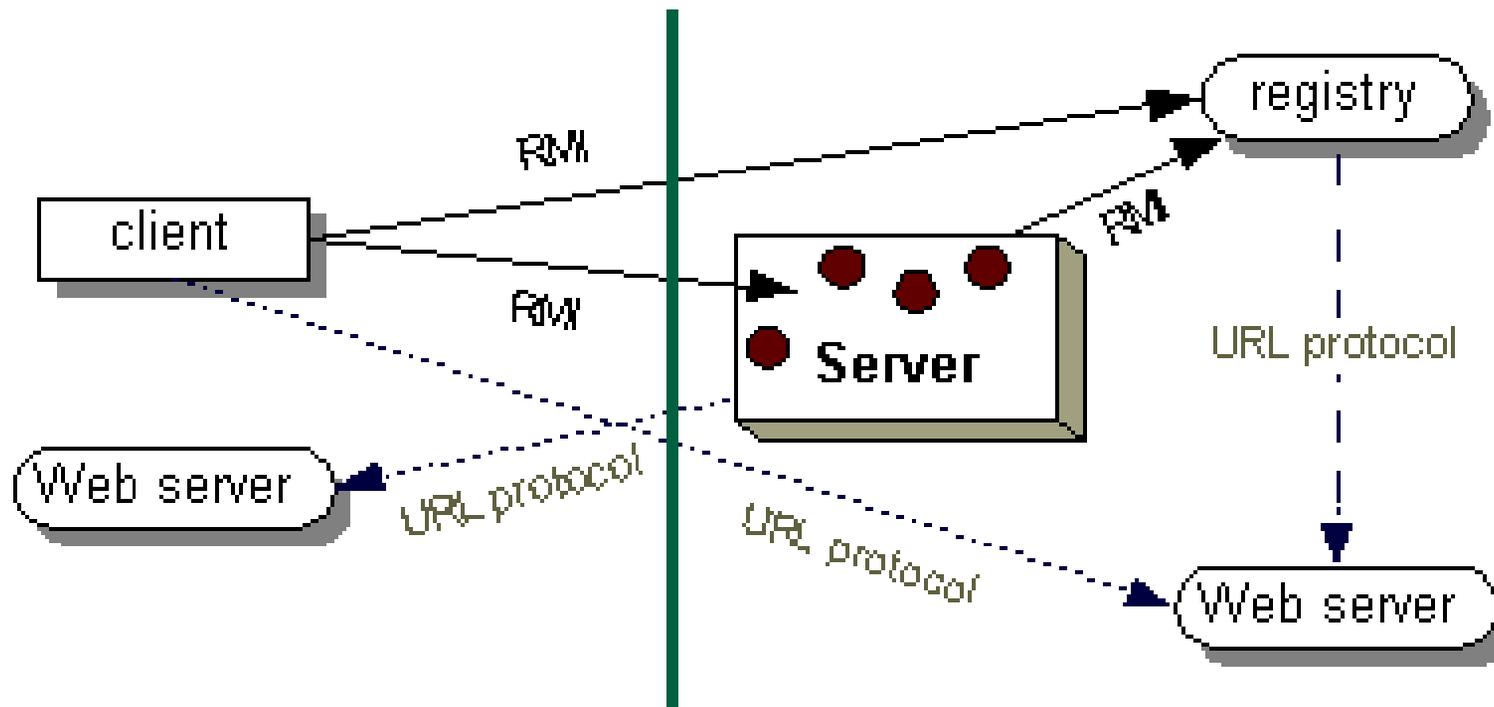


7 Remote Method Invocation (RMI)

- Verteilte Java Anwendungen; Client/Server Architektur



Aufgaben

- Server:
 - erstellt *remote objects*
Objekte, deren Methoden von einer anderen JVM aufgerufen werden können
Beschreibung durch *remote interfaces*
 - publiziert Referenzen darauf
- Client:
 - verschafft sich Referenzen auf *remote objects*
 - ruft dort Methoden auf
 - versendet lokale Objekte inklusive Kode

Anforderungen

- *remote objects* lokalisieren: Nameserver `rmiregistry` muss vor Server auf gleichem Rechner gestartet werden
- Kommunikation mit *remote objects*:
abgebildet auf Methodenaufruf
- Dynamisches Laden des Codes für Argumente und Ergebnisse
 - Typen bleiben erhalten
 - neue Typen auf der Client-Maschine
 - Versand via Webserver

Überblick

- Packages `java.rmi`, `java.rmi.server`
- Definition des “remote interface”
in Klasse `hellointerface.HelloInterface`
- Implementierung der Serverklassen
in Klasse `helloimplement.Hello`
- Implementierung der Clientklassen
in Klasse `helloclient.HelloClient`
- Objekt Serialisierung

