16. Remote Method Invocation

16.1 Object Request Broker

16.1.1 Remote Procedure Call

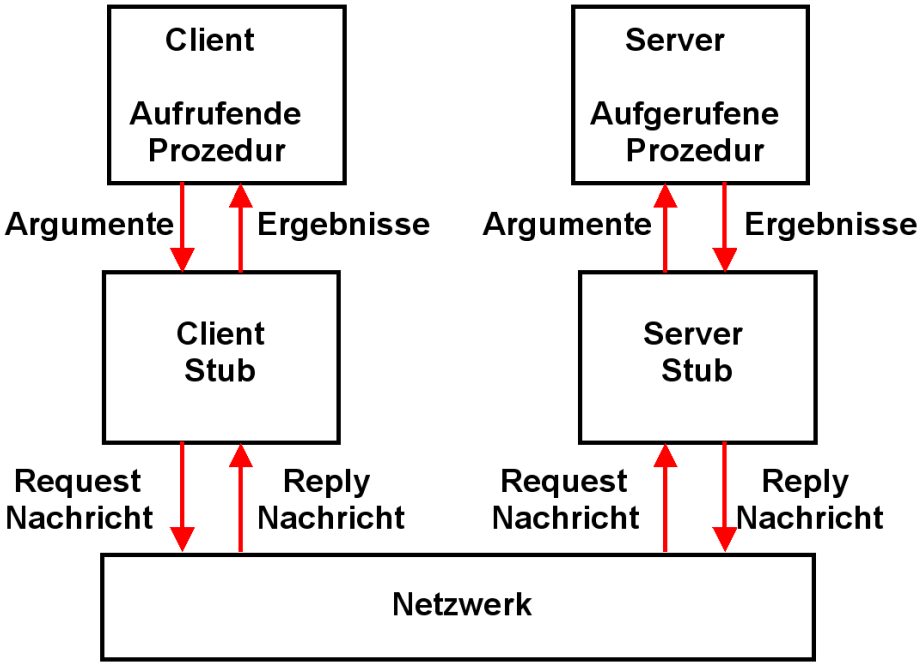

Abb. 16.1.1
Server und Client Stubs

Siehe hierzu auch Band 1, Abschnitt 10.1.1.

Client und Server laufen als zwei getrennte Prozesse.

Die beiden Prozesse kommunizieren über Stubs. Stubs sind Routinen, welche Prozedur-Aufrufe auf Netzwerk RPC Funktionsaufrufe abbilden.

Ein Server-seitiges RPC Programm stellt den Klienten seine Dienste über eine Interface (Schnittstelle) zur Verfügung. Es definiert seine Interface unter Benutzung einer Interface Definition Language (IDL).

Ein Stub Compiler liest die IDL Schnittstellenbeschreibung und produziert zwei Stub Proceduren für jede Server Procedure (Client Side Stub und Server Side Stub).

Der klassische RPC benutzt die Bezeichnung Server Stub. Modernere RPCs verwenden statt dessen die Bezeichnung Skeleton. Die Bezeichnungen Server Stub und Skeleton sind austauschbar.

Stubs bzw. Skeletons implementieren eine RPC Runtime. Sie übersetzen Aufrufe der Client API in Sockets und dann in Nachrichten, die über das Netz übertragen werden.

Beim CICS Distributed Program Link (DPL) benötigt der Aufruf eines entfernten Systems lediglich dessen Name (TRID) und die Bezeichnung der COMMAREA. Eine IDL ist nicht erforderlich.

16.1.2 Sockets und RPC

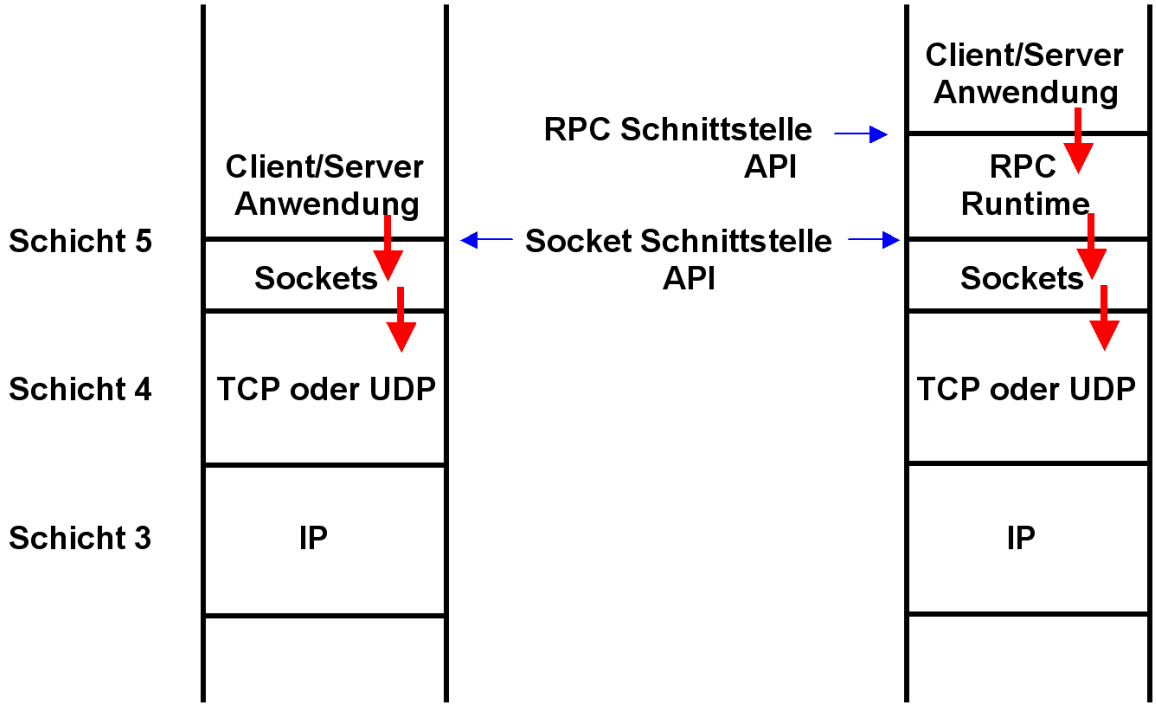

Abb. 16.1.2
Der RPC ist oberhalb der Socket Schicht einzuordnen

De Facto alle Client/Server Verbindungen benutzen heute TCP oder UDP in der Schicht 4 des OSI Modells. In der darüberliegenden Schicht 5 wird meistens das Socket Protokoll eingesetzt. Sockets sind leistungsfähig und vielseitig einsetzbar. Das Anwendungsprogramm des Klienten benutzt die Socket Schnittstelle, um mit dem Anwendungsprogramm (Service) des Servers zu kommunizieren. Sockets sind jedoch schwierig zu programmieren.

Deswegen wird universell der Remote Procedure Call (RPC) eingesetzt. Der RPC enthält bereits viele Funktionen, die bei der Nutzung von Sockets von Hand programmiert werden müssen. Das Anwendungsprogramm des Klienten benutzt die RPC Schnittstelle, um mit dem Anwendungsprogramm (Service, hier auch als Procedure bezeichnet) des Servers zu kommunizieren. Die RPC Runtime übersetzt die RPC Aufrufe in Socket Aufrufe.

Die RPC Runtime ist eine Sammlung von Routinen, welche die RPC Schnittstelle (API) implementieren.

16.1.3 IDL (Interface Definition Language)

Ein Server-seitiges RPC Programm stellt den Klienten seine Dienste über eine Interface (Schnittstelle) zur Verfügung. es definiert seine Interface unter Benutzung einer Interface Definition Language (IDL).

Die Prozedur eines Servers kann vom Klienten über eine Schnittstelle (Interface) in Anspruch genommen werden. Wenn man eine Server-seitige Anwendung entwickelt, erstellt man zwei Komponenten:

- **Die Anwendung selbst, oft als „Implementation“ bezeichnet,**
- **Die Schnittstelle (Interface), welche beschreibt, wie die Anwendung von einem Klienten aufgerufen werden kann, einschließlich der Beschreibung der übergebenen Parameter.**

Die Anwendung kann in einer beliebigen Programmiersprache geschrieben werden. Die Schnittstelle wird in einer spezifischen Sprache, der IDL (Interface Definition Language) beschrieben.

Mit Hilfe eines „IDL Compilers“ wird die in der IDL beschriebene Schnittstelle übersetzt. Das Ergebnis ist ausführbarer binary Code für Skeleton und Stub. Skeleton und Stub werden somit mit Hilfe des IDL Compilers automatisch generiert. Sie müssen die Eigentümlichkeiten der Schnittstelle der Server Prozedur kennen, nicht aber die Art, in der die Server Prozedur implementiert wurde. Spezifisch brauchen sie nicht zu wissen, in welcher Sprache (Cobol, C++, Java) die Server Prozedur implementiert wurde.

Die unterschiedlichen RPC Implementierungen verwenden unterschiedliche IDL Sprachen und damit unterschiedliche IDL Compiler.

Java nimmt eine Sonderstellung ein. Die Schnittstelle einer Java Klasse, z. B. die Remote Interface einer EJB, wird mit Hilfe eines Java Sprachelements, der Java Interface, beschrieben. Eine zusätzliche Java IDL ist in den meisten fällen nicht erforderlich.

16.1.4 Business Objekt

Business Objekte sind Repräsentationen von Dingen des täglichen Lebens.

Beispiel: Das Business Objekt „Fritz Meier“ ist eine Instanz (Ausprägung) der Objektklasse „Bankkonto“. Franz Müller, Barbara Schneider und Else Schmidt sind weitere Instanzen der Objektklasse Bankkonto.

Services der Objektklasse Bankkonto können in Anspruch genommen werden, indem man ihre Methoden aufruft. Derartige Methoden der Objektklasse Bankkonto können z.B. sein:

- **Kontostand abfragen**
- **Geld einzahlen**
- **Geld abheben**
- **Postanschrift ändern**
- **Vollmacht für Ehepartner erteilen**

usw.

Eine Methode wird aufgerufen, indem man an ihre Interface eine Nachricht schickt. Die Nachricht enthält einzelne Parameter, z.B.

- **Name/Adresse der Objektklasse, z.B. Bankkonto Fritz Meier**
- **Name der Methode, z.B. Kontostand abfragen**
- **übergebene Parameter, z.B. stand vom 1.Januar letzten Jahres**
- **Parameter der Antwort, z.B. € 638,15 .**

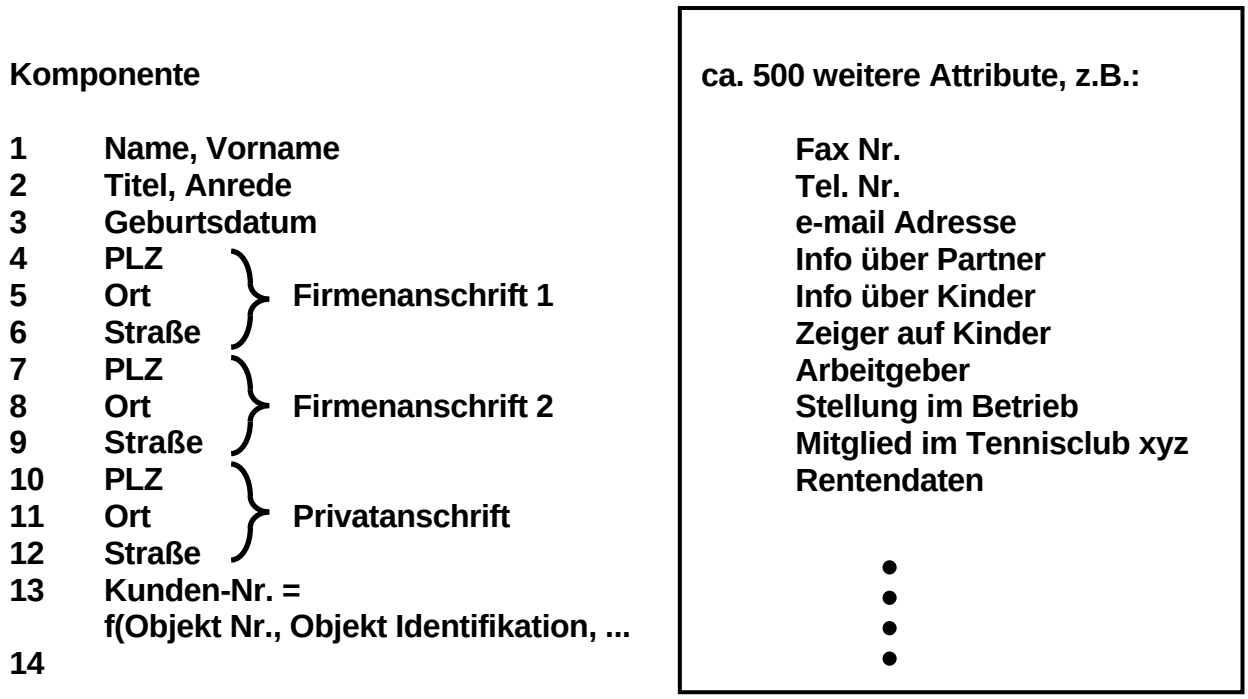

Abb. 16.1.3
Beispiel: Business Objekt “Kunde”

Abb. 8.2.3 stellt ebenfalls ein Business Objekt dar.

16.1.5 Wiederverwendbarkeit von Code

Vorbild: Entwurf / Bau einer Brücke im Bauingenieurwesen

Objekttechnologie ermöglicht Code Blöcke mit fest definierter Funktionalität, die in Klassen gekapselt werden. Ein Objekt als Instanz einer Klasse stellt sich dem Entwickler als Black Box mit einer öffentlichen Schnittstelle dar.

Wird nur in einer einzigen Sprache mit identischem Compiler entwickelt ist die Wiederverwendbarkeit von Klassen am leichtesten erreichbar. Beim Einsatz mehrerer Programmiersprachen muss die Objektphilosophie der Programmiersprache beachtet werden. Z.B. unterstützt C++ die Mehrfachvererbung, Java aber nur die Einfachvererbung.

