Verteilte Systeme
(Betriebssysteme II)

Kapitel 6: Verteilte Objektsysteme

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Java Remote Method Invocation (RMI)

- Distributed object applications need to do the following:
  - **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects.
    - register remote objects with RMI's simple naming facility, the RMI registry.
    - pass and return existing remote object references
  - **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
  - **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

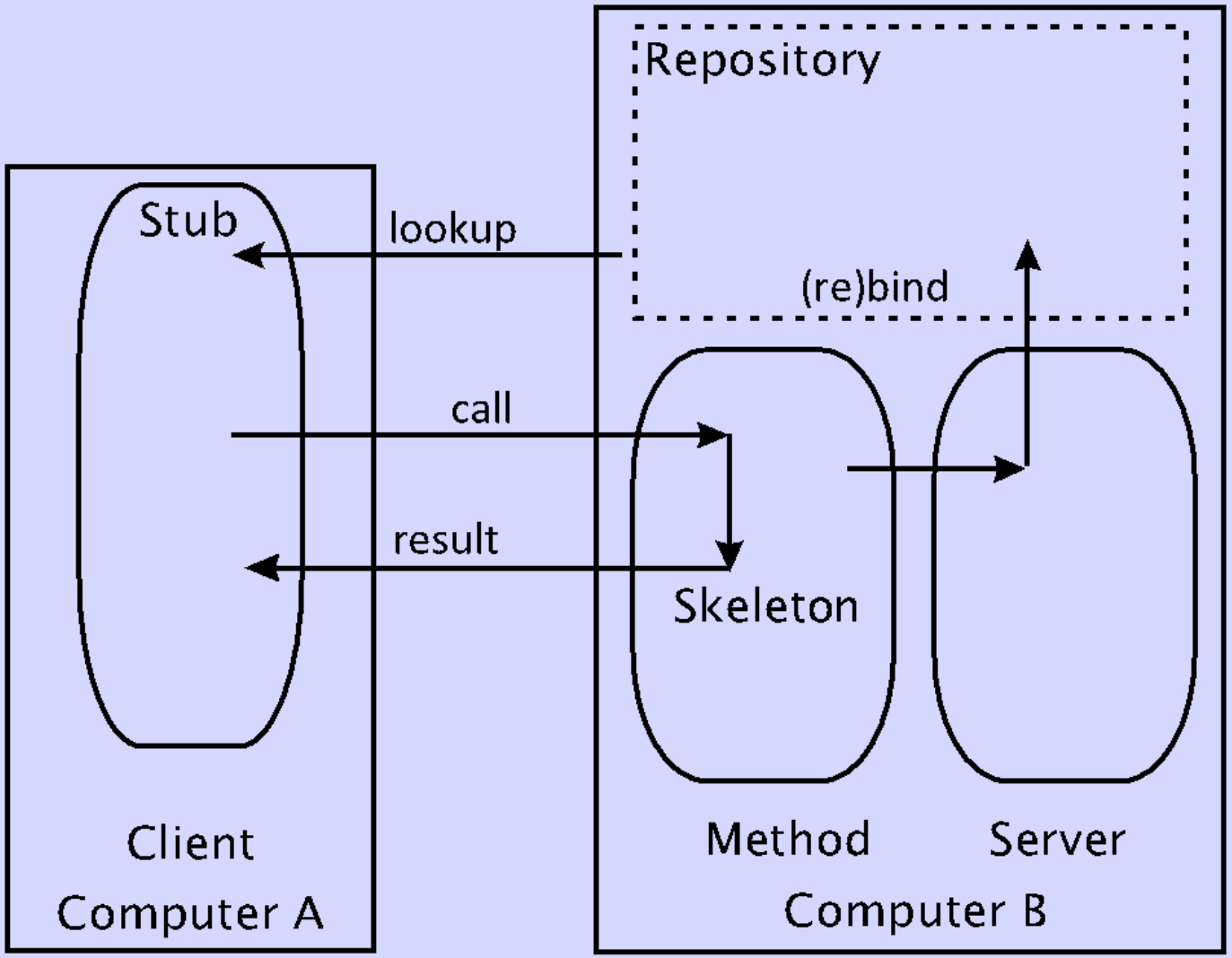
Java Remote Method Invocation (RMI)

- Entfernte Prozeduren → entfernte Methoden.
- RMI = RPC in Objektsystemen
  - Parameter können Objekte sein (Erschwernis)
    - Umgang mit Zeigern und mit Code
    - Objekte in Bytestrom übertragen (*serialisieren*)
    - verzeigerte Objekte (Listen, Bäume, Graphen): Objektgraphen
    - ohne Code (struct) oder mit Code (Objekt, insbes. Java)
  - Alle Information in der Klassendefinition gekapselt (Erleichterung)
- Java RMI ist eine Java spezifische Realisierung des RPC.
  - benutzt Java Objekt-Serialisierungsprotokoll
- *Common Object Request Broker Architecture (CORBA)*
  - Kapselung von Objekten verschiedener Programmiersprachen

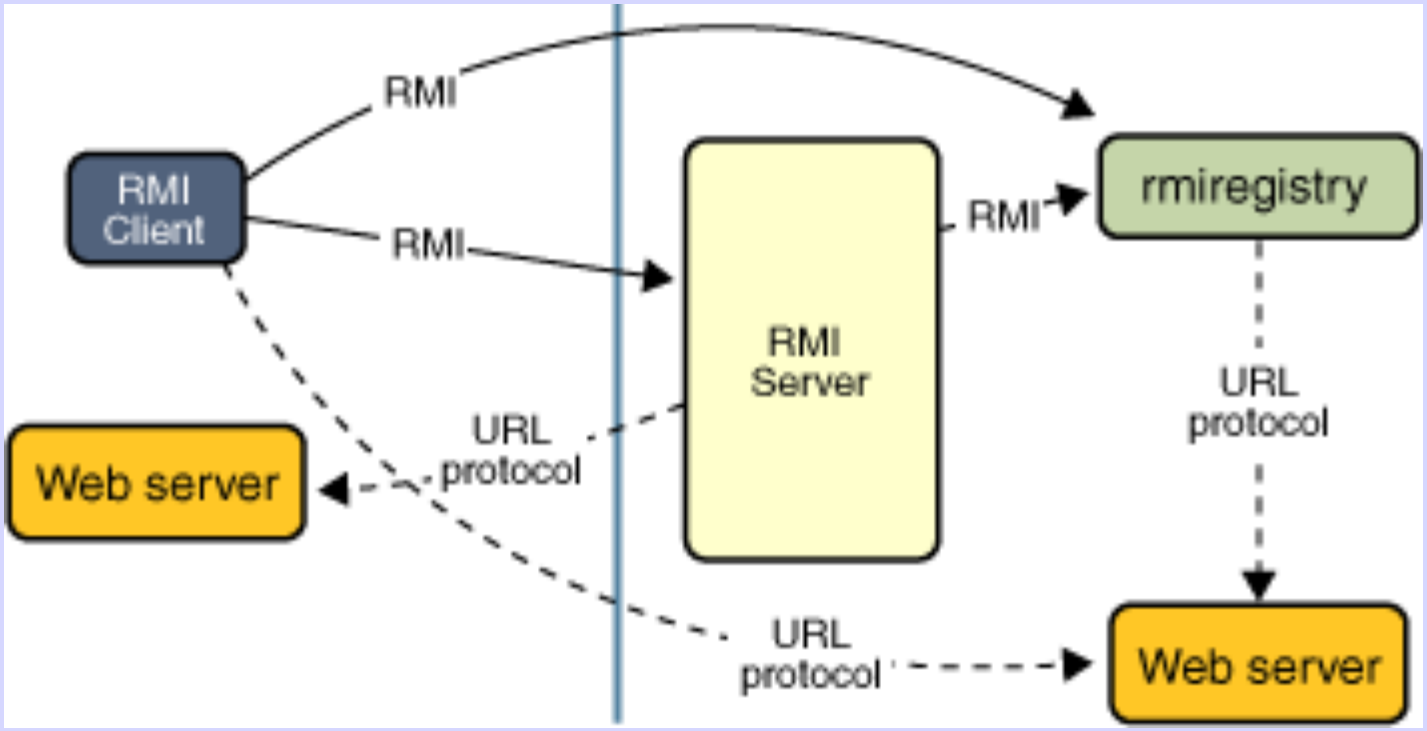
Java Remote Method Invocation (RMI)

- Im Folgenden ist die ursprüngliche Version von RMI beschrieben.
  - rmic ist inzwischen überflüssig (Stubs werden dynamisch erzeugt)
- Aktuelle Beschreibungen siehe
  - Remote Method Invocation Home:
    http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
  - Getting Started Using Java RMI:
    http://java.sun.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html
  - JDK 6 RMI:
    http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html
  - RMI Papers:
    http://java.sun.com/developer/technicalArticles/RMI/

RMI – Schema


Repository
Stub
lookup
(re)bind
call
result
Skeleton
Client
Computer A
Method
Server
Computer B

RMI – Schema


RMI
RMI Client
RMI
RMI Server
RMI
rmiregistry
URL protocol
URL protocol
Web server
URL protocol
Web server

Aufgabe von verteilten RPC Systemen

- Verteilte RPC Systeme (wie das verteilte Objektsystem CORBA) zielen auf die Lösung der Kommunikations-probleme, die auf der **Heterogenität** der beteiligten Systeme beruhen:
  - Unterschiedliche Programmiersprachen
  - Unterschiedliche Rechner
  - Unterschiedliche Betriebssysteme
  - Unterschiedliche Datenrepräsentation
  - Unterschiedlicher Maschineninstruktionssatz

Lösungsansatz bei RPC Systemen

- Stub, Skeleton bilden Aufrufschnittstelle und erledigen den eigentlichen Datenaustausch
- Marshaling, Unmarshaling zur (De-)Serialisierung
- Gemeinsame IDL (Interface Description Language) ermöglicht Kommunikation auch zwischen unterschiedlichen Programmiersprachen
  - Für jede beteiligte Sprache ein IDL Compiler
- Unterstützung entfernter Referenzen für Objekte

Probleme bei RPC

- Einschränkung hinsichtlich der übertragbaren Daten
  - Nur einfache Datentypen, die in allen unterstützten Programmiersprachen repräsentierbar sind und
  - Referenzen auf entfernte Objekte sowie
  - Komplexe Datentypen, die sich aus den zuvor genannten zusammensetzen
- Komplexität bei Typanpassung verbleibt beim Programmierer
- Life-Cycle-Management wird dem Programmierer auferlegt
  => Gefahr von Fehlern
- Sender und Empfänger müssen die übertragbaren Datentypen zum Zeitpunkt der Kompilierung bereits kennen.
  => Keine Unterstützung von Polymorphie

