

8 Java RMI

(Fortsetzung:)

8.4 Mobiler Code	2
8.5 Sicherheit	14
8.6 Hochladen von Code	24
8.7 Herunterladen von Code	29
Zusammenfassung	31

8.4 Mobiler Code

Ein Objekt besteht aus Daten und Code (gemäß seiner Klasse).

Wenn beim Fernaufruf ein **Fernverweis** übertragen wird,
braucht kein Code übertragen zu werden:
am Zielort ist der *Vertretercode* vorhanden
oder wird im Fluge aus der Schnittstelle erzeugt
(die am Zielort notwendigerweise vorhanden ist!).

Wenn eine **Objektkopie** übertragen wird,
ist der Code (`.class`) am Zielort womöglich unbekannt.
Die serialisierten Daten werden übertragen;
den Code muss der Empfänger auf anderem Weg erhalten !

`java.io.Serializable` sind beispielsweise

- `Number` mit Unterklassen `Integer`, ...
- `String`, `StringBuffer`
- alle Felder
- `java.util.ArrayList`, `.LinkedList`, `.HashSet`, ...
- `java.rmi.server.RemoteStub`, `.UnicastRemoteObject`, ...
- `class myClass implements Serializable` ...
Dieser Code wird u.U. nicht beim Empfänger bekannt sein !

- Der Klassename und die `serialVersionUID`* werden mit der Objektkopie übertragen.
- Die empfangende JVM sucht die Klasse im *classpath*.
- Wenn Sender und Empfänger das gleiche Dateisystem sehen, kann das Problem eventuell mit `java -cp <classpath>` gelöst werden.
- Wenn nicht, **Fernladen** von benötigten `.class`- oder `.jar`-Dateien über das Web:

* siehe `java.io.Serializable`

Fernladen:

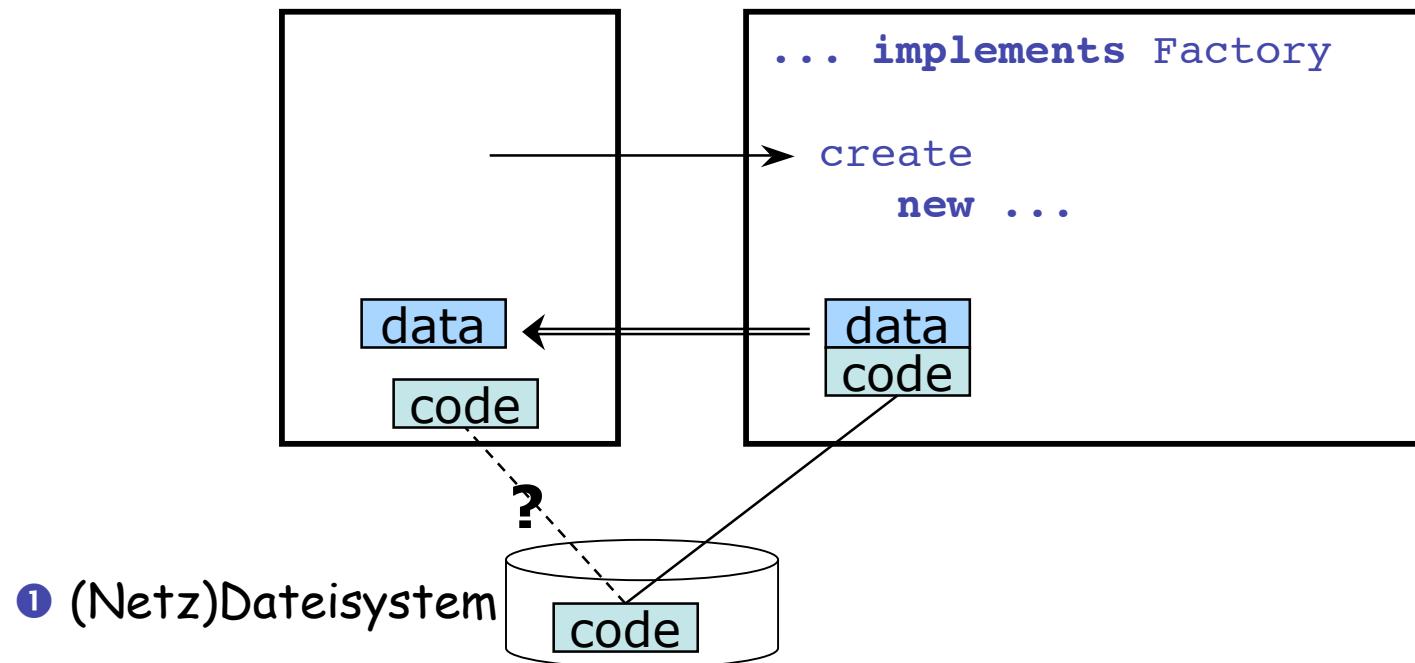
Situation: `myProgram` erzeuge Objekte von nur lokal bekannten Klassen, die nicht `Remote`, aber `Serializable` sind, und diese Objekte werden als Argumente oder Ergebnisse von Fernaufufen verschickt.

