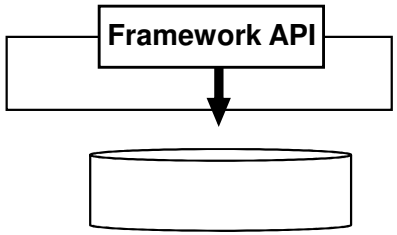
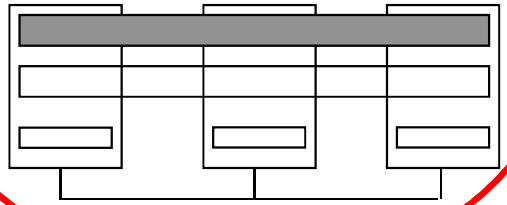
Software Engineering III – Wo sind wir?

Softwarearchitekturen

Softwarearchitekturen: Begriffe & Fallbeispiel Persistenz-Framework

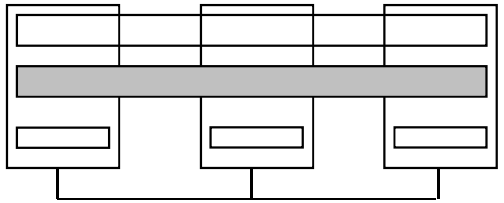


Verteilte Softwarearchitekturen



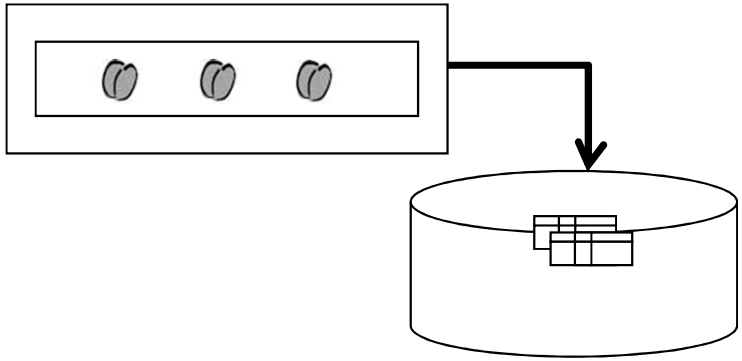
Middleware/Verteilung

Sockets, RMI, MoM/JMS Web Services



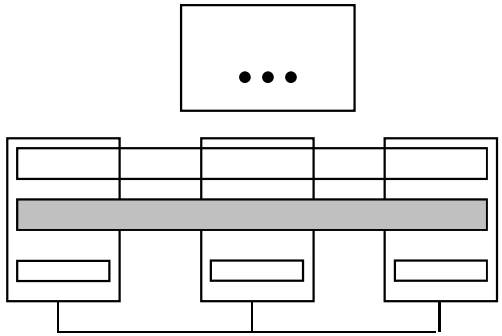
Enterprise Java

Java EE (EJB, …)



Verteilung – weitere Konzepte

Weitere Middleware



Gliederung

Definition: Verteilte Systeme

- **Verbund von Rechnern** mit separatem Speicher
  - Rechner sind durch ein Netzwerk verbunden
  - **Infrastruktur** für verteilte Anwendung (s.u.)

Abgrenzung: kein gemeinsamer Speicher

Definition: Verteilte Anwendung

- **Softwaresystem mit Komponenten auf verschiedenen Rechnern**
- Komponenten kooperieren über das Netzwerk miteinander
  - nutzen Interprozess-Kommunikation: Sockets, RMI, JMS, Web Services...

Bedeutung verteilter Systeme

- **zunehmende Vernetzung**
  - Internet, unternehmensinterne Netze
  - Mobilnetz
  - CAN (controller area network) in Autos, ...

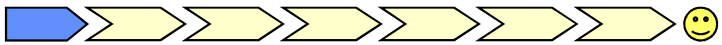
- **verteilte Anwendungen – Beispiele**
  - klassische Informationssysteme in Verwaltungen und Unternehmen
    - zentraler Datenbestand, Steuerung von Arbeitsabläufen (Workflow-Systeme)
  - Webbasierte Systeme (Amazon, Google Docs, ...), Cloud-Dienste
  - viele Smartphone-Apps
  - Supercomputer

Vorteile verteilter Systeme

- **bilden die Realität (= verteilten Gegebenheiten) ab**
  - Kommunikations- und Datenverbund für Benutzer an verschiedenen Standorten
  - gleichzeitige parallele Aktivitäten
- **Skalierbarkeit:**
  - Lastverteilung: Leistungsengpässe durch weitere Rechner kompensieren
  - mehrere Rechner arbeiten parallel an einer Aufgabe
- **Fehlertoleranz**
  - Softwarekomponenten auf mehreren Rechnern replizieren
  - kein Single Point of Failure
    (trotz Rechnerausfall ist komplette Funktionalität verfügbar)

Herausforderungen bei der Entwicklung verteilter Anwendungen

- **komplexe Kommunikation / Systeme**
  - Interprozess-Kommunikation statt lokaler Methodenaufrufe
  - heterogene Systeme (Rechner, Betriebssysteme, Hardware, Sprachen, . . .)
  - Parallelität
- **Performanz**
  - Kommunikation über langsame Netzwerke
- **Zuverlässigkeit**
  - alle Netzzugriffe sind potentiell unsicher
  - Maschinenausfälle
- **Transaktionssicherheit**
  - mehrere Benutzer greifen gleichzeitig auf Daten zu: Isolation von Transaktionen