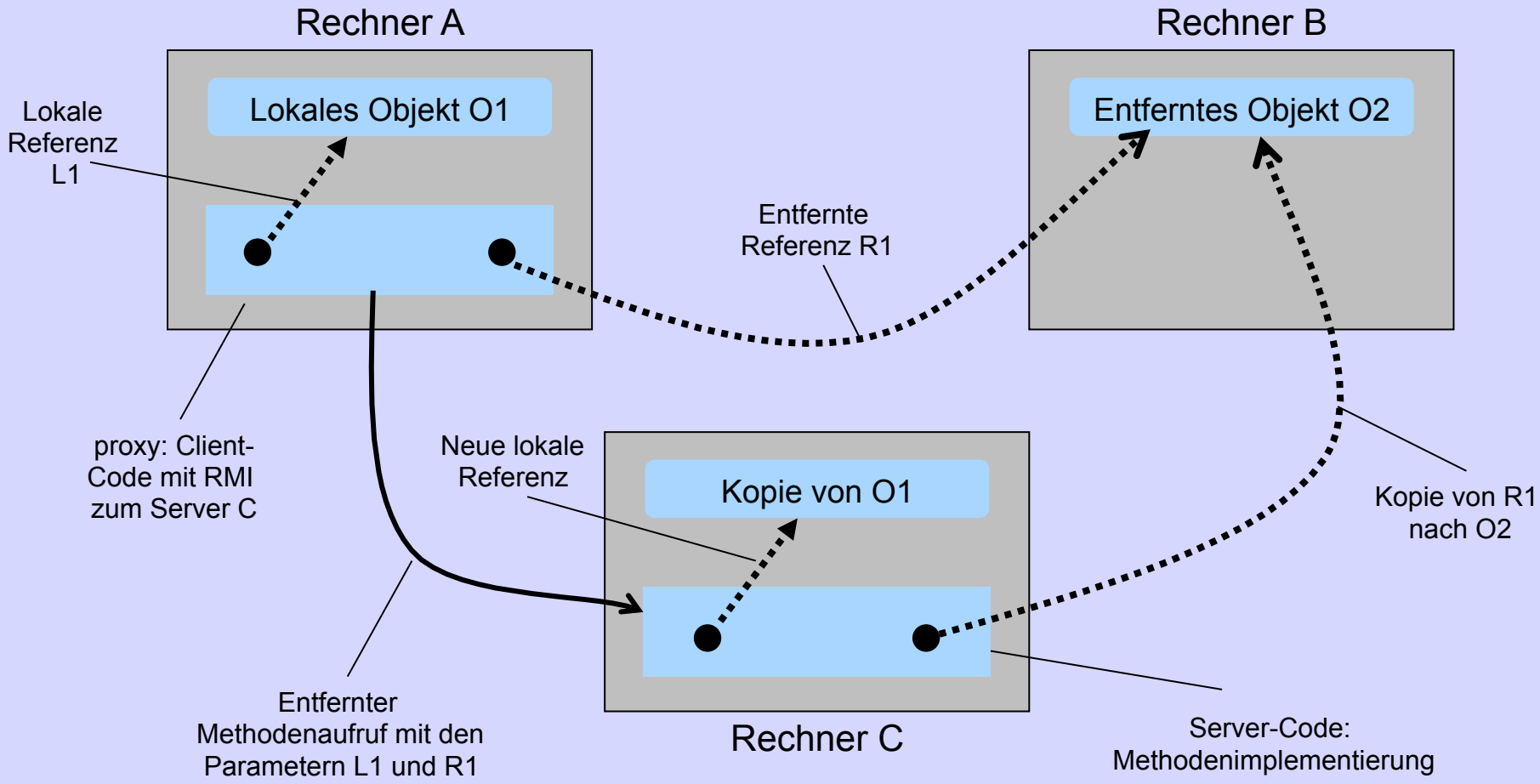
Lösung von Java RMI

- Heterogenität stellt kein Problem dar, da **Homogenität** durch die Java JVM gewährleistet ist.
  - Externe IDL nicht erforderlich (stattdessen Java interfaces)
  - Keine Einschränkung hinsichtlich der übertragbaren Datenstrukturen.
  - Java Objekt-Serialisierung ermöglicht exakte Typenprüfung
  - Dynamic Code Loading: Der den Kommunikationsfluss steuernde Programmcode kann auch erst während der Programmausführung zur Verfügung gestellt werden.
  - Java Objekt-Serialisierung und Dynamic Code Loading ermöglichen den Einsatz von Polymorphie und aller darauf aufbauenden Programmiermuster
  - Network Garbage Collection

Parameterübergabe und Rückgabewerte

- Java allgemein:
  - Alle Parameter (und Rückgabewerte) werden jeweils kopiert: call by value.
    Hinweis: Objektvariablen bilden dabei keine Ausnahme. Da es sich bei ihnen jedoch um Referenzen handelt, entspricht ihre call by value-Übergabe der Semantik von call by reference. Änderungen durch die aufgerufene Methode entfalten also Wirkung.
- Java RMI:
  - Lokale Datentypen und lokale Objekte werden kopiert: call by value
  - Entfernte Objekte durch Kopie (des entsprechenden proxies/ stubs): call by reference

Veranschaulichung der Parameterübergabe



Quelle: Tanenbaum, Distributed Systems, 2. Aufl., Abb. 10-8

=> Entfernte Objekte bleiben entfernt

Serialisierung in Java

- Serialisierung konvertiert im Hauptspeicher befindliche Objekte in ein Format (Bytestrom), das in eine Datei geschrieben oder über eine Netzwerkverbindung transportiert werden kann.
- Ziel von Serialisierung ist zumeist Persistenz
- Hier jedoch: Objektübertragung für Java RMI

Parameter Marshalling bei RMI

- Für das Parameter-Marshalling wird standardmäßig die Java Objekt-Serialisierung angewendet.
  - offenes Protokoll, kann (mit Einschränkungen) auch für C++ implementiert werden.
- Eine Java-Klasse, die serialisierbar sein soll, muss das (leere!) Interface `Serializable` implementieren.
  - Die Klasse muss also vom Typ `Serializable` sein.
  - Programmierer muss i.A. keine Methoden implementieren
  - Schema der Java-Objekt-Serialisierung kann automatisch und rekursiv über die Basisklassen und die Attribute einer Methode bis hinunter zu den Standard-Datentypen durchgeführt werden.

Serialisierung von Objekten

- Custom Schema für die Objekt-Serialisierung möglich durch Implementierung folgender Methoden:

```
private void readObject(java.io.ObjectInputStream stream)
    throws IOException, ClassNotFoundException;

private void writeObject(java.io.ObjectOutputStream stream)
    throws IOException;
```

- Kommunikation von RMI- und CORBA-basierten Anwendungen:
  - Java-Objekte beim RMI-Parameter-Marshalling gemäß des CORBA-IIOP-Protokolls serialisieren
  - (IIOP = Internet Inter ORB Protocol, RMI / IIOP).

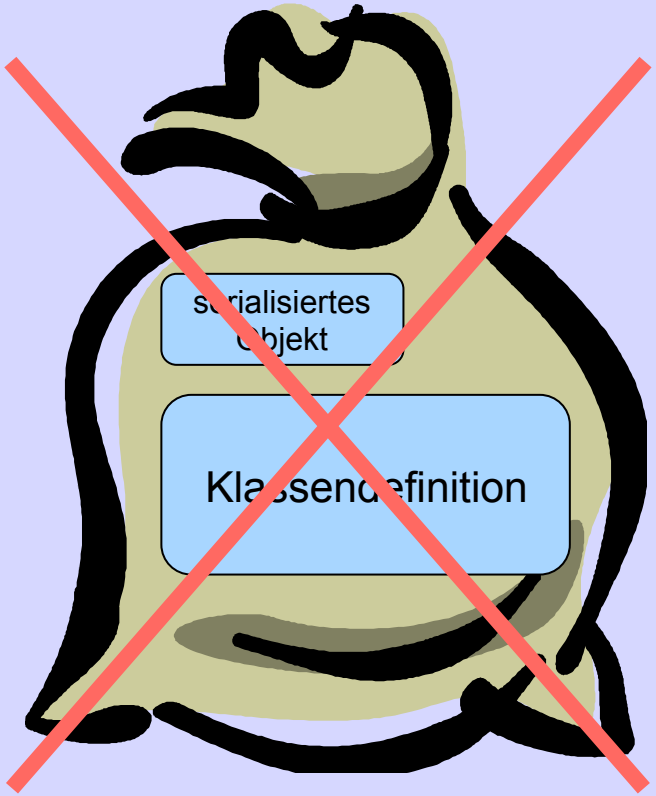
- Serialisierung von Objekten ist ein allgemeines Konzept.
  - Speichern von Objekten auf nicht-flüchtigem Speicher (z.B. Files, ...) *Objekt-Persistenz*
  - Copy- & Paste (bzw. drag and drop) von komplexen Objekten.

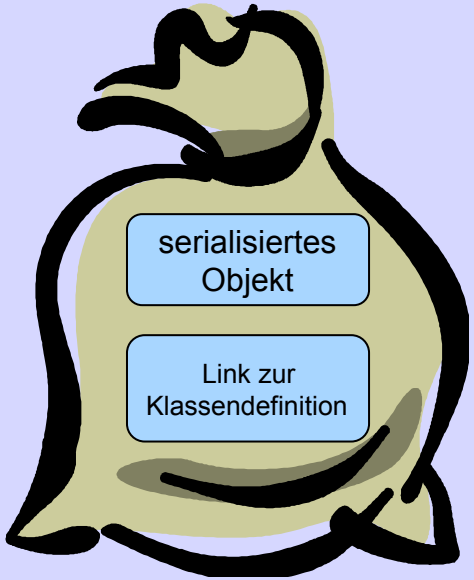
Aspekte der Serialisierung

- Versionierung erforderlich, um Inkonsistenzen durch die Trennung von Code und Daten zu vermeiden
- Interface Serializable enthält Hashcode `serialVersionID`, der sich aus den wichtigsten Eigenschaften der Klasse (Signaturen der Methoden, etc.) berechnet.

Optimierung der Serialisierung

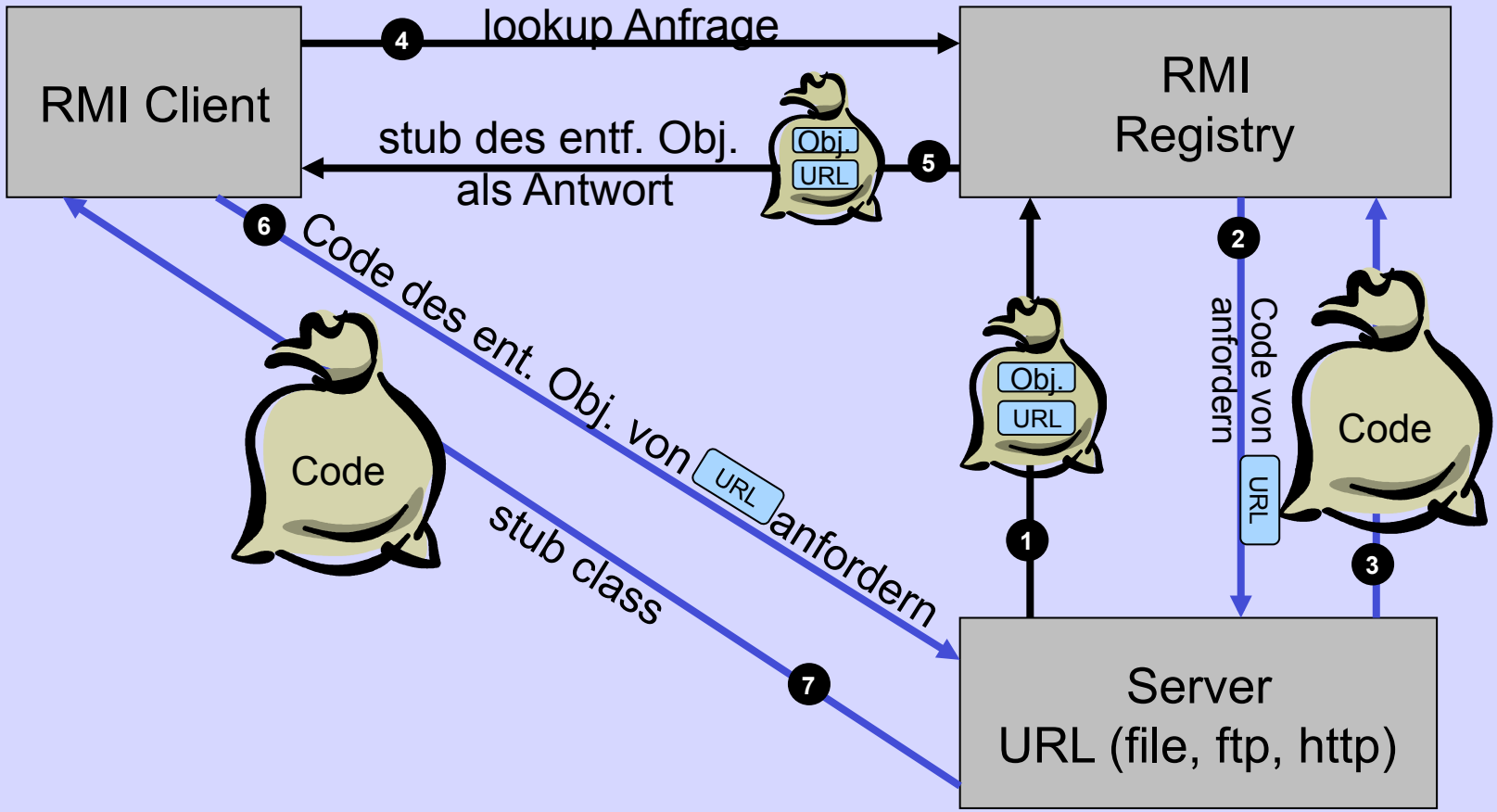
- Java RMI sendet mit dem serialisierten Objekt nur einen Link, von dem der Client benötigte Klassendefinitionen nachladen kann (Codebase).
- Vorteile: Proxy eines entfernten Objektes kann mit wenigen hundert Bytes übertragen werden. Außerdem kann der Client einmal geladene Klassendefinitionen in seinem Cache ablegen.