Maßnahme: Die benötigten `.class`- oder `.jar`-Dateien über einen Web Server verfügbar machen und beim Programmstart die Web-Adresse als **Codebasis** (`codebase`) angeben, z.B.

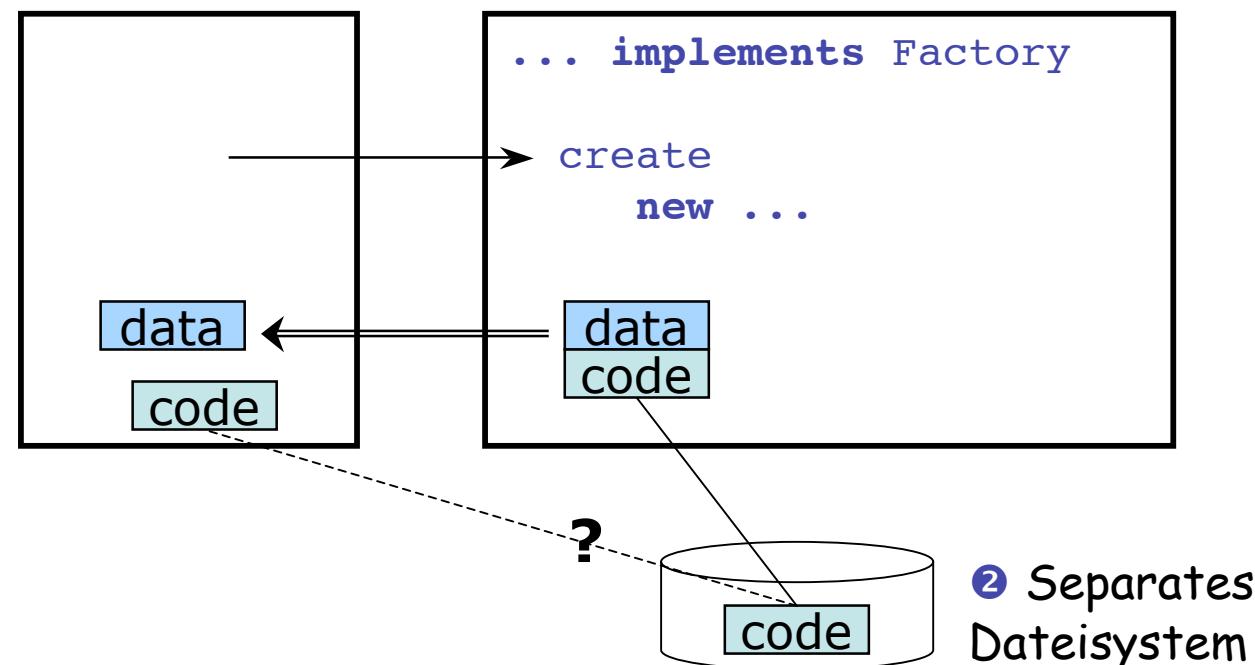
```
$ java -Djava.rmi.server.codebase=\n> http://page.mi.fu-berlin.de/~lohr/classes/ \\n> myProgram
```

(vgl. 8.2, S. 19!)

