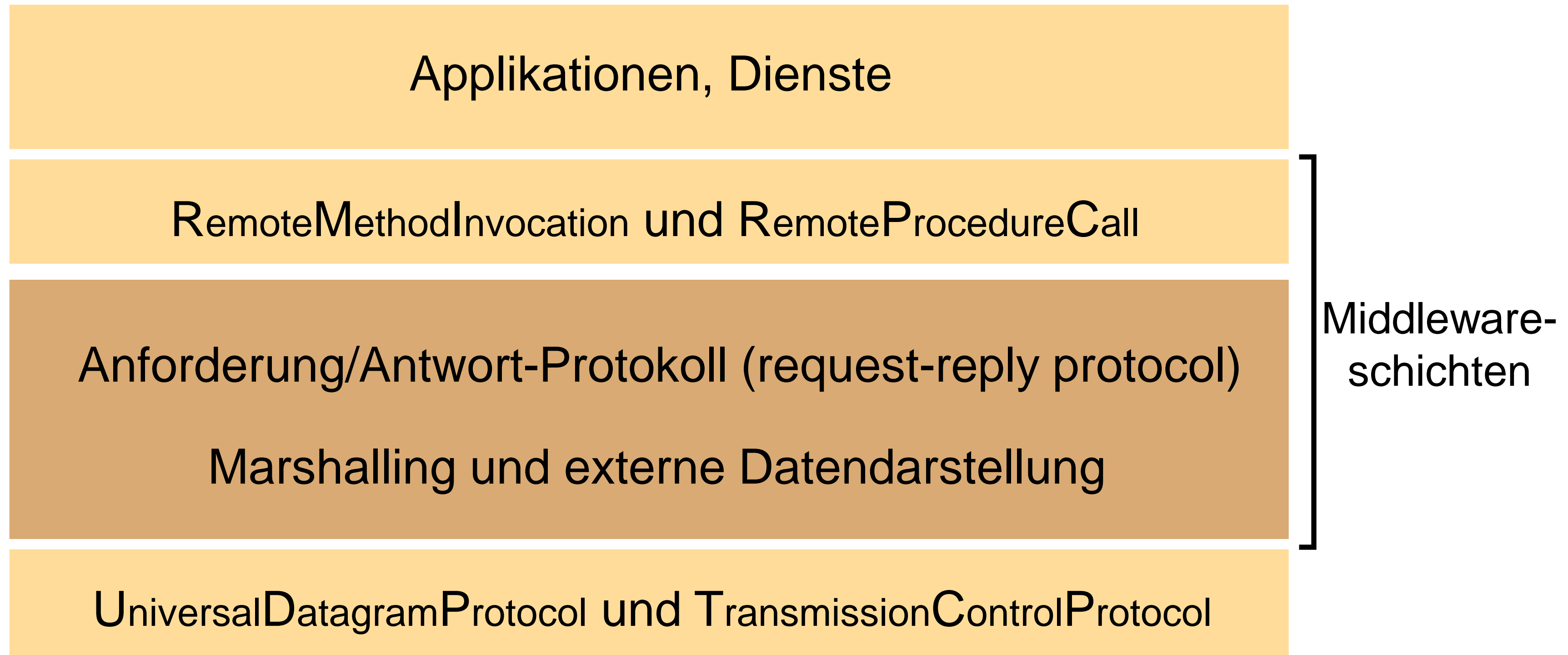
Verteilte Systeme

Interprozesskommunikation & Entfernter Aufruf

Interprozesskommunikation



ACHTUNG: In der „Zweitliteratur" wird RMI oft mit Java RMI gleichgesetzt!
Korrekt ist: Java-RMI ist eine konkrete Realisierung des RMI-Konzeptes.

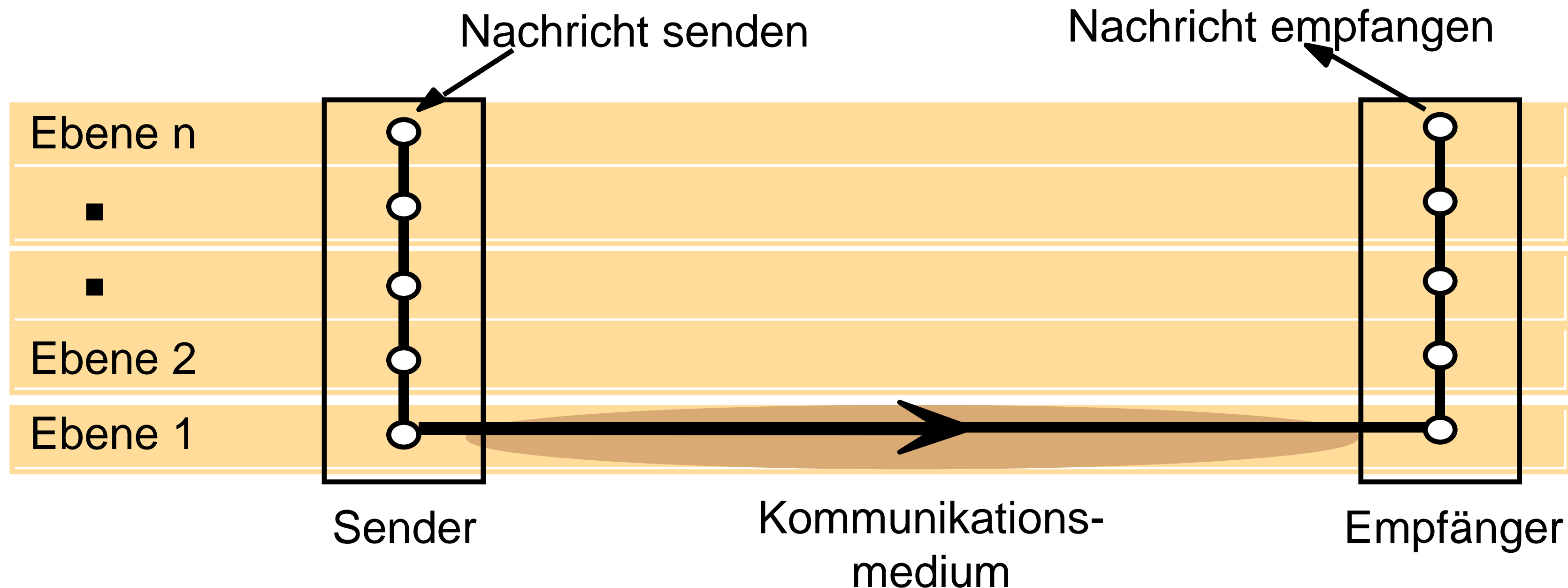
Interprozesskommunikation

- Anwendungsprogramme laufen in Prozessen ab.
- Ein Prozess ist ein **Objekt des Betriebssystems**, durch das Anwendungen sicheren Zugriff auf die Ressourcen des Computers erhalten. Einzelne Prozesse sind deshalb gegeneinander isoliert. (Aufgabe des Betriebssystems)
- Damit zwei Prozesse Informationen austauschen können, müssen sie **Interprozesskommunikation** (*i*nter*p*rocess-*c*ommunication, IPC) verwenden.
- IPC basiert auf (Speicher-/Nachrichtenbasierter) Kommunikation
  1. **gemeinsamen Speicher**: für VS nicht direkt verwendbar
  2. **Austausch von Nachrichten** (= Bytefolge) über einen Kommunikationskanal zwischen Sender und Empfänger.

Interprozesskommunikation

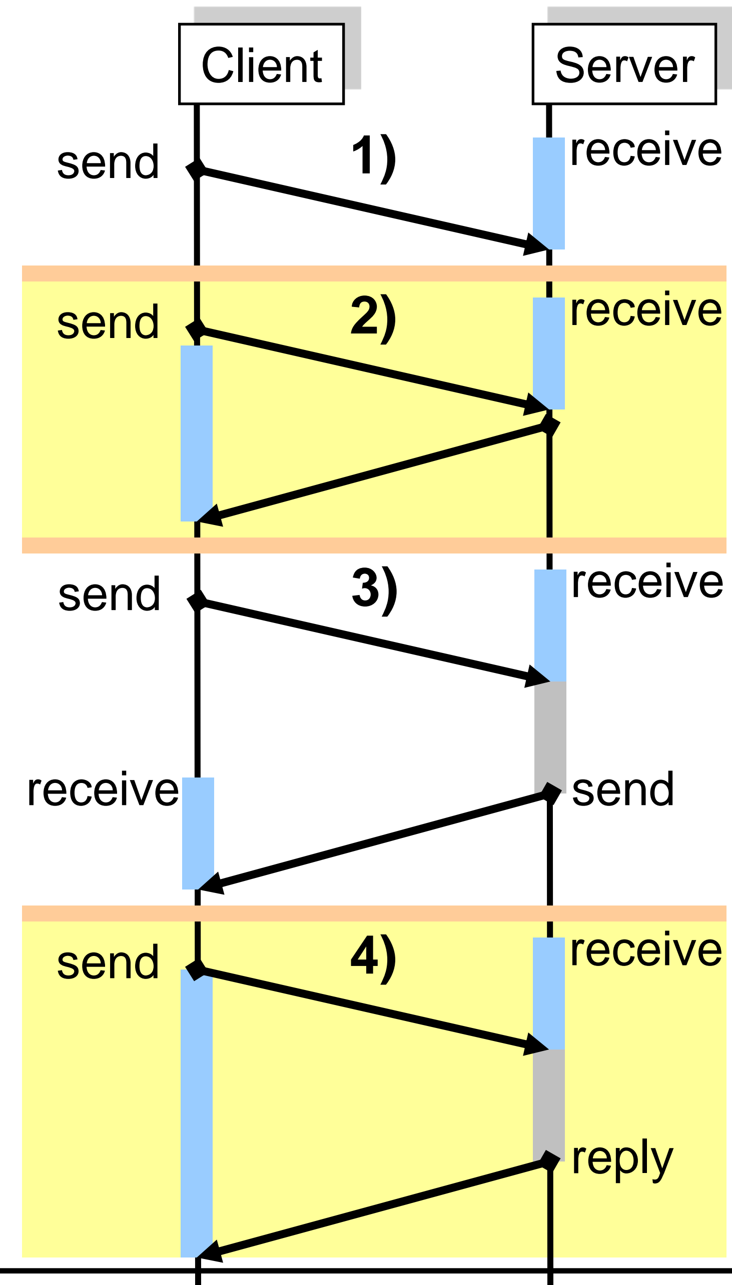
- Betriebssystem: Koordiniert IPC innerhalb dieses BS.
- IPC **in verteilten Systemen** geschieht ausschließlich über *Nachrichten*
- Koordination der IPC durch Middleware oder/und durch Entwickler
- Hierbei sind gewisse *Normen* zu beachten, damit die Kommunikation klappt!!

- **Protokoll** := Festlegung der **Regeln** und des **algorithmischen Ablaufs** bei der Kommunikation zwischen zwei oder mehr Partnern



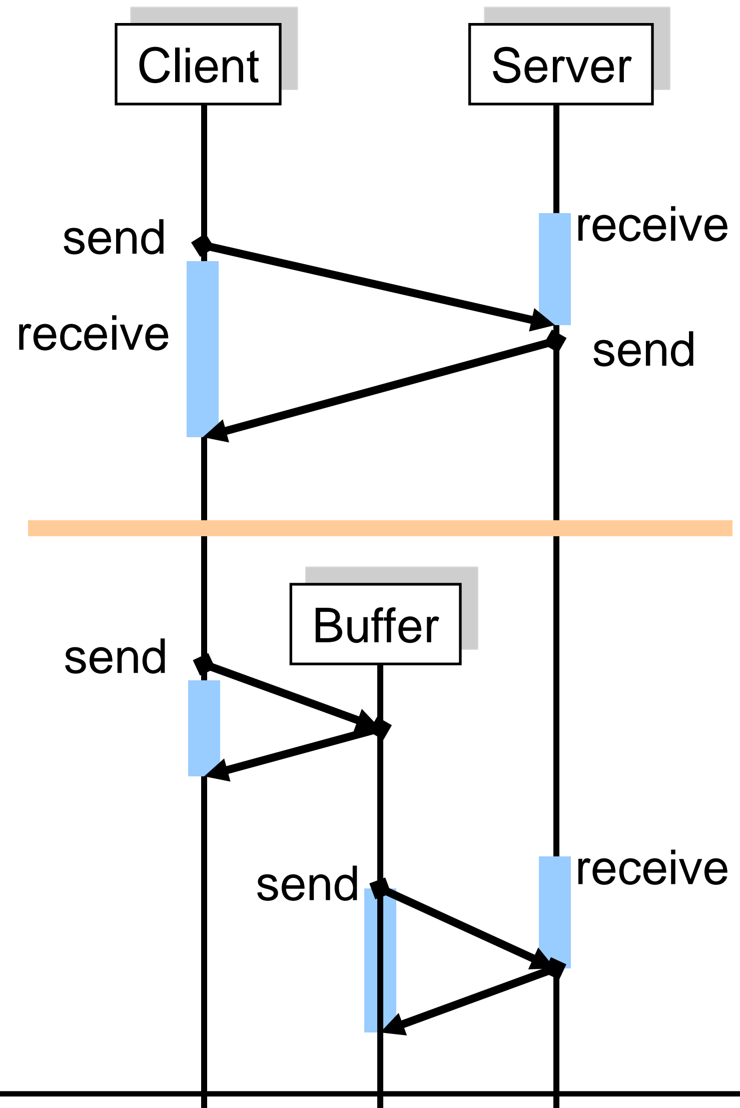
Kommunikationsmuster

	Synchronisationsgrad	
	Asynchron	**Synchron**
Mitteilung	No-wait-send Datagramm **1)**	Rendevous Stream **2)**
Auftrag	Remote Service Invocation **3)**	Remote Procedure Call **4)**

