Netzwerkintegration und verteilte Programmierung

Überblick

1. Was versteht man unter verteilten Programmen? Welche Vorteile bieten sie? Anwendungen?
2. Grundlegende Begriffe (Client, Server), Ziele
3. Verteilte Programmierung in Java (RMI)
4. Begriffe (Remote-Objekt, RMI, Stumpf, Skelett, Registratur)
5. Aufbau und Funktionsweise von RMI-Client/Server-Anwendungen
6. Game-Server-Beispiel
7. Verwandte Konzepte (RPC, CORBA (IDL))
8. Verteilte Programmierung in Java II (Sockets, UDP)

Client-Server-Systeme

Eine mögliche Form verteilter Systeme sind *Client-Server-Systeme*.

Server: Eine Soft- oder Hardware, die anderen Soft- oder Hardwarekomponenten Dienste über eine Kommunikationsschnittstelle anbietet.

Client: Eine Soft- oder Hardware, die von Servern angebotene Dienste nutzt.

Andere verteilte Systeme arbeiten teilweise ohne ausgezeichnete Rechner.

Anwendungsbeispiele für Client-Server-Systeme

- **Fahrplanauskunft:** Zentraler Server mit Daten der Bus- und Bahnverbindungen, Kundenschalter mit Client-Rechnern.

- **Flugbuchungssystem:** Zentrale Datenbank mit Flügen, Buchungen und noch freien Plätzen, Client-Rechner in den Reisebüros weltweit.

- **Kontenverwaltung einer Bank:** Zentrale Kontenverwaltung in der Hauptstelle der Bank und Clients in den Zweigstellen und in den Geldautomaten.

- **Netzweite Resourcennutzung in (lokalen) Rechnernetzen:** Einzelne Rechner übernehmen bestimmte Aufgaben (File-Server, Print-Server, Compute-Server, etc.), für die sie besonders ausgelegt sind, und bieten diese speziellen Dienstleistungen allen anderen Rechnern im Netz an (Clients).

Ziele beim Einsatz verteilter Systeme

- **Lastverteilung** bzw. **Netzlastreduzierung** durch verteiltes Rechnen und Verlagerung des Rechenaufwands.
- **Kostenersparnis** durch netzweite Resourcennutzung (nicht jeder Rechner braucht einen eigenen Drucker oder hohe Rechenleistung).
- **Datenkonsistenz** bei zentraler Datenhaltung im Gegensatz zum Arbeiten mit getrennt veränderten Kopien der Daten auf dezentralen Rechnern.
- **Erhöhung der Ausfallsicherheit** durch Replikation von Daten und Dienstleistungen (*erschwert evtl. die Konsistenzhaltung von Daten*).

Unterstützung verteilter Programmierung durch Java

Häufig bereiten die Heterogenität der Netzwerkumgebungen, Sicherheitsprobleme und die Komplexität der Protokollspezifikation Schwierigkeiten bei der Realisierung verteilter Anwendungen.

Das *Remote Method Invocation* (RMI) Konzept von JAVA löst zumindest zwei der drei Probleme:

- **Heterogenität**: da JAVA nahezu maschinenunabhängig ausführbar und für zahlreiche Plattformen verfügbar ist.
- **Komplexität**: da sich mit RMI mit minimalem Aufwand eine Client-Server-Anwendung entwickeln läßt.
- **Sicherheit**: dieses Problem löst JAVA nicht (Privatsphäre, Datenintegrität, Zugriffsschutz).

Remote Method Invocation (RMI)

Die RMI-Schnittstelle ermöglicht die Kommunikation von Objekten, die sich auf verschiedenen Rechnern befinden.

Remote-Objekt: ein Objekt, das zur Laufzeit durch die JAVA Virtual Machine A gehalten wird und von Objekten, die sich in einer Maschine B befinden, angesprochen werden kann. Das Remote-Objekt kann also auch als *Server-Objekt* aufgefaßt werden.

Objekte dieser Art werden durch eine oder mehrere *Remote-Schnittstellen* beschrieben, die die Methoden des Remote-Objekts deklarieren.

Remote Method Invocation: Aufruf von Methoden einer Remote-Schnittstelle für ein Remote-Objekt. Dabei ist die Syntax die gleiche wie bei Methodenaufrufen für lokale Objekte.

Unter der Oberfläche von RMI verbergen sich Konzepte wie Serialisierung, Sockets und Ströme, auf die an dieser Stelle nicht näher eingegangen werden soll.

