

## E Überblick über die 4. Übung

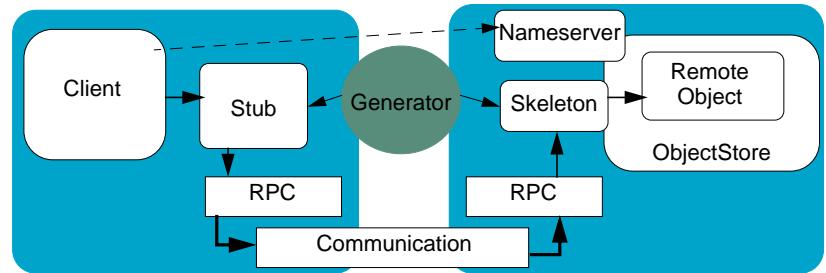
- RPC und ORB
  - ◆ Aufrufsemantiken
  - ◆ Hinweise Aufgabe 3

### E.1 Object Request Brokers

- ermöglichen Methodenaufrufe an entfernten Objekten (Objekte in anderen JVM)
- Beispiel-ORBs: RMI, JavaIDL

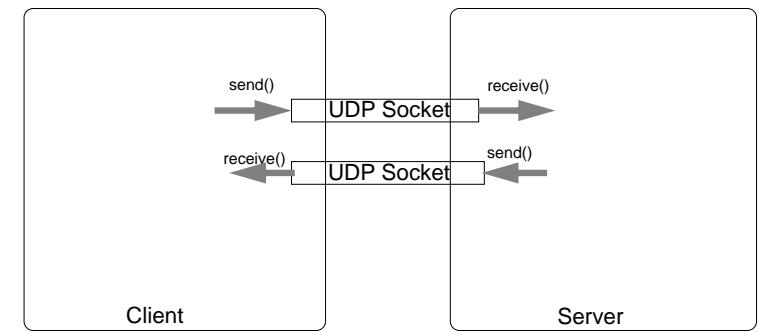
### 1 Komponenten eines ORBs

- *Kommunikationsschicht*: tauscht Daten zwischen zwei Rechnern aus
- *RPC Schicht*: definiert die Aufrufsemantik und das Marshalling
- *Object Store*: verwaltet den Lebenszyklus der Objekte
- *Stub / Skeleton Generator*: erzeugt Code für die Stubs und die Skeletons
- *Nameserver*: findet Objekte anhand deren Namen



### 2 Kommunikationsschicht

- zuständig für den Datenaustausch zwischen zwei Rechnern
- verwendet DatagramSocket (UDP)



## 3 RPC-Schicht

- erledigt die Weiterleitung von Methodenaufrufen
  - wird von den Stubs und den Skeletons verwendet
  - verwendet die Kommunikationsschicht um Bytes zu versenden

## E.1 Object Request Broker

## E.1 Object Request Brokers

## 3 Marshalling

- RPC-Schicht verpackt die Parameter in ein Anfragepaket und den Rückgabewert in ein Antwortpaket == Marshalling
  - kopiere alle Parameter zum Server  
(besser wäre es, eine Referenz zu verschicken wenn ein Parameter ein *remote Interface* implementiert)
  - verwende ObjectStreams/ByteArrayStreams/Datagramme um Objekte zu verschicken:

```
ByteArrayOutputStream stream = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(stream);
out.writeObject(...);
byte[] buf = stream.toByteArray();
DatagramPacket packet = new DatagramPacket(buf, ... );
```

