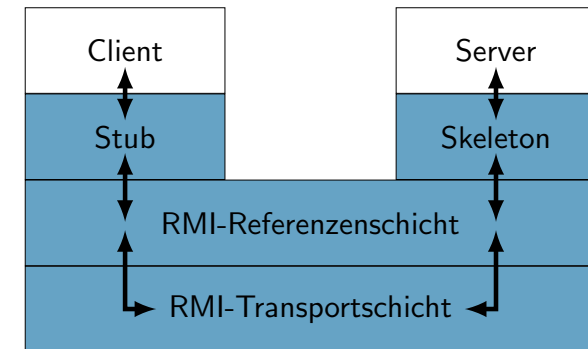


Java RMI

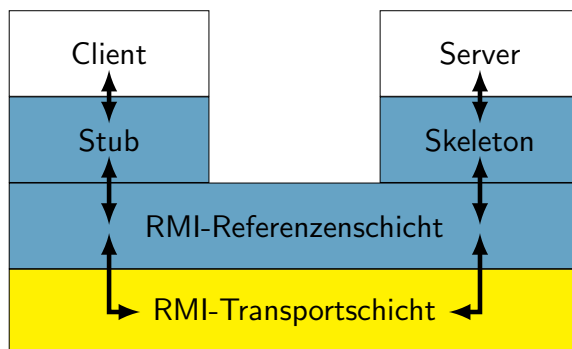
Java Remote Method Invocation
Marshalling und Unmarshalling
Aufgabe 1



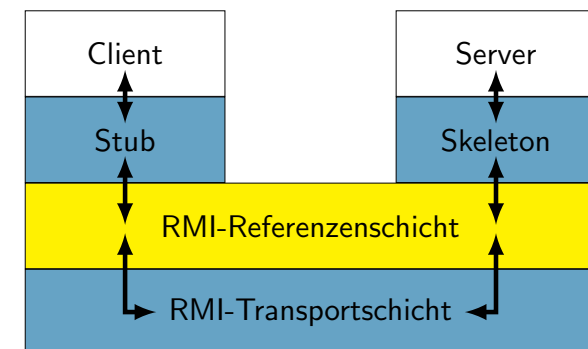
- Remote Method Invocation (RMI)
 - Aufrufe von Methoden an Objekten auf anderen Rechnern
 - *Remote-Referenz*: Transparente Objektreferenz zu entferntem Objekt
- Beispiel: Java RMI



- Datenübertragung zwischen Rechnern
- Implementierung
 - Aktueller Standard: Verwendung von TCP/IP-Sockets
 - Generell: verschiedene Transportmechanismen denkbar



- Verwaltung von Remote-Referenzen
- Implementierung der Aufrufsemantik (Beispiele)
 - Unicast, Punkt-zu-Punkt
 - Aufruf an einem replizierten Objekt
 - Strategien zum Wiederaufbau der Verbindung nach einer Unterbrechung

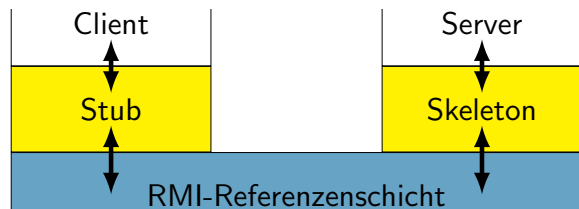


■ Stub

1. erhält einen Objekt-Ausgabe-Strom von der RMI-Referenzschicht
2. schreibt die Parameter in diesen Strom
3. teilt der RMI-Referenzschicht mit, die Methode aufzurufen
4. holt einen Objekt-Eingabe-Strom von der RMI-Referenzschicht
5. liest das Rückgabe-Objekt aus diesem Strom
6. liefert das Rückgabe-Objekt an den Aufrufer

■ Skeleton

1. erhält einen Objekt-Eingabe-Strom von der RMI-Referenzschicht
2. liest die Parameter aus diesem Strom
3. ruft die Methode am implementierten Objekt auf
4. holt einen Objekt-Ausgabe-Strom von der RMI-Referenzschicht
5. schreibt das Rückgabe-Objekt in diesen Strom