Stümpfe und Skelette

Wie kann ein Client-Objekt ein Remote-Objekt ansprechen, obwohl das Remote-Objekt lokal nicht existiert und dessen Struktur damit auch nicht bekannt ist?

Stumpf (*engl. Stub*): kleine Schnittstellen-Beschreibung einer Klasse, die die `UnicastRemoteObject`-Klasse erweitert. Der Stumpf wird zur Beschreibung eines Remote-Objektes an den Client-Rechner übertragen. Stümpfe werden über den `rmic`-Compiler durch den Programmierer erzeugt und sind durch Programmierer nicht veränderbar.

Skelett (*engl. Skeleton*): dem Stumpf ähnlich, verbleibt jedoch auf dem Server.

Für die Klasse eines Remote-Obejektes werden Stümpfe erzeugt. Da die Stümpfe nur die Beschreibung einer Klasse und nicht deren gesamte Funktionalität enthalten, können diese kostengünstig über das Netz übertragen werden.

Registratur (*engl. registry*): registriert und verwaltet Instanzen von Remote-Objekten, die in einer Maschine gehalten werden. Die Registratur stellt Informationen zu diesen Objekten bereit und kann über eine URL angesprochen werden.

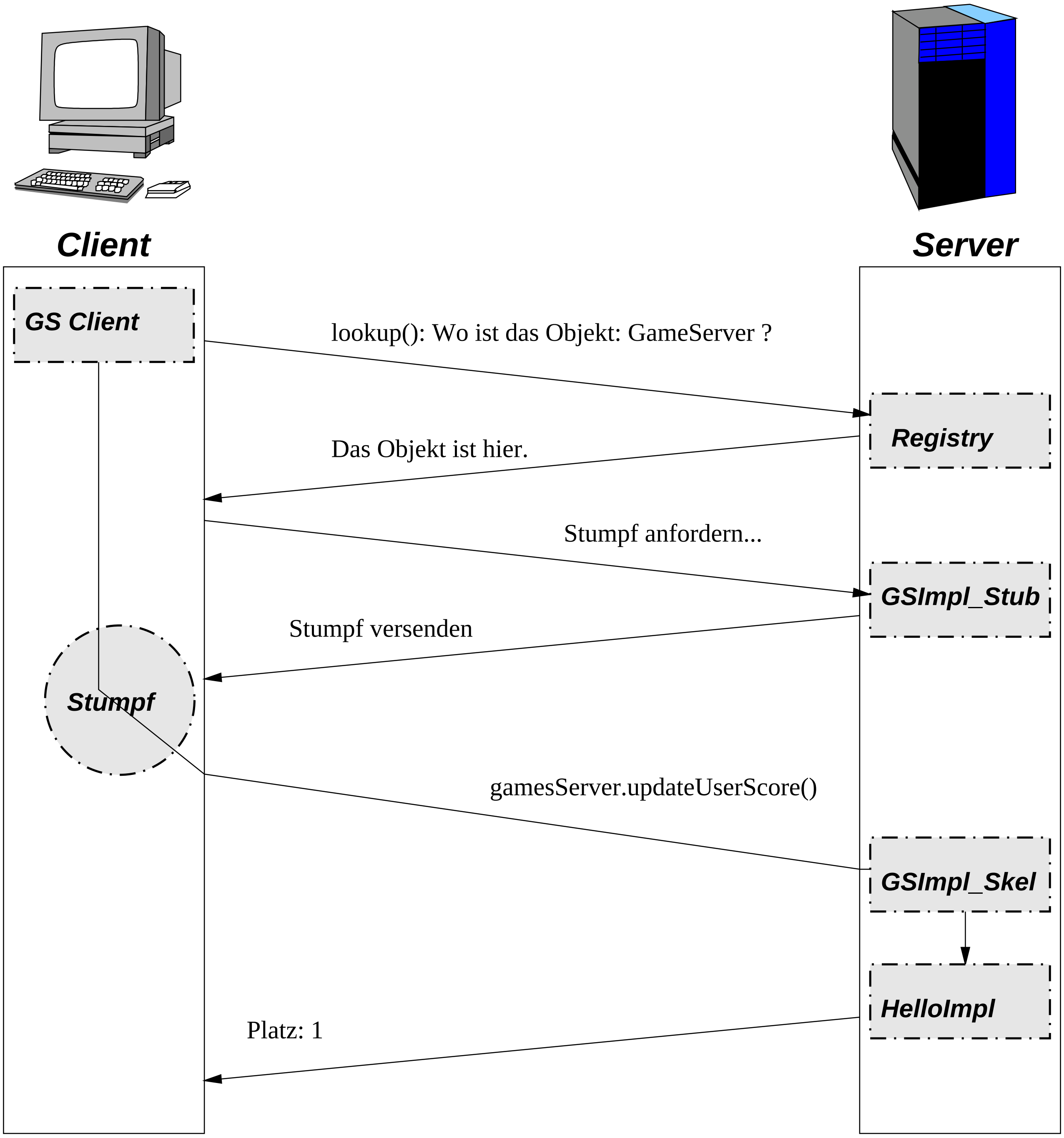
Wie erhält eine Anwendung den Stumpf?

