
Client/Server-Programmierung

WS 2019/2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. Januar 2020

Client/Server-Programmierung

WS 2019/2020

1 Grundlagen: Wiederholung

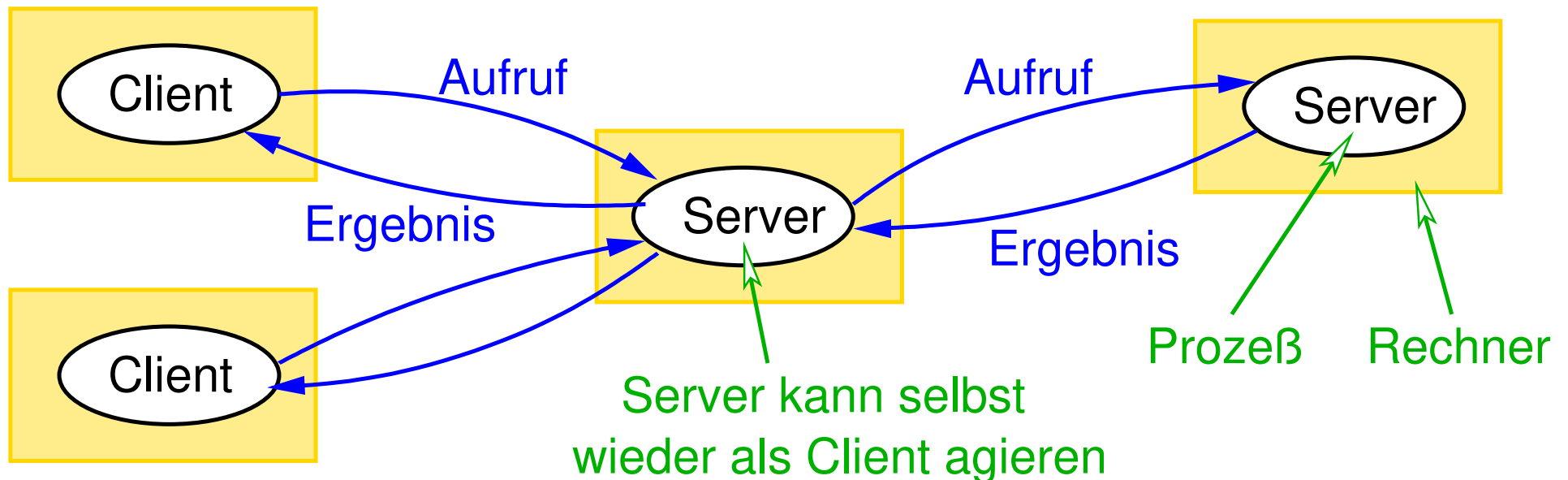


Inhalt

- ➔ Architekturmodelle
- ➔ Zeit und Zustand in verteilten Systemen
- ➔ Middleware
- ➔ Java RMI

Client/Server-Modell

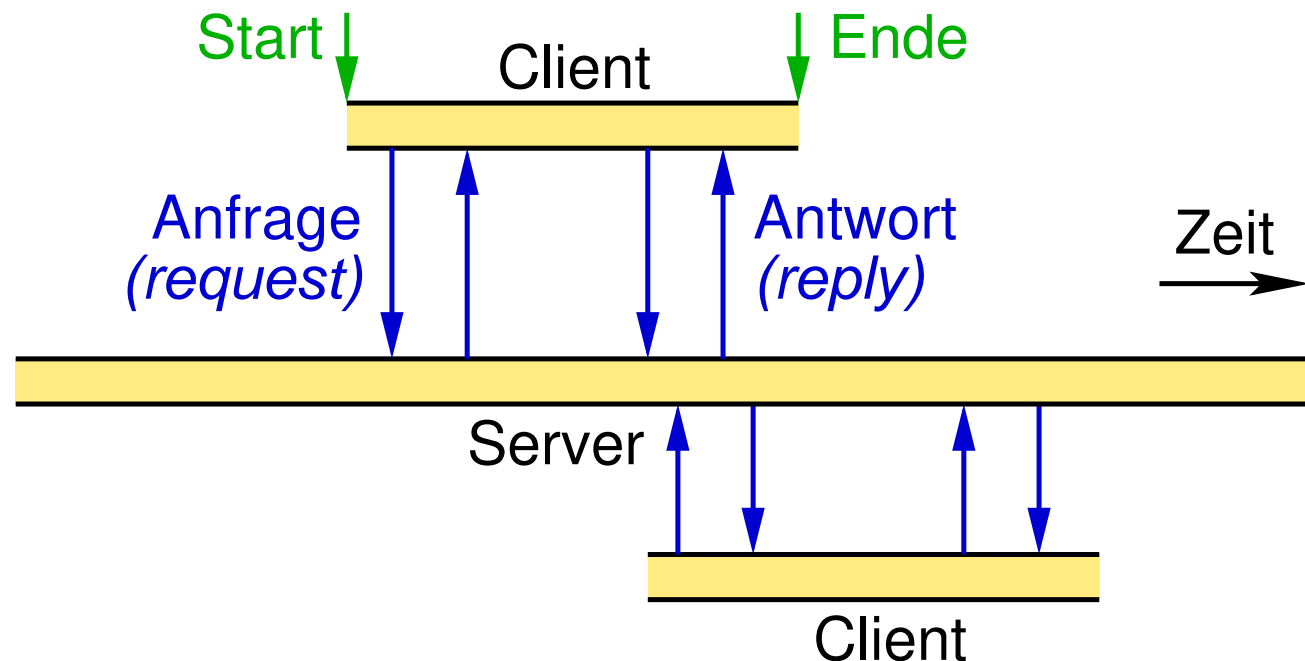
- ➔ Asymmetrisches Modell: Server stellen Dienste bereit, die von (mehreren) Clients genutzt werden können
- ➔ Server verwalten i.a. Ressourcen (zentralisiert)



- ➔ Häufigstes Modell für verteilte Anwendungen (ca. 80 %)

Client/Server-Modell ...

- ➔ I.A. nebenläufige Anfragen mehrerer Client-Prozesse an den Server-Prozeß



- ➔ Beispiele: Dateiserver, WWW-Server, DB-Server, DNS-Server, ...

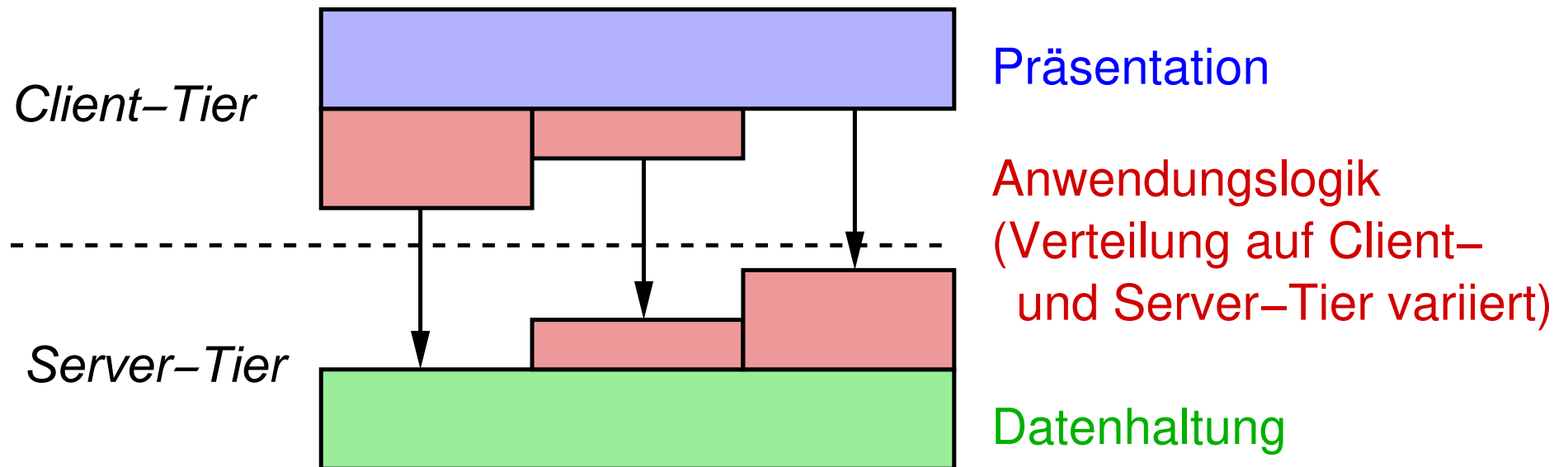


n-Tier-Architekturen

- ➔ Verfeinerungen der Client/Server-Architektur
- ➔ Modelle zur Verteilung einer Anwendung auf die Knoten einer verteilten Systems
- ➔ Vor allem bei Informationssystemen verwendet
- ➔ **Tier** (engl. Schicht / Stufe) kennzeichnet einen unabhängigen Prozeßraum innerhalb einer verteilten Anwendung
 - ➔ Prozeßraum kann, muß aber nicht physischem Rechner entsprechen
 - ➔ mehrere Prozeßräume auf einem Rechner möglich