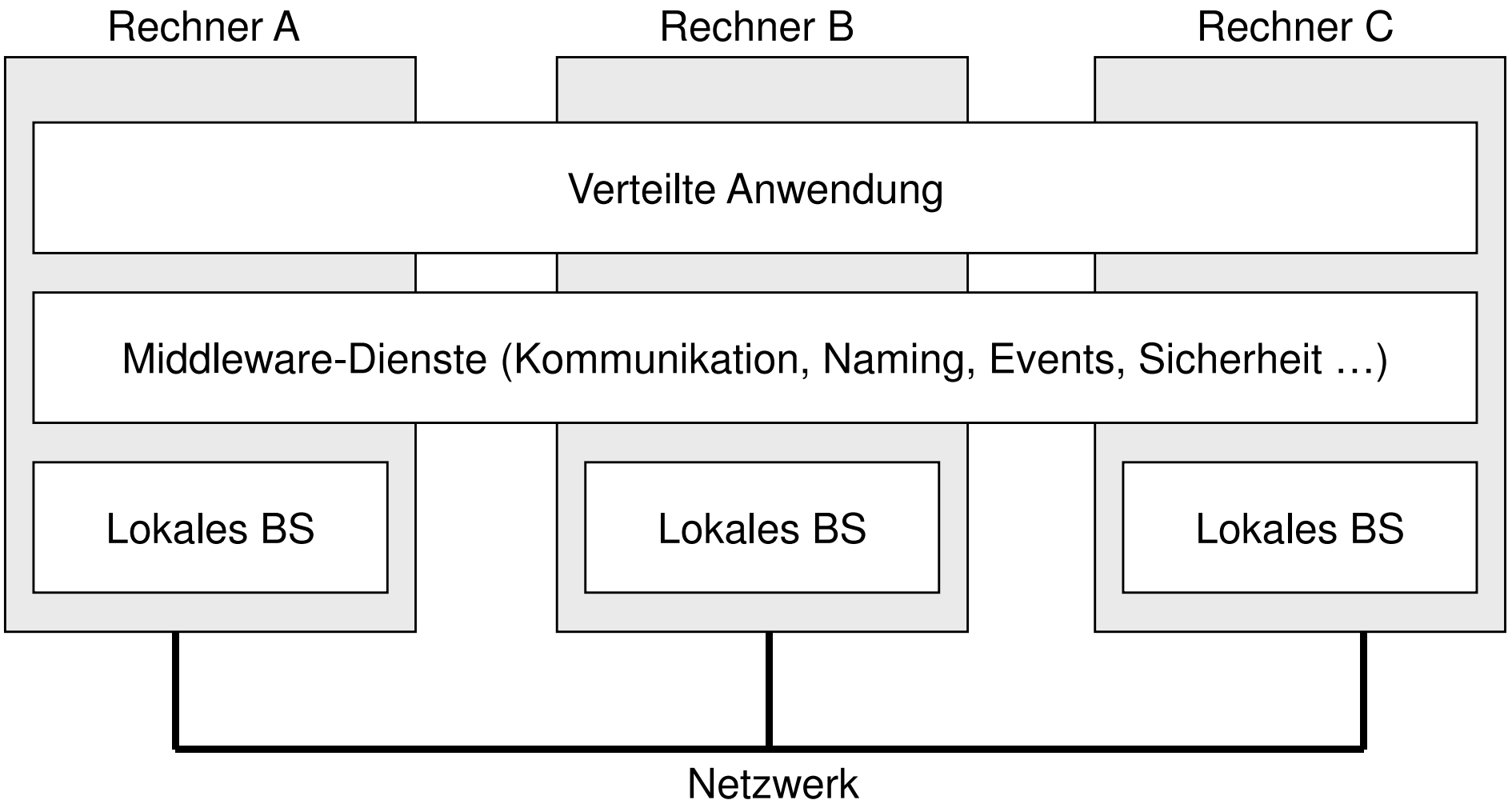


Middleware – Definition

- **infrastrukturelle Software zur Kommunikation zwischen SW-Komponenten**
  - bietet einheitliches Programmiermodell zur Entwicklung verteilter Systeme
  - Verteilungsplattform mit entsprechenden Protokollen
    - höheres Abstraktionsniveau als einfacher Datenaustausch
    - verbirgt Komplexität der zugrunde liegenden Applikationen /Infrastruktur

- **Ziele**:
  - Interoperabilität
  - vereinfachte Erstellung verteilter Anwendungen

