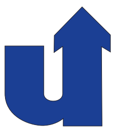
Verteilte Systeme

SoSe 2018

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

Stand: 24. November 2020

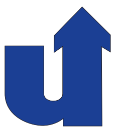


Verteilte Systeme

SoSe 2018

0 Organisation

Zu meiner Person

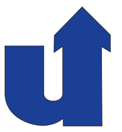


- Studium der Informatik an der Techn. Univ. München
  - dort 1994 promoviert, 2001 habilitiert
- Seit 2004 Prof. für Betriebssysteme und verteilte Systeme
- **Forschung:** Beobachtung, Analyse und Steuerung paralleler und verteilter Systeme
- **Mentor** für die Bachelor-Studiengänge Informatik mit Vertiefung Mathematik

- **e-mail:** roland.wismueller@uni-siegen.de
- **Tel.:** 0271/740-4050
- **Büro:** H-B 8404

- **Sprechstunde:** Mo., 14:15-15:15 Uhr

Zur Fachgruppe „Betriebssysteme / verteilte Systeme“



Andreas Hoffmann

andreas.hoffmann@uni-...

0271/740-4047

H-B 8405

- El. Prüfungs- und Übungssysteme
- IT-Sicherheit
- Web-Technologien
- Mobile Anwendungen

Damian Ludwig

damian.ludwig@uni-...

0271/740-2533

H-B 8402

- Capability-Systeme
- Compiler
- Programmiersprachen

Alexander Kordes

alexander.kordes@uni-...

0271/740-4011

H-B 5109

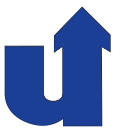
- *Automotive Electronics*
- Fahrzeugnetzwerke
- Mustererkennung in Fahrzeug-Sensordaten

Lehrangebot

Vorlesungen/Praktika

- Rechnernetze I, 5 LP (jedes SoSe)
- Rechnernetze Praktikum, 5 LP (jedes WiSe)
- Rechnernetze II, 5 LP (jedes SoSe)

- Betriebssysteme I, 5 LP (jedes WiSe)
- Parallelverarbeitung, 5 LP (jedes WiSe)
- Verteilte Systeme, 5 LP (jedes SoSe)
- Client/Server-Programmierung, 5 LP (jedes WiSe)



Projektgruppen

- z.B. Aufnahme und Analyse von Fahrzeugdaten
- z.B. Erkennung ungewöhnlicher Ereignisse in Kfz-Sensordaten

Abschlussarbeiten (Bachelor, Master)

- Themengebiete: sichere virtuelle Maschine, Parallelverarbeitung, Mustererkennung in Sensordaten, eAssessment, ...

Seminare

- Themengebiete: IT-Sicherheit, Programmiersprachen, Mustererkennung in Sensordaten, ...
- Ablauf: Blockseminare
  - 30 Min. Vortrag, 5000 Worte Ausarbeitung

Zur Vorlesung

Vorlesung

- Montag **12:20** - 13:50 Uhr, H-F 001

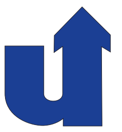
Übungen

- Di., 10:15-11:45, H-C 6336/37
- Do., 12:30-14:00, H-F 112

Übungsbeginn: ab Di., 24.04.

- Möglichkeit für praktische Übungen im Labor H-A 4111
- Ausgabe der Rechnerkennungen in der ersten Übung
  - Sie müssen die Benutzerordnung akzeptieren!
- Bitte vorab Kartenschlüsselantrag ausfüllen
  - bei Fr. Syska abstempeln lassen (H-B 8403, Mo. - Fr., 09:00 - 12:00), dann Abgabe bei Hr. Kiel (AR-P 209)
- Benutzerordnung und Kartenschlüsselantrag: siehe Webseite

Zur Vorlesung ...

