B Fernaufruf-Systeme aus Benutzersicht

- Zweck dieser Übung
  - Programmieren mit den Basismechanismen von existierenden einfachen Fernaufruf-Systemen, aus Sicht des Benutzers
  - Kennenlernen der wichtigstens Probleme und Aufgaben eines Fernaufruf-Systems
  - Kennenlernen der bereitgestellten Mechanismen und Werkzeuge in diesen Systemen
- Betrachtet werden
  - Sun RPC (Remote Procedure Call)
  - Java RMI (Remote Methode Invocation)
- Erwartetes Ziel
  - Selbständiger Entwurf einer einfachen verteilten Anwendung mit Hilfe von Sun RPC und Java RMI

B Fernaufruf-Systeme aus Benutzersicht

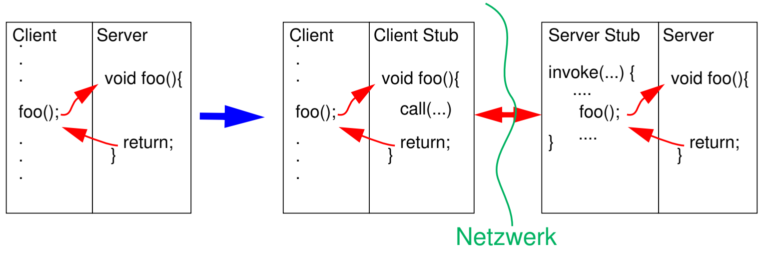
- Ziele in einem Fernaufruf-System
  - Transparenter Mechanismus (-> Netzwerktransparenz)
    - Methodenaufruf als Grundmechanismus im lokalen wie auch im verteilten Fall (Zugriffstransparenz)
    - Namensdienst: Zugriff auf Dienst, ohne Ort kennen zu müssen (Ortstransparenz)
  - Behandlung von Heterogenität (Heterogenitätstransparenz)
    - Sprache, Betriebssystem, Hardware, ...
  - Im Idealfall zusätzlich auch Fehlertransparenz, Migrationstransparenz, Replikationstransparenz, usw.
- Wir betrachten zunächst: Zugriffs- und Ortstransparenz in existierenden Systemen (Sun RPC, Java RMI)

B Fernaufruf-System aus Benutzersicht

- Sun RPC
  - Programmiersprache: Hauptsächlich C
  - Bibliotheken für Marshalling von Standard-Datentypen in architekturunabhängiges Netzwerkformat
  - Code-Generation aus Schnittstellenbeschreibung
- Java RMI
  - Programmiersprache: Java
  - In die Spreche integrierter Mechanismus
- Tiefergehende Grundlagen zu Fernaufrufen werden später in der Vorlesung aufgegriffen, und dann auch tiefgründiger gehend in der Übung behandelt (eigene Implementierung eines RPC-Systems!)

B.1 Sun RPC

- Überblick SUN-RPC
  - SUN-RPC basiert auf dem Stub-Mechanismus



  - Utility "rpcgen" zur automatischen Generierung der Stubs
  - Doku: "ONC+ Developer's Guide", Sun Microsystems
    `http://docs.sun.com/db/doc/805-7224`

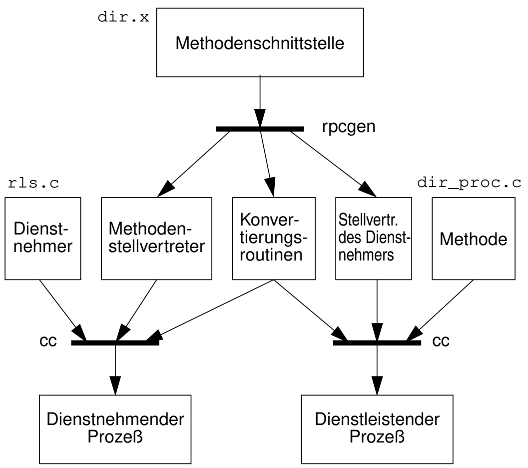
B.1 Sun RPC

Eigenschaften des Sun RPC

- Pro RPC-Aufruf ist lediglich ein Parameter erlaubt. Benötigt der RPC mehrere Parameter, muss eine Hilfsstruktur definiert werden! (Diese Aufgabe kann der rpcgen übernehmen)
- Der Host, an dem der RPC aufgerufen wird, muss beim Aufruf bekannt sein. Übliche Abbildungen via DNS und NIS/NIS+ werden durchgeführt.
- Als Transportschicht kann UDP und TCP verwendet werden.
- Für das Marshalling von Parametern und Rückgabewerten wird der XDR-Standard verwendet. Dadurch wird eine gewisse Kompatibilität mit alternativen Implementationen gewährleistet.
- Ein RPC wird durch ein Tripel (Programm-Nummer, Versions-Nummer, Prozedur-Nummer) identifiziert. Die Programm-Nummer repräsentiert dabei eine Gruppe von verwandten RPC's.