- Annahme für Aufgabe 3: alle Parameter passen in ein `DatagramPacket`

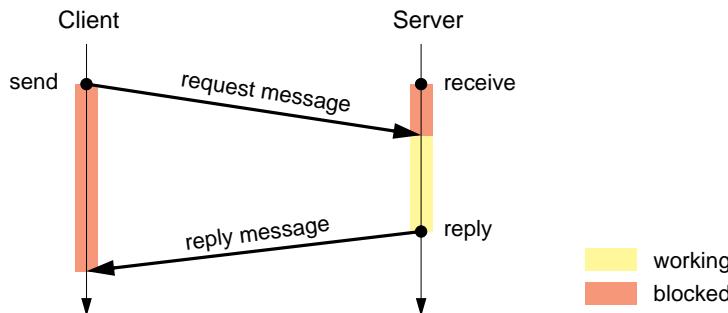
**Übungen zu Middleware**  
© Universität Erlangen-Nürnberg • Informatik 4, 2006

E=ORBB fm 2006-11-13 11 0

E.5

3 RPC

- einfachster RPC ist ein primitives (synchroes) request/reply-Protokoll:



## E.1 Object Request Broker

## E.1 Object Request Brokers

### 3 Fehlerbehandlung

- Implementierung einer bestimmten Aufrufsemantik
    - ◆ exactly once
    - ◆ at least once
    - ◆ at most once
    - ◆ last of many
    - ◆ ...
  - abhangig von der Servicequalitat der Kommunikationsschicht
  - muss mit Kommunikationsfehlern umgehen konnen:
    - ◆ verlorene Pakete
    - ◆ veranderte Reihenfolge (non-FIFO)
    - ◆ duplizierte Pakete
    - ◆ veranderte Pakete

**Übungen zu Middleware**  
© Universität Erlangen-Nürnberg • Informatik 4, 2006

E=ORBB fm 2006-11-13 11 0

E<sub>1</sub>E<sub>2</sub>

Researchatham (einf. Art und) Netzwerkanalyse dienen Unterstützungs- und Lehrmaterialien an der Universität Erlangen-Nürnberg. Inhalt des Zeitungsausschnitts: Am

E 8

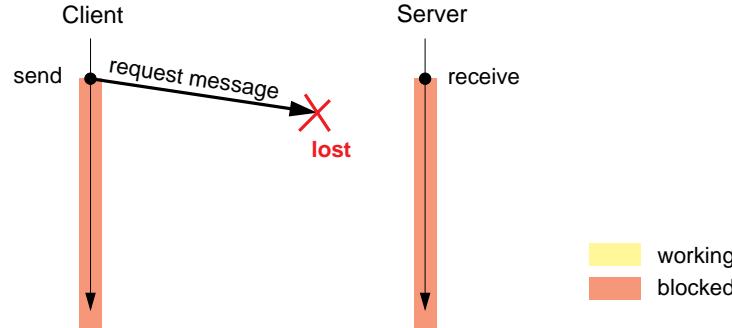
**Übungen zu Middleware**  
© Universität Erlangen-Nürnberg • Informatik 4, 2006

E.8

<sup>1</sup> Zusammenfassungen finden Sie unter [www.uni-freiburg.de/medizin/medien/medienwissenschaften/medienwissenschaften\\_im\\_fokus/medienwissenschaften\\_im\\_fokus.html](http://www.uni-freiburg.de/medizin/medien/medienwissenschaften/medienwissenschaften_im_fokus/medienwissenschaften_im_fokus.html).

### 3 Fehlerbehandlung - Problem 1

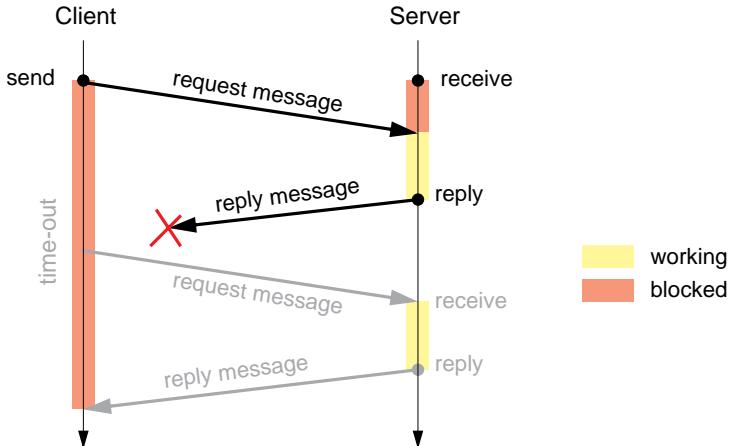
- Kommunikationsfehler: Verlust einer Anfragenachricht