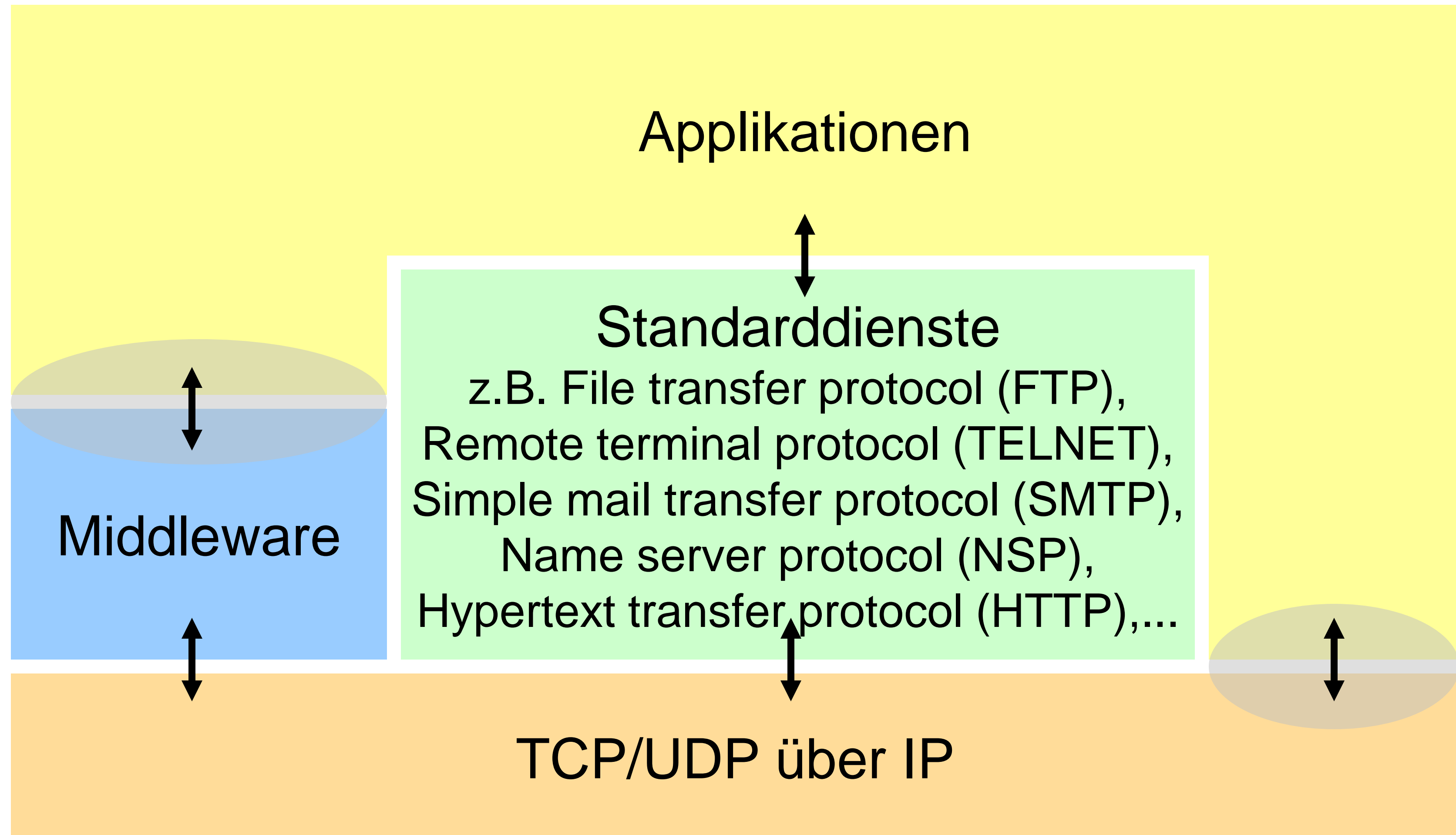


Dualität der Kommunikationsmuster

- Synchrone Kommunikation mittels asynchroner Kommunikation
  - Explizites Warten auf Acknowledgement im Sender direkt (!) nach dem *send*-Befehl (*receive*-Befehl ist i.allg. blockierend)
  - Explizites Versenden des Acknowledgements durch den Empfänger direkt nach dem *receive*-Befehl.
- Asynchrone Kommunikation mittels synchroner Kommunikation
  - Erzeugung eines zusätzlichem Prozesses, dem *Pufferprozess*
  - Zwischenpufferung aller Nachrichten im Pufferprozess



Implementierung verteilter Anwendungen



Direkte Netzprogrammierung & Middleware

Direkte Netzprogrammierung

(Grundbausteine der VS-Programmierung)

- Direkte Kontrolle aller Transportparameter
- größere Flexibilität bei der Entwicklung neuer Protokolle
- Kann in vielen Fällen bessere Performance bringen
- Zu lösende Probleme:
  - Datenrepräsentation
  - Signalisierung
  - Semantik
  - Fehlerbehandlung
- Typisch: in Anwendungsprotokollen

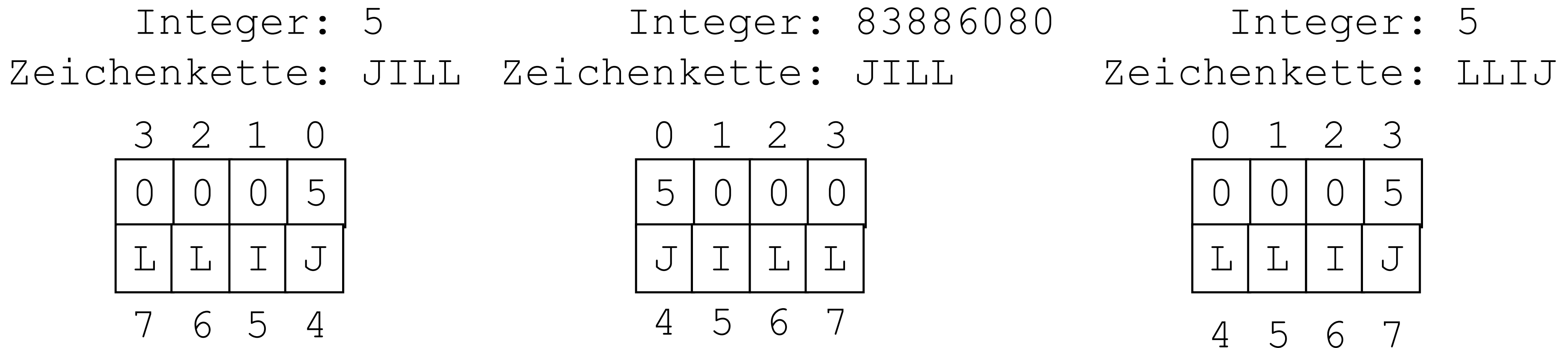
Middleware

(Höhere Sprache der VS-Programmierung)

- Sehr bequemer Weg zur Entwicklung von Anwendungen
- Datenrepräsentation, Objektlokalisierung, Transaktionsdienst, Fehlerbehandlung, Sicherheitsdienst, etc. muss nicht eingekauft werden.
- Overhead, da allgemein ausgelegt.

Problem der übertragbaren Daten

- Unterschiedliche Darstellungen des Wortes „JILL“ und der Zahl 5 am Beispiel Little-Endian (Intel) / Big-Endian (SPARC)



Integer werden durch die unterschiedliche Byteordnung gedreht, aber Zeichenketten nicht.

Externe Datendarstellung

- Es gibt eine Reihe bekannter Ansätze für ein gemeinsames Netzdatenformat.
- Idee:
  - Definiere eine **Menge von abstrakten Datentypen** und eine **Kodierung** (ein genaues Bit-Format) für jeden dieser Typen
  - Stelle **Werkzeuge** zur Verfügung, die die abstrakten **Datentypen in** Datentypen der verwendeten **Programmiersprache** übersetzen
  - Stelle **Werkzeuge** zur Verfügung, die die Datentypen der verwendeten **Programmiersprache in** die abstrakten **Datentypen** und damit in das kodierte Format übersetzen
  - *Senden (Marshalling)*: wenn ein bestimmter Datentyp übertragen werden soll, rufe die Kodierfunktion auf und übertrage das Ergebnis
  - *Empfangen (Un-Marshalling)*: dekodiere den Bit-String und erzeuge eine neue lokale Repräsentation des empfangenen Typs

Existierende Externe Datendarstellung

Sender und Empfänger sind sich über die Reihenfolge und die Typen der Datenelemente in einer Nachricht einig

- ISO: ASN.1 (*A*bstract *S*yntax *N*otation)
- Sun ONC (*O*pen *N*etwork *C*omputing)-RPC: XDR (e*X*ternal *D*ata *R*epresentation)
- OSF (*O*pen *S*ystem *F*oundation)-RPC: IDL (Interface Definition Language)
- *Corba:* IDL und CDR (*C*ommon *D*ata *R*epresentation): CDR bildet IDL-Datentypen in Bytefolgen ab.

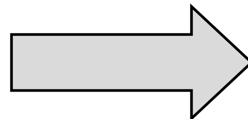
Vollständige Informationen über Reihenfolge und die Typen der Datenelemente sind in einer Nachricht enthalten

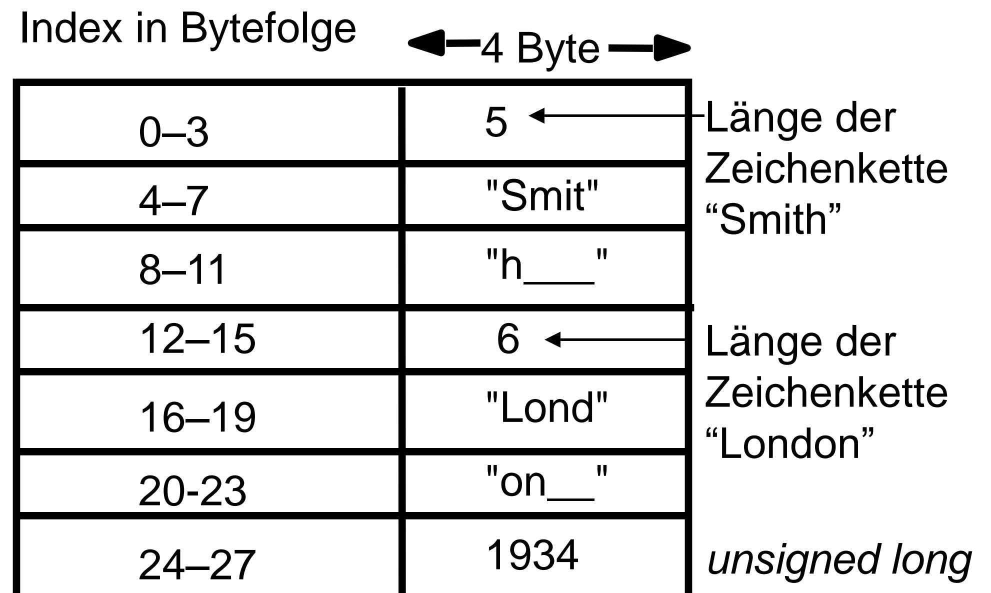
- Java: Objektserialisierung, d.h. Abflachung eines (oder mehrerer) Objektes zu einem seriellen Format inkl. Informationen über die Klassen. Deserialisierung ist die Wiederherstellung eines Objektes ohne Vorwissen über die Typen der Objekte.
- Google Protocol Buffers: Sprachneutrale Serialisierung komplexer Datenstrukturen.

Beispiel: Common Data Representation (CDR)

Typ	Darstellung
Sequence	Länge gefolgt von Elementen in der angegebenen Reihenfolge
String	Länge gefolgt von Zeichen in der angegebenen Reihenfolge
Array	Array-Elemente in der angegebenen Reihenfolge
Struct	Die Reihenfolge der Deklarationen der Komponenten
Enumerated	Unsigned Long

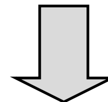
```
Struct Person{
    string name;
    string place;
    long year;
};
```





Beispiel: Java-Objektserialisierung

```
Public class Person implements Serializable{
   private String name;
   private String place;
   private int year;
   public Person(String aName, String aPlace, int aYear) {
      name = aName;
      place = aPlace;
      year = aYear;    }
   // gefolgt von Methoden für den Zugriff auf die Instanzvariablen
}
```



```
Person p = new Person(„Smith“,“London“,1934);
```

Person	8-Byte Versionsnummer		h0	Klassenname, Versionsnummer
3	int year	java.lang.String name:	java.lang.String place:	Nummer, Typ und Name der Instanzvariablen
1934	5 Smith	6 London	h1	Werte der Instanzvariablen

Das echte serialisierte Format enthält zusätzliche Typkennzeichner;
h0 und h1 sind Handles, also Verweise auf serialiserte Objekte

Fazit

- Zuerst die **schlechte Nachricht**: das sieht alles ziemlich kompliziert aus, und das ist ist es auch! Als Socket-Programmierer muss man sich um all diese Dinge selbst kümmern.

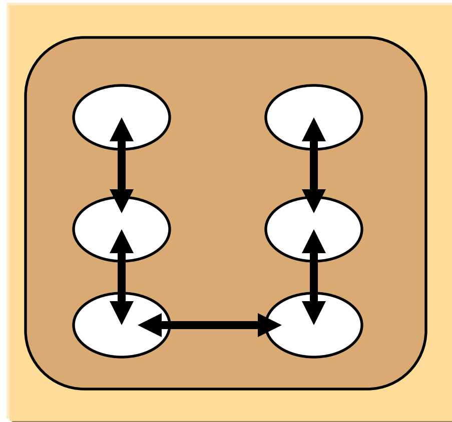
- Die **gute Nachricht**: die Aufgabe einer Middleware ist es, genau diese komplizierten Mechanismen automatisch zu erledigen. Der Anwendungsprogrammierer sieht davon nichts mehr.

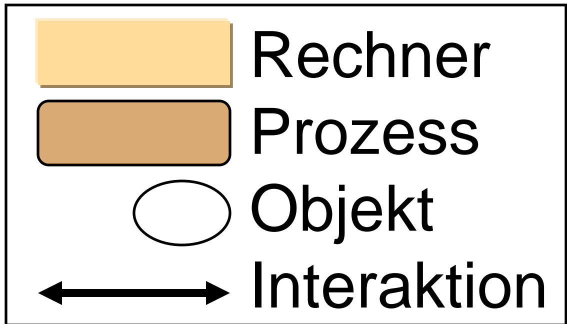
Verteiltes Objektsystem

Lokal

versus

Entfernt





Das verteilte Objektmodell

- **Verteiltes System**: Interagierende Objekte sind auf mehr als einen Prozess verteilt
- **Wichtige Begriffe** (Auswahl, vereinfacht):
  – Entfernte Objektreferenz: die „Adresse“/eindeutige Identität eines Objekts im *ganzen* verteilten System
  – Entfernte Schnittstellen: die Schnittstelle eines entfernten Objekts
  – Ereignisse/Aktionen: Ereignisse/Aktionen von Objekten können Prozessgrenzen überschreiten
  – Exceptions/Ausnahmen: verteilte Ausführung des Systems erweitert das Spektrum möglicher Fehler
  – Garbage Collection: Freigabe nicht mehr benutzten Speichers wird im verteilten System schwieriger

Entfernte Objektreferenz

- Über Raum und Zeit **garantiert eindeutig!**
- Bestehen aus
  - *Internetadresse*: gibt den Rechner an
  - *Port-Nummer* und *Zeit*: Identifizieren eindeutig den Prozess
  - *Objektnummer*: Identifiziert das Objekt
  - *Schnittstelle*: beschreibt die entfernte Schnittstelle des Objekts
- Werden erzeugt von einem speziellen Modul - dem entfernten Referenzmodul - wenn eine lokale Referenz als Argument an einen anderen Prozess übergeben wird und in dem korrespondierenden Proxy gespeichert.

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internetadresse	Port-Nummer	Zeit	Objektnummer	Schnittstelle des entfernten Objektes

Achtung: Diese Art der Referenz erlaubt kein Verschieben des Objektes in einen anderen Prozess!

Schnittstellen entfernter Objekte

- Die entfernte Schnittstelle gibt an, wie auf entfernte Objekte zugegriffen wird (Signatur der Methodenmenge).
- Ihre Beschreibung enthält
  - Den **Namen der Schnittstelle**
  - Möglicherweise **Datentypdefinitionen**
  - Die **Signatur** aller entfernt verfügbaren Methoden, bestehend aus
    - Dem Methodennamen
    - Ihrer Ein- und Ausgabeparameter
    - Ihrem Rückgabewert
- Jede Middleware besitzt eine eigene Sprache, um solche Schnittstellen zu beschreiben.