2-Tier-Architektur

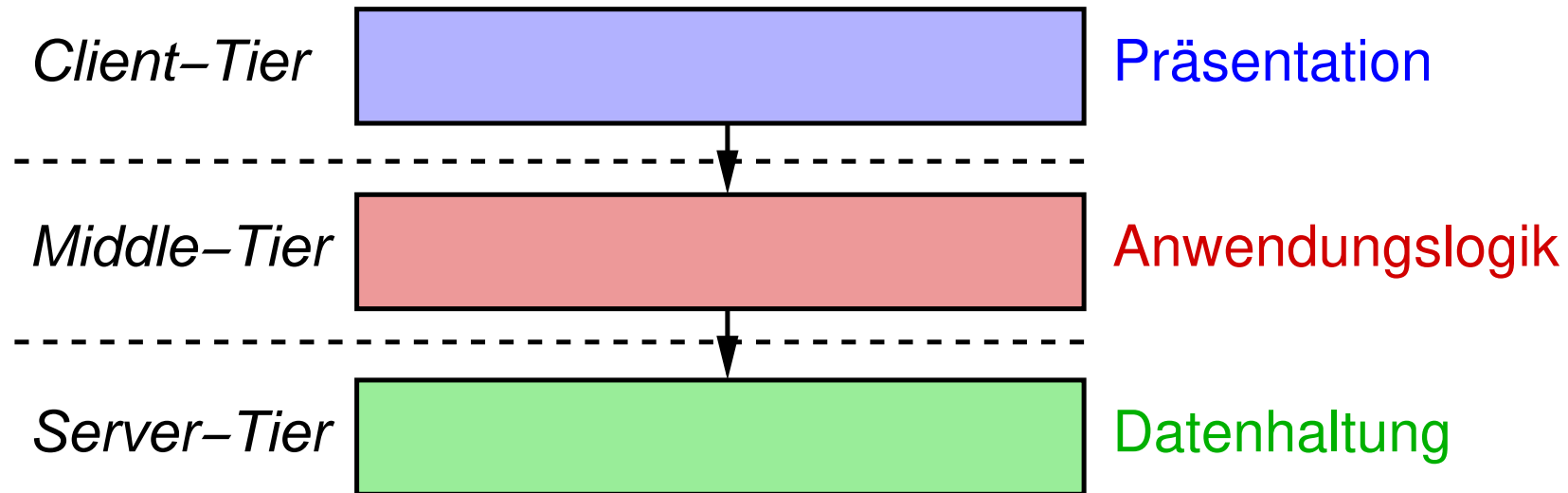
- ➔ Client- und Server-Tier
- ➔ Keine eigene Tier für die Anwendungslogik



- ➔ Vorteil: einfach, performant
- ➔ Nachteil: schwer wartbar, schlecht skalierbar