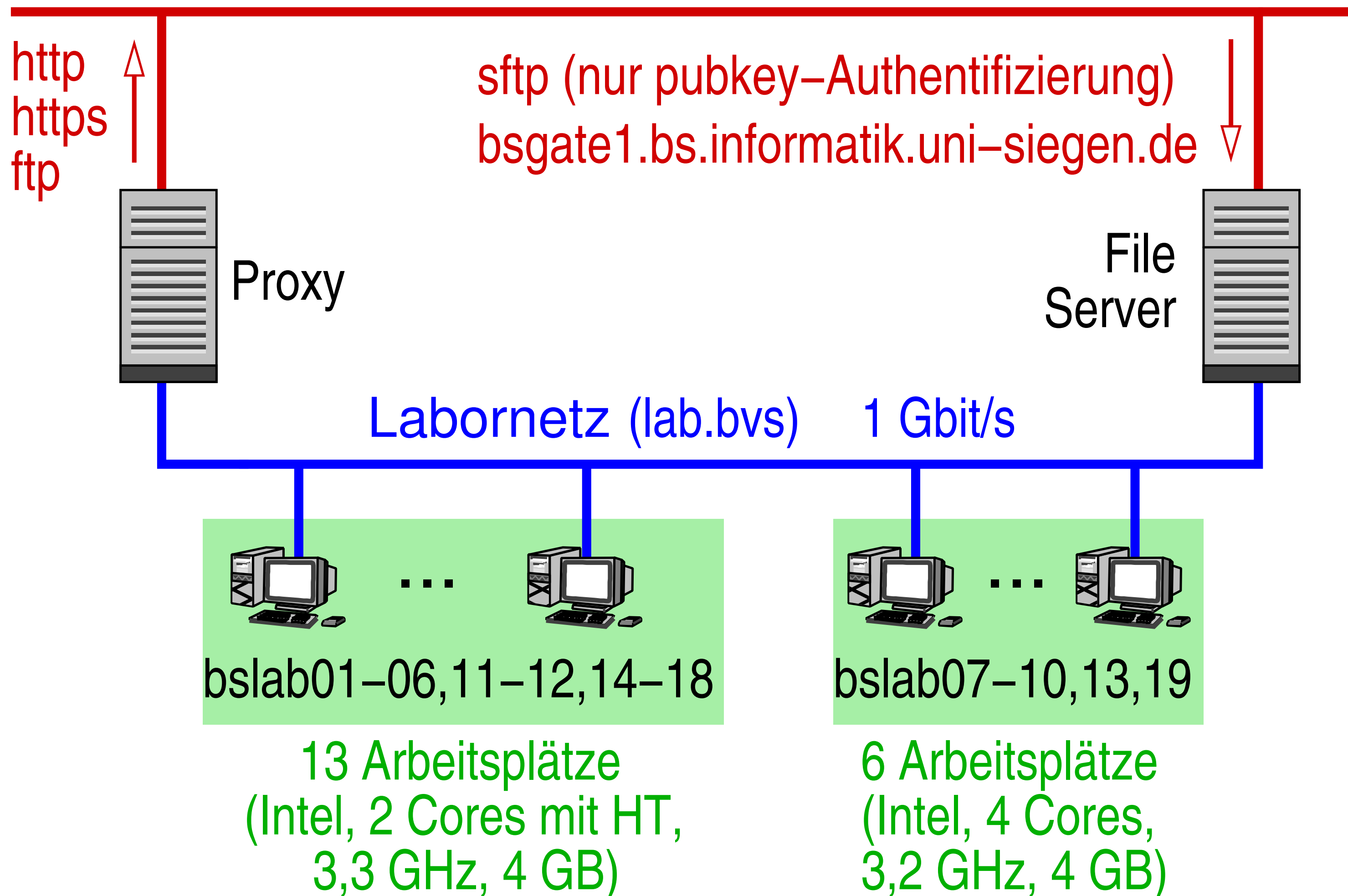


Information, Folien und Ankündigungen

- `http://www.bs.informatik.uni-siegen.de/lehre/ss18/vs`
- Ggf. Aktualisierungen/Ergänzungen kurz vor der Vorlesung
  - auf das Datum achten!
- Zum Ausdrucken: Druckservice des Fachschaftsrats!
- Übungsblätter werden als PDF online gestellt
  - bitte selbst ausdrucken und bearbeiten!