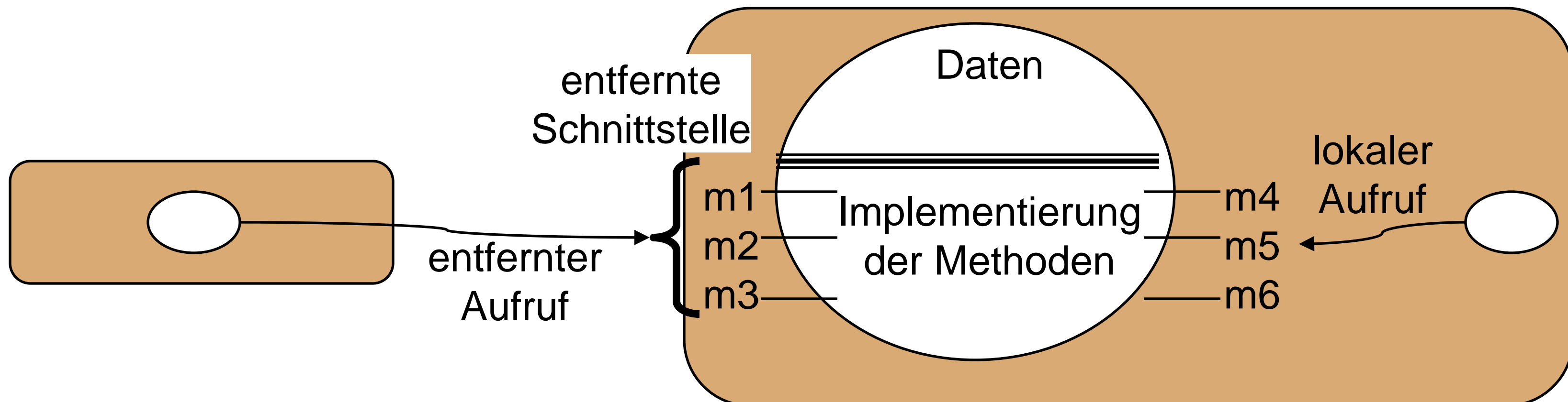
Schnittstellen entfernter Objekte: Beispiel Java RMI

Signatur: Definition der Methoden

entfernte Schnittstelle

```
public interface IPersonList extends Remote {
        public void addPerson(Person p) throws RemoteException;
        public void getPerson(String name, Person p) throws RemoteException;
        public int getNumber() throws RemoteException;
};
```

Remote Exception: Kommunikationsfehler



Entwurfsprobleme

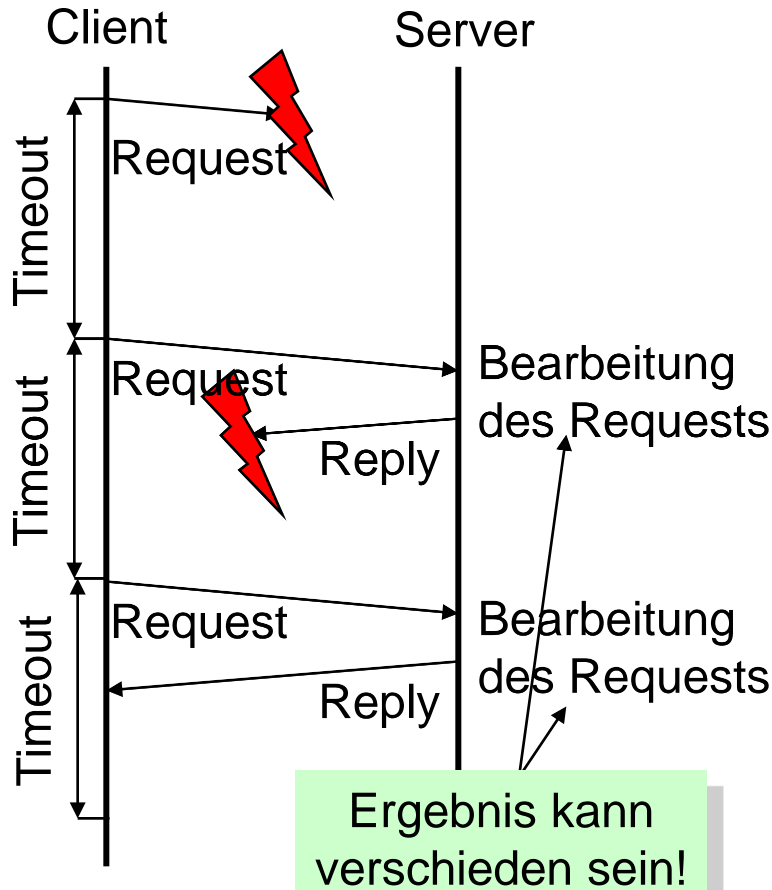
- Lokale Aufrufe werden genau einmal ausgeführt. Dies kann für entfernte Aufrufe nicht immer der Fall sein. Was sind die Alternativen ?

  Führt zur Definition einer Fehlersemantik

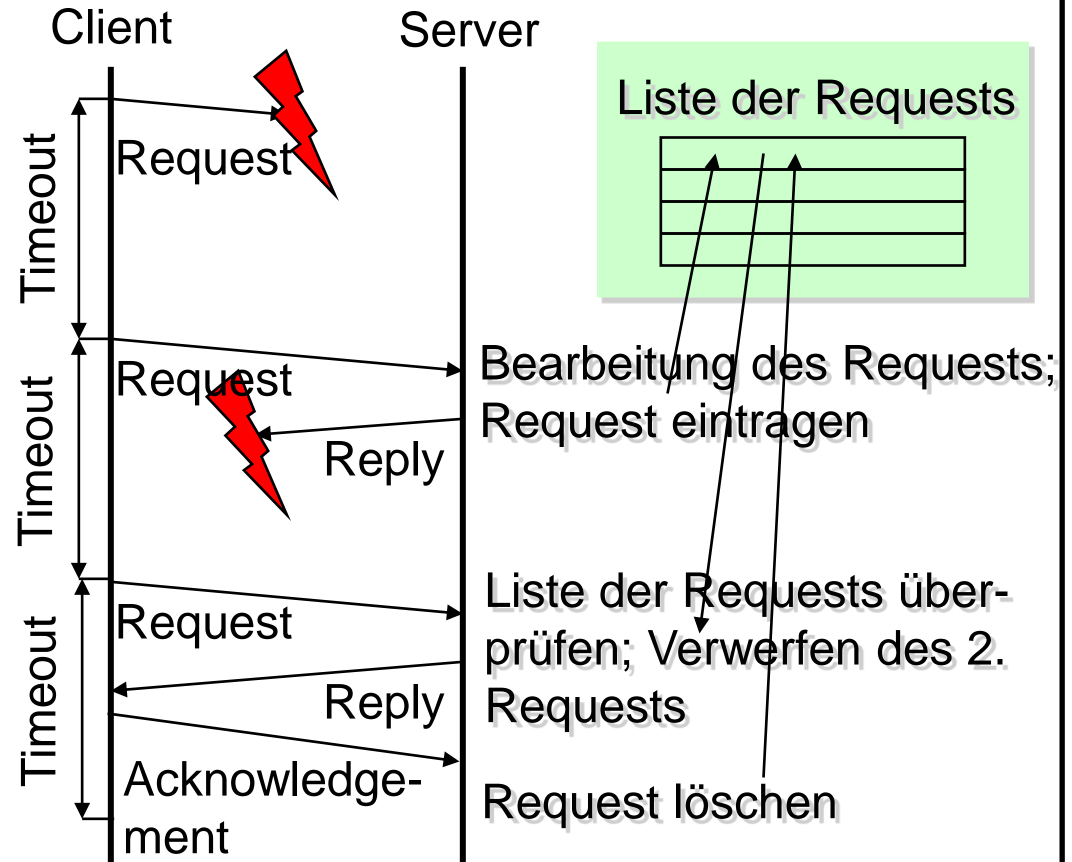
- Was ist der Transparenzgrad der entfernten Aufrufe ?
  Was ist gegeben, was muß der Programmierer selber sicherstellen ?

Fehlersemantik

at least once Semantik



at most once Semantik



Fehlersemantik

Fehlersemantik	Wiederholung einer Anfrage	Filterung von Duplikaten	
Maybe	Nein	Nein	
At-least-once	Ja	Nein	Wiederholte Ausführung
At-most-once	Ja	Ja	Wiederholte Antwort
Exactly-once	Nein	Nein	

Fehlersemantik

Fehlersemantik	Fehlerfreier Ablauf	Nachrichten-verluste	Ausfall des Servers
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-least-once	Ausführung: 1 Ergebnis: 1	Ausführung: >= 1 Ergebnis: >= 1	Ausführung: >=0 Ergebnis: >=0
At-most-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0
Exactly-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

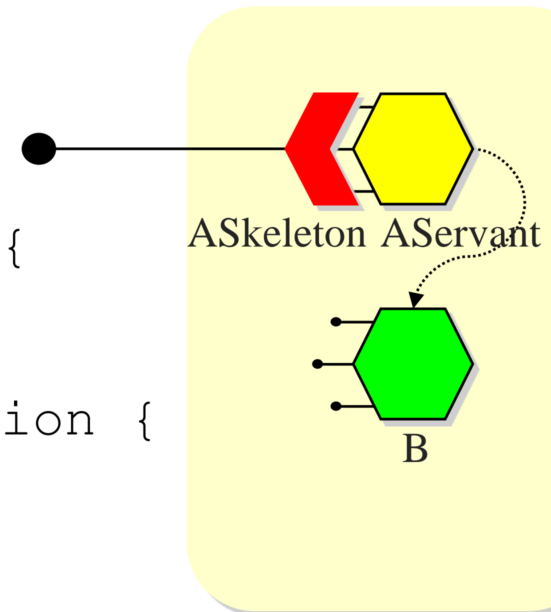
Referenz- und Kopiersemantik

- Entfernte Methodenaufrufe sollten **Parameterübergabe-Semantik** der verwendeten Programmiersprache respektieren:
  - In Java Übergabe von Werten per Kopie, Übergabe von Objekten per Referenz
  - In C++ freie Wahl der Übergabeart
- **Probleme**:
  - Entfernte Referenzen auf Werte prinzipiell nicht möglich
  - Entfernte Referenzen auf Objekte nur möglich, wenn entsprechende Stubs und Skeletons existieren
  - Empfänger benötigt Implementierungsklasse für erhaltenes Objekt (Kopiersemantik) bzw. Stub (Referenzsemantik)
- **Traditionelle Programmiermodelle brechen in Vert. Systemen**

Beispiel für Objektübergabe

```
import B;
public interface A {
extends Remote {
   public void setB(B b) throws RemoteException;
   public B getB() throws RemoteException;}}

public class AServant
  extends UnicastRemoteObject
  implements A {
   private B b;
   public void setB(B b)
               throws RemoteException {
      this.b = b;
   }
   public B getB() throws RemoteException {
      return this.b; }}
```



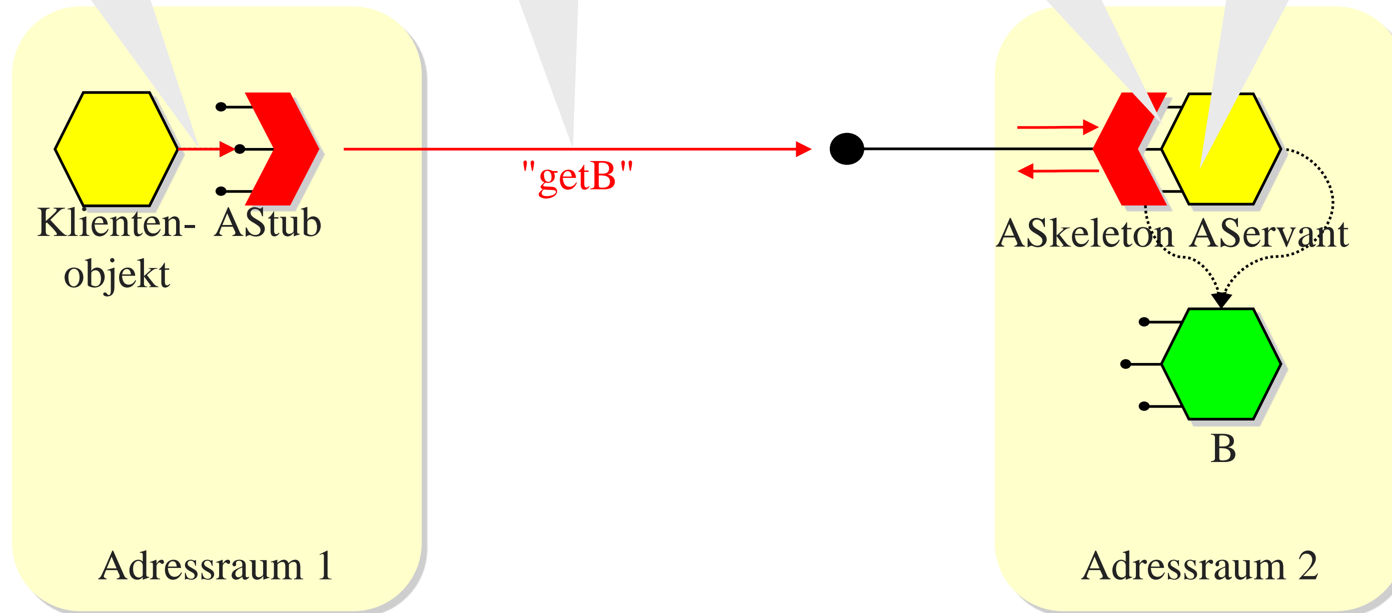
Beispiel für Objektübergabe: Kopiersemantik

1. Clientobjekt hält Referenz auf Instanz von A, ruft darauf Methode getB() auf.

2. Stub übermittelt Methodenaufruf an Skeleton

3. Skeleton delegiert Methodenaufruf an Servant

4. Servant übergibt Referenz auf Instanz von B an Skeleton



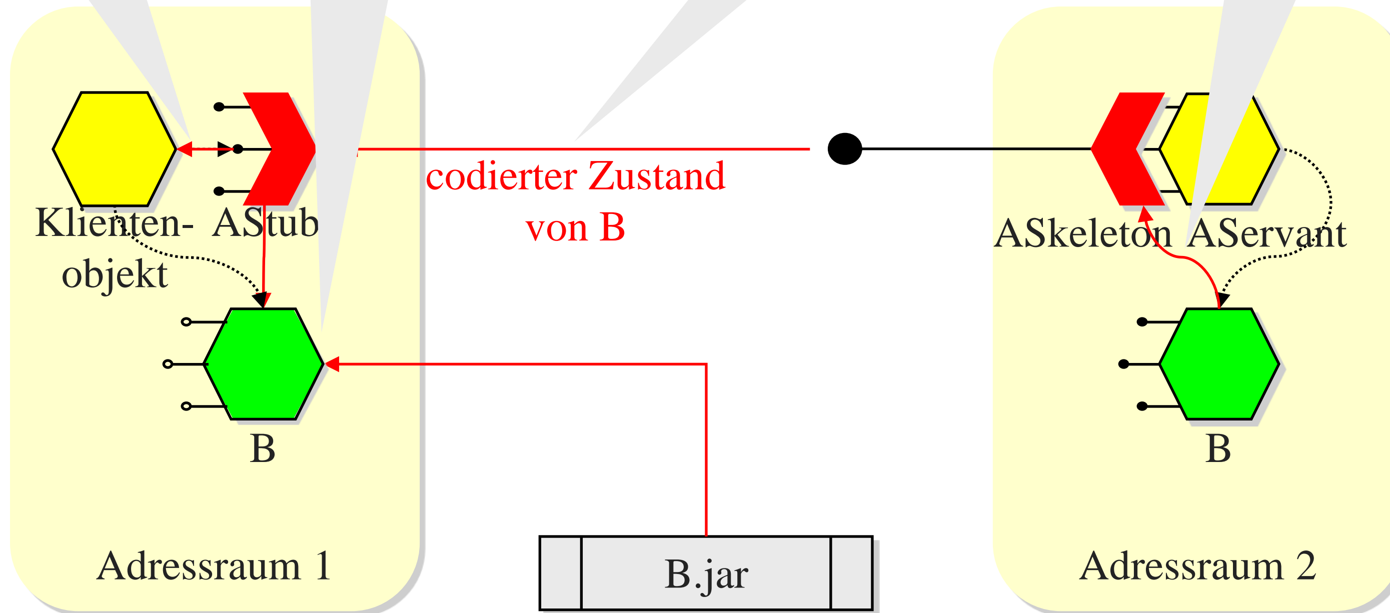
Beispiel für Objektübergabe: Kopiersemantik