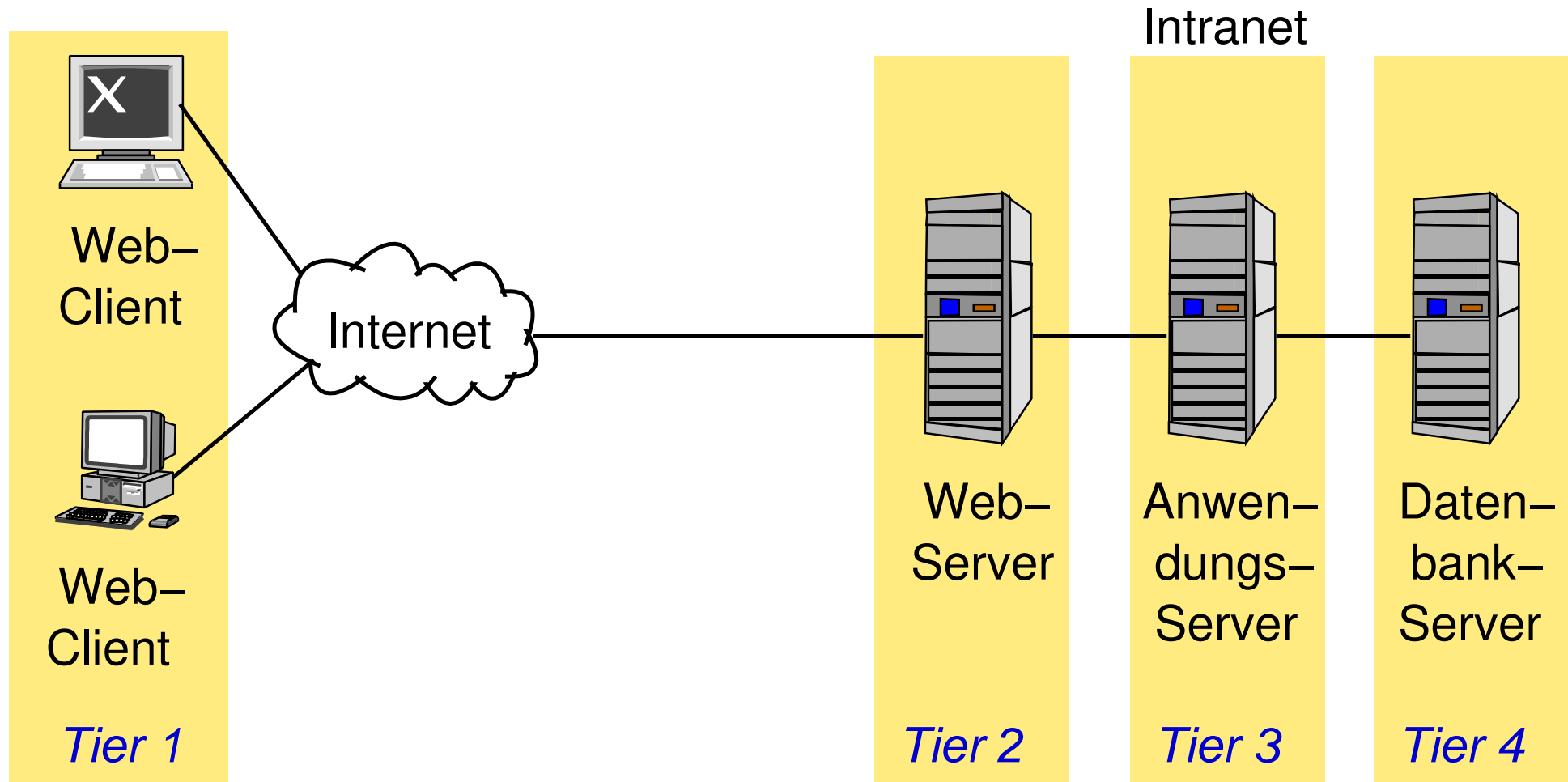


3-Tier-Architektur

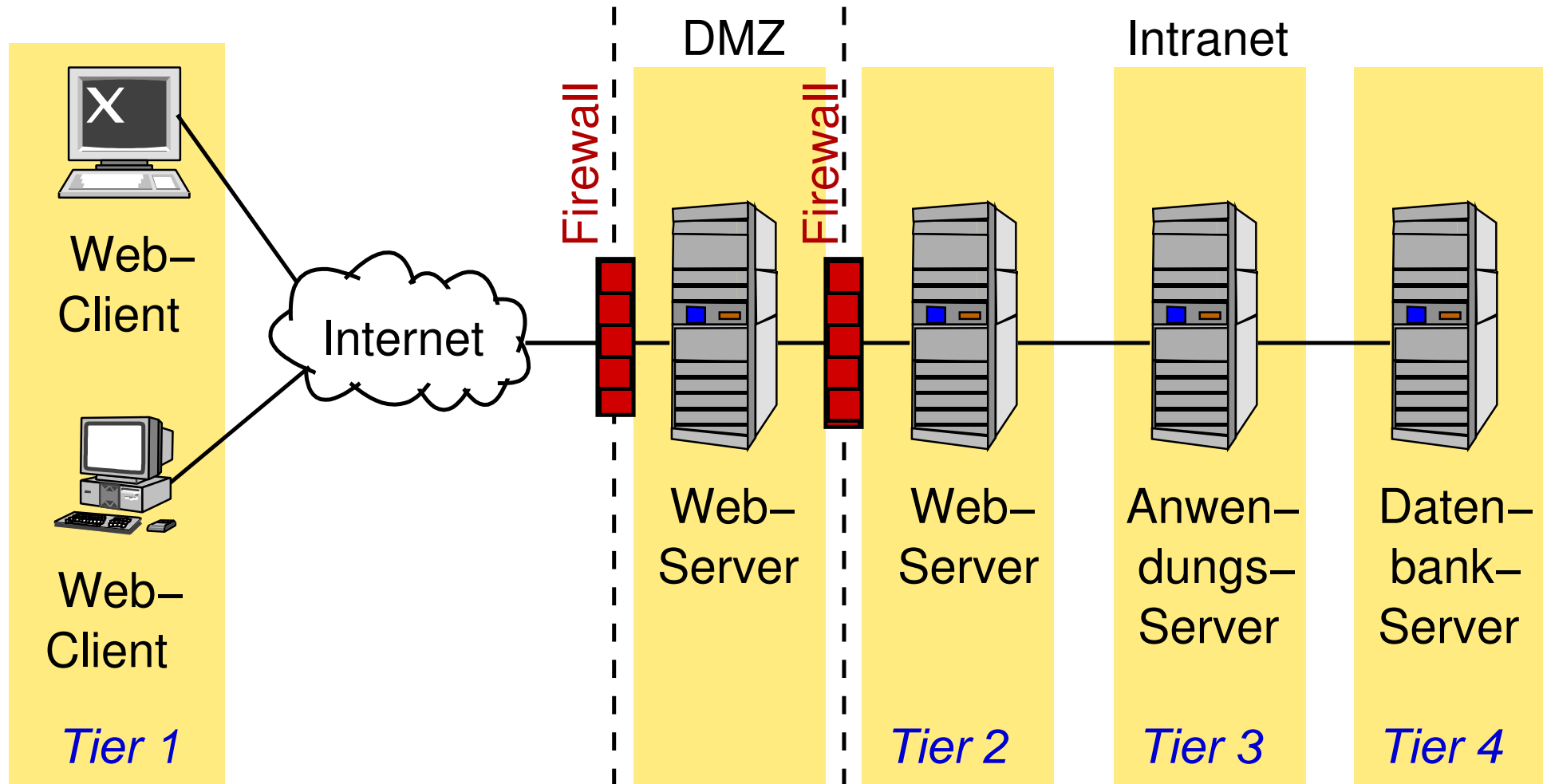


- ➔ Standard-Verteilungsmodell für einfache Web-Anwendungen:
 - ➔ *Client-Tier*: Web-Browser zur Anzeige
 - ➔ *Middle-Tier*: Web-Server mit Servlets / JSP / ASP
 - ➔ *Server-Tier*: Datenbank-Server
- ➔ Vorteile: Anwendungslogik zentral administrierbar, skalierbar

Beispiel: typische Internet-Anwendung



Beispiel: typische Internet-Anwendung





Was ist der Unterschied zwischen einem verteilten System und einem Ein-/Mehrprozessorsystem?

- ➔ Ein- bzw. Mehrprozessorsystem:
 - ➔ nebenläufige Prozesse: pseudo-parallel durch *time sharing* bzw. echt parallel
 - ➔ globale Zeit: alle Ereignisse in den Prozessen lassen sich zeitlich eindeutig ordnen
 - ➔ globaler Zustand: zur jeder Zeit kann ein eindeutiger Zustand des Systems angegeben werden
- ➔ Verteiltes System
 - ➔ echte Parallelität
 - ➔ keine globale Zeit
 - ➔ kein eindeutiger globaler Zustand



Globale Zeit

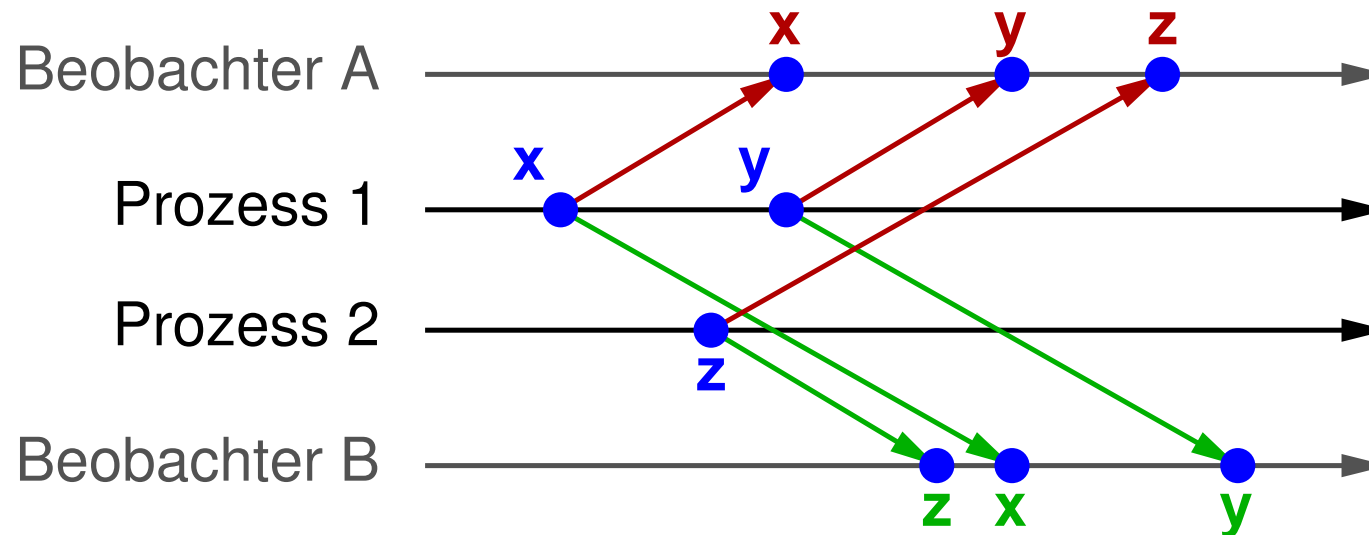
- ➔ Auf Ein-/Mehrprozessorsystem
 - ➔ jedem Ereignis kann (zumindest theoretisch) ein eindeutiger Zeitstempel derselben lokalen Uhr zugeordnet werden
 - ➔ bei Mehrprozessorsystemen: Synchronisation am gemeinsamen Speicher

- ➔ In verteilten Systemen:
 - ➔ viele lokale Uhren (eine pro Knoten)
 - ➔ exakte Synchronisation der Uhren (prinzipiell!) nicht möglich
 - ➔ \Rightarrow Reihenfolge von Ereignissen auf verschiedenen Knoten nicht (immer) eindeutig zu ermitteln
 - ➔ (vgl. spezielle Relativitätstheorie)



Eine Auswirkung der Verteiltheit

➔ Szenario: zwei Prozesse beobachten zwei andere Prozesse



- ➔ Die Beobachter sehen die Ereignisse ggf. in unterschiedlicher Reihenfolge!
- ➔ Problem z.B., falls die Beobachter replizierte Datenbanken und die Ereignisse Datenbank-Updates sind
 - ➔ Replikate sind nicht mehr konsistent!