Um objektorientierte Bibliotheken mit unterschiedlichen Sprachen und Compilern zu realisieren, ergeben sich eine Reihe von Problembereichen:

- **Sprachenabhängigkeit: Smalltalk- und C++-Objekte verstehen einander nicht.**
- **Compilerunabhängigkeit: Objekte zweier Compiler (z.B. Watcom C++ und *GNU* C++) können nicht miteinander kommunizieren, weil die Verwaltung interner Informationen nicht standardisiert ist.**
- **starre Kopplung zwischen Objekten: Wenn sich die Implementierung einer Klasse ändert, müssen alle Teile, die diese Klasse in irgendeiner Form nutzen, neu kompiliert werden. Beispiel: C++ Compiler benutzen Konstanten beim Zugriff auf Daten und Methoden.**
- **Beschränkung auf einen Prozessraum (Objekte können nicht über Prozessgrenzen hinweg kommunizieren).**

Zielsetzung: Unterschiedliche objektorientierte Klienten-Implementierungen, die auf unterschiedlichen Plattformen (z.B. HP-UX, AIX, Solaris, Linux, XP, z/OS) laufen, sollen mit unterschiedlichen objektorientierten Server-Implementierungen (ebenfalls unterschiedliche Plattformen) in Binärform kompatibel zusammen arbeiten.

Klienten sollen maximal portierbar sein, und ohne Änderungen auf Rechnern/Betriebssystemen unterschiedlicher Hersteller lauffähig sein.

Klienten sollen keine Kenntnis benötigen wie ein Objekt auf einem Server implementiert ist.

16.1.6 Objekt orientierter RPC

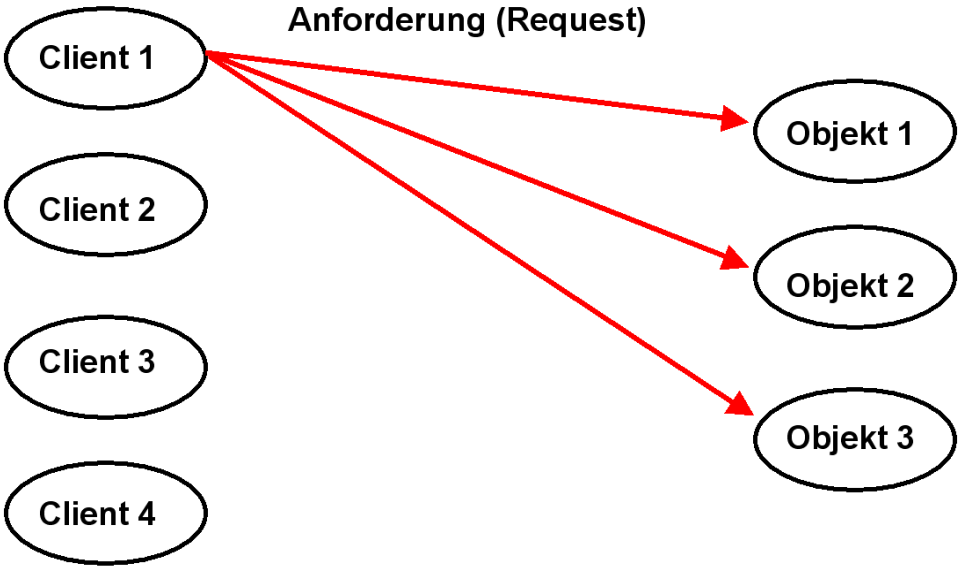

Abb. 16.1.4
In einem Objekt-orientierten RPC ersetzten Server Objekte die Services des Servers

In einem Objekt-orientierten RPC ist der Service (Dienstleistung eines Servers) als Objekt implementiert. Der Service wird in Anspruch genommen, indem die Methoden des Objekts aufgerufen werden.

Ein Klient ist eine Software-Einheit, die eine Operation (eine Methode) eines Objektes aufruft

Problematisch ist, dass Objekte in unterschiedlichen Sprachen implementiert werden, aber in binärer Form auf dem Server installiert sind. Die binäre Implementierung eines Client Objects ist ohne Anpassung nicht in der Lage mit der binären Implementierung eines Server Objects zu kommunizieren.

16.1.7 Kommunikations-Alternativen für den Objekt-orientierten RPC

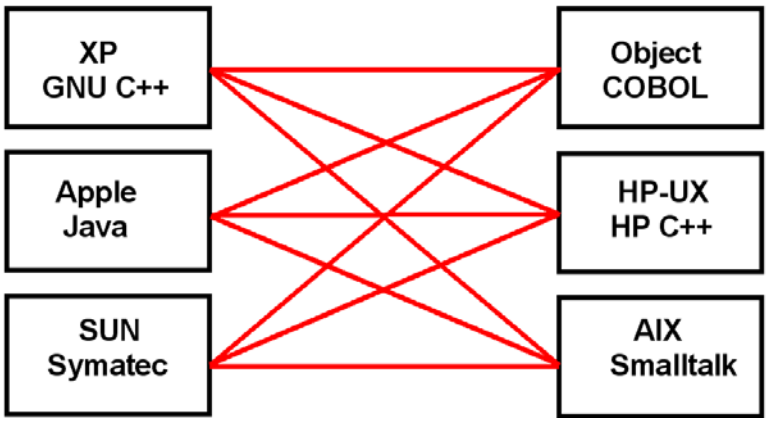

Abb. 16.1.5
Any to any Kommunikation

Ohne zusätzliche Vorrichtungen muss die Kommunikation jedes Klienten an jedes Server Objekt je nach verwendeter Sprache und darunterliegender Platform getrennt angepasst werden.

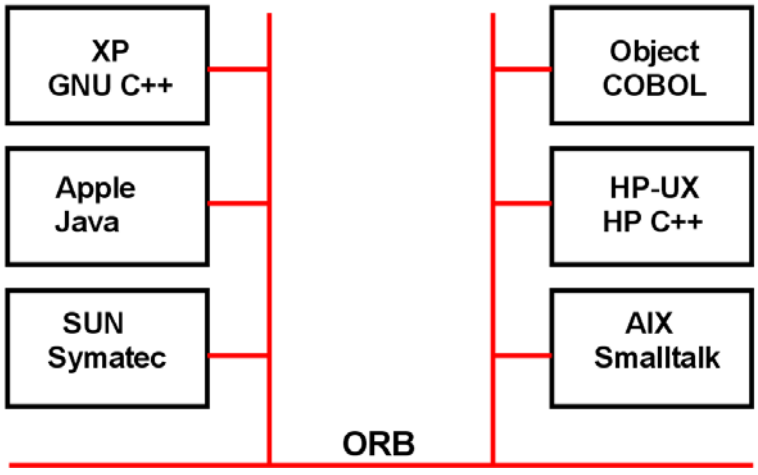

Abb. 16.1.6
Der ORB vereinheitlicht die Kommunikation

Unter Benutzung eines Object Request Brokers (ORB) können in unterschiedlichen Programmiersprachen entwickelte binäre Objekte plattform-unabhängig miteinander kommunizieren.

16.1.8 Object Request Broker

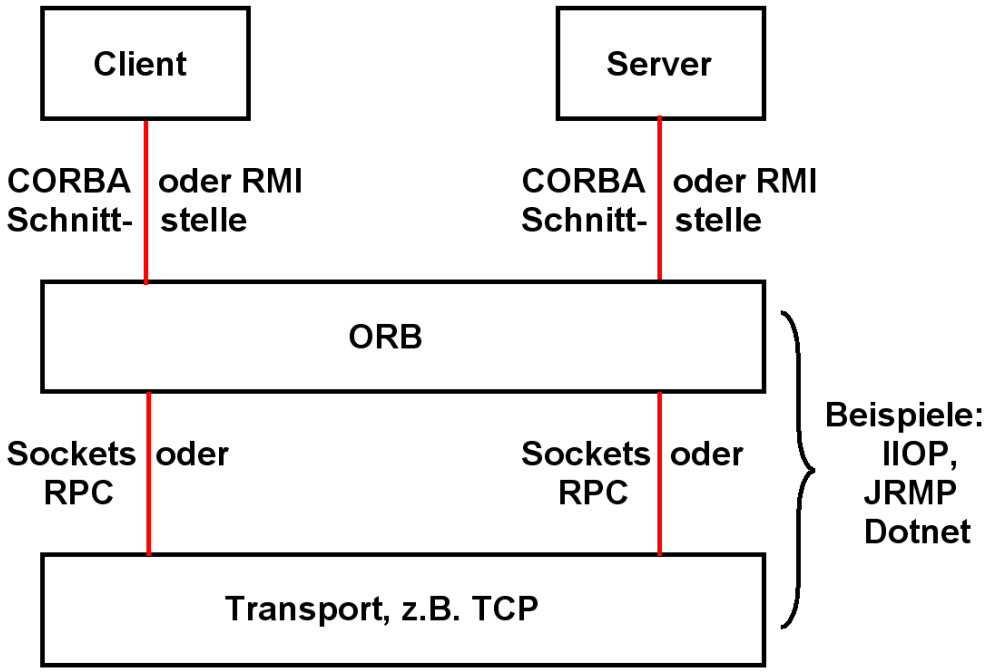

Abb. 16.1.7
Objektorientiertes Client/Server-System

In einem Objekt orientierten Client/Server System wird eine „Object Request Broker“ (ORB) Schicht zwischen dem Client und Server auf der einen Seite, und der Transport Schicht auf der anderen Seite zwischen geschaltet.

Ein ORB (Object Request Broker) ist eine Komponente eines Betriebssystems. Er ermöglicht es, dass Objekte, die in unterschiedlichen Programmiersprachen entwickelt wurden, in binärer Form miteinander zusammenarbeiten. Dies kann über Prozess- und physische Rechnergrenzen hinweg geschehen.

Hierfür stellt der ORB ein einheitliches Klassenmodell sowie Standards für die Organisation von Klassen-Bibliotheken und die Kommunikation zwischen Objekten zur Verfügung.

Es existieren mehrere Alternativen einen ORB zu implementieren:

- **CORBA Standard der OMG (Open Management Group),**
- **Remote Method Invocation (RMI), Teil des Java JDK der Fa. SUN**
- **DotNet der Fa. Microsoft.**

In eine ähnliche Richtung geht die Internet orientierte SOAP / UDDI / WSDL Initiative.

16.1.9 Interoperable Object Reference, IOR

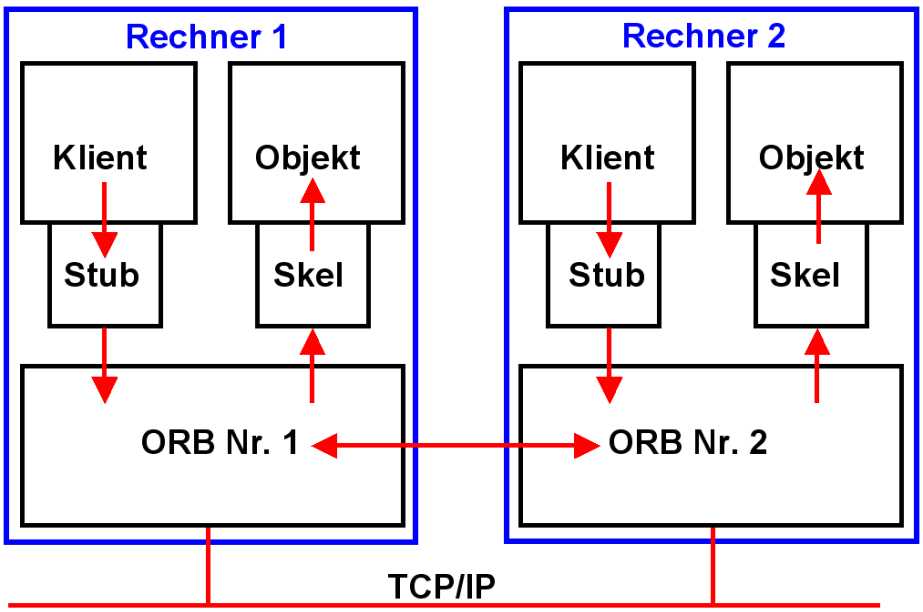

Abb. 16.1.8
Stubs und Skeleton Schnittstellen mit der lokalen ORB Komponente

Der ORB stellt eine Schnittstelle zwischen Client und Server dar, die ähnlich der Socket- oder RPC-Schnittstelle arbeitet, aber

- **auf einer höheren Ebene arbeitet (und deshalb evtl. Sockets oder RPC für die eigentliche Kommunikation verwendet,**
- **objektorientiert arbeitet,**
- **für die Kommunikation das IIOP oder das JRMP Protokoll verwendet (oder andere).**

Klienten und Server Objekte kommunizieren mit ihrem ORB über Stubs und Skeletons. Sie können transparent auf dem gleichen oder auf entfernten Rechnern arbeiten.