8. Stub übergibt Verweis auf neue Instanz an Aufrufer

7. Stub lädt Klasse B, dekodiert Zustand und erzeugt damit neue Instanz von B

6. Kodierter Zustand wird an Stub übertragen

5. Skeleton kodiert Zustand von Instanz gemäß Wire Protocol



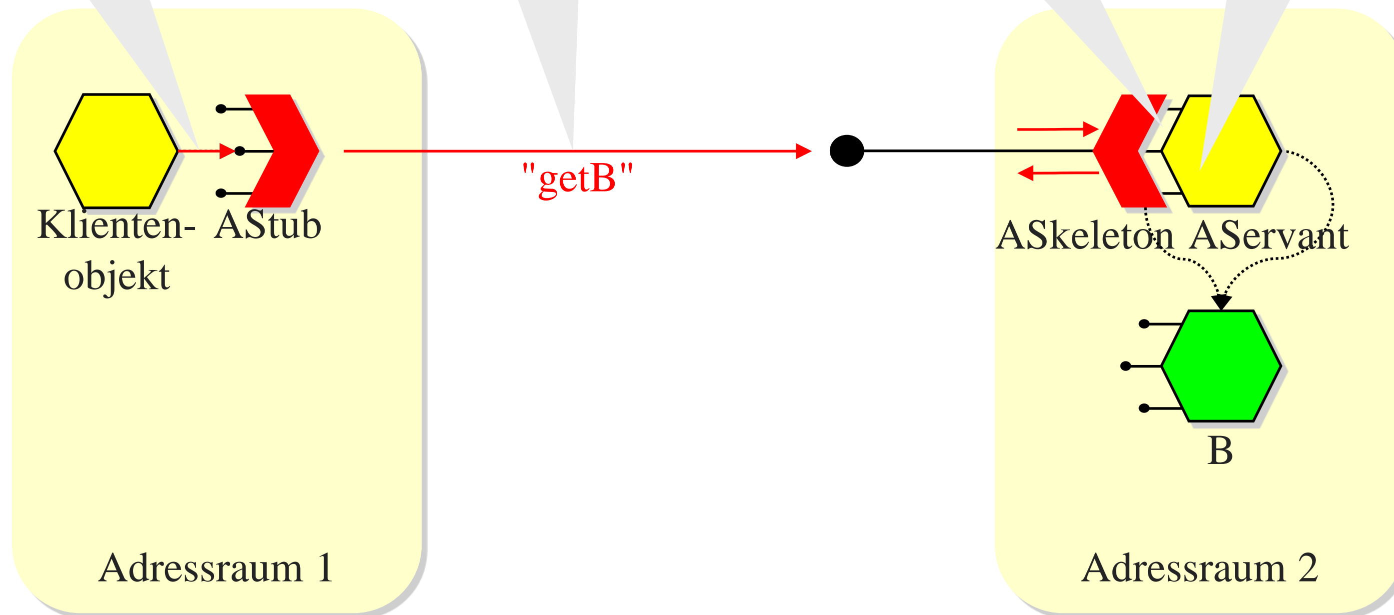
Beispiel für Objektübergabe: Referenzsemantik

1. Clientobjekt hält Referenz auf Instanz von A, ruft darauf Methode getB() auf.

2. Stub übermittelt Methodenaufruf an Skeleton

3. Skeleton delegiert Methodenaufruf an Servant

4. Servant übergibt Referenz auf Instanz von B an Skeleton



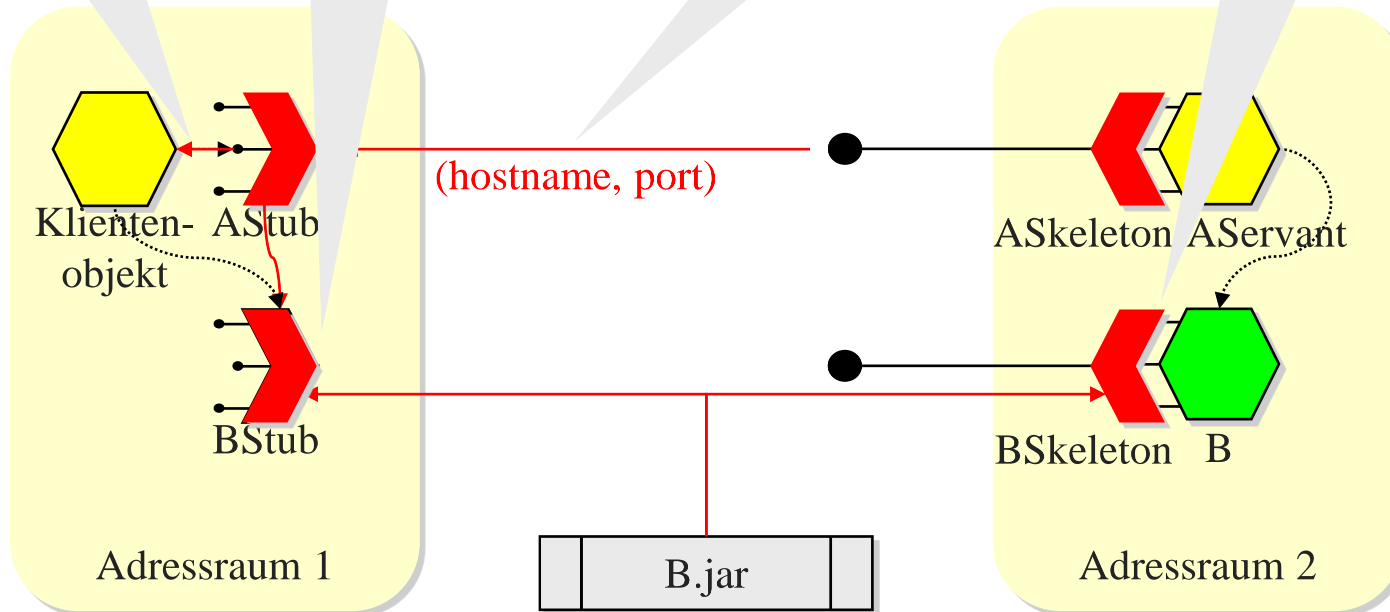
Beispiel für Objektübergabe: Referenzsemantik

8. A-Stub übergibt Verweis auf B-Stub an Aufrufer

7. A-Stub erzeugt neuen B-Stub, der Netzwerkadresse von B-Skeleton enthält

6. A-Skeleton sendet Netzwerkadresse von B-Skeleton an A-Stub

5. A-Skeleton erzeugt neues Skeleton für B, falls nicht bereits vorhanden



Weitere Aspekte der Objektübergabe

- Festlegung der **Übergabesemantik** i.A. durch **Typ** des formalen Parameters:
  - *Referenzen* und *keine Referenzen* sind zunächst alles **Werte**! Die Übergabesemantik regelt die Art der Interpretation.
  - **Referenzübergabe**, wenn formaler Parameter bestimmtes Interface (in Java z.B. java.rmi.Remote) implementiert
  - **Wertübergabe** sonst

- Bei Wertübergabe **Komplikationen** möglich:
  - Wenn übergebenes Objekt direkt oder indirekt andere Objekte referenziert, müssen diese ebenfalls übergeben werden (mit welcher Übergabesemantik?)
  - Sharing von Objekten muss auf der Clientseite rekonstruiert werden
  - Wenn übergebenes Objekt echter Untertyp des formalen Parameters ist, ist u.U. Downcast erforderlich

Transparenz des RMI

- ✓ **Zugriffstransparenz** ermöglicht den Zugriff auf lokale und entfernte Ressourcen unter Verwendung identischer Operationen.
  Ist realisiert: die Operationen sind identisch, die Syntax evtl. unterschiedlich.
- ✓ **Positionstransparenz** (Ortstransparenz) erlaubt den Zugriff auf die Ressourcen, ohne dass man ihre Position/ihren Ort kennen muss.
  Ist realisiert.
- ✘ **Nebenläufigkeitstransparenz** erlaubt, dass mehrere Prozesse gleichzeitig mit denselben gemeinsam genutzten Ressourcen arbeiten, ohne sich gegenseitig zu stören.
  Ist nicht realisiert.
- ✓ **Replikationstransparenz** erlaubt, dass mehrere Instanzen von Ressourcen verwendet werden, um die Zuverlässigkeit und die Leistung zu verbessern, ohne dass die Benutzer oder Applikationsprogrammierer wissen, dass Repliken verwendet werden.
  Ist manchmal realisiert.

Transparenz des RMI

- ✓ **Fehlertransparenz** erlaubt das Verbergen von Fehlern, so dass Benutzer und Applikationsprogrammierer ihre Aufgaben erledigen können, auch wenn Hardware- oder Softwarekomponenten ausgefallen sind.
  Ist teilweise realisiert (siehe Fehlersemantik)
- ✓ **Mobilitätstransparenz** erlaubt das Verschieben von Ressourcen und Clients innerhalb eines Systems, ohne dass die Arbeit von Benutzern oder Programmen dadurch beeinträchtigt wird.
  Mittels Namensdienst realisiert.
- ✘ **Leistungstransparenz** erlaubt, dass das System neu konfiguriert wird, um die Leistung zu verbessern, wenn die Last variiert.
  Ist nicht realisiert.
- ✓ **Skalierungstransparenz** erlaubt, dass sich System und Applikationen vergrößern, ohne dass die Systemstruktur oder die Applikationsalgorithmen geändert werden müssen.
  Ist durch die Objektorientiertheit bereits gegeben.

Implementierung eines RMI

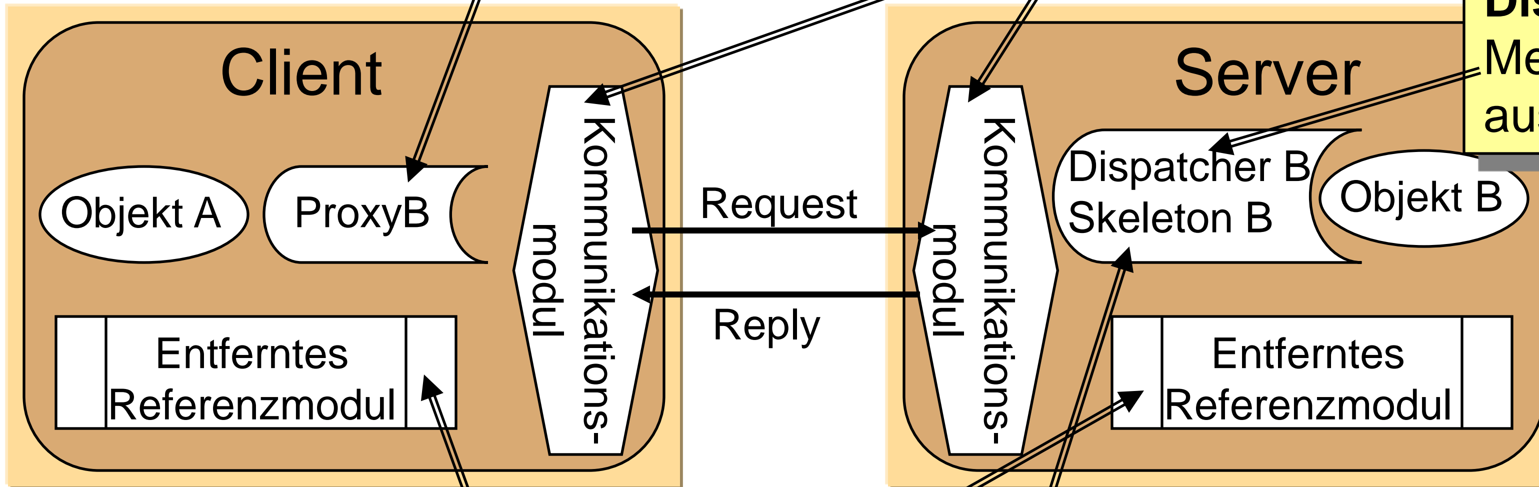
- **Kommunikationsmodul**: zuständig für das Request-/Reply-Protokoll
- **Entferntes Referenzmodul**: Übersetzt zwischen entfernten und lokalen Objektreferenzen; besitzt meist eine entfernte Objekt-Tabelle, in der diese Zuordnung eingetragen wird. Beim ersten Aufruf wird die entfernte Objektreferenz von diesem Modul erzeugt.

Rolle von Proxy und Skeleton

Proxy: macht RMI transparent für Client. Klasse implementiert entfernte Schnittstelle. Marshals Request und unmarshals Reply. Leitet Request weiter.

Ausführung des Request/Reply Protokolls

Dispatcher: wählt Methode im Skeleton aus.



Übersetzung zwischen lokalen und entfernten Objektreferenzen

Skeleton: implementiert Methoden der entfernten Schnittstelle. Unmarshals Request und Marshals Reply. Ruft Methode in entferntem Objekt auf.

Implementierung

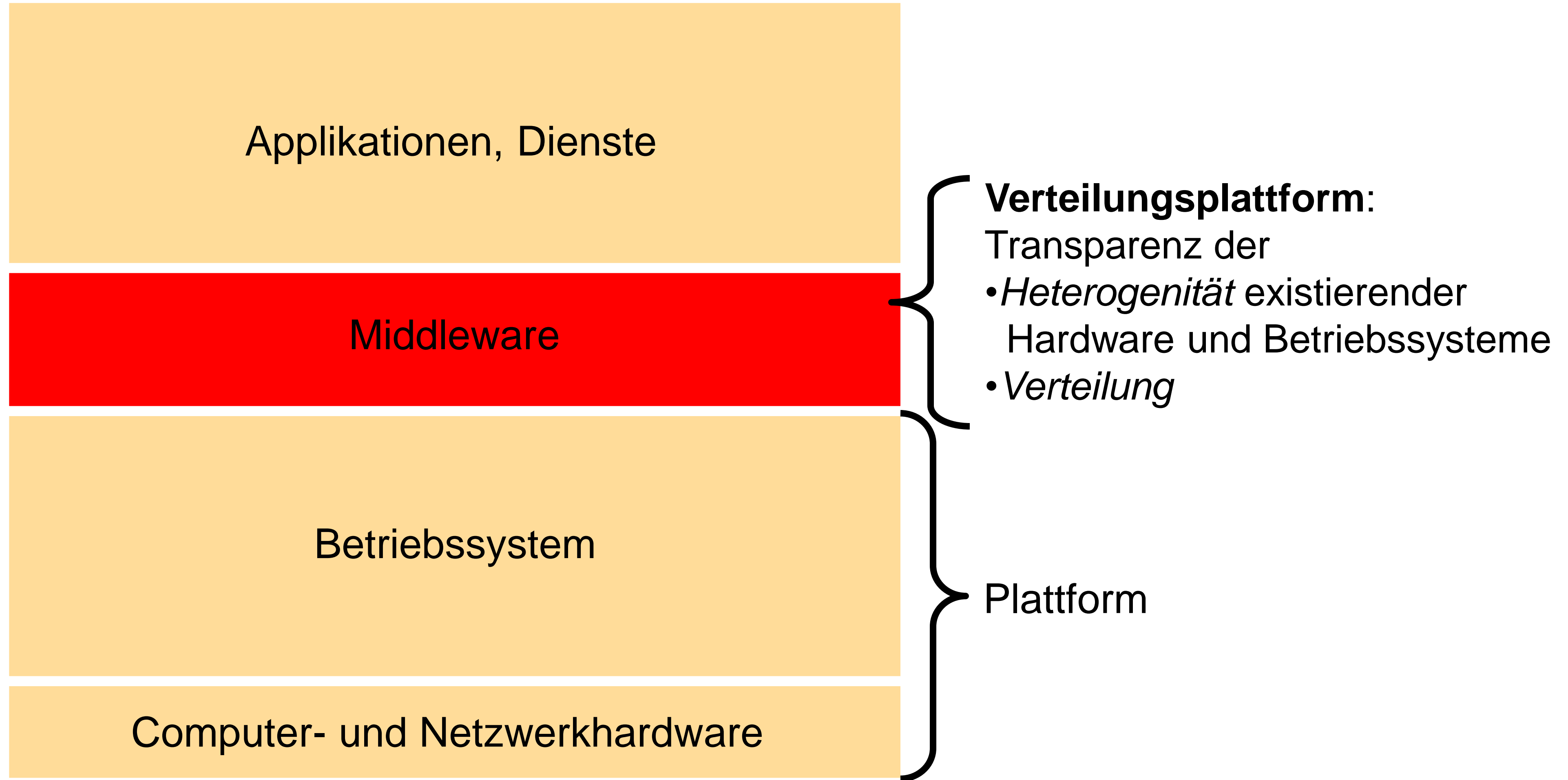
RMI-Software: Softwareschicht zwischen Objekten und Kommunikations- und entfernten Referenzmodulen.