Globaler Zustand: Ein Beispiel zur Motivation

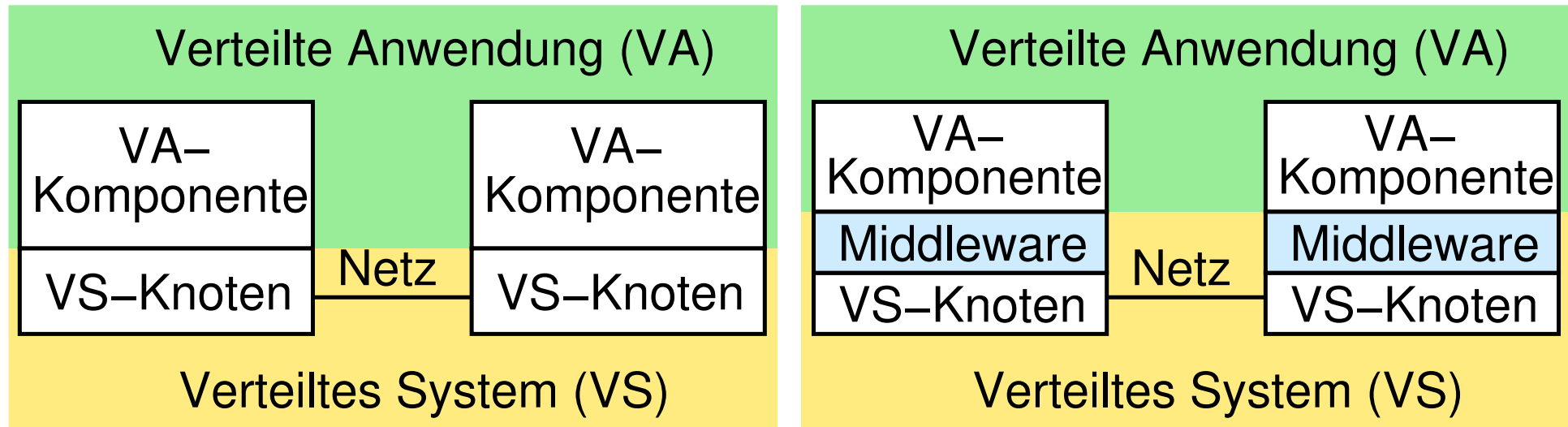
- ➔ Szenario: *Peer-to-Peer*-Anwendung, Prozesse senden sich gegenseitig Aufträge
- ➔ Frage: wann kann die Anwendung terminieren?
- ➔ **Falsche** Antwort: wenn kein Prozeß mehr einen Auftrag bearbeitet
 - ➔ Grund: Aufträge können noch in Nachrichten unterwegs sein!



- ➔ Weitere Anwendungen: verteilte *Garbage-Collection*, verteilte *Deadlock*-Erkennung, ...



- ➔ Wie bestimmt sich der Gesamtzustand eines verteilten Prozeßsystems?
 - ➔ naiv: Summe der Zustände aller Prozesse (**falsch!**)
- ➔ Zwei Aspekte müssen beachtet werden:
 - ➔ Nachrichten, die noch in Übertragung sind
 - ➔ müssen mit in den Zustand aufgenommen werden
 - ➔ Fehlen einer globalen Zeit
 - ➔ ein Globalzustand zur Zeit t kann nicht definiert werden!
 - ➔ Zustände der Prozesse beziehen sich immer auf lokale (und damit unterschiedliche) Zeiten
 - ➔ Frage: Bedingung an die lokalen Zeiten? ⇒ **konsistente Schnitte**



- ➔ VA nutzt VS für Kommunikation zwischen ihren Komponenten
- ➔ VSe bieten i.a. nur einfache Kommunikationsdienste an
 - ➔ direkte Nutzung: **Netzwerkprogrammierung**
- ➔ **Middleware** bietet intelligentere Schnittstellen
 - ➔ verbirgt Details der Netzwerkprogrammierung

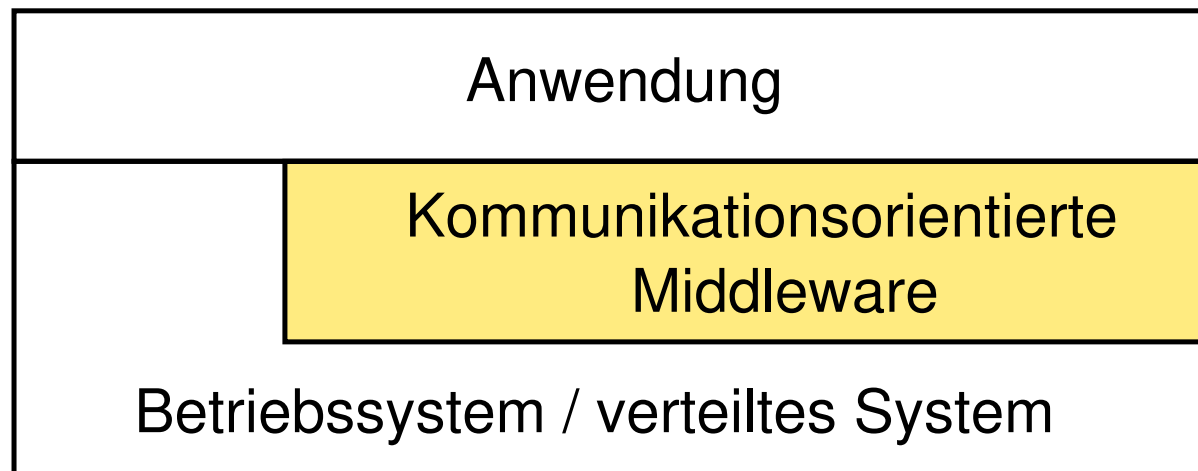


- ➔ Middleware ist Schnittstelle zwischen verteilter Anwendung und verteiltem System
- ➔ Ziel: Verbergen der Verteilungsaspekte vor der Anwendung
 - ➔ u.a. Zugriffs- und Orts-Transparenz
- ➔ Middleware kann auch Zusatzdienste für Anwendungen bieten
 - ➔ starke Unterschiede bei existierender Middleware
- ➔ Unterscheidung:
 - ➔ **kommunikationsorientierte Middleware**
 - ➔ (nur) Abstraktion von der Netzwerkprogrammierung
 - ➔ **anwendungsorientierte Middleware**
 - ➔ neben Kommunikation steht Unterstützung verteilter Anwendungen im Mittelpunkt

1.3.1 Kommunikationsorientierte Middleware

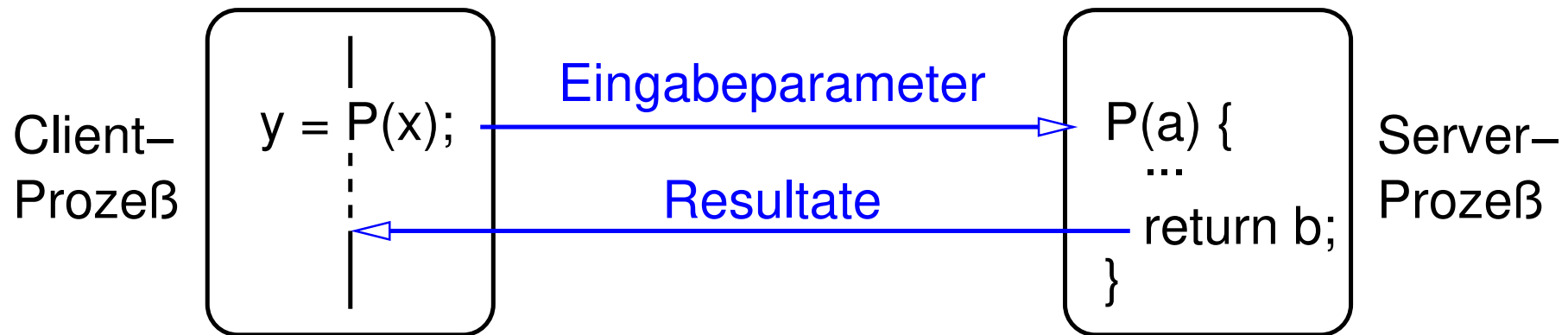


- ➔ Fokus: Bereitstellung einer Kommunikationsinfrastruktur für verteilte Anwendungen
- ➔ Aufgaben:
 - ➔ Kommunikation
 - ➔ Behandlung der Heterogenität
 - ➔ Fehlerbehandlung



Entfernter Prozeduraufruf (RPC, *Remote Procedure Call*)

- ➔ Ermöglicht einem Client den Aufruf einer Prozedur in einem entfernten Server-Prozeß



- ➔ Kommunikation nach Anfrage / Antwort-Prinzip

Entfernter Methodenaufruf (RMI, *Remote Method Invocation*)

- ➔ Ermöglicht einem Objekt, Methoden eines entfernten Objekts aufzurufen
- ➔ Prinzipiell sehr ähnlich zu RPC