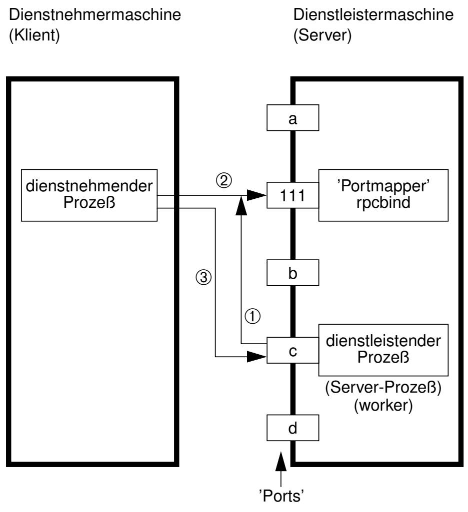
B.1 Sun RPC

Generierung von Klient und Server am Beispiel eines Fernaufrufs zur Ermittlung des Inhalts eines Verzeichnisses ("remote ls")



B.1 Sun RPC

Ports und Portmapper: Namensdienst



B.1 Sun RPC

Erzeugung der Marshaling-Methoden

```
faui40% rpcgen dir.x
   dir.x      Beschreibung der Methodenschnittstelle
              und der Einbettung in den Server
```

=> Erzeugt verschiedene Hilfsdateien:

```
dir.h          gemeinsame Typdefinitionen
dir_svc.c      Rahmenprogramm des Auftragnehmers
dir_xdr.c      Prozeduren zur (Rück-)Konvertierung
               in/aus der Datendarstellung, die bei
               der Übertragung von Parametern
               verwendet wird.
dir_clnt.c     Stellvertreter der Methode.
```

B.1 Sun RPC

- Erzeugung des Klienten (rls.c)

```
faui40%  cc rls.c dir_clnt.c dir_xdr.c -o rls -lnsl
```

- Erzeugung des Servers (Implementierung in dir_proc.c)

```
faui40%  cc dir_proc.c dir_svc.c dir_xdr.c -o dir_svc -lnsl
```

- Ausführung
  - ◆ Start des Dienstes (Server):

```
dir_svc&
```

  - ◆ Start des Klienten

```
rls faui02a ~
```


B.1 Sun RPC

- Datentypen im unverteilten Fall
  - ◆ Typ Dateinamen

```
typedef char *nametype;
```

  - ◆ Verkettete Liste aus Dateinamen

```
typedef struct namenode *namelist
struct namenode {
    nametype name;
    namelist next;
};
```

  - ◆ Ergebniscode, und falls dieser gleich 0, Liste der Dateinamen

```
struct readdir_res {
    int errno;
    namelist list;
};
```


B.1 Sun RPC

- Datentypen im verteilten Fall (Fernaufrufe): dir.x
  - ◆ Typ der Dateinamen

```
const MAXNAMELENGTH = 255;
typedef string nametype<MAXNAMELENGTH>;
```

  - ◆ Verkettete Liste aus Dateinamen

```
typedef struct namenode *namelist
struct namenode {
    nametype name;
    namelist next;
};
```

  - ◆ Typ des Methodenergebnisses, bestehend aus einer Fehlerkennung und (falls sie den Wert 0 hat), einer Liste von Dateinamen.

```
union readdir_res switch(int errno) {
  case 0:  namelist list;
  default: void;
};
```


B.1 Sun RPC

- ◆ Anmerkung:

  Bei "union" wird in den Programmen eine Struktur mit einer zusätzlichen Komponente gebildet. Die Union-Komponente wird durch den Union-Typnamen mit angeschlossener Zeichenfolge _u identifiziert.

```
struct readdir_res {
        int errno;
        union {
                namelist list;
        } readdir_res_u;
};
```

B.1 Sun RPC

- Die Schnittstellenbeschreibung (dir.x)

```
program DIRPROG {
    version DIRVERS {
        readdir_res READDIR(nametype) = 1;
    } = 1;
} = 0x20000076;
```

- ◆ Der Server erhält den Namen DIRPROG
- ◆ Der Server wird unter der Nummer 0x20000076 registriert.
- ◆ Die Version DIRVERS (==1) stellt die Methode READDIR zur Verfügung, die unter der Methodennummer 1 erreicht wird. Dienstnehmer rufen diese Methode mit dem Namen readdir_1 auf (die Zahl nach dem Unterstrich ist die Versionsnummer).