■ Remote-Objekt (entferntes Objekt)

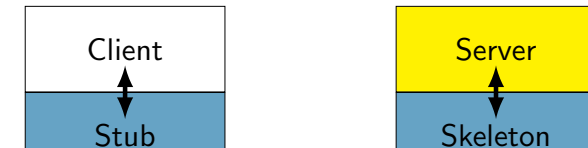
- Kann aus einer anderen Java Virtual Machine heraus genutzt werden
- Erst von außerhalb erreichbar, nachdem es **exportiert** wurde

■ Remote-Schnittstelle

- Beschreibt die per Fernaufruf erreichbaren Methoden des Objekts
- Abgeleitet von `java.rmi.Remote` (Marker-Schnittstelle)
- Einzige Möglichkeit mit Java RMI auf ein entferntes Objekt zuzugreifen

■ Remote-Exception (`java.rmi.RemoteException`)

- Muss im `throws`-Clause jeder Remote-Methode angegeben sein
- Beim Auftreten einer Remote-Exception weiß der Aufrufer nicht, ob die Methode komplett, teilweise oder gar nicht ausgeführt wurde



■ Namensdienst

- Bekanntmachen von Remote-Objekten
- Abbildung von Objektnamen auf Objektreferenzen

■ Registry-Schnittstelle

```
public interface Registry extends Remote {
    public void bind(String name, Remote obj);
    public Remote lookup(String name);
    [...]
}
```

- `bind()` Zuordnung eines Objekts zu einem eindeutigen Namen
- `lookup()` Rückgabe der Remote-Referenz zu einem Namen

■ Erzeugung und Verbindung zur Registry

```
public class LocateRegistry {
    public static Registry createRegistry(int port);
    public static Registry getRegistry(String host, int port);
    [...]
}
```

- `createRegistry()` Erzeugung einer Registry auf dem lokalen Rechner
- `getRegistry()` Holen einer Remote-Referenz auf eine Registry

■ Geldbetrag `VSMoney`

```
public class VSMoney implements Serializable {
    private float amount;

    public VSMoney(float amount) {
        this.amount = amount;
    }

    public float getAmount() { return amount; }
}
```

■ Konto `VSAccount` (Remote-Schnittstelle)

```
public interface VSAccount extends Remote {
    public void deposit(VSMoney money) throws RemoteException;
}
```

■ Bank `VSBank` (Remote-Schnittstelle)

```
public interface VSBank extends Remote {
    public void deposit(VSMoney money, VSAccount account)
        throws RemoteException;
}
```



- VSBankImpl: Implementierung der Remote-Schnittstelle VSBank
- Exportieren des Remote-Objekts
 - Implizit: Unterklasse von `java.rmi.server.UnicastRemoteObject`

```
public class VSBankImpl extends UnicastRemoteObject
    implements VSBank {

    // Konstruktor
    public VSBankImpl() throws RemoteException { super(); }

    // Implementierung der Remote-Methode
    public void deposit(VSMoney money, VSAccount account)
        throws RemoteException {
        account.deposit(money);
    }
}
```

```
VSBank bank = new VSBankImpl();
```

- Explizit: Aufruf von `UnicastRemoteObject.export()`

```
public class VSBankImpl implements VSBank { [...] }
VSBank b = new VSBankImpl();
VSBank bank = (VSBank) UnicastRemoteObject.exportObject(b, 0);
```



- Konto-Implementierung `VSAccountImpl`
 - Implementierung der Remote-Schnittstelle `VSAccount`
 - Exportieren analog zu `VSBankImpl`
 - Synchronisation paralleler `deposit()`-Aufrufe
 - [Auf welchem Rechner erscheint die Bildschirmausgabe?]

```
public class VSAccountImpl implements VSAccount {
    private float amount;

    public VSAccountImpl(float amount) {
        this.amount = amount;
    }

    public synchronized void deposit(VSMoney money) {
        amount += money.getAmount();
        System.out.println("New amount: " + amount);
    }
}
```