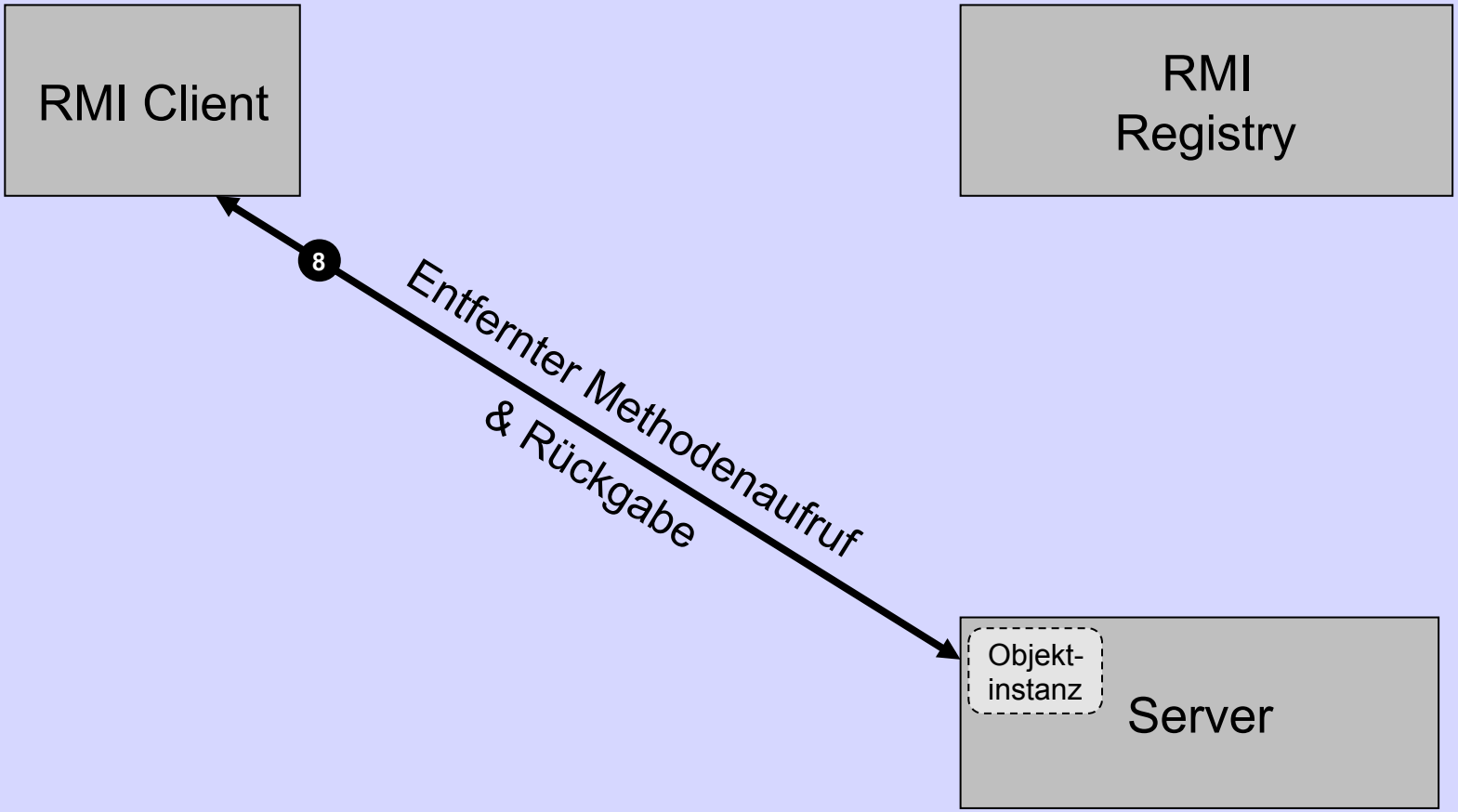
Java RMI

Schritt 1: Download der Java RMI stubs + code



Quelle: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

Schritt 2: Entfernter Methodenaufruf durch RMI Client

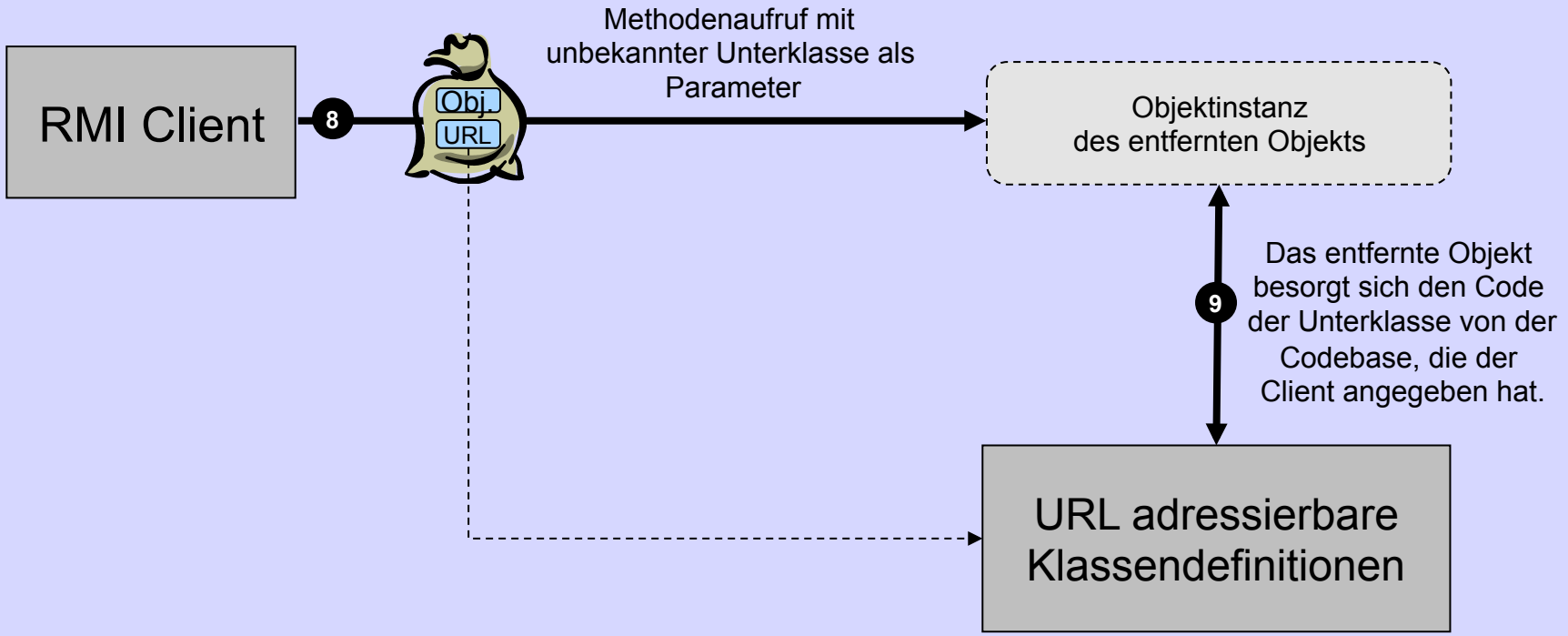


Quelle: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

RMI mit unbekannter Unterklasse

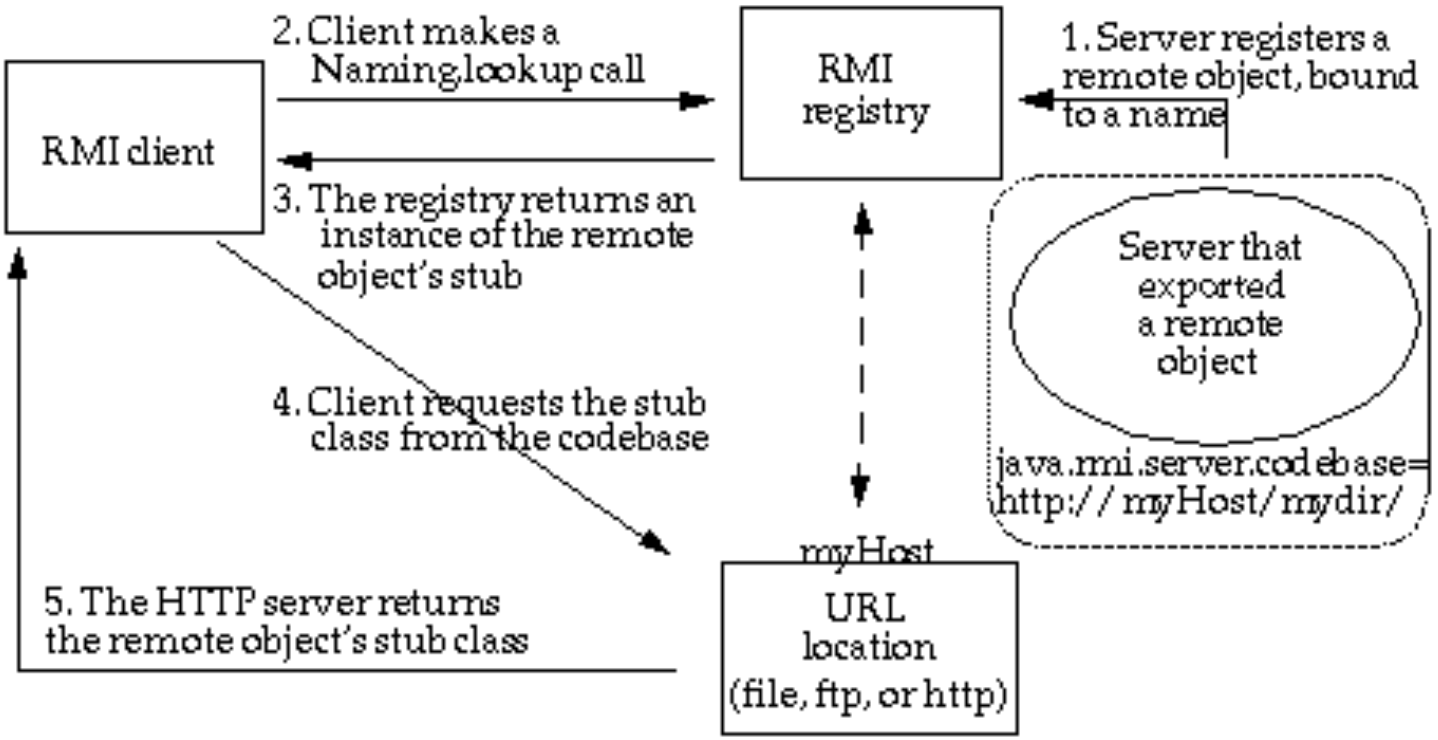
- Problemstellung:
  - Ein Server-Objekt `so` implementiert eine Methode `m(SomeClass o)`
  - im Verlauf von `m()` wird auf `o` eine Methode `o.m1()` aufgerufen
  - In einem konkreten Aufruf `so.m(do)` könnte `do` auch ein Objekt von einer Unterklasse `SomeDerivedClass` von `SomeClass` sein, wobei `m1` in `SomeDerivedClass` überschrieben wurde
  - Damit fehlt im Server der Code von `m1` in `SomeDerivedClass`
- Lösung
  - Der fehlende Code wird vom Server dynamisch von der Codebase von `SomeDerivedClass` nachgeladen
- In C++ ist ein RMI mit unbekannter Unterklasse nicht realisierbar