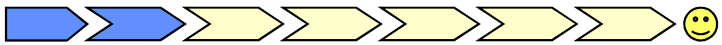
Platzierung der Middleware



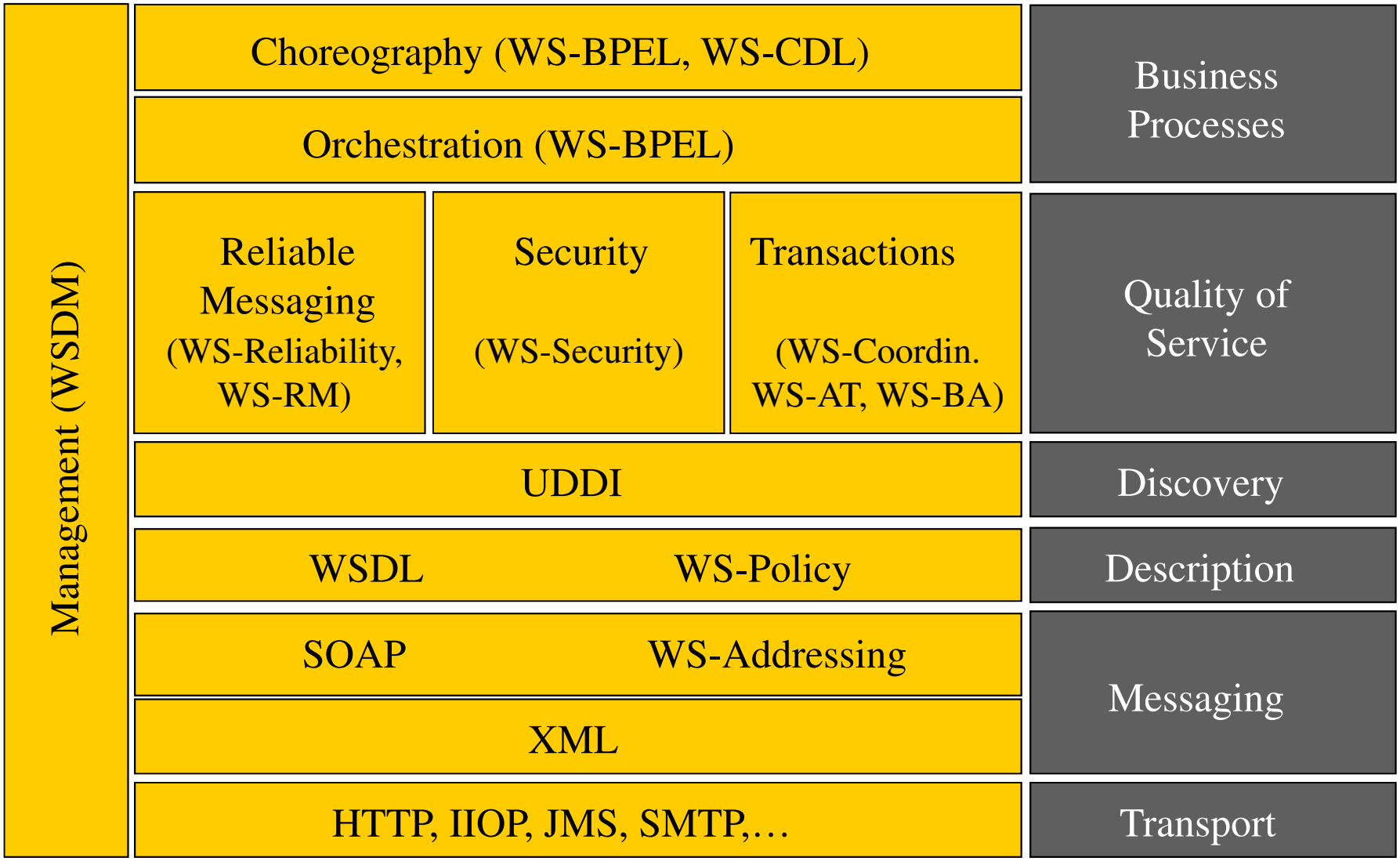
Arten von Middleware

- **Kommunikationsorientierte Middleware**
  - Schwerpunkt: Abstraktion von der Netzwerkprogrammierung
  - Beispiele: Java RMI, Web Services „pur“

- **Anwendungsorientierte Middleware**
  - Schwerpunkt: weitreichende Unterstützung verteilter Anwendungen
  - Komplexerer Aufbau, zahlreiche Zusatzdienste
    - Discovery, Sicherheit, Zuverlässigkeit, verteilte Transaktionen, Sessions
  - Beispiele:
    - verteilte Systemansätze wie CORBA, Java EE oder .NET
    - verteilte Betriebssysteme
    - . . .

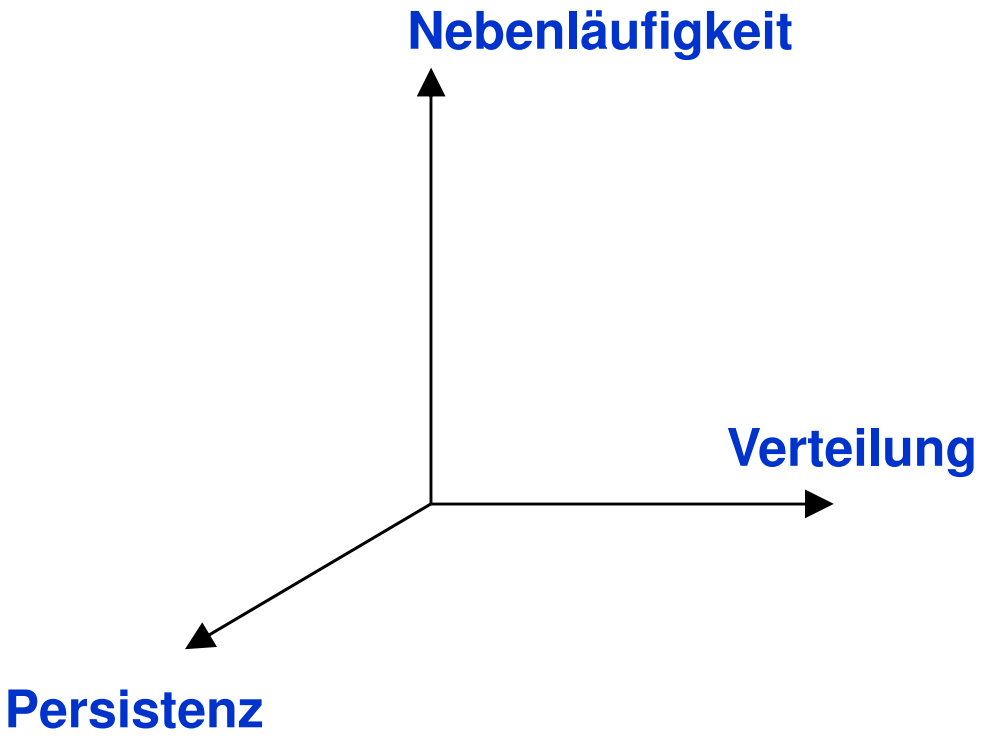


Beispiel für Middleware – WS-* Web Services Stack



Dimensionen verteilter Systeme

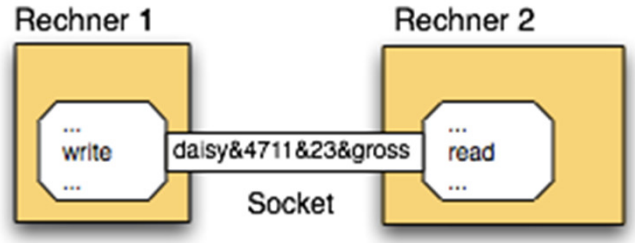
- verteilte Anwendungen besitzen typischerweise mindestens drei Dimensionen, die zunächst voneinander unabhängig (orthogonal) sind