Rechnerumgebung im Labor H-A 4111

➥ Linux-PCs, privates IP-Netz, aber `ssh`-Zugang zum Cluster

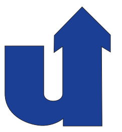


Prüfung

- Mündliche Prüfung
  - Dauer ca. 30 Minuten

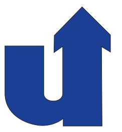
- Anmeldung:
  - Terminabsprache und Anmeldung über das Sekretariat
    - mindestens eine Woche vorher
    - Frau Syska, Raum H-B 8403, vormittags
    - Anmeldeformular (siehe Web-Seiten des Prüfungsamts) mitbringen!
  - Rücktritt bis 7 Tage vor Prüfungstemin möglich
    - über das Prüfungsamt!

Inhalt der Vorlesung



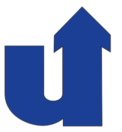
- Einführung
- Middleware
- Verteilte Programmierung mit Java RMI
- Namensdienste
- Prozeß-Management
- Zeit und globaler Zustand
- Koordination
- Replikation und Konsistenz
- Verteilte Dateisysteme
- Fehlertoleranz

Lernziele der Vorlesung



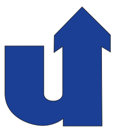
- Eigenschaften verteilter Systeme verstehen
  - Fehlen eines Globalzustands
  - Probleme bei der Synchronisation und Konsistenzsicherung replizierter Daten
- Verfahren zur Lösung der Probleme verstehen
  und auf gegebene Problemstellungen anwenden
- Architekturmodelle für verteilte Systeme sowie verschiedene
  Typen und Aufgaben von Middleware unterscheiden
  und Eignung für gegebene Problemstellungen einschätzen
- Einfache verteilte Anwendungen mit Java RMI entwickeln

Literatur

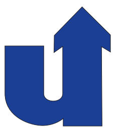


- Andrew S. Tanenbaum, Marten van Steen. *Verteilte Systeme, Grundlagen und Paradigmen*. Pearson Studium, 2003.
- Ulrike Hammerschall. *Verteilte Systeme und Anwendungen*. Pearson Studium, 2005.
- George Coulouris, Jean Dollimore, Tim Kindberg. *Verteilte Systeme, Konzepte und Design, 3. Auflage*. Pearson Studium, 2002.
- Andrew S. Tanenbaum. *Moderne Betriebssysteme, 2. Auflage*. Pearson Studium, 2003.
- William Stallings. *Betriebssysteme – Prinzipien und Umsetzung, 4. Auflage*. Pearson Studium, 2003.

Literatur ...



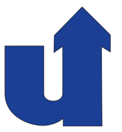
- Jim Farley, William Crawford, David Flanagan. *Java Enterprise in a Nutshell*. O'Reilly 2002.
- Cay S. Horstmann, Gary Cornell. *Core Java 2, Band 2 – Expertenwissen*. Sun Microsystems Press / Addison Wesley, 2008.
- Robert Orfali, Dan Harkey. *Client/Server-Programming with Java and Corba*. John Wiley & Sons, 1998.
- Torsten Langner. *Verteilte Anwendungen mit Java*. Markt + Technik, 2002.



Verteilte Systeme

SoSe 2018

1 Einführung

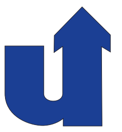


Inhalt

- Was macht ein verteiltes System aus?
- Software-Architektur
- Architekturmodelle
- Cluster

Literatur

- Hammerschall: Kap. 1
- Tanenbaum, van Steen: Kap. 1
- Colouris, Dollimore, Kindberg: Kap. 1, 2
- Stallings: Kap 13.4



1.1 Was ist ein verteiltes System?

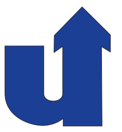
> Bei einem verteilten System arbeiten Komponenten zusammen,die sich auf unterschiedlichen Computern befinden und die ihre Aktionen durch den Austausch von Nachrichten koordinieren.
>
> *G. Coulouris*

> Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen.
>
> *A. Tanenbaum*

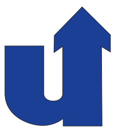
> Ein verteiltes System ist eine Sammlung von Prozessoren, die sich weder den Hauptspeicher noch eine Uhr teilen.
>
> *A. Silberschatz*

> Ein verteiltes System ist eines, auf dem ich keine Arbeit erledigen kann, weil irgendeine Maschine, von der ich noch nie gehört habe, abgestürzt ist.
>
> *L. Lamport*

1.1 Was ist ein verteiltes System? ...

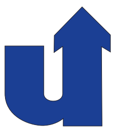


- Ein verteiltes System ist **ein System**
  - in dem sich **Hardware- und Software-Komponenten** auf **vernetzten Computern** befinden und
  - nur über den **Austausch von Nachrichten** kommunizieren und ihre Aktionen koordinieren.
- Die Grenzen des verteilten Systems werden durch eine gemeinsame Anwendung definiert
- Bekanntestes Beispiel: Internet
  - Kommunikation über die standardisierten Internet-Protokolle
    - IP und TCP / UDP (☞ Vorlesung Rechnernetze)
  - Nutzer können Dienste / Anwendungen nutzen, unabhängig vom gegenwärtigen Standort



Was ist eine verteilte Anwendung?

- Anwendung, die verteiltes System nutzt, um eine in sich geschlossene Funktionalität zur Verfügung zu stellen
- Anwendungslogik auf mehrere, weitgehend unabhängige **Komponenten** verteilt
- Komponenten oft auf verschiedenen Rechnern ausgeführt
- Beispiele:
  - einfache Internetanwendungen (z.B. WWW, FTP, Email)
  - verteilte Informationssysteme (z.B. Flugbuchung)
    - SW-intensiv, datenzentriert, interaktiv, stark nebenläufig
  - verteilte eingebettete Systeme (z.B. im Auto)
  - verteilte mobile Anwendungen (z.B. bei Handhelds)



Ein typisches verteiltes System

Email–Server

Desktop

Desktop

LAN

Internet

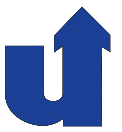
LAN

WWW–Server

Anwen–dungs–Server

Druck–Server

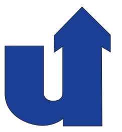
Daten–bank–Server



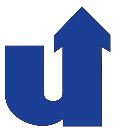
Warum Verteilung?

- Zentrale, nicht verteilte Anwendungen sind
  - im Allgemeinen sicherer und zuverlässiger
  - im Allgemeinen performanter
- Wesentlicher Grund für Verteilung: Gemeinsame Nutzung von Ressourcen
  - Hardware-Ressourcen (Drucker, Scanner, ...)
    - Kostenersparnis
  - Daten und Informationen (Fileserver, Datenbank, ...)
    - Informationsaustausch, Datenkonsistenz
  - Funktionalität (Zentralisierung)
    - Fehlervermeidung, Wiederverwendung

1.2 Charakteristika von verteilten Systemen

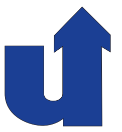


- Ressourcen (z.B. Rechner, Daten, Benutzer, ...) sind verteilt
  - ggf. auch weltweit
- Kooperation über Nachrichtenaustausch
- Nebenläufigkeit
  - aber: nebenläufige Bearbeitung **eines** Auftrags ist nicht primäres Ziel
- Keine globale Uhr (genauer: keine globale Zeit)
- Verteilte Zustandsinformation
  - kein eindeutig bestimmter globaler Zustand
- Partielle Fehler sind möglich (unabhängige Ausfälle)



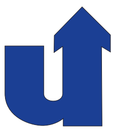
Parallele vs. verteilte Systeme

- Paralleles System:
  - Motivation: höhere Rechenleistung durch Parallelarbeit
  - Mehrere Tasks (Prozesse/Threads) arbeiten an einem Job
  - Tasks sind feingranular: häufige Kommunikation
  - Tasks arbeiten gleichzeitig (parallel)
  - homogene Hardware / BSe, regelmäßige Verbindungsstruktur
- Verteiltes System:
  - Motivation: verteilte Ressourcen (Rechner, Daten, Benutzer)
  - Mehrere Tasks (Prozesse/Threads) arbeiten an einem Job
  - Tasks sind grobgranular: Kommunikation weniger häufig
  - Tasks arbeiten synchronisiert (i.a. nacheinander)
  - inhomogen (Prozessoren, Netzwerke, BSe, ...)