- Schnittstellen-Compiler erzeugt automatisch Klassen für Dispatcher, Skeleton und Proxy (geschieht automatisch in Java RMI)
- Server-Programm enthält Klassen für Dispatcher, Skeleton und alle davon unterstützten entfernten Objekte (Servant-Klassen) sowie einen Initialisierungsabschnitt
- Client-Programm enthält Klassen für Proxies aller entfernten Objekte.
- Factory-Methode: Ersetzen Konstruktoren in den entfernten Schnittstellen, d.h. sind normale Methoden, die entfernte Objekte erzeugen können.

Implementierung

- **Binder**: Namensdienst, der Clients Objektreferenzen vermitteln kann
- **Server-Thread**: Um zu verhindern, dass ein entfernter Aufruf einen anderen Aufruf verzögert, weisen Server der Ausführung *jeden* entfernten Aufrufs einen eignen Thread zu!
- **Aktivierung**: Erzeugung einer Instanz und Initialisierung der Instanzvariablen.
- **Persistenter Objektspeicher**: Verwaltet persistente Objekte, also Objekte, die zwischen Aktivierungen weiterbestehen.
- **Verteilte Garbage Collection**: Stellt sicher, dass in einem verteilten System garbage collection durchgeführt wird. Problem: Referenzen, die nur in Nachrichten vorhanden sind.

Middleware



Arten von Middleware

- **Generisch**
  - Remote Procedure Call (RPC) — Entfernter Prozeduraufruf
    Remote Method Invocation (**RMI**) — Entfernten Methodenaufruf
  - Object Request Broker — Objektzugriff übers Netz
  - Message Passing — Send/Receive–Kommunikation
  - Virtual Shared Memory — Zugriff auf virtuell gemeinsamen Speicher
- **Speziell**
  - Dateitransfer — Fernzugriff auf gemeinsame Dateien
  - Datenbankzugriff — Datenzugriff auf entfernte DB
  - Transaktionsverarbeitung — Koordination verteilter Transaktionen
  - Groupware / Workflow — Zusammenarbeit verteilter Gruppen
  - Directories / AAA Services — Organisation arbeitsteiliger Prozesse

Beispiel: Java-RMI

- Definiert ein Rahmenwerk für die Kommunikation von Java-Objekten unabhängig von ihrem Ort
- Eine reine Java-Lösung
- Alle entfernten Objekte müssen eine entfernte Schnittstelle besitzen
- Die Generierung von Stubs und Skeletons wird seit Java 1.5 „versteckt".
- JDK stellt eine Implementierung des Naming-Service zur Verfügung: die *RMIregistry* (arbeitet nur lokal am Server).
- Ein RMI-Dämon erlaubt einen flexible (on-demand)-Instanziierung von Objekten.

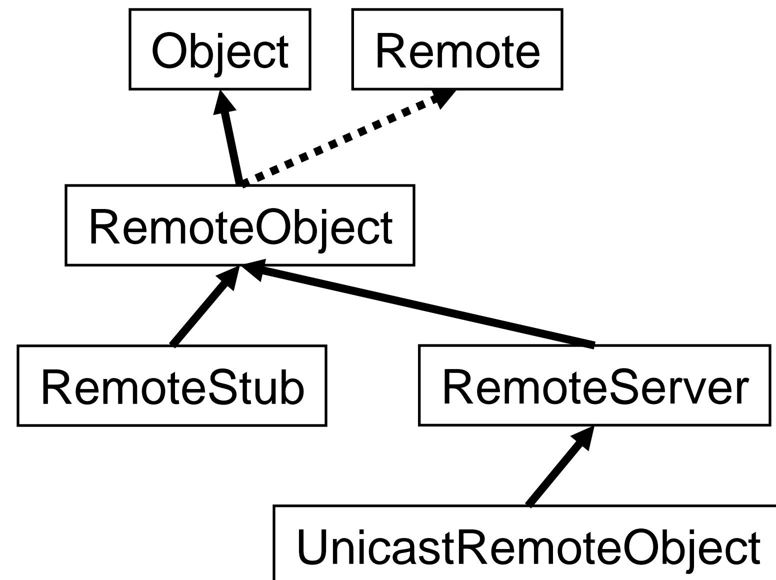
Java-RMI: Das entfernte Objekt

- Um den von der Schnittstelle „versprochenen“ Dienst zu erbringen, muss es ein entferntes Objekt geben, das die Methoden der Schnittstelle implementiert.
- Gewöhnlich erweitert es die Klasse UnicastRemoteObject was aus dem Objekt einen nichtreplizierten Server macht, der über TCP kommuniziert.

Datum.java | DatumImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

class DatumImpl extends UnicastR
    public DatumImpl() throws Re
        super();
        System.out.println("DatumI
    }
    public Date getDate() throws
        System.out.println("DatumI
        return new Date();
```



Java-RMI: Erzeugen einer Anwendung

Dieses Beispiel wurde unter Java 6 entwickelt und getestet.

- Remote Interface `Datum` definiert `getDate`-Methode die das Datum und die Uhrzeit auf dem Server zum Zeitpunkt des Aufrufes zurückgibt
- Server Implementierung `DatumImpl` implementiert das Interface und initialisiert den RMI-Server
- Client `DatumClient` ruft `getDate` entfernt auf und gibt das Ergebnis aus
- Client und Server werden in dem Beispiel auf dem selben Host ausgeführt

1. Definiere die entfernte Schnittstelle

```
1  package vs.rmiexample;
2  import java.rmi.Remote;
3  import java.rmi.RemoteException;
4  import java.util.Date;
5  /**
6   * Remote Interface
7   */
8  public interface Datum extends Remote {
9      /**
10      * Gibt eine Date-Objekt zurück das die aktuelle Zeit beim Erstellen repräsentiert
11      */
12     public Date getDate() throws RemoteException;
13 }
```

Java-RMI: Erzeugen einer Anwendung

2. Implementiere die entfernte Schnittstelle durch ein entferntes Objekt

```
1  package vs.rmiexample;
2  import java.rmi.RemoteException;import java.rmi.registry.LocateRegistry;import java.util.Date;
3  import java.rmi.registry.Registry;import java.rmi.server.UnicastRemoteObject;
4  // RMI-Server. Implementiert das Remote-Interface.
5  public class DatumImpl extends UnicastRemoteObject implements Datum  {
6      public DatumImpl() throws RemoteException {
7          super();
8      }
9      @Override
10     public Date getDate() throws RemoteException {
11         Date date = new Date();
12         System.out.println("Die Methode getDate wurde vom Client aufgerufen. Aktuelles Datum:" + date);
13         return date;
14     }
15     public static void main(String[] args){
16         try {
17             Datum datum=new DatumImpl();
18             // Registrieren des Datum-Objektes
19             Registry registry = LocateRegistry.getRegistry();
20             registry.rebind("datum", datum);
21             System.out.println("Server initialisiert!");
22         } catch (RemoteException e) {
23             System.err.println("Fehler bei der initialisierung des Datum-Servers:");
24             e.printStackTrace();
25         }
26     }
27 }
```

Java-RMI: Erzeugen einer Anwendung

3. Schreibe einen Client

```
1  package vs.rmiexample;
2  import java.rmi.NotBoundException;import java.rmi.RemoteException;import java.util.Date;
3  import java.rmi.registry.LocateRegistry;import java.rmi.registry.Registry;
4  //RMI-Server
5  public class DatumClient {
6      private Datum datum;
7      //Sucht in der RMI Registry nach der 'datum'-Instanz
8      public DatumClient() throws RemoteException, NotBoundException  {
9              // Die Registry wird auf dem Host rmiexample.inet.cpt.haw-hamburg.de auf dem Standardport 1099 gesucht.
10             Registry registry = LocateRegistry.getRegistry("rmiexample.inet.cpt.haw-hamburg.de");
11             datum = (Datum) registry.lookup("datum");
12     }
13     //Gibt das Datum des gerufenen Servers auf der Konsole aus.
14      public void printServerDate() throws RemoteException{
15             Date date =  datum.getDate();
16             System.out.println(date);
17     }
18     public static void main(String[] args) {
19         try {
20             DatumClient datumClient = new DatumClient();
21             datumClient.printServerDate();
22         } catch (NotBoundException e) {
23             System.err.println("Das Remote Objekt konnte in der Registry nicht gefunden werden.");
24             e.printStackTrace();
25         } catch (RemoteException e) {
26             System.err.println("Fehler bei der Kommunikation mit dem RMI-Server");
27             e.printStackTrace();
28         }
29     }
30 }
```

Java-RMI: Erzeugen einer Anwendung

4. Kompiliere Client und Server: Ab Java 5 werden Stubs und Skeletons automatisch erstellt. In älteren Versionen muss `rmic` verwendet werden
5. Starte den Namensdienst mit `rmiregistry`
6. Starte den Server