Schritt 2 –alternativ-: RMI mit unbekannter Unterklasse



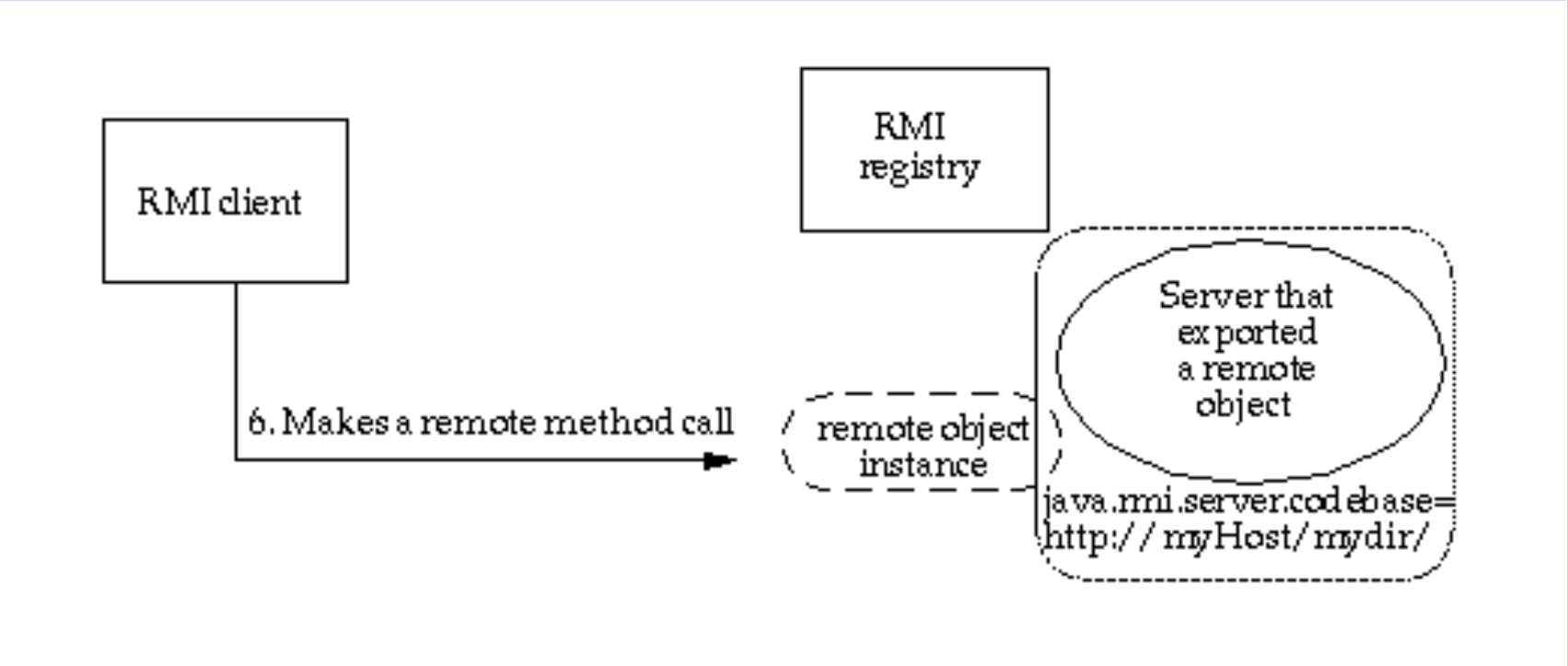
Quelle: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

Schritt 1: Download der Java RMI stubs



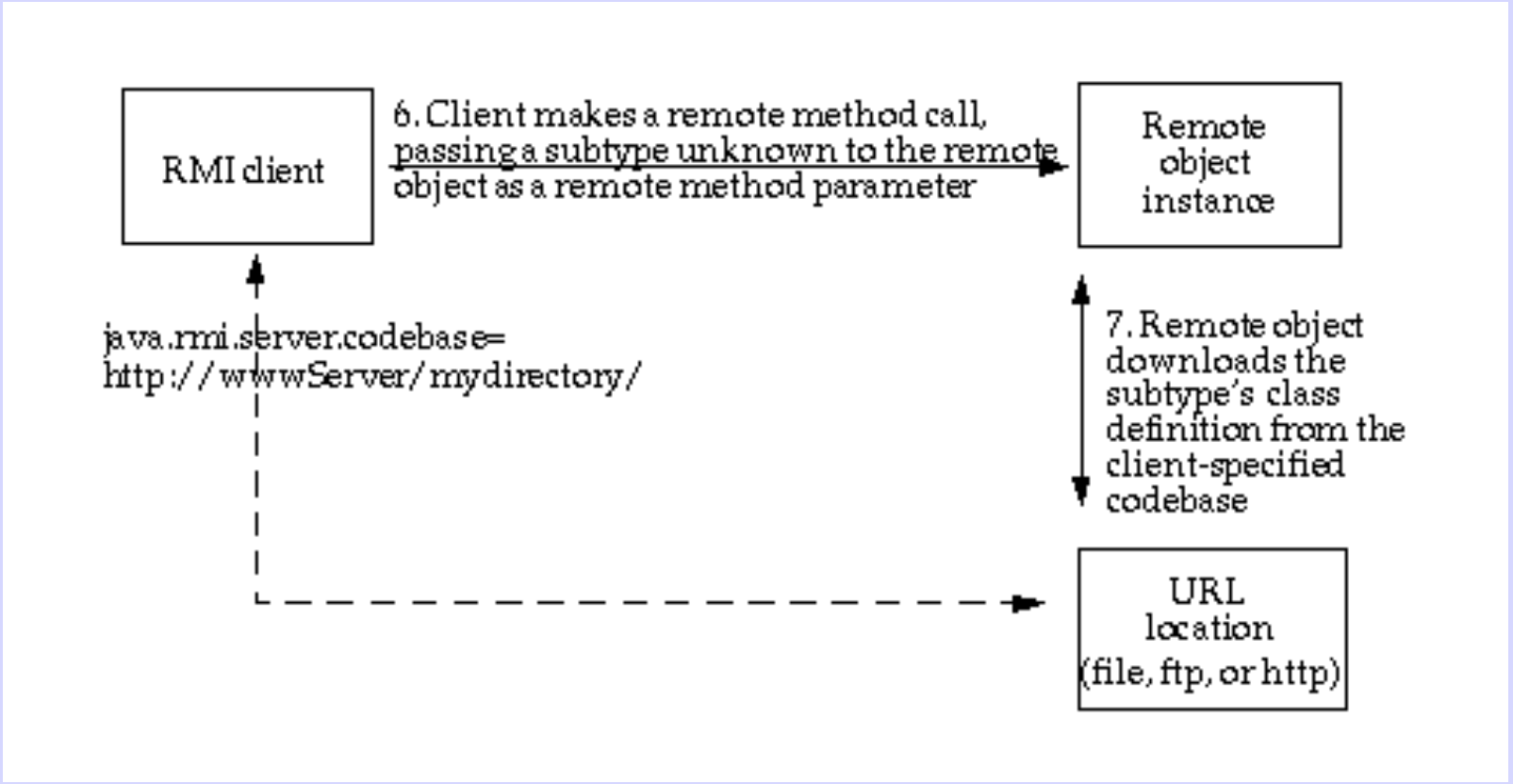
Quelle: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

Schritt 2: Entfernter Methodenaufruf durch RMI Client



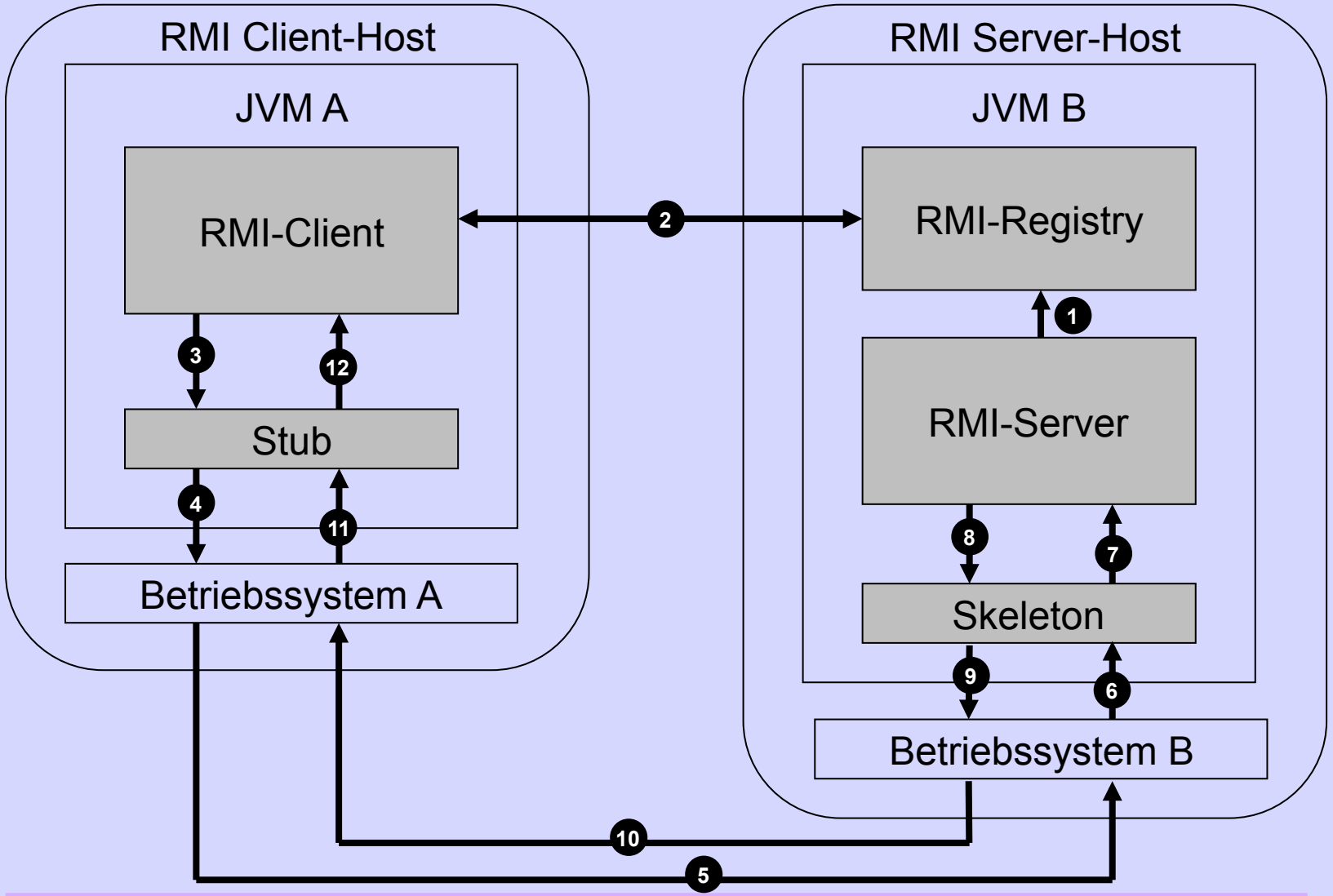
Quelle: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

Entfernter Methodenaufruf mit serverseitig nicht bekannten Objekttyp



Quelle: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html

RMI Ablaufschema



RMI Network garbage collection

- Reference-Count Mechanismus:
  - auf Server-Seite sollen nicht mehr benötigten Objekte Ressourcen freigeben
- Ablauf:
  - Wenn ein Client ein Remote-Objekt benötigt, so erhöht es einen Referenzen-Zähler von diesem Objekt;
  - benötigt es dieses Objekt nicht mehr, so wird dieser erniedrigt.
  - Falls der Reference-count =0 ist, so kann auf Server-Seite das Objekt gelöscht werden.
- Time-out
  - Problem: Bei einem Client Crash kann der Client die Referenz nicht mehr freigeben
  - → Referenzen mit einem Time-out versehen (Standard: 10 Minuten)
- Problem: Keine Garantie für vollständige *referentielle Integrität* möglich.
  - Nicht jede Referenz auf ein Remote-Objekt zeigt auch garantiert auf ein existierendes Objekt (→ Netzwerkverzögerungen)
  - → Werfen einer RemoteException

RMI Network garbage collection

- Zusammenfassung:
  - Die RMI „network garbage collection“ verwendet also einen Reference-count Mechanismus mit Time-outs und stellt keine wirkliche verteilte „garbage collection“ dar.