Beispiel: Fernaufrufbares Factory-Objekt zur Erzeugung von Serializable Objekten, die die Klienten sich per Fernaufruf beschaffen können.



Beispiel: Fernaufrufbares Factory-Objekt zur Erzeugung von Serializable Objekten, die die Klienten sich per Fernaufruf beschaffen können.



```
interface Echo { // is not Remote !
void print();      // display contents of Echo object
}
```

```
import java.rmi.*;

interface Factory extends Remote {
Echo create(String s) throws RemoteException;
}                                // delivers a new Echo object
                                  // equipped with string s
```

anna als Anbieterin der Fabrik stellt den Benutzern lediglich diese zwei Schnittstellen zur Verfügung.
Ihre Implementierung von Echo ist Serializable!

In Kenntnis dieser Schnittstellen, des Namens von `annas` Rechner und des Namens der Fabrik im dortigen Register verfasst `paul` ein Klientenprogramm:

```
import java.rmi.registry.*;  
  
public class Test {  
    public static void main(String[] arg) throws Exception{  
        Registry registry = LocateRegistry.getRegistry(  
            "lounge.mi.fu-berlin.de");  
        Factory f = (Factory)registry.lookup("factory");  
        Echo x = f.create(arg[0]);      // delivers copy !  
        x.print();                  // local call !  
    }  
}
```

① Netzdateisystem:

```
anna@lounge:~/echo/classes $ ls
Echo.class
EchoFactory.class
EchoImpl.class
Factory.class
anna@lounge:~/echo/classes $ rmiregistry &
[1] 1373
anna@lounge:~/echo/classes $ java EchoFactory
```

```
paul@xian:~/test/classes $ ls
Echo.class
Factory.class
Test.class
paul@xian:~/test/classes $ java Test Wesel
Exception in thread "main" java.rmi.UnmarshalException:
        error unmarshalling return; nested exception is:
                java.lang.ClassNotFoundException: EchoImpl      ....
```

Dies wird wie folgt geheilt:

```
paul@xian:~/test/classes $ \
> java -cp .:/home/fenn/anna/echo/classes/ Test Wesel
→ esel
paul@xian:~/test/classes $
```

Bemerkung 1: Dies macht es im übrigen entbehrlich, `Echo.class` und `Factory.class` bei `paul` vorzuhalten.

Bemerkung 2: `paul` muss natürlich auf die Dateien in `/home/fenn/anna/echo/classes` Lesezugriff haben - sonst gibt es wiederum eine `ClassNotFoundException: EchoImpl`

Bemerkung 3: Wenn die Klasse von der richtigen abweicht (!), `InvalidClassException`

② Separates Dateisystem:

Z.B. `paul@home: $ java Test Wesel ?`

Dem Objekterzeuger `EchoFactory` muss eine URL beigegeben werden, die die `Codebasis` für die zu versendende Objektkopie bezeichnet (dort ist der Code `EchoImpl.class` vorzuhalten).

```
anna@lounge:~/echo/classes $ java \
> -Djava.rmi.server.codebase=\
> http://page.mi.fu-berlin.de/~anna/classes/ \
> EchoFactory
```

Beim Eintreffen der Objektkopie sorgt der `RMI-Klassenlader` für das Herunterladen des Codes vom Web Server.

```
paul@home: $ java Test Wesel
Exception in thread "main" java.rmi.UnmarshalException:
    error unmarshalling return; nested exception is:
    java.lang.ClassNotFoundException: EchoImpl
(no security manager: RMI class loader disabled)
```

Das Nachladen von *Code über das Web* wird verweigert,
wenn kein **Security Manager** installiert ist:

Gefahr Trojanischer Pferde!