1. Die Anwendung öffnet einen *Kommunikationskanal zu einer Registratur*.
2. Die Anwendung fordert anhand eines Namens bei der Registratur eine Objekt-Instanz an.
3. Der *Stumpf* des angeforderten Objektes wird übertragen und kann in der Client-Anwendung angesprochen werden. Die Verbindung mit diesem virtuellen Objekt verdeckt die Kommunikationsvorgänge im Hintergrund.

Die Client-Anwendung benötigt lediglich den *Server-Namen* und die *Port-Nummer* der Registratur sowie den *Namen des angeforderten Objektes*.

Portnummer: eine Art Telefonnummer, auf der ein anderer Rechner einen bestimmten Dienst erreichen kann (z.B. Port 80 meist Port eines HTTP-Servers (WWW-Server), Port 1099 eine Registratur).

Kommunikation zwischen Client und Server für den Aufruf einer Remote-Objekt-Methode durch ein Client-Objekt:



Wie wird ein Objekt zu einem Remote-Objekt?

- Eine Schnittstelle `Server`, die die `Remote`-Schnittstelle erweitert, beschreibt ein Remote-Objekt, das alle angebotenen Methoden implementieren muß.

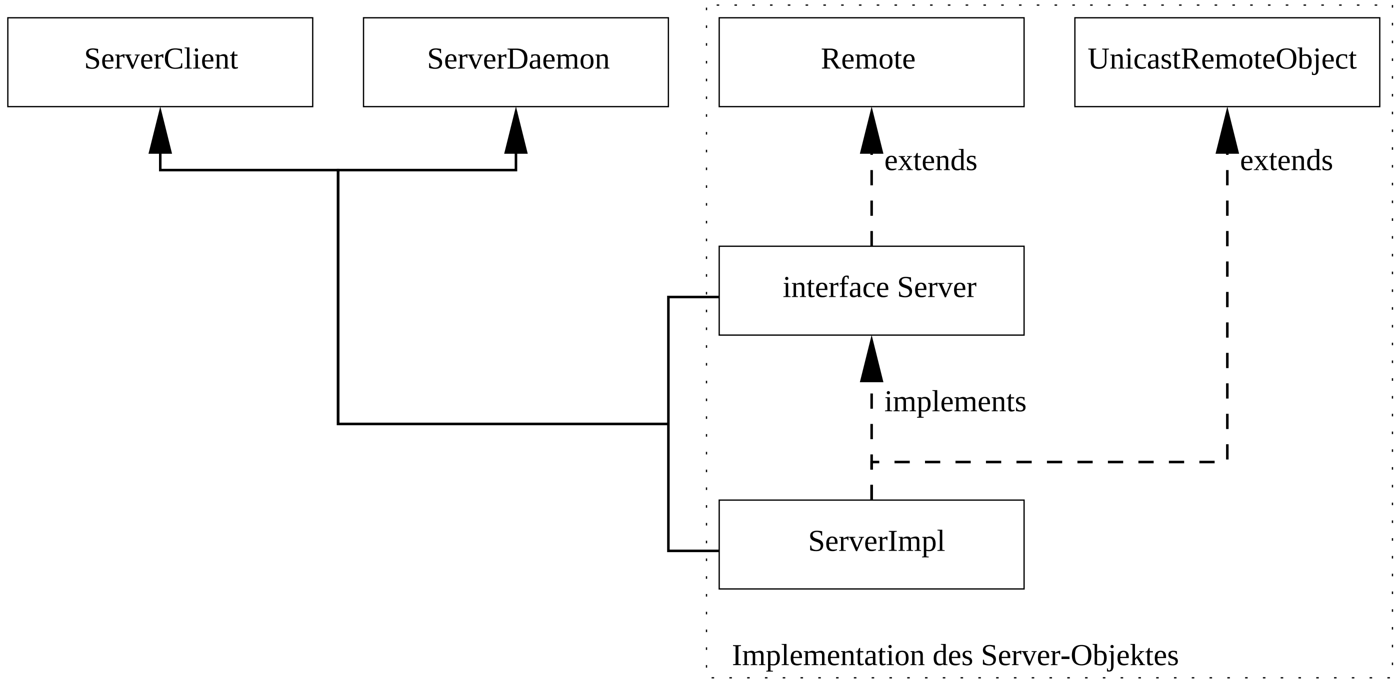
- Ein Klasse **ServerImpl**, die die Schnittstelle `Server` implementiert und die `UnicastRemoteObject`-Klasse erweitert, bildet die Remote-Objekt-Klasse.