### 3 Fehlerbehandlung - Problem 2

- Kommunikationsfehler: Verlust einer Antwortnachricht

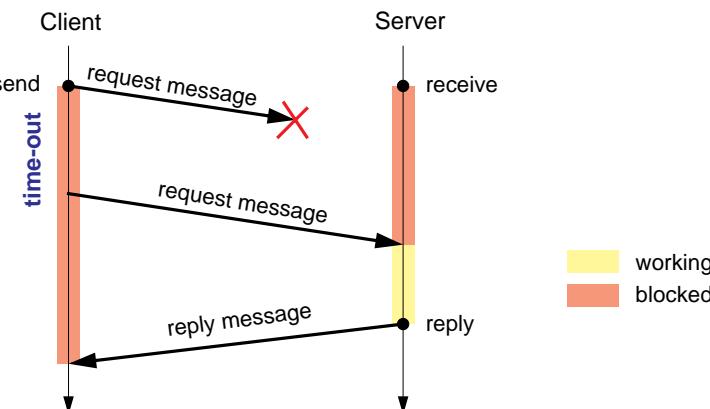
- bei Wiederholung wird die Methode mehrmals ausgeführt



### 3 Fehlerbehandlung - Lösung 1

- Verlust einer Anfragenachricht:

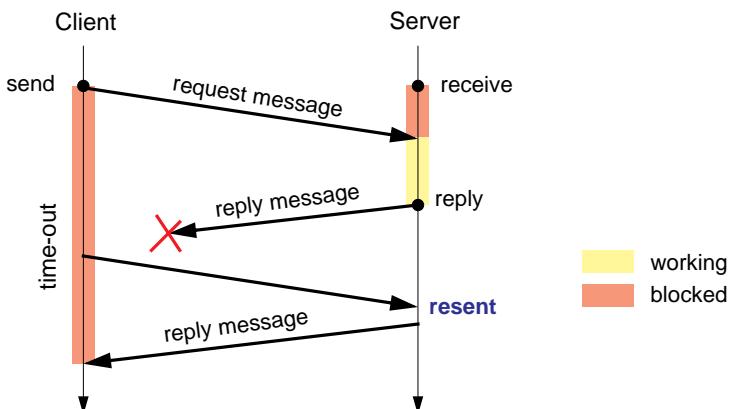
- mögliche Lösung: Anfrage erneut schicken