Eine eindeutige Objekt Bezeichnung (16-8, IOR) wird dem Objekt bei der Entstehung zugeordnet. Der Client nimmt die Dienste eines Objektes in Anspruch, indem er es zunächst mit Hilfe seiner IOR lokalisiert, und dann einen Methodenaufruf ausführt. Der Methodenaufruf enthält:

- **IOR**
- **Methodennamen**
- **Parameter**
- **Kontext (enthält Information über die Position des Aufrufers und nimmt Fehlermeldungen entgegen)**

16.1.10 Inter ORB Protokolle

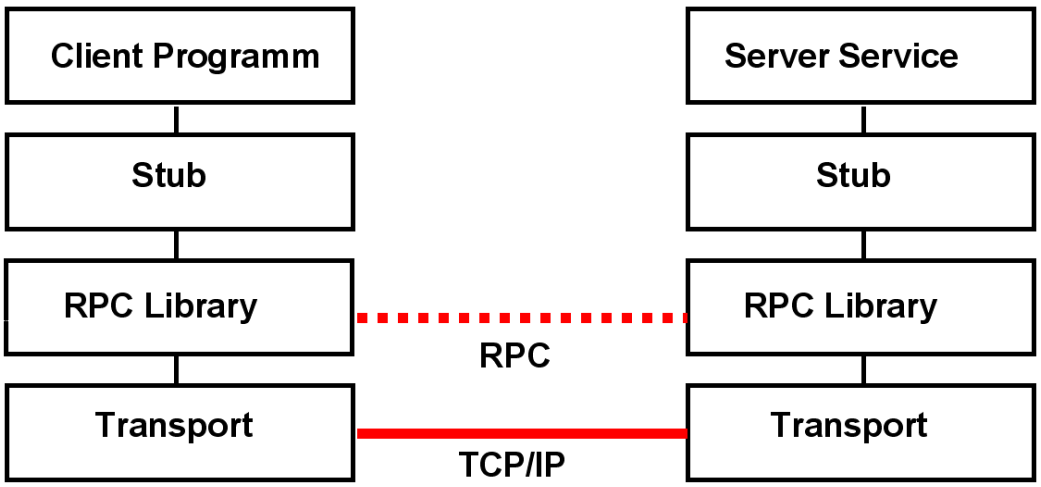

Abb. 16.1.9
RPC Library Kommunikation

Beim klassischen RPC kommuniziert die RPC Library des Klienten mit der RPC Library des Servers mittels eines RPC spezifischen Protokolls

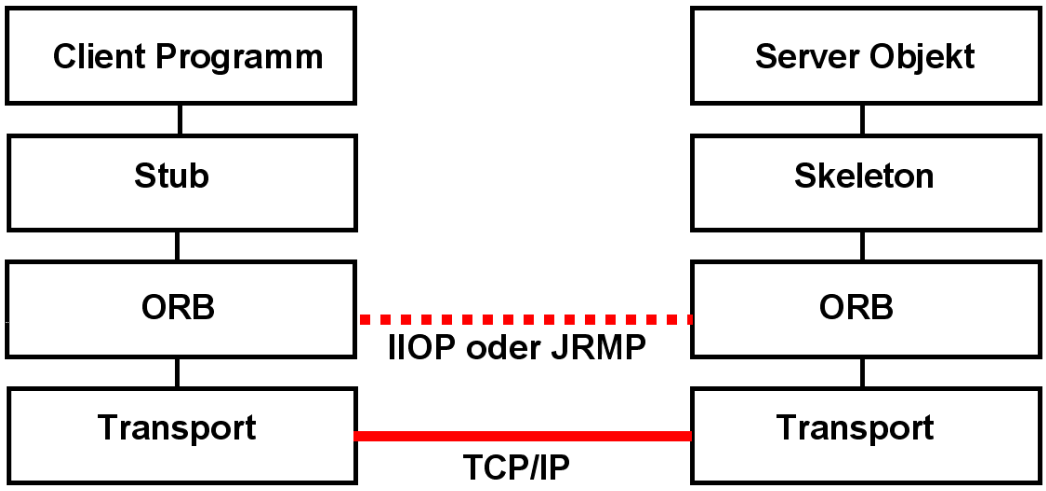

Abb. 16.1.10
ORB Kommunikation

Beim objektorientierten RPC kommuniziert die ORB Komponente des Klienten mit der ORB Komponente des Servers mittels der IIOP oder JRMP Protokolle.

IIOP (Internet Inter-ORB Protokoll) und JRMP (Java Remote Method Protokoll) sind auf der gleichen Ebene einzuordnen wie die http, SOAP, 3270 oder FTP Protokolle.

16.1.11 Integration von unterschiedlichen Object orientierten Client/Server Protokollen

Es existieren drei bedeutende objektorientierte Client/Server Standards:

- **CORBA**
- **JEE Remote Method Invocation (RMI)**
- **DotNet von Microsoft**

CORBA (Common Object Broker Architecture) ist ein Standard der OMG (Open Management Group). Die OMG ist ein 1989 gegründeter, internationaler, nicht profit-orientierter Zusammenschluss von zunächst 8 (Anfang 1996: Ca. 600) Softwareentwicklern, Netzbetreibern, Hardwareproduzenten und kommerziellen Anwendern von Computersystemen (ohne Microsoft).

CORBA unterstützt viele Sprachen, darunter C/C++, COBOL, PLI, ADA und Java. CORBA Produkte werden von zahlreichen Herstellern implementiert. Für alle von CORBA unterstützten Sprachen existiert eine einheitliche Interface Definition Language (IDL). IDL ist der OMG Standard um Sprach-neutrale APIs zu definieren. IDL ermöglicht eine Platform-unabhängige Beschreibung der Interfaces von verteilten (distributed) Objekten.

Der RMI-Standard bietet zwei alternative Protokolle: JRMP und RMI/IIOP. RMI/IIOP wurde entwickelt, um CORBA und RMI miteinander zu verbinden. Einige Hersteller von JEE Software bieten beide Alternativen in ihren Produkten an. IBM hat sich entschieden, ausschließlich RMI/IIOP einzusetzen, was konform mit dem RMI Standard ist.

Die Koexistenz Charakteristika von Corba und RMI sind sehr gut.

Auf der anderen Seite sind die Microsoft DotNet Produkte sehr unvereinbar mit der Corba/RMI Welt. Es ist möglich, eine Kommunikation zwischen beiden Alternativen mittels ORB Brücken zu erreichen. Allerdings ist die Leistung nicht optimal und die Integration erfordert Expertenwissen. Aus diesem Grund ist DotNet bei größeren Unternehmen, die Java häufig verwenden, nicht sehr beliebt. Es wird vor allem in kleineren Unternehmen eingesetzt, die hauptsächlich Microsoft Server Produkte einsetzen.

Web Services und ihr SOAP RPC sind eine moderne Alternative zu RMI und Corba.

16.1.12 RMI

Remote Method Invocation (RMI, deutsch etwa „Aufruf entfernter Methoden“), ist der Aufruf einer Methode eines entfernten Java Objekts und realisiert die Java-eigene Art des Remote Procedure Calls. „Remote“ bedeutet dabei, dass sich das Objekt in einer anderen Java Virtual Machine befinden kann, die ihrerseits auf einem entfernten Rechner oder auf dem gleichen lokalen Rechner laufen kann. Dabei sieht der Aufruf für das aufrufende Objekt (bzw. dessen Programmierer) genauso wie ein lokaler Methoden Aufruf aus; es müssen jedoch besondere Ausnahmen abgefangen werden, die zum Beispiel einen Verbindungsabbruch signalisieren können.

Auf der Client-Seite kümmert sich der Stub um den Netzwerktransport. Vor dem Erscheinen der Java Standard Edition (JSE) in Version 1.5.0 war der Stub eine mit dem RMI-Compiler rmic erzeugte Klasse. Seit der Version 1.5 ist es nicht mehr notwendig, den RMI-Compiler aufzurufen. Das Erstellen des Stubs wird von der Java Virtual Machine übernommen.

Für die erste Verbindungsaufnahme werden aber die Adresse des Servers und ein Bezeichner (z.B. eine RMI-URL) benötigt. Für den Bezeichner liefert ein Namensdienst auf einem Server eine Referenz auf das entfernte Objekt zurück. Damit dies funktioniert, muss sich das entfernte Objekt im Server zuvor unter diesem Namen (IOR oder String Name) beim Namensdienst registriert haben. Der RMI Namensdienst wird über statische Methoden der Klasse java.rmi.Naming angesprochen. Der Namensdienst ist als eigenständiges Programm implementiert. Es existieren zwei Arten des Namensdienstes, der RMI Registry für lokale Netze und der „Java Naming and Directory Service“ (JNDI) für weltweite Netze.

Eine Client/Server Anwendung, bei der sowohl der Client als auch der Server in Java programmiert sind, kann sowohl RMI als auch eine andere RPC Art verwenden (z.B. CORBA oder DCE). RMI ist aber ein besonders einfaches Verfahren, Client/Server Anwendungen zu erstellen.

16.2 RMI

16.2.1 Java Method Invocation

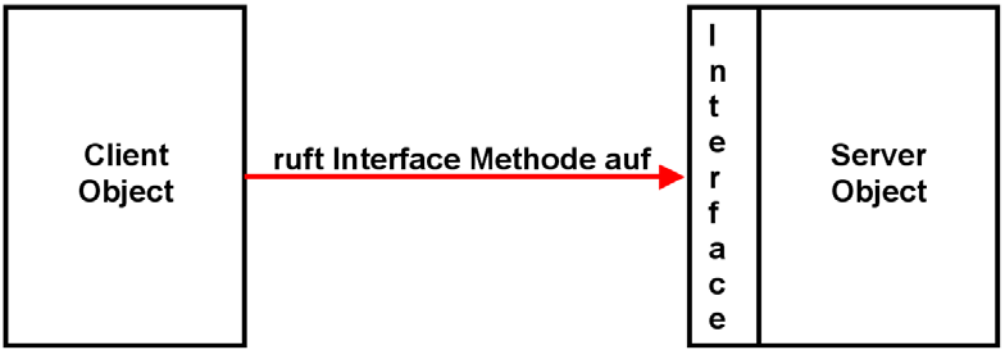

Abb. 16.2.1
Ein Java Client Objekt ruft eine Methode eines Java Server Objektes auf

Eine JVM kann viele Klassen (Objekte) speichern.

Eine Klasse kann mittels eines Methodenaufrufs eine Methode einer anderen fremden Klasse aufrufen. Der Methodenaufruf überträgt die Aktivität auf das durch eine Referenz angegebene (fremde) Objekt (Klasse).

Bei einem lokalen Aufruf ruft der Klient eine Methode (von potentiell mehreren) auf, die in der Interface des aufgerufenen Objektes beschrieben ist.

Was passiert beim Methodenaufruf in einer Zeile wie

```
String datei=gibtWertfuerParameter("-datei",args);
```

Auf der rechten Seite steht ein Funktions- oder Methodenaufruf. Ein Methodenaufruf ähnelt in seiner Form der ersten Zeile einer Methodendefinition: Auf einen Namen folgt ein Paar runder Klammern. Zwischen den Klammern können Parameter stehen (es gibt auch Methoden ohne Parameter).

In der großen Mehrzahl der Fälle erfolgen Java Methodenaufrufe nur innerhalb einer einzelnen JVM (lokaler Methodenaufruf). Beim lokalen Methodenaufruf rufen Sie eine Methode einer Klasse auf, die sich in der gleichen JVM befindet. Mit Hilfe der Remote Method Invocation (RMI) sind Objekte in einer bestimmten (lokalen) JVM in der Lage, Methoden von Objekten in einer entfernten JVM aufzurufen. Beim entfernten (remote) Methodenaufruf kann sich die angesprochene JVM entweder auf dem gleichen Rechner wie die aufrufende JVM befinden (meistens in einem anderen Address Space), oder sie kann sich auf einem beliebigen anderen Rechner im Internet befinden. Syntax und Semantik des Methodenaufrufs sind in beiden Fällen identisch; die JVMs managen die Unterschiede.

RMI ist ein Java spezifischer entfernter Methodenaufruf, ähnlich zum klassischen RPC (Remote Procedure Call), CORBA (Common Object Request Broker Architecture) RPC und dem Web Services (SOAP) RPC.

16.2.2 Arten des Methoden Aufrufs

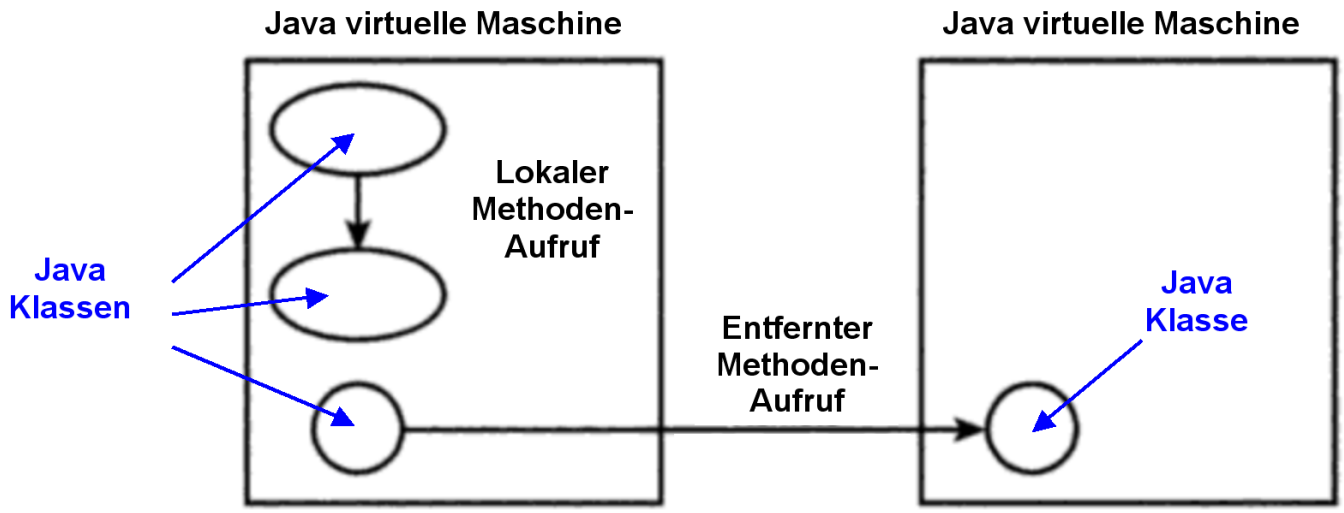

Abb. 16.2.2
Der entfernte Methodenaufruf erfolgt mittels des RMI Protokolls

Der entfernte Methodenaufruf, Remote Method Invocation (RMI) ist einer der Eckpfeiler von Enterprise JavaBeans und eine ausgesprochen praktische Möglichkeit, verteilte Java-Anwendungen zu erstellen.

