UNIVERSITÄT LEIPZIG

Mainframe Internet Integration

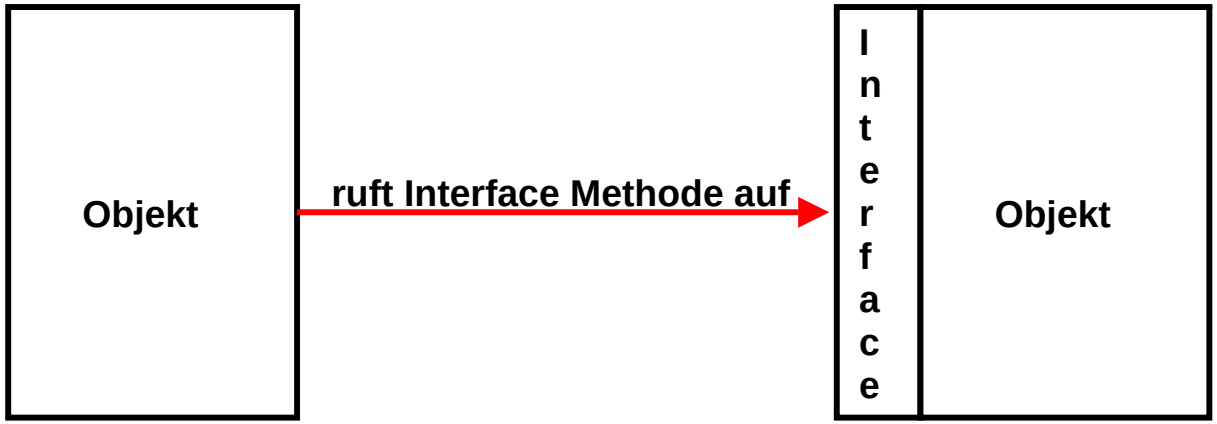
Prof. Dr. Martin Bogdan
Prof. Dr.-Ing. Wilhelm G. Spruth

SS2013

Java Remote Method Invocation Teil 2

RMI

Java Method Invocation



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).

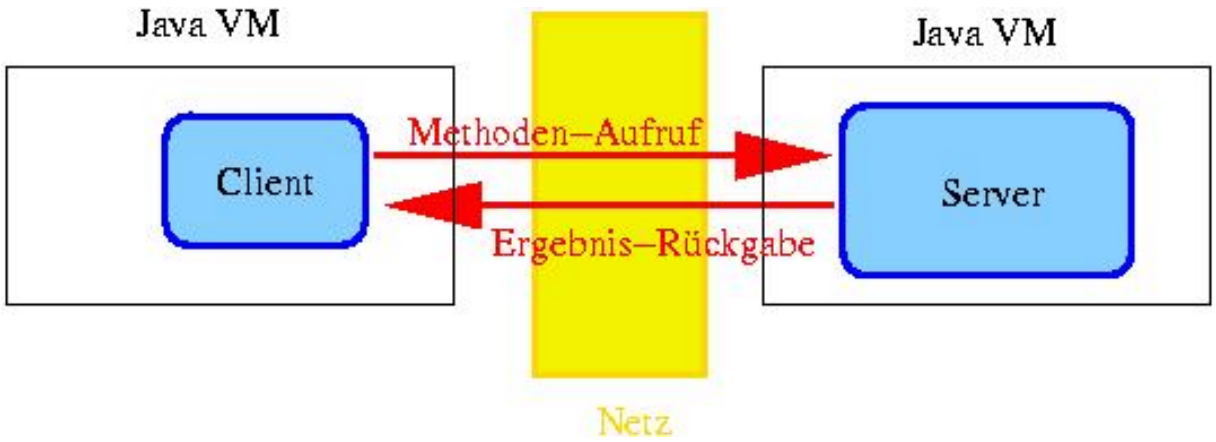
Bei einem lokalen Aufruf ruft der Klient eine Methode (von potentiell mehreren) auf, die in der Interface des aufgerufenen Objectes beschrieben ist.

