Remote Method Invocation

Netzwerkprogrammierung in Java

Package `java.net` implementiert Protokolle zur Kommunikation über das Internet Protocol (IP).

User Datagram Protocol (UDP)

- verbindungslos, es werden einfach Datenpakete verschickt und empfangen
- ohne Garantien, dass Pakete ankommen und in welcher Reihenfolge

Transfer Control Protocol (TCP)

- verbindungsorientiert
- mit Fehlerkorrektur

Applikationsprotokolle (HTTP, FTP, ...)

Möglichkeiten der Netzwerkkommunikation

Kommunikation über TCP

- Client und Server bauen eine Verbindung auf, über die Byteströme in beide Richtungen gesendet werden können.
- Daten müssen in Bytefolgen umgewandelt werden
- Applikationsprotokolle auf dieser Basis implementiert

Beispiel: SMTP

```
...
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Hallo!
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

Möglichkeiten der Netzwerkkommunikation

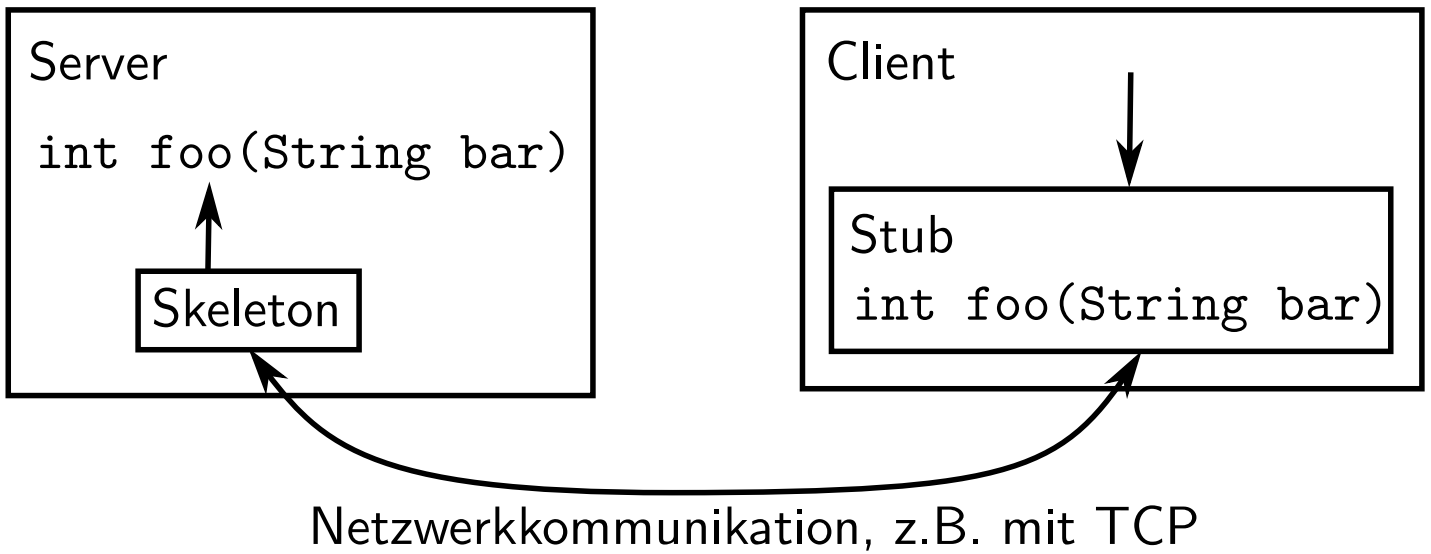
Kommunikation über TCP

- Client und Server bauen eine Verbindung auf, über die Byteströme in beide Richtungen gesendet werden können.
- Daten müssen in Bytefolgen umgewandelt werden
- Applikationsprotokolle auf dieser Basis implementiert
- aufwendig
  - Protokollentwurf
  - Kodierung und Dekodierung
  - Synchronisierung
- flexibel
  - heterogene Umgebungen
  - Fehlerbehandlung
  - effiziente Kodierungen möglich

Verteilte Java-Anwendungen

Typischer Fall:

- Server stellt Funktion bereit: `int foo(String bar)`
- Client möchte diese Funktion aufrufen
- Implementierung mittels Stubs/Skeletons.
  1. Stub kodiert Argument und schickt es an Skeleton
  2. Skeleton dekodiert es, ruft `foo` auf, kodiert das Ergebnis und schickt es zum Stub
  3. Stub dekodiert das Ergebnis und gibt es zurück



Remote Method Invocation (RMI)

Java-spezifische Methode zum entfernten Methodenaufruf

- automatische Generierung von Stub- und Skeletonklassen
- Client kann Methoden auf dem Server wie normale Java-Funktionen aufrufen
- transparentes Verschicken von Objekten, inklusive ihrer Funktionen
- dynamisches Nachladen von Objektklassen

RMI — Überblick

Entfernt aufrufbare Objekte haben folgenden Eigenschaften:

- Sie implementieren das (leere) Interface `java.rmi.Remote`
- Jede ihrer Methoden deklariert `throws java.rmi.RemoteException`

Server erzeugt ein solches Objekt und registriert es unter einem bestimmten Namen in der *RMI Registry* (Namensdienst).

Client holt sich eine Referenz auf das Objekt von der *RMI Registry*.

Alle Methodenaufrufe des Clients werden dann über das Netzwerk an das Objekt im Server weitergegeben.

Beispiel: Zeitserver

Entfernt aufrufbares Interface:

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

public interface Time extends Remote {
   public long getTime() throws RemoteException;
}
```