### 3 Fehlerbehandlung - Lösung 2

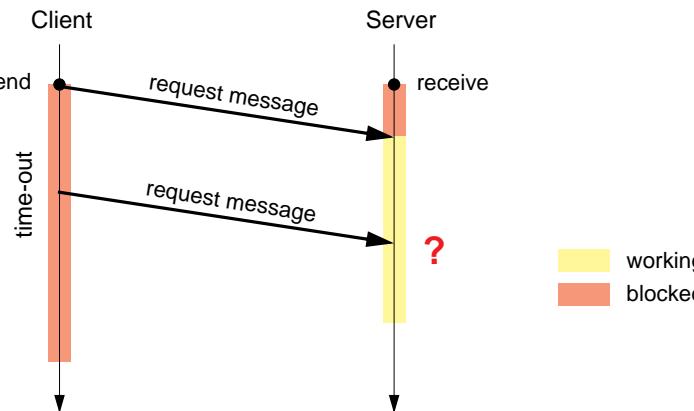
- Verlust einer Antwortnachricht

- Server hebt Antwort auf und wiederholt sie



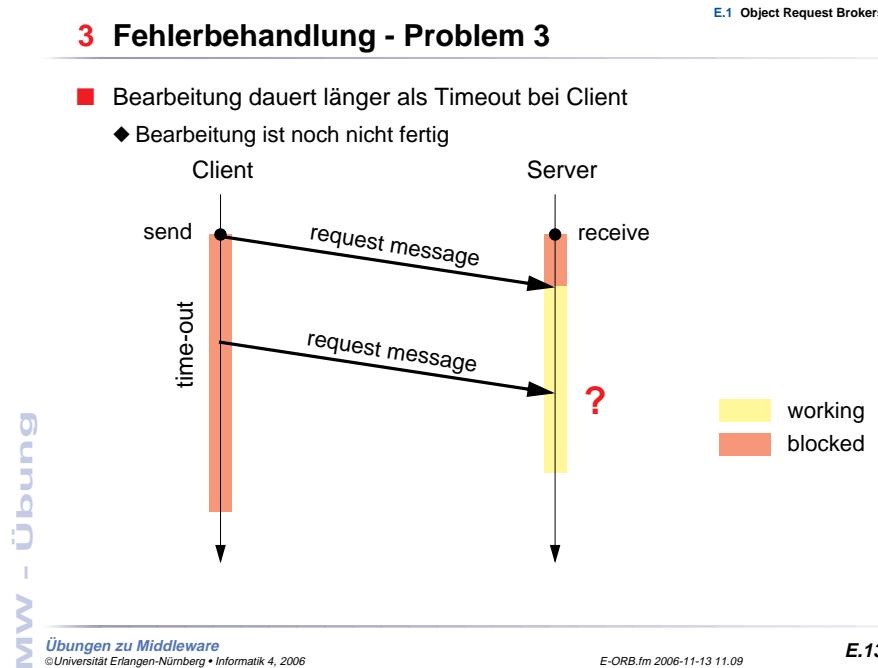
### 3 Fehlerbehandlung - Problem 3

- Bearbeitung dauert länger als Timeout bei Client
  - ◆ Bearbeitung ist noch nicht fertig



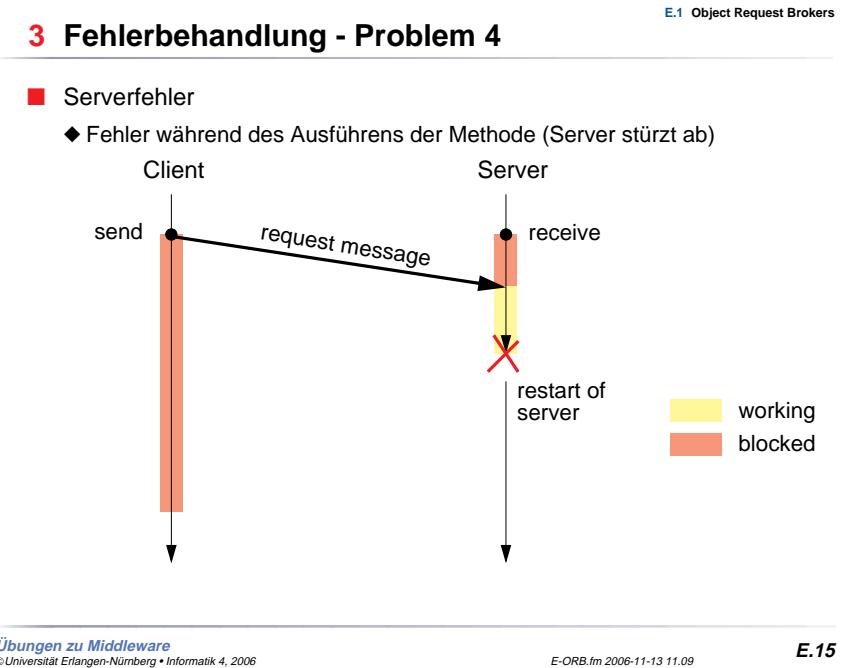
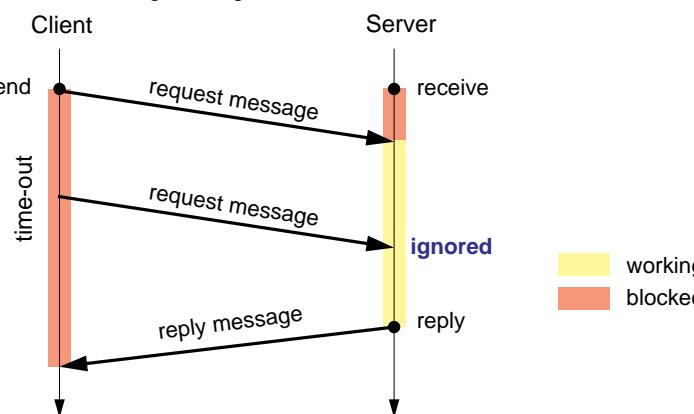
### 3 Fehlerbehandlung - Problem 4