Was passiert beim Methodenaufruf in einer Zeile wie

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

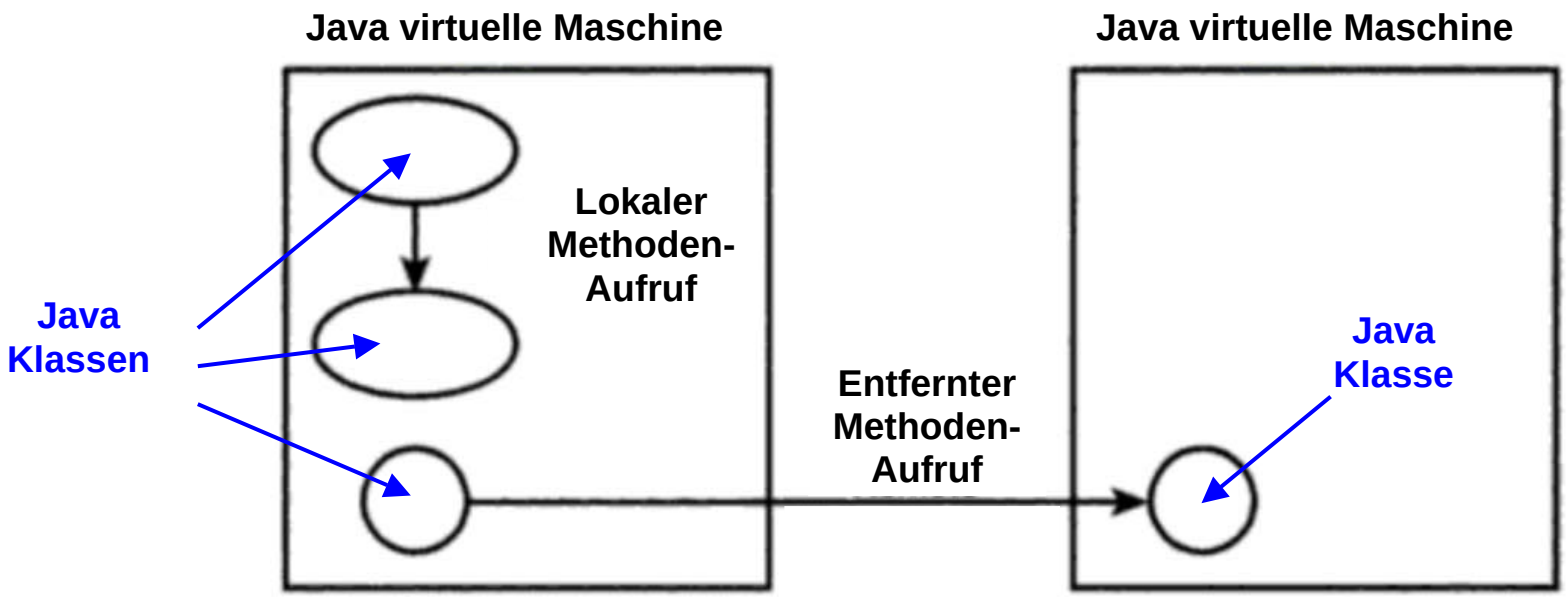
Auf der rechten Seite steht ein Funktions- oder Methodenaufruf. Ein Methodenaufruf ähnelt in seiner Form der ersten Zeile einer Methodendefinition: Auf einen Namen folgt ein Paar runder Klammern. Zwischen den Klammern können Parameter stehen (es gibt auch Methoden ohne Parameter).

RMI



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.