- Server-Implementierung `VSBankServer`
 - Erzeugen des Remote-Objekts
 - Exportieren des Remote-Objekts
 - Remote-Objekt mittels Registry bekannt machen

```
public class VSBankServer {
    public static void main(String[] args) throws Exception {
        // Remote-Objekt erzeugen
        VSBank bankImpl = new VSBankImpl();

        // Remote-Objekt exportieren
        VSBank bank = (VSBank)
            UnicastRemoteObject.exportObject(bankImpl, 0);

        // Remote-Objekt bekannt machen
        Registry registry =
            LocateRegistry.createRegistry(12345);
        registry.bind("bank", bank);
    }
}
```



- Client-Implementierung `VSBankClient`

```
public class VSBankClient {
    public static void main(String[] args) throws Exception {
        // Geldbetrag-Objekt anlegen
        VSMoney money = new VSMoney(10.0f);

        // Account anlegen und exportieren
        VSAccount accountImpl = new VSAccountImpl(100.0f);
        VSAccount account = (VSAccount)
            UnicastRemoteObject.exportObject(accountImpl, 0);

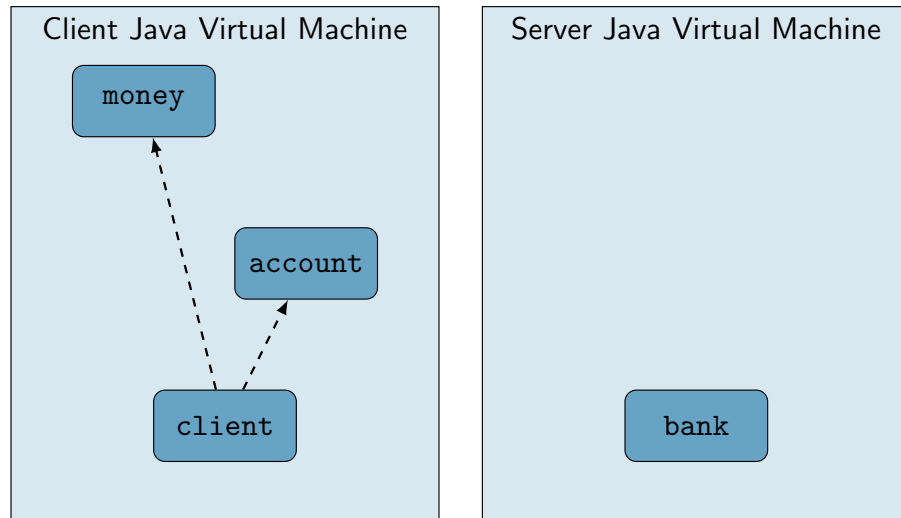
        // Remote-Referenz holen (Annahme: Server auf faui05a)
        Registry registry =
            LocateRegistry.getRegistry("faui05a", 12345);
        VSBank bank = (VSBank) registry.lookup("bank");

        // Geld einzahlen
        bank.deposit(money, account);

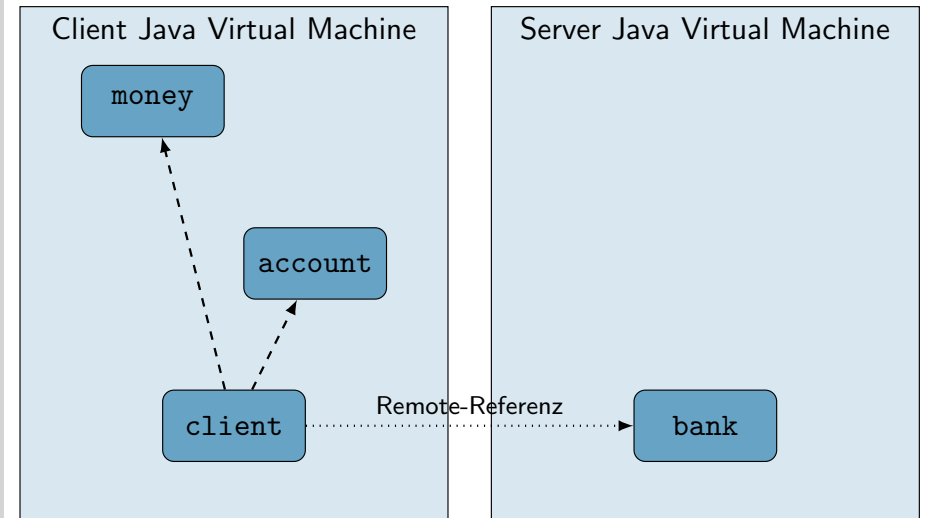
        System.exit(0);
    }
}
```