8.5 Sicherheit

Achtung - *mobiler Code* wirft grundsätzliche Sicherheitsfragen auf,
nicht nur bei RMI !

Prinzipiell: Bei jedem *fremden Code* muss man damit rechnen,
dass er sich anders verhält als erwartet -
das gilt für lokal geladenen Code in gleicher Weise
wie für übers Netz geladenen Code.

- **Schadcode** (*malware*) hat unerwünschte Funktionalität, die dem Benutzer Schaden zufügen kann. Typisch sind:
 - **Trojanische Pferde** - haben zwar die gewünschte Funktionalität, aber zusätzlich (und verdeckt) eine schädliche Funktionalität;
 - **Viren** - sind Schadcode, der zusätzlich zu seiner schädlichen Funktionalität auch anderen - bislang unschädlichen - Code infiziert.
- **! Sicherheitsbewusstes Verhalten:** nur solchen Code benutzen,
 - den man selbst geschrieben hat oder
 - dessen Autor man vertraut (?) oder
 - der erfahrungsgemäß richtig funktioniert (?).

... und auf RMI bezogen:

Vorsicht mit Fernaufruf-Argumenten/Ergebnissen,
deren Klassen `Serializable` Anwendungsklassen sind !
(Beispiel: `EchoImpl`, S. 12)

```
anna@lounge: $ java -Djava.rmi.server.codebase=\
> http://page.mi.fu-berlin.de/~anna/classes/ \
> EchoFactory
```

```
paul@home: $ java Test Wesel
```

Hier muss der von Anna bereitgestellten Code `EchoImpl` übers Netz geladen werden - ohne Security Manager scheitert das (S. 13) !

(Wenn Anna nur die Nutzung im gleichen (Netz-)Dateisystem erlauben will:
`-Djava.rmi.server.codebase=file:/home/fenn/anna/classes/`
- vgl 8.2, S. 19)

Schutz vor unerwünschten Aktionen fremden Codes:

Security Manager mitlaufen lassen:

- entweder \$ java -Djava.security.manager
- oder `if(System.getSecurityManager() == null)
 System.setSecurityManager(
 new SecurityManager());`
(typischerweise am Anfang von `main`)

Der *Security Manager* orientiert sich an benutzerdefinierten **Richtlinien** (*policies*), die in **Richtlinien-Dateien** (*policy files*) formuliert sind:

`~/.java.policy` (privat)
`$JAVAHOME/jre/lib/security/java.policy` (global)
... und evtl. andere, explizit benannte Dateien

Berechtigungen (*permissions*) sind der wesentliche Bestandteil einer Richtlinien-Datei: sie legen die erlaubten Aktionen fest.
Was nicht explizit erlaubt ist, ist verboten!