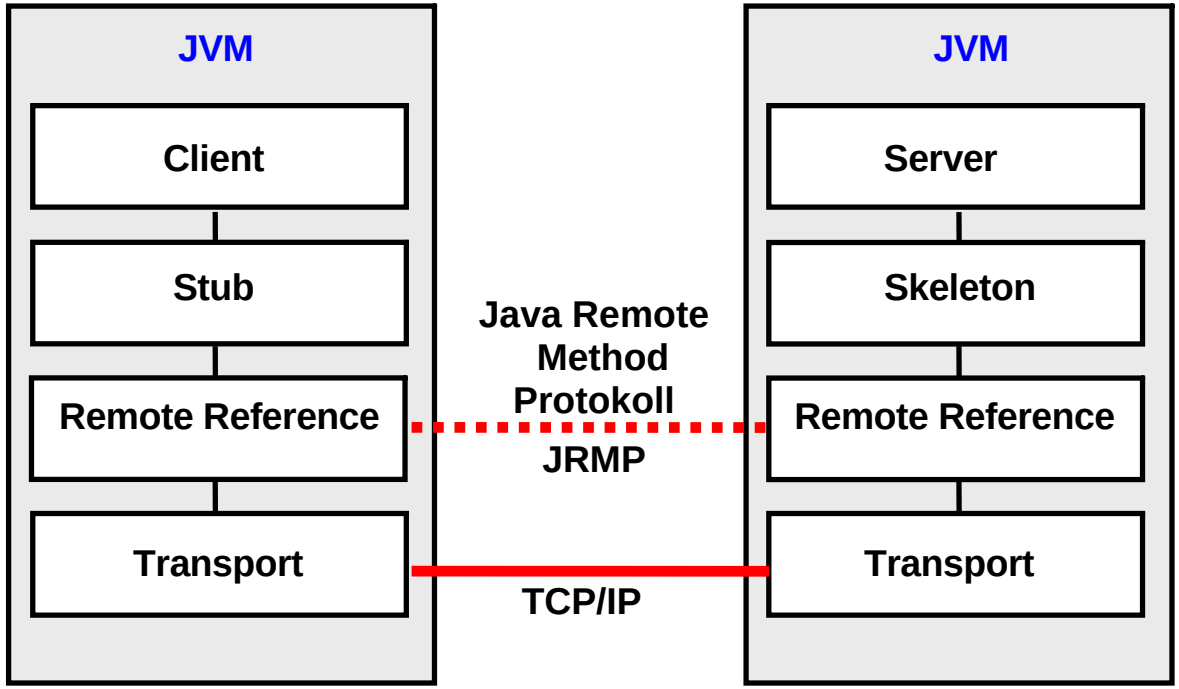
Der entfernte-Methodenaufruf, *Remote Method Invocation* (RMI) ist einer der Eckpfeiler von Enterprise JavaBeans und eine ausgesprochen praktische Möglichkeit, verteilte Java- Anwendungen zu erstellen.

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.**

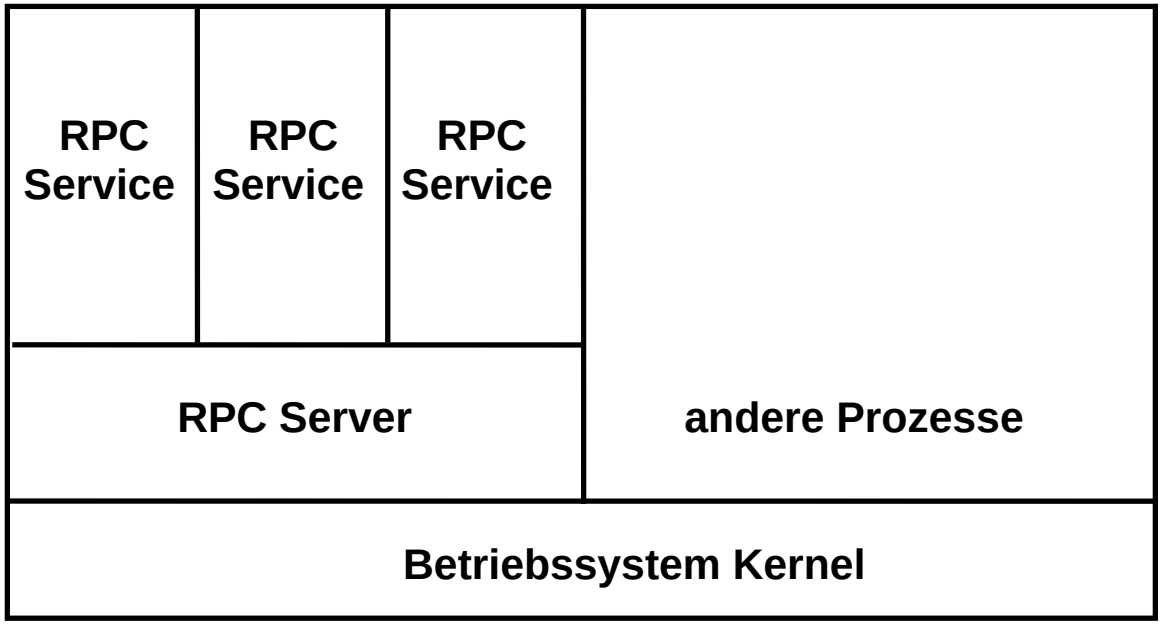
Remote Method Invocation (RMI)

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.