Lokale und entfernte Methodenaufrufe haben unterschiedliche Performance-Eigenschaften

- **Der Zugriff auf ein Remote Object erfordert die Erstellung von Stubs und Skeletons. Dies braucht CPU Zyklen.**
- **Bei der Übertragung von Variablen bestehen Performance Unterschiede. Unterschiedliche Arten von Daten haben einen unterschiedlichen Marshalling Overhead. Marshalling (von engl. to marshal, aufstellen, anordnen) ist das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übertragung über das Netz und die Übermittlung an andere Prozesse ermöglicht.**

16.2.3 Remote Method Invocation

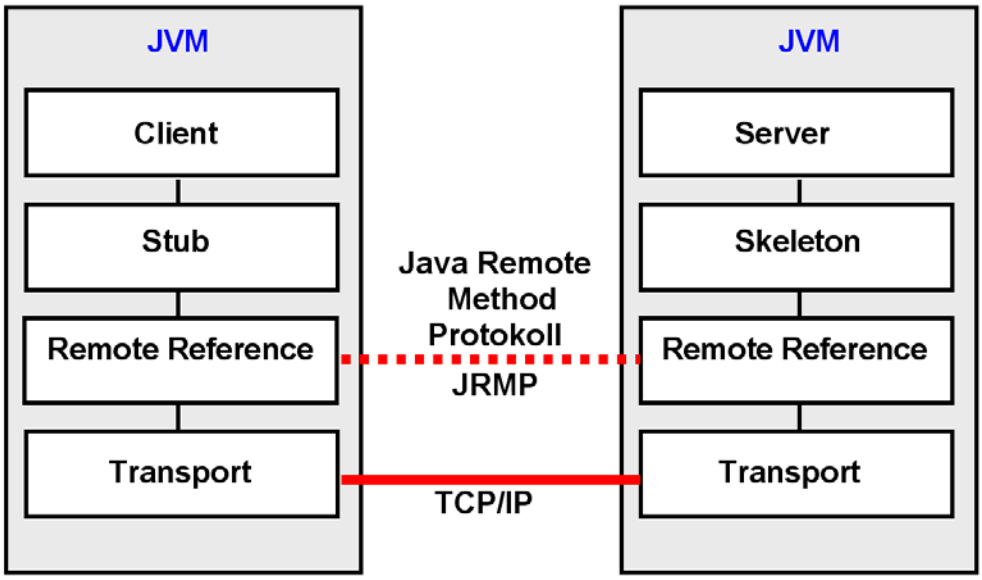

Abb. 16.2.3
JRMP ist das ursprüngliche RMI Protokoll

Remote Method Invocation (RMI) ist eine Client-Server-API für den Aufruf von Java Programmen auf geographisch entfernten Rechnern

Unter Benutzung von RMI können Objekte einer bestimmten JVM die Methoden von Objekten in einer entfernten JVM aufrufen.

Zur Realisierung wird ein Stellvertreter (Client-Stub) des entfernten Objekts in der lokalen JVM erzeugt. Dieser kommuniziert mit einem Stellvertreter (Server-Skeleton) des entfernten Objektes in dessen JVM.

16.2.4 RPC Server und RPC Service

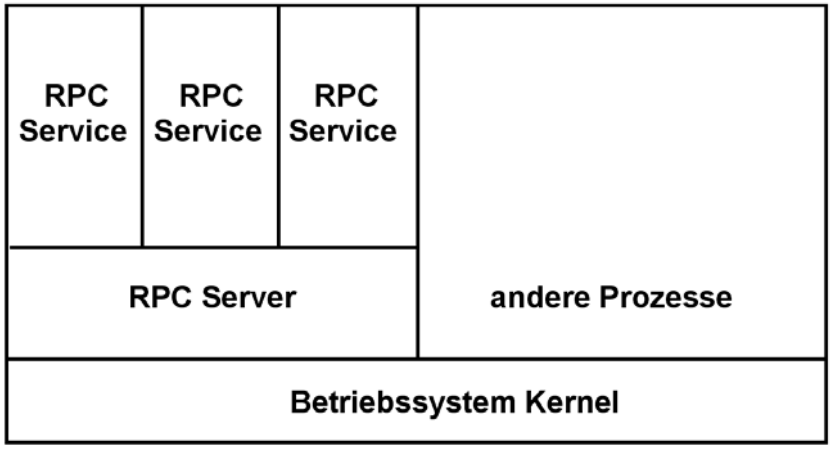

Abb. 16.2.4
RPC Server als Laufzeitumgebung für RPC Services

Eine aufgerufene Prozedur wird als RPC-Service (oder Implementation) bezeichnet. Ein RPC-Server bietet in der Regel viele unterschiedliche Arten von RPC Services an. Die RPC Services (RPC Dienstprogramme) können entweder in getrennten virtuellen Adressenräumen laufen, oder alternativ als Threads innerhalb eines einzigen virtuellen Adressenraums implementiert werden. In CICS ist der CICS Nucleus der RPC-Server; einzelne Transaktionen, die durch ihre TRID gekennzeichnet sind, sind die RPC Services.

Der RPC-Server liefert eine Laufzeitumgebung für die RPC-Services. Er stellt Verwaltungsdienste für seine RPC Services zur Verfügung, z.B.:

- **Registry Funktionen,**
- **Binding (Address Resolution),**
- **Sicherheits-Funktionen,**
- **Kommunikations-Funktionen, usw.**

Die RPC Services können in beliebigen Sprachen implementiert werden, auch als Java Objekte.

Ein physischer Rechner kann mehrere RPC-Server beherbergen, von denen jeder eine andere Laufzeitumgebung für seine RPC-Services zur Verfügung stellt. Ein Beispiel ist ein z/OS-System, welches einen CICS Server, einen DCE-Server mit relaxten Sicherheitsmerkmalen, und einen weiteren DCE-Server mit strengen Sicherheitsanforderungen enthält.

Häufig laufen Hunderte oder Tausende unterschiedlicher RPC Services auf dem gleichen Rechner.

Java RPC Services können als Threads innerhalb einer JVM implementiert werden. Alternativ kann ein Java RPC-Server auch mehrere JVMs, jeweils für Gruppen von Java Services, unterhalten.

16.2.5 Home Banking Beispiel

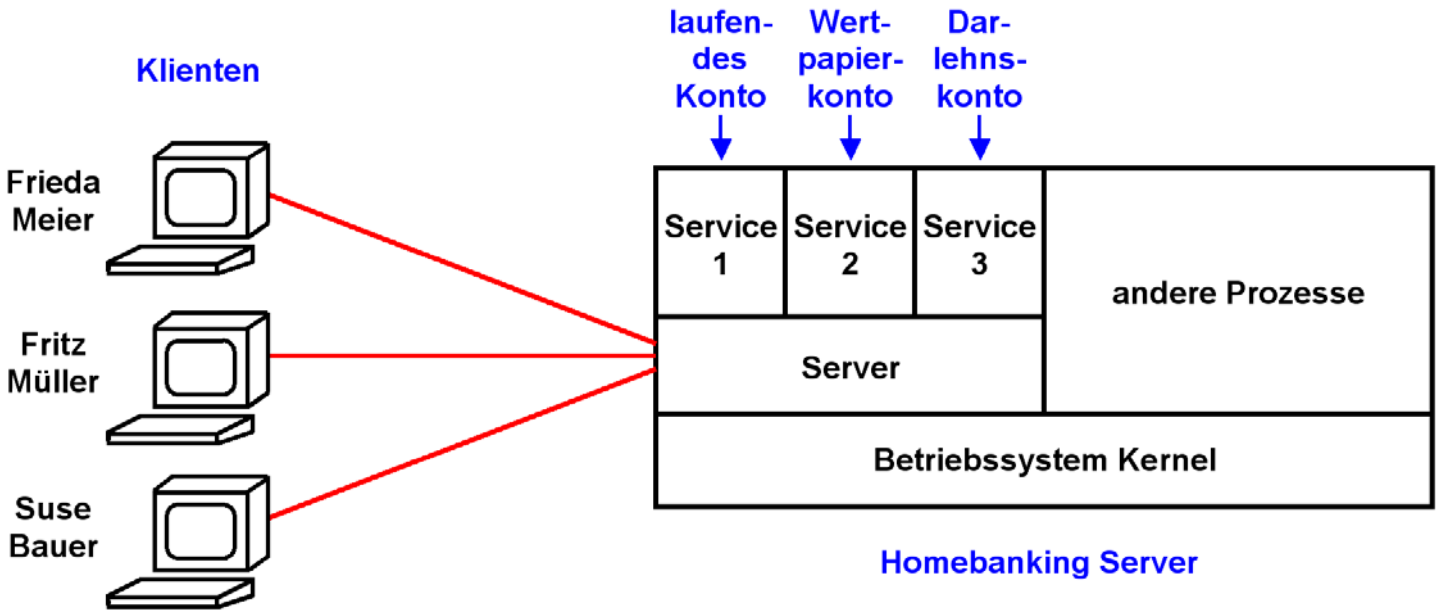

Abb. 16.2.5
Ein Service kann von mehreren Klienten gleichzeitig in Anspruch genommen werden

Dies sei an Hand eines Homebanking Beispiels erläutert. Angenommen, Klienten und Server sind in Java implementiert. Die drei Kunden Frieda Meier, Fritz Müller und Suse Bauer unterhalten bei einer Bank je ein laufendes Konto, ein Wertpapierkonto und ein Darlehnskonto. Auf ihren PCs ist ein Java Client in einer JVM installiert. Auf dem Server laufen drei Services als drei getrennte Prozesse, jeweils mit einer eigenen JVM. Services werden von den Klienten mittels RMI aufgerufen. Service 1 implementiert das laufende Konto mit Methoden wie kontostandabfragen, geldüberweisen, dauerauftragstornieren usw. Service 2 implementiert das Wertpapierkonto mit Methoden wie wertpapierkaufen, wertpapierverkaufen, aktienkursentwicklung usw. Service 3 implementiert das Darlehnskonto mit Methoden wie tilgungsstatus abfragen, sondertilgungvornehmen, usw.

Der Server implementiert für seine Services Dienstleistungen wie Authentification, Load Balancing, Errror Recovery, Namensdienste, Transaktionsdienste, Journalling usw.

16.2.6 Threads auf der Server-Seite

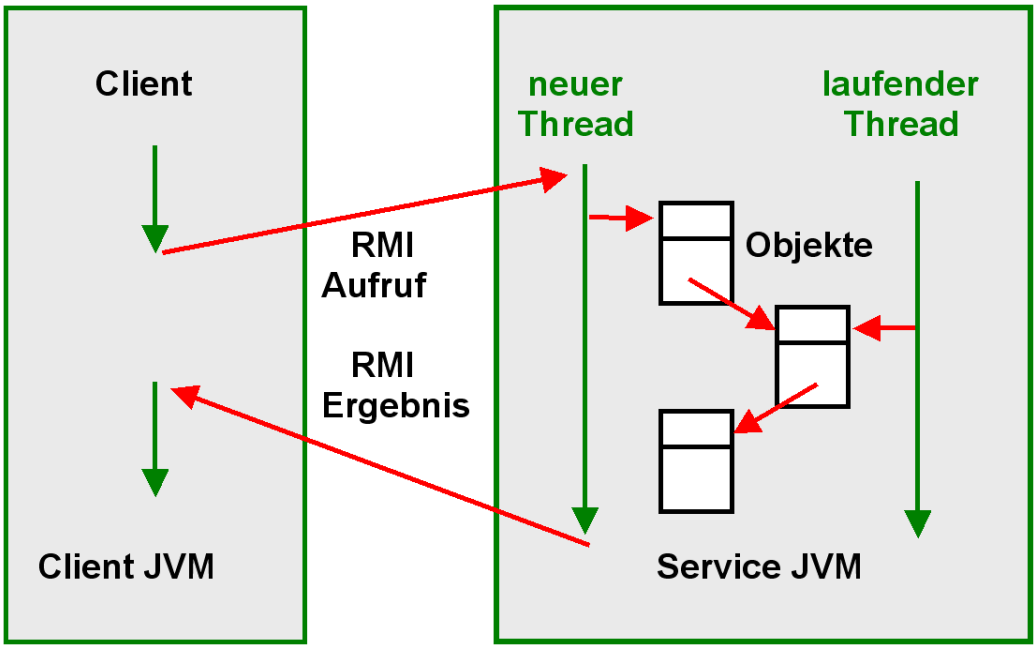

Abb. 16.2.6
Parallele Verarbeitung durch mehrere Java Threads

Ein Client ruft einen von mehreren Services mittels RMI auf. Die Services sind als Java Klassen (Java Objekte) implementiert. Jeder RMI-Aufruf erzeugt auf der Server-Seite einen neuen Thread. Jeder Thread ist eine Instanz der Service Klasse, ein Java Object. Das bedeutet, dass in der Server JVM mehrere solcher Threads gleichzeitig laufen können, zusammen evtl. mit weiteren dauerhaft laufenden Threads des Servers (z.B. Garbage Collector, Compute-Server,...). Wenn Frieda Meier und Fritz Müller gleichzeitig ihr laufendes Konto abfragen, wird also für beide je ein Java Thread mit den Objekt Variablen Frieda Meier und Fritz Müller angelegt.

Man muss sich also in jedem Fall Gedanken über eine notwendige Synchronisation (Concurrency) machen (mit synchronisierter Blockierung, ggf. auch mit wait() und notify()), um ein Zugriff mehrerer Threads auf gemeinsam genutzte Daten zu steuern. Auch wenn keine vollständigen ACID Eigenschaften vorhanden sind, dürfen die Threads sich nicht gegenseitig beeinflussen.