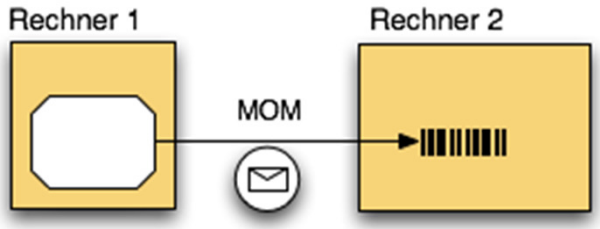
- weitere Dimensionen sind z.B.: Autonomie, Heterogenität, Quality of Service (QoS)

Verteilung (1)

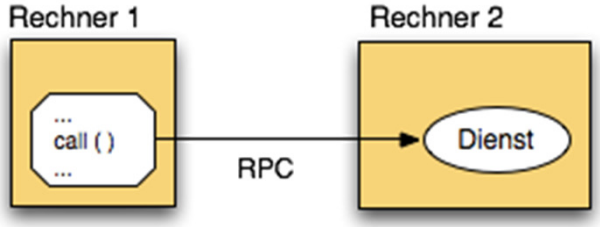
- Software ist über Rechner-/Prozessgrenzen verteilt
- Kommunikation erfolgt auf verschiedenen Abstraktionsebenen:
  - **Datenaustausch:**
    - Sockets, HTTP-Requests



  - **Nachrichtenaustausch:**
    - MOM, SOAP-Nachrichten



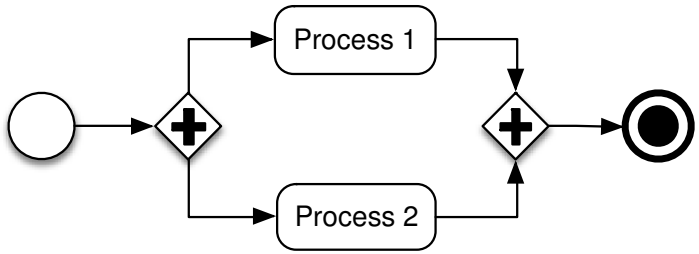
  - **Call Level-Interface**
    - Aufruf von Diensten
    - RMI in Java, CORBA, Web Services



- einige Technologien bieten **Quality of Service:** Sicherheit, Zuverlässigkeit, Transaktionen, ..

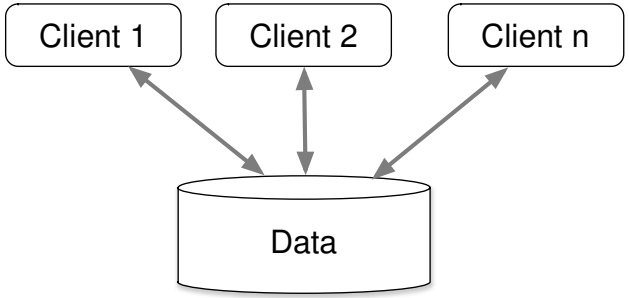
Nebenläufigkeit

- es gibt **mehrere Prozesse mit jeweils eigenem Kontrollfluss**
  - laufen (zeitweilig) unabhängig voneinander ab



  - Variante A: **ein Rechner** (=eine CPU) bearbeitet viele Prozesse (quasi-)parallel
    - leichtgewichtige Prozesse durch Threads realisiert
      (bekommen im round-robin-Verfahren Zeitscheiben zugeteilt)
  - Variante B: **echte Parallelität**
    - Prozesse werden parallel auf verschiedenen Rechnern ausgeführt
  - nebenläufige Prozesse müssen zur Kooperation synchronisiert werden

Persistenz



- **Daten werden an zentraler Stelle persistent verwaltet**
  - in RDBMS, NOSQL, XML-Datenbanken,..
  - stehen im gesamten Netz zur Verfügung
- **API zum netzweiten Zugriff auf Datenbanken**
  - per Datenbankzugriffsprotokoll (bspw. JDBC)
- **Entwicklung von Frameworks zur Realisierung von Objekt-Persistenz**
  - Eigenentwicklungen
  - Standards: Java Persistence API (JPA), ...

Grundbegriffe VS - Fragenblock

- **Was ist ein verteiltes System, was eine verteilte Anwendung?**
- **Was ist Middleware?**
- **Wo wird Middleware in einem VS angeordnet?**
- **Erklären Sie „Dimensionen verteilter Systeme“**

Realisierungskonzepte für entfernte Aufrufe

- Aufruf von Diensten = Call Level Interfaces (Remote Procedure Call – RPC)
  - konzeptionell identisch (für RMI, CORBA, EJB, einfache Web Services, ...)