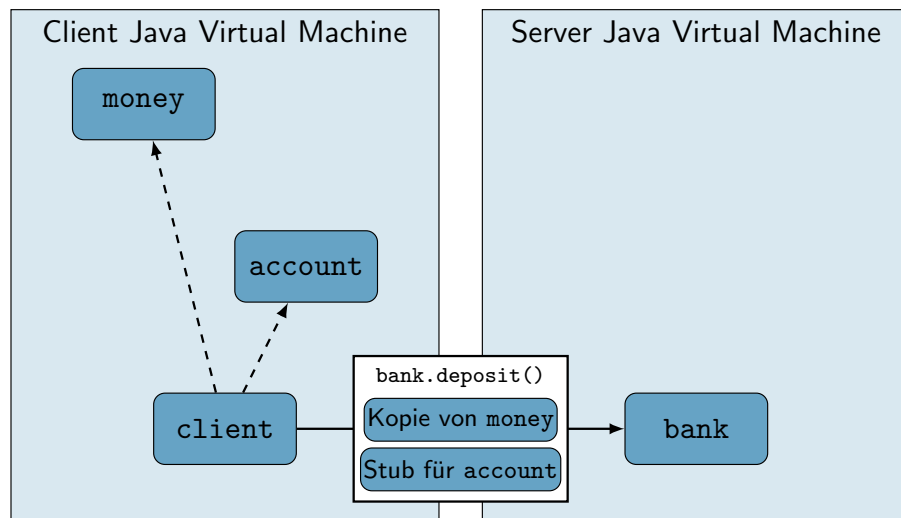
- Ausgangssituation vor Registry-Zugriff des Client



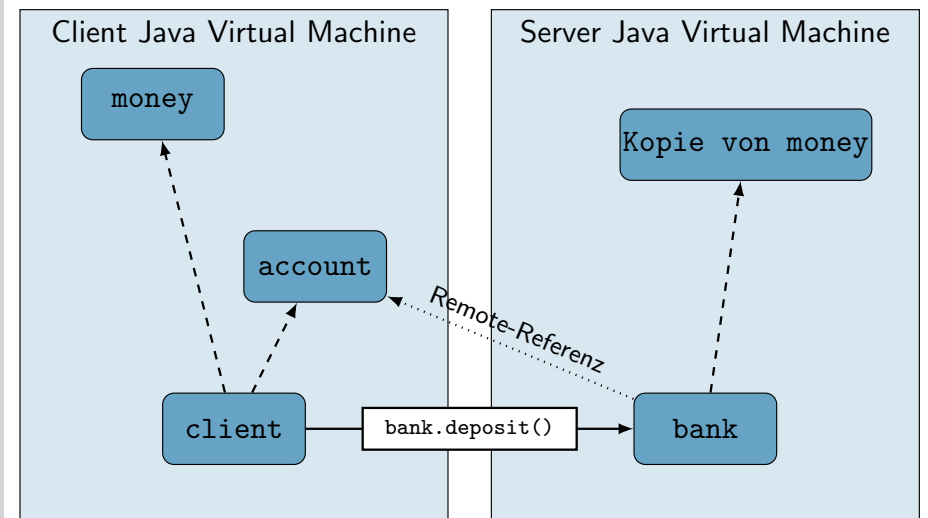
- Remote-Referenz auf bank von Registry holen