16.2.7 String Name und Objekt Referenz

In unserem Beispiel haben die Klienten die Option, auf einen von mehreren Services zuzugreifen: Laufendes Konto, Wertpapier Konto oder Darlehnskonto. Klienten selektieren den Service mit Hilfe des Namens des Services, z.B. laufendeskonto. Namen sind üblicherweise Zeichenketten (Strings), weshalb auch die Bezeichnung „String Name“ gebräuchlich ist.

Ein bestimmter Service ist gekennzeichnet durch zwei Bezeichner: Seinen Namen sowie die Adresse, über die er erreichbar ist. Wenn in unserem Beispiel jeder Prozess über eine eigene IP Adresse erreichbar ist, könnte dies z.B. die IP Adresse 134.2.205.55, Port Nr. 2012 sein. Diese Adressse wird als Objekt Referenz bezeichnet.

Wenn ein Klient auf einen Service zugreift, ist es daher erforderlich, den Namen in die Objekt Referenz zu übersetzen. Dies geschieht mit Hilfe eines getrennten Server Prozesses. Hier existieren zwei Alternativen:

- **Registry**
- **Namens- und Verzeichnis Dienstes**

Eine Registry ist ein einfacher Namensdienst. Er läuft als getrennter Prozess auf dem gleichen physischen Server, auf dem auch die Services laufen, und ist typischerweise auf eine LAN Umgebung begrenzt. Ein Namens- und Verzeichnisdienst (Naming and Directory Service) ermöglicht es, diesen (und auch die Services) auf physisch unterschiedlichen Servern unterzubringen, die sich irgendwo im weltweiten Netz befinden.

16.2.8 Remote Method Invocation

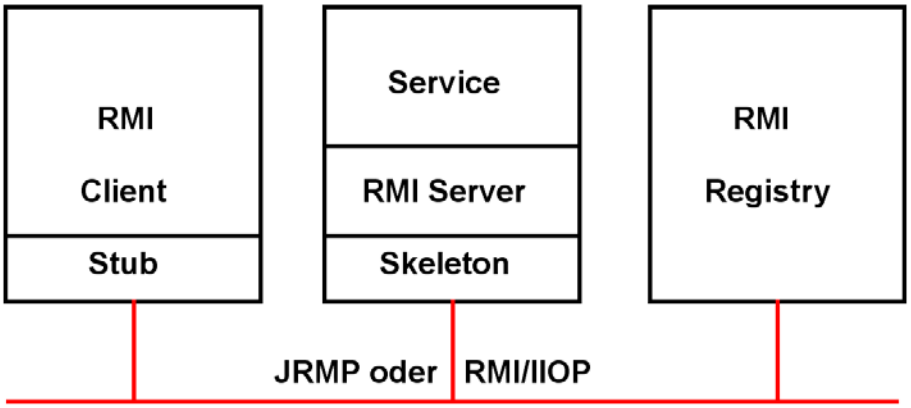

Abb. 16.2.7
Die Namensauflösung erfordert ein Registry

RMI erfordert drei verschiedene Prozesse, die auf dem gleichen Rechner oder auf entfernten Maschinen laufen können: Klient, Server und Namensdienst. RMI Registry ist ein einfacher RMI Namensdienst. Die Alternative ist JNDI

- **Der Klient besorgt sich eine Referenz (Handle) für das entfernte Objekt, indem er RMI Registry aufruft.**
- **Eine Referenz auf das entfernte Objekt wird zurückgegeben. Jetzt kann eine Methode des entfernten Objektes aufgerufen werden.**
- **Dieser Aufruf erfolgt zum lokalen Stub, der das entfernte Objekt repräsentiert.**
- **Der Stub verpackt die Argumente (Marshalling) in einen Datenstrom, der über das Netzwerk geschickt wird.**
- **Das Skeleton unmarshals die Argumente, ruft die Methode des RMI Services (Implementation) auf, marshals die Ergebniswerte und schickt sie zurück.**
- **Der Stub unmarshals die Ergebniswerte und übergibt sie an das Klientenprogramm.**

16.2.9 Zeitlicher Ablauf der RMI Remote Class Programmausführung

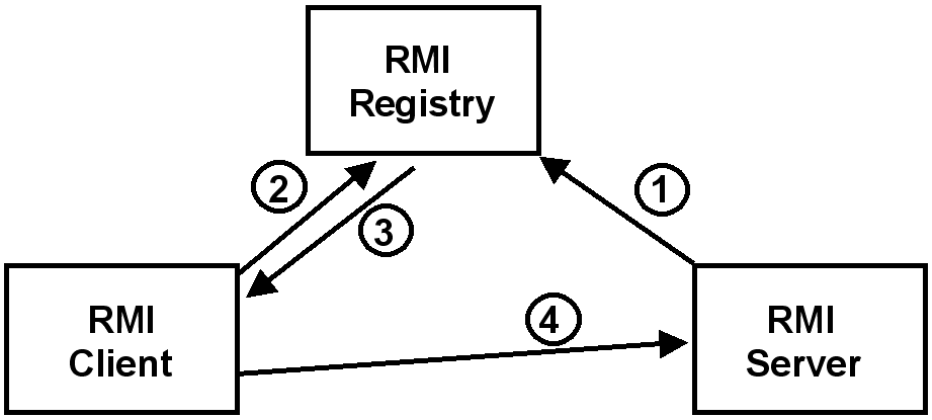

Abb. 16.2.8
Ein Server registriert jeden neuen Service in seinem Registry

Der RMI Server speichert zahlreiche Java Server Objekte (Services), auf die ein RMI Client zugreifen möchte. Hierzu:

1. **RMI Server speichert Objekt Namen und dazugehörige Objekt Referenz im RMI Registry Server. (Dieser läuft im Gegensatz zu anderen JNDI Implementierungen auf dem gleichen physischen Server wie der RMI Server).**
2. **Client verbindet sich mit dem Registry Server, und übermittelt den Objekt Namen (String Name) des gewünschten remote Objektes.**
3. **Registry stellt die Objekt Reference (Adresse) auf das remote Objekt zur Verfügung.**
4. **Zugriff auf den RMI Service.**

Ein Objekt Name ist vergleichbar mit einer URL wie z.B.
http://leia.informatik.uni-leipzig.de

Eine Objekt Referenz ist vergleichbar mit einer dazugehörigen IP Adresse wie z.B.
139.18.4.30

16.2.10 Registry Alternativen

Ein Namensdienst verwaltet Zuordnungen von Namen zu bestimmten Objekten. Alle Objekte werden nach einem standardisierten Namen angesprochen.

Ein Verzeichnisdienst (Directory Dienst) ist eine für Lese-Zugriffe optimiertet Datenbank, die Informationen in einem hierarchischen Informationsmodell speichert. Die in einem Namensdienst definierten Namen können in einem Verzeichnisdienst gespeichert werden; ein Verzeichnis Dienst ist eine Art Datenbank, die im Gegensatz zu einem Namensdienst weitere Informationen speichern kann, z.B. die geografische Lokation eines Services oder Servers.

Ein Namens- und Verzeichnisdienst ermöglicht es, diesen (und auch die Services) auf physisch unterschiedlichen Servern unterzubringen, die sich irgendwo im weltweiten Netz befinden.

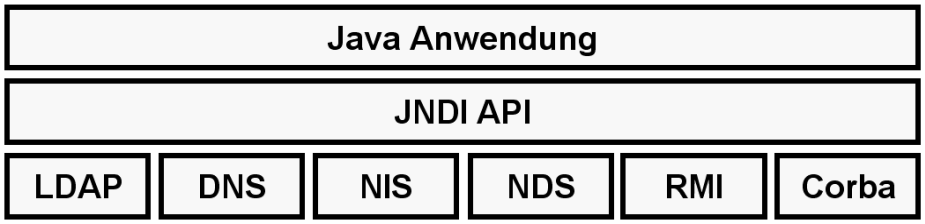

Abb. 16.2.9
Hinter der JNDI können sich beliebige Verzeichnisdienste verbergen

JNDI (Java Naming and Directory Interface) ist eine API, über die ein Java Programm einen Namens- und Directory Dienst in Anspruch nehmen kann. Der Namens- und Directory Dienst ermöglicht einen Lookup von RMI Objekten (Services) zwischen entfernten JVMs. Mögliche Implementierungen der JNDI Schnittstelle sind z.B. z.B. LDAP Server, Corba Common Object Services Naming (COSNaming) Server oder der rmiregistry Server. rmiregistry ist ein Bestandteil des JDK. JRMP verwendet standardmäßig einen rmiregistry Server.

Wie in Java RMI, erfolgt der Zugriff auf ein CORBA Server Objekt mit Hilfe einer Objekt-Referenz. Ein CORBA Server ORB ermöglicht das Registrieren einer Objekt-Referenz mit einem Corba konformen Verzeichnisdienst (Common Object Service (COS) Naming Server). Der CORBA COS (Common Object Services) Naming Service verfügt über eine Baum-artige Directory Struktur für Object Referencen.

Da CORBA im Gegensatz zu RMI sprachenabhängig ist, ist eine CORBA Objekt-Referenz eine abstrakte Entität. Diese wird von einem Client ORB in eine Sprachen-spezifische Objekt-Referenz abgebildet. CORBA-Objekt Referenzen werden als Interoperable Object References (IORs) bezeichnet.

Aus Kompatibilitätsgründen verlangt RMI/IIOP (siehe unten) die Benutzung von COS Naming für Zugriffe auf einen Corba Server.

16.2.11 Erstellen einer RMI Remote Class

Um entfernte Objekte mit ihren Methoden in Java-Programmen zu nutzen, müssen wir die folgenden Schritte durchführen:

1. **Wir definieren eine remote Schnittstelle eines geplanten Server Objektes, welche die Methode(n) des Server Objekts definiert.**

2. **Wir implementieren eine Klasse, die diese Schnittstelle implementiert und die Methoden mit Leben füllt. Dies bildet das entfernte Objekt (Service, Implementation)**

3. **Existiert die Implementierung, benötigen wir ein Exemplar dieses Objekts. Wir melden dieses bei einem Namensdienst an, damit andere es finden können. Dies bildet den Service.**

Wir implementieren einen Klienten und greifen damit auf die entfernte Methode zu.

RMI benötigt (wie Corba) einen RMI Server, unter dem die RMI Implementation (der Service) läuft.

Zu kodieren sind:

- **Interface**
- **Implementation**
- **Server**
- **Client**

Die Remote Interface enthält die Namen aller Methodenaufrufe und die dazugehörigen Parameter. Beispiel für die Interface einer Methode Addition, die zwei Zahlen a und b addiert:

```
public interface Addition extends
                java.rmi.Remote
{
     public long add(long a, long b)
     throws java.rmi.RemoteException;
}
```

Schritte zur Erstellung und Ausführung einer Anwendung:

1. **Interface definieren, mit der das Remote Object aufgerufen wird.**
   > extends interface java.rmi.Remote .

2. **Implementierung der Server Anwendung schreiben. Muss die Remote Interface implementieren. .**
   > extends java.rmi.server.UnicastRemoteObject

3. **Klassen kompilieren.**

4. **Mit dem Java RMI Compiler Client Stubs und Server Skeletons erstellen. > rmic xyzServerImpl**

5. **Klient implementieren und übersetzen.**

6. **Start Registry, Server starten, Klienten starten.**

16.2.12 RMI Performance

Table 1: Passing bytes by value

Function}	# of args	# of results	Local Call(sec)	Remote Call(sec)
Null	0	0	15.172	17.345
sendbyte	1	0	15.332	17.054
recvbyte	0	1	15.292	17.105

Table 2: Passing fixed length byte arrays

Function}	# of bytes (args)	# of bytes(results)	Local Call(sec)	Remote Call(sec)
senddata	1440	0	25.537	34.209
recvdata	0	1440	27.099	33.408

Abb. 16.2.10
RMI Performance Ergebnisse

RMI benötigt viele CPU Zyklen für die Ausführung. Die gezeigten Daten stammen von einer Untersuchung
http://www-csag.ucsd.edu/individual/achien/cs491-f97/projects/hprmi.html, oder
http://www.cedix.de/VorlesMirror/Band2/Perform02.pdf

Gemessen wurde die benötigte Zeit für 10 000 RMI Aufrufe zwischen 2 JVMs auf dem gleichen Rechner (hier als Local Call bezeichnet) und 2 JVMs auf 2 getrennten Rechnern verbundenen über ein Ethernet (Remote Call). Auf den Rechnern lief Windows NT; die Messungen erfolgten 1997.

Die Zeit pro Aufruf liegt im Millisekunden Bereich. Die Schlussfolgerung ist: "Java RMI performs poorly in comparison to other RPC systems" (z.B. DCE RPC).

Auch wenn heutige Rechner deutlich schneller sind, ist der Overhead beträchtlich. Im Internet existieren zahllose Untersuchungen und Vorschläge zum Thema RMI Performance Tuning – nicht ohne Grund.