- zunächst: Konzepte unabhängig von konkreter Technologie

Allgemeiner Ablauf am Beispiel eines entfernten Methodenaufrufs:

1. **Bereitstellen** einer entfernten Methode
2. **Auffinden** eines entfernten Objekts
3. **Ausführung** entfernter Methodenaufrufe
4. **Parameterübergabe**

1. Bereitstellung einer entfernten Methode

- Dienstanbieter (Server) bietet entfernten Clients bestimmte Dienste an

1.1 Spezifikation des Dienstes

- Klasse, Methodenname, Parameterstruktur und Rückgabewert
- Schnittstellenspezifikation z.B.: Java Interfaces, eigene Spezifikationssprache (IDL, WSDL), …

1.2 Implementierung des Dienstes

1.3 Registrierung des Dienstes im Server

- in einem Server-Prozess wird Dienst-**Objekt unter einem eindeutigen Namen registriert**
  - Laden von Dienst-Spezifikation und -Implementierung
  - Name Service-Prozess läuft unter bestimmtem Port
- mögliche technische Umsetzungen:
  - Programmier-Schnittstelle `(Naming.rebind("rmi://myPc/myJndiName", objectRef));`
  - Deployment-Werkzeug des Servers

2. Auffinden entfernter Objekte (1)

2.a) per **Namensdienst**

- Client kennt den im Server registrierten Namen des entfernten Objekts
- Client führt einen „Lookup“ auf dem Server durch und erhält eine Remote-Referenz:

```
IShop remoteShop = (IShop) Naming.lookup("rmi://myPc/shopJNDI");
```

Problem: Client muss den Namen kennen, mit dem ein Objekt registriert ist

- das geht offensichtlich nur für wenige Objekte !
- ggf. schwierig / kaum praktikabel für neu erzeugte Objekte

2. Auffinden entfernter Objekte (2)

2. b) Objekte durch entfernte Methodenaufrufe bereitgestellt

- **Client erhält Objekte** über remote-Methoden als Rückgabe

```
Catalog items = remoteShop.getCatalog();
```

- **Client übergibt Objekte** als Remote-Referenz
  ➔ stehen Server zur Verfügung
  - die Objekte können also direkt dort geändert werden

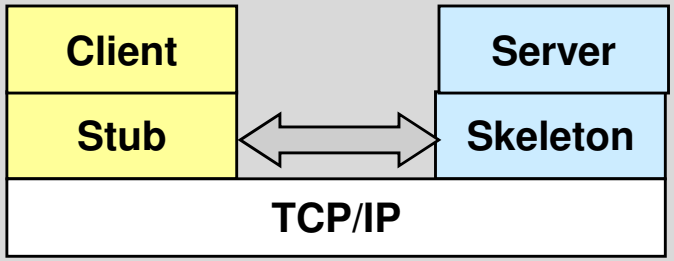
```
Person p = new Person("Susi Sorglos");
remoteShop.buyProduct("1", p);
//Personobjekt wird auf anderem Rechner geändert
```

3. Ausführung entfernter Methodenaufrufe (1)

Übertragung der erforderlichen Informationen an den Server

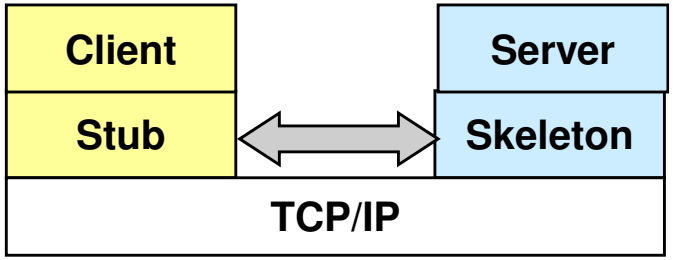
- **Name der Methode**
- Einpacken und Transport der Parameter (**Marshalling**)
- Entgegennahme der Rückgabewerte (**Demarshalling**)

- Marshalling /Demarshalling wird von zwei Klassen übernommen:
  - **Stub**: Stellvertreterobjekt auf Client
  - **Skeleton**: Serverseitiges Objekt



- **Stub und Skeleton werden generiert**
  - Stub-Compiler nutzt die Schnittstellenbeschreibung,
    z.B. `WSDL2java myService.wsdl`

3. Ausführung entfernter Methodenaufrufe (2)



Stub-Klasse

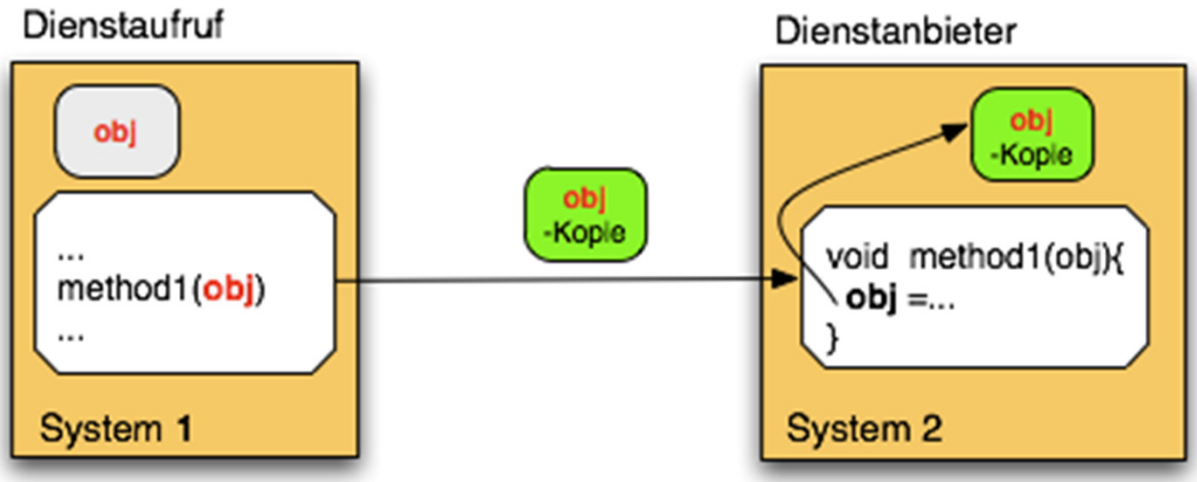
- Stellvertreterobjekt auf Client
  - zuständig für Marshalling/Demarshalling
- baut Socket-Verbindung zu Server auf (siehe Folgekapitel)
  - ähnlich einem I/O-Stream zu anderem Rechner
- schickt Namen der Methode + Parameter und holt Ergebnis ab