RPC Server und RPC Service

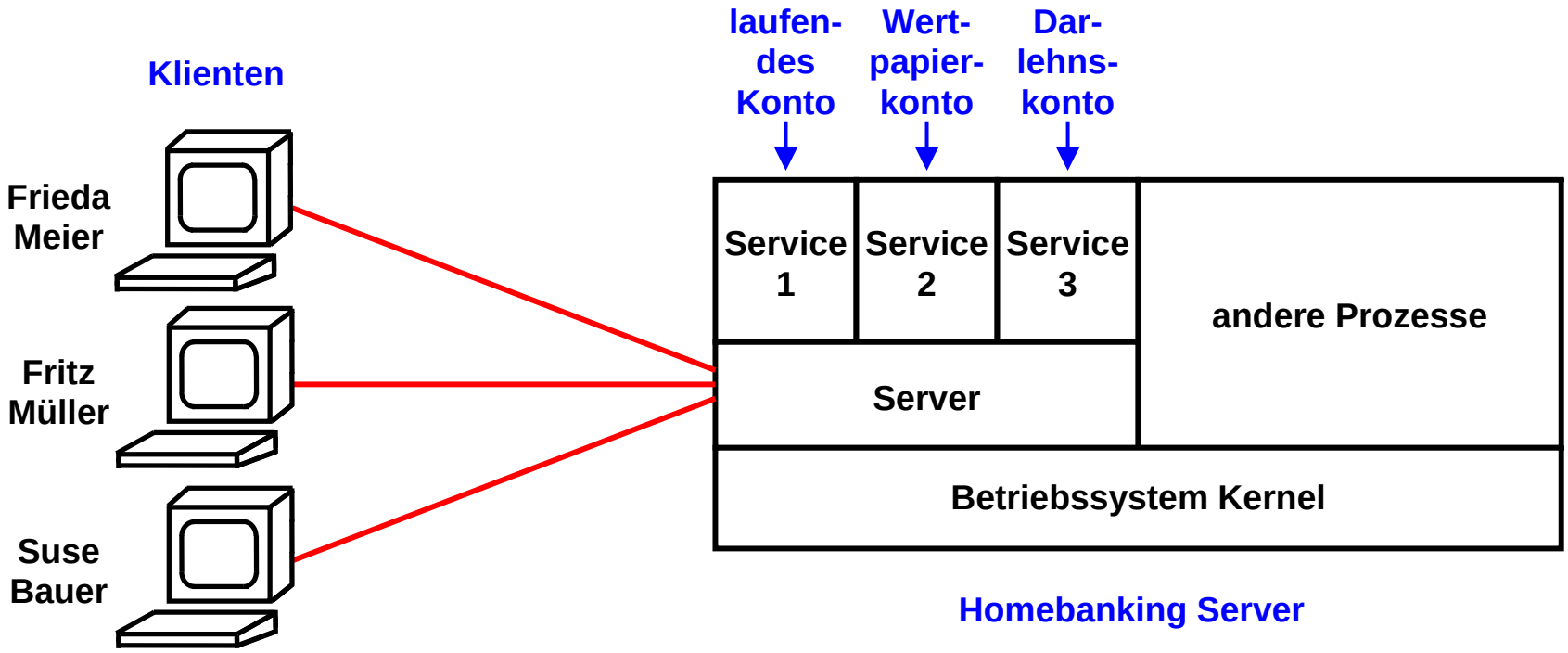
Die RPC Services (RPC Dienstprogramme) können entweder in getrennten virtuellen Adressenräumen laufen, oder alternativ als Threads innerhalb eines virtuellen Adressenraums implementiert werden. Java RPC Services können als Threads innerhalb einer JVM implementiert werden. Alternativ kann ein Java RPC-Server auch mehrere JVMs, jeweils für Gruppen von Java Services, unterhalten.

Ein RPC Service wird auch als „Implementation“ bezeichnet.