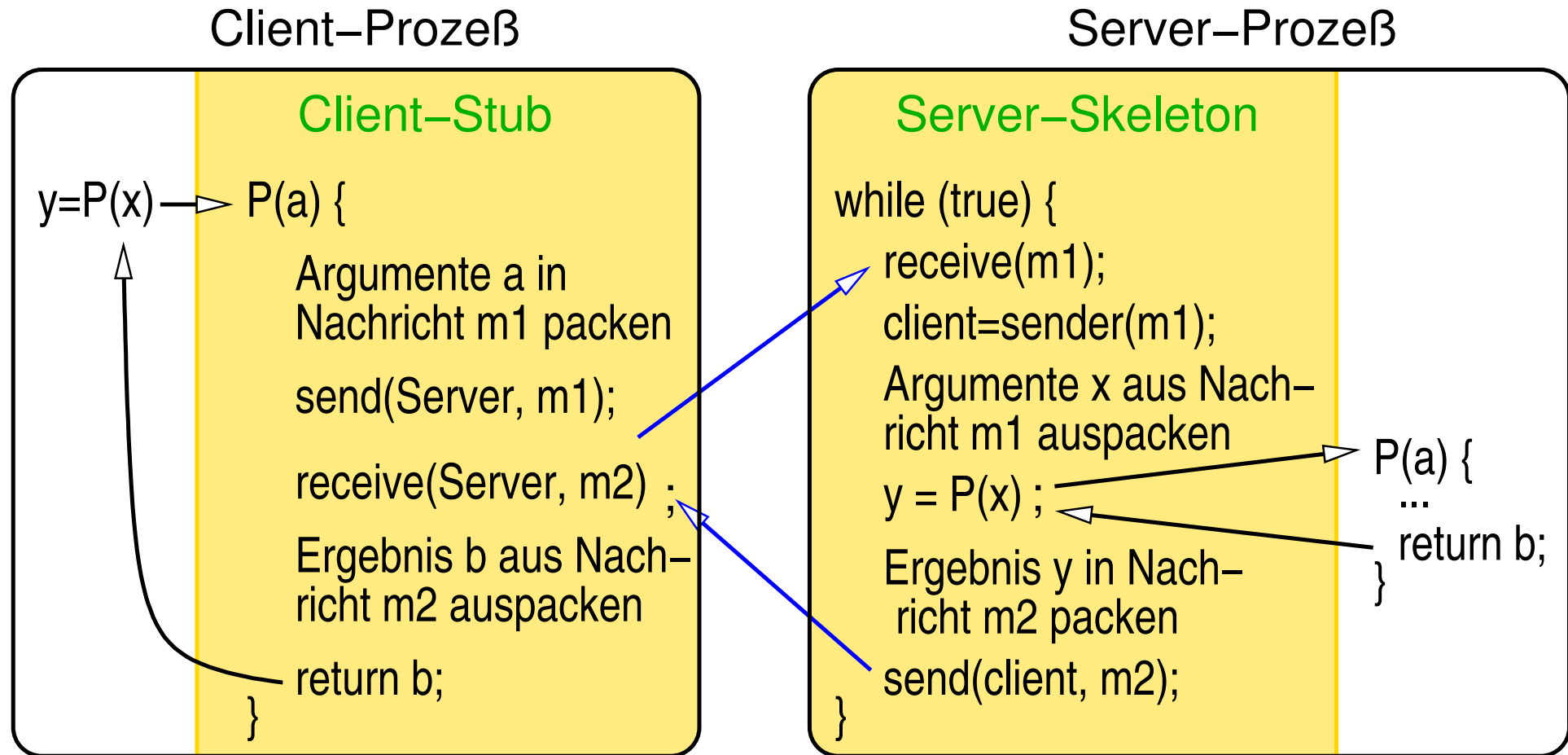


Gemeinsame Grundkonzepte entfernter Aufrufe

- ➔ Client und Server werden durch Schnittstellendefinition entkoppelt
 - ➔ legt Namen der Aufrufe, Parameter und Rückgabewerte fest
- ➔ Einführung von **Client-Stubs** und **Server-Skeletons** als Zugriffsschnittstelle
 - ➔ werden automatisch aus Schnittstellendefinition generiert
 - ➔ IDL-Compiler, *Interface Definition Language*
 - ➔ sind verantwortlich für *Marshalling / Unmarshalling* sowie für die eigentliche Kommunikation
 - ➔ realisieren Zugriffs- und Ortstransparenz



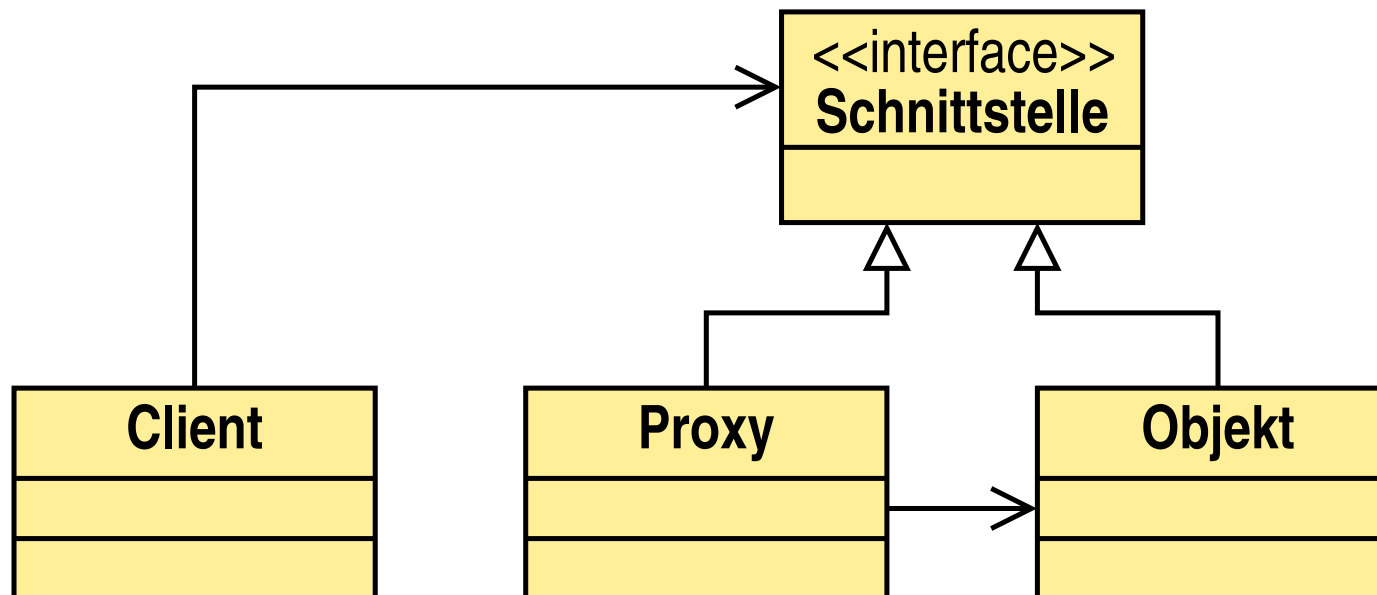
Funktionsweise der Client- und Server-Stubs (RPC)