B.1 Sun RPC

- XDR (eXternal Data Representation)

XDR sieht ein Muster zur Konvertierung zwischen C-Datentypen und einer externen Darstellung als Bitfolge vor. Für die Standardtypen von C sind Bibliotheksroutinen verfügbar. Diese Routinen zusammen mit unterstützenden Routinen bilden die Grundlage, auf der für jeden benutzerdefinierten Datentyp Routinen implementiert werden können, die die Konvertierung in beiden Richtungen vornehmen. Für jeden Datentyp wird eine Routine benötigt mit zwei Argumenten:

```
bool_t xdr_<type>(XDR *xdrs, <type> argresp)
```

xdrs verweist auf ein XDR-Objekt, in das bzw. aus dem konvertiert werden soll. Das XDR-Objekt enthält eine Komponente, die angibt, ob nach XDR konvertiert werden soll (encode) oder aus XDR (decode).

B.1 Sun RPC

- Struktur eines XDR-Objektes

```
enum xdr_op  {XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
    enum xdr_op x_op;
    struct xdr_ops {
        bool_t (*x_getlong)();
        bool_t (*x_putlong)();
        bool_t (*x_getbytes)();
        bool_t (*x_putbytes)();
        u_int (*x_getpostn)();
        bool_t (*x_setpostn)();
        long * (*x_inline)();
        void (*x_destroy)();
    } *x_ops;
    caddr_t x_public;      /* users' data */
    caddr_t x_private;     /* pointer to private data */
    caddr_t x_base;        /* private for position info */
    int     x_handy;       /* extra private word */
} XDR;
```


B.1 Sun RPC

- Aus den Datenstrukturen für rls werden unter Rückgriff auf die Konversionsroutinen für Standardatentypen folgende Routinen erzeugt; (dir_xdr.c)

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include "dir.h"
bool_t
xdr_nametype(XDR *xdrs, nametype *objp)
{
    register long *buf;
    if (!xdr_string(xdrs, objp, MAXNAMELENGTH))
        return (FALSE);
    return (TRUE);
}
```

B.1 Sun RPC

```
bool_t xdr_namelist(XDR *xdrs, namelist *objp)
{
    register long *buf;

    if (!xdr_pointer(xdrs, (char **)objp,
         sizeof (struct namenode),(xdrproc_t) xdr_namenode))
        return (FALSE);
    return (TRUE);
}

bool_t xdr_namenode(XDR *xdrs, namenode *objp)
{
    register long *buf;

    if (!xdr_nametype(xdrs, &objp->name))
        return (FALSE);
    if (!xdr_namelist(xdrs, &objp->next))
        return (FALSE);
    return (TRUE);
}
```


B.1 Sun RPC

```
bool_t xdr_readdir_res(XDR *xdrs, readdir_res *objp)
{
    register long *buf;

    if (!xdr_int(xdrs, &objp->errno))
        return (FALSE);
    switch (objp->errno) {
    case 0:
        if (!xdr_namelist(xdrs, &objp->readdir_res_u.list))
            return (FALSE);
        break;
    }
    return (TRUE);
}
```


B.1 Sun RPC

- Die Implementierung des Servers (dir_proc.c)

```
// Der Methodenname besteht aus einer symbolischen Bezeichnung
// gefolgt von einem Unterstrich und der Versionsnummer.
// Parameter- und Ergebnistypen werden in der
// Methodenschnittstelle definiert.
readdir_res *
readdir_1(nametype *dirname, struct svc_req *req)
{
  DIR *dirp;
  struct dirent *d;
  namelist *nlp, nl;
  static readdir_res res;

  dirp = opendir(*dirname);
  if(dirp == NULL) {
    res.errno = errno;
    return(&res);
  }
  // alten Speicher freigeben
  xdr_free(xdr_readdir_res, (char *)&res);
```


B.1 Sun RPC