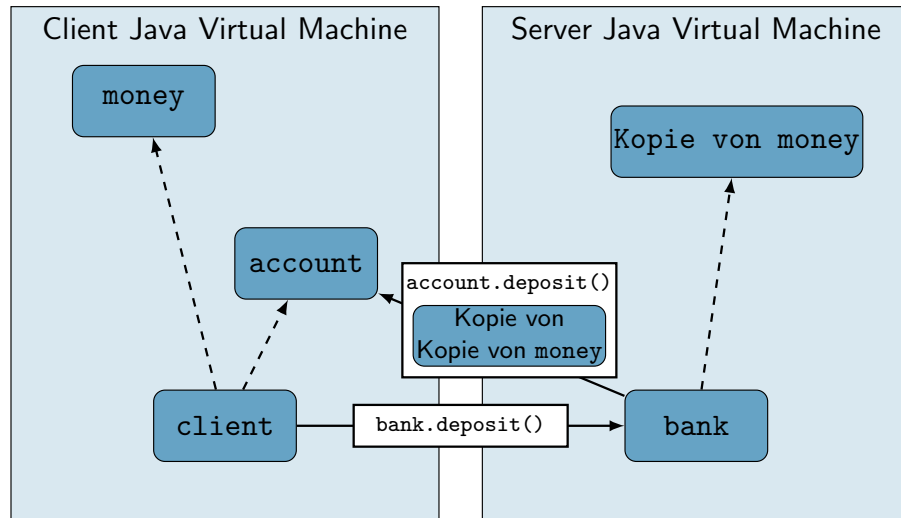
- Methodenaufruf von bank.deposit()



- Nach dem Auspacken der Parameter



- Methodenaufruf von `account.deposit()`



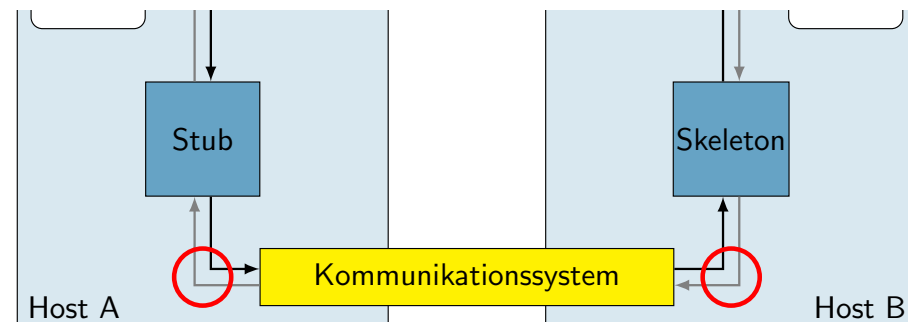
Java RMI

Java Remote Method Invocation
Marshalling und Unmarshalling
Aufgabe 1



Marshalling und Unmarshalling

- Definition
 - Marshalling*: Verpacken von Informationen in einer Nachricht
 - Unmarshalling*: Auspacken von Informationen aus einer Nachricht
- Problemstellungen
 - Unterschiedliche Datentypen
 - Heterogenität bei der lokalen Repräsentation von Datentypen
 - Unterschiedliche Parameterübergabearten



Unterschiedliche Datentypen

- Primitive Datentypen
 - z. B. `char`, `boolean`, `int`, ...
 - Benutzerdefinierte Datentypen
 - z. B. `structs`
 - Felder
 - z. B. `int[47]`
 - Referenzen
 - z. B. `char *`
 - Objekte
 - z. B. `Strings`, `Dateien`, ...
 - ...
- ⇒ **Kein allgemeines Vorgehen möglich**



- „Byte Sex“-Problem
 - Big Endian (Network Byte Order)
 - Most-significant byte first
 - z. B. SPARC, Motorola
 - Little Endian
 - Least-significant byte first
 - z. B. Intel x86
- Repräsentation von Fließkommazahlen
 - Allgemein
 - Vorzeichen (s)
 - Mantisse (m)
 - Exponent (e)
 - Zahlenwert: $(-1)^s * m * 2^e$
 - Variationsmöglichkeiten
 - Anzahl der Bits für m und e
 - Speicherreihenfolge von m , e und s
 - Byte-Order



- Kanonische Repräsentation
 - Nutzung einer allgemeingültigen Form als Zwischenrepräsentation
 - z. B. IEEE-Standard
- ⇒ Eventuell unnötige Konvertierungen
[z. B. wenn Sender und Empfänger identische Repräsentation nutzen]
- „Sender makes it right“
 - Sender kennt Datenrepräsentation des Empfängers
 - Sender konvertiert Daten
- ⇒ Multicast an heterogene Gruppe nicht möglich
- „Receiver makes it right“
 - Kennzeichnung des Datenformats
 - Empfänger konvertiert Daten
- ⇒ Bereitstellung sämtlicher Konvertierungsroutinen notwendig
[Unproblematisch für Byte-Order-Konvertierung]