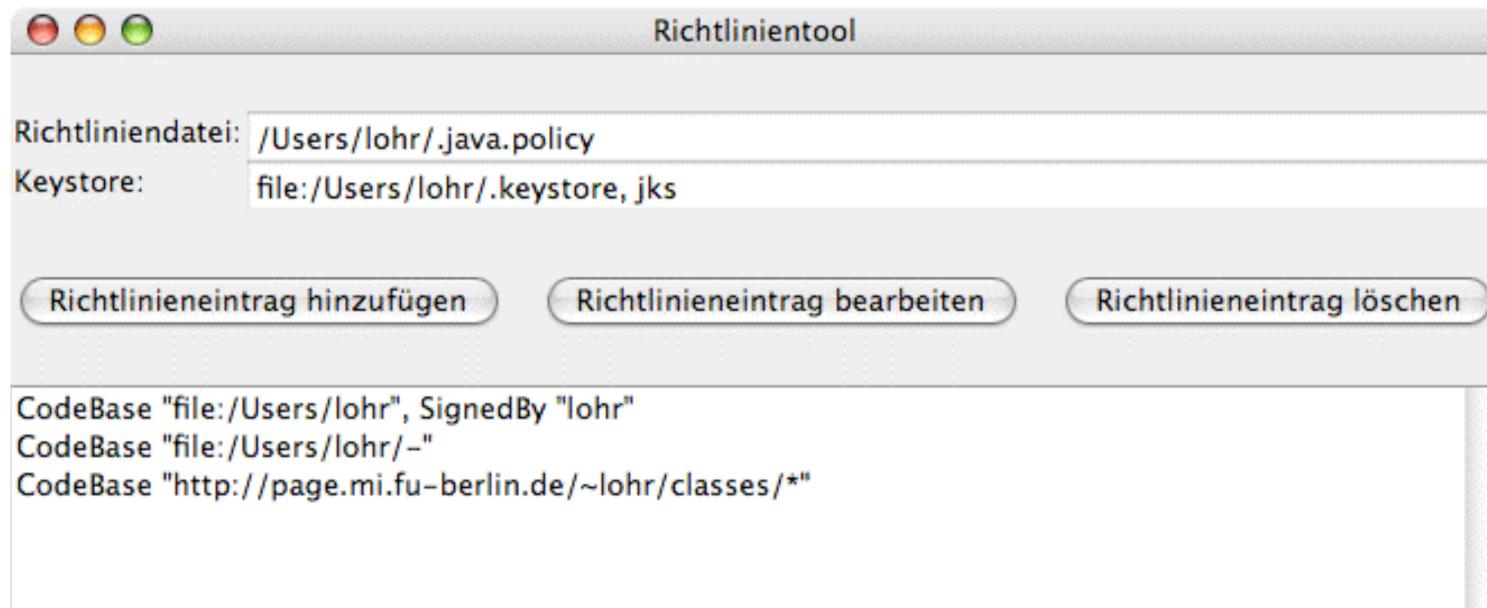
Wenn keine Richtlinien-Datei gefunden werden kann,
ist nichts erlaubt!

Beispiel für eine Richtlinien-Datei `mypolicy` von bob:

```
grant codeBase "file:${user.home}/*" {  
    permission java.security.AllPermission;  
}  
  
grant codeBase "file:/usr/ann/classes/*" {  
    permission java.io.FilePermission "/usr/bob/help/*", "read";  
    permission java.net.SocketPermission "localhost:1024-",  
        "connect, accept";  
}  
  
grant {           // allows any code to read in pub subtree  
    permission java.io.FilePermission "/usr/bob/pub/-", "read";  
};
```

Benutzung z.B. so: bob: java -Djava.security.manager \
-Djava.security.policy=mypolicy ...

Programm `policytool` erlaubt komfortable Bearbeitung von Richtlinien-Dateien, z.B. beim Mac:



Beispiel (ohne RMI): *System Property* erfragen

```
class GetProps {  
public static void main(String[] arg) throws Exception {  
    s = System.getProperty("java.home", "not specified");  
    System.out.println("Your Java home directory is: " + s);  
}  
}  
  
$ java GetProps  
Your Java home directory is: /System/Library/Frameworks/JavaVM.fr
```

```
$ java -Djava.security.manager GetProps  
Exception in thread "main" java.security.AccessControlException:  
access denied (java.util.PropertyPermission java.home read) ...  
$
```

Dies wird geheilt durch Bereitstellung einer einfachen Richtlinien-Datei `.java.policy` mit

```
grant codeBase "file:${user.home}/-" {  
    permission java.security.AllPermission;  
};
```

Damit kann endlich auch Paul die `EchoImpl` testen (S. 13, 16):

```
paul@home: $ java -Djava.security.manager \  
>           Test Wesel  
esel  
paul@home: $
```

... und das Kleingedruckte zu den Richtlinien-Dateien:

„The exact meaning of a codeBase value depends on the characters at the end. A codeBase with a trailing "/" matches all class files (not JAR files) in the specified directory. A codeBase with a trailing "/" matches all files (both class and JAR files) contained in that directory. A codeBase with a trailing "-/" matches all files (both class and JAR files) in the directory and recursively all files in subdirectories contained in that directory.“*

Aus

<http://docs.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html>

Abschnitt „The SignedBy, Principal, and CodeBase Fields“

8.6 Hochladen von Code

(nach <http://docs.oracle.com/javase/tutorial/rmi/>)

Wir haben rechenintensiven Code `compute` und wollen ihn auf einer fremden, leistungsfähigen Maschine zur Ausführung bringen.

Spezifikation der Berechnung:

```
public interface Task<T> {           // not Remote !
    T compute();                      // computes value of type T
}
```

Spezifikation eines entfernten Objekts `Computer`:

```
public interface Computer extends Remote {
    <T> T execute(Task<T> t) throws RemoteException;
}
```

Die tatsächliche Berechnung eines Objekts vom Typ `Double` -
hier im Beispiel die Berechnung von π mit n Stellen:

```
class ComputePI implements Task<Double>, Serializable {  
    ComputePI(int n) { this.n = n; }  
    final int n; // number of digits to be computed  
public Double compute() { ..... }  
}
```

(Die Implementierung

```
class ComputeEngine implements Computer { ..... }
```

ist dem Klienten unbekannt!)

... und das Rahmenprogramm des Klienten:

```
class RemotePiComputation {
public static void main(String arg[]) {
    .....
    Computer c = (Computer)registry.lookup("computer");
    Double pi  = c.execute(new ComputePI(100));
    .....
}
```

Die vollständigen .java-Dateien findet man unter
<http://www.inf.fu-berlin.de/lehre/WS12/alp5/folien/ComputeEngine/>

... und damit:

Fritz stellt auf `xian` eine `ComputeEngine` (mit eingebautem Register) bereit und informiert die Öffentlichkeit:

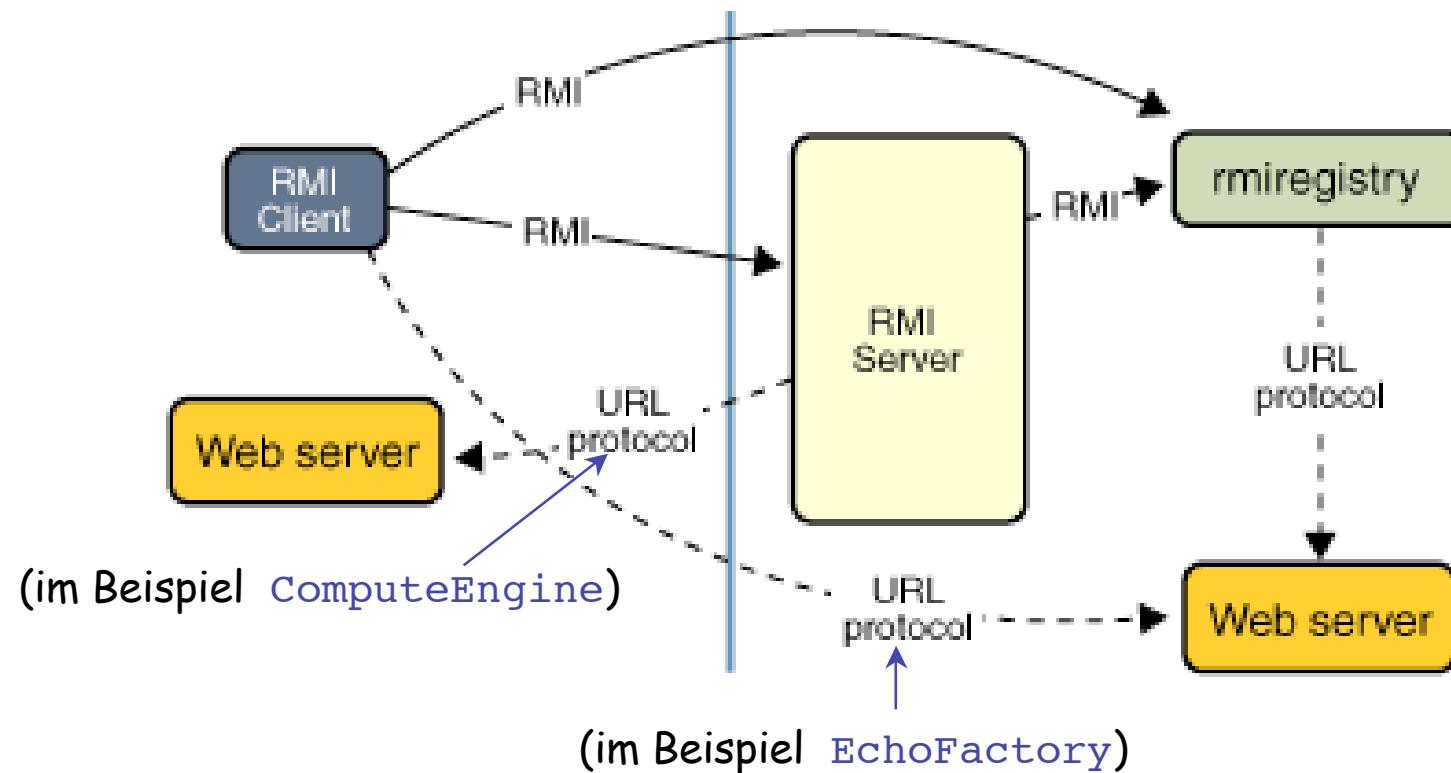
```
fritz@xian: $ java -Djava.security.manager \
>                         ComputeEngine &
ComputeEngine ready on 56789
```