```
  nlp = &res.readdir_res_u.list;
  printf("\n");
  while (d = readdir(dirp)) {
    nl = *nlp
       = (namenode *) malloc(sizeof(namenode));
    nl->name = strdup(d->d_name);
    nlp = &nl->next;
  }
  *nlp = NULL;

  res.errno = 0;
  closedir(dirp);
  printf("\n");
  return(&res);
}
```

B.1 Sun RPC

- Die Implementierung des Klienten (rls.c)

  Das Programm wird mit zwei Parametern gestartet, nämlich dem Namen des Rechners, auf dem das auszugebende Verzeichnis liegt und dem Pfadnamen des auszugebenden Verzeichnisses

```
main(int argc, char *argv[])
{
    CLIENT *cl;
    char *server, *dir;
    readdir_res *result;
    namelist nl;

    // Übernahme der Parameter.
    server = argv[1];
    dir = argv[2];
```


B.1 Sun RPC

```
    // Erzeugung eines Deskriptors für den Serverzugang
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");


    // Aufruf der Methode readdir am durch cl identifizierten
    // Server
    result = readdir_1(&dir, cl);

    // Ausgabe des Ergebnisses aus dem Methodenaufruf
    for (nl = result->readdir_res_u.list;
        nl != NULL;
        nl = nl->next) {
        printf("%s\n", nl->name);
    }

  exit(0);
}
```


B.1 Sun RPC

- Der Stellvertreter des Klienten (automatisch generiert)

```
static struct timeval TIMEOUT = { 25, 0 };

readdir_res *
readdir_1(nametype argp, CLIENT *clnt)
{
    static readdir_res clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, READDIR,
            (xdrproc_t) xdr_nametype, (caddr_t) argp,
            (xdrproc_t) xdr_readdir_res, (caddr_t) &clnt_res,
            TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```


B.1 Sun RPC

- Der Stellvertreter des Servers (automatisch generiert)

```
main()
{   register SVCXPRT *transp;
    struct netconfig *nconf = NULL;

    // Erzeuge eine RPC Server Handle
    transp = svc_tli_create(0, nconf, NULL, 0, 0))

    Registriere Server mit DIRPROG und DIRVERS
    svc_reg(transp, DIRPROG, DIRVERS, dirprog_1, 0);

    // Diese Routine kehrt nur bei einem Fehler zurück.
    // Sie wartet auf RPC-Aufrufe und leitet diese an die
    // entsprechende Service-Routine weiter
    svc_run();
}
```

B.1 Sun RPC

- Zusammenfassung
  - Explizite Schnittstellenbeschreibung ("dir.x")
  - Daraus automatische Generierung von Stubs und Marshalling-Funktionen mit Hilfe von "rpcgen"
  - Einheitliche Informationsdarstellung nach XDR für heterogene Systeme
  - Portmappter als lokaler Namensdienst

B.2 Jave Remote Methode Invocation (RMI)

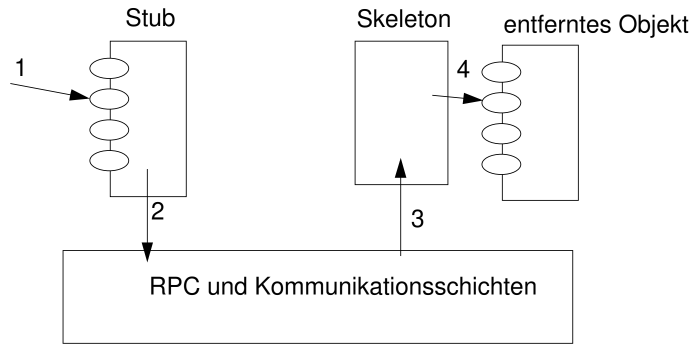
- ermöglicht Abstraktion in einem verteilten System

Socket-Kommunikation	keine Abstraktion
↓ Fernaufrufe (RPC)	prozedurale Abstraktion
↓ FernMethodenaufrufe (RMI)	Abstraktion auf Objektebene

- FernMethodenaufrufe: Aufruf von Methoden an Objekten auf anderen Rechnern
- Transparente Objektreferenzen zu entfernten Objekten
- Dokumentation: z.B.
  `http://java.sun.com/products/jdk/rmi/`

B.2 Überblick

- Stub: Stellvertreter (Proxy) des entfernten Objekts.
- Skeleton: Ruft die Methoden am entfernten Objekt auf.