```
java
-classpath bin/
-Djava.rmi.server.codebase=[Pfad zur codebase]
vs.rmiexample.DatumImpl &
```

   Die Codebase gibt an, wo die Server-Class-Dateien im Netzwerk verfügbar sind. In diesem lokalen Beispiel werden die Daten aus dem Filesystem geladen. Beispiel:
   `file:///home/user/vs/RmiServer/bin/`
7. Starte den Client

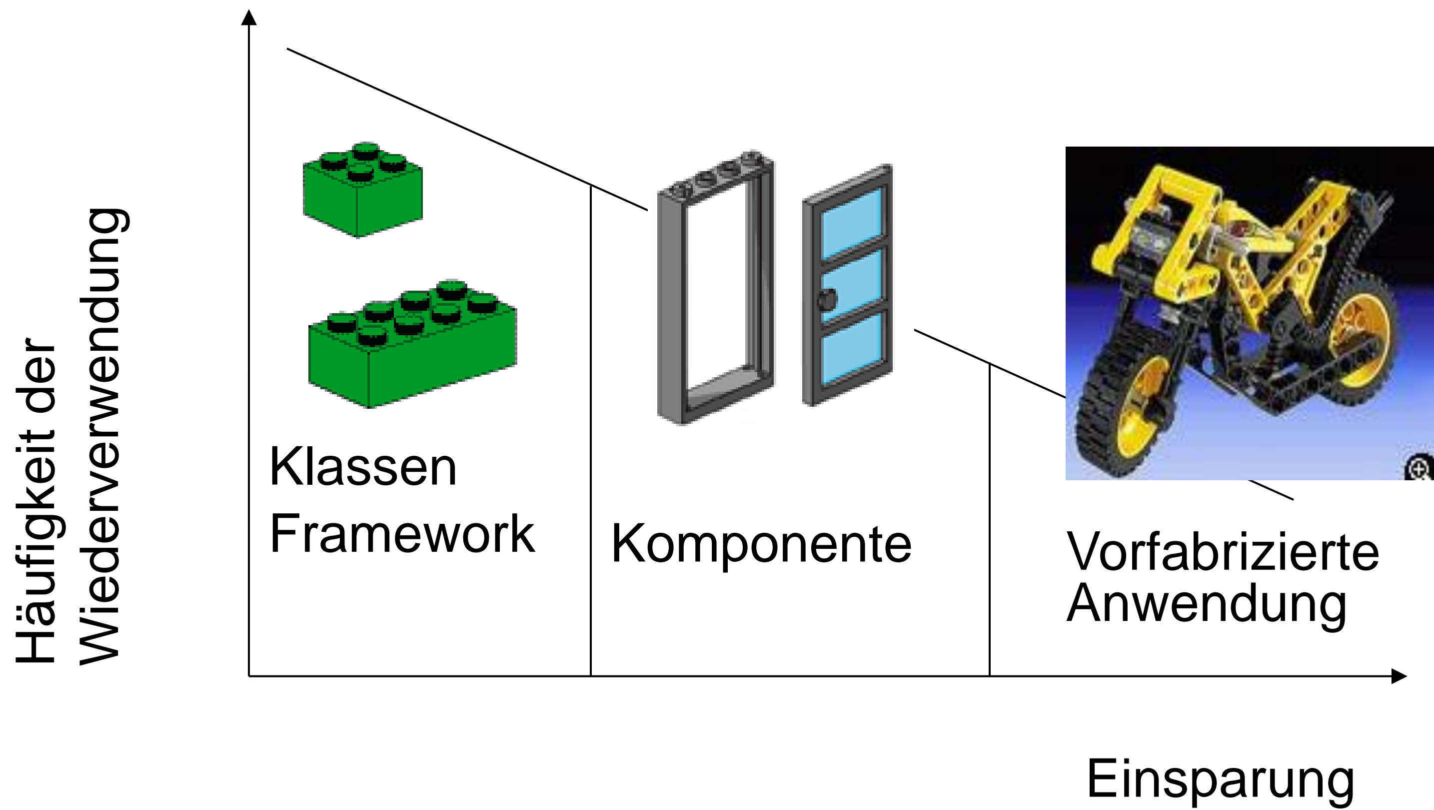
Zusammenfassung: Java RMI

- Java RMI ist Java spezifisch, wobei z.B. mit JNI eine grundsätzliche Möglichkeit besteht, ein Nicht-Java System als Server-Komponente via Java RMI zu verwenden.
- Java Anbindung direkter/eleganter als bei vergleichbaren generische RMI Lösungen (z.B. CORBA)
- Stellt integrierte Dienste wie etwa Distributed Garbage Collection zur Verfügung.
- RMI: registry wird über eine URL angesprochen, d.h. „einfache Struktur".

Idee: Komponenten

- **Komponente** = höhere Abstraktionsform von Objekten
  - Bestehen aus einem oder mehreren Objekten, welche in einen Container gepackt werden
- Komponenten **interagieren** u. **kooperieren** über verschiedene BS-plattformen, Sprachen, etc. hinweg
  - Bausteine für multitiered Anwendungen
- **Anwendungen** bestehen aus (dynamischen) Mengen interagierender Komponenten (monolithische Anwendungen aufbrechen)
- Dieses Modell hat enorme **Konsequenzen** bzgl.
  - **Entwurf** von Software („Lego-Bausteine“)
  - **Vertrieb** von Software („add-on Komponenten“, „late customizing“)
  - **Pflege** von Software (Wiederverwendung, Varianten)
  - **Funktionalität** (aktive, ggf. mobile Objekte)
  - **Marketing** (Komponenten-Markt)
- Erfordert **Standards** und **Infrastrukturservices**
  - für die Interaktion der Komponenten
  - für die Komponenten selbst (Versionskontrolle, Konfiguration)

Motivation



JEE Komponentenplattform

Die Java Enterprise Edition (JEE) ist eine Plattform für die komponentenorientierte Entwicklung von Anwendungen.

Sie besteht aus:

- Einer Spezifikation / Guidelines / Testsuite
- Java Komponenten
  - Java Beans (clientseitig)
  - Java Server Pages und Servlets
  - Enterprise Java Beans + persistence API (serverseitig)
- Verschiedene Container: Application, Web, EJB
- Java Naming and Directory Interface (JNDI)

JEE und Beans folgen dem Konzept der

„Konvention vor Konfiguration“

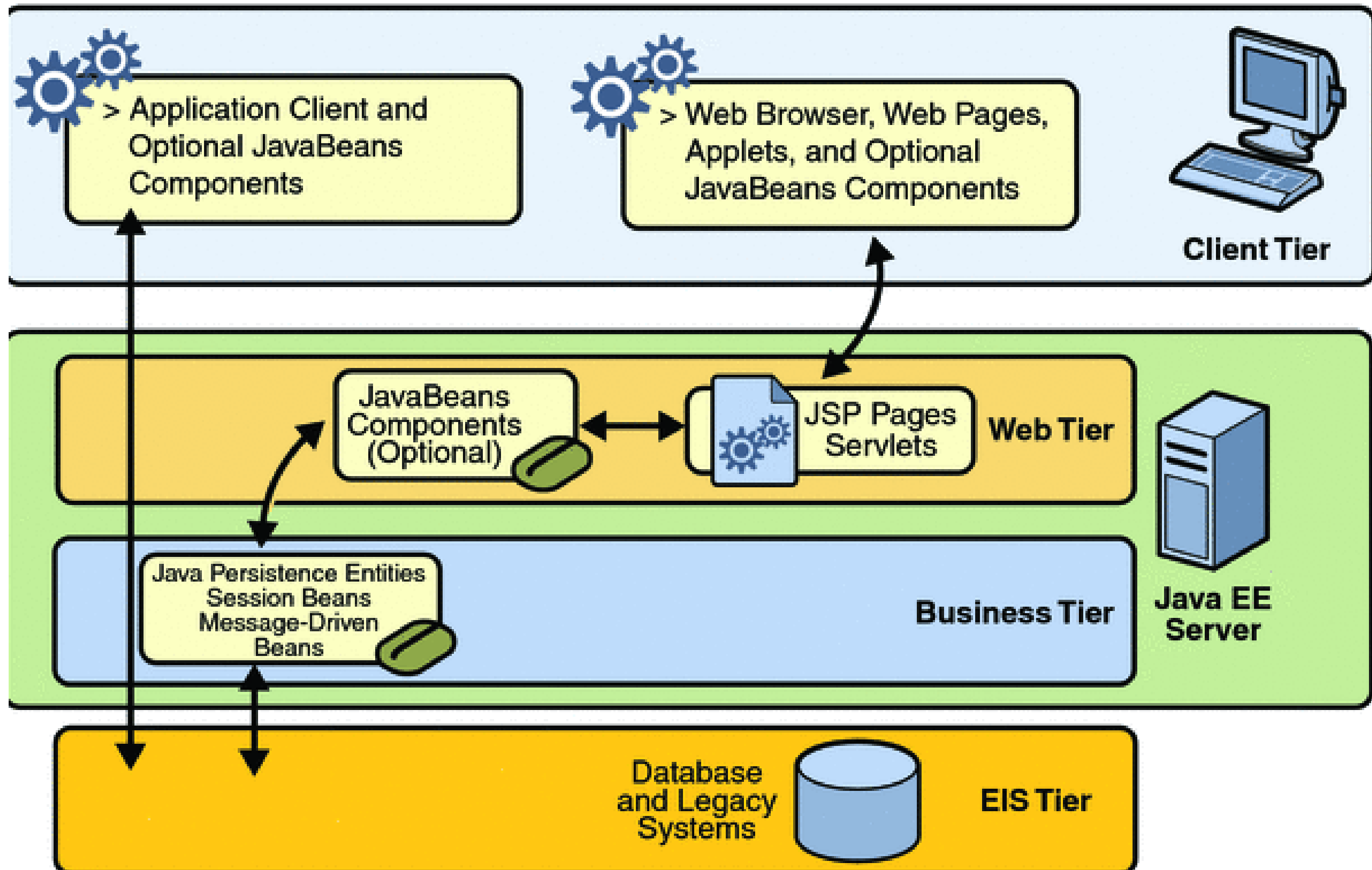
EJB Introduction

„An Enterprise JavaBeans (EJB) component, or enterprise bean, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the J2EE server.“

(J2EE Tutorial: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Overview7.html#wp86355)

- EJB 2.0 ist Bestandteil der J2EE 1.4 Spezifikation
- EJB 3.0 ist Bestandteil der J2EE 5.0 Spezifikation
- EJB 3.2 ist Bestandteil der JEE 7.0 Spezifikation
- EJB Container bildet die Laufzeitumgebung (Runtime Environment)
- Tutorial: http://docs.oracle.com/javaee/7/tutorial/doc/

JEE Mehrlagige Architektur



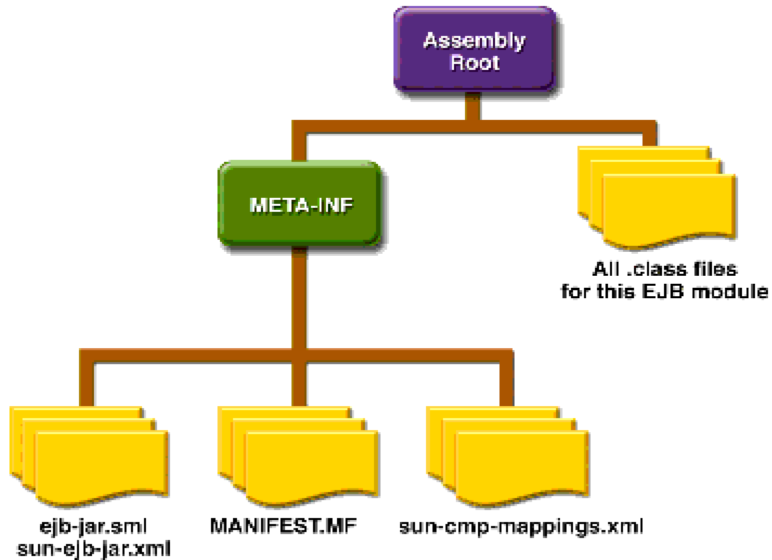
EJB Container

"Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server."

(J2EE Tutorial: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Overview3.html#wp79828)

- Bietet folgende Dienste für EJBs
  - Security Modelle
  - Unterstützung für Transaktionen
  - Naming Services (JNDI registry & lookup)
  - Initiiert und kontrolliert den Lebenszyklus der Beans
  - Datenpersistenz
  - Datenbankverbindungen
  - Ressourcen-Pooling

EJB Einsatz (Deployment)



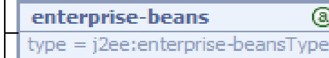
- Enterprise JAR Archiv enthält:
  - Deployment Descriptor:
    XML file (persistence type, transaction attributes…)
  - Interfaces (Remote und Home Interfaces für den Komponentenzugriff)
  - EJB classes (Implementierungen der Interfaces)
  - Helper classes (… was man sonst braucht)
- EJB-JAR Module können zusammengefasst werden in einem Enterprise Application Archive (EAR).

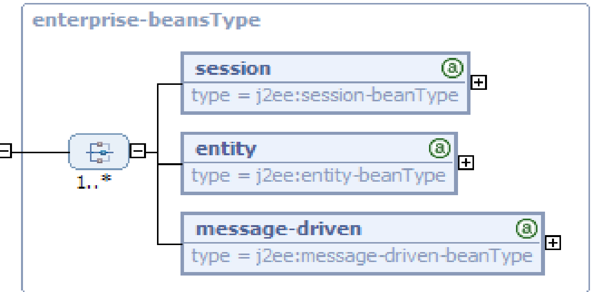
Siehe http://java.sun.com/xml/ns/j2ee/ für Details über deployment descriptors.

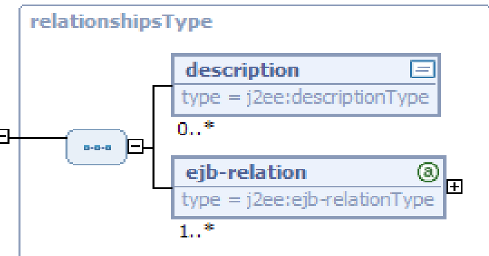
EJB-JAR

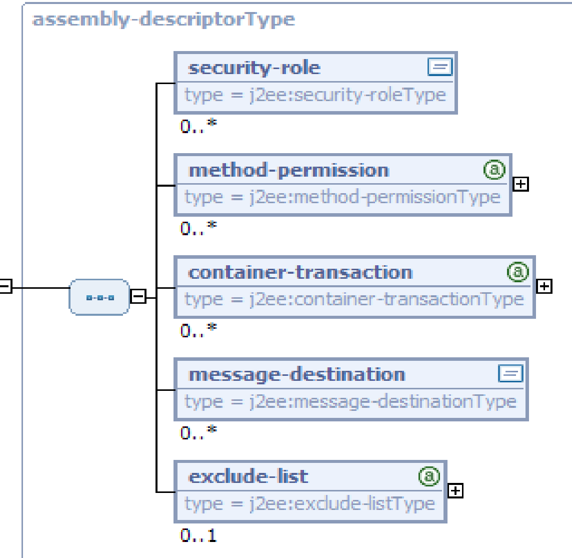
```
<ejb-jar>
 <display-name>
  MailApplicationEJB
 </display-name>
 <enterprise-beans>
  <session>
   <ejb-name>
      MailReader
   </ejb-name>
   <home>
      . . .MailReaderSessionHome
   </home>
   <remote>
      . . .MailReaderSession
   </remote>
   <ejb-class>
      . . .MailReaderSessionBean
   </ejb-class>
   <session-type>
      Stateful
   </session-type>
   <transaction-type>
      Container
   </transaction-type>
  </session>
 </enterprise-beans>
</ejb-jar>
```

ejb-jarType









EJB Typen

- Session Bean – wird für einen einzelnen Client ausgeführt
  - An die Lebenszeit einer Session gebunden
- Message-Driven Bean – reagiert auf JMS[1] Nachrichten
  - zustandslos, kommuniziert asynchron
- Java Persistence API – (JEE 1.5) standardisiert Zugriff auf objektrelationale Brücken (z.B. Hibernate)
  - API-Zugriff unmittelbar von Java Objekten (ohne Container)
  - Queries: Java Persistence Query Language / Database Query Language

[1] Java Message Service - http://java.sun.com/products/jms/

JAVA Persistence API

- Interface zu einer objekt-relationalen Abbildung (per Metadaten)
- Ermöglicht strukturierten Zugriff auf Entities (Objekte)
  - Persistenz diverser Datentypen
  - Primärschlüssel (IDs)
  - Entity-Relationen mit kaskadierten Beziehungen
  - Entity-Vererbung
- EntityManager API
  - Erzeugt, entfernt und persistiert Entities
  - Interface zur Query Language

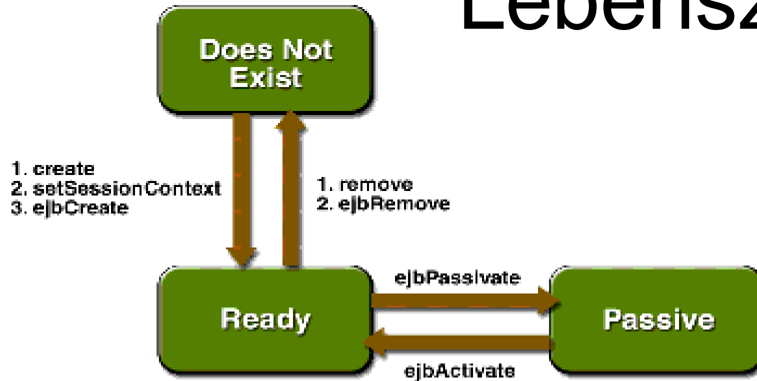
Session Beans

- Session Bean Eigenschaften:
  - Verbergen Komplexität der Business Logic
  - Nicht persistent
  - Repräsentieren eine (interaktive) Session für **einen** Client
  - Können nicht zwischen Clients geteilt werden
  - Beenden mit dem Client
- Session Bean Modi:
  - **Stateless** Session Beans:
    Variablenzustände leben nur während Methodenaufrufen
  - **Stateful** Session Beans:
    Variablenzustände bestehen während der Clientsitzung

Message-Driven Beans

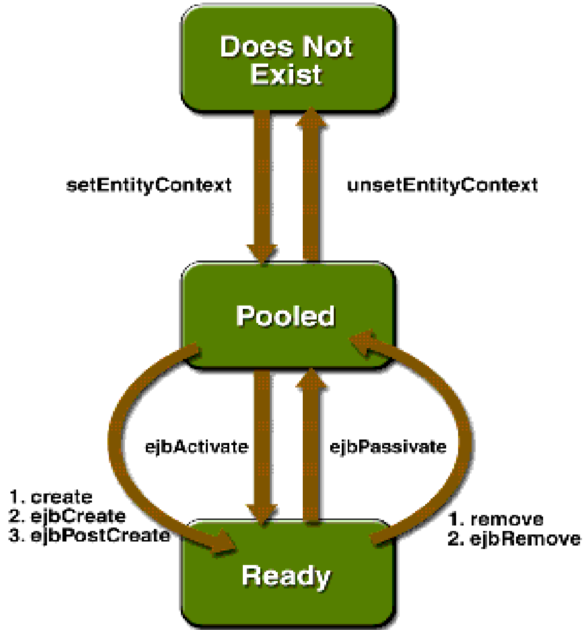
- Message-Driven Bean Eigenschaften:
  - Asynchrone Prozessierung eingehender Messages
  - Empfängt JMS Messages von Clients
  - Verarbeitet Messages einzeln.
  - Wird asynchron erzeugt
  - Lebt gewöhnlich nur kurz.
  - Repräsentiert keine persistenten Daten (zustandslos), kann aber auf persistente Daten zugreifen.
  - Kann transaktionsorientiert arbeiten.
  - Message-Driven Beans haben keine eigenen Interface Definitionen, werden also nicht direkt von Clients angesprochen

Lebenszyklus von EJBs

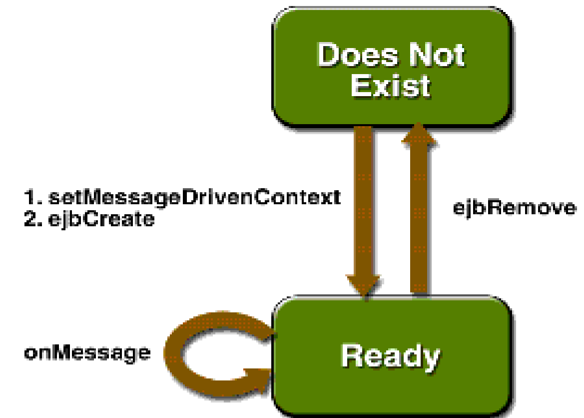


Stateful Session Bean

Stateless Session Bean



Entity Bean



Message-Driven Bean

Zugriff auf Beans

- Remote Zugriff
  - Ort des Beans ist für den Client transparent
  - Zugriff über JVMs hinweg möglich
  - Zu implementierende Schnittstellen:
    - Home Interface
    - Remote Interface
- Lokaler Zugriff
  - Ort des Beans ist für den Client nicht transparent
  - Bean und Client müssen in der gleichen JVM liegen
  - Zu implementierende Schnittstellen:
    - LocalHome Interface
    - Local Interface

Remote Interface (Session-Beans)

- Namenskonvention: ***classname***
- Erweitert **`EJBObject`** Interface
- Beschreibt die Schnittstellen der Anwendungslogik (Business logic) des Beans
- Beispiel:

```
public interface MailReaderSession extends EJBObject {
   public String getVersion() throws RemoteException;
   public String getUserName() throws RemoteException;
   // mehr Business Logic ...
}
```

Home Interface (Session-Beans)

- Namenskonvention: ***classname*Home**
- Erweitert **EJBHome** Interface
- LifeCylce Methoden (create, remove)
- Finder Methoden (Entity Beans)
- Das Bean muss für jede **create(…)**-Methode des Interfaces eine entsprechende **ejbCreate(…)**-Methode implementieren

Beispiel:

```
public interface MailReaderSessionHome extends EJBHome {
    public MailReaderSession create() throws RemoteException,
    CreateException;
    // evt. weitere create(…) Methoden
}
```

Local Interface

- Namenskonvention: ***classname*Local**
- Erweitert **EJBLocalObject**
- Beschreibt die Schnittstellen der Anwendungslogik (Business logic) des Beans
- Beispiel:

```
public interface MailReaderSessionLocal extends EJBLocalObject
  {
    public String getVersion() throws RemoteException;
    public String getUserName() throws RemoteException;
    // mehr Business Logic ...
 }
```

LocalHome Interface

- Namenskonvention: ***classname*LocalHome**
- Erweitert **EJBLocalHome** Interface
- LifeCylce Methoden (create, remove)
- Finder Methoden (Entity Beans)
- Das Bean muss für jede **create(…)**-Methode des Interfaces eine entsprechende **ejbCreate(…)**-Methode implementieren

Beispiel:

```
public interface MailReaderSessionLocalHome extends EJBLocalHome {
    public MailReaderSession create() throws RemoteException,
    CreateException;
    // evt. weitere create(…) Methoden o. find... Methoden
}
```

(Enterprise) Bean Klasse

- Namenskonvention: ***classname***
- Implementiert **`SessionBean`** oder **`MessageDrivenBean`**
- Konkrete Implementierung der Schnittstellen des Home- und des Remote-Interfaces
- Enthält Methoden die während des Bean Lifecycles vom EJB Kontainer aufgerufen werden (abhängig vom Typ des Beans) z.B.:
  - **`ejbCreate`** (analog zu den **`create(…)`**-Methoden des Home Interfaces)
  - **`ejbActivate()`**
  - **`ejbPassivate()`**

Message Driven Beans

- Im Gegensatz zu Entity o. Sessions Beans keine Local, Home o. Remote Interfaces
- Werden an eine Message-Queue des EJB Kontainers gebunden
- Muss `MessageListener` Interface implementieren (`onMessage(Message aMessage)` Methode behandelt eingehende Nachrichten)
- Jeweils genau eine `ejbCreate` und `ejbRemove` Methode

Message Passing

- Alternatives Programmiermodell
- **Kein** direkter Methodenaufruf zwischen Objekten
- **Kein** geteilter Speicher
- Objekte sind **aktiv** und voneinander **isoliert**
- Austausch zwischen Objekten, bzw. Komponenten, findet **ausschließlich** über Nachrichten statt
- Nachrichten können synchron oder asynchron sein, je nach Framework / Sprache (zuweilen wird beides angeboten)

Message Passing in Erlang

```
-module(foo).
-export([start/0, server/0]).

server() ->
    receive {calculate_answer, Client} ->
        Client ! {answer_to_life_the_universe_and_everything, 42}
    end.

start() ->
    spawn(foo, server, []) ! {calculate_answer, self()},
    receive {answer_to_life_the_universe_and_everything, R} ->
        io:format("the answer to life the universe and everything is ~p~n", [R])
    end.
```

Message Passing in Erlang

- Prozesse in Erlang werden mit der Funktion „spawn“ erzeugt
- Prozesse haben eine eindeutige ID, die mittels „self()“ erfragt werden kann und über die sie angesprochen werden können
- Nachrichten werden mit der Operator „!“ geschickt
- Es ist **irrelevant**, ob der Empfänger auf dem gleichen System oder auf einem beliebigen anderen Knoten im Netzwerk läuft
- Die physische Verteilung zur Laufzeit muss bei der Entwicklung nicht berücksichtigt werden und auch nicht bekannt sein – die **Kommunikation** ist **transparent**
- Kein zusätzlicher Aufwand für Programmierer mit Stubs o.Ä.

Fehlerbehandlung & Message Passing

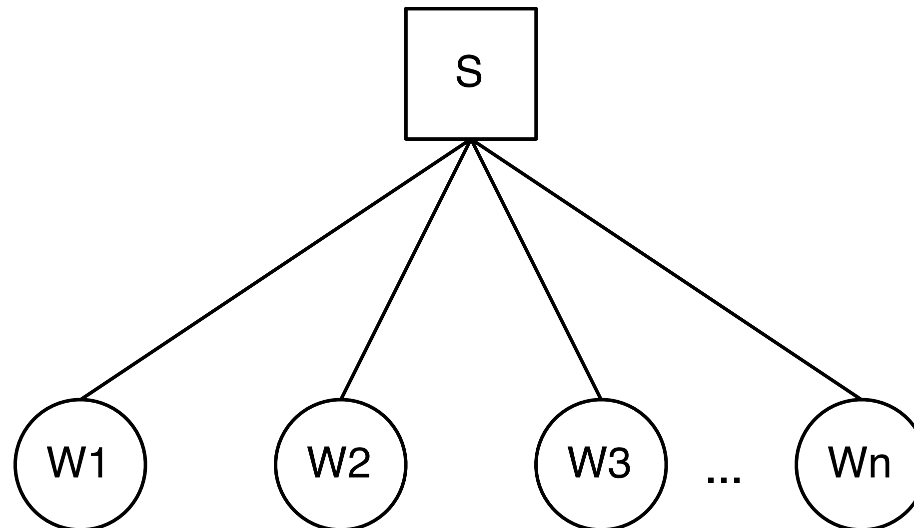
- Message Passing an sich sieht keine Absicherung auf Kommunikationsebene vor
- Ausfall beteiligter Komponenten muss erkannt werden
- Handgeschriebene Routinen (Protokoll mit Heartbeat-Nachrichten, etc.) fehleranfällig und aufwändig

Das Aktorenmodell

- Das **Aktorenmodell** beschreibt Komponenten – Aktoren –, die über Message Passing kommunizieren und sich wechselseitig überwachen können
- Erweiterung von Message Passing um eine Semantik für Fehlererkennung und -behebung, sowie Komposition von großen Systemen aus kleineren Teilsystemen
- Aktorensysteme sind i.d.R. hierarchisch organisiert, tiefstehende Aktoren können automatisiert (auch auf anderen Servern) neu gestartet werden bei Ausfall von Systemkomponenten
- Fehler werden propagiert durch Systemnachrichten

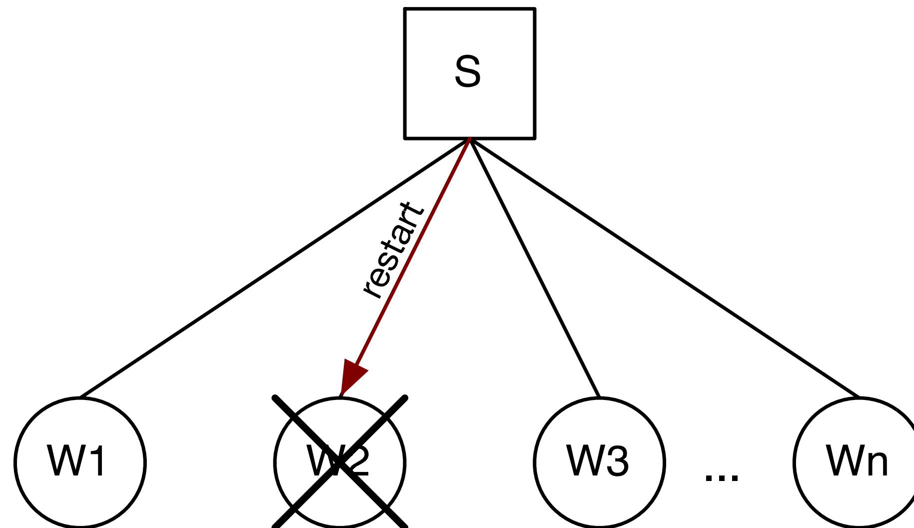
Supervision Tree

- Automatisierte Fehlerbehandlung in hierarchisch organisierten Aktorensystemen
- Teil der Standarddistribution von Erlang
- Server (S) überwacht beliebige Anzahl an Workern (W1..Wn)



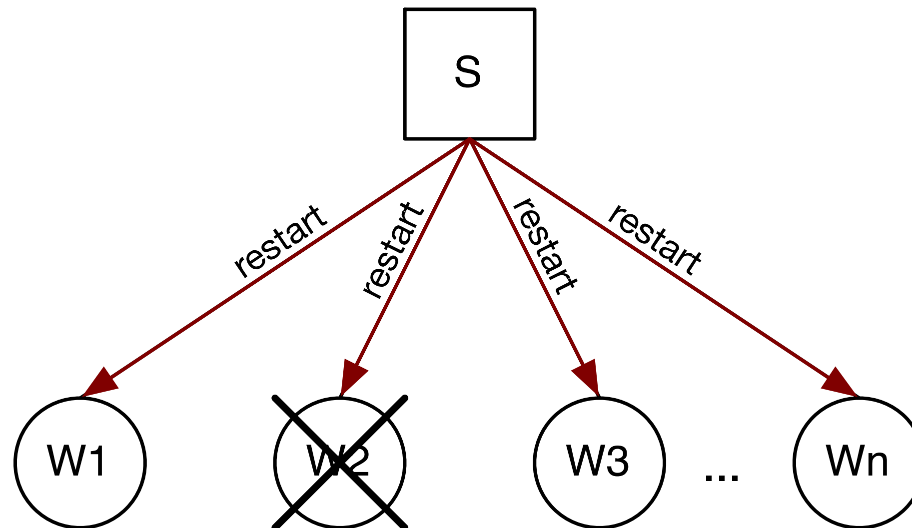
Supervision Strategien

- **„One for One“ Strategie**:
  - Startet ausschließlich den ausgefallenen Worker neu
  - Häufig verwendet bei voneinander unabhängigen Workern



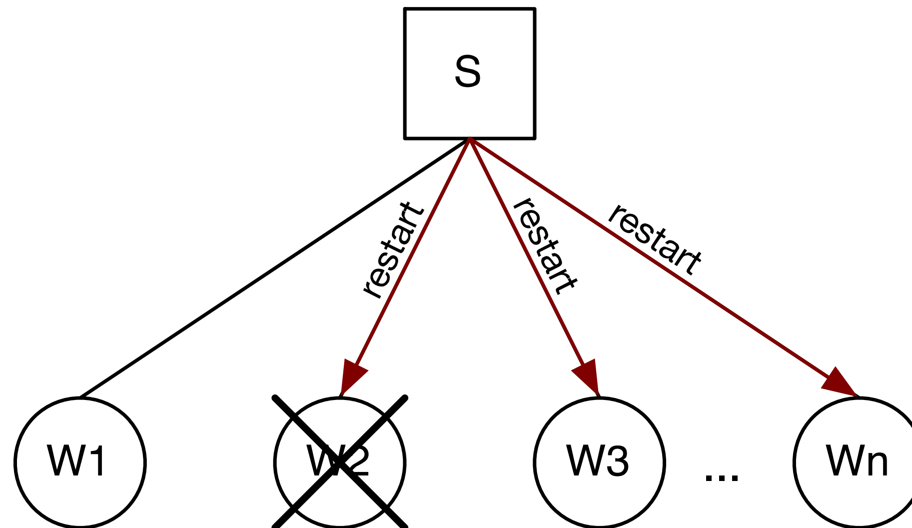
Supervision Strategien

- **„One for All“ Strategie**:
  - Bei Ausfall *eines* Workers werden *alle* neugestartet
  - Häufig verwendet wenn jeder Worker von mehreren anderen Workern (oder allen) abhängig ist



Supervision Strategien

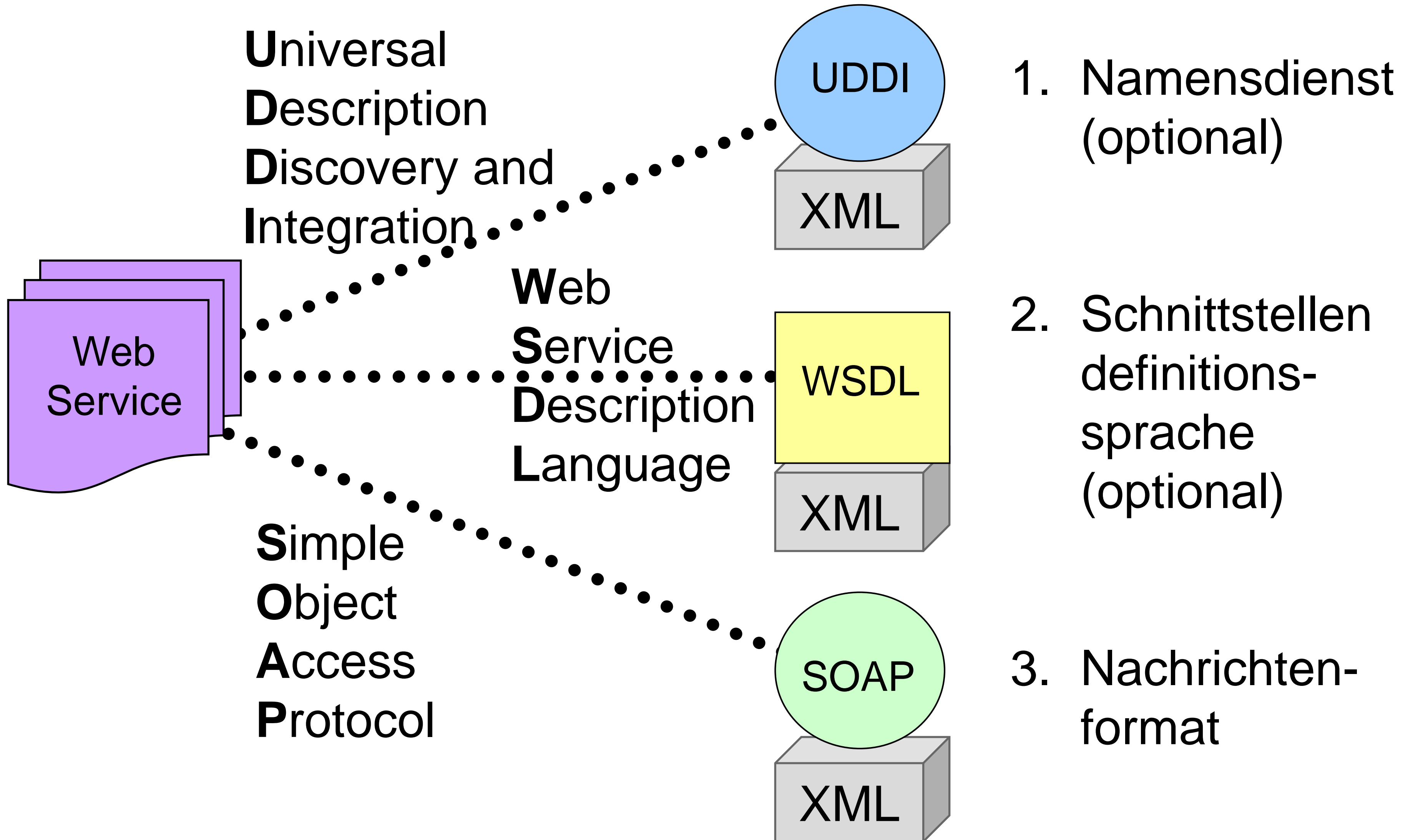
- **„Rest for One“ Strategie**:
  - Startet Worker und alle nachgeordneten Worker neu
  - Häufig verwendet wenn Worker von ihrem jeweiligen Vorgänger abhängig sind



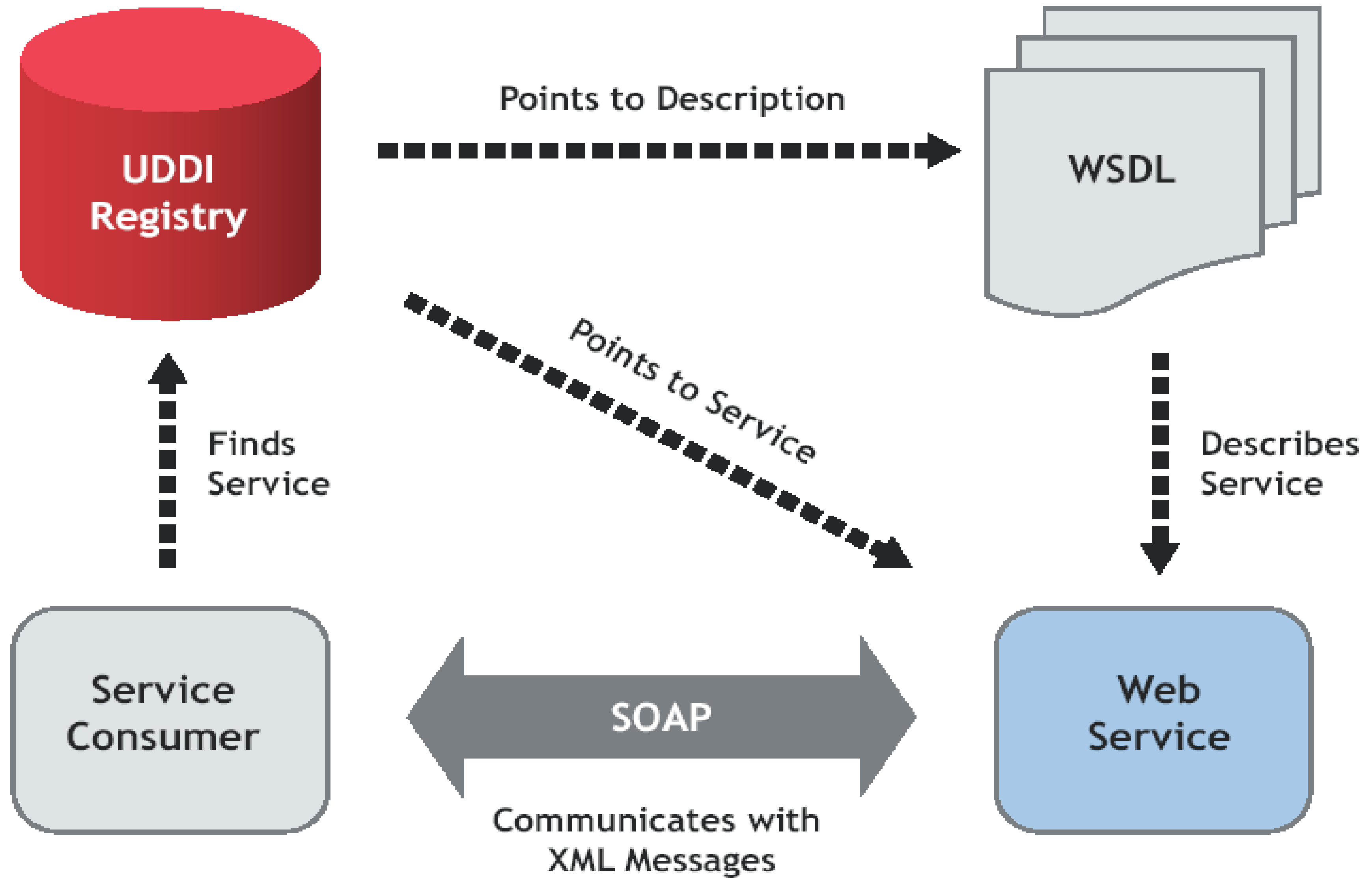
Web Services

- **Grundidee**: unvereinbare Dienste miteinander zu verknüpfen und kommunizieren zu lassen.
- Web Services sind Dienste (**Softwarekomponenten**), die im Web zur Verfügung stehen und miteinander kommunizieren.
- Offene und Hersteller unabhängige Standards:
  - **Eindeutige Identifizierung** eines Dienstes (**URI**)
  - **Autonome Dienste**, d.h. die Verarbeitung einer Nachricht eines Dienstes kann von außen nicht beeinflusst werden.
  - **Einheitliche** „mark up" Sprache für die **Kommunikation** (**XML**) mittels Internetprotokollen (z.B. HTTP, SMTP)
  - Einheitliches **Nachrichtenformat** zum Informationsaustausch (**SOAP**)
  - Einheitliches Format für die **Schnittstellen-/Servicebeschreibung** (**WSDL**)
  - **Gemeinsames Verzeichnis**, um Services auffindbar zu machen (**UDDI**)
  - **Empfehlung**: technische Schnittstellen mittels CORBA, „Dienste" mittels Web Services realisieren.

Begriffe



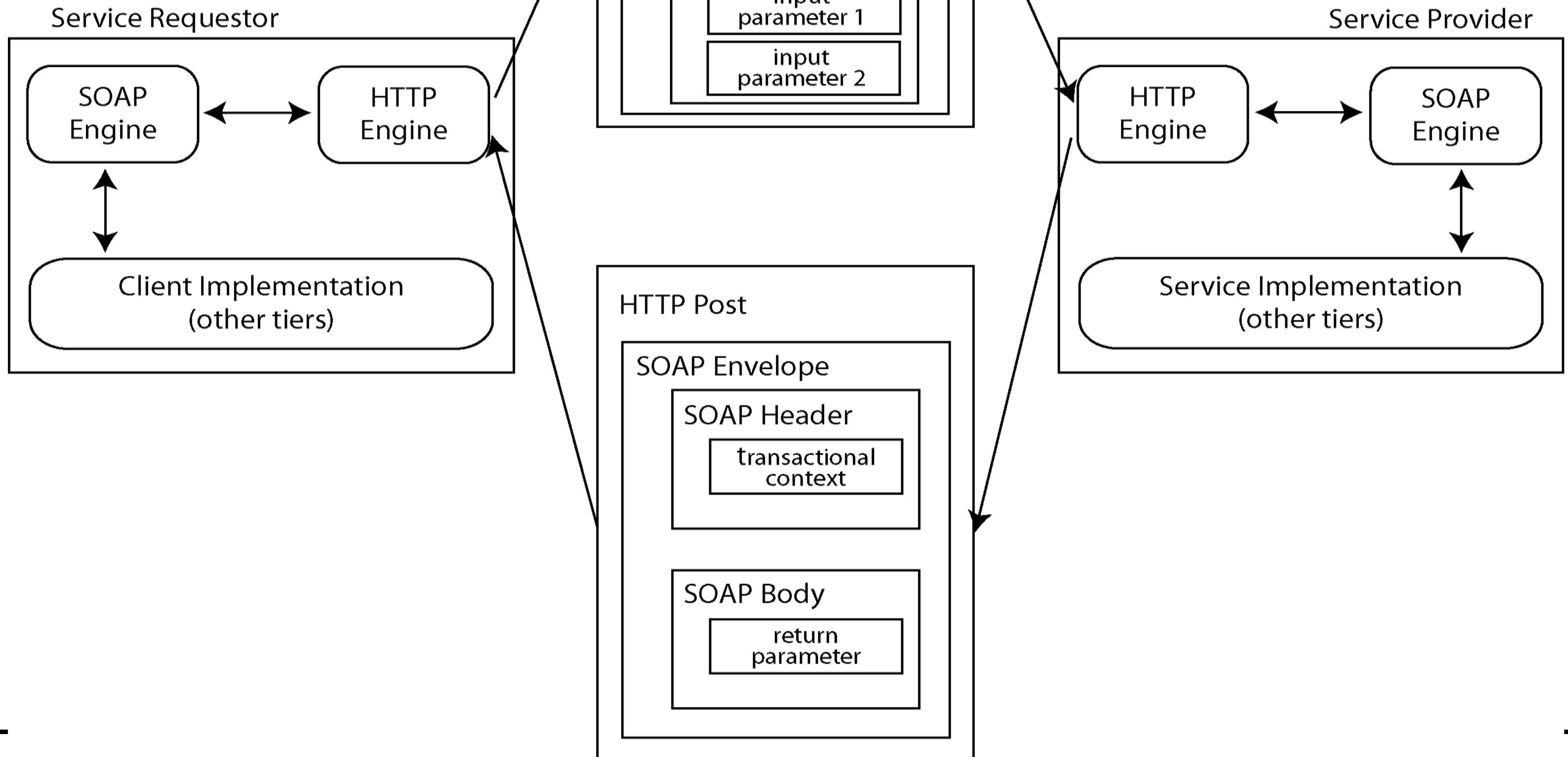
Rollen



SOAP

- SOAP **Envelope** (XML): strukturiertes und typisiertes XML-Dokument, zusätzliche Kontroll-Daten, z.B. bzgl. Transaktionssemantik, Sicherheit, Zuverlässigkeit
- SOAP **Transport Binding**: Für Kommunikation genutztes Netzwerkprotokoll (Standard: HTTP, möglich u.a. IIOP)
- SOAP **Encoding Rules**: Definition, wie (komplexe) Parameter und Ergebniswerte serialisiert werden
- SOAP **RPC** Mechanismus: Vorgänger ist XML/RPC
- SOAP **Intermediaries** (Mittelsleute): Bearbeiten ggf. die Nachricht auf dem Weg vom Sender zum Empfänger (z.B. Protokollierung, Abrechnung)

Typische SOAP Transaktion: RPC via HTTP



SOAP über HTTP: Anfrage