7.1 Remote Interface

```
package hellointerface;
public interface HelloInterface
        extends java.rmi.Remote {
    String sayHello(String what)
        throws java.rmi.RemoteException;
}
```

- Protokoll = Methoden des Interface
- Muss auf Client und Server bekannt sein
- Nur Methoden der Remote Interfaces eines Objekts sind auf dem Client aufrufbar
- Nur Server enthält Implementierung der Remote Interfaces
- Typen im Remote Interface auf Client und Server bekannt

Argument und Ergebnisse im Remote Interface

- Primitive Typen
- Remote Object
 - Muss als (remote) Interfacetyp deklariert werden, *nicht* als Klassentyp.
 - Für ein Remote Object wird nur ein Stub übergeben.
 - Referenz: Stub agiert als Proxy
- Lokales Objekt
 - Muss serialisierbar sein (`java.io.Serializable`)
 - Wird als Kopie übergeben, Überlappung (sharing) bleibt erhalten

Wohin mit dem Remote Interface

- Muss bei Übersetzung von Server und Client verfügbar sein:
- Kompilierten Code für remote Interface archivieren
`jar cvf hellointerface.jar hellointerface/*.class`
- Verzeichnis für `hellointerface.jar` wählen
sollte netzweit per HTTP-URL zugreifbar sein, z.B.
`/home/server/public_html/classes/`
- Kompilieren von Client- und Servercode mit
`-classpath ./home/user/public_html/classes/hellointerface.jar`

7.2 Implementierung des Servers

```
package helloimplement;
```

```
import java.rmi.*;
```

```
import java.rmi.server.UnicastRemoteObject;
```

```
public class Hello
```

```
    extends UnicastRemoteObject
```

```
    implements hellointerface.HelloInterface {
```

- erweitert den Server `UnicastRemoteObject`
 - einfaches Remote Object mit TCP-Kommunikation
 - ständig laufender Server
- implementiert gewünschtes Remote Interface

Konstruktor des Remote Object

```
private String name;
```

```
public Hello(String s) throws RemoteException {  
    super();  
    name = s;  
}
```

- **muss** Konstruktor von `UnicastRemoteObject` aufrufen:
 - “exportiert” das Objekt
 - erzeugt einen Server, der an einem anonymen Port auf Methodenaufrufe für `Hello` wartet
- `java.rmi.RemoteException`: ausgelöst, falls Export des Objekts nicht möglich oder keine Netzwerkressourcen vorhanden

Implementierung des Remote Interface

```
public String sayHello(String what) throws RemoteException {  
    System.out.println ("I got " + what);  
    return (name + " says: " + what);  
}
```

- Methode aus HelloInterface
- Überlappung (sharing) von nicht-remote Objekten bleibt erhalten
- Methode muss thread-safe sein
abhängig von der Implementierung von UnicastRemoteObject

Startmethode des Servers

```
public static void main(String args[]) {  
    System.setSecurityManager(new RMISecurityManager());  
}
```

- `SecurityManager` wacht darüber, daß keine illegalen Operationen ausgeführt werden
- auch selbstdefinierter `Security Manager` möglich
- ohne `Security Manager`: keine RMI Klassen möglich
- muss durch Datei und Properties konfiguriert werden

... Erzeugen des Remote Objects

```
try {  
    Hello obj = new Hello("HelloServer");
```

- mehrere Objekte möglich
- Objekt "hört", sobald es konstruiert ist
- ... aber keiner weiß es

Publikation des Remote Objects

```
Naming.rebind("//host/HelloServer", obj);  
System.out.println("HelloServer bound in registry");
```

- *host*: Hostname oder IP-Adresse des Serverrechners
- Verzeichnisdienst für remote objects:
//host[:port]/string
- “URL ohne Schema”
- Standardport des Verzeichnisdienstes: 1099
- Remote Objects verlassen niemals ihren eigenen Adressraum
- Registration nur direkt auf Serverrechner (Sicherheit!)
- Anfrage an Verzeichnisdienst \Rightarrow Referenz auf das Remote Object

7.3 Starten des Servers

Der Server startet nur unter folgenden Bedingungen korrekt:

1. Der Verzeichnisdienst muss bereits gestartet sein
2. Die Server-JVM muss die Erlaubnis haben, Sockets, ServerSockets und HTTP-Verbindungen herzustellen
3. Die Stub- und Skeletonklassen müssen erzeugt und per bekannter URL verfügbar sein

7.3.1 Starten des Verzeichnisdienstes

> cd /tmp

> unsetenv CLASSPATH

> rmiregistry &

- Achtung! Der Verzeichnisdienst darf die Implementierungsklassen der Remote Objects nicht über seinen CLASSPATH finden
- Daher:
 - Dienst in neutralem Directory starten
 - CLASSPATH löschen
- caches interfaces