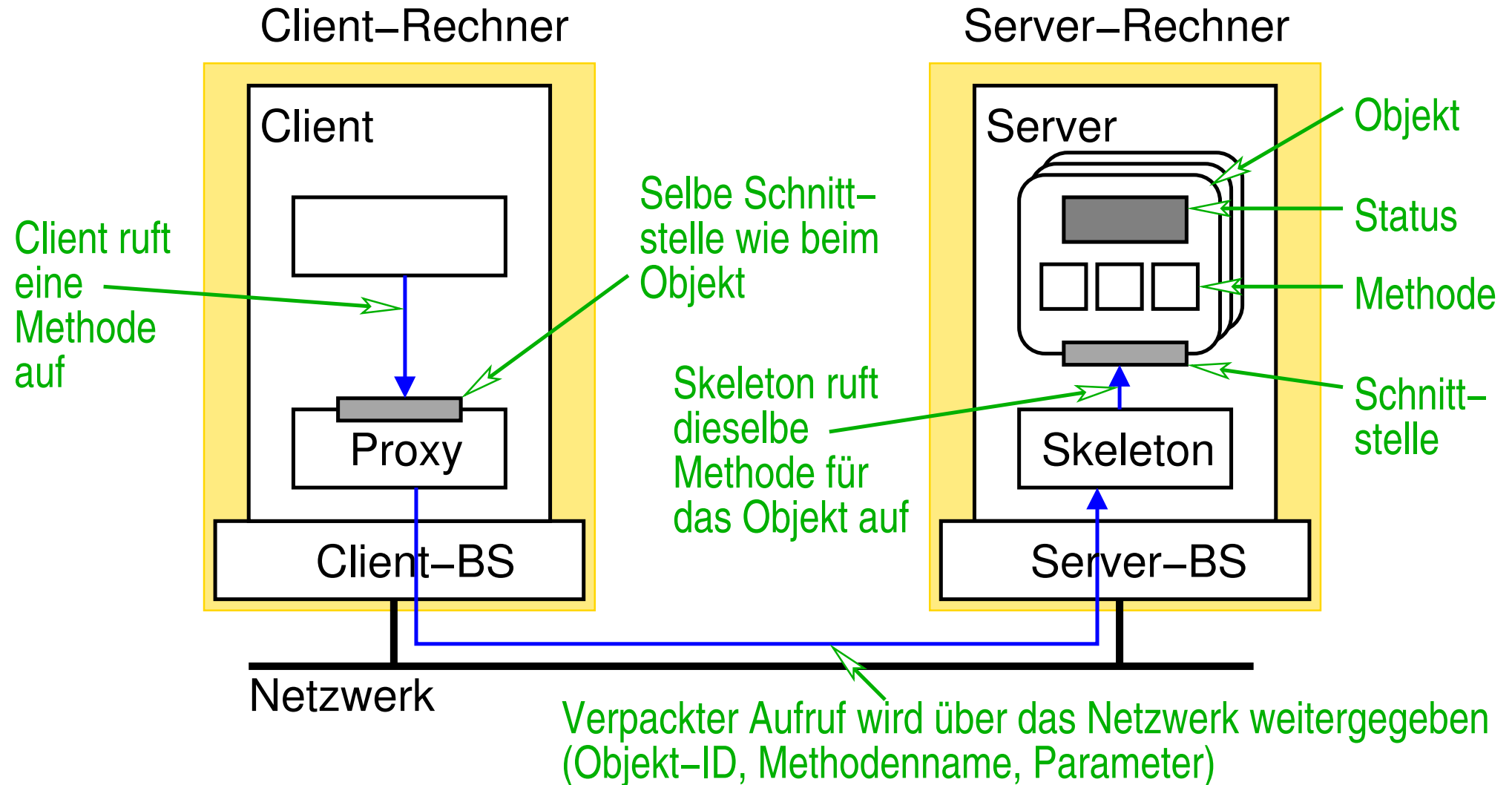
Basis von RMI: Das Proxy-Pattern

- ➔ Client arbeitet mit Stellvertreterobjekt (**Proxy**) des eigentlichen Serverobjekts
- ➔ Proxy und Serverobjekt implementieren dieselbe Schnittstelle
- ➔ Client kennt / nutzt lediglich diese Schnittstelle



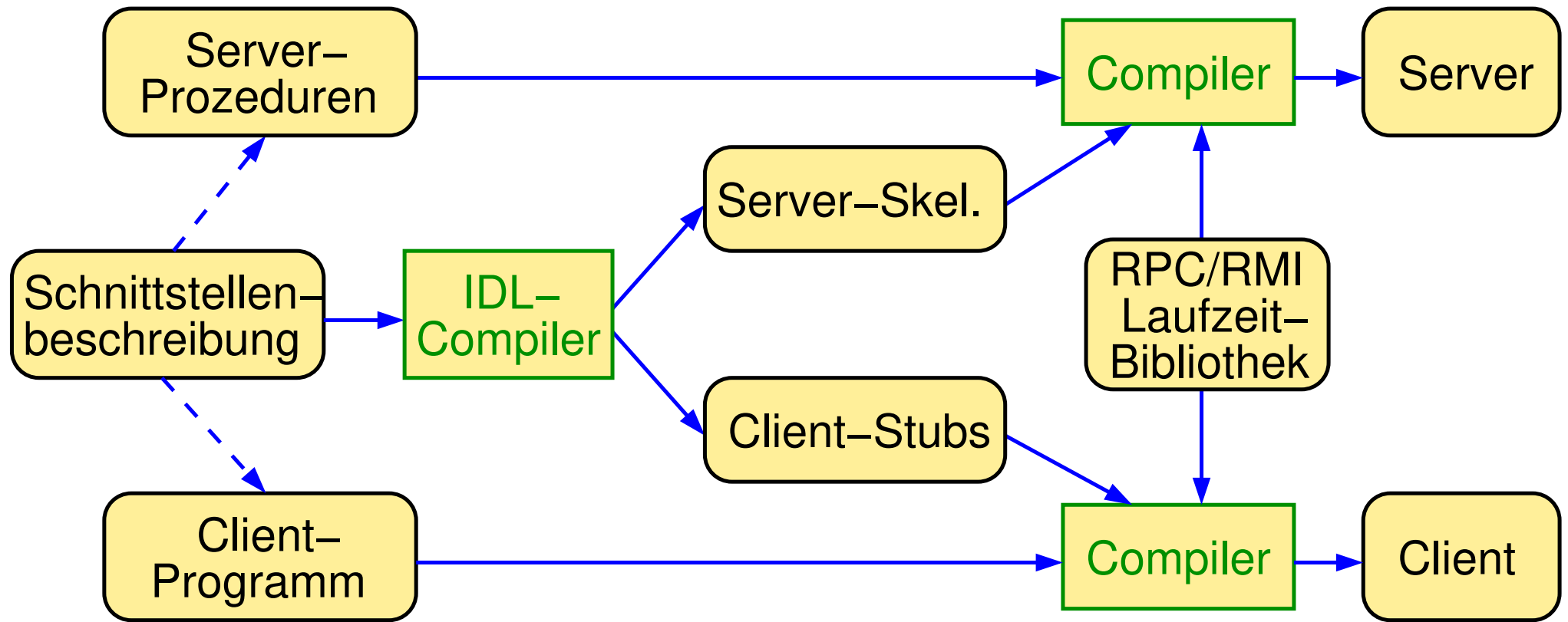


Ablauf eines entfernten Methodenaufrufs





Erstellung eines Client/Server-Programms

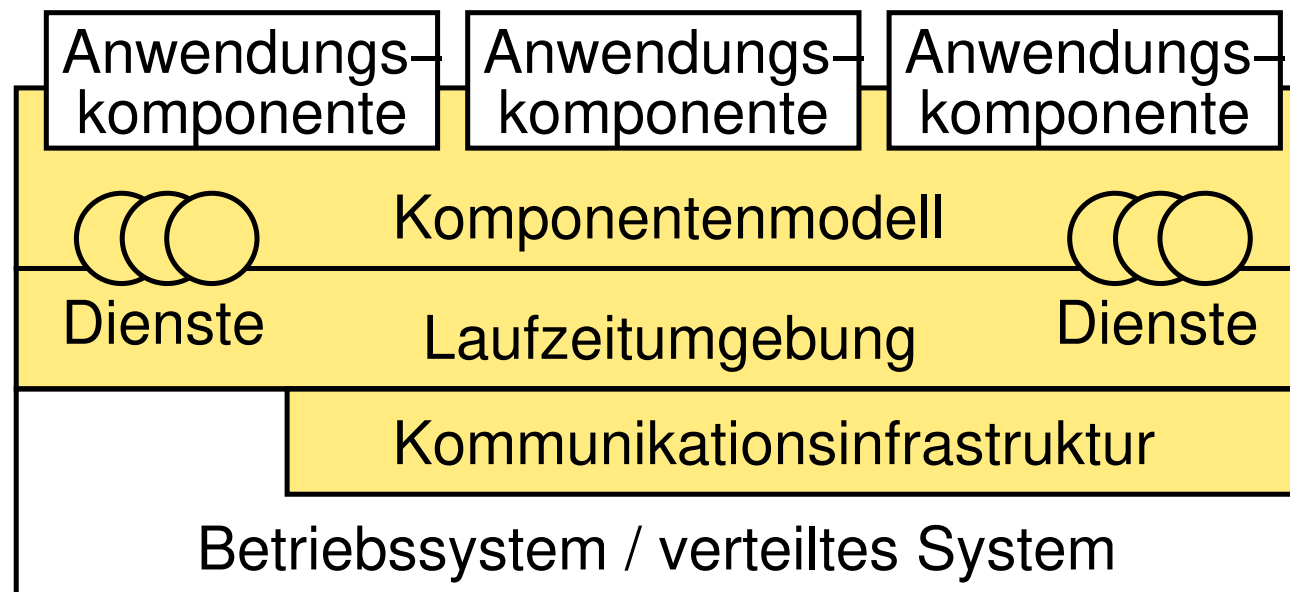


➔ Gilt prinzipiell für alle Realisierungen entfernten Aufrufe

1.3.2 Anwendungsorientierte Middleware



- ➔ Setzt auf kommunikationsorientierter Middleware auf
- ➔ Erweitert diese um:
 - ➔ Laufzeitumgebung
 - ➔ Dienste
 - ➔ Komponentenmodell