- Anfrage: Objekt ID, Methoden ID, Parameter

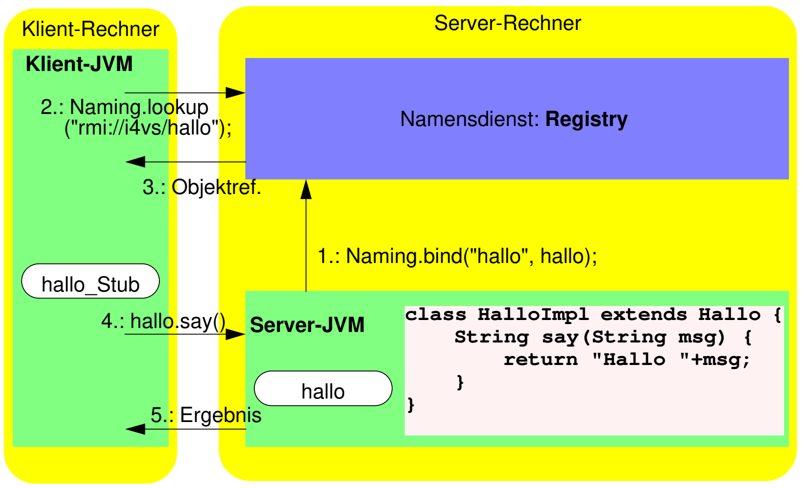
B.2 Einführung

- *Remote Object* (entferntes Objekt): Ein Objekt, welches aus einer anderen JVM heraus genutzt werden kann
- *Remote Interface*: Beschreibt die Methoden eines entfernten Objekts
- Ein *Remote Interface* muss von **`java.rmi.Remote`** abgeleitet sein
- Zugriffe auf ein entferntes Objekt sind nur über ein *Remote Interface* möglich
- Die Klasse eines entfernten Objekts muss mindestens ein *Remote Interface* implementieren
- Entfernte Objekte können selbst wieder entfernte Objekte nutzen.
- Parameter werden wie folgt übergeben:
  - by value: Bei allen Parametern, die keine entfernten Objekte sind
  - by reference: Bei Parametern welche **`java.rmi.Remote`** implementieren

B.2 lokaler vs. entfernter Methodenaufruf

- Jeder Aufruf kann eine **`RemoteException`** auslösen
  - Aufrufer weiß nicht, ob die Methode komplett, teilweise, oder gar nicht ausgeführt wurde
- Entfernte Objekte können nur über Interfaces angesprochen werden
- Normale Objekte werden kopiert, anstatt eine Referenz darauf zu übergeben

B.2 Ausführung



B.2 Registry (1)

- Die *Registry* übernimmt als Namesdienst die Abbildung von Objektnamen auf Objektreferenzen
- Der Zugriff erfolgt über die Klasse **`java.rmi.Naming`**
  - **`void bind(String name, Remote obj)`**
    registriert ein Objekt unter einem eindeutigen Namen;
    falls das Objekt bereits registriert ist wird eine Exception ausgelöst
  - **`void rebind(String name, Remote obj)`**
    registriert ein Objekt unter einem eindeutigen Namen;
    falls das Objekt bereits registriert ist wird der alte Eintrag gelöscht
  - **`Remote lookup(String name)`**
    liefert die Objektreferenz zu einem gegebenen Namen
  - **`void Naming.unbind(String name)`**
    löscht den Namenseintrag
- Die Registrierung ist nur bei der Registry auf dem aktuellen Rechner möglich!

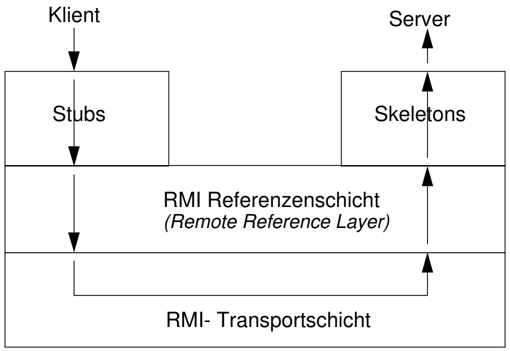
B.2 Registry (2)

- Einen bestimmten Port verwenden:
  - Die Registry nimmt TCP/IP-Verbindungen an Port 1099 entgegen.
  - als Parameter kann jedoch ein anderer Port angegeben werden
    z.B. 10412:
    `rmiregistry 10412`
  - Wenn eine Registry an einem bestimmten Port verwendet werden soll, so muss die URL, die bei **`bind/rebind/unbind/lookup`** verwendet wird diesen Port enthalten:

```
Naming.rebind("rmi://faui40:10412/hallo", hallo)

Naming.lookup("rmi://faui40:10412/hallo")
```