- CORBA: Problem völlig unspezifiziert
- Microsofts COM/DCOM: Reference-Count Mechanismus, ohne Time-outs

Benutzung von RMI

- Im Folgenden ist die ursprüngliche Version von RMI beschrieben.
  - rmic ist seit Java 2 überflüssig (Stubs werden dynamisch erzeugt)
  - Entfernte Objekte können seit Java 2 auf Anforderung (jeweils in einer eigenen JVM) dynamisch gestartet werden: `remote object activation`
- Aktuelle Beschreibungen siehe:
  - http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html
  - http://java.sun.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html

Benutzung von RMI

- Entfernte Klasse implementiert ein Interface (`Interf`), das das Interface `java.rmi.Remote` erweitert.
  - Alle Remote-Methoden müssen `RemoteExceptions` werfen können.
- RMI Stub compiler `rmic`: Aus der Remote-Implementierung den Stub und das Skeleton ableiten.
- RMI Repository (der „naming service“) mit dem Daemon `rmiregistry` aktivieren.
- Serverprozess: Instanz `obj` der Remote-Klasse erzeugen und beim Repository anmelden:

```
rmi.Naming.bind("name", obj)
```

- Clientprozess: Instanz `Interf obj` des Remote-Interfaces erzeugen:

```
(Interf) rmi.Naming.lookup("rmi://host/method")
```

- Alle Remote-Methoden dieser Instanz können wie lokale Methoden verwendet werden:

```
result = obj.method(parameters)
```

RMI-Registry starten

- Das Methoden-Repository vom RMI heißt **RMI-Registry**.
- Die RMI Registry ermöglicht Clients eine Referenz/Stub eines entfernten Objekts zu erhalten.
- Für RMI Repository (der „naming service“) Daemon auf Server Seite starten mit:

```
rmiregistry [port] &
```

- Default-Port ist 1099
  - Daemon kann also als gewöhnlicher user Prozess laufen.
- Falls Port schon von einem anderen Prozess belegt ist (z.B. der RMIregistry eines anderen Benutzers) so terminiert der `rmiregistry` Daemon mit einer Fehlermeldung.

Remote-Interface definieren

- Entfernte Klasse implementiert ein Interface, das das Interface `java.rmi.Remote` erweitert.
- Alle Remote-Methoden müssen `RemoteExceptions` werfen können. Hintergrund: Hinzugekommene Möglichkeit von Netzwerk-/Serverproblemen.

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Beispiel von http://java.sun.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html

Implementierung des Servers

1. **Instanz** der Remote Klasse **erzeugen und exportieren** um eingehende entfernte Methodenaufrufe empfangen zu können.

```
Server obj = new Server();
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

2. Remote Object bei der JAVA RMI Registry **registrieren.**

   Stub `registry` zur RMI Registry wird benutzt um einen Namen `Hello` an den `stub` des für den entfernten Zugriff bereitgestellten Objekts zu binden.

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
```

Implementierung des Servers - Zusammenfassung

```
package example.hello;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {

    public Server() {}

    public String sayHello() {
            return "Hello, world!";
    }

    public static void main(String args[]) {

            try {
                Server obj = new Server();
                Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

                // Bind the remote object's stub in the registry
                Registry registry = LocateRegistry.getRegistry();
                registry.bind("Hello", stub);

                System.err.println("Server ready");
            } catch (Exception e) {
                System.err.println("Server exception: " + e.toString());
                e.printStackTrace();
            }
    }
}
```

Implementierung des Clients

1. Stub `registry` für den Zugriff auf die RMI Registry des Servers empfangen.

```
Registry registry = LocateRegistry.getRegistry(host);
```

2. Aufruf der entfernten Methoden `lookup` um Stub `stub` des entfernten Objekts von der Registry zu erhalten.

```
Hello stub = (Hello) registry.lookup("Hello");
```

3. Aufruf der entfernten Methode `sayhello` über den `stub` des entfernten Objekts - als wäre es eine lokale Methode.

```
String response = stub.sayHello();
System.out.println("response: " + response);
```

Implementierung des Clients - Zusammenfassung

```
package example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Tools von RMI

- RMI besteht im wesentlichen aus den Klassen einer Klassenbibliothek und benötigt nur zwei Tools:
  - Das Programm `rmiregistry`.
  - Den RMI Stub compiler `rmic`.
    Dieser erzeugt den Client-Stub und das Server-Skeleton.

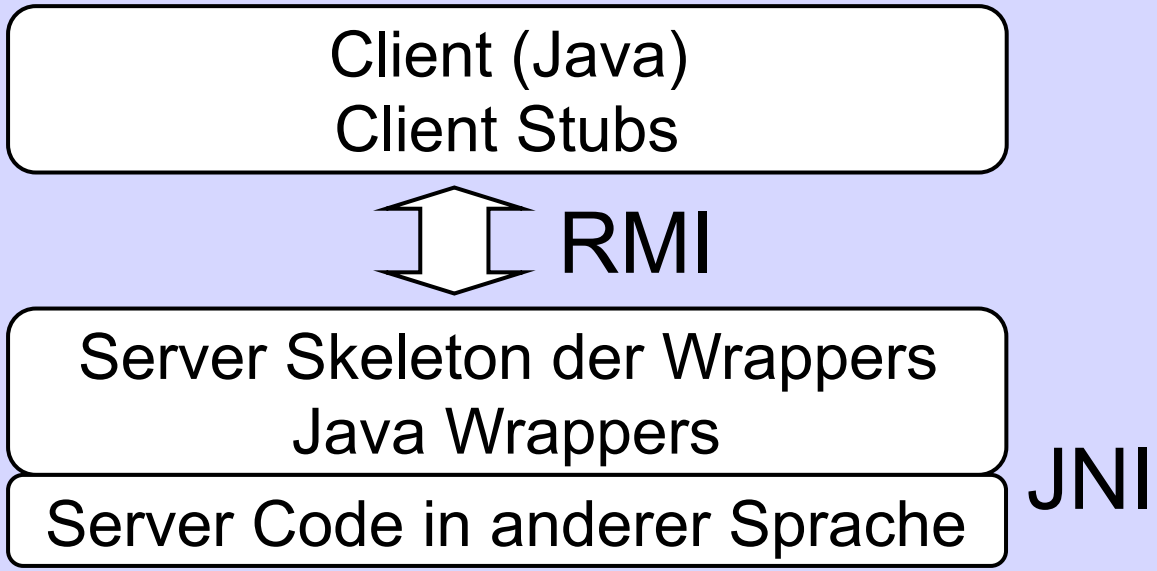
Der `rmic` Stub Compiler

- Der `rmic` Stub Compiler wird auf eine Byte-Code-Compilierte Klasse `MyRemoteClass` angewendet, die ein entsprechendes Remote-Interface implementiert.

- Er erzeugt zwei weitere Klassen:
  - `MyRemoteClass_Skel.class` ist die Skeleton-Klasse, die auf Server-Seite verwendet wird, und
  - `MyRemoteClass_Stub.class` ist die Klasse der Methoden-Stubs, die auf Client-Seite verwendet wird.

RMI und der Zugriff von Java Clients auf nicht-Java Server

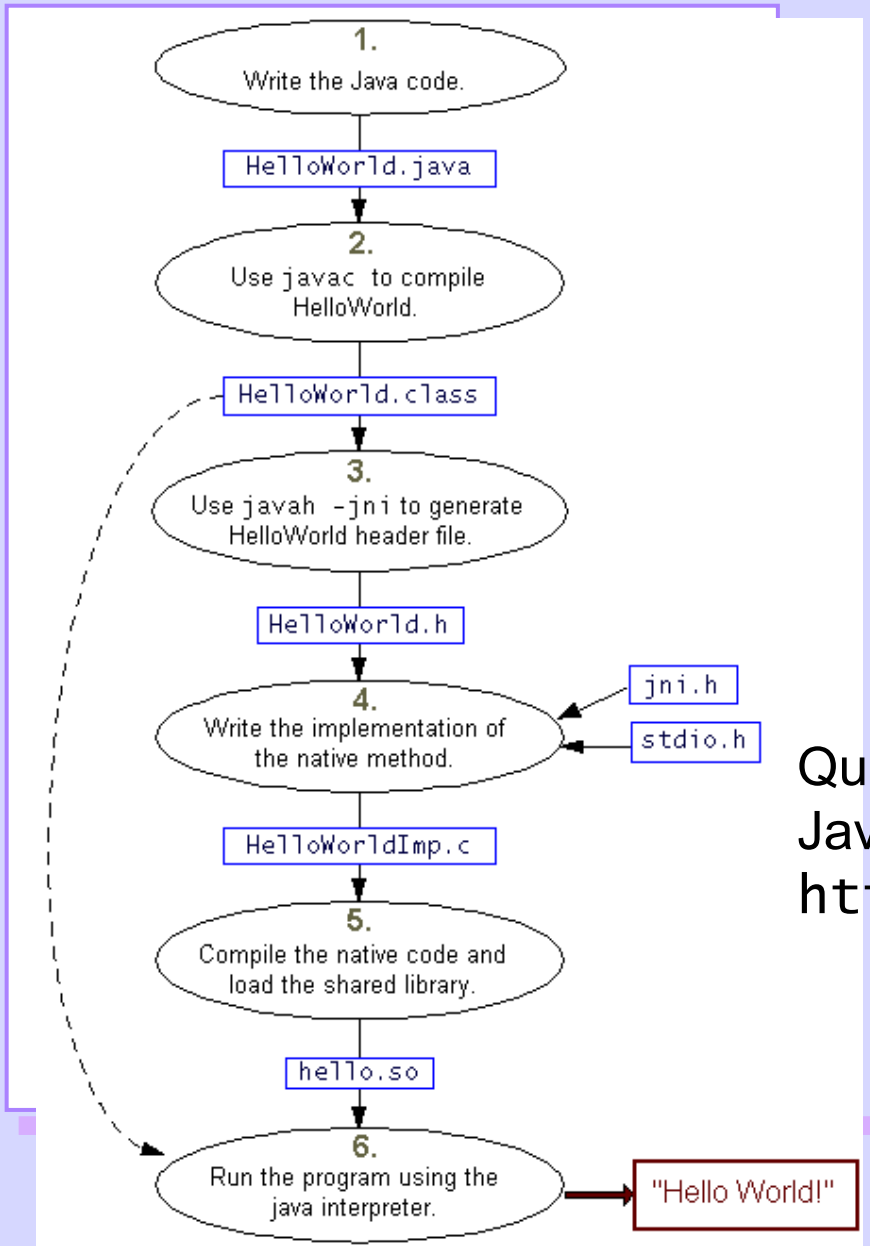
- RMI erwartet auf Client- und Serverseite Java-Code.
- Einwand: Server Java Code kann bei RMI auf Server Seite bleiben
  → Code kann in irgendeiner Programmiersprache geschrieben sein
  → Kapselung mittels des **Java native interfaces (JNI)** (oder `Runtime` Klasse).
  - wichtiges Einsatzgebiet von RMI
  - grundsätzliche Alternative zu CORBA.



Java Native Interface JNI

Exkurs JNI
„Java Native Interface“

Java Native Interface



Quelle:
Java Native Interface Tutorial
http://java.sun.com/docs/books/tutorial/native1.1/stepbystep/index.html

JNI – Schritt 1: Java-Programm schreiben

```
(1) class HelloWorld {
(2)   public native void displayHelloWorld();
(3)
(4)   static { System.loadLibrary("hello"); }
(5)
(6)   public static void main(String[] args) {
(7)         new HelloWorld().displayHelloWorld();
(8)   }
(9) }
```

- Schlüsselwort `native` gibt an, dass Implementierung der Methode in einer anderen Sprache erfolgt.
- Die Implementierung der `native`-Methode wird zu einer Bibliothek kompiliert.
  - Windows: `hello.dll`, Solaris: `libhello.so`
- In Zeile (4) wird die Bibliothek geladen

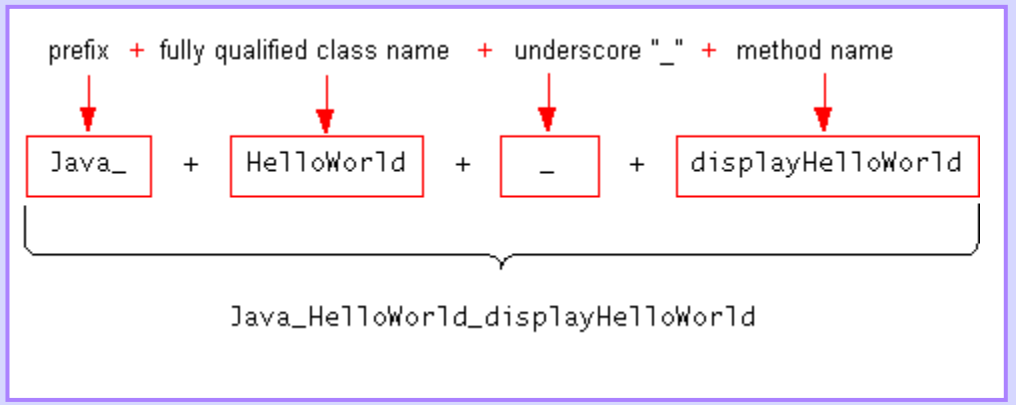
JNI – Schritt 2 & 3

Schritt 2: Java Code kompilieren

- `javac HelloWorld.java`

Schritt 3: Header Datei anlegen

- Aufruf: `javah -jni HelloWorld`
- Erzeugt `HelloWorld.h`
- Methode `Java_HelloWorld_displayHelloWorld` muss implementiert werden.
- Schema der Namensgebung:



JNI – Schritt 3: Header Datei anlegen

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/* Class: HelloWorld
 * Method: displayHelloWorld
 * Signature: ()V
*/
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
      (JNIEnv *, jobject);
#ifdef __cplusplus }
#endif
#endif
```

JNI – Schritt 3: Header Datei anlegen

- Hinweis: `Java_HelloWorld_displayHelloWorld`-Methode hat zwei Parameter, obwohl die `displayHelloWorld()`-Methode keine Parameter hat.
- Die beiden Parameter sind durch JNI vorgegeben
  - `JNIEnv` Interface Pointer: Zugriff auf Parameter, die von Java übergeben werden
  - `jobject`:
    - Referenz auf die Java-Klasse.
    - Entspricht ungefähr dem `this`-Pointer in Java.