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

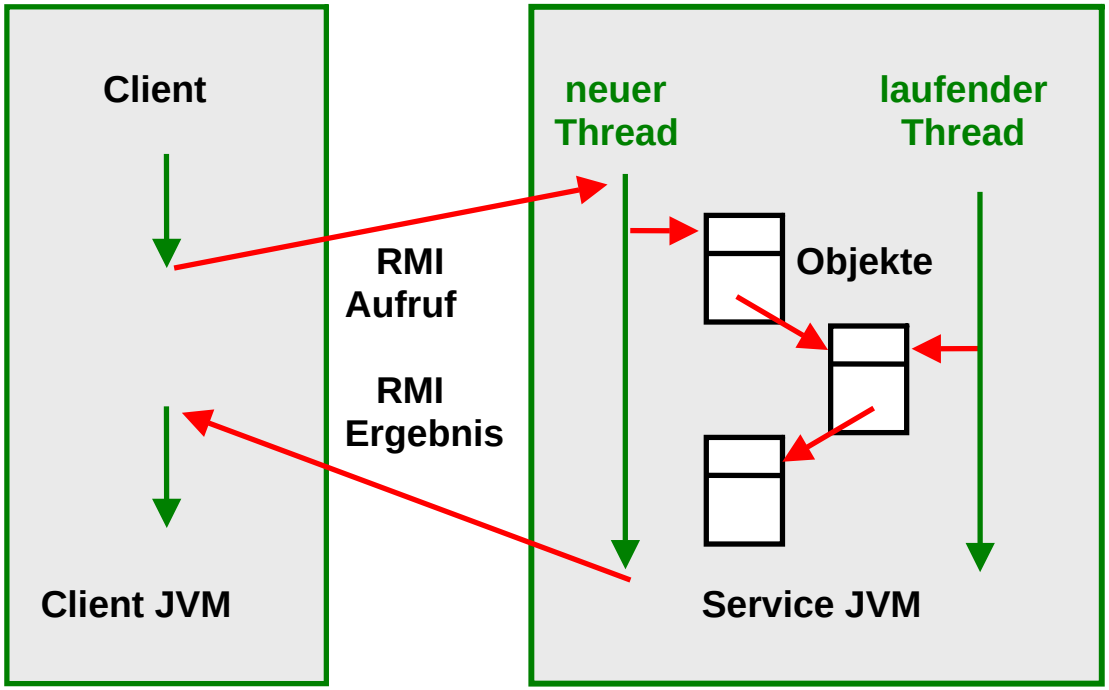
Auf einem Rechner können mehrere RPC Server laufen, die z.B. unterschiedliche Arten von RPC Services gruppieren.

Der RPC Server stellt Verwaltungsdienste für seine RPC Services zur Verfügung, z.B. Registry oder Sicherheits-Funktionen.



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 Authentifikation, Load Balancing, Errror Recovery, Namensdienste, Transaktionsdienste, Journalling usw.



Threads auf der Server-Seite

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 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.

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. Namen sind üblicherweise Zeichenketten (Strings), weshalb auch die Bezeichnung „String Name“ gebräuchlich ist.

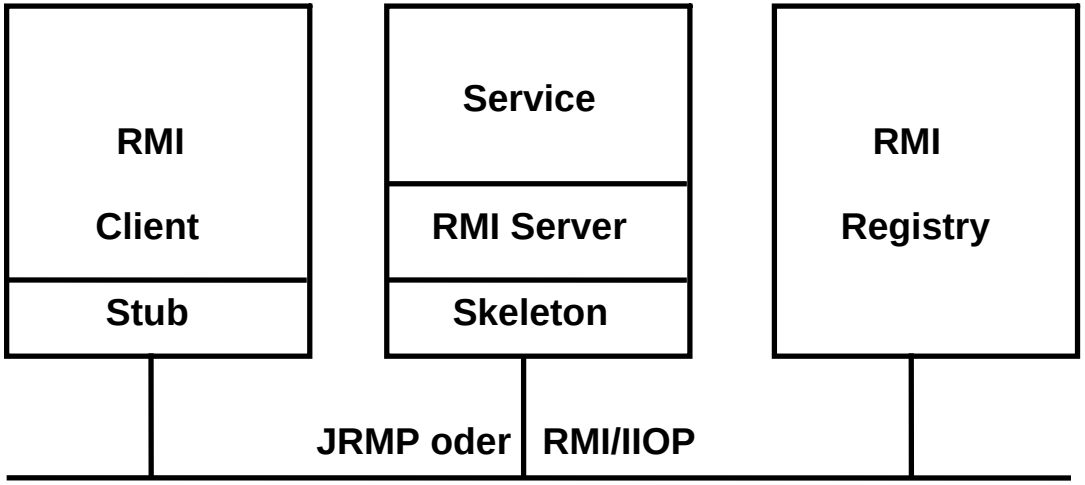
Ein bestimmter Service ist gekennzeichnet durch zwei Bezeichner: Seinen Namen sowie die Adresse, über die er erreichbar ist. Wenn in unserem Beispiel jeder Prozess über eine eigene IP Adresse erreichbar ist, könnte dies z.B. die IP Adresse 134.2.205.55, Port Nr. 2012 sein. Diese Adressse wird als Objekt 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 ermöglicht es, diesen (und auch die Services) auf physisch unterschiedlichen Servern unterzubringen, die sich irgendwo im weltweiten Netz befinden.