Skeleton-Klasse

- wartet über den Socket auf Methoden-Aufruf von Clients
- delegiert den Aufruf an das entsprechende (Java-)Objekt
- liefert Rückgabewert über den Socket

4. Parameter-Übergabe (1) – Wert-Semantik

- es werden **Kopien** der Objekte zwischen Client und Dienstanbieter verschickt
  - POJO (plain old java objects), value objects, Data Transfer Object (DTO)
- funktioniert immer (insbesondere auch bei Web Services, MOM)



Vorteile

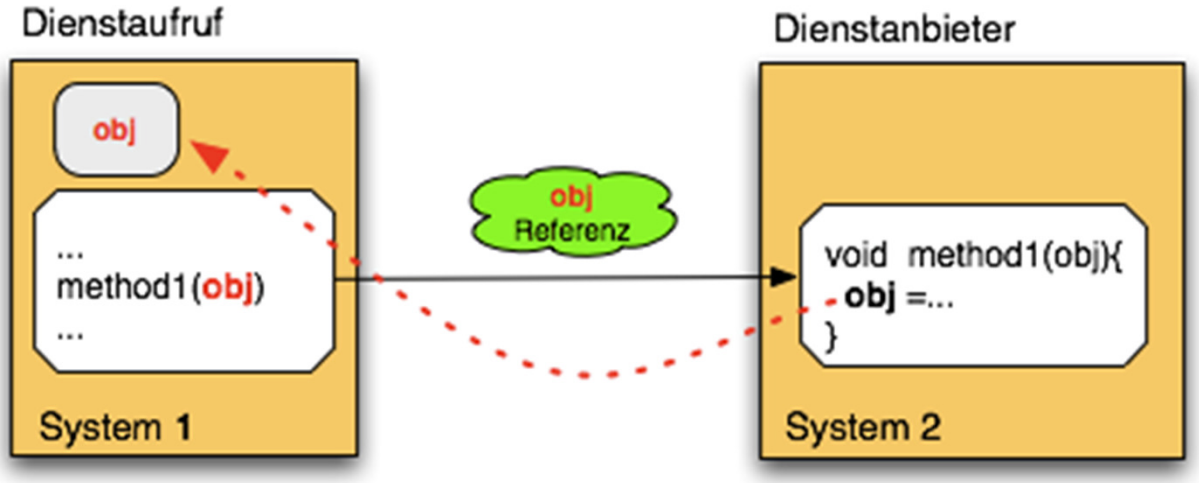
- die Anzahl der remote Zugriffe ist reduziert
- Dienstaufrufer schützt sein Objekt (es kann nicht geändert werden)

Nachteile

- redundante Daten ggf. mit Inkonsistenzen
- Kopier-/ Implementierungsaufwand

4. Parameter-Übergabe (2) – Referenz-Semantik

- dem Dienstanbieter wird eine (Remote) Referenz auf das Parameter-Objekt übertragen
- Remote-Refenzen gibt es nur in entsprechenden Umgebungen (Corba, RMI)



Vorteile

- keine Redundanz
- konsistente Daten

Nachteile

- Remote-Zugriffe sind teuer
- übergebenen Objektes kann von außen geändert werden (trust)
- stärkere Kopplung
- funktioniert nicht bei allen Technologien

Historie entfernter Aufrufe

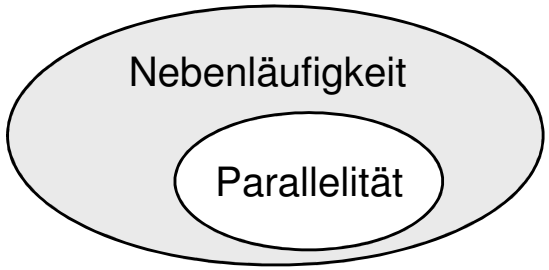
- **Remote Procedure Call (RPC) – 70er-Jahre**
  - prozedurales Programmierparadigma
  - plattform-neutrales Übertragungsformat: (XDR - External Data Representation, Sun)
- **OSF/DCE (Distributed Computing Environment) – 1990**
  - Middleware für RPC:
  - RPC mit: Sicherheits,- Namens- und Zeitdienst, Threads, verteiltes Dateisystem
- **CORBA / RMI – (1989/1992)**
  - Übertragung der RPC-Idee auf Objektorientierung
  - Unterschiede zu RPC: logische Referenzen, Garbage Collection
- **SOAP/WSDL Web Services – 1999**
  - Dienstbeschreibung, Aufruf und Datentransport über XML
- **REST – zunehmend populärer ab ca. 2005** (entstanden 2000)
- Google RPC, Doors, …
- und weitere....

Nebenläufigkeit (concurrency)

- es gibt **mehrere gleichzeitige Prozesse**
- die Prozess sind (weitgehend) voneinander **unabhängig**
- die Prozess erfordern ggf. eine **Synchronisation**
  - konkurrieren ggf. um gemeinsame Ressourcen
  - expliziter Austausch von Informationen
  - Synchronisation = wohldefinierte zeitliche Abfolge erzwingen

Parallelität vs. Nebenläufigkeit

- **Parallelität**: gleichzeitige Ablauf
  - erfordert mehrere CPUs
- **Nebenläufigkeit**: paralleler oder verzahnter Ablauf
  - Umschaltung zwischen mehreren Threads (1 CPU)