JNI – Schritt 4: Implementierung der native-Methode

Implementierung HelloWorldImp.c

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
                                (JNIEnv *env, jobject obj) {
      printf("Hello world!\n");
      return;
}
```

JNI – Schritt 5 & 6

Schritt 5: Anlegen der Bibliothek

- Solaris: Anlegen der Bibliothek `libhello.so`:
  - `cc -G -I/usr/local/java/include -I/usr/local/java/include/solaris HelloWorldImp.c -o libhello.so`
- Windows: Anlegen von `hello.dll` mit Microsoft Visual C++ 4.0:
  - `cl -Ic:\java\include -Ic:\java\include\win32 -LD HelloWorldImp.c -Fehello.dll`

Schritt 6: Anwendung auführen

- Aufruf: `java HelloWorld`
- Ergebnis: `Hello World!`

Java Native Interface

Exkurs „Java Native Interface“ Ende

CORBA

- *Common Object Request Broker Architecture*
- Allgemeiner *Architektur*-Standard für Entwicklung von Client/ Server Anwendungen
- Verschiedene konkrete Implementierungen: ORBs.
- Definiert von der *Object Management Group (OMG)*
  - Zusammenschluss von über 750 Unternehmen, Software-Entwicklern und Anwendern.
  - 1989 gegründet.
- Allgemeine Kommunikationsinfrastruktur zwischen verteilten Objekten, wobei für den Entwickler die Kommunikation weitestgehend transparent ist.

CORBA

- Folgende Merkmale machen CORBA aus:
  1. Objektorientierung
  2. Sprachunabhängigkeit
  3. Kommunikationsmechanismen für verteilte Systeme
  4. Allgemeines Konzept von kommunizierenden Objekten
  5. Plattformunabhängigkeit
  6. Herstellerunabhängigkeit
  7. Anbindung anderer Komponentensysteme
- CORBA geht daher nach folgendem Prinzip vor:
  Die *Schnittstellen* werden von der *Implementierung* streng *getrennt.*
- Durch die separate Definition der Schnittstellen kann die Kommunikation völlig unabhängig von der jeweiligen Implementierung betrachtet werden.
  - **IDL** (*Interface Definition Language*): Neutrale Spezifikationssprache beschreibt die Schnittstellen der beteiligten Objekte
  - **ORB** (*Object Request Broker*): Kommunikation zwischen den Objekten

IDL: Interface Definition Language

Objektorientierung

- Die IDL ist eine objektorientierte, rein deklarative Sprache, die sich nur auf die Schnittstelle eines Objekts bezieht.
- Bildung von Klassenhierarchien möglich
- Syntax ähnlich C++

Sprachunabhängigkeit

- Neutrale Interface Definition Language nötig
- Aus einer IDL-Spezifikation lässt sich Sourcecode für die gängigsten Programmiersprachen generieren.
- Kommunikation kann auch zwischen Objekten erfolgen, die in verschiedenen Sprachen implementiert sind.
- Für nicht-OO Sprachen können Object Wrapper geschrieben werden.

Merkmale von CORBA

Kommunikationsmechanismen

- Kommunikation zwischen den Objekten ist völlig transparent
  - Der ORB verdeckt das Auffinden des Zielobjekts, die Übertragung der Daten, sowie etwaige Konvertierungen zwischen Datenformaten.
- Kommunikation selbst ist plattform- und sprachunabhängig.
  - Seit CORBA 2.0 arbeiten auch Object Request Broker verschiedener Hersteller zusammen (durch das Internet Inter-Orb Protocol - IIOP).

Allgemeines Konzept von kommunizierenden Objekten

- Die bisherige starre Einteilung in Client und Server entfällt, in CORBA existieren gleichberechtigte Objekte.
  - Objekt A kann Server für ein Objekt B sein, während B gleichzeitig Serverfunktionalität für ein Objekt C anbietet.
  - Festlegung: Server ist das Objekt, das ein IDL-Interface implementiert.

Merkmale von CORBA

Plattformunabhängigkeit

- Für alle gängigen Plattformen sind CORBA Implementierungen verfügbar.

Herstellerunabhängigkeit

- Alle bedeutenden Hersteller von Software stehen hinter der CORBA-Architektur, außer Microsoft mit DCOM.

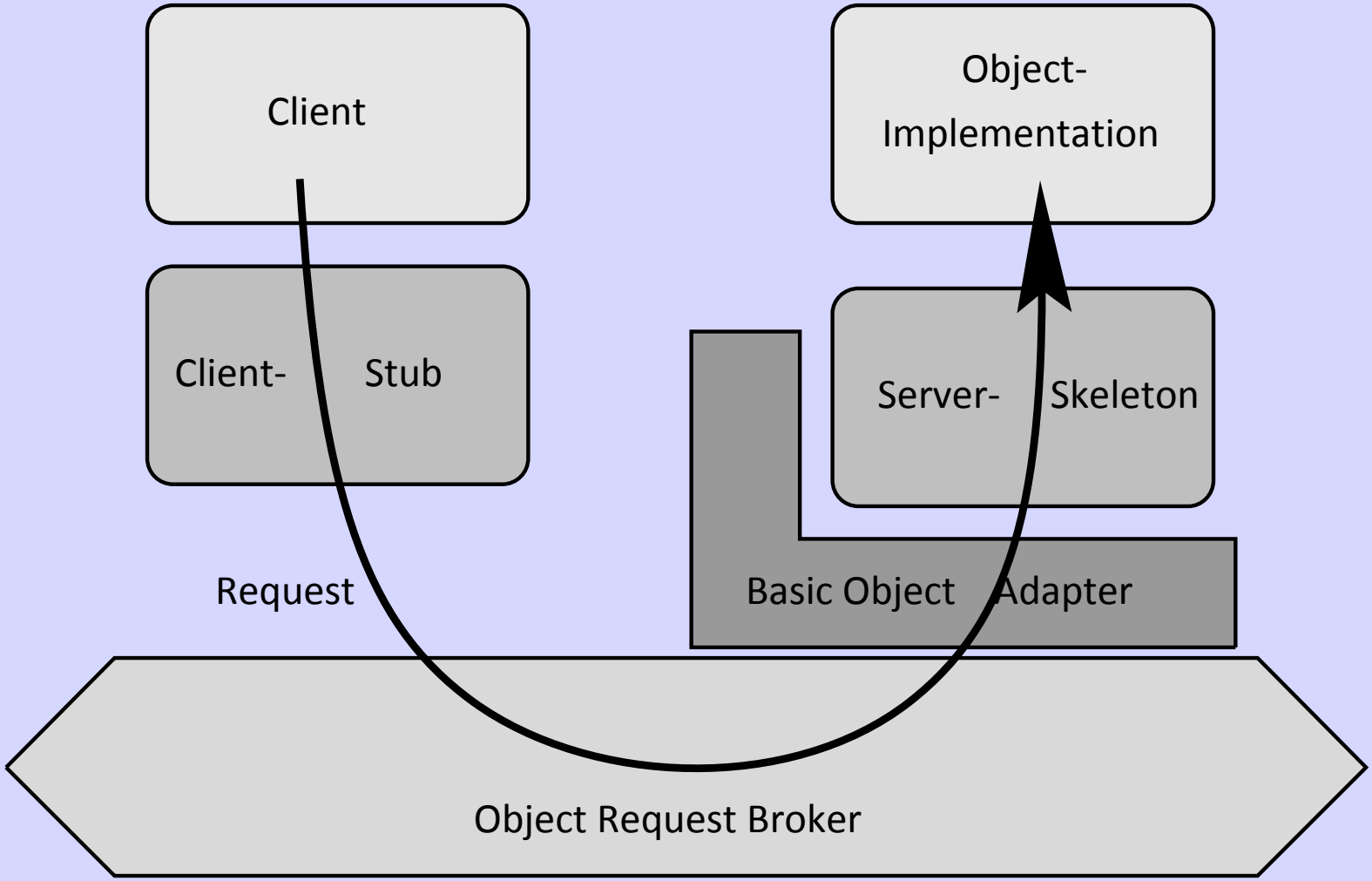
Anbindung anderer Komponentensysteme

- Die Microsoft Komponentensysteme OLE und COM können in CORBA-basierte Systeme integriert werden.

Grundprinzip von CORBA

- CORBA ist nicht einfach ein „objektorientierter Remote Procedure Call“
  - OMG legt nicht nur das Kommunikationsprinzip fest, sondern spezifiziert auch eine Architektur für verteilte Systeme im Intra-/Internet.
- Statt großer, monolithischer und unflexibler Anwendungen ein System von (relativ kleinen) Komponenten, die jeweils spezielle Dienste anbieten.
  - Auffinden des „Dienstanbieters“ und Kommunikation mittels ORB
- Die OMG spezifiziert mit CORBA „nur“ einen Standard, keine Implementierung.
- Um mit einem CORBA-System zu arbeiten, ist es notwendig, von einem Hersteller ein System zu beziehen.

Grundprinzip von CORBA



Grundprinzip von CORBA

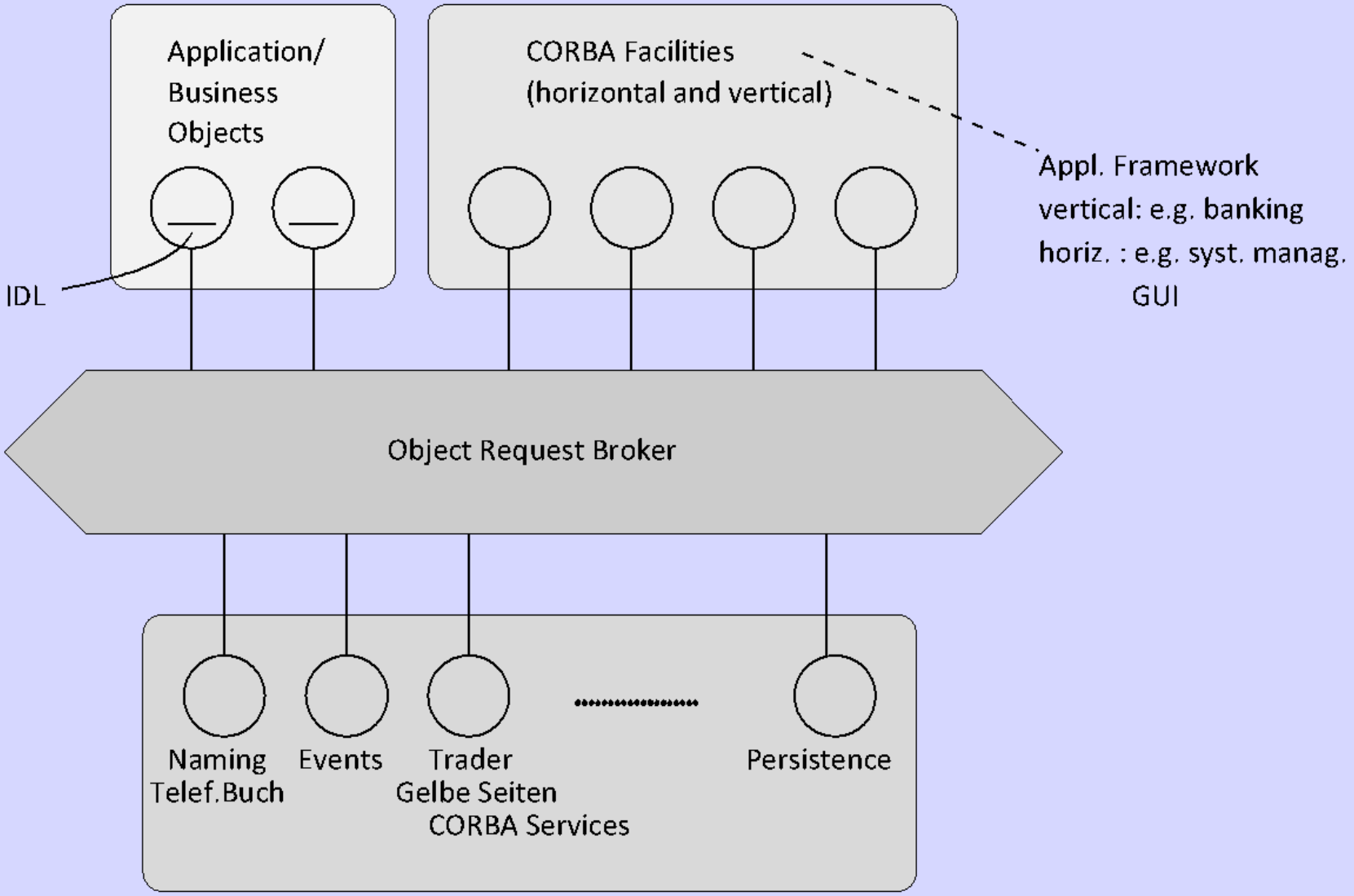
Erklärung zur Abbildung:

- Mit Hilfe der IDL wird ein Interface definiert.
- IDL-Compiler erzeugt aus dieser Schnittstellenbeschreibung Sourcecode in der gewünschten Sprache. Für den Client **Stub** und für den Server **Skeleton**.
