java ist ein Object Request Broker (ORB), der mit dem CORBA 2.0 Standard konform und komplett in Java geschrieben ist. Das Produkt wurde unter dem Namen Black Widow von der Firma PostModern Computing entwickelt und basiert auf deren CORBA Implementierung ORBeline, ein CORBA 2.0 konformer ORB mit Sprachanbindung für C++. Nach dem Zusammenschluß von PostModern Computing mit Visigenic Software wurden Black Widow und ORBeline in VisiBroker for Java bzw. VisiBroker for C++ umbenannt.

VisiBroker for Java Applets können in allen Java-fähigen Web-Browsern ausgeführt werden. Hierdurch kann das Applet innerhalb der Webseite als Client auftreten und aus dem Browser heraus alle CORBA 2.0 Services und Objekte benutzen.

3.1.1 Informationen

Die verfügbaren Informationen sind im Vergleich zu anderen Produkten relativ umfangreich und allesamt im Internet verfügbar. Dabei gibt es folgende Quellen:

- WWW-Server von Visigenic Software [VBrJ96]
Die Dokumentation zu VisiBroker for Java ist Online verfügbar und besteht aus einem Programmers Guide sowie einem Reference Guide. Beide Dokumente sind auch im FrameMaker und PDF Format erhältlich und können vom FTP-Server von Visigenic geladen werden. Dort gibt es auch Dokumentation zu VisiBroker for C++, die auch für VisiBroker for Java eine gewisse Relevanz hat.
- E-Mail an info@visigenic.com schicken
Im allgemeinen erhält man schnell und kompetent Antwort auf eine Frage.
- Zu Black Widow gab es eine Mailingliste, nach der Übernahmen von PostModern Computing durch Visigenic wurde diese aber eingestellt. Laut Informationen von Visigenic gibt es Pläne, die Mailingliste wieder einzurichten, was bis zum gegenwärtigen Zeitpunkt aber noch nicht geschehen ist.

3.1.2 Verfügbarkeit

VisiBroker for Java ist zum gegenwärtigen Zeitpunkt in der Version 2.5 erhältlich und unterstützt neben dem JDK 1.0 auch das neue JDK 1.1. Da der ORB komplett in Java implementiert ist, läuft er auf allen Plattformen, für die es eine virtuelle Java-Maschine gibt. Der sogenannte Smart Agent ist allerdings betriebssystemabhängig. Die Version 2.5 ist gegenwärtig für Sola-

ris und Windows 95/NT verfügbar. Die alte 1.2 Version gibt es für Solaris, Windows 95/NT, AIX, HP-UX und IRIX und unterstützt nur das JDK 1.0. Visigenic bietet die Möglichkeit, VisiBroker for Java mit einem Online-Formular im Internet zu bestellen und per FTP zu laden. Es gibt eine kostenlose Demoversion, die eine Gültigkeit von sechzig Tagen hat und die volle Funktionalität bietet. Der Erwerb einer Entwicklungslizenz kostet für Solaris 2.995 Dollar und Windows 95/NT 1.995 Dollar. Forschungseinrichtungen und Universitäten erhalten einen Preisnachlaß von vierzig Prozent des normalen Preises.

3.1.3 Funktionalität

VisiBroker for Java besteht aus zwei Hauptkomponenten:

- Die Entwicklungskomponente besteht aus dem Compiler `idl2java`, der aus einer CORBA IDL Java Code sowohl für den Client, als auch den Server erzeugt. Mit Hilfe dieser Stubs kann ein Java Client verteilte Server Objekte aufrufen, wie wenn diese lokal verfügbar wären. Das seit Ende Juni 1997 standardisierte Java Mapping wird ab der Version 2.5 unterstützt.
- Die Laufzeitkomponente sorgt für die Kommunikation zwischen Applets und verteilten Objekten. Sie ist in die Version 4.0 des Netscape Navigators und andere Netscape Server Produkte integriert. Die Client-Laufzeitkomponente ist in Java geschrieben und kann in jedem Java-fähigen Browser ablaufen.

Architektur

VisiBroker for Java verfügt über eine agenten-basierte Architektur. Mit Hilfe eines sogenannter Smart Agents (`osagent`) können Clients die Objekte eines Servers aufrufen, ohne dessen genauen Aufenthaltsort wissen zu müssen. Wird ein Server gestartet, so unterrichtet dieser den Smart Agent über seinen Aufenthaltsort, wozu die Methode `CORBA.BOA.obj_is_ready` verwendet wird. Ein Client kann dann von dem Smart Agent mit Hilfe der `bind` Methode erfahren, wo sich das Serverobjekt befindet. In einem LAN oder WAN können mehrere Smart Agents laufen, wodurch sich eine hohe Verfügbarkeit und Ausfallsicherheit ergibt. Die Agenten lokalisieren sich gegenseitig und können gegebenenfalls die Aufgaben des anderen übernehmen. Dies ist vor allem für die Lastenverteilung auf einem Web-Server von Vorteil. Da Java bislang aber keinen UDP Broadcast unterstützt, ist diese Funktion bisher nur für C++ Prozesse verfügbar und für Java Anwendungen nicht nutzbar.

Kommunikation

Im Gegensatz zu anderen Herstellern verwendet VisiBroker for Java nur ein Kommunikationsprotokoll, das Internet Inter-ORB Protokoll (IIOP). Das IIOP wird im CORBA 2.0 Standard der OMG spezifiziert und ermöglicht die Kommunikation zwischen ORB Implementierungen verschiedener Hersteller. Dadurch ergibt sich für den Programmierer der Vorteil, daß er sich um das verwendete Protokoll und nötige Anpassungen in der Konfiguration keine Gedanken machen muß. Die Kommunikation zwischen Clients und Servern erfolgt grundsätzlich über IIOP, es wird keine HTTP-Dämon benötigt, somit kann der Web-Server seiner eigentlichen Aufgabe, der Bereitsstellung von HTML-Seiten, nachgehen. Nachdem ein Client einen Server lokalisiert hat, wird eine dedizierte TCP Verbindung aufgebaut, der Smart Agent wird danach nicht mehr benötigt. Es gibt bislang aber keine Möglichkeit, einen Server bei Bedarf zu aktivieren, diese müssen von Hand gestartet werden. Ein Objekt-Aktivierungs-Dämon (OAD) wird erst in der Version 3.0 zur Verfügung stehen.

Bei Verwendung von Java Applets als Client wird der sogenannte *Gatekeeper* benötigt. Hierbei handelt es sich um eine Java Applikation, die für die IIOP-Kommunikation zwischen Client Applets und Servern verantwortlich ist (Secure Bridge). Mit Hilfe des Gatekeepers kann die Sicherheitsrestriktion umgangen werden, die es einem Applet nur erlaubt, zu dem Server Kontakt aufzunehmen, von dem es geladen wurde. Dadurch wird die von CORBA geforderte Ortstransparenz möglich, da der Gatekeeper Aufrufe entsprechend weiterleiten kann. Die Kommunikation kann dabei in beiden Richtungen erfolgen, d.h. der Server kann auch den Client im Browser kontaktieren (Remote Callback).

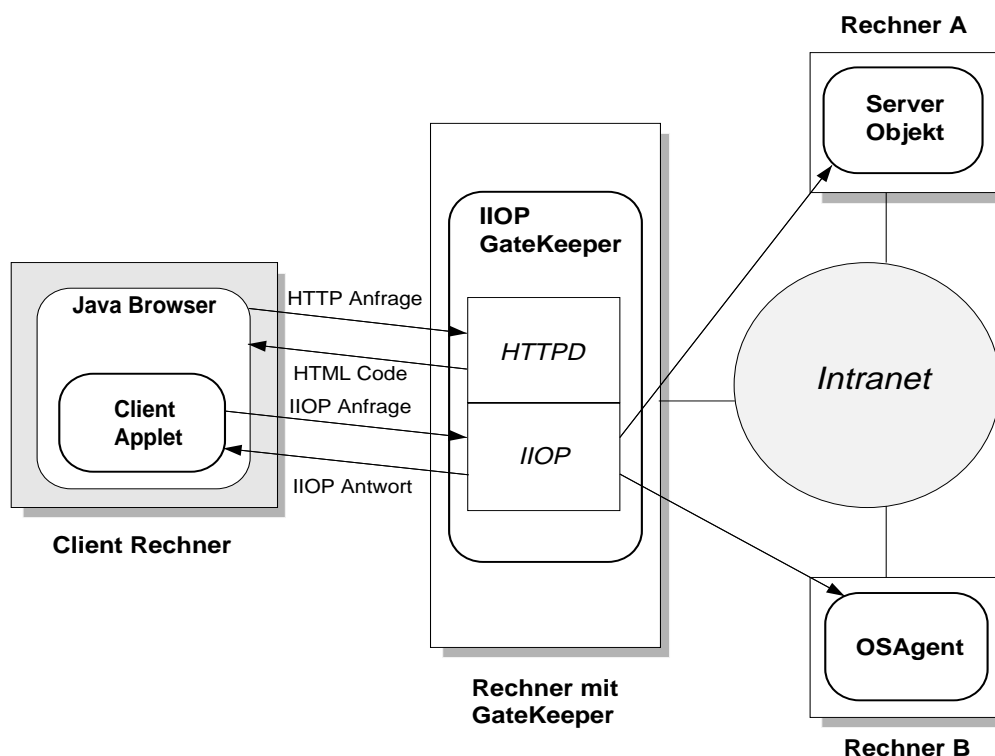


Abb. 3.1 VisiBroker for Java Architektur

Die Sicherheitsfunktionen des Gatekeepers sollen eine Kontrolle der exportierten Objekte und die Registrierung aller erlaubten und unerlaubten Zugriffsversuche ermöglichen. Genauere Informationen lassen sich der Dokumentation aber nicht entnehmen, lediglich das Debugging des IIOP Streams wird beschrieben, ohne jedoch auf Details einzugehen. Wie sicher der Gatekeeper wirklich ist, läßt sich nicht sagen. Netscape hat ihn wegen entsprechender Bedenken wieder aus dem Navigator 4.0 und dem Enterprise Server 3.0 entfernt.

Der Gatekeeper kann auch als HTTP-Dämon eingesetzt werden. Da es sich aber um eine Java Applikation handelt, ist dies aus Geschwindigkeitsgründen nicht ratsam. In der Praxis ergaben sich zudem Probleme beim Lesen von server-seitigen Dateien aus Applets. Leider kann die zusätzliche Web-Server-Funktionalität des Gatekeepers nicht abgestellt werden.

Ein gängiges Problem von Java ORBs besteht in der Kooperation mit Standard Firewalls. Firewalls schützen Intranets vor unberechtigten Zugriffen aus dem Internet, indem sie den Zugang zum Intranet nur über den Firewall Host erlauben und hereinkommende Daten filtern. Im allgemeinen werden Firewalls so konfiguriert, daß Web-bezogene Daten, wie HTTP-Anfragen oder E-Mails, die Firewall passieren dürfen. Andere Daten, wie z.B. IIOP-Anfragen, werden dagegen nicht durchgelassen.

Damit Browser dennoch auf Objekte im Internet zugreifen können, die durch eine Firewall geschützt sind, verfügt der Gatekeeper über ein zusätzliches Protokoll, das die Technik des sogenannten *HTTP Tunneling* unterstützt. Fordert ein Client ein Objekt an und kommt keine IIOp Kommunikation zustande, wird die IIOp Anfrage automatisch in einer HTTP Anforderung gekapselt. Diese wird von der Firewall nicht beanstandet und durchgelassen. Der Gatekeeper erhält die HTTP Anfrage und erkennt, daß es sich um eine gekapselte IIOp Anfrage handelt. Die IIOp Anfrage wird anschließend vom Gatekeeper extrahiert und zur Kommunikation mit dem Server verwendet. Das Konzept des HTTP Tunneling wird in Abb. 3.2 verdeutlicht.

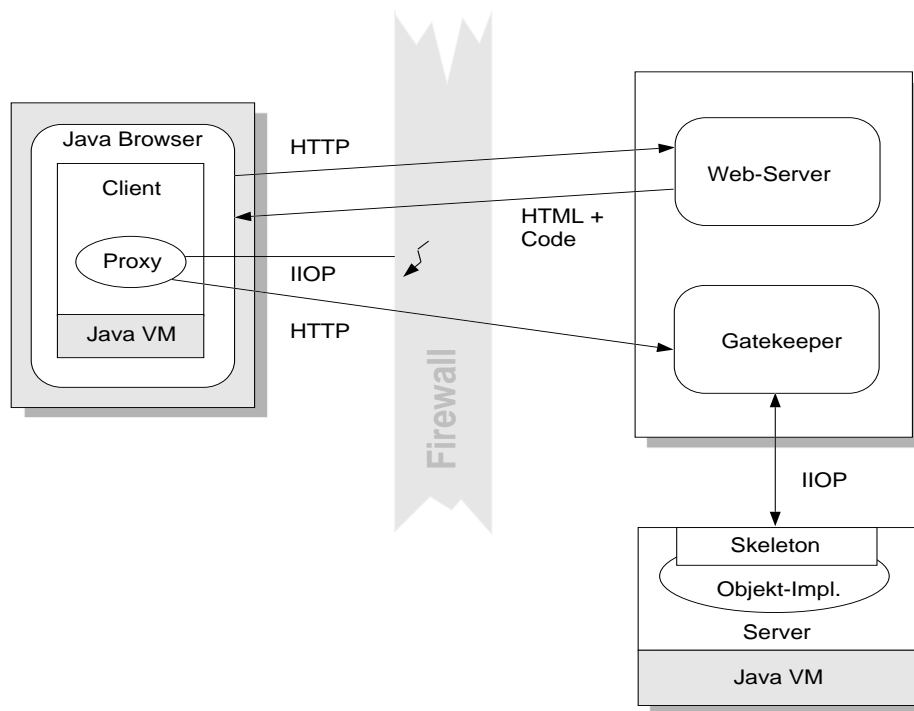


Abb. 3.2 HTTP Tunneling

Eine Einschränkung des Gatekeepers besteht darin, daß bei Verwendung des HTTP Tunneling keine Callbacks und Oneway Operationen unterstützt werden.

3.1.4 Besonderheiten

Unter dem Namen *Caffeine* hat Visigenic in Zusammenarbeit mit der Firma Netscape eine reine Java Lösung zur Entwicklung von CORBA IIOp Anwendungen entwickelt. Caffeine setzt auf VisiBroker for Java auf, bietet aber eine RMI-ähnliche Entwicklungsumgebung. Die Beschreibung von Schnittstellen und deren Implementierung erfolgt wie bei RMI in Java. Caffeine besteht aus folgenden Komponenten:

- **Java2IIOp**
Dieser Code Generator erstellt aus einer Java-Schnittstelle Stubs und Skeletons, wie der `idl2java` Compiler aus einer IDL.
- **Java2idl**
Optional ist es mit diesem Compiler möglich, für Java Schnittstellen entsprechende IDL Schnittstellen zu erzeugen. Caffeine Clients und Server sind aber auch ohne CORBA IDL benutzbar.

- **Web Naming**
Hierbei handelt es sich um einen Namensdienst, der auf URLs basiert. Objektreferenzen werden mit URLs verknüpft und können über einen Web-Server leicht zur Verfügung gestellt werden.

Der Vorteil von Caffeine gegenüber RMI besteht darin, daß der darunterliegende ORB eben VisiBroker for Java ist. Dadurch können alle Objekte aus der CORBA-Welt angesprochen werden, auch solche, die nicht in Java programmiert sind. Mit Hilfe des Web Naming Dienstes können Objektreferenzen ermittelt werden, um ohne OSAgent und Gatekeeper entfernte Objekte zu lokalisieren.

Als erste Firma bietet Visigenic Implementierungen der CORBA Naming und Event Services in Java an. Diese sind allerdings nicht Bestandteil von VisiBroker for Java und müssen separat erworben werden.

3.1.5 Gesamteindruck

VisiBroker for Java war der erste verfügbare Java ORB [Chal96] und von Anfang an konform zum CORBA 2.0 Standard. Der gebotene Funktionsumfang ist recht groß, wobei vor allem die flexible Handhabung hervorzuheben ist. Durch den Gatekeeper wird die Ortstransparenz von CORBA Objekten auch für Applets gewährleistet. Das Zusammenspiel von CORBA und Java ist recht gelungen. Das Erstellen von Programmen gestaltet sich relativ unkompliziert, wobei der Sourcecode recht übersichtlich bleibt. Als sehr störend erweist sich, daß das Laden der für den ORB benötigten Klassen sehr lange dauert. Deutlich negativ bemerkbar machte sich auch, daß lange Zeit keine schriftliche Dokumentation erhältlich war. Mit der Version 2.5 wird nun endlich eine umfassende Dokumentation ausgeliefert. Leider werden immer noch sehr wenig Beispiele mitgeliefert, hier haben andere Hersteller wesentlich mehr zu bieten. Dennoch kann man VisiBroker for Java wohl als das Produkt bezeichnen, das den ausgereiftesten Eindruck macht und das in der Entwicklung am weitesten vorangeschritten ist. Die Version 3.0 ist für Ende Juli 1997 angekündigt.

3.2 OrbixWeb (IONA Technologies Ltd.)

IONA Technologies ist einer der führenden Anbieter auf dem Gebiet der CORBA Technologie mit Sitz in Dublin (Irland). Bereits seit Juni 1993 bietet IONA die CORBA Implementierung Orbix mit der Sprachanbindung für C++ an. Dieses Produkt zählt zu den am weitesten verbreiteten ORBs mit Portierungen für Windows 3.1x, Windows NT/95, OS/2, Mac System 7.5, sowie für diverse UNIX Systeme. Laut den Angaben von IONA wird Orbix zur Zeit von mehreren tausend Entwicklern eingesetzt.

3.2.1 Informationen

Die Informationen zu OrbixWeb waren anfangs sehr spärlich, sind inzwischen aber recht umfangreich [Ion96a]. Auf der Homepage von IONA sind eine Produktübersicht, ein Whitepaper, ein *Internet Cookbook* und ein Online-Bestellformular verfügbar. Für häufig gestellte Fragen gibt es einen eigenen Bereich, in dem mögliche Lösungen und Erklärungen für diverse Probleme aufgeführt sind. Zudem gibt es einen sehr umfangreichen Programmmer's Guide und eine API Dokumentation als HTML-Seiten. Diese können auch lokal installiert und gelesen werden. Eine Postscript Version des Programmmer's Guide ist bislang nur für die Version 1.0 erhältlich und kann über FTP bezogen werden. Die gegenwärtig neuste Version 2.01 wird mit zwei umfangreichen Handbüchern ausgeliefert.

3.2.2 Verfügbarkeit

OrbixWeb ist gegenwärtig in der Version 2.01 verfügbar und kostet \$ 799. Es gibt wie bei VisiBroker for Java auch eine kostenlose Demoversion, deren Gültigkeit auf sechzig Tage beschränkt ist. Zur Bestellung der Demoversion ist auf der Homepage von IONA ein Bestellformular auszufüllen, daraufhin erhält man umgehend Zugang zu einem FTP-Server mit einem entsprechenden Passwort. An Plattformen werden bisher Solaris 2.x, 32-bit Windows und HP-UX 10.x unterstützt, die Portierung auf weitere UNIX-Plattformen ist angekündigt. Für die aktuelle Version werden Patches auf dem öffentlich zugänglichen FTP Server bereitgestellt.

3.2.3 Funktionalität

Durch die Implementierung von OrbixWeb in Java kann Orbix Client-Funktionalität als Java-Bibliothek in den aufrufenden Client geladen werden. Java Applets können so auf dem Zielrechner alle CORBA 2.0 Dienste nutzen. Ab der Version 2.0 bietet OrbixWeb sowohl eine client- als auch server-seitige Funktionalität, d.h. zur Entwicklung von server-seitigen Komponenten wird kein installiertes Orbix benötigt. Man kann sowohl Clients als auch Server in Java implementieren, die Benutzung von Orbix C++ Servern ist aber ebenso problemlos möglich. Das JDK 1.1 wird ab der Version 2.01 unterstützt.

Architektur

Wie bei VisiBroker for Java gibt es einen IDL Compiler, der aus einer IDL-Schnittstelle entsprechende Java Stubs und Skeletons erzeugt. Unterstützt werden alle IDL Typen (auch ANY) und TypeCodes. IONA verwendet ein eigenes Java-IDL Mapping, hat aber bereits angekündigt, das OMG Standard Mapping in der nächsten Version zu unterstützen.

Die OrbixWeb Laufzeitumgebung ist komplett in Java geschrieben. Die Funktionalität verteilt sich dabei auf mehrere Klassen, die bei Bedarf dynamisch nachgeladen werden können. Zur Kommunikation zwischen Clients und Servern unterstützt OrbixWeb das IIOP und das Orbix Protokoll, als Standard wird im Gegensatz zu früheren Versionen das IIOP verwendet. Die Architektur von OrbixWeb ist in Abb. 3.3 dargestellt.