Nebenläufige Prozesse auf verschiedenen Ebenen

- im **Betriebssystem**
  - Ressourcen werden von mehreren Prozessen genutzt: Drucker, Platten,...
  - (pseudo-)parallele Ausführung mehrerer Programme auf einem Rechner
- **innerhalb eines Programms**
  - (pseudo-)parallele Bearbeitung unabhängiger Probleme mittels Threads
  - z.B. Computerspiel: paralleler Ablauf von Audio, Einfangen von GUI-Ereignissen
- **in verteiltem System**
  - Programmcode wird auf verschiedenen Rechnern ausgeführt
  - Problem: parallelisierbare Teilaufgaben finden

Prozesse – Eigenschaften

- besitzen **vollständigen Ausführungskontext**
  - eigenen Zustand: Programmzähler, Register, eigene Variablen
  - eigenen Adressraum
- werden **auf Betriebssystem-Ebene ausgeführt**
  - es ist zeitaufwändig zwischen Prozessen zu wechseln
- **kommunizieren** miteinander **mithilfe von Interprozess-Kommunikation**
  - können nicht über gemeinsamen Adressraum kommunizieren
  - Sockets, RMI, CORBA, SOAP, Messages, ...
- **besitzen ggf. mehrere Threads**

Threads – Eigenschaften

- **implementieren Nebenläufigkeit innerhalb eines Programms**
  - quasi-parallele Abläufe, bspw. Musik u. Videoclip in Computerspiel
  - Reaktion auf Benutzereingaben bei gleichzeitiger Verarbeitung
- **leichtgewichtige** Prozesse (lightweight processes)
  - gemäß Zeitscheibenverfahren wird CPU-Zeit zugeteilt und entzogen
  - Threads teilen sich Daten, Code und Betriebsmitte
  - gemeinsamer Adressraum
- **Kommunikation über gemeinsamen Adressraum**
  - Threads können auf gemeinsame Objekte zugreifen
  - Mechanismen zur Synchronisation: Threads starten, anhalten, aufwecken, beenden,...
- **in Programmiersprachen** unterstützt
  - durch spezielle Bibliotheken (Java, C++,..)

Konzepte entfernter Aufrufe - Fragenblock

- **Erläutern Sie den allgemeinen Ablauf eines entfernten Methodenaufrufs.**
- **Erläutern Sie das Stub/Skeleton-Modell.**
- **Erklären Sie Parallelität & Nebenläufigkeit.**

Server / Handler-Muster

Problem:

- Server bietet Dienste an
- Clients fordern (häufig) Dienste an, die vom Server ausgeführt werden
- Server kann neue Anforderungen nicht annehmen, weil er beschäftigt ist

Ziel: Server soll mehrere Aufgaben gleichzeitig annehmen können und erreichbar bleiben

Lösung: **Server delegiert Aufgabe an eigene Klasse (= Handler)**

- Server erzeugt für jede neue Anforderung ein eigenes Handler-Objekt
- Handler läuft in eigenem Thread und bearbeitet die eigentliche Anfrage
- **Server** benötigt nur kurze Zeit zum Erzeugen des Handler und **steht sofort wieder zur Verfügung**

3.5.3 Nebenläufigkeit in verteilten Systemen – Muster: *Server* / *Handler*

```
public class Server extends Thread {

   public Server() {
           this.start();
   }

   public void run() {
        String job="";
        while (true) {
            System.out.println("...Server waiting for new task");
            try {
                job = new BufferedReader(
                                new InputStreamReader(System.in).readLine();
             } catch (IOException e) { ... }
            Handler handler = new Handler(job);
         }
   }

   public static void main(String[] args) {
         new Server();
   }
}
```

```
public class Handler extends Thread {
    String job;
    public Handler(String job) {
        this.job = job;
        this.start();
    }

    public void run() {
        System.out.println("Handler is doing some work for ..." + job);
        try {
            Thread.sleep(1000); // hier könnte eine sinnvolle Aufgaben gelöst werden
        } catch (InterruptedException exc) {  ...   }
        System.out.println("...Handler has finished the work ");
    }
}
```

Asynchrone Aufrufe

synchrone Aufrufe

- Client ruft Dienst auf und wartet bis dieser beendet ist, d.h. der Client ist zunächst blockiert

asynchrone Aufrufe

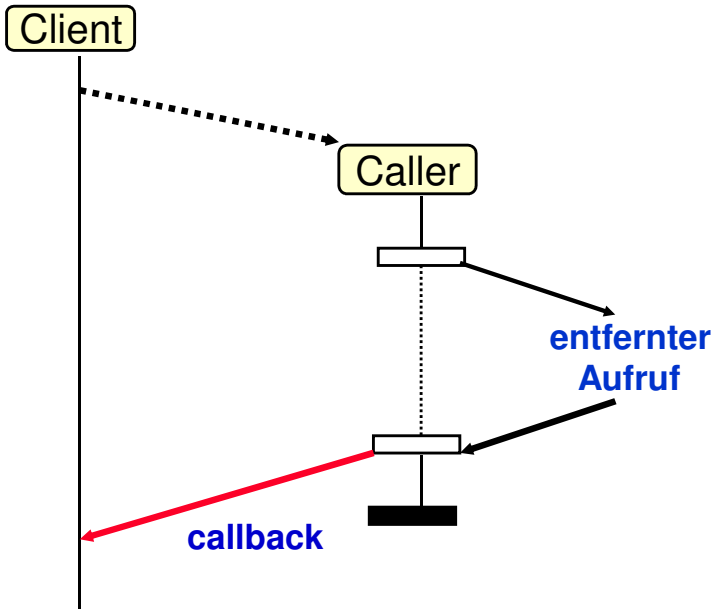
- Client setzt nach Dienstaufruf seine Aufgaben unmittelbar fort
  - ermöglicht parallele Ausführung in verteilten Systemen (=bessere Performanz)
  - insbesondere keine Verzögerungen durch hohe Übertragungszeiten
- in Java: alle Methodenaufrufe sind synchron (auch bei RMI)
  - Asynchronität muss selbst implementiert werden (mithilfe von Threads)

Idee: eigenen Thread erzeugen, der Methode nebenläufig zu Hauptaktivitätsstrang ausführt

Problem: wie erhält Hauptaktivitätsstrang den Rückgabewert ?