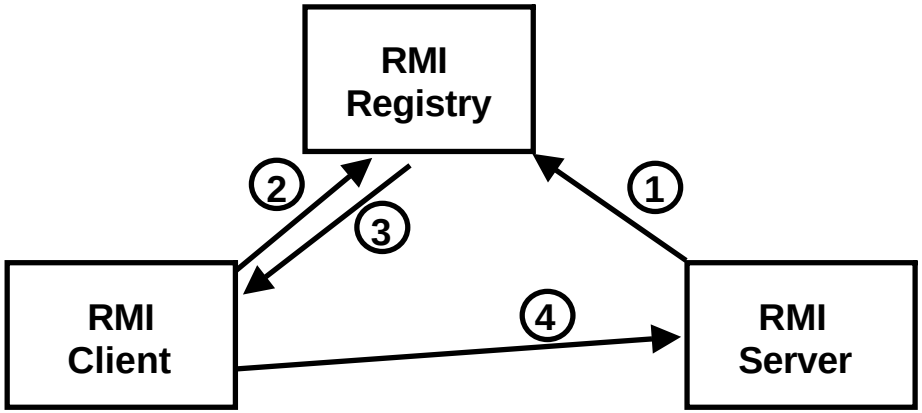
Remote Method Invocation (RMI)



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 (marshaling) in einen Datenstrom, der über das Netzwerk geschickt wird.**
- **Das Skeleton unmarshals die Argumente, ruft die Methode des RMI Services (Implementation) auf, marshals die Ergebnisswerte und schickt sie zurück.**
- **Der Stub unmarshals die Ergebnisswerte und übergibt sie an das Klientenprogramm.**

RMI Remote Class Programmausführung
Zeitlicher Ablauf



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.**

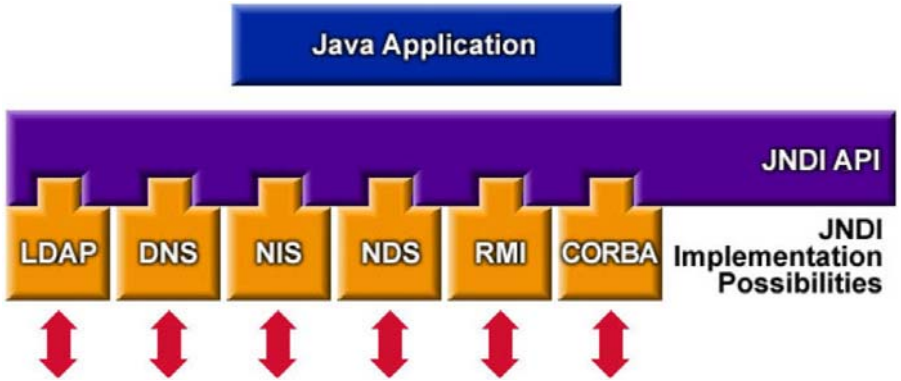
Ein Objekt Name ist vergleichbar mit einer URL wie z.B. http://leia.informatik.uni-leipzig.de
Eine Objekt Referenz ist vergleichbar mit einer dazugehörigen IP Adresse wie z.B. 139.18.4.30

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.



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.

Corba Common Object Service Naming- und Directory Service

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). DerCORBA COS (Common Object Services) Naming Service verfügt über eine Baum-artige Directory Struktur für Object Referencen.