Franz stellt `ComputePI` auf Web Server bereit und startet (mit VPN!) einen Klienten `RemotePiComputation` auf `home`:

```
franz@home $ java -Djava.rmi.server.codebase=\
> http://130.133.48.229/~franz/classes/ \
> RemotePiComputation xian.imp.fu-berlin.de 56789
PI is 3.1415926.....
```

Freya stellt „gleichzeitig“ `ComputeE` auf Web Server !

Übersicht über die eingesetzten Protokolle:



8.7 Herunterladen von Code

Graphische Benutzerschnittstelle (GUI)

- wird programmatisch konstruiert als Objekt `gui` einer Klasse
 - `class GUI extends java.awt.Frame ...`
- wird sichtbar gemacht durch `gui.setVisible(true)`
- ist `Serializable` und kann daher *per Fernaufruf* von einem Server-Objekt heruntergeladen werden !

```
App application = (App)Naming.lookup(rmiurl);
GUI gui = application.getGUI();
gui.setVisible(true);
```

Zusammenfassung

- Mobiler Code bei RMI ist überhaupt nur möglich, weil es sich um interpretierten Bytecode handelt, der auf allen Plattformen lauffähig ist.
- Ohne die Unterstützung von Code-Mobilität wäre die Benutzung von `Serializable` Argumenten/Ergebnissen nur sehr eingeschränkt möglich.
- Mobiler Code bringt ein erhöhtes Sicherheitsrisiko mit sich; RMI erzwingt daher den Einsatz des *Security Manager* und zwingt den Benutzer, sich über die Rechtevergabe an fremden Code Gedanken zu machen.

Quellen

Oracle Corp.: *Java Security.*
<http://docs.oracle.com/javase/6/docs/technotes/guides/security>

Oracle Corp.: *Java Security Tutorial.*
<http://docs.oracle.com/javase/tutorial/security>

... und zu mobilem Code allgemein:

A. Fuggetta, G.P. Picco, G. Vigna: *Understanding Code Mobility.* IEEE Trans. on Software Engineering 24.5, May 1998.
www.computer.org/csdl/trans/ts/1998/05/e0342-abs.html