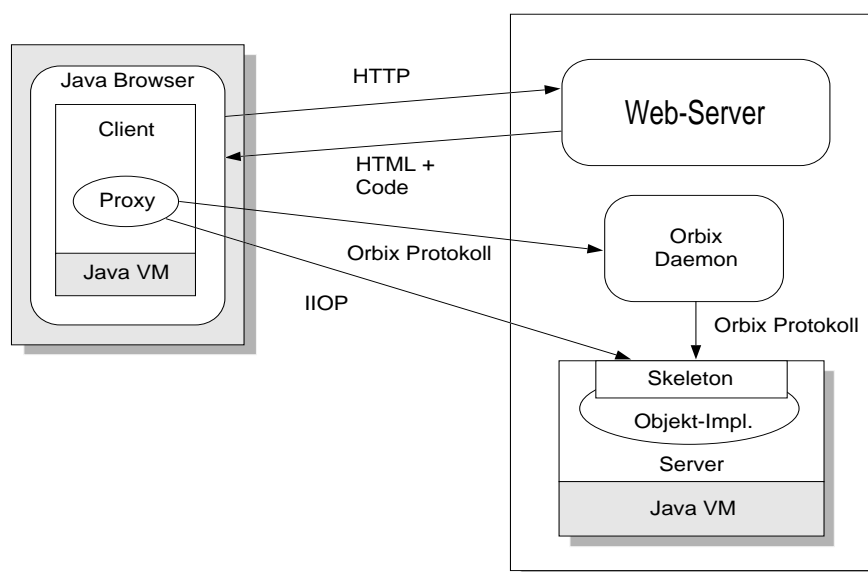


Abb. 3.3 OrbixWeb Architektur

Bei Verwendung des IIOP werden Server Objekte durch eine IOR (Interoperable Object Reference) identifiziert, d.h. ein Server muß von sich selbst eine IOR ausgeben, die ein Client liest und zur Herstellung einer Verbindung mit dem Server benutzt. Am einfachsten kann man diese IOR vom Server in eine Datei schreiben lassen. Komfortabler ist dagegen der Einsatz von OrbixNames als Repository für IORs. OrbixNames ist ein CORBA konformer Naming Service, bei dem Server ihre IORs registrieren können. Clients wiederum können diesen Dienst nutzen, um IORs abzufragen.

Optional wird auch das Orbix Protokoll unterstützt, das es erlaubt, Server auf Anforderung von Clients automatisch zu starten. Dazu wird auf dem Server-Rechner ein Orbix-Dämon (orbixd) benötigt, der Details zu den Servern in einem Implementation Repository verwaltet. Clients können mittels der `_bind()` Methode Serverobjekte anfordern. Ist ein angeforderter Server noch nicht instanziiert, wird er vom Orbix-Dämon lokalisiert und gestartet. Der Orbix-Dämon wurde so modifiziert, daß er auch Java Server ohne Umweg über Startscripts starten kann.

Mit OrbixWeb werden viele Beispiele mitgeliefert, mit denen die gebotenen Möglichkeiten demonstriert werden. Auch für die im folgenden beschriebenen Besonderheiten von Orbix-Web sind einfache Beispiele vorhanden.

3.2.4 Besonderheiten

OrbixWeb bietet eine Reihe von zusätzlichen Features, die für den Programmierer nützlich sein können, in der CORBA Spezifikation aber nicht festgelegt wurden [Merk97].

- **Filters**
Mit Hilfe von Filtern kann der Programmierer an bestimmten Stellen im Programm zusätzliche Anweisungen ausführen lassen, um z.B. Debuginformationen auszugeben oder Verschlüsselungen vorzunehmen.
- **SmartProxies**
Der Zugriff auf entfernte Objekte erfolgt über Proxy- oder Stellvertreterobjekte, die vom IDL Compiler automatisch erzeugt werden. In bestimmten Fällen kann es sinnvoll sein, die Implementierung der Proxy-Klasse selbst vorzunehmen. Diese SmartProxies können speziellen Bedürfnissen angepasst werden, z.B. um auf Client-Seite einen Cache für häufig benötigte Werte anzulegen.
- **Loaders**
Normale CORBA-Objekte sind stets transient, d.h. sie existieren nur während der Lebenszeit ihres Servers. Mit Hilfe der sogenannten Loaders können CORBA-Objekte mit Persistenz versehen werden.
- **Locators**
Um bei einem entfernten Methodenaufruf den entsprechenden Server automatisch zu finden, wird ein Locator verwendet. Der Default Locator wird innerhalb des Orbix-Dämons zur Verfügung gestellt. Man kann aber auch einen eigenen Locator implementieren.

Für die Probleme der Ortstransparenz und der Zusammenarbeit mit Firewalls bietet IONA eine Lösung namens *Wonderwall* an. Ähnlich wie beim Gatekeeper von VisiBroker for Java gibt es einen Dämon-Prozess, der Methodenaufrufe an Hand der Informationen aus einer interoperablen Objektreferenz (IOR) an den entsprechenden Server weiterleitet. Wonderwall benutzt keine existierende Firewall, sondern stellt selbst eine Firewall dar, die auf dem Firewall Host

installiert werden muß und an einem wohldefinierten Port arbeitet. Das Konzept von Wonderwall wird in Abb. 3.4 verdeutlicht.

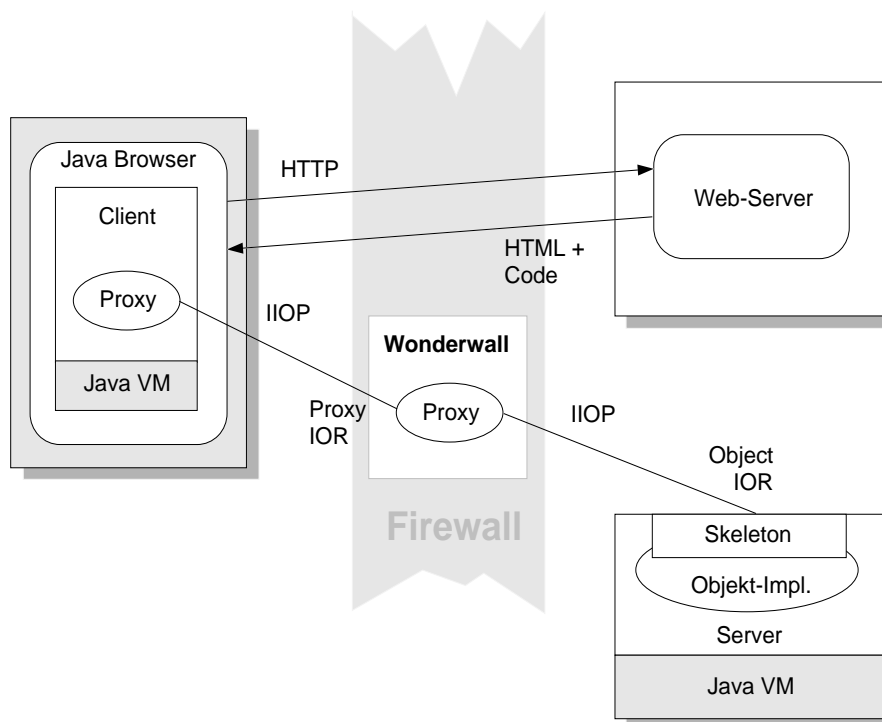


Abb. 3.4 Wonderwall

Die IOR eines CORBA Servers (Object IOR), der sich hinter der Firewall befinden, wird zu den Konfigurationsdaten des Wonderwall Proxys hinzugefügt. Mit Hilfe des Programms `iortool` erstellt man eine Proxy IOR, in der Server Host und Port der ursprünglichen IOR durch den Host und Port des Wonderwall Proxys ersetzt werden. Ein Client benutzt die Proxy IOR, um ein Server-Objekt anzufordern. Diese Anfrage gelangt erst einmal zum Wonderwall Proxy, der sie überprüft und die interne Kopie der IOR (Object IOR) benutzt, um den gewünschten Server zu kontaktieren.

Die Integration von Wonderwall in eine bestehende Firewall erfordert im Vergleich zu Visigenics Gatekeeper einen erheblich größeren administrativen Aufwand. Ein weiteres Problem besteht darin, daß es für ein Objekt nun zwei Objektreferenzen gibt. Um einen Server hinter einer Firewall über Wonderwall aufzurufen, muß die Objektreferenz so modifiziert werden, daß sie Informationen über das Proxy Objekt auf dem Wonderwall Host enthält. Eine zweite Objektreferenz gibt es für den Gebrauch ohne Wonderwall, z.B. innerhalb des Intranets.

IONA sieht die Schwächen des HTTP Tunneling Konzepts, das z.B. von Visigenic verwendet wird, darin, daß die Sicherheit allein von Seiten der Anwendung gewährleistet werden muß und nicht mehr von der Firewall erbracht werden kann. Performanceeinbußen durch das Einfügen einer weiteren Codierungs-/Decoderierungsschicht sind ebenfalls zu erwarten. Dennoch bietet IONA für OrbixWeb 2.01 einen Patch an, der HTTP Tunneling ermöglicht. Es gibt dazu aber keine näheren Informationen.

3.2.5 Gesamteindruck

IONA hat mit OrbixWeb 2.01 [Ion96b] gegenüber der Konkurrenz enorm aufgeholt. Es ist nun möglich, nicht nur Clients, sondern auch Server in Java zu implementieren. Dadurch ergibt

sich auch die Möglichkeit von Callbacks von Servern zu Clients. Zur Erstellung von Servern wird kein installiertes Orbix mehr benötigt, die notwendigen Zusatzprogramme, wie z.B. der Orbix-Dämon werden mitgeliefert. Die Dokumentation ist sehr umfangreich und geht detailliert auf die gebotenen Funktionen ein. Die mitgelieferten Beispiele vermitteln einen guten Eindruck über die Möglichkeiten von OrbixWeb und decken alle gebotenen Funktionen ab. Wie bei allen anderen Produkten auch, ist die Performance nicht besonders gut. Das Nachladen der Klassen dauert in der Beta 2 im Vergleich zur Version 1.0 sogar noch länger, was sich durch die neu hinzugekommenen Klassen erklären läßt, die für die server-seitige Funktionalität zuständig sind. Erfreulich ist die Preisgestaltung von IONA, das den Preis für eine Entwicklungslizenz stark reduziert hat und keine client-seitigen Runtime-Gebühren erhebt.

3.3 Joe (SunSoft)

Sun gehört zu den Gründungsmitgliedern der OMG und bietet mit NEO seit längerem eine CORBA-Implementierung an, die es erlaubt, in IDL definierte Objekte mit C oder C++ zu implementieren. Seit Juli 1996 ist von SunSoft auch ein Java ORB namens Joe erhältlich, der den Zugriff auf NEO Objekte unterstützt. Da Joe vollständig in Java geschrieben ist und der IDL Compiler Java Code erzeugt, besteht auch bei Joe die Möglichkeit, Client Applets innerhalb von Webseiten ablaufen zu lassen.

3.3.1 Informationen

SunSoft bietet auf seiner Internet Homepage einige aufwendig gestaltete Web-Seiten zu Joe und NEO an [Joe96]. Erhältlich sind Produktüberblicke zu Joe und NEO, White Papers und FAQs.

3.3.2 Verfügbarkeit

Joe ist inzwischen in der Version 2.0 verfügbar und nur noch als Bestandteil von NEO 2.0 oder dem Internet Workshop 1.0 erhältlich. Die Version 1.0 kann weiterhin kostenlos über das Internet geladen werden, Preise für NEO 2.0 und Internet Workshop 1.0 waren bisher nicht zu erfahren. An Plattformen wird nur Solaris ab der Version 2.4 unterstützt, zusätzlich wird Solaris NEO 1.0 bzw. NEO 2.0 benötigt, da auf Server-Seite nur NEO Services unterstützt werden. Das Internet Inter-ORB Protokoll ist in Joe und NEO ab der Version 2.0 integriert und erlaubt es Joe Clients, auch mit anderen CORBA Implementierungen zu kommunizieren.

3.3.3 Funktionalität

Joe ist nur eine client-seitiger Java ORB, man kann also keine Server-Komponenten damit implementieren. Server-seitige Objekte müssen für die NEO Plattform geschrieben werden, was den Einsatz auf Solaris beschränkt. Die Klassen des Joe ORBs lassen sich automatisch mit einem Java Applet in einen Web-Browser laden. Joe kümmert sich dann um die Erstellung einer Verbindung zwischen lokalen Java Objekten und entfernten NEO Objekten und deren Kommunikation. An Protokollen unterstützt Joe das proprietäre NEO und Door Protokoll. Ab der Version 2.0 ist auch die Kommunikation über IIOP möglich, wodurch man auf Server-Seite nicht mehr nur auf NEO beschränkt ist. Abb. 3.5 gibt eine Übersicht über das NEO/Joe System.

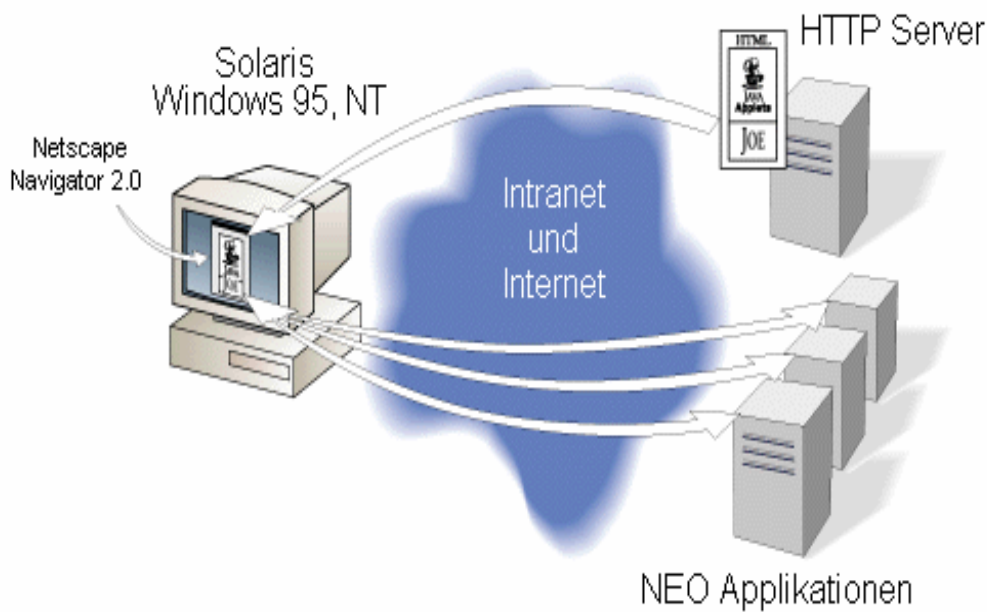


Abb. 3.5 Überblick über das Joe System

Der IDL Compiler von Joe erzeugt automatisch die Java Klassenstubs aus einer IDL-Schnittstelle für NEO Objekte, wie Abb. 3.6 zeigt. Die Implementierung der Schnittstellen ist sprachunabhängig, wobei man bei NEO allerdings auf C oder C++ eingeschränkt ist.

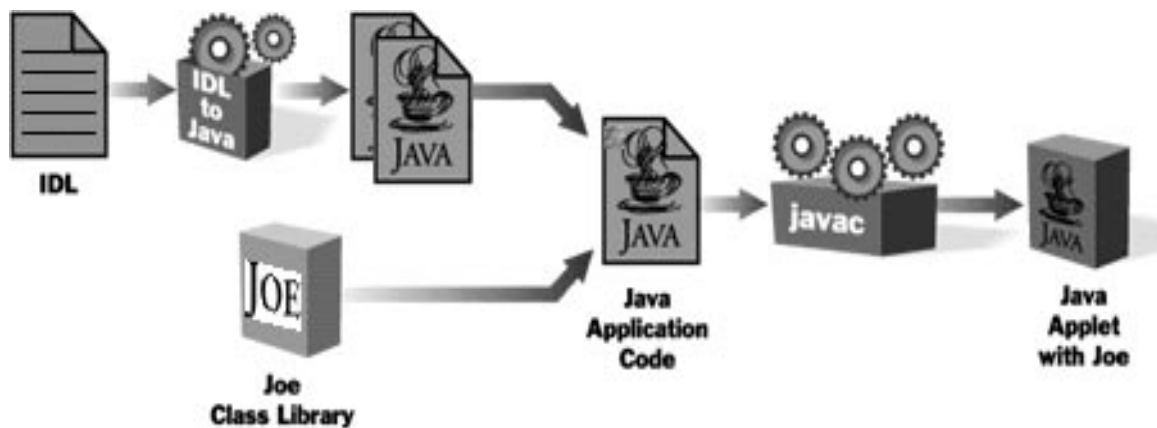


Abb. 3.6 Die einzelnen Schritte bei der Erzeugung eines Java Applets mit Joe

Die skalierbare Architektur von Joe ermöglicht es, das Aufkommen hoher Lasten zu regulieren. Sobald ein Client Applet in den Browser geladen ist, erleichtert Joe die direkte Kommunikation mit Server Objekten durch Umgehung des HTTP Servers. Der HTTP Server wird dadurch wieder frei für die Aufgaben der Bereitstellung von HTML Seiten und Java Applets.

Durch die Unterstützung von Remote Callbacks können Java Applets asynchron Nachrichten empfangen. Dies erspart das kostenintensive Nachfragen, ob sich beim Server Veränderungen ergeben haben.

Zusätzlich werden noch einige Tools zur Verwaltung, Entwicklung und Programmierung mitgeliefert.

- Administrationstools
Mit Hilfe dieser Tools können Netzwerkadministratoren die Dienste von NEO über einen Web-Browser zu verwalten.
- Entwicklungstools
Diese Tools sollen den Programmierer bei Projekt-Spezifikation, Management und Entwicklung unterstützen.
- Programmierertools
Zusammen mit Joe erhält man einige Programming Support Libraries, welche die Netzwerkprogrammierung für verteilte Objekte vereinfachen sollen.

Joe ermöglicht den Zugriff auf verteilte Objekte und deren Dienste auch durch Standard-Firewalls hindurch. Dazu wird wie bei VisiBroker for Java die Technik des HTTP Tunneling verwendet. Der mit Joe mitgelieferte Apache HTTP Server wird mit einem Patch so modifiziert, daß IIOP Anfragen in HTTP gekapselt werden. Ein Nachteil dieses Verfahrens ist natürlich die Einschränkung auf einen speziellen Web-Server.

3.3.4 Gesamteindruck

Joe [Merk96] bietet einen ähnlichen Funktionsumfang wie VisiBroker for Java, es lassen sich allerdings keine Server mit Java erstellen. Die Informationen zum Produkt sind relativ umfangreich und es werden viele nützliche Werkzeuge mitgeliefert. Deutlich negativ ins Gewicht fällt allerdings, daß bisher nur NEO Services und Solaris 2.4 unterstützt werden. Durch die Integration des IIOP ist es ab der Version 2.0 aber möglich, Verbindung zu anderen ORBs aufzunehmen, wie es mit OrbixWeb und VisiBroker for Java seit längerem möglich ist. Da Joe 1.0 das IIOP nicht unterstützt, konnten mit diesem Produkt keine praktischen Erfahrungen gewonnen werden. Wie zu erwarten war, ist Joe 2.0 nicht mehr kostenlos verfügbar. Einzige Neuerung scheint nach jetzigem Kenntnisstand allerdings nur die Integration des IIOP zu sein.

3.4 PowerBroker CORBAplus Java Edition (Expersoft)

Die Firma Expersoft bietet unter dem Namen PowerBroker eine Produktpalette an, die ORBs, Object Services und Object Management Tools beinhaltet. PowerBroker CORBAplus ist eine CORBA-Implementierung, die zum 2.0 Standard konform ist. Neben den seit längerem verfügbaren Versionen für C++, OLE und Smalltalk ist seit Ende Januar 1997 auch eine sogenannte Java Edition erhältlich, eine Implementierung von PowerBroker CORBAplus in Java. Im folgenden wird dieses Produkt als PBC+ abgekürzt.

3.4.1 Informationen

Auf der Homepage von Expersoft [PBC97] findet man eine Presseverlautbarung, eine kurze Produktübersicht und eine Liste mit Antworten zu häufig gestellten Fragen. Im Rahmen eines Wettbewerbs, bei dem Anwendungen in Verbindung mit dieser Software erstellt werden sollen, versucht Expersoft zusätzlich Werbung für seine Java Edition zu machen. Die Dokumentation ist inzwischen auch Online abrufbar.

3.4.2 Verfügbarkeit

PBC+ Java Edition ist derzeit in der Version 2.1 verfügbar. Das Produkt kann kostenlos von

der Expertsoft Homepage geladen werden und ist ohne Aufpreis auch als Bestandteil von CORBAplus for C++ erhältlich. Der optionale Support kostet \$ 450 pro Jahr und Entwicklerlizenz. An Plattformen werden Solaris und Windows 95/NT unterstützt. An zusätzlicher Software wird der IDL Compiler IDLgen 2.2 alpha von JavaSoft benötigt, das Produkt verfügt über keinen eigenen IDL Compiler.

3.4.3 Funktionalität

Die Java Edition von PBC+ basiert auf den bereits von Expertsoft verfügbaren ORBs für C++, OLE und Smalltalk und ist zum CORBA 2.0 Standard konform. Als Protokoll wird das IIOP verwendet, wodurch die Verbindung zu anderen ORBs möglich wird. Expertsoft verzichtet auf einen eigenen IDL Compiler und setzt eine Benutzung des IDL Compilers von JavaSoft voraus, der noch das alte Java Mapping von JavaSoft verwendet. Durch die Anlehnung an dieses Mapping entspricht PBC+ im Hinblick auf Architektur und Funktionsumfang der 2.2 Alpha Version von Java IDL. Man kann sowohl Clients, als auch Server in Java implementieren. Der mit Java IDL 1.1 mitgelieferte IDL Compiler wird nicht unterstützt, da dieser bereits im wesentlichen das von der OMG standardisierte Java Mapping verwendet. Der OMG Standard soll von PBC+ in der nächsten Version unterstützt werden.

PBC+ verfügt weder über einen Namensdienst, noch gibt es wie bei anderen Produkten Dämon-Prozesse. Ein Server wird an an einem festen Port gestartet, pro Server wird also ein Port verbraucht. Zur Verbindung mit Servern benutzt ein Client eine sogenannte URL Objektreferenz, welche die Form `iiop://<host>:<port>/server` hat. Die von CORBA geforderte Ortstransparenz läßt sich mit PBC+ folglich nicht erreichen. Es gibt auch keine Möglichkeit, Server bei Bedarf zu aktivieren. Bisher sind nur statische Methodenaufrufe möglich, das DII wird nicht unterstützt.