B.2 Systemarchitektur

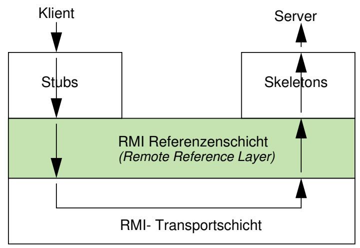


B.2 Stubs und Skeletons

- Stub (auf Klientenseite) - implementiert das *remote interface*
  1. erhält einen `ObjectOutputStream` von der RMI - Referenzenschicht
  2. schreibt die Parameter in diesen Strom
  3. teilt der RMI - Referezenschicht mit, die Methode aufzurufen
  4. holt einen `ObjectInputStream` von der RMI - Referezenschicht
  5. liest das Rückgabeobjekt aus diesem Strom
  6. liefert das Rückgabeobjekt an den Aufrufer
- Skeleton (auf Serverseite)
  1. erhält einen `ObjectInputStream` von der RMI - Referezenschicht
  2. liest die Parameter aus diesem Strom
  3. ruft die Methode am implementierten Objekt auf
  4. holt einen `ObjectOutputStream` von der RMI - Referezenschicht
  5. schreibt das Rückgabeobjekt in diesem Strom

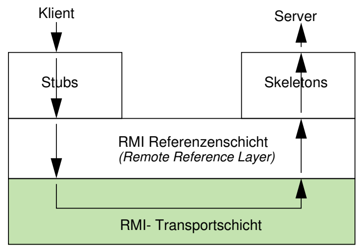
B.2 Referenzenschicht (Remote Reference Layer)

- Verantwortlich für das Aufräumen (Garbage Collection) der entfernten Objekte
- Implementiert die Aufrufsemantik, z.B.:
  - unicast, Punkt-zu-Punkt
  - Aufruf an einem replizierten Objekt
  - Strategien zum Wiederaufbau der Verbindung nach einer Unterbrechung

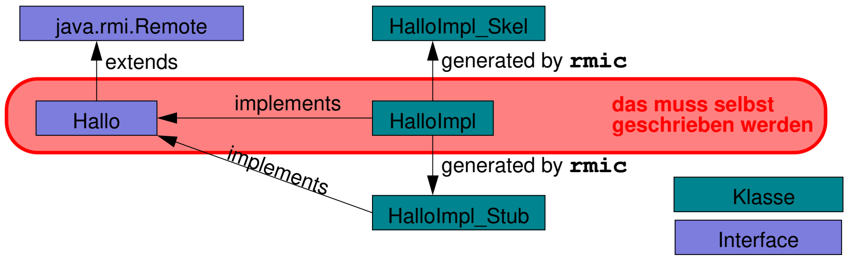


B.2 Transportschicht

- Verantwortlich für die Datenübertragung zwischen den Rechnern
- Aktuelle Implementierung basiert auf TCP/IP Sockets
- Verschiedene Transportmechanismen sind denkbar



B.2 Interface und Implementierung



- Der Programmierer schreibt das *remote Interface* **`Hallo`** und die Implementierung **`HalloImpl`**
- Der Stellvertreter (Stub) **`HalloImpl_Stub`** und das Skeleton **`HalloImpl_Skel`** werden durch **`rmic`** aus der Implementierung **`HalloImpl`** generiert

B.2 Ein einfaches Beispiel

1. Remote Interface
2. Server
3. Serverinitialisierung
4. Klient
5. Starten des Systems

1 Beispiel - Remote Interface

- Jedes entfernte Objekt muss ein Interface implementieren, das alle Methoden enthält, die das Objekt seinen Klienten anbieten soll
- Das Interface muss von **`java.rmi.Remote`** erben
- Alle Methoden können **`java.rmi.RemoteException`** werfen
- Alle Parameter und Rückgabewerte müssen
  - entweder serialisierbar sein (d.h. sie müssen das Interface **`java.io.Serializable`** implementieren)
  - oder sie müssen entfernte Objekte sein.

1 Beispiel - Remote-Interface

- Remote Interface *Bank*:

```
public interface Bank extends java.rmi.Remote {
    void deposit(Money amount, Account account)
                           throws java.rmi.RemoteException;
}
```

- Remote Interface *Account*:

```
public interface Account extends java.rmi.Remote {
        void deposit(Money amount)
                           throws java.rmi.RemoteException;
}
```

- Parameter *Money*:

```
public Klasse Money implements java.io.Serializable {
    private float value;
    public Money(float value) { this.value = value; }
}
```