- restart on change

7.3.2 Konfiguration des Netzzugriffs

- Konfigurationsdatei (in Datei `java.policy`)

```
grant {  
    permission java.net.SocketPermission  
        "*:1024-65535", "connect,accept";  
    permission java.net.SocketPermission  
        "*:80", "connect";  
};
```

- Definition der security policy beim Start des Servers

```
> java -Djava.security.policy=java.policy ...
```

7.3.3 Erzeugung der Stubklassen

- `rmic` erzeugt (client) stubs und (server) skeletons aus class files (Bsp: `helloimplement.Hello`)
 - Client Stub-Klasse
 - * implementiert alle Remote Interfaces der Klasse
 - * serialisiert Argumente
 - * implementiert RMI Protokoll
 - Server skeleton
 - * führt Deserialisierung durch
 - * ruft Methode auf
- Ausgabe: class files

Anwendung von rmic

- Erzeugen von Serverstubs (und ggf. Skeletons)

```
rmic -d /home/server/public_html/classes/ helloimplement.Hello
```

- Stubs sind per HTTP-URL verfügbar

```
http://myhost/~server/classes/...
```

Diese URL muss als `java.rmi.server.codebase` Property beim Starten des Servers angegeben werden

- Publizieren des restlichen Codes der Implementierung

```
cd /home/server/public_html/classes/  
jar xvf hellointerface.jar
```

7.3.4 Aufruf des Servers

```
java -Djava.rmi.server.codebase=http://localhost/~server/classes/ \  
    -Djava.rmi.server.hostname=localhost \  
    -Djava.security.policy=java.policy \  
    helloimplement.Hello FlashGordon
```

- Achtung! Der “/” am Ende der codebase Property ist unbedingt erforderlich.

7.4 Client für das Remote Object

```
package helloclient;
import java.rmi.*;

public class HelloClient {
    public static void main (String args[]) {
        String message = "";
        System.setSecurityManager (new RMISecurityManager ());
        try {
            hellointerface.HelloInterface hello =
                (hellointerface.HelloInterface)
                Naming.lookup ("//localhost/HelloServer");
            for (int i=0; i<args.length; i++) {
                message = hello.sayHello (arg[i]);
                System.out.println (message);
            }
        } catch (Exception e) { e.printStackTrace (); }
        return;
    }
}
```

Verwendung

1. `Naming.lookup ("//host/service")`
 - kontaktiert den Verzeichnisdienst auf *host* und fragt dort nach *service*
 - falls er existiert, so liefert `lookup` ein Stub-Objekt, das mit dem *service* Objekt verbunden ist
 - die entsprechende Klasse wird ggf. noch geladen
2. `hello.sayHello ()` aktiviert das Stub-Objekt
 - serialisiert die Parameter und verschickt sie
 - empfängt die Antwort (warten)
 - deserialisiert sie in ein Objekt entsprechend des Ergebnistyps `String`
3. Anzeige durch `println ()`

Aufruf des Clients

```
java -Djava.security.policy=java.policy \  
    helloclient.HelloClient \  
    localhost Ich wünsch mir ne kleine Miezekatze
```

```
FlashGordon says: Ich  
FlashGordon says: wünsch  
FlashGordon says: mir  
FlashGordon says: ne  
FlashGordon says: kleine  
FlashGordon says: Miezekatze
```

7.5 Einbettung in Applet

```
<applet codebase="http://myhost/~server/classes/"  
        code="helloclient.HelloApplet"  
        width=500 height=120>  
</applet>
```

- webserver muss = RMI Objektserver
- Clientcode auch über Codebase zugreifbar
- codebase muss mit "/" enden!
- code Attribut muss voll qualifiziert sein (Pfadname):
 - HTML Seite in `http://myhost/ server/`
 - Class files in
`http://myhost/ server/classes/helloclient/`

Vorspann und Nachspann

```
package helloclient;
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {
    String message = "blank";

    public void paint(Graphics g) {
        g.drawString(message, 25, 50);
    }
    // init()
}
```

RMI Aufruf aus Applet

```
public void init() {
    String srv = "://" + getCodeBase().getHost() + "/HelloServer";
    try {
        hellointerface.HelloInterface hello =
            (hellointerface.HelloInterface) Naming.lookup(srv);
        message = hello.sayHello("It's me!");
    } catch (Exception e) {
        System.out.println("HelloApplet exception: " +
            e.getMessage());
        e.printStackTrace();
    }
}
```

7.6 Weitere RMI Features

- Firewall verhindert direkte Socketkommunikation
 - RMI kann Nachrichten in HTTP POST-Requests verpacken
 - geschieht transparent für Anwendung und Server
- Aktivierbare Objekte
 - Remote Objects, die auf Anforderung (bei Methodenaufruf) erzeugt werden
 - können sich selbst wieder deaktivieren
 - Standardzustand: passiv
 - Benötigt activation daemon (`rmid`)