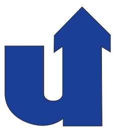


- **Heterogenität**: Rechnerhardware, Netzwerke, BSe, Programmiersprachen, Implementierungen durch verschiedene Entwickler, ...
  - Lösung: **Middleware**
    - Softwareschicht, die Heterogenität verbirgt, indem sie ein einheitliches Programmiermodell bereitstellt
    - z.B. CORBA: verteilte Objekte, entfernte Methodenaufrufe
    - z.B. Web Services: entfernte Prozeduraufrufe (Dienste)
- **Offenheit**: problemlose Erweiterbarkeit (mit neuen Diensten)
  - Dazu:
    - Schlüsselschnittstellen sind veröffentlicht / standardisiert
    - einheitliche Kommunikationsmechanismen / Protokolle
    - Komponenten müssen standardkonform sein

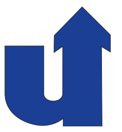
[Coulouris, 1.4]



- **Sicherheit**
  - Information: Vertraulichkeit, Integrität, Verfügbarkeit
  - Benutzer: Authentifizierung, Autorisierung
  - mobiler Code
- **Skalierbarkeit**: Zahl der Ressourcen bzw. Benutzer kann ohne negative Auswirkungen auf Leistung und Kosten wachsen
- **Fehlerverarbeitung** (partielle Fehler)
  - Fehlererkennung (z.B. Prüfsummen)
  - Fehlermaskierung (z.B. Neuübertragung einer Nachricht)
  - Fehlertoleranz (z.B. Browser: „*server not available*")
  - Wiederherstellung (von Daten) nach Fehlern
  - Redundanz (von Hard- und Software)

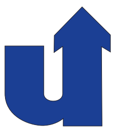


- **Nebenläufigkeit**
  - Synchronisation, Konsistenzerhaltung von Daten
- **Transparenz**
  - Zugriffs∼: lokale und entfernte Zugriffe identisch } Netzwerk∼
  - Orts∼: Zugriff ohne Kenntnis des Orts möglich } Netzwerk∼
  - Mobilitäts∼: transparentes Verschieben von Ressourcen
  - Replikations∼: transparente Replikation von Ressourcen
  - Nebenläufigkeits∼: gemeinsame Ressourcennutzung ohne Störungen
  - Fehler∼: Verbergen von Fehlern durch Komponenten-Ausfall
  - Leistungs∼: Leistung ist i.W. unabhängig von der Last
  - Skalierungs∼: Skalierung ohne Auswirkungen auf Benutzer



Arten von Betriebssystemen für verteilte Systeme

- Netzwerk-Betriebssystem:
  - herkömmliches BS, erweitert um Unterstützung für Netzwerk-Anwendungen (API für Sockets, RPC, ...)
  - jeder Rechner hat sein eigenes BS, kann aber Dienste anderer Rechner (Dateisystem, Email, `ssh`, ...) nutzen
  - Existenz der anderen Rechner ist sichtbar
- Verteiltes Betriebssystem:
  - einheitliches BS für Netzwerk von Rechnern
  - transparent für den Anwender
  - erfordert Zusammenarbeit der BS-Kerne
  - bisher i.W. Forschungsprojekte



Typische Schichten in einem verteilten System

Anwendungen
Dienste (allgemein oder anwendungsspezifisch)

API

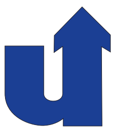
Middleware

(Netzwerk–) Betriebssystem

Rechner– und Netzwerk–Hardware

Plattform(en)