Unterschiedliche Parameterübergabearten

- Eingabeparameter
 - Teil der Anfragenachricht
 - z. B. value in `void setValue(int value);`
- Ausgabeparameter
 - Teil der Antwortnachricht
 - z. B. String-Rückgabewert in `String toString();`
- Ein-/Ausgabeparameter
 - Teil der Anfrage- und Antwortnachricht
 - z. B. stack in `Object pop(Stack stack);`
- Lösungsansatz
 - Semantiken: *Call-by-Value*, *Call-by-Result*, *Call-by-Value-Result*
 - Behandlung von Referenzen
 - Zeiger dereferenzieren: Objektkopie erzeugen
 - Remote-Referenzen



Serialisierung & Deserialisierung in Java

- Primitive Datentypen
 - Hilfsklasse `java.nio.ByteBuffer`

```
public abstract class ByteBuffer [...] {
    public static ByteBuffer allocate(int capacity);
    public static ByteBuffer wrap(byte[] array);
    public byte[] array();
    public ByteBuffer put<Datentyp>(<Datentyp> value);
    public <Datentyp> get<Datentyp>();
    [...]
}
```

 - `allocate()` neues (leeres) Byte-Array anlegen
 - `wrap()` bestehendes Byte-Array verwenden
 - `array()` Byte-Array zurückgeben
 - `put*()`, `get*()` Daten einfügen bzw. lesen
 - Beispiel: {S,Des}erialisierung eines `double`-Werts


```
double d = 0.47;
ByteBuffer buffer1 = ByteBuffer.allocate(Double.SIZE / 8);
buffer1.putDouble(d);
byte[] byteArray = buffer1.array();
ByteBuffer buffer2 = ByteBuffer.wrap(byteArray);
double d2 = buffer2.getDouble();
```



Serialisierung & Deserialisierung in Java

■ Objekte

- Objekt ⇔ Stream: `java.io.Object{Out,In}putStream`

```
public class ObjectOutputStream [...] {
    public ObjectOutputStream(OutputStream out);
    public void writeObject(Object obj); // Objekt
    [...]                               // serialisieren
}

public class ObjectInputStream [...] {
    public ObjectInputStream(InputStream in);
    public Object readObject(); // Objekt deserialisieren
    [...]
}
```

- Stream ⇔ Byte-Array: `java.io.ByteArray{Out,In}putStream`

```
public class ByteArrayOutputStream extends OutputStream {
    public byte[] toByteArray(); // Rueckgabe des Byte-Array
    [...]
}

public class ByteArrayInputStream extends InputStream {
    public ByteArrayInputStream(byte buf []);
    [...]
}
```



Serialisierung & Deserialisierung in Java

■ Schnittstellen

- Automatisierte {S,Des}erialisierung: `java.io.Serializable`
 - Muss von jedem Objekt implementiert werden, das von einem `Object{Out,In}putStream` serialisiert bzw. deserialisiert werden soll
 - Marker-Schnittstelle → keine zu implementierenden Methoden

⇒ {S,Des}erialisierung wird vom `Object{Out,In}putStream` übernommen

- Manuelle {S,Des}erialisierung: `java.io.Externalizable`
 - Klassenspezifische {S,Des}erialisierung

```
public interface Externalizable extends Serializable {
    void writeExternal(ObjectOutput out);
    void readExternal(ObjectInput in);
}
```

- `writeExternal()` Objekt serialisieren
- `readExternal()` Objekt deserialisieren

⇒ {S,Des}erialisierung wird vom Objekt selbst übernommen



Serialisierung & Deserialisierung in Java

■ Geschützte Attribute

- Einige Attribute einer Klasse sollen nicht serialisiert werden
 - Sicherheitsaspekte
 - Effizienzüberlegungen
- Einige Objekte können nicht serialisiert & deserialisiert werden, da sich ihr Zustand nicht so ohne weiteres wiederherstellen lässt
 - `FileInputStream`
 - `Socket`, `ServerSocket`
 - `Thread`