```
POST /EndorsementSearch HTTP/1.1
Host: www.stock-info.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 261
SOAPAction:"http://www.stock-info.com/StockQuoteService"
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
  ...
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
   <m:GetLastTradePrice xmlns:m="http://namespaces.stock-info.com">
     <symbol>IBM</symbol>
   </m:GetLastTradePrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Service Lokation

Methoden-Aufruf "GetLastTradePrice"

Methoden-Parameter "GetLastTradePrice"

SOAP über HTTP: Antwort

```
HTTP / 1.1 200 OK
Content-Type: text/xml; charset="utf-8"
...............
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
  <m:GetLastTradePriceResponse xmlns:m="http://namespaces.stockinfo.com">
    <price>135.34</price>
  </m:GetLastTradePriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Antwortname "GetLastTradePriceResponse"

Ergebnis <price> 135.34 ...

WSDL

- Beschreibt abstrakt, d.h. unabhängig vom Nachrichtenformat oder Netzwerkprotokoll, Web Services als eine Menge von **Zugriffsendpunkten**, die untereinander **Nachrichten** auf prozedur- oder dokumentenorientierter Weise **austauschen**
- WSDL ist eine **XML-Grammatik**
- Beschreibung beinhaltet Informationen über:
  - Funktionsweisen eines Web Services *Was*
  - zulässigen Datenformate (`types`) *Wie*
  - Form der Operationsaufrufe (`PortType`) *Wie*
  - Ort des Web Services (`service`) *Wo*
- WSDL ist ein **Rezept**, das dazu dient, die Details der Kommunikation zwischen Anwendungen zu automatisieren.

WSDL Beispiel (V1.1)

Datentypen und Nachrichten

```
<types> </types>
<message name="getLastTradePriceRequest">
   <part name="companyName type="xsd:string”/>
</message>
<message name="getLastTradePriceResponse">
   <part name="price" type="xsd:float"/>
</message>
<portType name="StockQuotePortType">
   <operation name="getLastTradePrice">
       <input   message="myns:getLastTradePriceRequest"/>
       <output  message="myns:getLastTradePriceResponse"/>
   </operation>
</ portType>
```

Aufrufbare Methoden

WSDL Beispiel (V1.1)

Client –Server Kommunikation

```
<binding name="StockQuoteSoapBinding„
   type="tns:StockQuotePortType">
<soap:binding style="rpc" transport="http"/>
<operation name="GetLastTradePrice">
   Soap-spezifische Einstellungen...
</operation>
</binding>
<service name="StockQuoteService">
<port name="StockQuotePort"
   binding="tns:StockQuoteBinding">
<soap:address
   location="http://www.stockquoteserver.com/stockquote"/>
</port>
</service>
```

Ort des Web Services

REST-style Web Services

- URL identifiziert **genau einen** Inhalt
- Zustandsloses Client-Server-Protokoll
- **Keine** Zusatzinhalte bei HTTP-GET, Anfrage codiert in URL
- Antwortformat des Servers frei wählbar, je nach Anwendung

REST Request

- Anfrage Kodiert in URL, z.B. „/users/12345“ für den Namen des Benutzers mit der ID 12345

```
GET /users/12345 HTTP/1.1
Host: www.example.com
```

REST Antwort auf Request

- Antwort als HTTP Response im Klartext (andere Formate wären z.B. HTML, XML, etc.)

```
HTTP/1.1 200 OK
Date: Wed, 13 May 2009 16:26:47 GMT
Last-Modified: Wed, 13 May 2009 15:26:47 GMT
Content-Length: 12
Connection: close
Content-Type: text/plain

Dirty Harry
```

REST Post

- Schreib-Anfrage ebenfalls als URL kodiert
- Beispiel: der ID 42 den Namen „Douglas Adams“ zuweisen

```
POST /users HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 33

userID=42&userName=Douglas+Adams
```

REST Antwort auf Post

- Format wiederum frei wählbar (hier: Klartext)

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 14

Douglas Adams
```

REST Zusammenfassung

- Direkt auf HTTP abbildbar
- Keine zusätzlichen Protokolle
- Verbindungslos (selbst wenn TCP benutzt wird)
- Datenformat zwischen Client und Server frei wählbar, je nach Anfrage („Content-Type“-Feld in HTTP-GET) kann der Server die gleichen Daten in unterschiedlichen Repräsentationen ausliefern
- Skalierbarkeit durch lose Kopplung zwischen Client und Server: bei replizierten Servern kann jede Anfrage von einer anderen Maschine bearbeitet werden