Da CORBA im Gegensatz zu RMI sprachunabhängig ist, ist eine CORBA Objekt-Referenz eine abstrakte Entität. Diese wird von einem Client ORB in eine Language-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.

Erstellen einer RMI Remote Class (1)

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.**

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

Erstellen einer RMI Remote Class (2)

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 dazügehörigen 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;
}
```

Erstellen einer RMI Remote Class(3)

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.

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

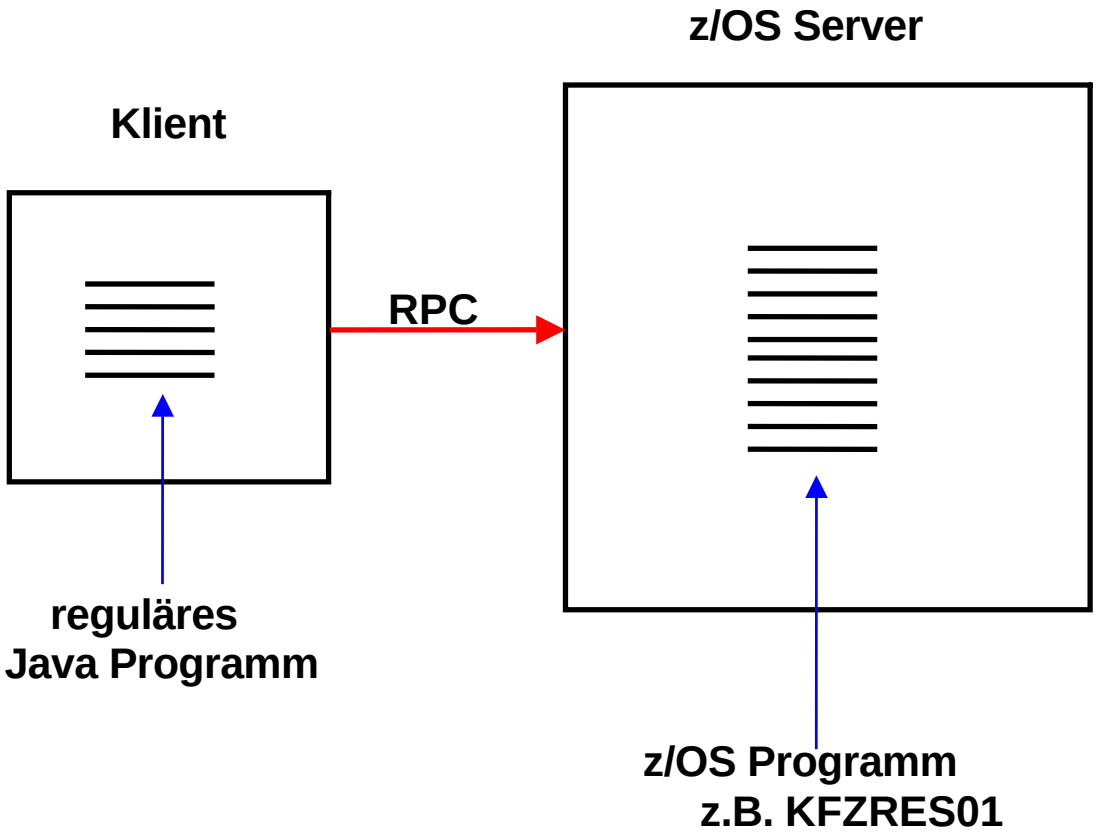
RMI benötigt viele CPU Zyklen für die Ausführung. Die gezeigten Daten stammen von einer Untersuchung http://www-csag.ucsd.edu/individual/achien/cs491-f97/projects/hprmi.html, oder http://jedi.informatik.uni-leipzig.de/de/VorlesMirror/ii/Vorles/Perform02.pdf.

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.



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 inherent objekt-orientiert.

Es ist aber auch möglich, Java als Programmiersprache für einen klassischen DCE RPC einzusetzen.

Hierbei gehen die Vorteile der Objektorientierung allerdings verloren.

DCE RPC mit Java Klienten ohne Objektorientierung

Es muss nicht alles RMI sein. Man kann auch Client/Server Anwendungen in Java schreiben ohne Benutzung von RMI. 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.

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;
  }
}
```

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.