3.4.4 Gesamteindruck

Das Produkt ist noch ziemlich neu und daher vom Entwicklungsstand nicht so weit wie die kommerziellen Mitbewerber. PBC+ ist der einzige Java ORB, der über keinen eigenen IDL Compiler verfügt. Man ist deshalb auf den IDL Compiler von JavaSoft angewiesen, noch dazu auf das 2.2 Alpha Release. Neuere Versionen des IDL Compilers werden noch nicht unterstützt, man hinkt also immer einen Schritt hinterher. Dies wird sich wohl erst bessern, wenn das standardisierte Java Mapping unterstützt wird. Positiv ist die Verwendung des IIOP als Protokoll, das sich auf breiter Front durchzusetzen scheint. Expertsoft hat mit seiner PowerBroker Produktpalette zwar einige Preise gewonnen, die gegenwärtige Version fällt gegenüber der Konkurrenz von Visigenic, Iona und Sun aber deutlich ab.

3.5 JacORB (Freie Universität Berlin)

JacORB ist ein Object Request Broker, der in Java geschrieben ist und an der Freien Universität in Berlin von Gerald Brose entwickelt wird. In seiner jetzigen Form ist JacORB nicht vollständig konform zum CORBA Standard und noch im frühen Entwicklungsstadium. Das erklärte Ziel dieses Projekts ist allerdings die Konformität zum CORBA Standard, was mit fortlaufender Entwicklung umgesetzt werden soll.

3.5.1 Informationen

Alle derzeit verfügbaren Informationen stammen von der JacORB Homepage [Bros96]. Es

gibt einen Überblick über das Projekt und dessen Zielsetzung, eine API Dokumentation, Informationen zum IDL Compiler und eine Beschreibung des verwendeten Java Mappings. Der Autor von JacORB hat auch eine ganze Reihe von Artikeln veröffentlicht [Bro97a, Bro97b, BrBo97].

3.5.2 Verfügbarkeit

Die aktuelle Version von JacORB trägt die Versionsnummer 0.6c und kann kostenlos von der Homepage geladen werden. Darin enthalten sind die komplette Software, Dokumentation und Beispiele. Da sämtliche Komponenten in Java geschrieben wurden, kann JacORB auf allen Plattformen eingesetzt werden, für die es eine Java-Unterstützung gibt. Ab der Version 0.5d wird das JDK 1.1 benötigt.

3.5.3 Funktionalität

In der zur Zeit verfügbaren Version von JacORB fehlen einige CORBA Details wie das Dynamic Invocation Interface und der Basic Object Adapter. Deshalb sind nur statische Aufrufe möglich, Server müssen von Hand gestartet werden und können nicht automatisch aktiviert werden. Als Kommunikationsprotokoll wird, wie inzwischen allgemein üblich, das IIOP verwendet. Beim Start eines Servers gibt dieser eine Inter Object Reference (IOR) von sich aus, die bei dem als CORBA Object Service implementierten Namensdienst registriert wird und von aufrufenden Clients abgerufen werden kann. Mit Hilfe dieser IOR kann sich der Client direkt mit dem Server verbinden, weitere Dämon-Prozesse werden nicht benötigt. Als Entwicklungskomponente dient ein IDL Compiler, der wie üblich Java Stubs und Skeletons erzeugt und zum jetzigen Zeitpunkt noch nicht das CORBA Any unterstützt.

3.5.4 Gesamteindruck

JacORB ist derzeit noch ein sehr eingeschränkter Java ORB, der sich im frühen Entwicklungsstadium befindet. Aus diesem Grunde fehlen einige CORBA Komponenten, die aber mit fortlaufender Entwicklung integriert werden sollen, was im Falle des IIOP bereits geschehen ist. Im Gegensatz zu den kommerziellen Konkurrenzprodukten sind wirklich alle Bestandteile in Java geschrieben, wodurch sich JacORB sehr flexibel einsetzen läßt. Zudem ist das Produkt kostenlos, auch der Sourcecode ist frei verfügbar. Ob das Produkt gegen die kommerzielle Konkurrenz bestehen kann, bleibt abzuwarten, die Ansätze sehen allerdings recht vielversprechend aus.

3.6 JIDL (Sandia National Labs)

JIDL ist ein CORBA IDL Compiler, der Java Code erzeugt. Dadurch besteht die Möglichkeit, innerhalb von Java Applets auf verteilte CORBA Objekte zuzugreifen, als wären sie lokale Objekte.

3.6.1 Informationen

Die Informationen über JIDL [Frie96] sind sehr spärlich und beschränken sich auf einen kurzen Überblick, einen Entwurf zum IDL nach Java Mapping und ein Beispiel zum Ausprobieren. Es handelt sich dabei um das von Orbix her bekannte Grid Beispiel, in dem ein Array mit Werten gefüllt werden kann, die sich nachher abfragen lassen. Der Sourcecode zum Beispiel ist frei verfügbar. Eine Dokumentation aller von JIDL benutzten Klassen, sowie auch aller

sonstigen JDK 1.0 Klassen, ist Online abrufbar.

3.6.2 Verfügbarkeit

JIDL befindet sich noch im frühen Alpha Stadium und ist nur innerhalb von Sandia National Labs zugänglich. Eine öffentlich erhältliche Betaversion war für Februar 1996 angekündigt, ist aber bis zum jetzigen Zeitpunkt nicht erhältlich. Mit dem Erscheinen von kommerziellen Konkurrenzprodukten wie OrbixWeb und VisiBroker for Java wurde die Entwicklung im Alphastadium gestoppt und bisher nicht mehr aufgenommen.

3.6.3 Funktionalität

Das JIDL System, das von Ernest Friedman-Hill in den Sandia National Laboratories geschrieben wurde, arbeitet mit dem kommerziellen ORB Orbix von IONA zusammen. Der von JIDL erzeugte Code kommuniziert mit Orbix durch ein generisches (nicht objektspezifisches) Script auf der Serverseite, welches TclDii (Tcl Dynamic Invocation Interface), eine Erweiterung von Tcl, benutzt. Diese Vorgehensweise wurde aber nur verwendet, um möglichst schnell einen lauffähigen Prototypen erstellen zu können. Spätere Versionen von JIDL sollen auch das IIOP unterstützen.

3.6.4 Gesamteindruck

Die Informationen zu JIDL sind sehr dünn gesät, das Produkt war bis zu seiner Einstellung nicht mehr als ein Prototyp. Obwohl für Februar 1996 angekündigt, liegt bisher keine öffentlich verfügbare Betaversion vor. Die Entwickler wurden offensichtlich von der kommerziellen Konkurrenz überholt, auf deren Produkte sie auch verweisen. Da das JIDL System Public Domain sein sollte, lohnte sich eine Weiterentwicklung im Anbetracht der bereits verfügbaren und besseren Konkurrenzprodukte nicht mehr.

3.7 Spring Java IDL (Sun)

Mit dem Spring-Java-IDL System können Java Applets geschrieben werden, die über eine IDL Schnittstelle mit einem Spring 1.1 System oder mit anderen Applets kommunizieren können. Spring ist ein modulares, objektorientiertes, verteiltes Betriebssystem, das auf einer einheitlichen Schnittstellendefinitionssprache beruht, die im wesentlichen mit der IDL der OMG übereinstimmt.

3.7.1 Informationen

Alle verfügbaren Informationen stammen von SUNs World Wide Web Server [Sun96]. Es gibt nur einen sehr kurzen Überblick über den Inhalt des Spring Java IDL Systems und dessen Grenzen. Die Seiten sind allerdings seit Monaten nicht mehr verändert bzw. aktualisiert worden.

3.7.2 Verfügbarkeit

Das Spring Java IDL System kann als Teil der Spring Research Distribution 1.1 erworben werden und kostet \$75 für Universitäten, \$750 für alle anderen. An Plattformen wird nur SUN SPARC ab Solaris 2.3 oder SpringOS 1.1 unterstützt.

3.7.3 Funktionalität

Das Spring-Java IDL System ist noch eine sehr neue Komponente in Spring und stellt bestenfalls eine Alphaversion dar, die mit einer frühen Beta Version des Java Systems erstellt wurde. Es enthält den Stub Compiler `contojava`, welcher sowohl Server, als auch Client Java-Code erzeugt. Weiter sind die Java Klassen enthalten, die das Spring Proxy Protokoll implementieren, einige kleine Demos und eine kurze Dokumentation, sowie eine Kopie des Java Development Kits (JDK) in einer frühen Beta Version.

Die Kommunikation kann nur wahlweise zum Spring System über UDP erfolgen oder zu anderen Java Programmen über TCP. Dazu muß beim Start einer Java IDL Applikation die Klasse `sun.spring.Proxy` entsprechend initialisiert werden. Ein Java Programm kann nicht gleichzeitig mit einem Spring System und einem anderen Java Programm kommunizieren.

Der ORB Kern dieses Systems und der Java Stub Compiler sind die wesentlichen Bestandteile gewesen, um die Java-zu-NEO Verbindung zu implementieren, die unter dem Namen Joe veröffentlicht wurde.

3.7.4 Gesamteindruck

Das Spring-Java IDL System ist kein ORB, sondern bietet lediglich eine Java Sprachanbindung für das Betriebssystem Spring. Es besteht keine Möglichkeit mit anderen ORBs zu kommunizieren, die Kommunikation ist auf Spring 1.1 und andere Java Applikationen beschränkt und somit stark limitiert. Genauere Angaben zur Unterstützung von Applets sind nicht vorhanden. Da sich Umfang der vorhandenen Informationen und deren Inhalt über Monate nicht geändert hat, kann man davon ausgehen, daß Sun mehr auf NEO und Joe setzt und Spring nur schnell um das populäre Java erweitert wurde.

3.8 Java IDL / Remote Objects for Java (JavaSoft)

Die Firma JavaSoft ist eine Tochter von Sun und für die Entwicklung von Java verantwortlich. Zur Erstellung von verteilten Anwendungen mit Java bietet die Firma neben Java RMI ein weiteres Produkt namens Java IDL an. Wie der Name bereits andeutet, wird zur Beschreibung von Schnittstellen die IDL verwendet. Mit einem Compiler können daraus Stubs und Skeletons erzeugt werden, über die Java Clients auf CORBA-Objekte zugreifen können.

3.8.1 Informationen

Informationen über dieses Produkt können im World Wide Web [Jav97a] abgerufen werden. Es gibt einen Überblick über Java IDL, Dokumentation zum IDL nach Java Mapping und dem IDL Compiler, eine Beschreibung des Namensdienstes und Beispiele für Java IDL Anwendungen.

3.8.2 Verfügbarkeit

Java IDL hat den Status einer „Early Access“ 1.1 Version und ist zum gegenwärtigen Zeitpunkt nur für Solaris und 32-bit Windows verfügbar. Die Software kann kostenlos über FTP bezogen werden. Unter Windows ist zusätzlich noch Visual C++ 4.0 oder höher erforderlich, da zumindest dessen Precompiler benötigt wird. Javasoft hat angekündigt, Java IDL mit einer zukünftigen Ausgabe des Java Development Kits zu vertreiben (JDK 1.2). Da das Java Mapping erst Ende Juni 1997 verabschiedet wurde, ist Java IDL noch nicht im JDK 1.1 vorhanden.

3.8.3 Funktionalität

Die IDL Schnittstellenbeschreibung kann mit Hilfe des mitgelieferten `idltojava` Stub Generators kompiliert werden, der sowohl die Interfacedefinitionen, als auch die Client- und Serverstubs erzeugt. Der IDL Compiler ist nicht in Java geschrieben, was nicht ganz der „100% Pure Java“ Kampagne von Javasoft entspricht. Laut Auskunft von JavaSoft wurde der IDL Compiler aus Zeitgründen bisher noch nicht nach Java portiert. Dies kann und wird sich bis zur Verfügbarkeit der endgültigen Version sicher noch ändern.

Java IDL beinhaltet in der „Early Access“ Version:

1. Client Runtime Libraries, die es ermöglichen, Java Clients sowohl als stand-alone Java Anwendungen, als auch als Applets innerhalb von Web Browsern laufen zu lassen
2. Server Runtime Libraries, die es Java Applikationen ermöglichen, als Server für CORBA Clients zu fungieren
3. Stub Generator `idltojava`, der automatisch die entsprechenden Stubs aus der IDL Schnittstellenbeschreibung erzeugt.
4. `nameserv`, eine Implementierung des CORBA (COS) Namensdienstes

Der portable ORB Kern

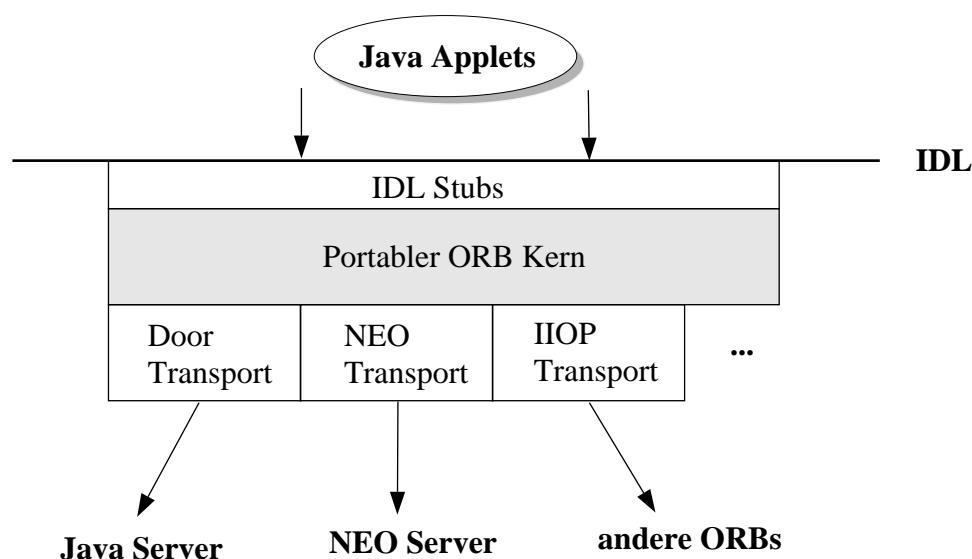


Abb. 3.7 Architektur des Java IDL Systems

Das Java IDL System basiert auf einem portablen ORB Kern, der so strukturiert ist, daß leicht andere ORB Protokolle eingefügt werden können. Die von dem Stub Generator erzeugten Stubs sind unabhängig von dem verwendeten ORB und rufen für Datenumwandlungen („Marshalling“) und andere ORB spezifische Operationen das entsprechende ORB Modul auf.

Java IDL verfügt über einen CORBA (COS) konformen Namensdienst, der es ermöglicht, CORBA Objekte mit Namen zu verknüpfen und Objektreferenzen über einen Namen abzurufen. Beim Start legt ein Server seine Objektreferenz unter einem Namen beim Namensdienst ab. Ein Client kann über diesen Namen vom Namensdienst die entsprechende Objektreferenz erfragen und damit die gewünschten Servermethoden aufrufen. In der aktuellen Version werden nicht alle CORBA Details unterstützt. So kann man nur statische Methodenaufrufe aus-

führen. Dynamische Methodenaufrufe sind nicht möglich, da es keine Interface Repository gibt.

Intern verwendet Java IDL ein eigenes Protokoll, das sogenannte Door ORB Protokoll. Darüber hinaus wird aber auch das IIOP unterstützt: Die Kommunikation zu CORBA Objekten, die von anderen ORBs verwaltet werden, ist also gewährleistet. SunSoft arbeitet an einem Modul zur direkten Anbindung von NEO.

3.8.4 Gesamteindruck

Java IDL ähnelt vom Konzept her stark dem in Kapitel 3.5 beschriebenen JacORB. Beide Produkte setzen auf einen Namensdienst zur Bereitstellung von Objektreferenzen. Der einzige Unterschied besteht darin, daß der IDL Compiler von Java IDL nicht in Java geschrieben ist. Das IIOP ist inzwischen integriert und ermöglicht die Verbindung zu anderen CORBA ORBs.

Der wesentliche Vorteil von Java IDL ist darin zu sehen, daß es Bestandteil des JDK 1.2 sein wird. Dadurch wird es früher oder später Zugang zu allen Java-fähigen Browsern finden, auch dem Internet Explorer von Microsoft. Es wäre dann von jedem Java Browser aus möglich, direkt ohne Nachladen eines ORBs über IIOP auf CORBA Objekte zuzugreifen. Wie schnell sich dies verwirklichen läßt, wird die Zeit zeigen. Nach der Auslieferung eines neuen JDKs durch JavaSoft brauchen die Hersteller von Java-fähigen Browsern im Minimum ein halbes Jahr, um ihre Browser anzupassen.

3.9 HORB (Electrotechnical Lab. / Dr. Hirano Satoshi)

HORB ist eine portable, verteilte, objektorientierte Programmiersprache, die nicht dem CORBA Standard der OMG folgt. Es handelt sich im wesentlichen um einen Zusatz für Java, der es ermöglicht, verteilte Objekte zu erstellen und auf diese zuzugreifen.

3.9.1 Informationen

Die HORB Home Page [HORB96] bietet eine Fülle von Informationen, wie Produktbeschreibungen, Installationshinweise, Demos und Dokumentationen. Die Dokumentation ist sehr umfangreich und wird ständig erweitert. Wenn man Hilfe braucht, kann man sich in eine Mailingliste eintragen lassen, die auch archiviert wird, oder in einem FAQ nachlesen.

3.9.2 Verfügbarkeit

HORB gibt es seit Ende November 1995 und ist inzwischen als Betaversion 1.3.1 b1 veröffentlicht. Das HORB Paket kann von der Home Page kostenlos geladen werden und beinhaltet den Compiler horbc, den HORB Server, das ORB Laufzeitsystem, die gesamte Dokumentation und den vollständigen Sourcecode. Durch seine Architekturunabhängigkeit läuft HORB auf allen Plattformen, die von Java unterstützt werden. Der Einsatz von HORB in Rahmen von Forschungs- und Lehrvorhaben ist kostenlos, für kommerzielle Nutzung wird eine Lizenz benötigt.

3.9.3 Funktionalität

HORB ist entwickelt worden, um speziell paralleles und verteiltes Programmieren zu erleichtern, ohne sich auf systemspezifische Dinge wie Sockets konzentrieren zu müssen. Das System beinhaltet dafür kleine Runtime Bibliotheken, die es ermöglichen, verteilte Objekte zu

erstellen und deren Methoden aus anderen Objekten heraus aufzurufen.

Architektur

HORB ist hundert prozentig kompatibel zu Java, der Compiler `horbc` nutzt intern den Java Compiler `javac`. Bei der Erstellung von HORB wurde darauf geachtet, den Sourcecode von SUN nicht zu verändern, wodurch HORB portabel bleibt und frei von SUNs Sourcecode Lizenzen ist.

Bei der Programmerstellung hat man die Möglichkeit, das Client Objekt in einem java-fähigen Browser zu nutzen oder als eigenständiges Programm zu realisieren. Zunächst schreibt man eine Server und eine Client Klasse. Beim Programmstart erzeugt das Client Objekt eine Instanz des Server Objekts im Server System und ruft die von dem Server zur Verfügung gestellten Methoden auf. Damit die Methodenaufrufe zwischen den beiden Objekten transparent sind, wird dazu ein Proxy Objekt, ein Skeleton Objekt und ein ORB benötigt.

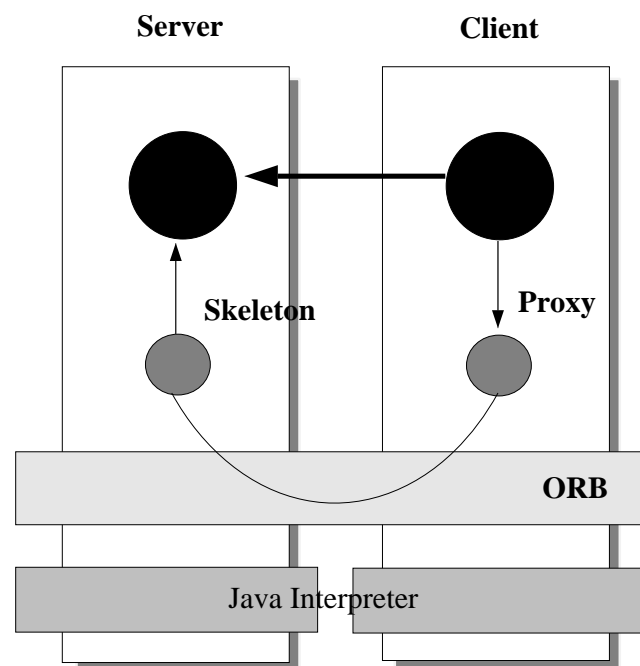


Abb. 3.8 Architektur von HORB

Das Proxy Objekt ist ein Stellvertreter für das Server Objekt und verhält sich für den Client wie der Server, enthält aber nur die Stubroutinen. Das Skeleton Objekt ist der Gegenpart zum Proxy Objekt und enthält ebenfalls die Stubroutinen. Der ORB ist zuständig für die Bereitstellung und Lokalisierung der Server Objekte.

Der HORB Compiler

Der Compiler `horbc` ist ein richtiger Übersetzer, d.h. er übersetzt Java Quelldateien in die entsprechenden Java Klassen. Darüberhinaus erzeugt er für die Serverseite aus dem Servercode sowohl das Proxy, als auch das Skeleton Objekt. Auf der Clientseite kann auch der normale Java Compiler `javac` verwendet werden, da hier ja kein Proxy oder Skeleton Objekt benötigt wird. Man kann aber auch den HORB Compiler benutzen, wobei der Parameter `-c` verhindert, daß Proxy und Skeleton Klassen erzeugt werden.

Sicherheitsaspekte

Es gibt die Möglichkeit über eine Zugriffskontroll-Liste (ACL: Access Control List) diejenigen Hostrechner und Benutzer festzulegen, die auf bestimmte Klassen oder Methoden zugreifen dürfen. In HORB gibt es sogar eine verteilte ACL, d.h. mehrere ACLs von verteilten Rechnern werden in einer Liste integriert. Die Zusammenstellung dieser Liste muß in einer Konfigurationsdatei festgelegt werden.