Dennoch ist Java RMI ein bequemes und populäres Verfahren, um Remote Method Calls in verteilten (distributed) Object Systemen zu implementieren. Auch ist die RMI Performance deutlich besser als die Web Service RPC Performance, see:
„Comparison of Performance of Web services, WS-Security, RMI, and RMI–SSL". Journal of Systems and Software 79 (2006) 689–700.
http://www.cedix.de/VorlesMirror/Band2/Perform03.pdf

16.2.13 DCE RPC mit Java Klienten ohne Objektorientierung

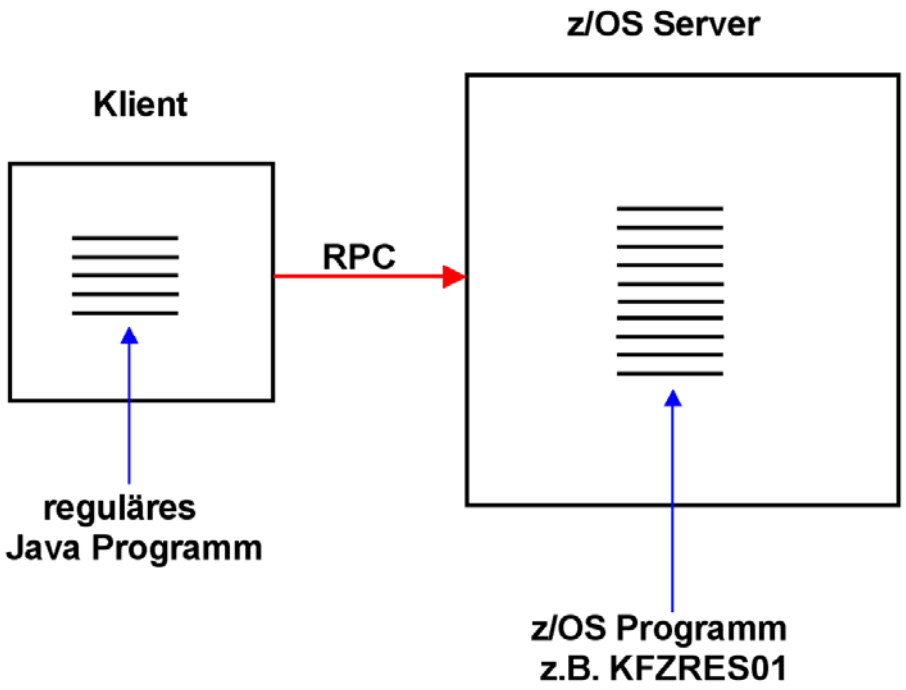

Abb. 16.2.11
Eine Java zu Java Kommunikation kann auch über den klassischen RPC erfolgen

Klassische Remote Procedure Calls (RPC) sind der Sun und der DCE Remote Procedure Calls.

Corba, RMI und Web Services (SOAP) sind moderne Versionen des RPC. Corba und RMI sind inhärent objekt-orientiert.

Es muss nicht alles RMI sein. Man kann auch Client/Server Anwendungen in Java schreiben ohne Benutzung von RMI. Hierbei gehen die Vorteile der Objektorientierung allerdings verloren.

Gezeigt ist ein Beispiel, wo ein Java Programm einen z/OS Service unter Benutzung des klassischen Remote Procedure Calls aufruft.

Der DCE RPC ist der bevorzugte klassische RPC unter z/OS (und auch Windows). Die DCE Software ist Bestandteil von z/OS.

16.2.14 Aufruf eines z/OS Programms durch einen Java Klienten

```
/** *Folgende Methode ruft einen RPC
     auf dem Mainframe auf. */

private boolean rpcCall(String inputString,
                        StringBuffer outBuffer){
  RPC m_rpc =new RPC();
  try {
    //Mainframe braucht UserID und Password!
    m_rpc.setUser("k3216 ");
    m_rpc.setPwd("TESTPW ");
    //Name des aufzurufenden Programms setzen
    m_rpc.setRpcName("KFZRES01 ");
    //remote procedure call ausführen
    m_rpc.execute(inputString,outBuffer);
    return true;
  }
  catch (Exception e) {
    m_stateField.setText("Execute-Error:"
                         +m_rpc.getApiMessage());
    return false;
  }
}
```

Abb. 16.2.12
Client Programm für den Aufruf eines z/OS Service

Die Integration von z/OS Mainframe-Programmen in Java-basierte Client/Server-Systeme funktioniert prinzipiell recht einfach. In der folgenden Methode wird ein z/OS RPC-Programm namens „KFZRES01“ aufgerufen. Diesem Programm werden zwei Zeichenketten (Strings) übergeben, einer für die Eingabeparameter, der zweite für die durch KFZRES01 erzeugte Ausgabe.

Seitens der z/OS Mainframe-Programme ändert sich rein gar nichts gegenüber der Nutzung per 3270-Terminal. Das Programm KFZRES01 „weiß “also nicht, ob es durch ein Java-Programm benutzt wird oder über konventionelle z/OS Mainframe-Terminals aufgerufen wird.

16.3 RMI over IIOP

16.3.1 Warum RMI zusätzlich zu CORBA ?

Neben der reinen Java-Lösung RMI gibt es auf dem weiten Feld der Standards noch das komplexere CORBA.

RMI setzt voraus, dass Client und Server in Java geschrieben sind. Im Gegensatz zu RMI definiert CORBA ein großes Framework für unterschiedliche Programmiersprachen. Die Definition von CORBA durch die OMG (Object Management Group) geht in das Jahr 1991 zurück, also auf die Zeit vor RMI.

Die Frage nach dem Sinn von RMI ist also erlaubt. Die Antwort liegt in der Einfachheit und Integration von RMI.

Seit 2001 hat die Firma Sun RMI an den CORBA Standard angepasst. Die Stellvertreter-Objekte (Stubs, Skeletons) unterstützen mittlerweile nicht nur das Java eigene JRMP Protokoll, sondern zusätzlich auch das CORBA eigene Inter-ORB Protocol (IIOP). Diese Lösung heißt RMI/IIOP („RMI over IIOP“).

Mit RMI/IIOP lässt sich sowohl eine Verbindung zwischen zwei in Java geschriebenen Objekten, als auch eine Verbindung zwischen Java Objekten und nicht-Java Objekten herstellen.

Vergleich CORBA – RMI:

- **Corba Anwendungen können in vielen unterschiedlichen Sprachen geschrieben werden, solange für diese Sprachen ein „Interface Definition Language (IDL) mapping“ vorhanden ist. Dies ist der Fall für Cobol, PL/1, C/C++, Ada, Fortran, SmallTalk, Perl, Ruby, Python und viele weitere Sprachen. RMI in der JRMP Version ist auf Java beschränkt.**

- **Mit der IDL (Interface Definition Language) ist die Interface von der Implementierung sauber getrennt. Es können unterschiedliche Implementierungen unter Benutzung der gleichen Interface erstellt werden. RMI Anwendungen sind einfacher zu programmieren als Corba Anwendungen, weil die Notwendigkeit der IDL Definition entfällt. Die Interface ist bereits ein Sprachkonstrukt von Java.**

- **RMI ermöglicht serialisierbare Klassen. Code und Objekte können über das Netz übertragen werden (can be marshaled), solange der Empfänger über eine Java Virtuelle Maschine (JVM) verfügt. CORBA erlaubt keine Übertragung von Code oder Objekten; es können nur Datenstrukturen übertragen werden.**

- **Corba hat ein besseres Leistungsverhalten als RMI (keine Interpretation).**

Die Vorteile beider Ansätze lassen sich durch den Einsatz des RMI/IIOP Protokolls kombinieren.

16.3.2 Vor dem Erscheinen von RMI/IIOP

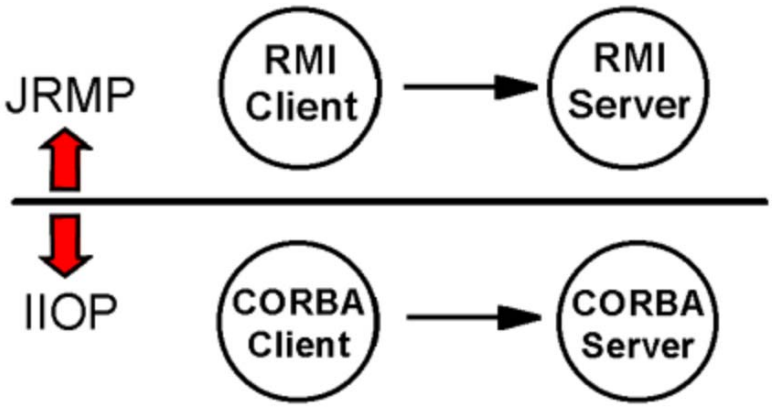

Abb. 16.3.1
Mit JRMP ist keine Kommunikation zwischen RMI und Corba möglich

Schauen Sie sich Abb. 16.3.1 an. Der Raum über der mittleren horizontalen Linie stellt die ursprünglichen Domäne von RMI und dessen JRMP Protokoll dar; der untere Bereich stellt die Welt von CORBA und IIOP dar. Diese beiden getrennten Welten, die unabhängig voneinander entwickelt wurden, sind historisch nicht imstande gewesen, miteinander zu kommunizieren. Zum Beispiel kann das native RMI Protokoll JRMP (Java Remote Method Protocol) sich nicht mit IIOP (dem Corba Protokoll), oder anderen Protokollen verbinden.

16.3.3 Interoperability mit CORBA

Wenn Java die einzige Programmiersprache ist, die Sie in einem neuen Projekt benötigen, können Sie RMI und JRMP (siehe Abb. 16.3.1) verwenden.

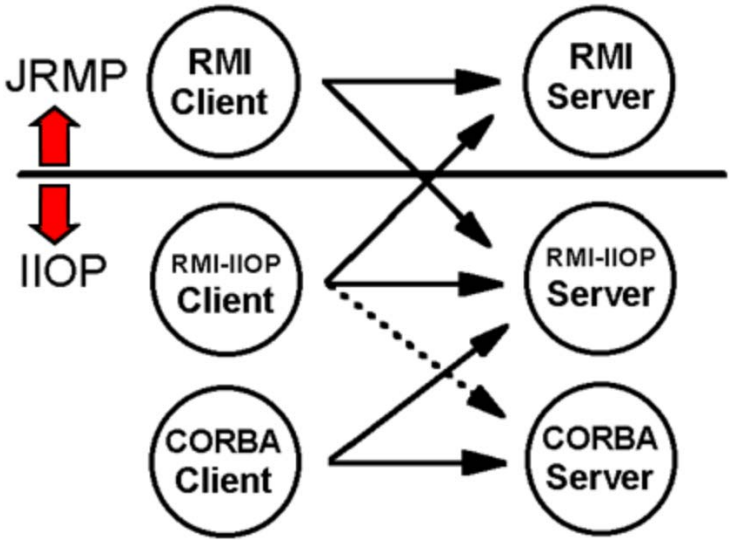

Abb. 16.3.2
Kommunikation zwischen RMI und Corba

CORBA auf der anderen Seite ermöglicht bei der Programmierung von Anwendungen mit verteilten Objekten die Benutzung unterschiedlicher Programmiersprachen. Verteilte Objekte werden nicht nur in neuen Projekten benutzt. Sie treten auch bei der Erweiterung älterer Software-Ressourcen (Legacy Software) auf. Legacy-Software ist in den meisten Fällen in anderen Sprachen als Java programmiert. In solchen Situationen müssen die Entwickler das Corba IIOP Protokoll, nicht RMI/JRMP einsetzen.

Um die komplexe Corba Programmierung zu vermeiden, können Java Programmierer an Stelle von JRMP das Corba kompatible RMI/IIOP Protokoll einsetzen.

In Abb. 16.3.2 stellt der obere Abschnitt der RMI/JRMP Modell, der mittlere Abschnitt der RMI/IIOP Modell, und der untere Abschnitt das CORBA Modell dar. Ein Pfeil stellt eine Situation dar, in der ein Client einen Server anruft. RMI/IIOP gehört in die IIOP Welt unterhalb der horizontalen Linie.

Interessant sind die diagonalen Pfeile, die die Grenze zwischen der JRMP Welt und der IIOP Welt überschreiten. Danach kann ein RMI/JRMP Client auf einen RMI/IIOP-Server zugreifen, und umgekehrt. RMI/IIOP unterstützt sowohl JRMP als auch IIOP Protokolle.

Eine Server Java Binary (d.h. eine Class File), die mit RMI/IIOP APIs erstellt wurde, kann entweder als JRMP oder als IIOP exportiert werden. Beim Wechsel von JRMP nach IIOP, oder umgekehrt, ist es nicht erforderlich, den Java-Quellcode umzuschreiben oder neu zukompilieren. Es ist nur erforderlich, Parameter wie Java System Properties zu ändern. Alternativ können Sie das zu benutzende Protokoll bestimmen, indem Sie es in dem Java-Quellcode spezifizieren.

Die diagonalen Pfeile in der Abbildung oben sind möglich, weil die RMI/IIOP APIs sowohl JRMP als auch IIOP Protokolle unterstützen. Dies bedeutet, dass ein RMI/JRMP Server Objekt ohne Umschreiben des Quellcodes durch einen neuen RMI/IIOP Client aufgerufen werden kann. Ebenso kann ein RMI/JRMP Server Objekt durch ein neues RMI/IIOP Objekt ersetzt werden, ohne Umschreiben des Quellcodes eines RMI/JRMP Clients.

Schauen wir wieder auf Abb. 16.3.2. Der Abschnitt unterhalb der horizontalen Linie ist die IIOP Welt, wo ein RMI/IIOP Client einen CORBA-Server aufruft und ein CORBA Client einen RMI/IIOP Server aufruft. Mit RMI/IIOP Client meinen wir ein Client-Programm, das von einem RMI-Programmierer, der nichts über CORBA oder IDL weis, geschrieben wurde. Ebenso ist ein CORBA-Client ein Client-Programm, das von einem CORBA-Programmierer ohne RMI Kenntnisse geschrieben wurde. Die Trennung der Interface von der Implementierung ist eine gut etablierte Technik. Sie ermöglicht es einem Programmierer auf verschiedene Ressourcen zuzugreifen ohne Wissen wie diese implementiert wurden. Benutzer sowohl von RMI/IIOP als auch von CORBA können die Dienste des anderen Protokolls verwenden , wenn sie Zugang zu seiner Interface bekommen. Ein RMI Java Interface-Datei ist die Interface für RMI/IIOP Benutzer, während IDL die Interface für CORBA Benutzer definiert. Die Interoperabilität zwischen RMI/IIOP und CORBA in der obigen Abbildung wird erreicht, indem jedem Benutzer seine erwartete Interface zur Verfügung gestellt wird.