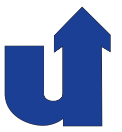
[Coulouris, 2.2.1]



Middleware

- Aufgaben:
  - Verbergen der Verteiltheit und der Heterogenität
  - Bereitstellen eines gemeinsamen Programmiermodells / APIs
  - Bereitstellung von allgemeinen Diensten
- Funktionen z.B.:
  - Kommunikationsdienste: entfernte Methodenaufrufe, Gruppenkommunikation, Ereignisbenachrichtigungen
  - Replikation gemeinsam genutzter Daten
  - Sicherheitsdienste
- Beispiele: CORBA, EJB, .NET, Axis2 (Web Services), ...
  (☞ Vorlesung Client/Server-Programmierung)

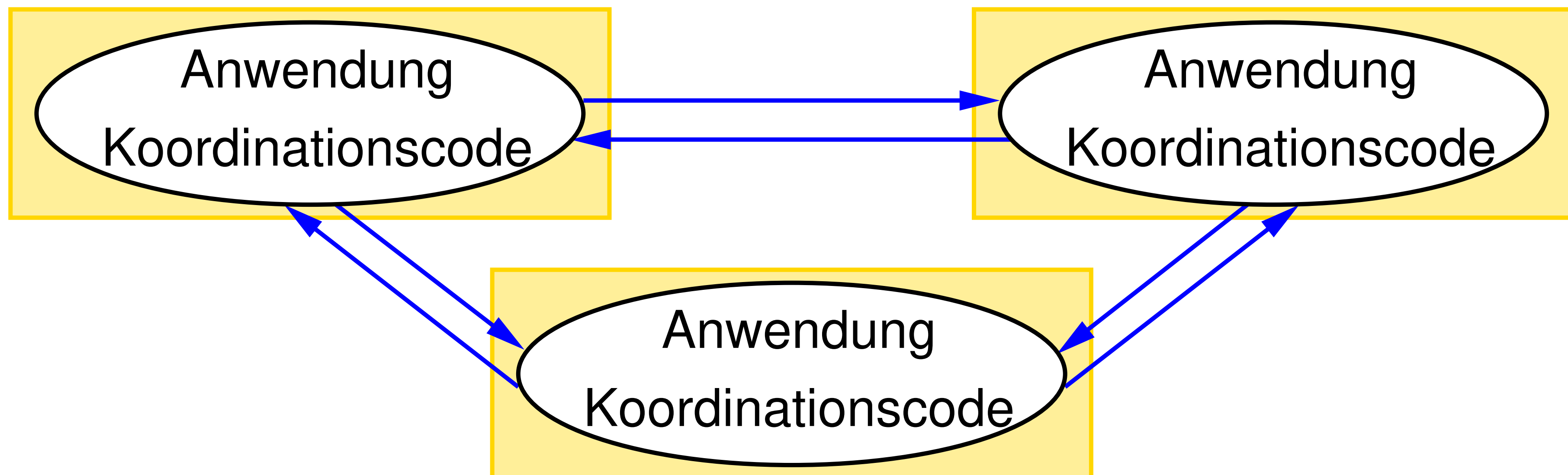
1.5 Architekturmodelle



- Architekturmodell beschreibt:
  - Rollen einer Anwendungskomponente innerhalb der verteilten Anwendung
  - Beziehungen zwischen den Anwendungskomponenten
- Rolle durch Prozeßtyp definiert, in dem Komponente läuft:
  - Client Prozeß
    - kurzlebig (für die Dauer der Nutzung durch Anwender)
    - agiert als Initiator einer Interprozeßkommunikation (IPC)
  - Server Prozeß
    - lebt 'unbegrenzt'
    - agiert als Diensterbringer einer IPC
  - Peer Prozeß
    - kurzlebig (für die Dauer der Nutzung durch Anwender)
    - agiert als Initiator und Diensterbringer

Peer-to-Peer-Modell

- Zusammenarbeit gleichrangiger Prozesse für verteilte Aktivität
  - jeder Prozeß verwaltet lokalen Teil der Ressourcen
  - verteilte Koordination und Synchronisation der Aktionen auf Anwendungsebene