2 Beispiel - Server

- Der Server implementiert das Interface
- Methoden brauchen keine **`RemoteException`** zu werfen.
- Zwei Alternativen zur Erzeugung eines Remote-Objektes
  - ◆ Der Server wird von **`UnicastRemoteObject`** abgeleitet

```
import java.rmi.server.*;
public Klasse BankImpl extends UnicastRemoteObject implements Bank
{
        public BankImpl () throws java.rmi.RemoteException {...}

        public void deposit(Money amount, Account account)
                                throws java.rmi.RemoteException {
                account.deposit(amount);
        }
}
```

  - ◆ oder es wird mit **`exportObject`** ein Remote-Objekt erzeugt:

```
public Klasse BankImpl implements Bank {... }
Remote bank = UnicastRemoteObject.exportObject(new BankImpl())
```


3 Beispiel - Serverinitialisierung

- Das Remote-Objekt muss mit **`bind`** oder **`rebind`** in die Registry eingetragen werden.

```
Naming.bind("rmi://faui40u:10412/bank", bank);
```

Protokoll   Rechnername   Objektname

- RMISecurityManager:
  - ◆ ohne SecurityManager können keine Klassen vom Netz geladen werden
  - ◆ Server-JVM muss einen SecurityManager setzen, z.B.

```
System.setSecurityManager(new RMISecurityManager());
```

  - ◆ **`RMISecurityManager`** ist ähnlich **`AppletSecurityManager`**

4 Beispiel - Klient

- Der Klient kann sich mit **`lookup`** eine Referenz auf das entfernte Objekt (den Server) besorgen
- Beispiel:

```
public Klasse Client {
    public static void main (String argv[]) throws
                                java.net.MalformedURLException,
                                java.rmi.NotBoundException,
                                java.rmi.RemoteException  {
        Bank bank= (Bank)java.rmi.Naming.lookup
                                ("rmi://faui40u:10412/bank");
        Account account = new AccountImpl();
        bank.deposit(new Money(10), account);
    }
}
```


5 Beispiel - System starten

- Klassen-Pfad setzen, damit **`rmic`** und java (server) die Klassen finden

```
setenv CLASSPATH /proj/i4vs/felser/aufgabe1
```

- Stubs und Skeletons erzeugen

```
rmic -d ${CLASSPATH} BankImpl
```

- Auf dem Serverrechner die Registry starten

```
rmiregistry 10412&
```

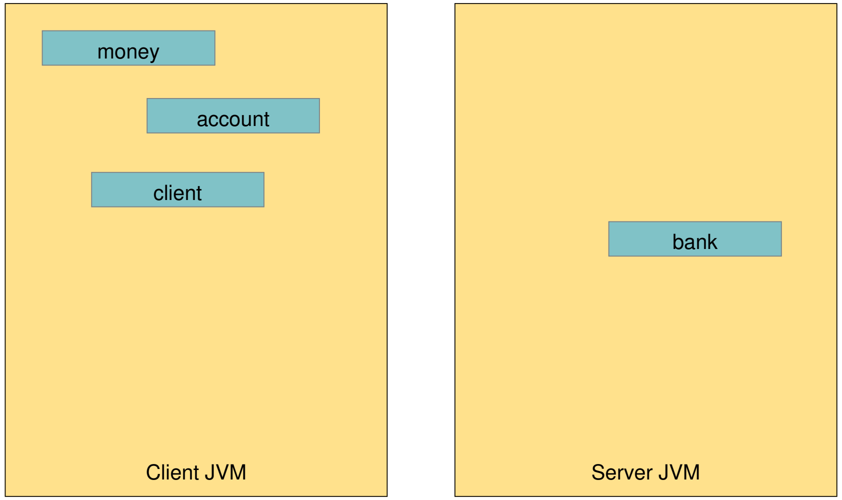
- Auf dem Serverrechner das Serverobjekt starten

```
java -Djava.rmi.server.codebase=
              file:///proj/i4vs/felser/aufgabe1/ BankImpl
```