- Serverfehler
  - ◆ Fehler während des Ausführens der Methode (Server stürzt ab)



### 3 Fehlerbehandlung - Lösung 3

- Bearbeitung ist noch nicht fertig
  - ◆ wiederholte Anfrage wird ignoriert



### 3 Fehlerbehandlung - mögliche Implementierung

- Annäherung an *exactly once*:
  - ◆ Um verlorene Anfragenachrichten entgegenzuwirken werden die Pakete nach einem Timeout erneut gesendet.
  - ◆ Zum Vermeiden von mehrfachen Ausführungen bei doppelt gesendeten Paketen werden IDs verwendet, welche bei einem erneuten Versenden der gleichen Anfrage nicht verändert werden.
  - ◆ Der Server soll abgesendete Antworten puffern, für den Fall, dass eine Antwort verloren geht und um eine erneute Ausführung der Methode zu vermeiden.
  - ◆ Den Absturz eines Servers zu verkraften ist schwieriger ....  
..... daher ignorieren wir das Problem.

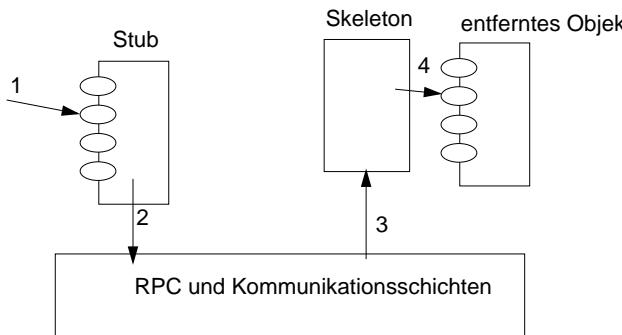
### 4 Stub

- implementiert den gleichen Typen wie das entfernte Objekt (gleiches Interface)
- verpackt einen Methodenaufruf in ein Request-Objekt:
  - ◆ Objekt ID, Methoden ID, Parameter
- verwendet die RPC-Schicht um eine Anforderung zu versenden
- transformiert das Rückgabeobjekt in den entsprechenden Typ

### 4 Stub Beispiel

```
package test;
import orb.*;
public class AccountImpl_Stub implements test.Account {
    int oid;
    ClientContext ctx;
    public AccountImpl_Stub(int oid, ClientContext ctx) {
        this.oid = oid;
        this.ctx = ctx;
    }

    // erzeugte Methoden (Bsp ohne Exceptionbehandlung)
    public int deposit(int param0) {
        Object[] parameters = new Object[1];
        parameters[0] = new Integer(param0);
        Request req = new Request(ctx, oid, 9, parameters);
        Object ret = req.send();
        return ((Integer)ret).intValue();
    }
    // weitere erzeugte Methoden ...
}
```