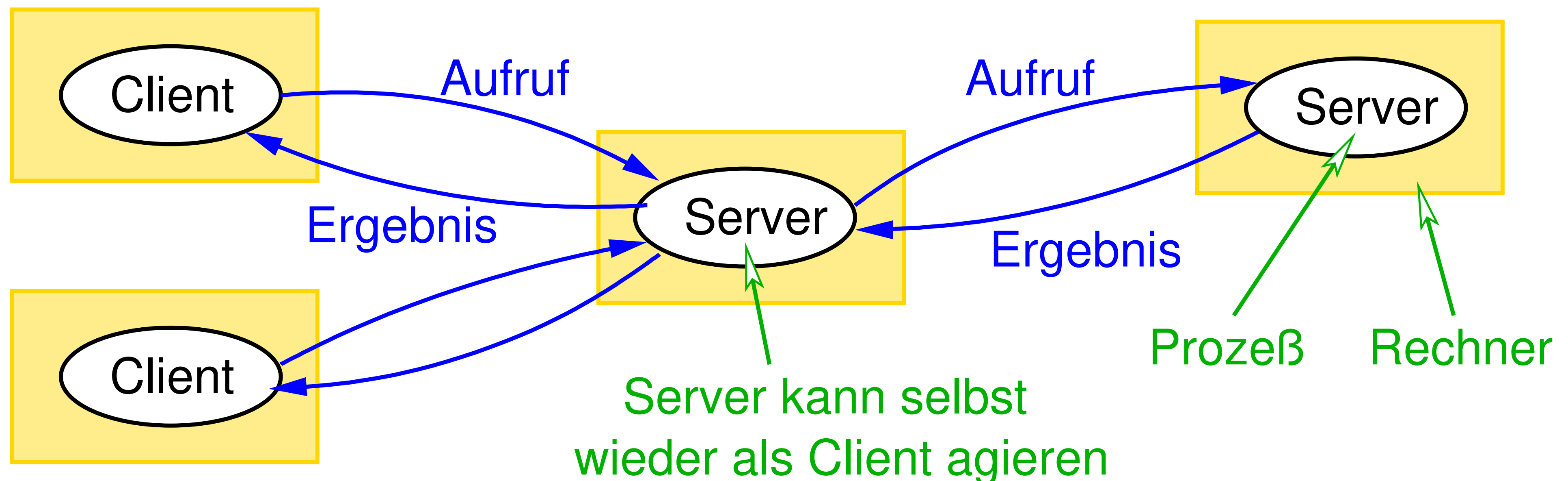
- Z.B.: *File sharing*-Anwendungen, Router, Videokonferenzen, ...

1.5 Architekturmodelle ...

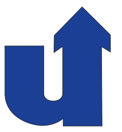
Client/Server-Modell

[Coulouris, 2.2.2]