3.9.4 Vergleich mit CORBA 2.0

- HORB benutzt im Gegensatz zum CORBA Standard keine IDL.
- CORBA 2.0 unterstützt die Objektübergabe per Referenz, nicht aber durch Kopieren. HORB unterstützt beide Arten.
- CORBA Programme sind im Gegensatz zu HORB Programmen im allgemeinen nicht portabel.

3.9.5 Gesamteindruck

HORB ist kein zum CORBA Standard konformer ORB, da auf eine IDL verzichtet wird. Das Produkt ist eher vergleichbar mit Java RMI. Dennoch kann man in ähnlicher Weise verteilte Objekte erzeugen und auf diese zugreifen. Der von HORB gebotene Funktionsumfang ist sehr groß und es gibt eine Reihe nützlicher Tools. Die Dokumentation ist umfangreich und geht detailliert auf die einzelnen Funktionen und Tools ein. Positiv ist weiterhin, daß HORB für den nicht kommerziellen Gebrauch nach wie vor kostenlos ist.

Aufgrund der gebotenen Funktionalität, der guten Dokumentation und der freien Verfügbarkeit ist HORB eine sehr interessante Alternative, auch wenn HORB nicht CORBA konform ist. Die Ergebnisse von Benchmarktests mit HORB und RMI sprechen deutlich für HORB.

3.10 JYLU (Stanford Digital Library Testbed Development)

Jylu ist eine Implementierung des ILU 2.0 (Inter-Language Unification) Runtime Kernels von Xerox PARC in der Sprache Java mit entsprechender Sprachanbindung. Wie mit den anderen vorgestellten Produkten kann man mit JYLU Client Applets innerhalb von Webseiten ablaufen lassen.

3.10.1 Informationen

Die im Internet gefundenen Informationen [JYLU96] beinhalten einen kurzen Produktüberblick und eine Beschreibung des ISL nach Java Mappings. Eine Tabelle gibt Auskunft über die unterstützten Merkmale, weitere Dokumentation ist nicht vorhanden. Unter der Adresse jylu@findmail.com gibt es auch eine Mailingliste.

3.10.2 Verfügbarkeit

JYLU liegt derzeit unter der Versionsnummer 0.17 vor, die kompatibel zu ILU 2.0 alpha 9 ist und kostenlos vom WWW-Server geladen werden kann. Da Jylu komplett in Java geschrieben wurde, kann es auf allen von Java unterstützten Plattformen ausgeführt werden. Den Java Stub Generator gibt es in zwei verschiedenen Versionen. Zum einen kann man eine Web Version

benutzen, die eine IDL oder ISL akzeptiert und eine tar Datei mittels CGI zurückliefert. Zum anderen kann man den Java Stubber selbst installieren, wobei allerdings ILU 2.0 installiert sein muß.

3.10.3 Funktionalität

ILU ist für die transparente, adressraumübergreifende, system- und sprachunabhängige Zusammenarbeit von Objekten gedacht. Hinter der von ILU zur Verfügung gestellte Objektschnittstelle werden Implementierungsdetails zwischen verschiedenen Sprachen, Adreßräumen und Betriebssystemen verborgen.

ILU ist kein kompletter ORB im Sinne des CORBA Standards, da einige Funktionen nicht unterstützt werden. Es fehlt das Dynamic Invocation Interface, ebenso wie das Interface- und Implementation Repository.

Zudem arbeitet ILU mit einer eigenen Schnittstellen-Beschreibungssprache ISL (Interface Specification Language), einer leicht abgewandelten Form der IDL. Der Stubcompiler von JYLU kann aber sowohl ISL als auch IDL Schnittstellenbeschreibungen übersetzen. JYLU unterstützt zur Zeit noch nicht das IIOP und kann deshalb nicht mit ORBs des CORBA 2.0 Standards kommunizieren.

3.10.4 Gesamteindruck

Jylu befindet sich noch in einem sehr frühen Alpha Stadium und hat selbst nach Aussage der Entwickler noch einige Ecken und Kanten. Die Methode, den Java Stubber über das WWW zu betreiben, ist sicherlich nicht sehr praktikabel und stellt nur eine Notlösung dar. Nachteil der Installation des Java Stubbers ist, daß ILU 2.0 benötigt wird. Einige CORBA Merkmale fehlen, und IIOP wird nicht unterstützt, obwohl ILU 2.0 dies laut dem Produktüberblick von Xerox beinhaltet.

3.11 Jade (Architecture Projects Management Ltd.)

APM Ltd. entwickelt bereits seit Anfang 1995 Technologien um CORBA und das World-Wide Web zu verbinden. Im Rahmen des ANSA Programms wurden zwei mögliche Wege für diese Integration untersucht. Zum einen gibt es die Möglichkeit mit Hilfe von sogenannten Gateways HTTP in IIOP zu übersetzen, was mit dem Produkt ANSAweb verwirklicht wurde. Zum anderen gibt es ein Projekt, das sich zum Ziel gesetzt hat, WWW-Clients und Server mit CORBA-Fähigkeiten zu versehen. Das Produkt Jade bietet dabei die Möglichkeit IIOP Funktionalität in Form eines Java Moduls in den Browser zu laden und mittels Java Applets auf CORBA 2.0 Dienste zuzugreifen.

3.11.1 Informationen

Die spärlichen Informationen können der Jade Web-Page [Jade96] entnommen werden und beinhalten ein kurzen Überblick über das Projekt und eine Beschreibung der Jade API. Demos- anwendungen sind ebenfalls vorhanden und können Online ausprobiert werden, teilweise ist dafür auch eine Dokumentation erhältlich.

3.11.2 Verfügbarkeit

ANSA Sponsoren und Object Lab Mitglieder können Jade kostenlos vom FTP-Server von APM laden. Alle übrigen müssen zur Vereinbarung von Lizenzbedingungen APM kontaktieren. Verfügbar sind das Jade CORBA 2.0 IIOP Modul, Demos, Dokumentation und Infos zur Vermarktung des Produkts. Da Jade komplett in Java geschrieben ist, sollten alle Plattformen unterstützt werden, für die es eine virtuelle Java-Maschine gibt.

3.11.3 Funktionalität

Jade beinhaltet ein in Java geschriebenes CORBA 2.0 IIOP Modul, das in einen Web-Browser geladen werden kann und damit dem Client Zugriff auf CORBA-basierte Serveranwendungen ermöglicht. Durch Verwendung des IIOP ist man dabei nicht auf einen einzelnen ORB Hersteller festgelegt, sondern kann auf Serverseite alle CORBA 2.0 konformen ORB Implementierungen benutzen. Jade ist also nur ein Client-seitiges Kommunikationsmodul, zur Implementierung von Servern wird ein ORB benötigt. Für die auf den Web-Seiten verfügbaren Beispielanwendungen wurde auf Serverseite VisiBroker for Java eingesetzt.

Weitere Informationen über Möglichkeiten und Aufbau von Jade waren leider nicht zu erhalten. Die im WWW zur Verfügung gestellten Seiten beinhalten im wesentlichen nur Werbung für die Kernidee des Zugriffs auf CORBA-Dienste über das Internet.

3.11.4 Gesamteindruck

Jade ist nicht mehr und nicht weniger als eine client-seitige Kommunikationskomponente, die es Java Applets erlaubt auf CORBA Server zuzugreifen. Wenig Sinn macht eine Kombination mit Java ORBs, da diese bereits zum Großteil das IIOP auf Client- und Serverseite unterstützen. Interessant ist Jade sicherlich für Anbieter bestehender CORBA Anwendungen, da diese ihre Dienste leicht über das WWW anbieten können, sofern die verwendete CORBA-Implementierung IIOP-fähig ist. Die verfügbaren Informationen geben aber keinen Aufschluß über die Leistungsfähigkeit von Jade, die Demoanwendungen zeigen ähnliche Ladezeiten und Performance wie VisiBroker und OrbixWeb. Da Netscape die VisiBroker for Java Runtime in den Navigator 4.0 integriert, erwächst diesem Projekt eine starke Konkurrenz, da die IIOP Unterstützung bereits in den Browser eingebaut ist.

3.12 Zusammenfassung

Die Anzahl verfügbarer CORBA/Java Produkte ist im steten Wachstum begriffen. Die vorgestellten elf Produkte stellen den Stand der Dinge zum 31. Juli 1997 dar, es wird aber kein Anspruch auf Vollständigkeit erhoben. In der schnellebigen Zeit von Multimedia und Internet kann davon ausgegangen werden, daß sich in der Entwicklung von Java und CORBA in den nächsten Wochen und Monaten noch einiges tun wird. Bei der Zusammenstellung dieser Übersicht ergaben sich nahezu fortlaufend Änderungen. Einige Hersteller haben im Zuge der Umstellung auf das standardisierte Java Mapping bereits neue Versionen ihrer Produkte angekündigt.

Unter den kommerziellen Produkten läßt sich kein definitiver Gewinner ausmachen. Alle Produkte haben gewisse Vorzüge, aber auch Schwachpunkte. Man kann aber sicherlich VisiBroker for Java als das Produkt bezeichnen, das einen gewissen Vorsprung gegenüber der Konkurrenz aufweist. Dies zeigt sich auch darin, daß dieser Java ORB inzwischen von vielen renommierten Firmen lizenziert wurde.

Dicht dahinter folgen nahezu gleichauf OrbixWeb und Joe, wobei Joe aufgrund der geringen Plattform-Unterstützung etwas abfällt. Das Schlußlicht der kommerziellen Java ORBs bildet die Java Edition von PowerBroker CORBA Plus, die den geringsten Funktionsumfang bietet, allerdings auch noch sehr neu ist.

Die sieben vorgestellten Forschungsvorhaben bzw. nicht kommerziellen Produkte unterscheiden sich zum Teil erheblich und sind allesamt zum gegenwärtigen Zeitpunkt nicht in der Lage, mit der kommerziellen Konkurrenz mithalten. Wesentlicher Vorteil dieser Produkte ist, daß sie wesentlich kostengünstiger sind.

Die interessantesten Produkte sind sicherlich JacORB und Java IDL. JacORB ist gegenwärtig der einzige „100% Pure Java“ ORB, da alle Komponenten in Java geschrieben sind. Der Vorteil von Java IDL besteht darin, daß es Bestandteil der Java Plattform werden soll.

JYLU und HORB entsprechen nicht dem CORBA Standard, stellen aber weitere Möglichkeiten dar, um verteilte Anwendungen mit Java zu realisieren. Das Produkt Jade bietet als reines IIOP Modul relativ wenig. Spring-IDL von Sun ist aufgrund dessen, daß sich das Betriebssystem Spring kaum durchgesetzt hat, nahezu bedeutungslos, ebenso wie das eingestellte Projekt JIDL.

Die folgende Tabelle gibt einen Überblick über die wichtigsten Details, der in diesem Kapitel vorgestellten Produkte.

Produkt	CORBA 2.0 konform	Protokoll	Java Server Fkt.	Unterstützte Plattformen	Preis	Besonderheiten
VisiBroker for Java 2.5	ja	IIOP	ja	Solaris, Win 95/NT, [AIX, IRIX, HP-UX]	\$ 2.995 (UNIX), \$ 1.995 (Winows)	erster verfügbarer Java ORB, integriert in Netscape 4.0
OrbixWeb 2.01	ja	IIOP, Orbix	ja	Solaris, Win 95/NT, HP-UX	\$ 799	umfangreiche Dokumentation, Adm. Tools
Joe 2.0	ja	IIOP	nein	Solaris	Vers. 1.0 kostenlos, 2.0 ?	benötigt NEO, Adm. Tools
PowerBroker CORBAplus 2.1	ja	IIOP	ja	Solaris, Win 95/NT	noch kostenlos	noch sehr neu, verfügt über keinen eigenen IDL Compiler

Tabelle 1: Übersicht CORBA & Java Produkte

Produkt	CORBA 2.0 konform	Protokoll	Java Server Fkt.	Unterstützte Plattformen	Preis	Besonderheiten
JacORB 0.5e	nur zum Teil	IOP	ja	alle von Java unterstützen	kostenlos	mit Sourcecode, 100% Pure Java
JIDL	nein	?	nein	?	nicht verfügbar	Projekt eingestellt
Spring-Java IDL	nein	?	ja	Spring 1.1, Solaris	\$ 750	
Java IDL 1.1	nein	Door ORB, IOP	ja	Solaris, Win 95/NT	kostenlos	voraussichtlich Bestandteil des JDK 1.2
HORB 1.31b	nein	eigenes	ja	alle von Java unterstützen	kostenlos	umfangreiche Dokumentation, mit Sourcecode
JYLU 0.17	nur zum Teil	?	ja	alle von Java unterstützen	kostenlos	
Jade	kein ORB	IOP	nein	alle von Java unterstützen	Lizenzgebühren unbekannt	reines IOP Modul

Tabelle 1: Übersicht CORBA & Java Produkte

4 Kommunikation zwischen ORBs unterschiedlicher Hersteller

Die wesentliche Schwäche der CORBA 1.x Spezifikation besteht darin, daß es keine verbindlichen Vorgaben zur Interoperabilität von ORBs unterschiedlicher Hersteller gibt. Da fast jedes CORBA 1.x konforme Produkt eine eigene Implementierungsstrategie verfolgt, können diese Produkte bis auf wenige Ausnahmen nicht zusammenarbeiten. Die OMG rief deshalb mit CORBA 2.0 einen Nachfolgestandard ins Leben, in dem festgelegt wird, wie unterschiedliche ORBs miteinander kommunizieren können [OMG95a]. Dieses Kapitel geht kurz auf die wesentlichen Details der Interoperabilitäts-Architektur ein und veranschaulicht diese anhand eines einfachen Beispiels unter Verwendung zweier ORBs unterschiedlicher Hersteller.

4.1 Die ORB Interoperabilitäts-Architektur

Im Abschnitt *Interoperability* der CORBA 2.0 Spezifikation werden Mechanismen und Strukturen festgelegt, um die Interoperabilität von unabhängig entwickelten ORBs zu gewährleisten. Als ORB Domäne wird darin der Bereich bezeichnet, in dem auf Objekte mittels desselben Kommunikationsprotokolls, derselben Sicherheitsvorkehrungen und dergleichen Art der Objektidentifizierung zugegriffen werden kann. Befinden sich zwei ORBs in derselben Domäne, können sie direkt miteinander kommunizieren. Soll ein Methodenaufruf eine Domäne verlassen, sieht die Spezifikation zwei Möglichkeiten vor: Inter-ORB Bridges und ein Interoperabilitäts-Protokoll.

4.1.1 Inter-ORB Bridges

Mit Hilfe von Inter-ORB Bridges kann ein Object Request Broker Methodenaufrufe an einen anderen ORB weiterleiten. Die Aufgabe einer Bridge besteht darin, den Aufruf vom Format des aufrufenden ORB in das Format des aufgerufenen zu konvertieren. Kommunizieren zwei Request Broker über eine Bridge, so wird auf beiden Seiten eine Half-Bridge benötigt, die stellvertretend für das Zielobjekt ein Proxy-Objekt erzeugt. Dieses übersetzt das lokale Aufrufformat in das Aufrufformat des Empfänger-ORB.

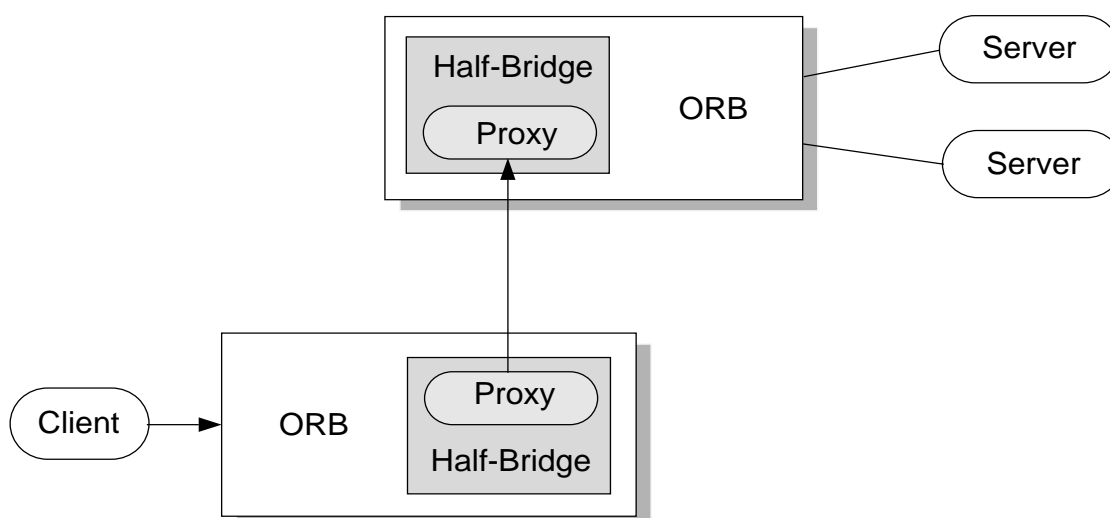


Abb. 4.1 Interoperabilität mittels Inter-ORB Bridges

Zur Kommunikation verwenden Bridges Standardprotokolle wie das GIOP (General Inter-ORB Protocol) oder das ESIOP (Environment-Specific Inter-ORB Protocol). CORBA 2.0 Implementierungen müssen das GIOP beinhalten, die Unterstützung von ESIOP ist optional. Das GIOP definiert Nachrichten- und Datenformate für die Übermittlung von CORBA Methodenaufrufen, ohne jedoch ein spezielles Netzwerktransportprotokoll zu benötigen. Inter-ORB Bridges können auch verwendet werden, um die Interoperabilität zu nicht-CORBA Systemen, wie z.B. Microsofts Component Object Model (COM), zu ermöglichen.

4.1.2 Das Interoperabilitäts-Protokoll

Im Gegensatz zu den Inter-ORB Bridges ermöglicht das Interoperabilitäts-Protokoll eine direkte Kommunikation von ORBs. Zuerst wird eine allgemeine Darstellung von Objekt-Referenzen benötigt. Eine domänen-übergreifende Beschreibung von Objektreferenzen wird durch IORs (Interoperable Object Reference) festgelegt, die zusätzliche Informationen wie etwa den Typ des Zielobjektes und dessen Aufenthaltsort enthalten. Als eigentliches Protokoll zur Übermittlung von Methodenaufrufen wird das IIOP (Internet Inter-ORB Protocol) definiert, dessen Unterstützung für jeden CORBA 2.0 konformen ORB verpflichtend ist. Das IIOP bildet das im GIOP festgelegte Nachrichten- und Datenformat auf TCP/IP ab, welches das Standard-Netzwerkprotokoll im Internet ist. Es legt fest, wie das Kodieren von Objektreferenzen, Methodenaufrufen, deren Parametern und Rückgabewerten in TCP/IP vorzunehmen ist. Das IIOP kann wie im Falle der ORBs der Firma Visigenic auch als internes Protokoll verwendet werden.

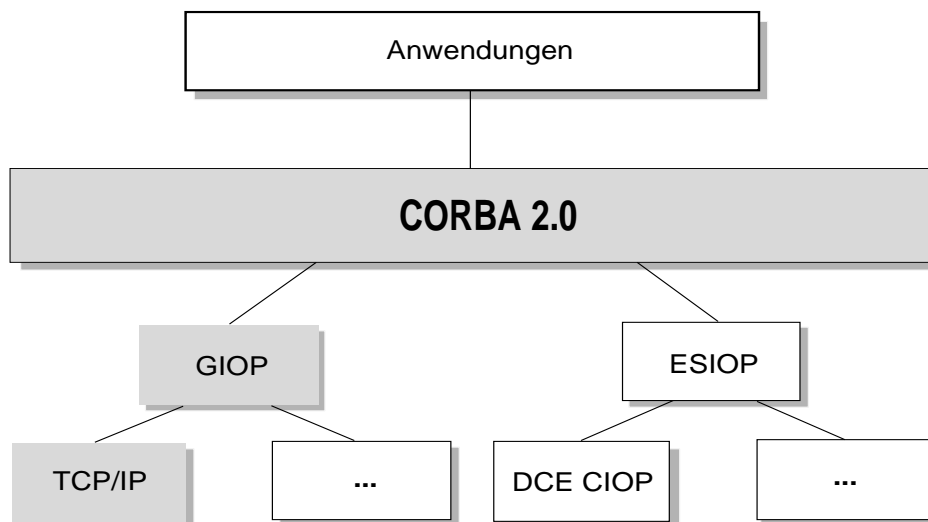


Abb. 4.2 Zusammenhang der verschiedenen Interoperabilitäts-Protokolle, grau dargestellte Elemente sind für CORBA 2.0 verpflichtend

Neben den für die CORBA 2.0 Konformität verbindlichen Protokollen GIOP und IIOP ist die Bereitstellung von ESIOPs möglich. Hierunter sind standardisierte Protokolle zu verstehen, die eine Interoperabilität von ORBs in speziellen Umgebungen ermöglichen. Ein Beispiel ist das DCE CIOP (DCE Common Inter-ORB Protocol), das von verschiedenen Herstellern vor der Einführung von GIOP und IIOP verwendet wurde. Es beschreibt die Abbildung von CORBA Methodenaufrufen auf DCE-Kommunikationsformate.