- Jedes entfernt aufrufbare Interface leitet von `Remote` ab.
- Jede Methode muss die Exception `RemoteException` werfen können.

Implementierung

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

public class TimeImpl implements Time {
   public long getTime() throws RemoteException {
      return System.currentTimeMillis();
   }
}
```

Server

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

public class Server {
public static void main(String args[]) {
    try {
      TimeImpl ts = new TimeImpl();
      Time stub = (Time)UnicastRemoteObject.exportObject(ts, 0);

      // Den Stub registrieren
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("Time", stub);

      System.err.println("TimeServer bereit");
    } catch (Exception e) {...}
  }
}
```

Server

```
TimeImpl ts = new TimeImpl();
Time stub =
 (Time)UnicastRemoteObject.exportObject(ts,0);
```

- `UnicastRemoteObject.exportObject(Remote obj, int port)` erzeugt ein Stub-Objekt für `obj` und exportiert `obj` über einen TCP-Port (`port=0` für einen anonymen Port).
- Das Stub-Objekt werden sich später die Clients holen und es dann benutzen, um mit dem Server zu kommunizieren.
- `Unicast` bedeutet, dass das Stub-Objekt mit einem einzigen Server kommuniziert (hier mit Objekt `ts`).

Server

```
Registry registry = LocateRegistry.getRegistry();
registry.rebind("Time", stub);
```

- Registriere das Stub-Objekt unter dem Namen „Time“ in der RMI Registry.
- Clients können es sich jetzt unter diesem Namen erfragen.

Server starten

1. `Time.java`, `TimeImpl.java` und `Server.java` normal kompilieren, z.B. im Verzeichnis `/home/schoepp/rmi/`
2. Registry starten:

   `rmiregistry &` (Unix)

   `start rmiregistry` (Windows)
3. Server starten:

   ```
   java -Djava.rmi.server.codebase=url Server
   ```

   Dabei ist `url` eine URL, die auf die Klassen des Servers zeigt, z.B. `file:/home/schoepp/rmi/` (letztes '/' ist wichtig!)

   Die URL muss man angeben, damit die RMI-Implementierung die Stubs aus den Klassen des Servers erzeugen kann.

Client Implementierung

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

public class Client {
  public static void main(String[] args) {
    String host = (args.length < 1) ? null : args[0];
    try {
      Registry reg = LocateRegistry.getRegistry(host);
      Time server = (Time)reg.lookup("Time");
      System.out.println("Zeit auf dem Server: "
                         + server.getTime());
    } catch (Exception e) {...}
  }
}
```

Client starten

1. `Time.java` und `Client.java` normal kompilieren
2. Client starten:

   ```
   java -Djava.rmi.server.codebase=url Client
   ```

   Dabei ist nun `url` eine URL, die auf die Klassen des Clients zeigt.

   Die URL wäre nötig, wenn der Server sich noch Klassen vom Client holen muss, die dieser an den Server schicken will.

   In diesem Beispiel ist das aber nicht nötig und man könnte die `codebase`-URL auch weglassen.

Weitere Themen

- Übergabe von komplexen Argumenten
- Callbacks
- Nebenläufigkeit
- dynamisches Nachladen von Klassen

Argumentübergabe

Welche Klassen `A` und `B` können in einem `Remote`-Interface benutzt werden?