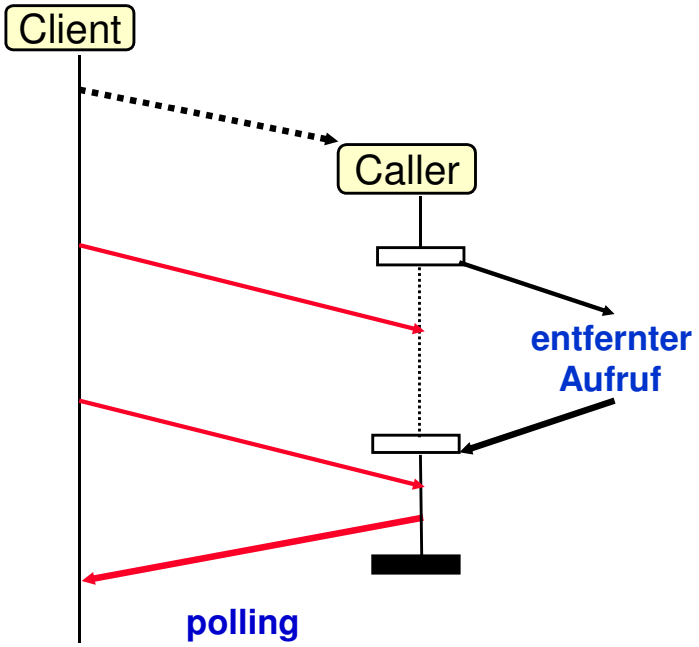
3.5.3 Nebenläufigkeit in verteilten Systemen – *Asynchrone Aufrufe*

- Caller-Objekt führt asynchronen Aufruf in eigenem Thread durch



A. Callback-Verfahren

→ *Caller* ruft callback-Methode des *Client* auf,falls Ergebnis vorliegt



B. Polling

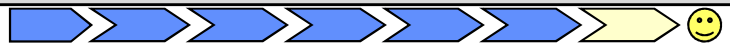
→ *Client* fragt laufend *Caller*, ob Ergebnis vorliegt

→ ohne Ergebnis bearbeitet Client andere Aufgaben

3.5.3 Nebenläufigkeit in verteilten Systemen – asynchrone Aufrufe: Polling

```
public class PollingClient {
    public static void main(String[] args) {
        PollingCaller caller = new PollingCaller();

           while(caller.returned()==false) {   //polling!!
            System.out.print("." ); // hier könnte etwas sinnvolles gemacht werden
             try {
                Thread.sleep(1000);
             } catch (InterruptedException e) {e.printStackTrace();}
         }
         System.out.println(" Ergebnis vom asynchronen Aufruf ");

   }
```

```
public class PollingCaller extends Thread {
        private boolean returned = false;

        public PollingCaller () { this.start(); }

   public void run() {
           System.out.println(" entfernter Aufruf läuft u. wartet auf Taste ");
           try {
               System.in.read();
            } catch (IOException e) { e.printStackTrace();    }
            returned = true;
       }
      public boolean returned() {    return returned;   }

}
```

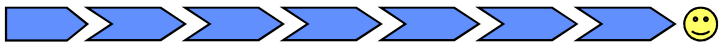
3.5.4 Nebenläufigkeit in verteilten Systemen – *asynchrone Aufrufe: Callback*

```
public class CallbackClient {
    public CallbackClient() {
      boolean stop = false;
      CallbackCaller caller = new CallbackCaller(this);      // delegiert asynchronen Aufruf
           while (!stop){
               System.out.print(".");
               try {
                      Thread.sleep(1000);
               } catch (InterruptedException e) { e.printStackTrace();
           }
       }
    }
    public void callback() {
       System.out.println(" Ergebnis vom asynchr. Aufruf"); stop = true;
    }
    public static void main(String[] args) { new CallbackClient();  }
}
```

```
public class CallbackCaller extends Thread {          // führt asynchronen Aufruf aus
       CallbackClient client;
       public CallbackCaller (CallbackClient c) { client = c; this.start();  }
    public void run() {       // Hier könnte ein entfernter Aufruf stehen
          System.out.println(" entfernter Aufruf läuft u. wartet auf return  ");
          try {
             System.in.read();
          } catch (IOException e) { e.printStackTrace(); }
        client.callback();
       }
}
```


Verteilte Software-Architekturen (1)

- Verteilte Systeme – Definitionen und Bedeutung
- Middleware als Infrastruktur für verteilte Systeme
- Dimensionen verteilter Systeme
  - Nebenläufigkeit, Verteilung, Persistenz
- Anforderungen an verteilte Systeme
  - Kommunikationsmechanismen, Netztransparenz, inhärente Nebenläufigkeit



Verteilte Software-Architekturen (2)

- Konzepte entfernter Aufrufe
  - genereller Ablauf
  - Parameterübergabe
- Nebenläufigkeit
  - Definitionen
  - Eigenschaften
  - Nebenläufigkeit in verteilten Systemen - Muster
    - Server-Handler-Muster
    - Muster: Asynchrone Aufrufe

Literatur

- M. Boger: Java in verteilten Systemen, Dpunkt Verlag, 1999
- Rainer Oechsle: Parallele und verteilte Anwendungen in Java, Hanser, 2. Aufl., 2007.
- Jürgen Dunkel, Andreas Eberhart, Stefan Fischer, Carsten Kleiner, Arne Koschel: Systemarchitekturen für Verteilte Anwendungen - Client-Server, Multi-Tier, SOA, EDA, Grid, P2P, …, Hanser, 2008.