⇒ Schlüsselwort `transient`

- Mit `transient` gekennzeichnete Attribute werden bei der automatischen {S,Des}erialisierung vom `Object{Out,In}putStream` ignoriert
- Beispiel

```
public class TransientExample {
    private transient Thread t = new Thread();
}
```



Überblick

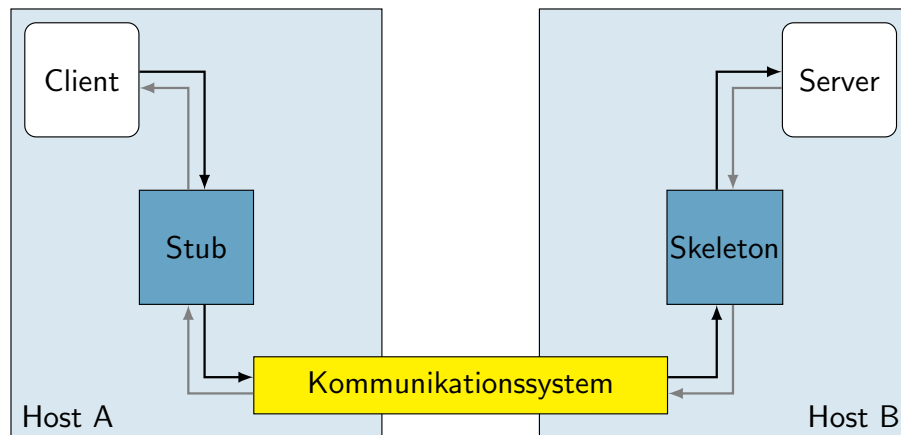
Java RMI

Java Remote Method Invocation
Marshalling und Unmarshalling
Aufgabe 1



Übungsaufgabe 1

- Programmieren mit Java RMI
- Implementierung der Kommunikationsschicht



Programmieren mit Java RMI

- Beispielanwendung: Schwarzes Brett

```
public interface VSBoard {
    public void post(VSBoardMessage message);
    public VSBoardMessage[] get(int n);
    public void listen(VSBoardListener listener);
}
```

```
public interface VSBoardListener {
    public void newMessage(VSBoardMessage message);
}
```

- `post()` Hinzufügen von neuen Botschaften
- `get()` Abfragen der neuesten Botschaften
- `listen()` Automatische Benachrichtigung über neue Botschaften

- Verteilung mittels Java RMI

- Server
 - Bereitstellung der Anwendung als Remote-Objekt
 - Registrierung des Diensts bei einer Registry
- Client
 - Zugriff auf den Dienst über Fernaufrufe
 - Interaktion mit dem Nutzer per Kommandozeile



Implementierung der Kommunikationsschicht

- Übertragung von Datenpaketen

```
public class VSConnection {
    public void sendChunk(byte[] chunk);
    public byte[] receiveChunk();
}
```

- Übermittlung von Daten über eine TCP-Verbindung
- Senden und Empfangen von Byte-Arrays beliebiger Länge

- Übertragung von Objekten

```
public class VSObjectConnection {
    public void sendObject(Serializable object);
    public Serializable receiveObject();
}
```

- Senden und Empfangen von beliebigen Objekten
- Marshalling und Unmarshalling von Nachrichten



Optimierte {S,Des}erialisierung

- Ziel

Minimierung der über das Netzwerk zu übertragenden Daten

- Ausgangspunkt

- Analyse der vom `ObjectOutputStream` erzeugten Daten
- Beispielklasse

```
public class VSTestMessage implements Serializable {
    private int integer;
    private String string;
    private Object[] objects;
}
```

- Reduzierung der benötigten Datenmenge

- Anwendung der Schnittstelle `Externalizable`
- Manuelle Implementierung der {S,Des}erialisierungsmethoden

- Hinweis: Ausgabe eines Byte-Array als Zeichenkette in Eclipse

```
byte[] chunk = [...];
System.out.println(new String(chunk, "COMPOUND_TEXT"));
```