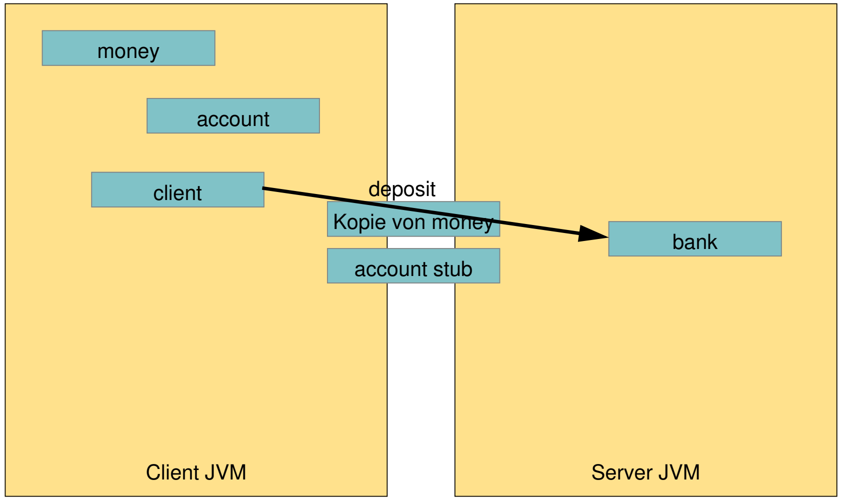
- Von jedem beliebigen Rechner aus kann der Klient nun benutzt werden:

```
java Client
```

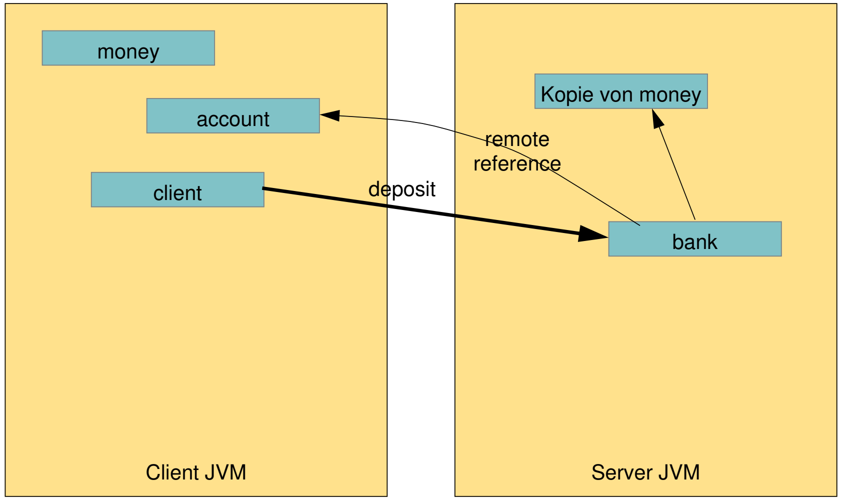
6 Interaktionen während des FernMethodenaufrufes



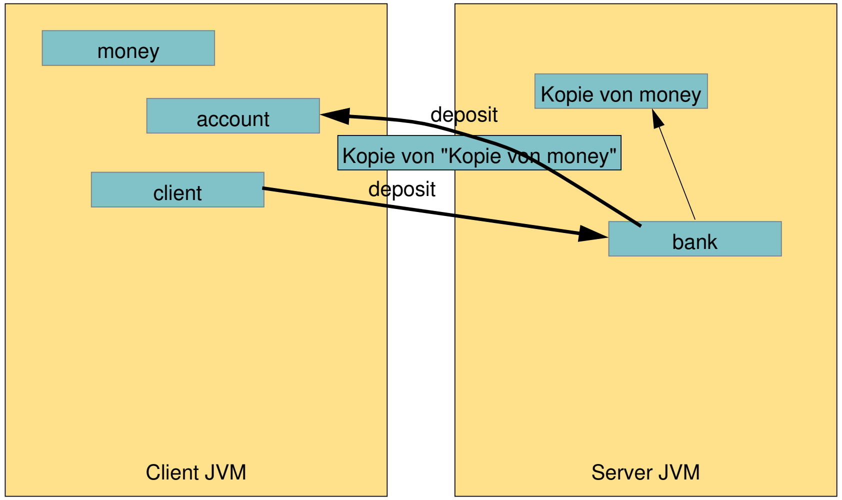
6 Interaktionen während des FernMethodenaufrufes (2)



6 Interaktionen während des FernMethodenaufrufes (3)



6 Interaktionen während des FernMethodenaufrufes (4)



B.2 Zusammenfassung

- Ein entferntes Objekt muss ein “*remote Interface*” implementieren.
- Alle Methoden des Interfaces müssen eine **`java.rmi.RemoteException`** werfen.
- Um ein entferntes Objekt zu erzeugen, kann man entweder direkt von **`java.rmi.server.UnicastRemoteObject`** ableiten, oder die Methode **`exportObject`** verwenden
- Um entfernte Objekte an der *Registry* zu registrieren oder zu suchen, kann man **`java.rmi.Naming`** verwenden.
- Klienten verwenden nur das remote interface.