4.2 Ein Beispiel

An Hand eines einfachen Beispiels soll gezeigt werden, wie die Kommunikation mittels IIOP in der Praxis funktioniert. Auf Serverseite wird dazu Orbix 2.01 mit C++ verwendet, für die Programmierung des Clients wird VisiBroker for Java 1.2 verwendet. Als Beispiel dient der mit Orbix 2.01 mitgelieferte Grid Server, der ein zweidimensionales Array mit Zahlenwerten verwaltet. Die Werte können von einem Client gesetzt und gelesen werden. Die IDL-Schnittstelle sieht folgendermaßen aus:

```
// grid.idl

interface grid {
    readonly attribute short height; // Höhe des Arrays
    readonly attribute short width; // Breite des Arrays

    // weise Element [n,m] des Arrays den Wert value zu
    void set(in short n, in short m, in long value);

    // lies den Wert des Arrays an der Stelle [n,m]
    long get(in short n, in short m);
};
```

Modifizierung des Server-Hauptprogramms

Im folgenden wird darauf eingegangen, an welchen Stellen das Server-Hauptprogramm des Grid Servers modifiziert werden muß, um über IIOP kommunizieren zu können.

1. Als erster Schritt muß der Server beim Start eine interoperable Objektreferenz (IOR) von sich selbst ausgeben. Diese muß an einer dem Client zugänglichen Stelle abgelegt werden, hier z.B. in einer Datei.

```
int main() {

    // Erzeuge grid Objekt unter Verwendung der Implementierung grid_i
    grid_i myGrid(10,10);

    // öffne Datei, in welche die IOR geschrieben wird
    ofstream strm("/fzi/dbs/wipf/tmp/grid.ior");
    if (!strm) {
        cout << "Fehler: Konnte Datei grid.ior nicht erzeugen"
             << endl;
        return 0;
    }

    char * stringObj;
```

2. Der Server muß persistent, d.h. von Hand, gestartet werden. Bei persistenten Orbix Servern muß unbedingt darauf geachtet werden, daß der Servername korrekt gesetzt wird, bevor eine Interaktion mit Orbix erfolgt. Zum Beispiel sollte ein persistenter Server keine Objektreferenz von sich oder eines seiner Objekte ausgegeben werden, bevor der Servername mit Hilfe der Methode `CORBA::Orbix.setServerName()` festgelegt wurde.

```
try {
    CORBA::Orbix.setServerName("grid");
```

3. Nun muß die Objektreferenz des Servers in einen String umgewandelt werden. Dabei muß noch festgelegt werden, daß als Protokoll das IIOP verwendet wird. Per Standardeinstellung wird sonst das Orbix Protokoll verwendet:

```
stringObj = CORBA::string_dupl(
    myGrid._object_to_string(CORBA::IT_INTEROPERABLE_OR_KIND));
}
catch(CORBA::SystemException &sysEx) {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx ;
}
```

4. Die IOR kann nun als String in die Datei `grid.ior` geschrieben werden. Anschließend wird der Speicherplatz des Strings wieder freigegeben.

```
strm << stringObj << endl;
CORBA::string_free(stringObj);

...
```

Es folgen noch ein paar Codezeilen, die aber dem Code der nicht IIOP-fähigen Version des Grid Servers entsprechen, weshalb darauf nicht weiter eingegangen wird. An der Implementierung des Grid Servers muß nichts geändert werden, d.h. die Implementierung des Grid Servers ist unabhängig vom verwendeten Kommunikationsprotokoll.

Erstellen des Client Applets

Es folgen nun die wichtigsten Programmausschnitte eines IIOP-fähigen Clients, der als Java Applet unter Verwendung von VisiBroker for Java 1.2 realisiert wird.

1. Das Client Applet muß nun als erstes die IOR aus der Datei `grid.ior` lesen. Bei Applets geht dies am einfachsten über einen Web-Server und eine URL Verbindung. Die Datei `grid.ior` muß dabei vom Web-Server zur Verfügung gestellt werden. In diesem Beispiel wird davon ausgegangen, daß das Client Applet in der HTML Seite `gridiiop.html` verankert ist, diese Seite von einem Web-Server über die URL `http://delhi.fzi.de/gridiiop.html` geladen werden kann und die Datei `grid.ior` über die URL `http://delhi.fzi.de/tmp/grid.ior` erreichbar ist.

...

```
public void init() {
    String IOR;

    try {
        // erzeuge URL für die Datei grid.ior
        URL url = new URL(this.getDocumentBase(), "tmp/grid.ior");
        // öffne URL Verbindung
        URLConnection connection = url.openConnection();
        // erzeuge Eingabe-Stream auf geöffneter URL Verbindung
        DataInputStream in =
            new DataInputStream(connection.getInputStream());
        // lies den Inhalt der Datei
        IOR = in.readLine();
        // schließe die Verbindung
        in.close();
    }
    catch (Exception e) {
```

```

        System.out.println("Error: " + e);
        return;
    }

```

2. Aus der Stringform der IOR muß nun wieder eine richtige Objektreferenz gemacht werden und diese mit der Methode `narrow()` auf den lokalen Adressraum abgebildet werden:

```

try {
    // initialisiere den ORB
    CORBA.ORB orb = CORBA.ORB.init(this);
    // wandle String in Objektreferenz um
    CORBA.Object obj = orb.string_to_object(IOR);
    // Abbildung auf lokalen Adressraum
    myGrid = IDL_GLOBAL.grid_var.narrow(obj);
}
catch(CORBA.SystemException e) {
    System.out.println("Fehler beim Aufruf von narrow()");
    System.out.println(e);
    return;
}

```

Mit Hilfe dieser Objektreferenz können nun wie gewohnt die Methoden des Servers aufgerufen werden, wie folgendes Beispiel zeigt:

```

try {
    w = myGrid.width();
    h = myGrid.height();
}

```

Probleme und Lösungen

Das größte Problem bei der Kommunikation unterschiedlicher ORBs über IIOP besteht sicher darin, die interoperable Objektreferenz zwischen Server und Client auszutauschen. Das Schreiben in eine Datei ist nicht sehr komfortabel und nur in Verbindung mit einem Web-Server ein gangbarer Weg. Besser wäre es, die IOR bei einem CORBA Name Service zu registrieren, bei dem der Client die gewünschte Referenz abfragen kann. Bei unterschiedlichen ORBs stellt sich aber wieder die Frage, wie die initiale Referenz zu diesem Namensdienst ermittelt wird. Die OMG ist deshalb bestrebt, eine Spezifikation für einen interoperablen Namensdienst festzulegen. Die Firmen IBM, Netscape, Oracle, SunSoft und Visigenic haben bereits ein entsprechendes Dokument veröffentlicht [OMG97a].

Ein weiteres Problem ist darin zu sehen, daß Server bei Verwendung von Orbix 2.01 nicht bei Bedarf über den Orbix-Dämon gestartet werden können, sondern manuell aufgerufen werden müssen. Ab der Version 2.1 von Orbix soll auch die Aktivierung von Servern über IIOP möglich sein, da man einen speziellen IIOP Port festlegen kann. IIOP Anfragen werden vom Orbix-Dämon an diesem Port aufgenommen und angeforderte Server bei Bedarf gestartet. Tests mit Orbix 2.2 MT haben gezeigt, daß dies im Prinzip funktioniert. Der oben erwähnte Grid Server wird bei Aufruf über IIOP gestartet, durch einen offensichtlichen Fehler in der Multithreaded Version von Orbix 2.2 werden allerdings falsche Ergebnisse vom Server geliefert. Die Single-threaded Version funktioniert dagegen korrekt.

5 Einsatz im Rahmen von WWW-UIS

Im Rahmen des Forschungsvorhabens GLOBUS des Umweltministeriums Baden-Württemberg ist ein ressortübergreifendes Umweltinformationssystem auf der Basis des World Wide Web entwickelt worden. Das so benannte *WWW-UIS* stellt zahlreiche umweltrelevante Daten über eine Schnittstelle zu einem Datenbanksystem bereit [Krau96].

An Hand von Szenarien, die dem WWW-UIS entnommen wurden, sollten mit den am FZI zur Verfügung stehenden Java ORBs kleine Beispielanwendungen entworfen werden. Dabei wurden OrbixWeb 1.0 von IONA und VisiBroker for Java 1.2 von Visigenic eingesetzt.

5.1 Datenbankzugriff mit OrbixWeb 1.0

Im Rahmen von Umweltinformationssystemen werden in großem Umfang Zugriffe auf Datenbanken benötigt. Die CORBA Technologie eignet sich dabei sehr gut, um solche Datenbankzugriffe zu kapseln, wodurch eine Unabhängigkeit von dem darunterliegenden Datenbanksystem erreicht wird. Im Rahmen des WWW-UIS besteht ein besonderes Interesse darin, Datenbankabfragen über das World Wide Web möglich zu machen. Mit Hilfe von OrbixWeb lassen sich Java Applets programmieren, die mit Orbix in C++ implementierte Datenbankzugriffe aufrufen können.

Als einfaches Beispiel diente ein bestehender Orbix Server, der eine Anfrage nach einem Benutzer an eine Oracle Datenbank stellt und die zu diesem Benutzer gefundenen Informationen zurückliefert. Mit OrbixWeb wurde dazu ein Applet entworfen, das die Eingabe eines Benutzernamens erlaubt und die gefundenen Anfrageergebnisse in einem eigenen Fenster darstellt. Die Kombination von Java Client und C++ Server war dabei problemlos möglich.

In einem weiteren Beispiel sollten Datenbankzugriffe über einen dynamischen SQL Server, der am FZI entwickelt wurde, ausprobiert werden. Dieser Server ist sehr flexibel und ermöglicht es, beliebige Anfragen an eine Oracle Datenbank zu stellen. Im Gegensatz dazu war im ersten Beispiel nur eine fest vorgegebene Anfrage möglich.

Einem bestehenden C++ Client wurde ein Applet nachempfunden, das es ermöglichen sollte, beliebige SQL Anfragen abzusetzen. Leider war in diesem Falle die Zusammenarbeit von OrbixWeb und Orbix nicht möglich, da der OrbixWeb Client keine Verbindung zu dem SQL Server aufnehmen konnte. Es konnte nicht geklärt werden, ob die Gründe dafür in der Struktur des SQL Servers liegen oder in der frühen und unausgereiften Version von OrbixWeb zu suchen sind. Ein identischer C++ Client funktionierte jedenfalls einwandfrei.

Bei der Implementierung zeigte sich, daß Orbix und OrbixWeb leider nicht immer zusammenarbeiten. So kann man OrbixWeb bei Verwendung des Orbix Protokolls erst ab der Version 2.01 zusammen mit Orbix 2.1 und 2.2 benutzen.

5.2 Ereignis-Monitor mit VisiBroker for Java 1.2

Eine wichtige Rolle in Umweltinformationssystemen spielen aktive Mechanismen [KK+97]. Als Beispiel sei eine Grenzwert-Datenbank genannt. Beim Überschreiten von vorgegebenen Grenzwerten müssen bestimmte Aktionen ausgeführt werden, wie z.B. das Auslösen eines Ozon-Alarms. Für einen Client gibt es zwei Möglichkeiten, das Überschreiten von Grenzwerten zu erkennen. Zum einen kann er periodisch die aktuellen Werte abfragen, was aber sehr kostspielig ist. Oft werden Werte umsonst abgefragt oder das Prüfintervall wurde so groß

gewählt, daß Überschreitungen zu spät erkannt werden. Hier bietet sich ein aktiver Mechanismus an, der die Überschreitung eines Grenzwerts erkennt und den Client umgehend benachrichtigt. Dazu registriert sich der Client bei einem Server, der ihm asynchron Nachrichten schickt, d.h. Client und Server tauschen ihre Rollen. Diese Vorgehensweise minimiert die erforderliche Netzwerkkommunikation, da dem Client die wiederkehrende Anfrage nach eventuell vorhandenen Nachrichten erspart wird. Sobald ein Ereignis entdeckt wird, erfolgt unaufgefordert die Benachrichtigung des Clients.

Im Rahmen des GLOBUS III Projektes wurde in Zusammenarbeit mit dem Institut für Kernenergetik und Energiesysteme (IKE) der Universität Stuttgart eine Anwendung zur Integration und Kombination von Simulations- und Datenbankdiensten mit CORBA entwickelt [KoSc96]. Da in dieser Anwendung umfangreiche Datenbankzugriffe und Simulationsrechnungen stattfinden und das System längere Zeit für den Benutzer unsichtbar beschäftigt ist, wurde eine Komponente benötigt, um den Ablauf der Anwendung zu visualisieren. Die Ausführung bestimmter Programmteile und Datenbankzugriffe sollte in einem Applet angezeigt werden, sobald die entsprechende Stelle erreicht wurde.

Hier bietet sich die eingangs erwähnte Vorgehensweise an. Der Benutzer kann eine Web-Seite aufrufen und ein Applet laden, das sich bei einem Ereignis-Server anmeldet. Der Datenbank Wrapper und die Simulationsrechnungen erzeugen bei Abarbeitung bestimmter Programmteile entsprechende Meldungen, die an den Ereignis-Server geschickt werden. Dieser leitet die Meldungen an alle registrierten Client Applets weiter. Die Architektur für den Datenbankzugriff dieser Anwendung zeigt Abb. 5.1.

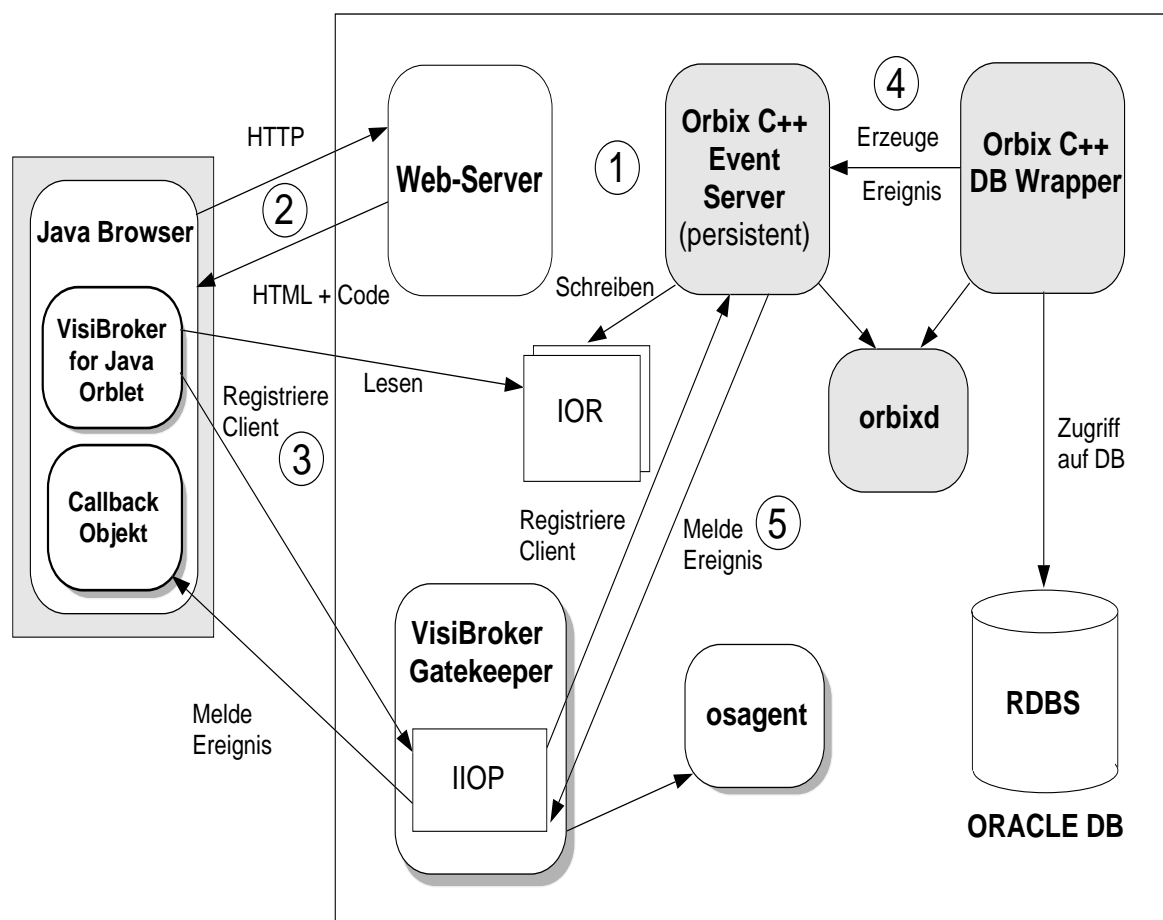


Abb. 5.1 Szenario und Architektur der Ereignis-Monitor Anwendung

Auf Client-Seite wurde VisiBroker for Java eingesetzt, da dies zum Zeitpunkt der Implementierung der einzige Java ORB war, der über ein server-seitiges Mapping verfügte und das IIOP unterstützte. Um eine leichte Integration in die mit Orbix programmierte Demonstrationsanwendung zu gewährleisten, wurde der Ereignis-Server ebenfalls mit Orbix in C++ implementiert. Da es sich um ORBs unterschiedlicher Hersteller handelt, erfolgt die Kommunikation über das IIOP. Durch die Verwendung des IIOP sind alle Komponenten sehr flexibel einsetzbar.

Die IDL Schnittstellen-Beschreibung sieht folgendermaßen aus:

```
// event.idl

interface Callback;

// Schnittstelle fuer den Client
interface EventHandler {

    // Erzeugen von Meldungen
    void CreateMessage (in string message);

    // Registrieren eines Clients
    void RegisterClient (in Callback obj);

    // Registrierung für einen Client aufheben
    void RemoveClient (in Callback obj);
};

// Schnittstelle für den Server zum asynchronen Rückruf
interface Callback {

    // Meldungen an den Client schicken
    oneway void NewMessage (in string message);
};
```

Die Schnittstelle `EventHandler` stellt Methoden zur Verfügung um Clients beim Ereignis-Server zu registrieren (`RegisterClient`) bzw. deren Registrierung aufzuheben (`RemoveClient`). Übergeben wird dabei die Referenz auf ein `Callback`-Objekt, das vom Ereignis-Server aufgerufen werden kann. Mittels der Methode `CreateMessage` kann ein Client Meldungen an den Ereignis-Server schicken.

Die Schnittstelle `Callback` ermöglicht es dem Ereignis-Server, über die Methode `NewMessage` Meldungen an registrierte Clients zu schicken. Dies bedeutet, daß Client und Server die Rollen tauschen.

Implementierung des Servers

Der Server wurde mit Orbix in C++ implementiert. Die Objektreferenzen registrierter Clients werden in dem Array `ConnectedClients` verwaltet, dessen Größe auf 16 Einträge beschränkt ist, was für Demonstrationszwecke völlig ausreicht. Die entsprechende Initialisierung erfolgt im Konstruktur.

```
// Konstruktor
EventHandler_i::EventHandler_i() {
    max_clients = 16;
    count = 0;
    for (int i=0; i<max_clients; i++) {
        ConnectedClients[i] = Callback::_nil();
    }
```

```

}
}

```

Zum registrieren von Clients wird die übergebene Objektreferenz auf das client-seitige Callback-Objekt in das Array `ConnectedClients` eingetragen.

```

void EventHandler_i::RegisterClient (Callback_ptr obj,
                                     CORBA::Environment &IT_env) {

    ConnectedClients[count] = Callback::_duplicate(obj);
    cout << "Client " << count << " registriert" << endl;
    count++;
    count %= max_clients;
}

```

Die Methode zum Erzeugen von Meldungen entnimmt dem Array `ConnectedClients` die Objektreferenzen aller registrierten Clients und schickt diesen die übergebene Meldung. Dazu wird die Methode `NewMessage` des Callback-Objektes des Clients aufgerufen.

```

void EventHandler_i::CreateMessage (const char * message,
                                     CORBA::Environment &IT_env) {

    Callback_ptr cbPtr;
    int connected = 0;
    for (int i=0; i<max_clients; i++) {
        if (!CORBA::is_nil(ConnectedClients[i])) {
            cbPtr = Callback::_narrow(ConnectedClients[i]);
            cbPtr->NewMessage(message);
            connected = 1;
        }
    }
    if (!connected) {
        cout << "Keine Clients registriert" << endl;
    }
}

```

Um die Registrierung eines Clients aufzuheben, z.B. wenn der Client beendet wird, muß lediglich die entsprechende Objekt-Referenz aus dem Array `ConnectedClients` entfernt werden.

```

void EventHandler_i::RemoveClient (Callback_ptr obj,
                                    CORBA::Environment &IT_env) {

    char* stringObj;
    char* tmp;
    int removed = 0;
    stringObj = CORBA::Orbix.object_to_string(obj);
    for (int i=0; i<max_clients; i++) {
        if (!CORBA::is_nil(ConnectedClients[i])) {
            tmp = CORBA::Orbix.object_to_string(ConnectedClients[i]);
            if (strcmp(stringObj, tmp) == 0) {
                ConnectedClients[i] = Callback::_nil();
                cout << "Client " << i << " entfernt" << endl;
                removed = 1;
                break;
            }
        }
    }
    if (!removed) {
        cout << "Nichts zu tun" << endl;
    }
}

```

Implementierung des Clients

Der Client wurde als Applet unter Verwendung von VisiBroker for Java programmiert. Neben dem eigentlichen Applet, muß der Client ein Callback-Objekt definieren, das vom Server angesprochen werden kann. Hierzu dient die Klasse `LocalImpl`, in der die Methode `NewMessage` der `CallBack` Schnittstelle implementiert wird.

```
class LocalImpl extends IDL_GLOBAL._sk_CallBack
implements MonitorCallback {

    Monitor eventMonitor;
    IDL_GLOBAL.EventHandler myEvent;

    // Konstruktor
    public LocalImpl(IDL_GLOBAL.EventHandler myEvent) {
        super();
        this.myEvent = myEvent;
        eventMonitor = new Monitor("Ereignis Monitor", this);
        eventMonitor.setText("Lokale Implementierung verfügbar\n");
    }

    // Methode zum Anzeigen von Ereignismeldungen
    public void NewMessage(String message) {
        System.out.println("Ereignis empfangen : " + message);
        eventMonitor.appendText(message + "\n");
    }
}
```

Das Applet muß als erstes die interoperable Objektreferenz (IOR) aus der Datei `eventiiop.ref` lesen. Dies kann einfach über eine URL Verbindung zum Web-Server erfolgen und entspricht der in Kapitel 4.2 beschriebenen Vorgehensweise.

```
public class EventApplet extends Applet {
    ...

    public void init() {
        String IOR;

        try {
            URL url = new URL(this.getDocumentBase(), "tmp/eventiiop.ref");
            URLConnection connection = url.openConnection();
            DataInputStream in = new DataInputStream(
                connection.getInputStream());

            IOR = in.readLine();
            in.close();
        }
        catch (Exception e) {
            System.out.println("Exception: " + e);
            return;
        }
    }
}
```

Anschließend werden ORB und BOA initialisiert. Aus der Stringform der IOR wird eine richtige Objektreferenz gemacht und diese mittels der Methode `narrow()` auf den Adressraum des Clients abgebildet:

```
try {
    orb = CORBA.ORB.init(this);
    boa = orb.BOA_init();
    CORBA.Object obj = orb.string_to_object(IOR);
    myEvent = IDL_GLOBAL.EventHandler_var.narrow(obj);
}
```

```

    }
    catch(CORBA.SystemException e) {
        System.out.println("Exception bei narrow(): " + e);
        return;
    }
}

```

Der Client erzeugt eine Instanz des Callback-Objekts und registriert diese beim Server. Zum Test wird eine Meldung an den Server geschickt, die umgehend an alle registrierten Clients gesandt wird:

```

try {
    LocalImpl loc = new LocalImpl(myEvent);
    boa.obj_is_ready(loc);
    System.out.println("Lokaler Ereignis Handler verfuegbar");
    myEvent.RegisterClient(loc);
    myEvent.CreateMessage("Client Applet auf "
        + host
        + " gestartet und registriert!");
}
catch (CORBA.SystemException ex) {
    System.out.println(
        "Exception beim Aufruf des Event Servers");
    System.out.println(ex.toString());
}

```

Auf die Details des Monitor-Objekts, das ein Fenster zum Anzeigen von Meldungen realisiert, wird an dieser Stelle nicht näher eingegangen. Die vollständigen Quelltexte der Ereignis-Monitor Anwendung können Anhang C entnommen werden.