Als letztes Detail in der obigen Abbildung ist der gepunktete Pfeil zu erklären. Er zeigt einen RMI/IIOP Client, der einen CORBA Server aufruft. Warum ist dieser Pfeil gepunktet ? Ein RMI/IIOP Client kann nicht unbedingt auf alle vorhandenen CORBA Objekte zugreifen. Die Semantik der in IDL definierten CORBA-Objekte kann Funktionen enthalten, die in RMI/IIOP Objekten nicht vorhanden sind. Eine bestehende CORBA Objekt IDL kann nicht immer in einem RMI/IIOP Java-Interface abgebildet werden. Der gepunktete Pfeil zeigt, dass eine Verbindung zu einem bestehende CORBA-Server-Objekt manchmal - aber nicht immer - möglich ist.

Diese Einschränkung gilt nicht für neu entwickelte Nicht-Java-CORBA-Server-Objekte (zum Beispiel geschrieben in C/C++). Es ist einfach, ein RMI/ IIOP Interface hierfür zu erzeugen.

Um zusammenzufassen: Wenn Sie Ihren Server in RMI/IIOP implementieren, haben Sie die größte Auswahl an Klienten. Ebenso, wenn Sie Ihren Klienten in RMI/IIOP implementieren, können Sie mit der größten Auswahl an Servern kommunizieren. Hierbei kann es einige Beschränkungen im Fall von bestehenden CORBA-Objekte geben.

16.3.4 Interoperability mit CORBA

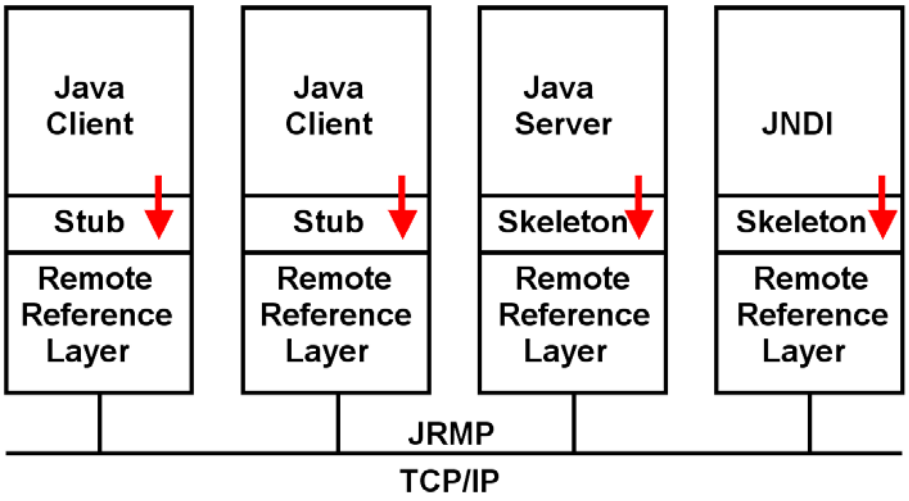

Abb. 16.3.3
JRMP Kommunikation

Java Client und Server benutzen die RMI interface ↓ um mit Stub und Skeleton zu kommunizieren. Der JDK enthält die “Remote Reference Layer” Komponente, welche alle Communication Funktionen implementiert. Es stellt auch die Stub und Skeleton Interfaces mit TCP/IP in OSI Layer 4 zur Verfügung.

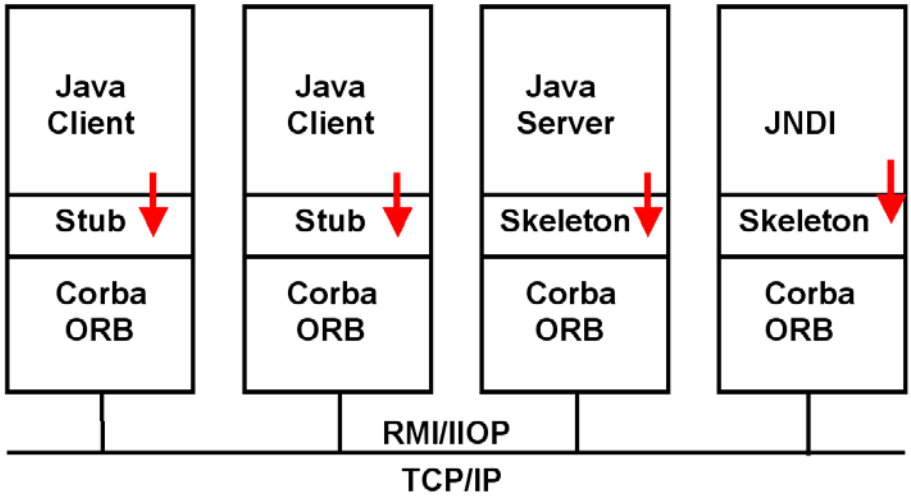

Abb. 16.3.4
RMI/IIOP Kommunikation

Bei der Benutzung von RMI/IIOP wird die Remote Reference Layer durch den Corba Object Request Broker (ORB) ersetzt.

Die RMI Interface, die vom Java Client und Java Server Object Code für die Kommunikation mit Skeleton und Stub benutzt wird, verändert sich nicht.

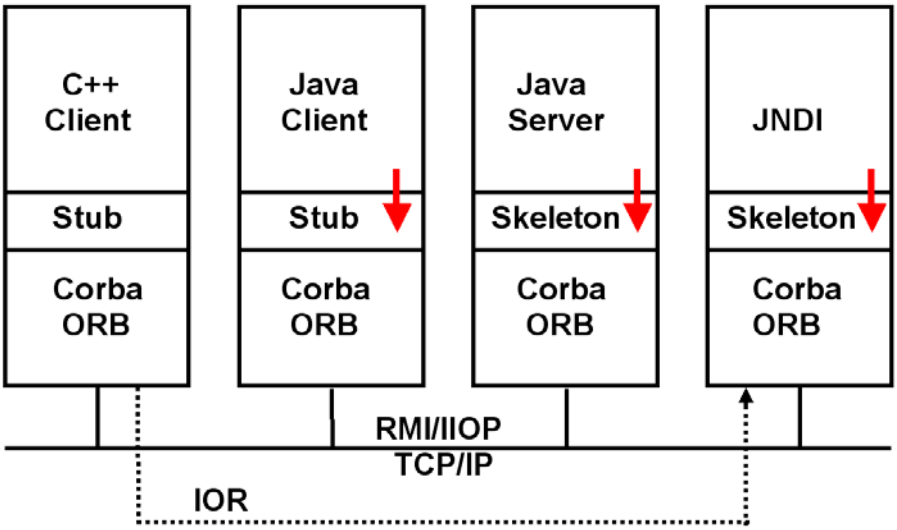

Abb. 16.3.5
Ein C++ Klient greift mittels RMI/IIOP auf einen Java Server zu

Wenn ein Corba Object Request Broker (ORB) für alle Java Komponenten benutzt wird, kann ein non-Java Member der Gruppe beitreten (ein C++ Klient in diesem Beispiel).

Die RMI Interface, die vom Client und Server Object Code für die Kommunikation mit Skeleton und Stub benutzt wird, verändert sich nicht.

Der vorhandene Java JEE Standard legt fest, dass alle Java Software Produkte RMI/IIOP unterstützen. Die Unterstützung für JRMP ist optional.

In Übereinstimmung mit dieser Spezifikation unterstützen IBM WebSphere und CICS unter z/OS nur RMI/IIOP. Ihre EJB Container haben die Funktionalität eines CORBA ORB.

Interessanterweise ist die Performance von RMI/IIOP häufig besser als die von JRMP.

16.3.5 JRMP Entwicklung und Ausführung

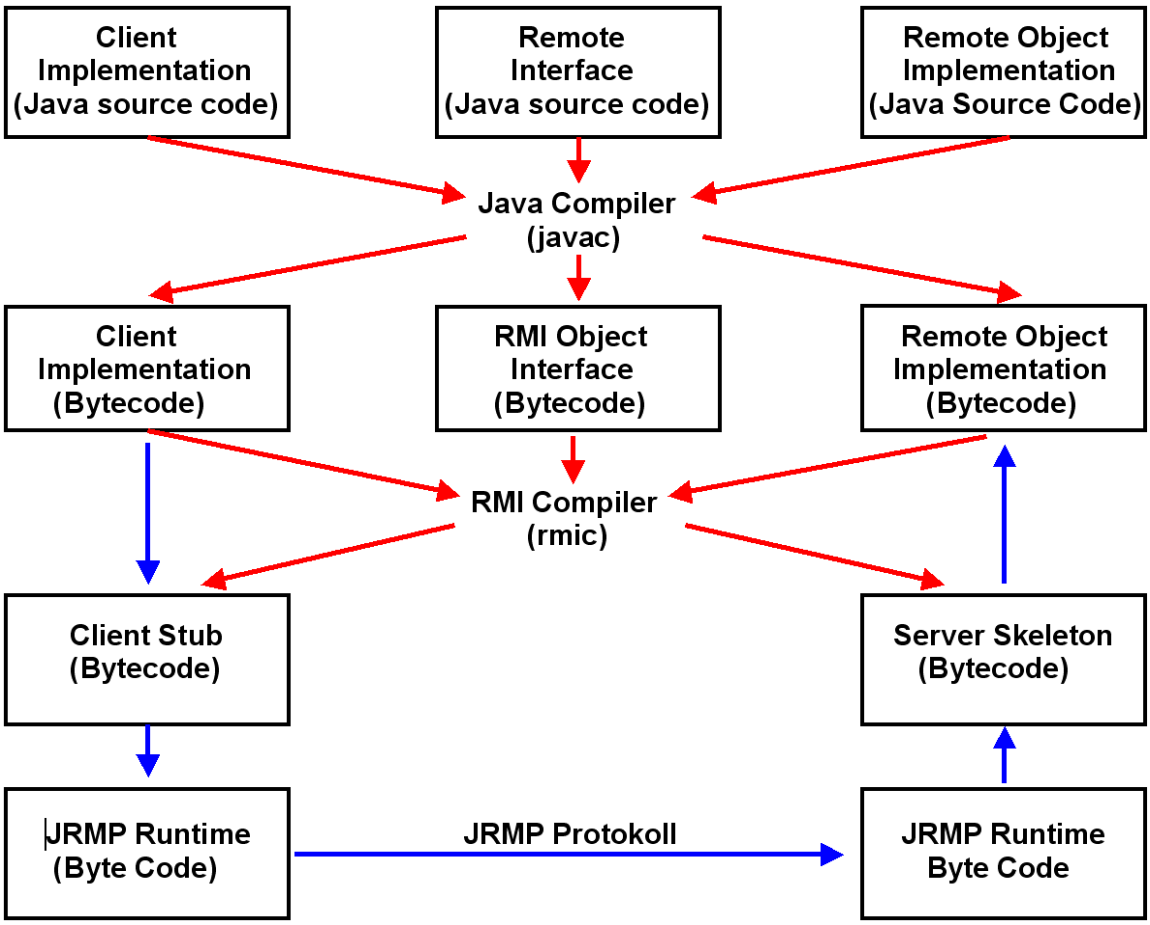

Abb. 16.3.6
Entwicklung und Ausführung mittels JRMP

In Abb. 16.3.6 ist die Entwicklung für den JRMP Server und den JRMP Klienten dargestellt. Folgen Sie den roten Pfeilen:

Der Quellcode der Server-side Object Implementierung und deren Java-Interface werden wie gewohnt mit dem regulären Java-Compiler (`javac`) übersetzt. Dies generiert Byte-Code.

Der Byte-Code wird als Eingabe für den Java RMI-Compiler (`rmic`) verwendet, zusammen mit der Remote-Interface Beschreibung des Java-Server-Objekts. Hiermit wird eine Byte Code Version sowohl des Server Skeletons als auch des Client-Stubs generiert.

Folgen Sie nun den blauen Pfeilen.

Wenn der Bytecode der Client-Implementierung das Remote-Objekt aufruft, überträgt es eine Nachricht an den Client-Stub. Die Client-Stub-Klassen übernehmen die Funktionen, die unique für diese Client-Implementierung sind. Die JRMP Laufzeitklassen führen die generischen JRMP Funktionen aus und starten die Kommunikation durch die Übertragung eines JRMP Nachricht an die TCP-Komponente des Client-Systems.

