Remote Method Invocation

Bisher

Objekt-Orientierung in Java:

Objekt A ruft Methode m bei Objekt B auf

A und B leben beide in derselben JVM

nicht ausreichend für verteilte Anwendungen

entweder

- weniger Objekt-Orientierung, oder
- Erweiterung auf mehrere JVMs

Jetzt

Objekt A ruft Methode m bei Objekt B auf

A und B evtl. auf verschiedenen JVMs

Kommunikation (auf nicht-Java-Ebene) über TCP/IP

auf Java-Ebene: RMI

Das Interface `Remote`

Objekte, auf die *remote* zugegriffen werden kann, müssen `Remote` sein

typische Nutzung:

```
public interface MeinService extends java.rmi.Remote {
  methode(...) throws java.rmi.RemoteException;
}

public class MeinServiceImplementation
implements MeinService {
  methode(...) throws java.rmi.RemoteException { ... }
}
```

Deklaration dieser Exception ist Pflicht!

remote Teil der Anwendung muss (zuerst) nur `MeinService` kennen

Woher weiß. . .

. . . B, was A ihm übergibt?

. . . A, was B berechnet hat?

Antwort: *stubs*

- lokale Sicht eines remote Objektes
- Funktionsweise:
  - nimmt Argumente
  - verschickt sie zur JVM, in der B lebt
  - wartet auf Ergebnis
  - stellt dies zur Verfügung

Stubs erzeugen

mit dem RMI-Compiler

Aufruf `rmic MeinServiceImplementation`, nimmt `.class`-File

Achtung, remote Teil (Sourcecode) benutzt nur `MeinService` und stubs führen selbst keinen Code aus

Warum nicht `rmic MeinService`? $\leadsto$ stub wichtig zur Laufzeit, nicht Compilezeit

Die Klasse `UnicastRemoteObject`

benötigt zum Erhalten eines Stubs und Exportieren eines remote Objektes

2 Möglichkeiten

- `public class MeinServiceImplementation extends UnicastRemoteObject implements MeinService ...`
- `UnicastRemoteObject.exportObject(...)`

Was kann überhaupt verschickt werden?

Welche Argumente können Methoden von `Remote` haben?

- primitive Datentypen
- Objekte vom Typ `java.io.Serializable`

Die `rmiregistry`

Nährboden für RMI

lauscht auf TCP/IP-Port, default 1099

nur 1mal pro Host und Port

name server, Tabelle `(String` s`, Object` o`)`

Idee: Objekt o bietet Service unter Namen s an

Stub wird zu eingetragenen Objekten benötigt, nicht zu deren Interfaces

`rmiregistry` starten

2 Möglichkeiten

- per Hand: `.../jdk1.5.0/bin/rmiregistry &`
  - \+ ist für mehrere Klassen, User, etc. verfügbar
  - \- leicht zu vergessen
- per Methode: `java.rmi.registry.LocateRegistry.createRegistry()`
  - \+ Erzeugung flexibler
  - \- Mehrfacherzeugung mit `getRegistry(...)` vermeiden

Service eintragen

```
public class MeinServiceImplementation
implements MeinService {

  public MeinServiceImplementation() { ... }

  public static void main(String[] args) {
     ...
     service = new MeinServiceImplementation();
     java.rmi.Naming.rebind("Servicename", service);
     ...
  }
}
```

Zugriff auf nicht-lokale Registry: `//host:port/Servicename`

Service erhalten

```
public class BenutzerImplementation
implements Benutzer {
  ...
  service = (MeinService) java.rmi.Naming.lookup("Servicenam
  ...
  service.methode(...);
  ...
}
```

Service auf remote Host abfragen: “`//host:port/Servicename`”

Rest wie üblich

Ein Wort zu Ports

`rmiregistry` braucht offenen Port

Netzwerksicherheit braucht geschlossene Ports

im CIP-Pool z.B.:

- Ports nach aussen hin geschlossen
- innerhalb des Netzwerks offen

Tip: verteilte Anwendung erst auf einem Host, dann auf verschiedenen innerhalb eines Netzes laufen lassen

am Ende: `killall rmiregistry` !!!

Ein Wort zu Prozessen

`main`-Methode in Service braucht keine Endlosschleife

in `rmiregistry` eingetragenes Objekt

- ruht, wartet auf Methodenaufrufe von aussen,
- wird nicht gegarbagecollectet

keine unnötige CPU-Zeit verbraten

Remote-Anwendungen starten mehrere `java`-Prozesse

am Ende: `killall java` !!!

Literatur

- Sun’s Infos zu RMI
  `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/`
- Sun’s RMI Tutorial
  `http://java.sun.com/docs/books/tutorial/rmi/`
- Another RMI Tutorial
  `http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html`
- ...

Beispiele

Hausaufgabe

Hausaufgabe

Erstellung eines Chat-Services: Server + Client

Anforderungen:

- Server implementiert Interface `Server`
- Client implementiert Interface `Client`
- Teilnehmer = Client + Name + aktiv? (abstrakte Klasse vorgegeben)
- Teilnehmer können sich beim Server an-/abmelden
- Server benachrichtigt alle Teilnehmer über An-/Abmeldungen
- Teilnehmer können einzeilige Nachrichten (Strings) verschicken, entweder an alle oder nur an einen
- Nachrichten dürfen verloren gehen
  - bei Serverausfall
  - bei Abmelden eines Teilnehmers

Hausaufgabe

- Nachrichten gehen nicht verloren bei temporärem Ausfall eines Clients
- Server dirigiert, d.h. kein Zugriff zwischen Clients
- Client hat GUI mit
  - Textfeld für Eingabe, Textfeld für Ausgabe
  - Buttons zum Verschicken an alle, einzelne Teilnehmer
- Exception-Handling (s. API)
- Java-Doc-Dokumentation

Jar-Archiv mit Interfaces und abstrakter Klasse Teilnehmer auf WWW-Seite