Ablauf der Anwendung

Um sich den Ablauf der Anwendung klarzumachen, betrachtet man am besten Abb. 5.1. Im ersten Schritt (1) wird der Orbix Ereignis-Server persistent, d.h. von Hand, gestartet. Beim Start schreibt er eine interoperable Objektreferenz (IOR) von sich in eine Datei. Danach kann das Client Applet in einem Java-fähigen Browser aufgerufen werden (2). Die VisiBroker for Java Klassen werden automatisch nachgeladen und ein Callback-Objekt instanziiert. Nachdem das Applet die IOR aus der Datei gelesen hat, kann es sich mit dem Ereignis-Server verbinden und registriert dort die Objektreferenz auf das client-seitige Callback-Objekt (3). Eintreffende Ereignisse werden vom Ereignis-Server erkannt (4) und an alle registrierten Clients unter Verwendung der übergebenen Objektreferenzen geschickt (5). Die eintreffenden Nachrichten werden in dem Monitor-Fenster angezeigt.

6 Erfahrungen bei der Realisierung

Im folgenden werden die Erfahrungen, Probleme und mögliche Lösungen beschrieben, die sich bei der Implementierung der Beispielprogramme ergaben. Dieses Kapitel vermittelt auch Tips und Tricks, die im Umgang mit Java ORBs und allgemein bei der Programmierung mit Java hilfreich sein können.

6.1 Erfahrungen mit Java

6.1.1 Die Umgebungsvariable CLASSPATH

Als Fehlerquelle Nummer eins erwies sich die von Java benötigte Umgebungsvariable `CLASSPATH`. Sie bestimmt den Ort, von dem Java seine Klassendateien lädt. Ist `CLASSPATH` nicht auf die richtigen Verzeichnisse gesetzt, so kann dies zu den absonderlichsten Fehlermeldungen führen, die aber nicht auf eine solche Ursache schließen lassen.

Besonders störend an dieser Umgebungsvariable ist, daß sie sowohl von Java, als auch vom Netscape Navigator 2.x benutzt wird. Dies führt dazu, daß sich der Java Applet Viewer und Navigator 2.x nicht vertragen. Die vom JDK 1.0.2 verwendeten Klassen in `classes.zip` sind offenbar nicht kompatibel mit den von Netscape 2.x verwendeten Klassen in `moz2_x.zip`. Hier wäre es sicher besser, wenn Netscape eine eigene Umgebungsvariable, wie z.B. `MOZ_CLASSPATH`, benutzen würde. Ab der Version 3.0 des Navigators tritt dieses Problem nicht mehr auf, hier wird `CLASSPATH` nicht mehr für die virtuelle Java-Maschine im Browser benötigt.

6.1.2 Applet Viewer contra Netscape Navigator

Das Testen von Applets im Netscape Navigator erwies sich als recht umständlich. Es gibt keine Möglichkeit, die virtuelle Java-Maschine des Browsers zu beenden, außer man beendet den Browser selbst. Wenn man Änderungen an einem Applet vorgenommen hat, werden diese durch ein Reload im Browser nicht sichtbar.

Hier bietet sich der Appletviewer des JDK an, der wesentlich schneller ist und auch weniger Zeit benötigt, um geladen zu werden. Dieser läßt allerdings keinen Aufruf einer URL zu. Grundsätzlich kann man auf den Test im Netscape Browser nicht verzichten, dort sollen die Anwendungen schließlich auch ihren Einsatz finden.

Im Laufe der Implementierungsarbeiten zeigte sich, daß ein Applet, das im Appletviewer fehlerfrei läuft, seinen Dienst im Netscape Navigator noch lange nicht verrichten muß. Der gängigste Fehler ist dabei, daß die benötigte Klassen nicht gefunden werden können.

Mögliche Ursachen sind:

- `CLASSPATH` ist falsch gesetzt, oftmals wurde ein Verzeichnis falsch geschrieben oder auch das aktuelle Verzeichnis vergessen (nur mit Navigator 2.x)
- die benötigten Klassen müssen für alle lesbar sein, sonst kann sie der Browser nicht laden
- in der HTML-Seite wurde unter `CODEBASE` ein falsches Verzeichnis angegeben, leider läßt sich hier auch nur ein Verzeichnis angeben

6.1.3 Debugging von Applets im Netscape Navigator

Der Netscape Navigator bietet einige versteckte Funktionen in der Java Console, mit denen sich Details zu geladenen Applets anzeigen lassen. Durch Positionieren des Mauszeigers in der Java Console und anschließendes Drücken der Ziffer 9 läßt sich der Applet Debug Level auf 9 setzen. Unter UNIX (und dem entsprechenden Window Manager) darf man im Gegensatz zu Windows NT zuvor nicht mit der Maus in das Fenster klicken, sonst funktioniert es nicht. Es ertönt lediglich ein Piepen aus dem Lautsprecher.

Durch das Setzen des Debug Levels werden von nun an alle vom Browser geladenen Java Klassen in der Java Konsole angezeigt. Entsprechend läßt sich der Applet Debug Level mit den restlichen Ziffern setzen. In Verbindung mit Java ORBs kann es interessant sein, genau zu wissen, welche Klassen in den Browser geladen werden, um mit diesen Informationen ein ZIP Archiv mit allen benötigten Klassen zu erstellen (siehe hierzu auch Kapitel 6.2.2).

Zusätzlich zu den Zifferntasten bewirkt die Taste `f` den Aufruf der `finalize()` Methode von Objekten, die sich in der Finalization Queue befinden. Die Taste `g` löst eine Garbage Collection aus. Durch Drücken der Taste `d` werden zusätzliche Informationen über das geladene Applet ausgegeben.

Der Netscape Navigator 4 verfügt noch über weitere Debug Optionen, die mit der Taste `h` angezeigt werden können. Die folgende Tabelle stellt alle in der jeweiligen Version verfügbaren Tasten und die zugehörigen Aktionen dar.

Taste	Aktion	Version
0 - 9	setzt den Applet Debug Level	3, 4
c	löscht den Inhalt der Java Konsole	4
d	gibt Informationen zu geladenen Applets aus	3, 4
f	Aufruf von <code>finalize()</code>	3, 4
g	führt Garbage Collection aus	3, 4
h	gibt Hilfestellung zu allen Tasten aus	4
l	speichert alle geladenen Java Klassen in einem Verzeichnis	4
m	gibt Speicherverbrauch in der Java Konsole aus	4
q	schließt die Java Konsole	4
s	gibt Speicherverbrauch in die Datei <i>memory.out</i> aus	4
t	schreibt Informationen über Threads in die Datei <i>memory.out</i>	4
x	schreibt Speicherbelegung in die Datei <i>memory.out</i>	4
X	schreibt detaillierte Speicherbelegung in die Datei <i>memory.out</i>	4

Tabelle 2: Debug Optionen in der Java Konsole des Netscape Navigators

6.2 Erfahrungen mit Java ORBs

6.2.1 Bereitstellen der benötigten Java Klassen

Damit man ein Applet in einen Java Browser laden kann, muß man dem Browser mitteilen, wo sich die benötigten Java Klassen befinden. Entweder sucht der Browser die Klassen in dem Verzeichnis, von dem die HTML Seite geladen wurde, oder an Hand des CODEBASE Eintrags im Applet Tag. Mit CODEBASE kann man aber nur ein Verzeichnis angeben. Bei Java/CORBA Anwendungen befinden sich Applet und ORB Klassen („Orblet“) aber in verschiedenen Verzeichnissen.

Eine Möglichkeit besteht darin, das Applet und seine Klassen in das Klassenverzeichnis des Java ORBs zu kopieren. Dies ist allerdings nicht ideal, da alle Entwickler Zugang haben müßten, was normalerweise nicht der Fall ist.

Eine bessere Lösung bietet das Betriebssystem UNIX an. Hier hat man die Möglichkeit, aus dem Verzeichnis des Applets symbolische Links auf die Klassenverzeichnisse des Java ORBs einzurichten. Eine weitere Möglichkeit, die zudem den Ladevorgang der Klassen beschleunigt und auch unter Windows NT verfügbar ist, wird im folgenden Kapitel beschrieben.

6.2.2 Beschleunigung des Ladevorgangs der ORB Klassen

Java ORBs bestehen aus vielen kleinen Java Klassen. Im Falle von VisiBroker for Java müssen über sechzig Klassen in den Browser geladen werden, wobei für jede Klasse ein Verbindung zum Web-Server hergestellt werden muß. Deshalb dauert es auch sehr lange, bis der komplette ORB geladen ist. Im JDK 1.1 gibt es aus diesem Grund die Möglichkeit, die benötigten Klassen und andere Ressourcen in eine Java Archive (JAR) zu packen und dies mit einer einzigen Verbindung zu laden. Das JAR Format ermöglicht es zudem, den Inhalt des Archivs zu komprimieren, wodurch sich die Ladezeit zusätzlich verkürzt. JAR Archive werden aber erst ab dem Netscape Navigator 4.0 unterstützt.

Bereits mit dem Netscape Navigator 3.0 ist es möglich, Java Klassen in ZIP Dateien zu packen und dann in einem Stück in den Browser zu laden. Das ZIP Format ist weit verbreitet und auf vielen Plattformen verfügbar. Zur Verwendung mit dem Navigator 3.0 darf das ZIP Archiv allerdings nicht komprimiert werden und muß die von den Java Packages benötigte Verzeichnisstruktur beinhalten. Das Erstellen eines ZIP Archivs erfolgt mit folgendem Aufruf:

```
prompt> zip -r0 myclasses.zip *
```

Dadurch werden alle Dateien im aktuellen Verzeichnis und den zugehörigen Unterverzeichnissen in die Datei myclasses.zip gepackt (ohne Komprimierung). Der Aufruf des Applets HelloWorld in der HTML Seite erfolgt über folgenden HTML Code:

```
<applet code=HelloWorld.class width=50 height=50  
  archive="myclasses.zip">  
</applet>
```

Der Internet Explorer von Microsoft unterstützt keine ZIP Dateien, sondern nur die hauseigenen Cabinet Dateien (Dateiendung CAB). Anstelle des ARCHIVE Tags werden die Angaben über das Cabinet Archiv im APPLET Tag als Parameter übergeben. Für das CAB Archiv hello.cab sieht der HTML Code folgendermaßen aus:

```
<applet code=HelloWorld.class width=50 height=50
  <param name=codebase value="http://fzi.de">
  <param name=cabbase value="hello.cab">
</applet>
```

Die beiden Varianten lassen sich auch folgendermaßen kombinieren:

```
<applet code=HelloWorld.class width=50 height=50
  archive="myclasses.zip">
  <param name=codebase value="http://fzi.de">
  <param name=cabbase value="hello.cab">
</applet>
```

Browser, die den ARCHIVE Tag nicht kennen und mit CAB Dateien nichts anfangen können (wie z.B. der Netscape Navigator 2.x), ignorieren diese Angaben und laden die Java Klassen nacheinander wie bisher.

Da nicht alle Klassen eines Java ORBs in den Browser geladen werden, sollte man nur die benötigten Klassen in ein Archiv stecken. Welche Klassen benötigt werden, kann man ermitteln, indem man in der Java Konsole den Applet Debug Level durch Drücken der Zifferntaste 9 setzt. Dadurch werden alle von nun an geladenen Java Klassen angezeigt und man kann sich so ein speziell zugeschnittenes Archiv erstellen. Mit Netscape 4 kann man durch Eingabe der Taste 1 sogar erreichen, daß alle geladen Klassen in einem lokalen Verzeichnis abgelegt werden. Diese kann man dann ganz einfach in eine ZIP- oder JAR Datei packen.

IONA bietet für OrbixWeb auf seiner Homepage bereits entsprechende Archive an, welche die für die jeweilige Version von OrbixWeb benötigten Klassen enthalten.

6.2.3 VisiBroker for Java im Netscape Navigator 4 (Communicator)

Die Firma Netscape hat in ihren Navigator 4 das Laufzeitsystem von VisiBroker for Java integriert. Erste Erfahrungen konnten mit der Beta Version 4.01b6 unter Solaris gewonnen werden. Netscape bezeichnet das Produkt als Internet Service Broker (ISB) for Java, das identisch mit VisiBroker for Java 2.5 sein soll. Bei ersten Tests stellte sich aber heraus, daß Netscape sowohl in der Beta 6 unter Solaris, als auch in der Endversion 4.01 unter NT das Package `com.visigenic.vbroker.Gatekeeper` entfernt hat, weshalb es keine Unterstützung für den Gatekeeper von VisiBroker gibt. Objekte lassen sich dadurch nicht über den Gatekeeper, sondern nur über IIOP mittels einer interoperablen Objektreferenz lokalisieren. Dies funktioniert allerdings sehr gut. Über IIOP lassen sich problemlos CORBA 2.0 konforme ORBs ansprechen, was am Beispiel von Orbix erfolgreich getestet werden konnte. Probleme bereiten allerdings Callbacks über IIOP, dies scheint noch nicht richtig zu funktionieren. Obwohl die Demoanwendung aus Kapitel 5.2 mit Netscape 3.0 und VisiBroker for Java 2.5 funktioniert, ist sie im Navigator 4.01b6 nicht lauffähig.

Da alle benötigten ORB Klassen bereits in den Browser eingebaut sind und nicht über das Netz nachgeladen werden müssen, läßt sich diesbezüglich eine große Geschwindigkeitssteigerung erreichen. Die bisher unzumutbar langen Ladezeiten entfallen, es müssen lediglich die für das Applet benötigten Klassen übertragen werden. Diese lassen sich in ein Java Archiv (JAR) packen und in einem Ladevorgang übermitteln.

7 Zusammenfassung und Ausblick

Die Kombination von CORBA und Java bringt für beide Technologien Vorteile. Durch das Zusammenspiel gut zueinander passender Konzepte ergeben sich vielfältige neue Möglichkeiten für Anwendungen im Intra- und Internet. Java ORBs sind speziell als Infrastruktur für Netzwerk-Computer und für Anwendungen in heterogenen Intranets bestens geeignet. Entsprechende Bandbreiten vorausgesetzt, sind auch komplexere Anwendungen im Internet möglich. Auf Server-Seite hat der Programmierer alle Freiheiten und kann für eine geforderte Aufgabe die am besten geeignete Plattform und Programmiersprache auswählen. Clients wiederum lassen sich durch die Portabilität von Java nahezu überall einsetzen. Mit Java können sehr einfach grafische Benutzeroberflächen programmiert werden, weshalb sich Java ORBs vor allem auf Clientseite anbieten.

Da Java eine interpretierte Sprache ist, können Java ORBs mit der Performanz von C++ ORBs nicht mithalten. Dies wird sich mit fortlaufender Entwicklung der Java Virtual Machine und der Verfügbarkeit von Just In Time (JIT) Compilern aber verbessern. Die virtuelle Java Maschine des neuen JDK 1.1 zeigt bereits ein wesentlich besseres Laufzeitverhalten, als sie mit dem JDK 1.02 erzielbar ist. Das Problem der langen Ladezeiten der vielen Klassen eines Java ORBs läßt sich mit ZIP- bzw. JAR-Archiven beheben. Beim Netscape Navigator 4 sind die Java ORB Klassen sogar schon in den Browser eingebaut.

Konkrete Anwendungen mit Java ORBs

Java ORBs sind wie die Sprache Java selbst noch sehr neu, weshalb konkrete Anwendungen bisher sehr selten sind. Dennoch gibt es ein starkes Interesse an dieser Technologie. Dies zeigt sich darin, daß VisiBroker for Java inzwischen von mehreren namhaften Firmen lizenziert wurde, die diesen Java ORB in ihre Produkte integrieren wollen.

Neben der bereits erwähnten Integration in Netscape Produkte lizenziert die Firma Oracle VisiBroker for Java für den Einsatz in ihren Netzwerk Computern (NC). Borland wird den Java ORB von Visigenic in eine Client/Server-Variante seiner Java Entwicklungsumgebung JBuilder integrieren. Weitere Lizenznehmer sind Sybase, Novell und Circom.

Das derzeit größte CORBA/Java Projekt, das sich im Rahmen einer Milliarden-Investition bewegt, wird im Auftrag der Hongkong Telekom mit OrbixWeb entwickelt und hat die Erstellung eines Interactive Multimedia Service (IMS) zum Ziel [Ion96c]. Diese Dienste sollen u.a. Home Shopping, Electronic Banking und Video-On-Demand ermöglichen. Für das Management der Kommunikations-Infrastruktur wird Orbix zur Bereitstellung von Medien-Servern verwendet. Zur Interaktion mit dem Benutzer kommt OrbixWeb in Set-Top-Boxen zum Einsatz.

Der Blick nach vorn

Die OMG ist bereits dabei, das Konzept für CORBA 3.0 zu entwerfen. Eine wesentliche Zielsetzung ist dabei, die Entwicklung von CORBA Anwendungen zu vereinfachen. Dazu beitragen sollen eine Script-Sprache und ein Komponenten-Modell. Vorschläge für ein Komponenten-Modell wurden bereits von IBM, Oracle, Sun und Netscape eingebracht [OMG97b] und basieren auf dem Konzept der sogenannten *Java Beans*, der von JavaSoft entwickelten Komponenten-Architektur für Java. Die Verbindung der einzelnen Komponenten soll mittels einer Script-Sprache erfolgen. Hier sind verschiedene Sprachen im Gespräch, u.a Javascript von Netscape.

Nachdem Java bisher vor allem auf Client-Seite eingesetzt wird, soll es nach dem Willen von JavaSoft nun mehr und mehr auch auf Server-Seite eingesetzt werden. Mit der Java Enterprise API steht dem Entwickler ein umfangreiches Paket zur Entwicklung von Server-Anwendungen zur Verfügung. Zentraler Bestandteil sind dabei die Enterprise JavaBeans, eine Erweiterung des JavaBeans Komponenten-Modells um nicht-visuelle Komponenten, sogenannte Business-Objekte ohne eigene Oberfläche. Weitere Bestandteile sind u.a. die Java Transaction Services (JTS) und das Java Naming und Directory Interface (JNDI). Beides basiert auf den von der OMG seit längerem spezifizierten CORBA Transaction bzw. Naming Services. Hinzu kommt neben Java RMI auch das in Kapitel 3.8 besprochene Java IDL, das die Verbindung mit CORBA Diensten gewährleistet.

Auf der Basis von CORBA entwickelte Anwendungen haben sich im industriellen Einsatz bereits bewährt. Java ORBs bieten ein großes Potential, von dem Entwickler und Benutzer gleichermaßen profitieren werden. Geringere Kosten bei der Softwareerstellung bei gleichzeitiger Reduzierung von Entwicklungs- und Wartungszeiten verdeutlichen dies [KoHW97].