- Server wird implementiert und ist über das Skeleton für andere Objekte zugänglich. Über den *Basic Object Adapter* (**BOA**) meldet sich der Server beim ORB an und ist jetzt bereit, Aufrufe anderer Objekte zu empfangen.
- Der Client kann nun über den Stub auf den Server zugreifen. Dieser Zugriff läuft über den ORB.

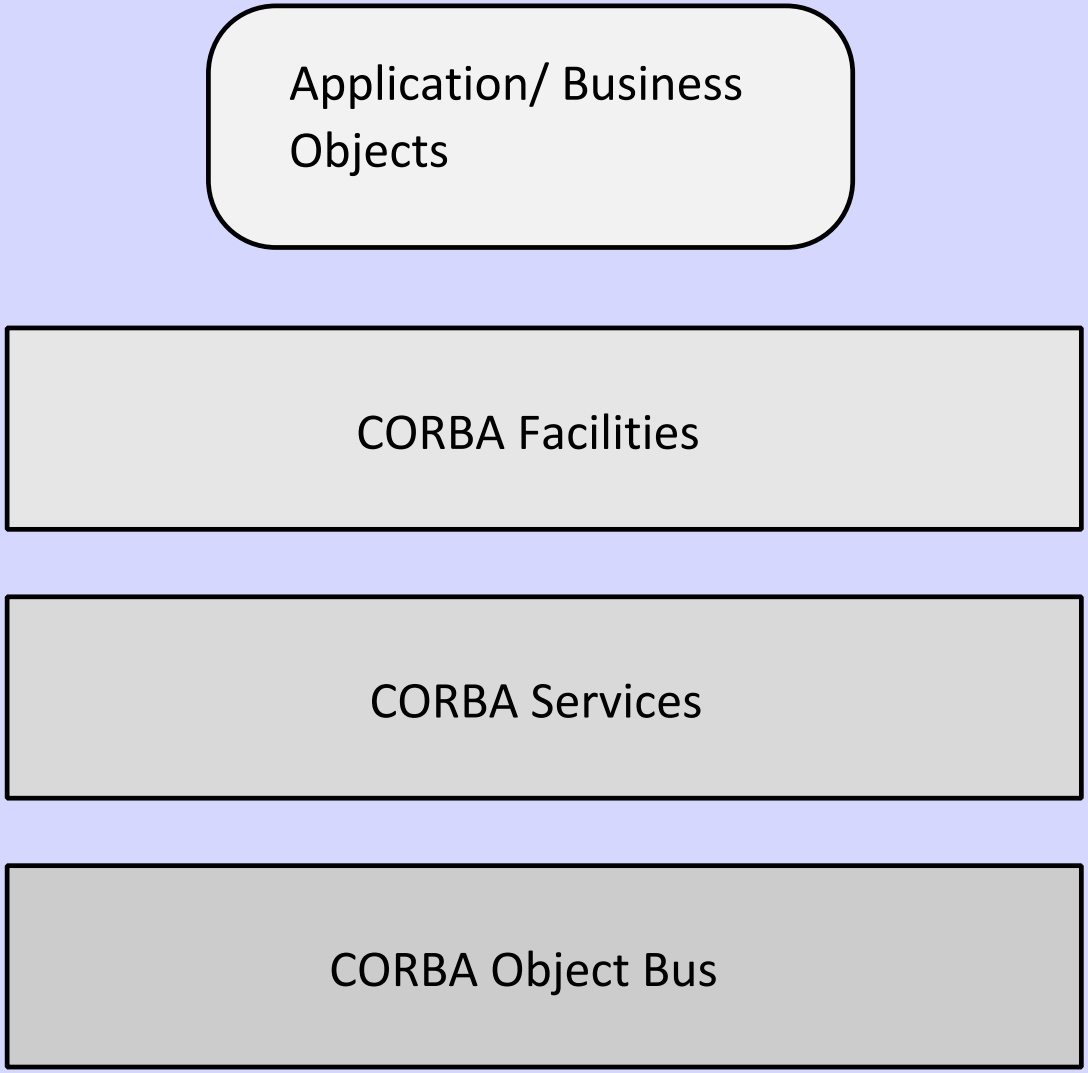
Grundprinzip von CORBA

- Ablauf entfernter Methodenaufruf:
  - Object Request Broker fängt Aufruf ab und lokalisiert das Zielobjekt
  - Übergabeparameter werden verpackt und an den Server geschickt.
  - Dort werden die Parameter wieder entpackt und die Methode auf dem Server ausgeführt.
  - Resultat wird verpackt und an den Aufrufer zurückgesendet.
  - Gesamter Vorgang wird vom ORB verdeckt.
- Client benötigt für entfernten Methodenaufruf eine *Referenz* auf das entfernte Objekt: *Object-Reference*
  - eindeutige ID eines bestimmten Objekts
- Angesprochen wird die Server-Komponente über die automatisch generierte ”Stub-Klasse“ → hohe Typsicherheit

Object Management Architecture



Hierarchie der Object Management Architecture



Object Request Broker

- Object Request Broker (ORB) ist das „Herzstück“ des Komponentensystems.
  - Zuständig für die gesamte Kommunikation der verteilten Objekte.
  - Daher Bezeichnung auch als Object-Bus
- Die meisten ORB’s sind kommerziell
  - z.B. ORBIX von IONA, Visibroker von Visigenic, . . .
- Beispiel frei erhältlicher ORB’s
  - Mico: `http://www.vsb.cs.uni-frankfurt.de/~mico`
  - Java IDL: Im JDK enthalten; `http://java.sun.com/`
  - ILU. `ftp://ftp.parc.xerox.com/pub/ilu/`

Object Request Broker – Grundfunktionalität

Statische und dynamische Aufrufe von Methoden

- Statischer Aufruf: sichere Typprüfung
- Dynamische Aufrufe: höhere Flexibilität

Kommunikation auf Hochsprachen-Niveau

- Es ist nicht notwendig, Parameter „einzupacken“ (marshaling) oder Befehle in einer Kurzform zu übertragen.

Selbstbeschreibendes System

- Jeder ORB besitzt eine Datenbank, das sog. *Interface Repository*.
  - Enthält Meta-Informationen über die bekannten Interfaces
  - Daten werden automatisch verwaltet und gepflegt, i.a. durch den IDL-Compiler.

Transparenz zwischen lokalen und entfernten Aufrufen

- Umwandlungen zwischen Datenformaten (z.B. little endian/big endian, usw.) werden vom ORB durchgeführt.
- Kompatibilität bzgl. Art und Größe der Typen durch IDL gewährleistet.

Object Request Broker – Grundfunktionalität

Polymorphismus

- Implementierungen verschiedener Objekte eines bestimmten Interfaces können sich unterscheiden.
- Somit können Objekte auf denselben Methodenaufruf unterschiedlich reagieren.

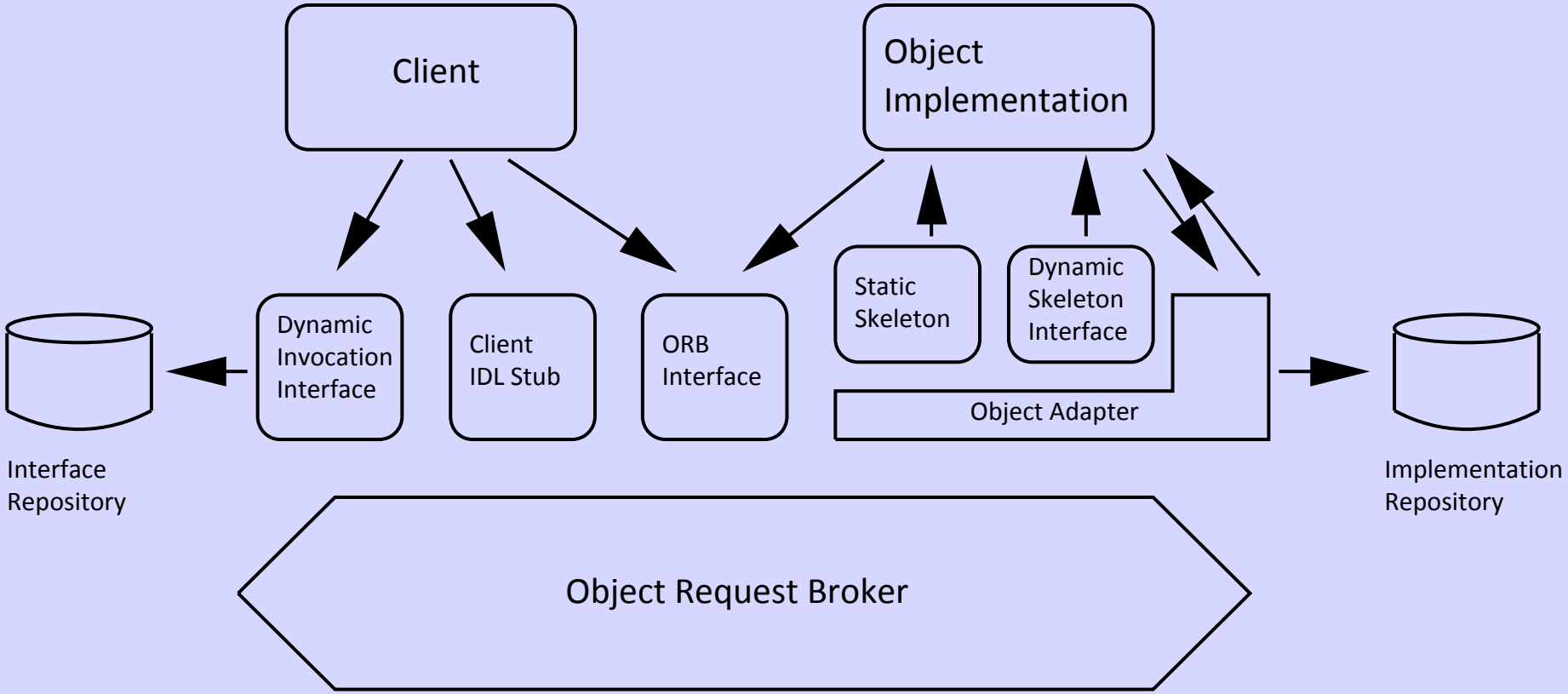
Einbindung bestehender Software

- Mit Hilfe von IDL-Definition Wrapper erstellen, die den bestehenden Code kapseln. → Tie-Approach

Kommunikation zwischen Objekten

- Vorteile (im Vergleich zu z.B. RPC):
  - Objekte kommunizieren miteinander, es werden nicht nur entfernte Methoden ausgeführt
  - Objekt kann einen Status besitzen, der zwischen zwei Client-Aufrufen erhalten bleibt.

Aufbau des CORBA ORB



Aufbau des CORBA ORB

Erklärung zur Abbildung: Relevante Teile für den Client

- Client IDL Stubs
  - Beim **statischen** Aufruf als sog. Proxy-Objekt (oder Stellvertreter-Objekt) „wirken“ die Stubs wie ein lokales Objekt.
  - Übergabeparameter werden automatisch in ein Nachrichtenformat überführt, um sie an das Server-Objekt senden zu können.
  - Dieser Code wird vom IDL-Compiler automatisch erzeugt.
- Dynamic Invocation Interface (DII)
  - Mit DII können Methoden aufgerufen werden, deren Interfaces zur Compilierzeit noch nicht bekannt waren.
- Schnittstelle zum Interface Repository
  - Datenbank mit sog. Method Signatures
    - maschinenlesbare Versionen der IDL-Definitionen.
  - Repository wird beim dynamischen Methodenaufruf benutzt
  - macht das System selbstbeschreibend.

Aufbau des CORBA ORB

Erklärung zur Abbildung: Relevante Teile für den Server

- Server IDL Stubs (Skeletons)
  - realisieren die Serverseite des statischen Interfaces
  - ebenfalls vom IDL-Compiler erzeugt
- Dynamic Skeleton Interface (DSI)
  - Analog zum Dynamic Invocation Interface beim Client
  - ankommende Requests können an Komponenten weitergeleitet werden, auch wenn diese kein statisches IDL-Interface anbieten.
- Object Adapter
  - auf der Serverseite für die Anbindung und Registrierung der Server-Implementierungen zuständig
  - Ankommende Requests werden mit Hilfe des Implementation Repository an das richtige Objekt weitergeleitet.
  - der sog. Basic Object Adapter (BOA) muss implementiert sein, zusätzlich sind aber auch noch andere erlaubt.
- Implementation Repository
  - Dieses Repository enthält Laufzeitinformationen über die Objekte, die ein Server implementiert, sowie deren eindeutige Bezeichner.

Beispiel für eine IDL Definition