- Die Stümpfe und Skelette werden mit dem `rmic`-Compiler erzeugt (hier: `rmic ServerImpl`).

- Eine Instanz der Remote-Objekt-Klasse wird in der Applikation **ServerDaemon** erzeugt und in einer Registratur registriert.

- Das Objekt kann nun durch eine Applikation **ServerClient** angesprochen werden.

Zentrale Schnittstellen und Klassen für ein RMI-Client-Server-System



Schnittstelle `GameServer`

Die Schnittstelle `GameServer` erweitert die Schnittstelle `Remote` und beschreibt die vom Server zur Verfügung gestellten Methoden (Verwaltung einer Spielstandstabelle mit Rangfolge gemäß der absoluten Spielstände).

```
package RMIBeispiel;

import java.rmi.*;
import RMIBeispiel.DatabaseException;

public interface GameServer extends Remote {

    // Spielerstand in die Datenbank des Servers eintragen
    public int updateScore (String userID, int score)
        throws RemoteException,
        DatabaseException;

    // Liefert Spielernamen, Score des n-ten Tabellenplatzes
    public String getUserScore (int placement)
        throws DatabaseException, RemoteException;

}                                           // GameServer
```

Klasse `GameServerImpl`

Die Klasse `GameServerImpl` implementiert die von der Scnittstelle `GameServer` geforderten Methoden sowie einen Konstruktor und weitere Methoden. Die `GameServerImpl`-Klasse muß die Klasse `UnicastRemoteObject` erweitern.

```
package RMIBeispiel;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

import RMIBeispiel.DatabaseException;
import RMIBeispiel.Entry;

import java.util.*;

public class GameServerImpl
extends UnicastRemoteObject implements GameServer {

    ...
```

...

```
    // Konstruktor: erzeugt eine Spielstandstabelle
    public GameServerImpl (String filename)
        throws RemoteException, DatabaseException {
        ...
    }

    public int updateScore (String userID, int score)
        throws RemoteException, DatabaseException {
        ...
    }

    public String getUserScore ( int placement )
        throws DatabaseException, RemoteException {
        ...
    }
}
```

Klasse `GameServerDaemon`

Die Klasse `GameServerDaemon` erzeugt und verwaltet eine RMI-Registratur, erzeugt zumindesten eine Instanz der Klasse `GameServerImpl`, die es in der Registratur registriert.

```
package RMIBeispiel;

import RMIBeispiel.DatabaseException;
import RMIBeispiel.GameServerImpl;
import RMIBeispiel.GameServer;

import java.io.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.net.MalformedURLException;

public class GameServerDaemon {
    private GameServer gameServer;

    ...
```

...

```
public GameServerDaemon () {
    System.runFinalizersOnExit (true);        // deprecated

    if (System.getSecurityManager ()==null) {
        System.setSecurityManager (
                        new RMISecurityManager ());
    }

    try {
        LocateRegistry.createRegistry (2060);
        gameServer = new
                    GameServerImpl (“spielstand.dat“);
        Naming.bind ( “//kiew.cs.uni−dortmund.de“
                + “:2060/GameServer“, gameServer);
    } catch (Exception e) {
        e.printStackTrace () ;   // alle Ausnahmen fangen
    }
}

public static void main (String[] args) {
    final GameServerDaemon daemon =
                        new GameServerDaemon ();
}
}
```

Klasse `GameServerClient`

Die Klasse `GameServerClient` realisiert einen Beispiel-Client, der eine Verbindung zur Registratur herstellt und eine virtuelle Kopie des registrierten `GameServer`-Objektes erzeugt. Nach der Erzeugung kann auf dieses Objekt wie üblich zugegriffen werden und dessen Methoden wie die eines lokalen Objektes aufgerufen werden.

```
package RMIBeispiel;

import RMIBeispiel.DatabaseException;
import RMIBeispiel.GameServerImpl;
import RMIBeispiel.GameServer;

import java.rmi.*;
import java.rmi.registry.*;
import java.net.MalformedURLException;

public class GameServerClient {
    private GameServer gameServer;

    ...
```