8 Literaturverzeichnis

- [Bros96] Gerald Brose. Informationen zu JacORB, URL <http://www.inf.fu-berlin.de:80/~brose/jacorb/>, 1996.
- [Bro97a] Gerald Brose. JacORB - a Java Object Request Broker. *Technical Report B-97-02*, Freie Universität Berlin, April 1997.
- [Bro97b] Gerald Brose. JacORB: Implementation and Design of a Java ORB. In *Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*. Chapman & Hall, September 1997.
- [BrBo97] Gerald Brose und Boris Bokowski. Ein Object Request Broker für Java. *Informatik/Informatique, Zeitschrift der schweizerischen Informatikorganisation*, (3):27–30, Juni 1997.
- [Chal96] Suresh Challa. BlackWidow - An ORB Designed For The Internet. *First Class*, 1:17,20, 1996.
- [EFVo96] Wolfgang Emmerich, Fabrizio Ferrandina und Andreas Vogel. Integration von Java mit CORBA: Portable Client/Server-Applikationen im Internet. *Object Spectrum*, 1996. URL <http://www.dstc.edu.au/AU/staff/andreas-vogel/papers/object-spectrum96/paper.html>.
- [Flan96] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., 1996.
- [Frie96] Ernest Friedman-Hill. Informationen zu JIDL, URL <http://herzberg.ca.sandia.gov/jidl/>, 1996.
- [HORB96] Hirano Satoshi (Electrotechnical Lab.). HORB Homepage, URL <http://ring.etl.go.jp/openlab/horb/>, 1996.
- [Hran97] Norbert Hranitzky. Spieglein, Spieglein an der Wand. *Java Spektrum*, 5(1):22–27, Januar 1997. Artikel zu Java RMI.
- [Ion96a] Iona Technologies. Informationen zu OrbixWeb, URL <http://www-usa.iona.com/Products/Orbix/OrbixWeb/index.html>, 1996.
- [Ion96b] Iona Technologies. Java, OrbixWeb and IIOP - A component-based architecture for internet development. *Orbix Journal*, 1996.
- [Ion96c] Iona Technologies. Press Release: Hongkong Telecom IMS Opts for Orbix and Java. URL <http://www.iona.com/Press/PR/HongKong.html>, August 1996.
- [Jade96] The Jade Web Pages, URL <http://iiop.ansa.co.uk:8080/~jade/>, 1996.
- [Jav97a] JavaSoft. Informationen zu Java IDL, URL <http://www.javasoft.com/products/jdk/idl/index.html>, 1997.
- [Jav97b] JavaSoft. Java Homepage, URL <http://www.javasoft.com>, 1997.
- [Jav97c] JavaSoft. Java-Based Distributed Computing - RMI and IIOP in Java, Juni 1997. URL <http://www.javasoft.com/pr/1997/june/statement970626-01.html>.
- [Joe96] Informationen zu Joe von SunSoft, URL <http://www.sun.com/solaris/neo/joe>, 1996.
- [JYLU96] JYLU - ILU for Java, URL <http://www-db.stanford.edu/~hassan/Jylu/>, 1996.

- [KK+97] Arne Koschel, Ralf Kramer, Günter von Bültzingsloewen, Thomas Bleibel, Petra Krumlinde, Sonja Schmuck und Christian Weinand. Configurable Active Functionality for CORBA. In *ECOOP'97 Workshop – CORBA: Implementation, Use and Evaluation*, Jyväskylä, Finnland, Juni 1997. URL <http://sirac.inrialpes.fr/~bellissa/wecoop97>.
- [KoHW97] Arne Koschel, Dirk Hoffmann und Matthias Wipf. Kombination von CORBA und Java für verteilte Client/Server-Anwendungen im Intranet/Internet. In *7. Kolloquium Softwareentwicklung - Methoden, Werkzeuge, Erfahrungen*, Esslingen, Deutschland, September 1997.
- [KoSc96] Arne Koschel und Martin Schöckle. Integration und Kombination von Simulationsdiensten und Datenbankzugriffsdiensten mit CORBA. Technischer Bericht, Ministerium für Umwelt und Verkehr Baden-Württemberg, 1996. Wissenschaftliche Berichte FZKA 5900, Projekt Globus III.
- [Krau96] Ralph Krause. Einsatz und Erprobung von Java in den Graphischen Diensten des WWW-UIS. Diplomarbeit, Universität Karlsruhe, Institut für Photogeometrie und Fernerkundung (IPF), August 1996.
- [Merk96] Bernhard Merkle. Der Kaffee ist fertig ... *Java Spektrum*, 4(6):29–34, November 1996. Bericht über Joe von SunSoft.
- [Merk97] Bernhard Merkle. Flugzeug-Buchungssystem mit Java und CORBA. *Java Spektrum*, S. 30–36, Juli 1997. Bericht zu OrbixWeb.
- [MSSSt96] Stefan Middendorf, Reiner Singer und Stefan Strobel. *Java Programmierhandbuch und Referenz*. Verlag dpunkt, 1996.
- [OMG95a] Object Management Group. The Common Object Request Broker Architecture and Specification, Revision 2.0, Juli 1995. Architecture.
- [OMG95b] Object Management Group. CORBAservices: Common Object Services Specification, März 1995. Event Services, Persistence.
- [OMG95c] Object Management Group. Common Facilities Architecture, Revision 4.0, November 1995. Rule Facility.
- [OMG97a] Object Management Group. Interoperable Naming Service, Juni 1997. Document orbos/97-06-03.
- [OMG97b] Object Management Group. CORBA Component Imperatives, Mai 1997. Document orbos/97-05-25.
- [OrHa97] Robert Orfali und Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley Computer Publishing, 1997.
- [Piet97] Stephen R. Pietrowicz. The Java Book Pages, URL <http://lightyear.ncsa.uiuc.edu/~srp/java/javabooks.html>, 1997.
- [PBC97] Informationen zu PowerBroker CORBAplus, Java Edition, URL http://www.expersoft.com/prod_ser/index.htm, 1997.
- [Stal95] Michael Stal. Der Zug rollt weiter. *iX*, (5), Mai 1995.
- [Sun96] Sun Microsystems. Informationen zu Spring-Java-IDL, URL <http://www.sun.com/tech/projects/spring/spring-java.html>, 1996.
- [VBrJ96] Visigenic Software Inc. Information zu VisiBroker for Java,

URL <http://www.visigenic.com>, 1996.

- [Vog96a] Andreas Vogel. Building Distributed Systems in Java. *Object Expert*, 1996.
<http://www.dstc.edu.au/AU/staff/andreas-vogel/papers/obj-exp96/paper.html>
- [Vog96b] Andreas Vogel. WWW and Java - Threat or Challenge to CORBA? *Spectrum Reports*, Mai 1996.
<http://www.dstc.edu.au/AU/staff/andreas-vogel/papers/mws96/paper.html>.
- [VoDu97] Andreas Vogel und Keith Duddy. *Java Programming with CORBA*. Wiley Computer Publishing, 1997.
- [WWW] W3C Konsortium. Homepage des World Wide Web Konsortiums,
URL <http://www.w3.org/pub/WWW/>, 1997.

Anhang A Hinweise zur Installation

A.1 VisiBroker for Java 1.2

Die Software für VisiBroker for Java 1.2 liegt als Datei `VisiJava_12_sol.tar.gz` vor. Die Lizenz erhält man per E-Mail von Visigenic Software und wird in die Datei `license.dat` geschrieben.

Die Installationsschritte im einzelnen

1. Festlegen des Verzeichnisses, in das VisiBroker for Java installiert werden soll, und Anlegen des Verzeichnisses, zum Beispiel:
`prompt> mkdir $HOME/VBJ-1.2`
2. Kopiere die gepackte Datei `VisiJava_12_sol.tar.gz` in dieses Verzeichnis:
`prompt> cp VisiJava_12_sol.tar.gz $HOME/VBJ-1.2`
3. Wechsle in dieses Verzeichnis und entpacke die Datei:
`prompt> cd $HOME/VBJ-1.2`
`prompt> gunzip VisiJava_12_sol.tar.gz`
4. Extrahiere den Inhalt der resultierenden Datei:
`prompt> tar -xvf VisiJava_12_sol.tar`
5. Richte ein Verzeichnis für die Lizenz ein:
`prompt> mkdir adm_delhi`
6. Kopiere die Lizenz in dieses Verzeichnis und richte die Umgebungsvariable `ORBELINE` entsprechend ein:
`prompt> cp license.dat $HOME/VBJ-1.2/adm_delhi`
`prompt> setenv ORBELINE $HOME/VBJ-1.2/adm_delhi`
7. Zusätzlich müssen noch zwei Umgebungsvariablen gesetzt werden:
`prompt> setenv CLASSPATH .:$HOME/VBJ-1.2/classes`
`prompt> setenv PATH $PATH:$HOME/VBJ-1.2/bin`
8. Mache das Verzeichnis `VBJ-1.2/classes` und alle untergeordneten Dateien und Verzeichnisse für alle lesbar:
`prompt> cd $HOME`
`prompt> chmod o+rX VBJ-1.2`
`prompt> cd VBJ-1.2`
`prompt> chmod -R o+rX classes`

Am einfachsten lassen sich alle Umgebungsvariablen setzen, indem man folgende Zeilen in die Datei `.login` einfügt:

```
setenv VBJ_HOME $HOME/VBJ-1.2
setenv ORBELINE $VBJ_HOME/adm_delhi
setenv CLASSPATH .:$VBJ_HOME/classes:$CLASSPATH
setenv PATH $PATH:$VBJ_HOME/bin
```

A.2 OrbixWeb 1.0

Voraussetzung für den Einsatz von OrbixWeb 1.0 ist eine bereits installierte Version von Orbix, da diese benötigt wird, um die Orbix Server zu implementieren. Die Software kann von dem FTP-Server von Iona geladen werden und liegt als Datei `OrbixWeb1.0.tar.gz` vor.

Die Installationsschritte im einzelnen

1. Festlegen des Verzeichnisses, in das OrbixWeb installiert werden soll, zum Beispiel im Homeverzeichnis `$HOME`.
2. Kopiere die gepackte Datei `OrbixWeb1.0.tar.gz` in dieses Verzeichnis:
`prompt> cp OrbixWeb1.0.tar.gz $HOME`
3. Wechsle in dieses Verzeichnis und entpacke die Datei:
`prompt> cd $HOME`
`prompt> gunzip OrbixWeb1.0.tar.gz`
4. Extrahiere den Inhalt der resultierenden Datei:
`prompt> tar -xvf OrbixWeb1.0.tar`
5. Editiere die Datei `Makefile.config` im neu entstandenen Verzeichnis `OrbixWeb1.0`:
`JAVAORB = /fzi/dbs/wipf/OrbixWeb1.0`
`JAVAHOME = /tools/JDK-1.0.2`
`ORBIXHOME = /tools/Orbix-2.0.1/bin`
6. Mache das Verzeichnis `OrbixWeb1.0/classes` und alle untergeordneten Dateien und Verzeichnisse für alle lesbar:
`prompt> cd $HOME`
`prompt> chmod o+rX OrbixWeb1.0`
`prompt> cd OrbixWeb1.0`
`prompt> chmod -R o+rX classes`
7. Eventuell muß die Datei `Configure.java` im Verzeichnis `OrbixWeb1.0/java` angepaßt und neu übersetzt werden. Hier können einige Einstellungen vorgenommen werden, wie zum Beispiel der Port für den Orbix-Dämon. Zum Übersetzen kann das `Makefile` verwendet werden.
8. Werden nicht die mit OrbixWeb mitgelieferten `Makefiles` verwendet, ist es sinnvoll, noch einige Umgebungsvariablen zu setzen:
`prompt> setenv ORBIXWEB_HOME $HOME/OrbixWeb-1.0`
`prompt> setenv CLASSPATH .:$ORBIXWEB_HOME/classes`

Das Verzeichnis mit dem IDL Compiler sollte nicht in `PATH` aufgenommen werden, sonst kommt es zu Konflikten mit dem leider gleichnamigen IDL Compiler von Orbix.

A.3 OrbixWeb 2.0 beta2

OrbixWeb 2.0 beta2 kann von dem FTP-Server von Iona mit entsprechendem Passwort geladen werden und liegt als Datei `OrbixWeb2.0_beta2.tar.gz` vor. Im Prinzip wird keine installierte Version von Orbix benötigt, da es nun auch möglich ist Server mit OrbixWeb zu implementieren. Die Verwendung des Orbix-Dämons ist nicht unbedingt nötig, da auch das IIOP verwendet werden kann. Die Nutzung des Orbix-Dämons ist auf sechzig Tage beschränkt, er erlaubt aber im Gegensatz zu einem normalen Orbix-Dämon das direkte Starten von Java Servern ohne den Umweg über ein Shell-Script.

Die Installationsschritte im einzelnen

1. Festlegen des Verzeichnisses, in das OrbixWeb installiert werden soll, zum Beispiel im Homeverzeichnis `$HOME`
2. Kopiere die gepackte Datei `OrbixWeb2.0_beta2.tar.gz` in dieses Verzeichnis:
`prompt> cp OrbixWeb2.0_beta2.tar.gz $HOME`
3. Wechsle in dieses Verzeichnis und entpacke die Datei:
`prompt> cd $HOME`
`prompt> gunzip OrbixWeb2.0_beta2.tar.gz`
4. Extrahiere den Inhalt der resultierenden Datei:
`prompt> tar -xvf OrbixWeb2.0_beta2.tar`
5. Editiere die Datei `Makefile.config` im neu entstandenen Verzeichnis `OrbixWeb2.0b2`:
`JAVAORB = /fzi/dbs/wipf/OrbixWeb2.0b2`
`JAVAHOME = /tools/JDK-1.0.2`
`UTILSBINDIR = /tools/Orbix-2.0.1/bin`
6. Editiere die Datei `jrunit` im Verzeichnis `OrbixWeb2.0b2/bin`:
`JAVAHOME = /tools/JDK-1.0.2`
`JAVAORB = /fzi/dbs/wipf/OrbixWeb2.0b2`
7. Mache das Verzeichnis `OrbixWeb2.0b/classes` und alle untergeordneten Dateien und Verzeichnisse für alle lesbar:
`prompt> cd $HOME`
`prompt> chmod o+rX OrbixWeb2.0b2`
`prompt> cd OrbixWeb2.0b2`
`prompt> chmod -R o+rX classes`
8. Die Datei `Configure.java` im Verzeichnis `OrbixWeb2.0b2/java` muß angepaßt und anschließend neu übersetzt werden:
`_CORBA.IT_ORBIXD_PORT = 21300;`
`_CORBA.IT_LOCAL_DOMAIN = "fzi.de";`
9. Werden nicht die mit OrbixWeb mitgelieferten Makefiles verwendet, ist es sinnvoll, noch einige Umgebungsvariablen zu setzen:
`prompt> setenv ORBIXWEB_HOME $HOME/OrbixWeb-2.0b2`
`prompt> setenv CLASSPATH .:$ORBIXWEB_HOME/classes`

Das Verzeichnis mit dem IDL Compiler sollte nicht in `PATH` aufgenommen werden, sonst kann es zu Konflikten mit dem leider gleichnamigen IDL Compiler von Orbix kommen.

Anhang B Erstellen einer einfachen Client/Server Anwendung mit Java

Am Beispiel eines verteilten „Hello World“ Programms soll gezeigt werden, wie sich die Programmierung mit verschiedenen Java ORBs unterscheidet und was zu beachten ist. Der Client gibt eine Meldung aus, die er vom Server durch Aufruf der Methode `sayHello()` erhält. Die IDL Schnittstelle sieht wie folgt aus:

```
// HelloWorld.idl
module HelloWorld {
    interface Hello {
        string sayHello();
    };
};
```

B.1 VisiBroker for Java 1.2

Einrichten der Umgebung

Als erstes muß die Umgebung richtig eingerichtet werden, siehe hierzu Anhang A.1. Als nächstes müssen die von VisiBroker for Java benötigten Programme gestartet werden:

- OSAgent starten:
`prompt> osagent &`
Der OSAgent benutzt per default den Port 14000. Dies kann über die Umgebungsvariable `OSAGENT_PORT` geändert werden.
- Bei Verwendung eines Applets als Client muß der Gatekeeper gestartet werden:
`prompt> gatekeeper`
Der GateKeeper arbeitet per default an Port 15000. Der Port kann mittels der Umgebungsvariable `OAPORT` geändert werden.

Compilieren der IDL

Der nächste Schritt ist die Übersetzung der IDL:

```
prompt> idl2java HelloWorld.idl
```

Der Compiler erzeugt ein neues Verzeichnis, das ein Java Package enthält. Verzeichnis und Package werden nach dem IDL Module benannt, hier also `HelloWorld`.

Erstellen einer Client Applikation

```
// HelloClient.java
public class HelloClient {
    public static void main(String args[]) {

        try {
            // initialisiere den ORB
            CORBA.ORB orb = CORBA.ORB.init();
            // lokalisiere ein Hello Objekt
            HelloWorld.Hello myHello =
                HelloWorld.Hello_var.bind("HelloSrv");
            // rufe entfernte Methode auf und gib Ergebnis aus
```

```

        System.out.println(myHello.sayHello());
    }
    catch (CORBA.SystemException e) {
        System.err.println(e);
    }
}
}

```

Erstellen der Objekt Implementierung

```

// HelloImpl.java
class HelloImpl extends HelloWorld._sk_Hello {

    private String locality;

    // Konstruktor
    public HelloImpl(String locality) {
        super("HelloSrv");
        // initialisiere locality
        this.locality = new String(locality);
    }

    public String sayHello() throws CORBA.SystemException {
        return "Hallo Welt aus " + this.locality;
    }
}

```

Erstellen des Servers

```

// HelloServer.java
public class HelloServer {

    public static void main(String args[]) {
        if (args.length != 1) {
            System.out.println(
                "Aufruf: java HelloWorldServer <location>");
            System.exit(1);
        }

        try {
            // initialisiere den ORB
            CORBA.ORB orb = CORBA.ORB.init();
            // initialisiere den Basic Object Adapter
            CORBA.BOA boa = orb.BOA_init();
            // erzeuge Hello Objekt
            HelloImpl myHello =
                new HelloImpl(args[0]);
            // exportiere Objektreferenz
            boa.obj_is_ready(myHello);
            // warte auf Anfragen
            boa.impl_is_ready();
        }
        catch(CORBA.SystemException e) {
            System.err.println(e);
        }
    }
}

```

Übersetzen aller Programmteile und Starten

- Compilieren des Java Sourcecodes:

```
prompt> javac HelloClient.java
prompt> javac HelloImpl.java
prompt> javac HelloServer.java
```

- Server starten:

```
prompt> bw HelloServer Karlsruhe
```

- Client starten:

```
prompt> bw HelloClient
```

Erstellen eines Client Applets

Das Erstellen eines Client Applets unterscheidet sich nur geringfügig von der Programmierung einer Client Applikation. Der Client muß von der Klasse `java.applet.Applet` abgeleitet werden und der ORB mit dem Applet initialisiert werden. Der einfacheren Bedienung wegen erfolgt der Aufruf der entfernten Methode über einen Button, die Ausgabe erfolgt in einem Textfeld.

```
// HelloApplet.java

import java.awt.*;

public class HelloApplet extends java.applet.Applet {

    private HelloWorld.Hello myHello;
    private Button helloB;
    private TextField textF;

    public void init() {
        helloB = new Button("Sag Hallo !");
        textF = new TextField();
        textF.setEditable(false);

        setLayout(new GridLayout(2,1));
        add(helloB);
        add(textF);

        try {
            // initialisiere den ORB unter Verwendung des Applets
            CORBA.ORB orb = CORBA.ORB.init(this);
            // bind an Objekt
            myHello = HelloWorld.Hello_var.bind();
        }
        catch(CORBA.SystemException e) {
            System.err.println(e);
        }
    }

    public boolean action(Event ev, Object arg) {

        if(ev.target == helloB) {
            // rufe entfernte Methode auf
            try {
                textF.setText(myHello.sayHello());
            }
        }
    }
}
```

```

        catch(CORBA.SystemException e) {
            System.err.println(e);
        }
        return true;
    }
    return false;
}
}

```

Übersetzen des Applets

```
prompt> javac HelloApplet.java
```

Erstellen einer HTML Seite

Mit Hilfe des `<applet>` Tags wird das Client Applet in die HTML Seite eingebunden und kann dann mit dem Appletviewer oder einem Java-fähigen Browser gestartet werden.

```

<html>
<body>
<center>
<h1>Hello World Beispiel</h1>
<hr>
<applet code=HelloApplet.class width=400 height=80>
</applet>
<hr>
</center>
</body>
</html>

```

Starten der Anwendung

Der Server sollte noch laufen, sonst wie oben beschrieben starten. Die Applet Version des Clients kann man nun auf mehrere Arten ausprobieren:

- lokal ohne Web-Server
Dies erfordert, daß sich Client und Server auf demselben Rechner befinden und der Appletviewer benutzt wird. Dabei ist zu beachten, daß sich der Appletviewer des JDK 1.0 nicht mit dem Netscape Navigator 2.0 verträgt, d.h. Netscape muß zuerst entfernt werden. Neuere Versionen des Netscape Navigators bereiten keine Probleme mehr. Der Gebrauch des Appletviewers auf einem anderen Rechner, als auf dem der Gatekeeper gestartet wurde, wird von VisiBroker for Java nicht unterstützt, dafür muß ein Web-Server benutzt werden. Die `CLASSPATH` Variable muß alle Verzeichnisse enthalten, in denen sich die benötigten Java Klassen befinden.

Der Aufruf erfolgt dann mit:

```
prompt> appletviewer HelloWorld.html
```

- mit einem Web-Server
Als Web-Server kann man den Gatekeeper von VisiBroker oder einen richtigen Web-Server wie z.B. Apache verwenden.

Gatekeeper als Web-Server

Das Applet kann über die folgende URL gestartet werden:

```
http://<maschine>:<port>/HelloWorld.html
```

Dabei gibt `<maschine>` den Namen des Rechners an, auf dem der Gatekeeper läuft, und `<port>` den von diesem benutzten Port (default 15000). Beim Start des Gatekeepers muß darauf geachtet werden, daß alle benötigten Klassen in `CLASSPATH` liegen.

Normaler Web-Server

Bei Verwendung eines normalen Web-Servers wird CLASSPATH nicht verwendet. Java-fähige Browser suchen die benötigten Klassen an Hand des CODEBASE Eintrags im Applet Tag. Dabei ist darauf zu achten, daß alle Klassen, auch die vom Java ORB benötigten Klassen für alle Benutzer lesbar sind. Wird der CODEBASE Eintrag weggelassen, sucht der Browser in dem Verzeichnis, von dem die HTML Seite geladen wurde. Da sich die ORB Klassen und unsere Beispiel Klassen nicht im selben Verzeichnis befinden, richtet man unter UNIX am einfachsten Links auf die Verzeichnisse mit den VisiBroker Klassen ein. In unserem Fall:

```
prompt> ln -s $VBJ_HOME/classes/CORBA CORBA
prompt> ln -s $VBJ_HOME/classes/pomoco pomoco
```

Der Gatekeeper *muß* auf demselben Rechner laufen, auf dem sich auch der Web-Server befindet. Die URL entspricht der obigen, wobei Rechnername, Port und Pfad zur HTML Seite anzupassen sind.