```
module CA {

   typedef sequence<unsigned long> Number;
   struct BigInteger {
      boolean sign;
      Number digits;
   };

   typedef sequence<long> PowerProduct;

   struct Monomial {
      BigInteger coefficient;
      PowerProduct pp;
   };

   typedef sequence<Monomial> Polynomial;
```

Beispiel für eine IDL Definition

```
interface SAClib {
   void Init();
   void End();

   void PolyGCDC(in Polynomial poly1, in Polynomial poly2,
          out Polynomial gcd, out Polynomial cofac1,
          out Polynomial cofac2);

   Polynomial PolyGCD(in Polynomial poly1, in Polynomial poly2);

   Polynomial PolyQuotient(in Polynomial poly1,
          in Polynomial poly2);

   void PolyQuotientRemainder(in Polynomial poly1,
          in Polynomial poly2, out Polynomial quotient,
          out Polynomial remainder);
};
```

Beispiel für eine IDL Definition

```
   interface Factory {
      SAClib CreateSAClib();
      void StopCAServer();
   };
};
```

Common Object Services

Common Object Services (CORBAservices)

- Reihe von Diensten, die das Entwickeln von CORBA basierten Systemen erleichtern und standardisieren sollen.

16 verschiedene Dienste:

Life Cycle Service

- Methoden für das Erzeugen, Kopieren, Verschieben und Löschen von Objekten

Persistence Service

- CORBA-Objekte dauerhaft in einer Datenbank sichern.

Naming Service

- Beziehung zwischen Namen und einer Object-Reference möglich (sog. Name binding).
- Client kann damit Server-Objekt lokalisieren

Common Object Services

Event Service

- Übertragung von Events (sog. Event-Channel).
- Objekte können sich Events registrieren lassen und werden benachrichtigt falls ein solcher Event eintritt
- keine direkte Kommunikation zwischen Erzeuger und Empfänger von Events, denn Erzeuger schickt Event an Event-Channel.

Notification Service (Erweiterung, 2000):

- Event-Filtering Mechanismus
- Notification-Channel

Common Object Services

Concurrency Service

- Verwaltung von Sperren für Transaktionen oder für Synchronisation bei parallelen Programmen

Transaction Service

- Zwei-Phasen-Transaktionen, die auch verschachtelt sein können.

Relationship Service

- Erstellung dynamischer Assoziationen zwischen Komponenten
- Weiterhin z.B. Funktionen, um den entstandenen Graphen zu traversieren.

Common Object Services

Externalization Service

- Daten in Streams schreiben, bzw. Daten aus Streams lesen

Query Service

- Query Operationen über Objekten angeboten.
- Basiert auf SQL3 und der Object Query Language (OQL) der Object Database Management Group (ODMG).

Licensing Service

- einzelne Komponenten eines Systems können lizenziert werden

Properties Service

- Objekten können Eigenschaften zugeordnet und abgefragt werden

Time Service

- Synchronisation in einem verteilten System
- Events zu bestimmten Zeitpunkten

Common Object Services

Security Service

- Authentifizierung, Verwaltung von Zugriffsberechtigungen, usw.

Trader Service

- Zuordnung zwischen Eigenschaften von Objekten (sog. Properties) und deren Referenzen.
- Objekte können sich unter Angabe ihrer angebotenen Dienste registrieren lassen und Clients können passende Dienstleister erfragen.
- „Gelbe Seiten“

Collection Service

- Verwaltung der gängigsten Collections: queues, stacks, trees, usw.

➢ Idee: Neue Systeme können über das Intranet mit bereits bestehenden, bzw. zugekauften Komponenten kommunizieren und deren angebotene Dienste in Anspruch nehmen

Vergleich CORBA mit RMI

- RMI und CORBA grundsätzlich sehr ähnliche Architekturen
- Hauptunterschied:
  - RMI ist Java spezifisch,
  - CORBA kann auch Objekte in verschiedenen Sprachen miteinander verbinden.
- Einwand:
  - JNI: Grundsätzliche Möglichkeit, ein Nicht-Java System als Server-Komponente via RMI zu verwenden.
- Beispiel für technische Unterschiede:
  - CORBA: Naming Service erlaubt hierarchische Gliederung
  - RMI: registry wird über eine URL angesprochen → „flache Struktur“.

Vergleich CORBA mit RMI

- Szenarien/relevante Punkte bei denen CORBA im Vorteil ist
  - Vielschichtige verteilte Systeme, bei denen an vielen Stellen Nicht-Java Code verwendet wird.
  - Auf Client-Seite wird Java verwendet, auf Server Seite C++.
  - Granularität der Berechnungen bei Server-Komponenten ist nicht sehr grob.
  - CORBA stellt sehr viel mehr Dienste zur Verfügung als RMI.
  - CORBA kann Dienste unterschiedlicher Sprachen und Systeme effizient sogar in einem Prozess vereinigen.
- Vorteile von Java RMI gegenüber CORBA.
  - Java Anbindung direkter/eleganter als das CORBA Java language mapping.
  - Stellt integrierte Dienste wie die Network garbage collection zur Verfügung.

Referentielle Transparenz

- Client, der ein Server-Objekt über CORBA aufruft, darf nicht wissen, wo sich das Server Objekt befindet.
- Es gibt folgende Möglichkeiten:
  - Server-Objekt befindet sich auf anderem Rechner.
  - Server-Objekt befindet sich auf gleichem Rechner.
  - Server-Objekt befindet sich sogar im gleichen Betriebssystemprozess (Adressraum)
- bei spezielleren Fällen effizientere Kommunikationsmechanismen benutzen als für die allgemeineren
  - Kommunikation über Shared memory statt über Sockets
  - direkter Funktionsaufruf über einen Funktionszeiger statt ein entfernter Funktionsaufruf

Referentielle Transparenz

- Client Objekt weiß nicht, wie effizient ein Aufruf eines Server-Objekts ist;
  - Positive und negative Aspekte
- positiver Aspekt:
  - Bei manchen Anwendungen ist es nicht a priori klar, ob sie in einem gemeinsamen Prozess ohne gegenseitige Störung laufen können
    - beide benutzen z.B. garbage collection, sind multi-threaded, benutzen gar eine verteilte garbage collection oder benutzen Signale
- Vorgehen via CORBA:
  - Applikationen via CORBA kapseln
  - Funktionalität der einen Applikation der anderen zur Verfügung stellen
  - Testen, ob diese gekapselten Objekte in einem Prozess lauffähig sind
  - Falls nicht: Objekte in verschiedenen Prozessen auf dem gleichen Rechner halten
  - Kommunikation etwas langsamer, aber die Anwendung läuft und keine Änderungen des Design nötig

Referentielle Transparenz

IPGCDC statistics	Example 1 (Liu) (times in ms)			Example 2 (Bini) (times in ms)		
	#1	#2	#3	#1	#2	#3
number of remote calls	546	546	546	28	28	28
av. total calling time	405	159	99	177028	177889	174001
av. conv. time per call (client)	25	25	3	11	12	5
av. conv. time per call (server)	46	46	8	19	12	10
av. comm. time per call	247	0.04	0.03	50	0.04	0.04
av. execution time of IPGCDC	87	88	88	176948	177865	173986
overhead due to comm. and conv. % of total calling time	79	45	11	0.05	0.01	0.01

#1: separate address spaces, conversion via decimal format
#2: common address space, conversion via decimal format
#3: common address space, binary integer conversions

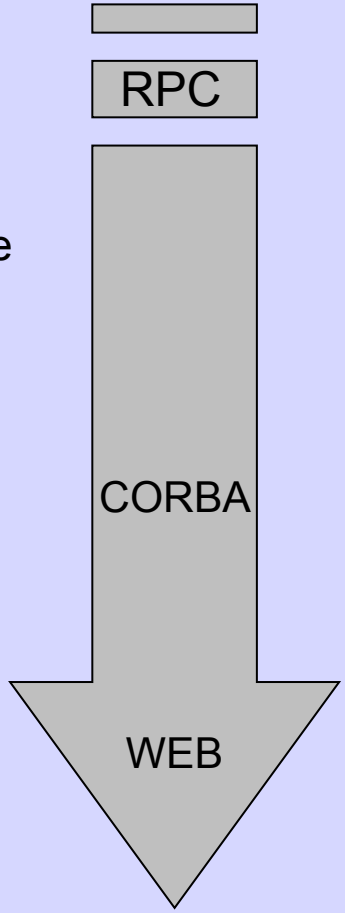
Referentielle Transparenz

IPQ statistics	Example 1 (Liu) (times in ms)			Example 2 (Bini) (times in ms)		
	#1	#2	#3	#1	#2	#3
number of remote calls	712	712	712	22	22	22
av. total calling time	253	91	54	154	77	52
av. conv. time per call (client)	26	26	3	28	28	11
av. conv. time per call (server)	18	17	3	14	23	13
av. comm. time per call	161	0.03	0.03	85	0.04	0.04
av. execution time of IPQ	48	48	48	27	27	27
overhead due to comm. and conv. % of total calling time	81	47	11	82	66	46

#1: separate address spaces, conversion via decimal format
#2: common address space, conversion via decimal format
#3: common address space, binary integer conversions

Gründe für den Niedergang von CORBA

- Technische Gründe
  - Komplexität von CORBA, API
  - Unterstützung von C++ fehlerträchtig
  - Sicherheitsmängel
    - unverschlüsselter Datenaustausch
    - pro Dienst ein offener Port in Firewall erforderlich
  - Fehlende Abwärtskompatibilität für auf CORBA aufbauende Software
  - Viel Redundanz, keine Kompression
  - Keine Thread-Unterstützung
  - Fehlende Unterstützung von C#, .NET, DCOM
  - Fehlende Integration des Web
    - Aufkommen von XML, SOAP, SOAP-RP
    - → E-business Lösungen generell ohne CORBA
- Soziale Gründe / Verfahrensfehler
  - Zu viele Köche im Standardisierungsprozess
  - Z.T. Fehlende Referenzimplementierungen
  - Ungetestete Innovationen in Standards
  - Hohe Lizenzgebühren für kommerzielle Impl.
  - open-source Implementierungen zu spät
  - Mangel an erfahrenen Entwicklern



Quelle: Michi Henning, ACM Queue, http://www.acm.org/acmqueue/digital/protected/Queuevol5no4_May2007.pdf