Laufzeitumgebung

- ➔ Ressourcenverwaltung
 - ➔ *Pooling* von Prozessen, Threads, Verbindungen
 - ➔ Steuerung der Nebenläufigkeit
 - ➔ Verbindungsverwaltung
- ➔ Verbesserung der Verfügbarkeit
 - ➔ Replikation, Clustering
- ➔ Sicherheitsmechanismen
 - ➔ Authentifizierung und Autorisierung
 - ➔ Vertraulichkeit und Integrität



Dienste

- ➔ Namensdienst (Verzeichnisdienst)
 - ➔ Zuordnung von Namen zu Referenzen (Adressen)
- ➔ Sitzungsverwaltung
- ➔ Transaktionsverwaltung
- ➔ Persistenzdienst
 - ➔ z.B. objektrelationaler Mapper (OR-Mapper)

Komponentenmodell

- ➔ Komponentenbegriff, Schnittstellenverträge, Laufzeitumgebung

Client/Server-Programmierung

WS 2019/2020

14.10.2019

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. Januar 2020



- ➔ Java RMI ist fester Bestandteil von Java
 - ➔ erlaubt Nutzung entfernter Objekte
- ➔ Wichtige Elemente von Java RMI (im Paket `java.rmi`):
 - ➔ entfernte Objektimplementierungen
 - ➔ Client-Schnittstellen (Stubs) zu entfernten Objekten
 - ➔ Namensdienst, um Objekte im Netz auffindig zu machen
- ➔ Stub- und Skeleton-Klassen werden automatisch aus Schnittstellendefinition (Java Interface) generiert
 - ➔ ab JDK 1.5 dynamisch zur Laufzeit
- ➔ Namensdienst: RMI *Registry*
- ➔ Verteilte *Garbage-Collection*



1.4.1 Hello World mit Java RMI

Client-JVM

Interface

```
interface Hello {  
    String sayHello();  
}
```

Server-JVM

Client-Klasse

```
class HelloClient {  
    ...  
    Hello h;  
    ...  
    s = h.sayHello();  
    ...  
}
```

Server-Klasse

```
class HelloServer  
    implements Hello {  
    String sayHello() {  
        return "Hello World";  
    }  
    ...  
}
```



Ablauf der Entwicklung:

1. Entwurf der Schnittstelle für das Server-Objekt
2. Implementierung der Server-Klasse
3. Entwicklung der Server-Anwendung zur Aufnahme des Server-Objekts
4. Entwicklung der Client-Anwendung mit Aufrufen des Server-Objekts
5. Übersetzen und Starten des Systems



Entwurf der Schnittstelle für das Server-Objekt

- ➔ Wird als normale Java-Schnittstelle spezifiziert
- ➔ Muß von `java.rmi.Remote` abgeleitet werden
 - ➔ kein Erben von Operationen, nur Markierung als *Remote-Interface*
- ➔ Jede Methode muß die Ausnahme *java.rmi.RemoteException* (oder eine Basisklasse davon) auslösen können
 - ➔ Basisklasse für alle möglicherweise auftretenden Fehler
 - ➔ im Client, bei der Übertragung, im Server
- ➔ Keine Einschränkungen gegenüber lokalen Schnittstellen
 - ➔ aber: semantische Unterschiede (Parameterübergabe!)



Hello-World Interface

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Hello extends Remote {  
    String sayHello() throws RemoteException;  
}
```

Marker-Schnittstelle ,
enthält keine Methoden,
markiert Interface als
RMI-Schnittstelle

RemoteException zeigt
Fehler im entfernten
Objekt bzw. bei Kommu-
nikation an



Implementierung der Server-Klasse

- ➔ Eine Klasse, die *remote* nutzbar sein soll, muß:
 - ➔ ein festgelegtes *Remote*-Interface implementieren
 - ➔ i.d.R. von `java.rmi.server.UnicastRemoteObject` abgeleitet werden
 - ➔ definiert Aufrufsemantik: Punkt-zu-Punkt
 - ➔ einen Konstruktor besitzen, der `RemoteException` werfen kann
 - ➔ Erzeugung des Objekts muß in `try-catch`-Block stehen
- ➔ Methoden brauchen `throws RemoteException` nicht nochmals anzugeben
 - ➔ ausser sie werfen diese Exception explizit selbst



Hello-World Server (1)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloServer extends UnicastRemoteObject
    implements Hello {
    public HelloServer() throws RemoteException {
        super();
    }
    public String sayHello() {
        return "Hello World!";
    }
}
```

Remote Methode



Entwicklung der Server-Anwendung zur Aufnahme des Server-Objekts

- ➔ Aufgaben:
 - ➔ Erzeugen eines Server-Objekts
 - ➔ Registrieren des Objekts beim Namensdienst
 - ➔ unter einem festgelegten, öffentlichen Namen
- ➔ Typischerweise keine neue Klasse, sondern `main`-Methode der Server-Klasse



Hello-World Server (2)

```
public static void main(String args[]) {
    try {
        HelloServer obj = new HelloServer();
        Naming.rebind("rmi://localhost/Hello-Server", obj);
    }
    catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Erzeugen des Server-Objekts

Registrieren des Server-Objekts unter dem Namen "Hello-Server" beim Name-Server (RMI-Registry, lokaler Rechner, Port 1099)



Entwicklung der Client-Anwendung mit Aufrufen des Server-Objekts

- ➔ Client muß sich zunächst beim Namensdienst über den Namen eine Referenz auf das Server-Objekt holen
 - ➔ *Type cast* auf den korrekten Typ erforderlich
- ➔ Dann: beliebige Methodenaufrufe möglich
 - ➔ syntaktisch kein Unterschied zu lokalen Aufrufen
- ➔ Anmerkung: Client kann *Remote*-Referenzen auch auf anderen Wegen erhalten
 - ➔ z.B. als Rückgabewert einer *Remote*-Methode



Hello-World Client

```
import java.rmi.*;