16.3.6 RMI/IIOP Entwicklung

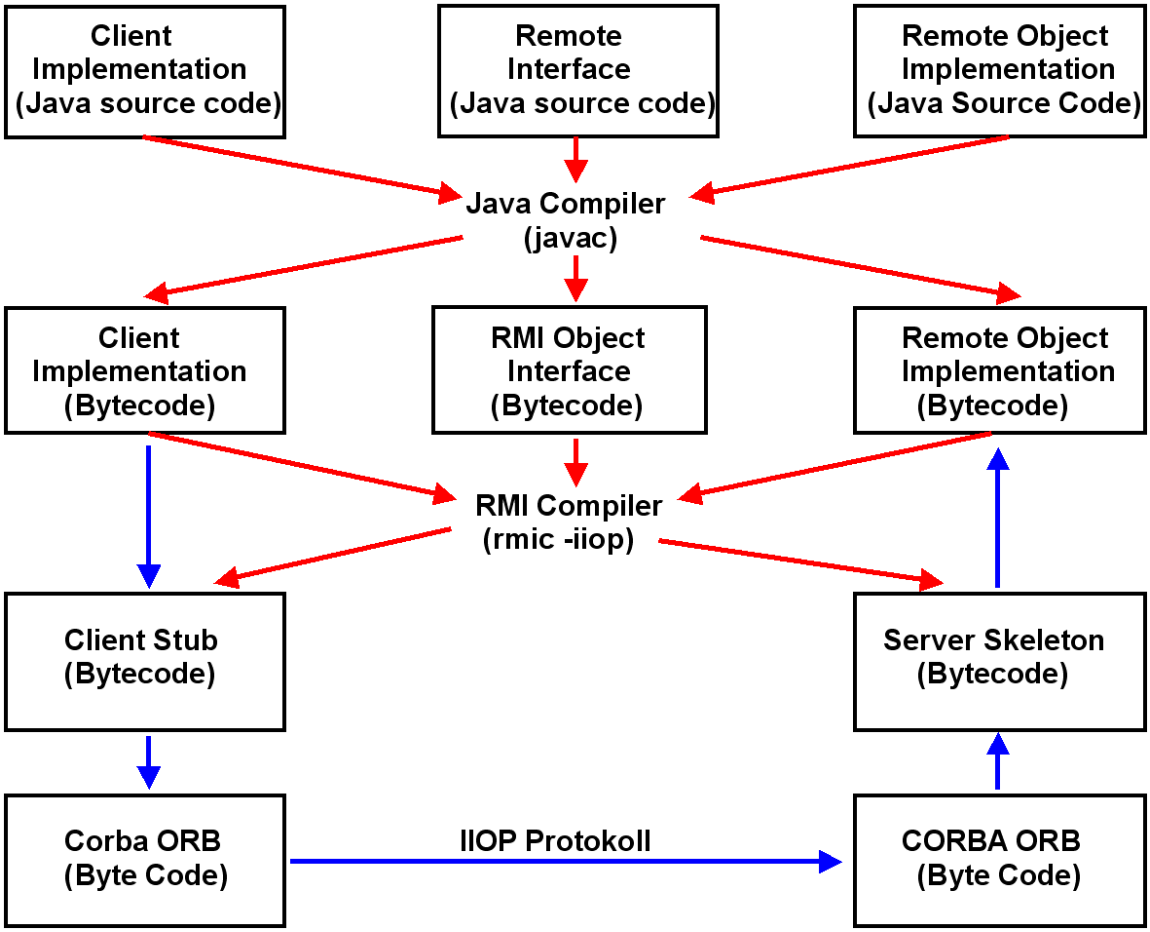

Abb. 16.3.7
Entwicklung und Ausführung mittels RMI/IIOP

Abb.16.3.7 zeigt das Entwicklungs-Verfahren für den RMI/IIOP Server und Klienten. Es entspricht Abb.16.3.6 . Sie werden feststellen, es ist fast das gleiche Verfahren wie bei RMI/JRMP.

Genauso wie bei RMI/JRMP enthält eine RMI Java-Interface die verteilte Server-Objekt-Definition. Ein Unterschied ist der `iiop` Parameter des `rmic` Compilers. Diese Option wird benutzt, damit `rmic` Stubs erzeugt werden, und diese mit dem IIOP Protokoll anstelle des JRMP Protokolls verknüpft werden. Obwohl die Entwicklungsverfahren für RMI/IIOP fast die gleichen wie für RMI/JRMP sind, ist die Laufzeitumgebung verschieden. Die Kommunikation erfolgt über einen CORBA-konformen ORB. IIOP wird für die Kommunikation zwischen Server und Client benutzt.

RMI/IIOP verwendet den Java CORBA Object Request Broker (ORB) und IIOP. Sie können den gesamten Code in der Programmiersprache Java schreiben, und den `rmic` Compiler (mit der `iiop` Option) benutzen. Die resultierende Java Anwendung kann über das Internet InterORB Protocol (IIOP) mit anderen in einem CORBA-kompatiblen Sprache geschrieben Anwendungen kommunizieren.

Der J2EE-Standard bietet zwei alternative Protokolle: JRMP und RMI/IIOP. Einige Hersteller unterstützen beide RMI Protokolle. IBM hat beschlossen, ausschließlich RMI/IIOP in ihren Produkten zu verwenden. Dies ist konform mit dem RMI-Standard.

16.3.7 Coexistence Beispiel: C++ client ruft ein Java RMI Object auf

CORBA und RMI/IIOP verwenden den gleichen Internet Inter-ORB Protocol Kommunikationsstandard. Wenn erforderlich ist es möglich, die IDL-Definitionen für die beteiligten RMI/IIOP Datenstrukturen erzeugen. Diese Definitionen können benutzt werden, um die Interoperabilität zwischen den RMI/IIOP Anwendungen und normalen CORBA-Anwendungen zu gewährleisten.

Mit RMI/IIOP können Entwickler Remote Interfaces in der Java-Programmiersprache schreiben und nur mit Hilfe von Java-Technologie und den Java RMI APIs implementieren. Diese Interfaces können in jeder anderen von CORBA unterstützten Sprache implementiert werden, vorausgesetzt es existiert ein ORB für diese Sprache. Ebenso können Klienten, die in anderen Sprachen geschrieben sind, IDL benutzen, die von den Remote-Java-Technologie basierenden Interfaces abgeleitet ist.

Verwenden Sie den RMI/IIOP Compiler mit der `iiop` Option um Stubs und Skeletons für Remote-Objekte zu erzeugen, welche das IIOP-Protokoll verwenden. Der `rmic` Compiler kann auch benutzt werden, um IDL für CORBA Entwicklungen auf der Java-Plattform zu erzeugen.

Als Beispiel betrachten wir den Fall, wo ein C++ CORBA Client ein RMI-Server-Objekt aufruft

Der `rmic-iiop` Compiler generiert Stubs und Skeletons für Java-Clients und Server.

Die C++ CORBA Client benötigt einen C++ CORBA IIOP Stub um das RMI-Server Objekt aufzurufen. Stubs werden mit Hilfe der Interface Definition des Server-Objektes erzeugt. Hierzu brauchen wir eine normale Corba IDL Beschreibung der Server Interface. Die Beschreibung der Server Interface liegt aber als Java Interface, und nicht als Corba IDL Interface vor.

Um den Client-Stub zu generieren, beginnen wir mit dem Java-Server-Interface.

Abb. 16.3.8
Generierung einer Corba Kompatiblen IDL File

Wie in Abb. 16.3.9 dargestellt, verwenden wir die Java-Interface und den `rmic` Compiler mit der `idl` Option um eine IDL-Beschreibung der Interface (IDL-Datei) zu generieren.

Mit dem IDL Beschreibung der Interface können wir jetzt den Client-Stub generieren.

Die IDL-Beschreibung, zusammen mit dem Client C++-Code wird von dem regulären CORBA-Compiler (der idl2cpp Compiler) verwendet, um den Code für CORBA C++ kompatible Stubs und Skeletons zu erzeugen. Der Stub wird mit dem C++-Client integriert, das Skelett wird verworfen, da das Java RMI-Objekt das regulären Skeleton verwendet, das durch `die rmic-iiop` Compiler generiert wurde.

16.3.8 Generating the Client Stub Code

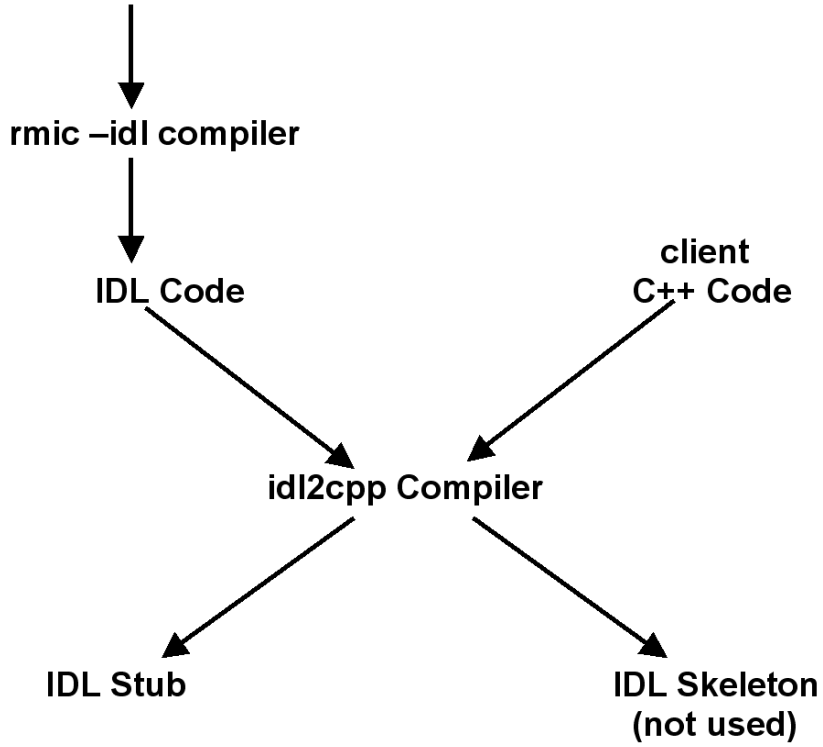

Abb. 16.3.9
Erstellen des Stub Codes für den C++ Client

Der idl2cpp Compiler wird von CORBA standardmäßig benutzt, um für ein C++ Objekt Skeleton und Stub zu erzeugen. Ähnliche Compiler existieren, um für Cobol, ADA, PL/1 Objekte Skeleton und Stub zu erzeugen. Weil der Server die Java Interface benutzt, wird das Server Skeleton vom rmic Compiler mit der iiop Option erzeugt.

Es ist möglich, eine Java Client/Server Anwendung mit Corba an Stelle von RMI zu entwickeln. Hierbei würden Stubs und Skeletons mittels des Corba IDL Compilers für Java erzeugt. Die Entwicklung ist aber nicht mehr so einfach wie unter RMI.

16.3.9 WebSphere CORBA Unterstützung

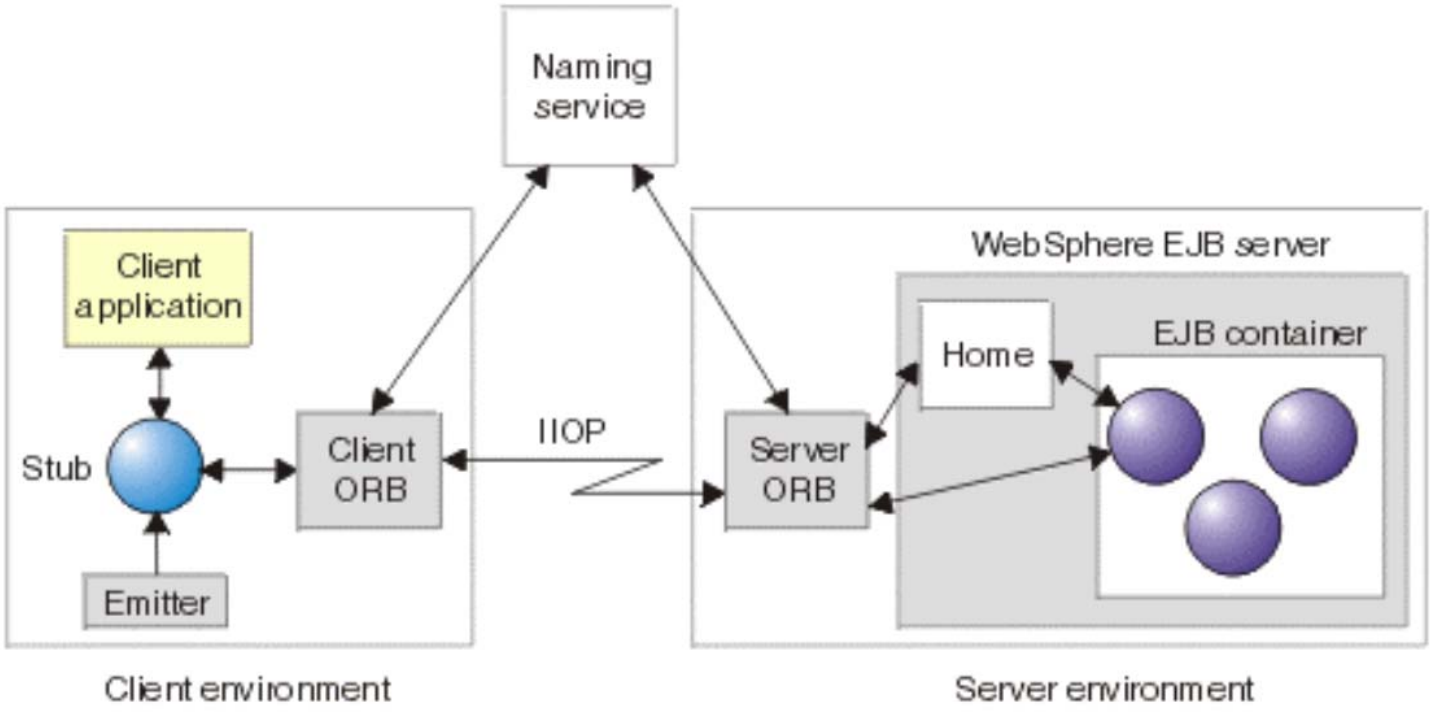

Abb. 16.3.10
WebSphere beinhaltet einen regulären CORBA ORB

Die IBM WebSphere Web Application Server beinhaltet CORBA-Unterstützung. Diese ermöglicht den Einsatz von CORBA-Interfaces zwischen einem Server-Objekt, das einen Service anbietet, und einem Client, der diesen Service benutzt. Diese Option ist zusätzlich zu den normalen Java Interfaces verfügbar. In der Praxis bedeutet dies:

- **WebSphere C++ CORBA-Server und WebSphere EJB-Services können von CORBA-Clienten aufgerufen werden, und**
- **WebSphere CORBA-Clienten können auf beliebige CORBA-Server zugreifen.**

Als Teil der WebSphere JEE-Umgebung beinhaltet die C++ CORBA-Unterstützung eine zusätzliche Basis CORBA Umgebung an. Diese kann in den JEE Namensraum integriert werden, und kann JEE-Transaktionen aufrufen.

16.4 Weiterführende Information

http://www.cedix.de/VorlesMirror/Band2/RmiBeispiel.pdf

enthält ein einfaches, aber detailliertes RMI Programmierbeispiel, welches Sie auf Ihrer Workstation ausführen können.

Ein weiteres RMI Tutorial finden sie unter

http://www.cedix.de/VorlesMirror/Band2/RmiBeispiel02.pdf

Die URL

http://www.youtube.com/watch?v=CLjyhH28AnE

zeigt ein Video, in dem RMI unter Eclipse entwickelt wird