```
public interface I extends Remote {
  public A funktion(B x) throws RemoteException;
}
```

1. Klassen, die `java.rmi.Remote` implementieren.
   Objekte, die zu solchen Klassen gehören, werden per *Referenz* übergeben (wie sonst auch).
2. Klassen, die `java.io.Serializable` implementieren.
   Objekte, die zu solchen Klassen gehören, werden *kopiert*!

   Beispiel: `List<List<Integer>>` (`int[][]` ist effizienter)

Wenn `A` oder `B` keines der beiden Interfaces implementiert, dann wird der Aufruf (zur Laufzeit!) fehlschlagen.

Beispiel: Callbacks

- Server bietet Funktion `int getNumber()` an.
- Client möchte wissen, wenn sich die Zahl ändert.
- Möglichkeiten
  1. Polling:
     Client fragt alle 100ms beim Server nach.
  2. Callbacks:
     Server sagt dem Client, wenn sich die Anzahl geändert hat.

Beispiel: Callbacks

```
public interface Server extends java.rmi.Remote {
  public int getNumber();
  public void registerChangeNotifier(ChangeNotifier n);
}

public interface ChangeNotifier extends java.rmi.Remote {
  public void changed();
}
```

- Client erzeugt Objekt, das `ChangeNotifier` implementiert und gibt eine Referenz darauf mit `registerChangeNotifier` an den Server.
- Server kann jetzt `changed()` aufrufen, um dem Client zu sagen, dass etwas passiert ist.

Übergabe der Referenzen

1. Server und Client erzeugen Objekte `ServerImpl` und `NotifierImpl`, die `Server` und `ChangeNotifier` implementieren.
2. Server gibt eine Referenz auf `ServerImpl` an die RMI Registry.
3. Client holt sich Referenz auf `ServerImpl` von der Registry
4. Client gibt eine Referenz auf `NotifierImpl` an den Server, indem er `registerChangeNotifier` aufruft.
5. Server kann nun `NotifierImpl.changed()` aufrufen.

Würde `NotifierImpl` statt `Remote` das Interface `Serializable` implementieren, dann würde `NotifierImpl.changed()` auf dem Server ausgeführt.

Nebenläufigkeit

Spezifikation:

> *A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads.*

- Suns RMI-Implementierung benutzt Thread-Pools.
- Jede Methode in einem `Remote`-Objekt kann gleichzeitig von mehreren Clients aufgerufen werden.

$\Rightarrow$ Man muss selbst für Synchronisierung sorgen.

Dynamisches Laden von Klassen

```
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t)
      throws RemoteException;
}
public interface Task<T> extends java.io.Serializable {
    T execute();
}
```

Implementierung im Server:

```
T executeTask(Task<T> t) {
  return t.execute();
}
```

Client kann beliebige Funktionen an den Server schicken, die dann dort ausgerechnet werden.

Quelle: `http://java.sun.com/docs/books/tutorial/rmi`

Dynamisches Laden von Klassen

Zweites Beispiel:

- Ein Serverobjekt implementiert zwei voneinander unabhängige `Remote`-Interfaces `A` und `B`.
- Der Client kennt nur das Interface `A`.

Zur Übertragung des Stubs für das Serverobjekt, muss RMI beide Interfaces `A` und `B` auf der Client-Seite haben.

Dynamisches Laden von Klassen

RMI lädt fehlende Klassen automatisch nach, *wenn das vom Security Manager erlaubt ist*.

- Übertragung sowohl vom Server zum Client als auch vom Client zum Server.
- RMI findet die Klassen, indem es unter den URLs in `java.rmi.server.codebase` nachschaut.

Standardmäßig ist kein Security Manager installiert und es werden deshalb keine Klassen nachgeladen.

Security Manager

RMI Security Manager installieren

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

Sicherheitsregeln beim Start von Server/Client übergeben:

```
java -Djava.rmi.server.codebase=url
     -Djava.security.policy=security.policy
     Client
```

Datei `security.policy` enthält die Sicherheitsregeln.

Sicherheitsregeln

Beispiele für die Datei `security.policy`:

- Alles ist erlaubt (inkl. Festplatte löschen):

```
grant {
  permission java.security.AllPermission;
};
```

- Kommunikation über Sockets (TCP) und lesender Dateizugriff

```
grant {
  permission java.net.SocketPermission
    "*:1024-65535", "connect,accept";
  permission java.io.FilePermission
    "<<ALL FILES>>", "read";
}
```

Aufgabe

Jede Gruppe schreibt bis nächste Woche mit RMI ein Client/Server-Chatprogramm.

Interfacevorschlag:

```
public interface ChatServer extends java.rmi.Remote {
  ServerInterface register(String nick, ClientInterface client)
    throws java.rmi.RemoteException;
}

public interface ClientInterface extends java.rmi.Remote {
  void hear(String nick, String message)
    throws java.rmi.RemoteException;
}

public interface ServerInterface extends java.rmi.Remote {
  void say(String message) throws java.rmi.RemoteException;
  void signoff() throws java.rmi.RemoteException;
}
```