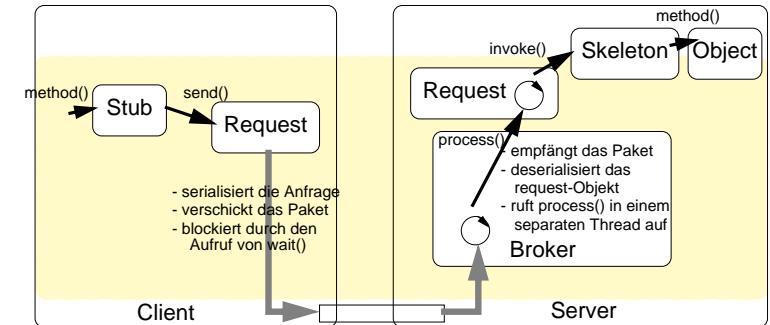
## 4 Skeleton

- ruft Methoden am "echten" Objekt auf
- notwendige Informationen:
  - ◆ Objektreferenz
  - ◆ Methoden ID
  - ◆ Parameter

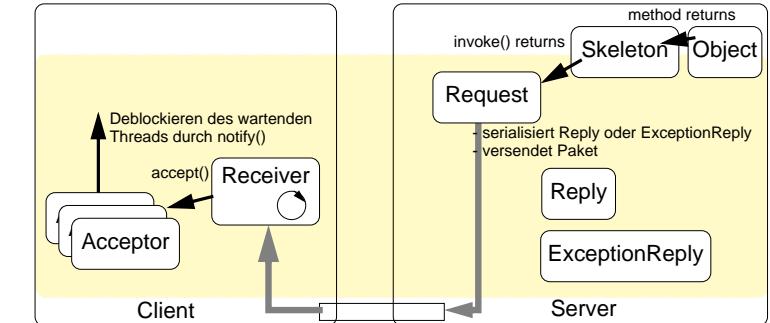
## 4 Skeleton Beispiel

```
package test;
import orb.*;
public class AccountImpl_Skel implements Skeleton {
    AccountImpl object;
    public AccountImpl_Skel() { /* ... */ }
    public void init(int oid, Object object) {
        this.oid = oid;
        this.object = (AccountImpl) object;
    }
    public Object invoke(int mid, Object[] parameters)
        throws Exception {
        switch(mid) {
            ...
            case 9: {
                int p1 = ((Integer)parameters[0]).intValue();
                int result = object.deposit(p1);
                return new Integer(result);
            }
        }
    }
}
```

## 4 Request



## 4 Reply



## 5 Object Store

### ■ Interface:

- ◆ `int registerObject(Object obj)`  
gibt eine eindeutige Objekt-ID zurück.

- ◆ `Object lookupByID(int oid)`  
liefert das Objekt zur gegebenen Objekt-ID

### ■ Implementationstechnik:

- ◆ `java.util.Hashtable` kann verwendet werden um die Abbildung zwischen Objekt-IDs und Objekten vorzunehmen (Hinweis: die Hashtabelle kann nur Objekte speichern, die *OID* sollte daher in ein `Integer`-Objekt gekapselt werden.)

## 7 Nameserver

### ■ Stub-Erzeuger

- ◆ sucht die ID (und den Klassennamen); erzeugt daraufhin ein Stub-Objekt
  - `Object lookup(String name)`
- ◆ der Nameserver lädt die Klasse und erzeugt eine Instanz der Klasse

## 6 Nameserver

### ■ ist aufgeteilt in einen ID-Finder und einen Proxy-Erzeuger

### ■ ID Finder

- ◆ bildet Namen auf OIDs ab:

- `void bind(String name, int oid)`
- `int lookupID(String name)`

- ◆ verwendet `Hashtable`

- ◆ ist selbst als entferntes Objekt mit der OID 1 implementiert

- ein realistischer Nameserver würde Referenzen auf Stub-Objekte zurückliefern anstatt OIDs

- das erfordert, dass zusammen mit der OID der Klassename des Stubs zurückgegeben wird.