public class HelloClient {
    public static void main(String args[]) {
        try {
            Hello obj =
                (Hello)Naming.lookup("rmi://bspc02/Hello-Server");
            String message = obj.sayHello();
            System.out.println(message);
        }
        catch (Exception e) {
            ...
        }
    }
}
```

Objektreferenz vom Name-Server holen

Aufruf der Methode des entfernten Objekts

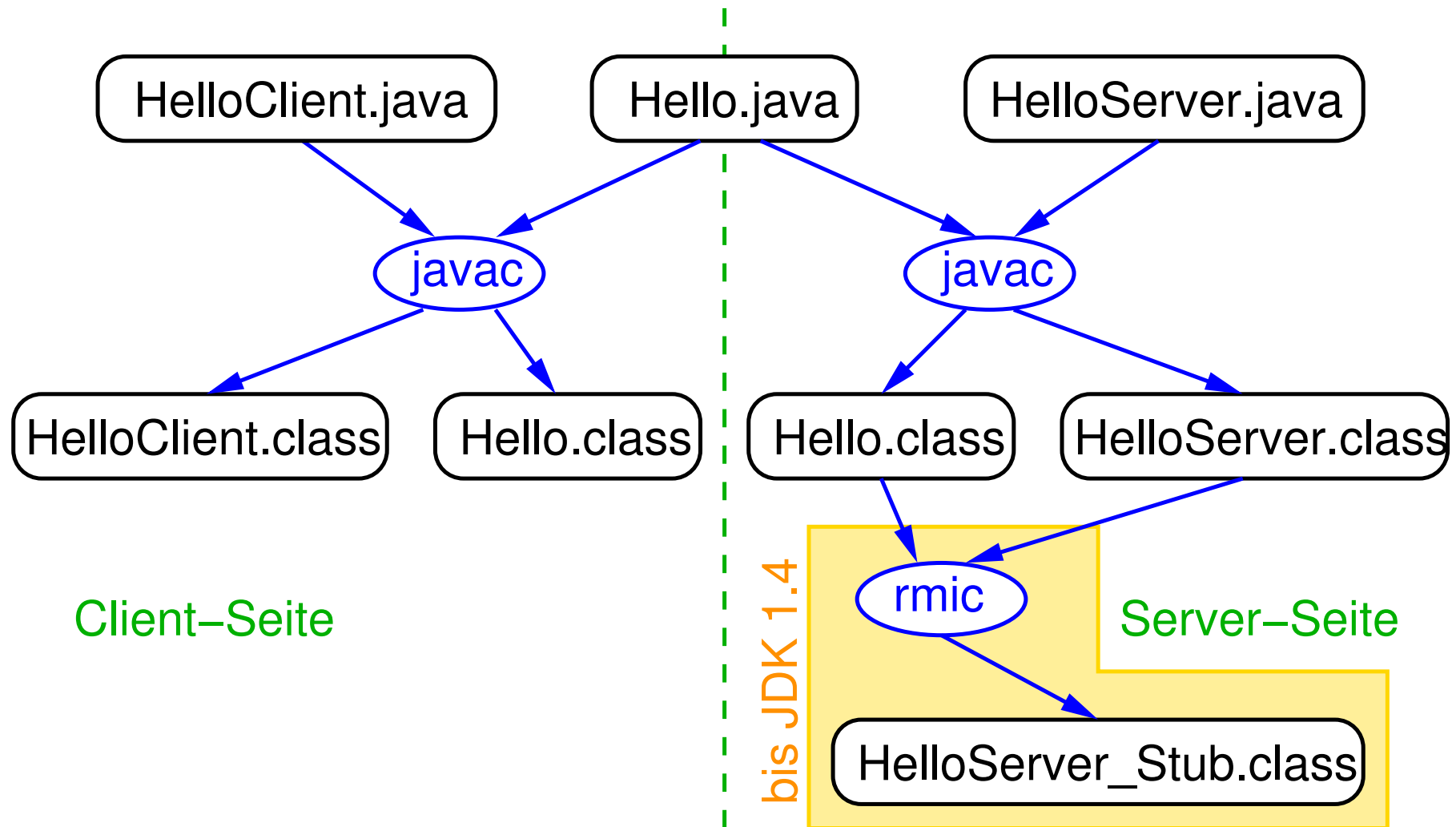


Übersetzen und Starten des Systems

- ➔ Übersetzen der Java-Quellen
 - ➔ Quelldateien: `Hello.java`, `HelloServer.java`, `HelloClient.java`
 - ➔ Aufruf: `javac *.java`
 - ➔ erzeugt: `Hello.class`, `HelloServer.class`, `HelloClient.class`
- ➔ Erzeugen des Client-Stubs (Proxy-Objekt)
 - ➔ für Clients bis JDK 1.4:
 - ➔ Aufruf: `rmic -v1.2 HelloServer`
 - ➔ erzeugt `HelloServer_Stub.class`
 - ➔ ab JDK 1.5: Client erzeugt Proxy-Klasse zur Laufzeit
 - ➔ durch `java.lang.reflect.Proxy`



Übersetzen und Starten des Systems ...





Übersetzen und Starten des Systems ...

- ➔ Starten des Namensdienstes
 - ➔ Aufruf: `rmiregistry [port]`
 - ➔ erlaubt aus Sicherheitsgründen nur die Registrierung von Objekten auf dem lokalen Host
 - ➔ d.h. *RMI-Registry* muß auf Server-Rechner laufen
 - ➔ Standard-Port: 1099
- ➔ Starten des Servers
 - ➔ Aufruf: `java HelloServer`
- ➔ Starten des Clients
 - ➔ Aufruf: `java HelloClient`

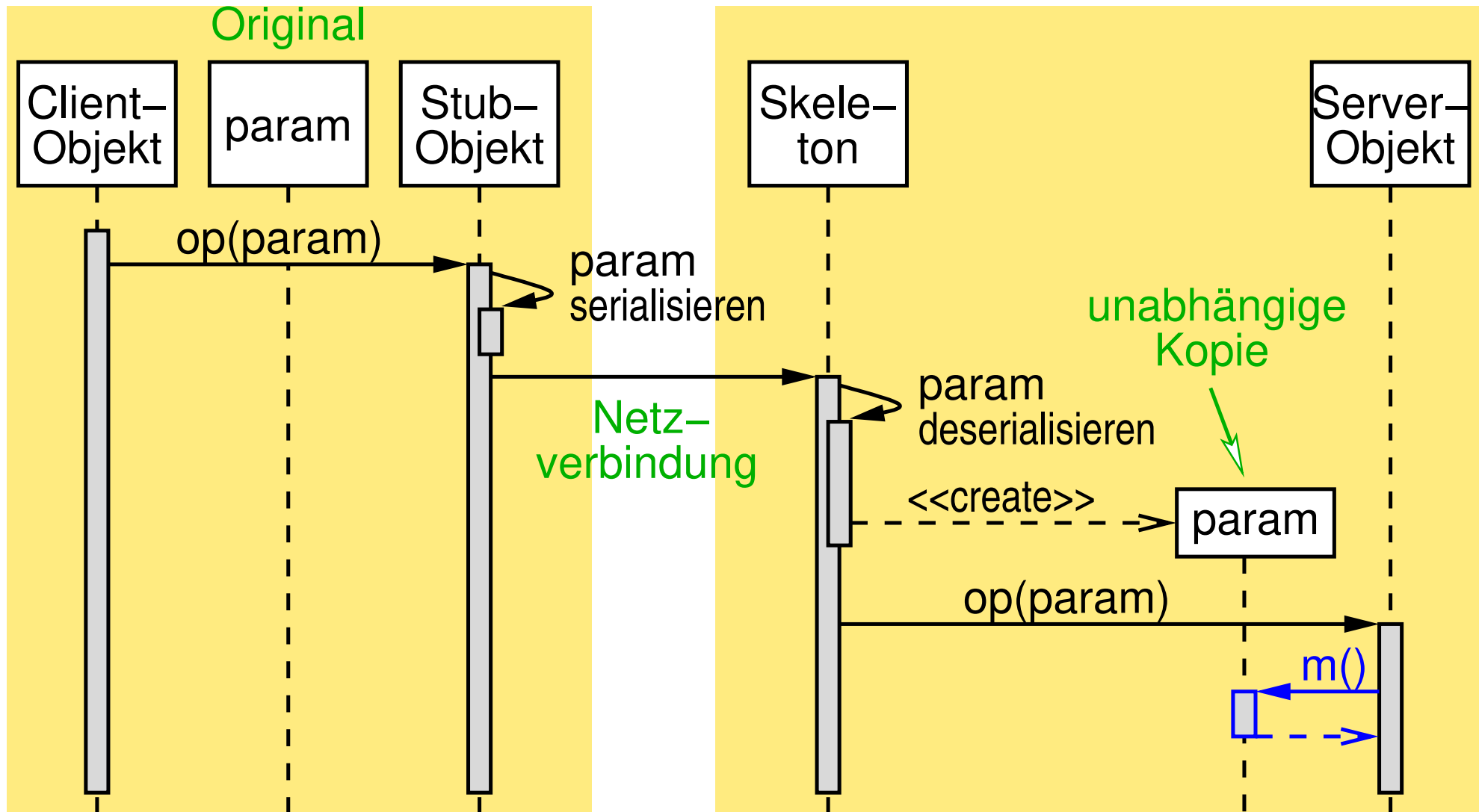


1.4.2 Parameterübergabe

- ➔ Übergabe von Parametern an *Remote*-Methoden erfolgt
 - ➔ entweder über *call-by-value*
 - ➔ für Werttypen und serialisierbare Objekte
 - ➔ oder über *call-by-reference*
 - ➔ für Objekte, die `Remote` implementieren
- ➔ Entscheidung wird z.T. erst zur Laufzeit getroffen!
- ➔ Rückgabe des Ergebnisses folgt selben Regeln wie Parameterübergabe



Übergabe eines serialisierbaren Objekts





Übergabe eines *Remote-Objekts*