B.2 OrbixWeb 2.0

Wie die OrbixWeb Umgebung eingerichtet werden muß, kann Anhang A.3 entnommen werden. Für unser Beispiel ist dabei auch der Punkt 9 wichtig, in dem einige Umgebungsvariablen gesetzt werden.

Aufruf des IDL Compilers

```
prompt> $ORBIXWEB_HOME/bin/idl -jQ -jO . HelloWorld.idl
```

Mit dem Parameter -jO kann angegeben werden, wo die erzeugten Java Sourcen abgelegt werden. Per default werden sie sonst in das Verzeichnis java_output geschrieben.

Erstellen einer Client Applikation

```
// HelloClient.java
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;

public class HelloClient {
    public static void main(String args[]) {

        HelloWorld._HelloRef myHello = null;
        String hostname;

        // benutze Orbix Protokoll nicht IIOP
        _CORBA.IT_BIND_USING_IIOP = false;

        if (args.length < 1) // benutze lokalen orbixd
            hostname = new String(_CORBA.Orbix.myHost());
        else // benutze Remote Host
            hostname = new String(args[0]);

        try {
            // erfrage Objektreferenz
            myHello = HelloWorld.Hello._bind(":HelloSrv", hostname);
            System.out.println(myHello.sayHello());
        }
        catch (SystemException e) {
```

```

        System.out.println("Exception during bind: " + e.toString());
        System.exit(1);
    }
}

```

Erstellen der Objektimplementierung

```

import IE.Iona.Orbix2.CORBA.SystemException;

class HelloImpl extends HelloWorld._boaimpl_Hello {
    private String locality;

    public HelloImpl(String locality) throws SystemException {
        this.locality = locality;
    }

    public String sayHello() {
        return "Hallo Welt aus " + locality;
    }
}

```

Erstellen des Servers

```

import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;

public class HelloServer {
    public static void main(String args[]) {

        HelloWorld._HelloRef myHello = null;

        if (args.length != 1) {
            System.out.println("Aufruf: java HelloServer <location>");
            System.exit(1);
        }

        try {
            // erzeuge neues Hello Objekt
            myHello = new HelloImpl(args[0]);
            // warte auf Anfragen
            _CORBA.Orbix.impl_is_ready("HelloSrv");
        }
        catch(SystemException e) {
            System.out.println("Exception: " + e.toString());
            System.exit(1);
        }
    }
}

```

Übersetzen aller Programmteile und Starten

Mit dem Java Compiler werden wieder alle Sourcdateien übersetzt. Dies geht ganz einfach mit folgendem Aufruf:

```
prompt> javac *.java
```

Der Server muß nun noch beim Orbix-Dämon registriert werden. Für den mit OrbixWeb 2.0b2 mitgelieferten Orbix-Dämon haben wir leider keine Lizenz mehr, da es nur eine sechzig Tage

Testversion war. Deshalb müssen wir auf einen normalen mit Orbix 2.0 mitgelieferten Orbix-Dämon zurückgreifen. Dieser kann aber Java Server nicht direkt starten. Der Aufruf des Servers muß deshalb über ein Shell-Script erfolgen. Dazu benutzen wir das Script `jrunit` im `OrbixWeb /bin` Verzeichnis.

```
prompt> orbixd &
prompt> putit HelloSrv "$ORBIXWEB_HOME/bin/jrunit -classpath
                /fzi/dbs/wipf/<path_to_server> HelloServer Karlsruhe"
```

Danach kann der Client aufgerufen werden:

```
prompt> java HelloClient
```

oder beim Aufruf von einem nicht lokalen Rechner:

```
prompt> java HelloClient <server_hostname>
```

B.3 Java IDL

Java IDL ist bisher nur in einer „Early Access“ Version erhältlich, wird aber Bestandteil des für Ende 1997 angekündigten JDK 1.2 sein. Das Produkt sei in dem mit der Umgebungsvariable `JAVAIDL_HOME` bezeichneten Verzeichnis installiert.

Aufruf des IDL Compilers

```
prompt> $JAVAIDL_HOME/bin/idltojava -fclient -fserver HelloWorld.idl
```

Erstellen einer Client Applikation

```
// HelloClient.java
import HelloWorld.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient {
    public static void main(String args[]) {
        try {
            // initialisiere den ORB
            ORB orb = ORB.init(args, null);

            // ermittle Root Naming Context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // erfrage Objekt Referenz vom Naming Service
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

            // rufe entfernte Methode auf und gib Ergebnis aus
            System.out.println(helloRef.sayHello());
        }
        catch (Exception e) {
            System.out.println("ERROR : "+ e) ;
            e.printStackTrace();
        }
    }
}
```

```
}
```

Erstellen der Objektimplementierung

```
// HelloImpl.java
import HelloWorld.*;

class HelloImpl extends _HelloImplBase {
    private String locality;
    public HelloImpl(String locality) {
        this.locality = new String(locality);
    }

    public String sayHello() {
        return "Hallo Welt aus " + locality;
    }
}
```

Erstellen des Servers

```
// HelloServer.java
import HelloWorld.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloServer {

    public static void main(String args[]) {

        try {
            // initialisiere den ORB
            ORB orb = ORB.init(args, null);

            // erzeuge Hello Objekt und registriere es beim ORB
            HelloImpl helloRef = new HelloImpl("Karlsruhe");
            orb.connect(helloRef);

            // ermittle den Root Naming Context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // registriere Objekt Referenz beim Naming Service
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, helloRef);

            // warte auf Aufrufe von Clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }
        }
        catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Übersetzen der Programmteile

Alle Source Dateien werden wie gewohnt mit dem Java Compiler übersetzt. Dabei müssen sich die Java IDL Klassen in CLASSPATH befinden.

```
prompt> setenv CLASSPATH ./ $JAVAIDL_HOME/lib/classes.zip
prompt> javac *.java
```

Starten der Anwendung

Zuerst muß der Namensdienst von Java IDL gestartet werden. Mit dem Parameter ORBInitialPort kann man den Port einstellen, den der Name Server verwendet.

```
prompt> nameserv -ORBInitialPort 21380
```

Danach starten wir den Server. Auch hier müssen wir den verwendeten Port als Parameter übergeben. Soll der Server auf einem anderen Rechner laufen, als auf dem der Name Server gestartet wurde, muß noch der Parameter ORBInitialHost und der Rechnername des Namensdienstes angegeben werden.

```
prompt> java HelloServer -ORBInitialPort 21380
```

oder

```
prompt> java HelloServer -ORBInitialPort 21380 -ORBInitialHost <host>
```

Mit den gleichen Parametern starten wir den Client:

```
prompt> java HelloClient -ORBInitialPort 21380
```

B.4 RMI

```

private String locality;

public HelloRMIIImpl(String name, String locality)
    throws java.rmi.RemoteException {

    super();
    try {
        // registriere Servername und Objektreferenz
        Naming.rebind(name, this);
        this.locality = new String(locality);
    }
    catch(Exception e) {
        System.out.println("Exception: " + e.getMessage());
    }
}

public String sayHello() throws RemoteException {
    return "Hallo Welt aus " + locality;
}
}

```

Übersetzen der Schnittstelle und deren Implementierung, Aufruf des RMI Compilers

Die Java Schnittstelle und deren Implementierung müssen mit dem Java Compiler übersetzt werden. Mit dem Parameter `-d` legen wir fest, daß das Java Package bzw. das Verzeichnis `HelloWorld` unterhalb des aktuellen Verzeichnisses angelegt wird:

```

prompt> javac -d . HelloRMI.java
prompt> javac -d . HelloRMIIImpl.java

```

Dann können mit dem RMI Compiler Stubs und Skeletons erzeugt werden:

```

prompt> rmic -d . HelloRMIIImpl

```

Erstellen des Servers

```

// HelloRMIServer.java
import java.rmi.*;
import java.rmi.server.*;

public class HelloRMIServer {
    public static void main(String args[]) {

        // Erzeuge und initialisiere den Security Manager
        System.setSecurityManager(new RMISecurityManager());

        if (args.length != 1) {
            System.out.println(
                "Usage: java HelloRMIServer <location>");
            System.exit(1);
        }

        try {
            // erzeuge HelloRMI Objekt
            HelloWorld.HelloRMIIImpl obj =
                new HelloWorld.HelloRMIIImpl("HelloSrv", args[0]);
            System.out.println("HelloRMI Server bereit");
        }
        catch(Exception e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}

```

```
}  
}  
}
```

Erstellen des Clients

```
// HelloRMIClient.java  
import java.rmi.*;  
  
public class HelloRMIClient {  
    public static void main(String args[]) {  
  
        // erzeuge und initialisiere den Security Manager  
        System.setSecurityManager(new RMISecurityManager());  
  
        if (args.length != 1) {  
            System.out.println(  
                "Usage: java HelloRMIClient <server_location>");  
            System.exit(1);  
        }  
  
        try {  
            // erfrage Objektreferenz aus der Registry  
            HelloRMI myHello =  
                (HelloRMI)Naming.lookup("rmi://" + args[0] + "/HelloSrv");  
            System.out.println(myHello.sayHello());  
        }  
        catch (Exception e) {  
            System.out.println("Exception: " + e.getMessage());  
        }  
    }  
}
```

Übersetzen von Client und Server

Mit dem Java Compiler werden Client und Server übersetzt:

```
prompt> javac HelloRMIClient.java  
prompt> javac HelloRMIServer.java
```

Ausführen des Programms

Zuerst muß der RMI Namensdienst, die sogenannte RMI Registry, gestartet werden:

```
prompt> rmiregistry &
```

Danach wird der Server gestartet:

```
prompt> java HelloRMIServer Karlsruhe
```

Nun kann der Client aufgerufen werden:

```
prompt> java HelloRMIClient
```

B.5 Caffeine

Caffeine soll es dem Programmierer ermöglichen, CORBA Anwendungen mit VisiBroker for Java zu erstellen, ohne dabei IDL Schnittstellen benutzen zu müssen. Dabei wird der gleiche Ansatz wie bei RMI verwendet, daß Schnittstellen in Java beschrieben werden. Die Schnittstelle ist fast identisch zu der aus Anhang B.4, wird aber von `CORBA.Object` abgeleitet.

```
// HelloCaffeine.java
package HelloWorld;

public interface HelloCaffeine extends CORBA.Object {
    String sayHello();
}
```

Mit dem Java Compiler wird die Schnittstelle übersetzt:

```
prompt> javac -d . HelloCaffeine.java
```

Nun lassen wir uns Stubs und Skeletons erzeugen:

```
prompt> java2iiop HelloWorld.HelloCaffeine
```

Die Implementierung der Schnittstelle sowie von Client und Server können wir uns ersparen, da sie bis auf die Namensgebung identisch zu der in Anhang B.1 ist. Der `java2iiop` Compiler hat aus der Java Schnittstelle genau dieselben Java Klassen erzeugt, wie der `idl2java` Compiler aus der entsprechenden IDL.

Leider funktioniert Caffeine erst ab VisiBroker for Java 2.5, weshalb noch kleinere Details aufgrund des neuen Mappings zu ändern sind. Zudem muß das JDK 1.1 verwendet werden, da einige erst in dieser Version verfügbaren Funktionen benutzt werden. Danach läßt sich die Anwendung wie in Anhang B.1 beschrieben ausführen.

Anhang C Quelltexte der Beispiel-Anwendung „Ereignis-Monitor“

Auf den folgenden Seiten finden Sie die kompletten Quelltexte zu der in Kapitel 5.2 vorgestellten Beispielanwendung.

C.1 IDL Beschreibung

```
// event.idl
// IDL Beschreibung für einfachen Ereignis-Monitor
// mit CORBA, Java und IIOP (Orbix/VisiBroker for Java)

interface Callback;

// Schnittstelle fuer den Client
interface EventHandler {

    // Erzeugen von Meldungen
    void CreateMessage (in string message);

    // Registrieren des Client Applets, dabei wird eine Referenz
    // auf das Objekt uebergeben, das eine Methode zur Entgegen
    // nahme von Meldungen (Events) enthaelt, muss vor
    // CreateMessage aufgerufen werden
    void RegisterClient (in Callback obj);

    // Registrierung fuer einen Client aufheben
    void RemoveClient (in Callback obj);
};

// Schnittstelle fuer den Server zum asynchronen Rueckruf,
// wird nur innerhalb der Serverimplementierung benutzt
interface Callback {

    // Meldungen an den Client schicken
    oneway void NewMessage (in string message);
};
```



```

    in.close();
}
catch (Exception e) {
    System.out.println("Exception: " + e);
    return;
}

try {
    orb = CORBA.ORB.init(this);
    CORBA.Object obj = orb.string_to_object(IOR);
    myEvent = IDL_GLOBAL.EventHandler_var.narrow(obj);
}
catch(CORBA.SystemException e) {
    System.out.println("Exception bei narrow(): " + e);
    return;
}

// initialisiere den BOA
try {
    boa = orb.BOA_init();
}
catch (CORBA.SystemException ex) {
    System.out.println("Exception beim Initialisieren des BOA");
    System.out.println(ex.toString());
}

try {
    host = InetAddress.getLocalHost().getHostName();
}
catch(UnknownHostException u) {
    System.out.println("Exception: Host unbekannt");
}

start = new Button("Starte Ereignis Monitor");
this.add(start);
}

public boolean action(Event e, Object obj) {
    if (e.target == start) {
        try {
            LocalImpl loc = new LocalImpl(myEvent);
            boa.obj_is_ready(loc);
            System.out.println("Lokaler Ereignis Handler verfuegbar");
            myEvent.RegisterClient(loc);
            myEvent.CreateMessage("Client Applet auf "
                + host
                + " gestartet und registriert!");
        }
        catch (CORBA.SystemException ex) {
            System.out.println(
                "Exception beim Aufruf des Event Servers");
            System.out.println(ex.toString());
        }
        return true;
    }
    return false;
}
}

/***** End of EventApplet.Java *****/

```

```

// Monitor.java
// Monitor-Komponente zur Darstellung von Meldungen in einem Fenster

import java.awt.*;

public class Monitor extends Frame {

    private Panel buttons;
    private TextArea eventarea;
    private Button quit, reset;
    MonitorCallback callback;

    public Monitor(String title, int rows, int cols,
                  MonitorCallback c) {
        super(title);
        callback = c;

        this.setLayout(new BorderLayout());
        eventarea = new TextArea(rows, cols);
        eventarea.setEditable(false);
        this.add("Center", eventarea);
        quit = new Button("Beenden");
        reset = new Button("Anzeige löschen");
        buttons = new Panel();
        buttons.add(reset);
        buttons.add(quit);
        this.add("South", buttons);
        this.pack();
        this.show();
    }

    public Monitor(String title, MonitorCallback c) {
        this(title, 10, 40, c);
    }

    public void setText(String text) {
        eventarea.setText(text);
    }

    public void appendText(String text) {
        eventarea.appendText(text);
    }

    public boolean action(Event e, Object obj) {
        if (e.target == quit) {
            this.hide();
            this.dispose();
            callback.quit();
            return true;
        }
        else if (e.target == reset) {
            this.setText("");
            return true;
        }
        else return false;
    }
}

interface MonitorCallback {
    // wird beim Beenden des Monitors aufgerufen
    void quit();
}

```

C.3 Server (Orbix)

Es folgen die Quelltexte der server-seitigen Komponenten, die mit Orbix in C++ implementiert wurden. Zusätzlich wurde noch ein einfacher Orbix Client geschrieben, mit dem Meldungen an den Ereignis-Server geschickt werden können.

```
// event_i.h
// Header für die Implementierung des Ereignis-Servers

#include "event.hh"
#include <iostream.h>

class EventHandler_i : public EventHandlerBOAImpl {
private:
    int max_clients;
    Callback_ptr ConnectedClients[16];
    int count;
public:
    EventHandler_i();
    ~EventHandler_i();
    virtual void CreateMessage (const char * message,
                               CORBA::Environment &env);
    virtual void RegisterClient (Callback_ptr obj,
                                 CORBA::Environment &env);
    virtual void RemoveClient (Callback_ptr obj,
                               CORBA::Environment &env);
};

class Callback_i : public CallbackBOAImpl {
public:
    virtual void NewMessage (const char * message,
                             CORBA::Environment &env);
};
```

```

// event_i.cc
// Implementierung des Ereignis-Servers

#include "event_i.h"
#include <iostream.h>

EventHandler_i::EventHandler_i() {
    max_clients = 16;
    count = 0;
    for (int i=0; i<max_clients; i++) {
        ConnectedClients[i] = Callback::_nil();
    }
}

EventHandler_i::~EventHandler_i() {}

void EventHandler_i::CreateMessage (const char * message,
                                   CORBA::Environment &IT_env) {

    Callback_ptr cbPtr;
    int connected = 0;
    for (int i=0; i<max_clients; i++) {
        if (!CORBA::is_nil(ConnectedClients[i])) {
            cbPtr = Callback::_narrow(ConnectedClients[i]);
            cbPtr->NewMessage(message);
            connected = 1;
        }
    }
    if (!connected) {
        cout << "Keine Clients registriert" << endl;
    }
}

void EventHandler_i::RegisterClient (Callback_ptr obj,
                                     CORBA::Environment &IT_env) {

    ConnectedClients[count] = Callback::_duplicate(obj);
    cout << "Client " << count << " registriert" << endl;
    count++;
    count %= max_clients;
}

void EventHandler_i::RemoveClient (Callback_ptr obj,
                                   CORBA::Environment &IT_env) {

    char* stringObj;
    char* tmp;
    int removed = 0;
    stringObj = CORBA::Orbix.object_to_string(obj);
    for (int i=0; i<max_clients; i++) {
        if (!CORBA::is_nil(ConnectedClients[i])) {
            tmp = CORBA::Orbix.object_to_string(ConnectedClients[i]);
            if (strcmp(stringObj, tmp) == 0) {
                ConnectedClients[i] = Callback::_nil();
                cout << "Client " << i << " entfernt" << endl;
                removed = 1;
                break;
            }
        }
    }
    if (!removed) {
        cout << "Nichts zu tun" << endl;
    }
}

```

```

// srv_main.cc
// Server-Hauptprogramm

#define USE_IIOP

#include <iostream.h>
#include <stdlib.h>
#include <stream.h>
#include "event_i.h"

int main() {

    // erzeuge EventHandlerer Objekt
    EventHandlerer_i myEvent;

    ofstream strm("/fzi/dbs/wipf/tmp/eventiiop.ref");
    if (!strm) {
        cout << "Fehler: Konnte Datei nicht erzeugen - "
             << "/fzi/dbs/wipf/tmp/eventiiop.ref" << endl;
        return 0;
    }

    char * stringObj;

    try {
        // erst Servername setzen, sonst ist kein manuelles Starten des
        // Servers moeglich, da IOR falsch (Orbix Restriktion)
        CORBA::Orbix.setServerName("EventMultiUser");
        stringObj = CORBA::string_dupl(
            myEvent._object_to_string(CORBA::IT_INTEROPERABLE_OR_KIND));
    }
    catch(CORBA::SystemException &sysEx) {
        cerr << "Fehler: " << &sysEx;
    }

    strm << stringObj << endl;
    CORBA::string_free(stringObj);

    try {
        cout << "Der Ereignis Server wurde gestartet" << endl;
        // teile Orbix mit, daß die Server-Implementierung verfügbar ist
        // und setze Timeout auf unendlich
        CORBA::Orbix.impl_is_ready("EventMultiUser",
            CORBA::Orbix.INFINITE_TIMEOUT);
    }
    catch (CORBA::SystemException &sysEx ){
        cerr << "Fehler: " << &sysEx << endl;
        exit(1);
    }

    cout << "Server beendet" << endl;

    return 0;
}

```

```

// client.cc
// Client fuer Java/CORBA Ereignis-Monitor
// Ermöglicht es, auf der Konsole Nachrichten einzugeben, die
// dann im Browser bzw. dem Ereignis-Monitor dargestellt werden

#include "event.hh"
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char **argv) {
    EventHandler_var eventVar; // Zeiger auf das EventHandler Objekt

    if (argc < 2) {
        cout << "Benutzung: "<< argv[0] << " <hostname>" << endl;
        exit (-1);
    }
    try {
        // argv[1] muss den Host enthalten, auf dem der Event Server
        // registriert wurde
        eventVar = EventHandler::_bind(":EventMultiUser", argv[1]);
        cout << "Bind zum Ereignis-Server erfolgreich" << endl;
    }
    catch (CORBA::SystemException &sysEx) {
        cerr << "Fehler: " << &sysEx;
        exit(1);
    }
    catch(...) {
        // alle möglichen Fehler abfangen
        cerr << "Fehler beim Aufruf von bind()" << endl;
        exit(1);
    }

    // bilde eine Konsole nach, in der Eingaben gemacht werden können,
    // die dann im Ereignis-Monitor angezeigt werden

    char linebuf[256];

    // Endlosschleife bis exit oder quit eingegeben wird
    for (;;) {
        cout << "EVENT> " ;
        cin.getline(linebuf, 256, '\n');

        // prüfe, ob Programm beendet werden soll
        if ((strncasecmp(linebuf, "exit", 4) == 0) ||
            (strncasecmp(linebuf, "quit", 4) == 0)) {
            return -1;
        }
        else {
            try {
                // Erzeuge Meldung mit dem Inhalt der Eingabezeile
                eventVar->CreateMessage(linebuf);
            }
            catch (CORBA::SystemException &sysEx) {
                cerr << "Fehler: " << &sysEx;
                exit(1);
            }
            catch(...) {
                cerr << "Aufruf von CreateMessage fehlgeschlagen" << endl;
                exit(1);
            }
        }
    }
}

```

```
}  
return 0;  
}
```