...

```
public GameServerClient () {
    System.runFinalizersOnExit (true);      // deprecated
    System.setSecurityManager (
                          new RMISecurityManager ());

    try {
        gameServer = (GameServer)Naming.lookup (
                          "//kiew.cs.uni−dortmund.de"
                              + ":2060/GameServer");
    } catch (Exception e) { e.printStackTrace () }

    try {
        for (int i=1; i<=1000; i++) {
            System.out.println ("Platz "+ i + ": "
                          + gameServer.getUserScore(i));
        }
    } catch (DatabaseException e) { }
    catch (RemoteException e) {
        e.printStackTrace (); }

    try {
        gameServer.updateScore ("peter",1);
        gameServer.updateScore ("paul",100000);
        gameServer.updateScore ("mary",50);
    } catch (Exception e) { e.printStackTrace (); }

    ...
```

```
        ...

        try {
            for (int i=1; i<=1000; i++) {
                System.out.println ("Platz "+i+": "+
                                gameServer.getUserScore(i)) ;
            }
        } catch (DatabaseException e) { }
        catch (RemoteException e) {
            e.printStackTrace () ; }
    }

    public static void main (String[] args) {
        GameServerClient client = new GameServerClient ();
    }
}
```

Starten des Servers und des Clients

- Erzeugen der Stümpfe und Skelette mit dem `rmic`-Compiler des JDK:
  `rmic RMIBeispiel.GameServerImpl`
- Starten des Servers von Kommandozeile (beim JDK 1.2):

```
  java -Djava.rmi.server.codebase=
          file:/home/ralf/java/ProgVorlesung/Packages/RMIBeispiel \
       -Djava.security.policy=java.policy \
       RMI.Beispiel.GameServerDaemon &
```

- Starten des Clients von Kommandozeile (beim JDK 1.2):

```
  java -Djava.rmi.server.codebase=
          file:/home/ralf/java/ProgVorlesung/Packages/RMIBeispiel \
       -Djava.security.policy=java.policy \
       RMI.Beispiel.GameServerClient
```

- Datei `java.policy`:

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

- Siehe auch Hinweise im Skript sowie Quelltexte und Dateien in `~gvpr000/ProgVorlesung/Packages/RMIBeispiel/` .

Zu RMI ähnliche Konzepte verteilter Programmierung

- **Remote Procedure Call (RPC):** ältere Technik, die wie RMI von SUN entwickelt wurde und sprach- und prozessorunabhängig ist, während RMI nur in Java verfügbar ist. Dafür ist der Overhead bei RPC z.B. wegen notwendiger Datenkonvertierungen größer und RPC ist nicht auf die objektorientierte Programmierung ausgerichtet.

- **Common Object Request Broker Architecture (CORBA):** erlaubt den Austausch von Objekten zwischen Programmen, die in verschiedenen Programmiersprachen geschrieben sind. CORBA wird in Java unterstützt (*Java Interface Definition Language, IDL*).

Verteilte Programmierung in Java

Weitere von Java unterstützte Kommunikationsformen:

- Verbindungsorientierte Kommunikation über **Sockets**: Kommunikation zwischen Programmen auf dem Netzwerk, (simulierte) Verbindungen mit *Sockets* als Endpunkten, Austausch von Nachrichten nach einem anwendungsabhängig zu vereinbarenden Protokoll, z.B. auch in Client-Server-Systemen.

- Paketorientierte Kommunikation über **Datagramme**: Verbindungsloser Verkehr über einzelne Pakete, der Ankunft und Ankunftsreihenfolge nicht garantiert sind.

- **URL-Verbindungen** zum Laden von Dateien bzw. Dokumenten aus dem Internet.

Zusammenfassung der heutigen Themen

Netzwerkintegration und verteilte Programmierung

- Verteilte Programme, was man darunter versteht, Anwendungen
- Grundlegende Begriffe (Client, Server), Ziele
- Verteilte Programmierung in Java mit RMI
- Begriffe (Remote-Objekt, RMI, Stumpf, Skelett, Registratur)
- Aufbau und Funktionsweise von RMI-Client/Server-Anwendungen
- Game-Server-Beispiel
- Verwandte Konzepte (RPC, CORBA (IDL))
- Weitere Konzepte verteilter Programmierung in Java (Sockets, UDP)