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

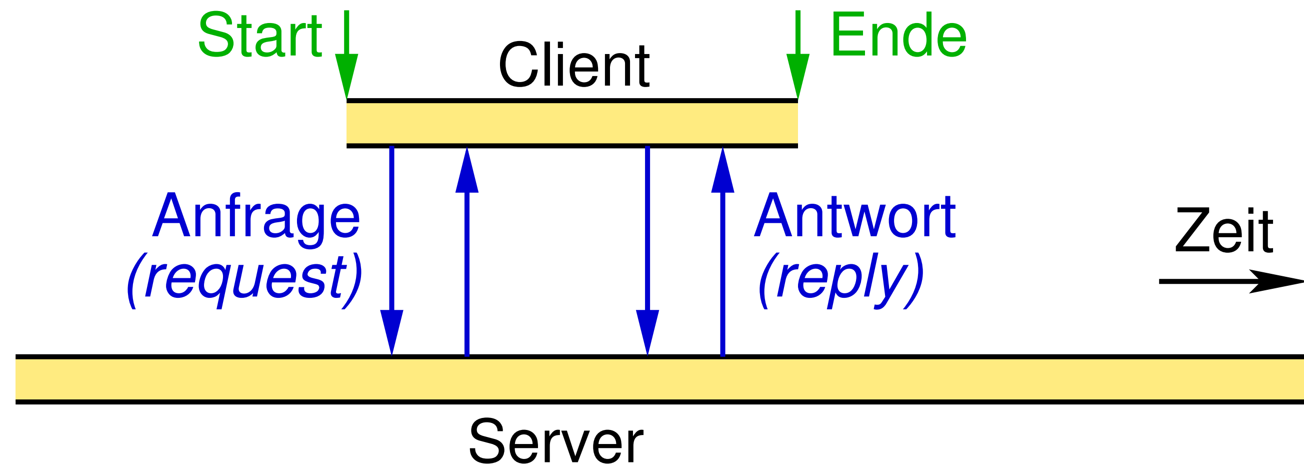


- Häufigstes Modell für verteilte Anwendungen (ca. 80 %)



Client/Server-Modell ...

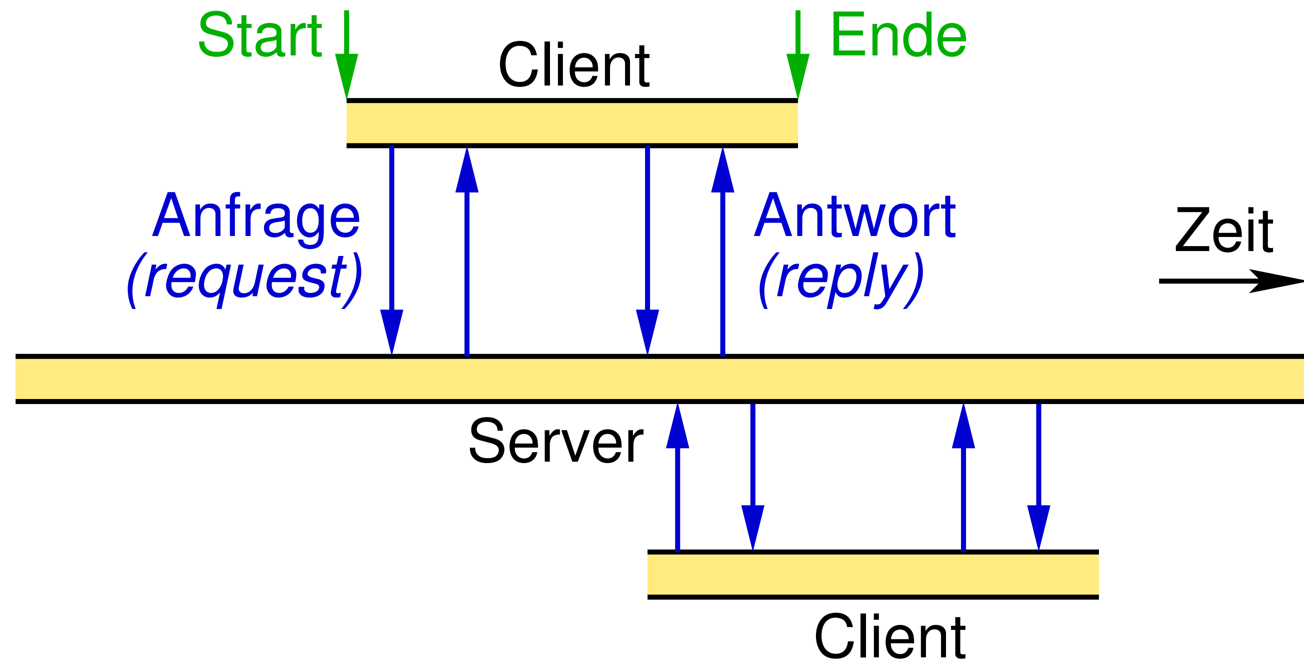
- I.A. nebenläufige Anfragen mehrerer Client-Prozesse an den Server-Prozeß



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

Client/Server-Modell ...

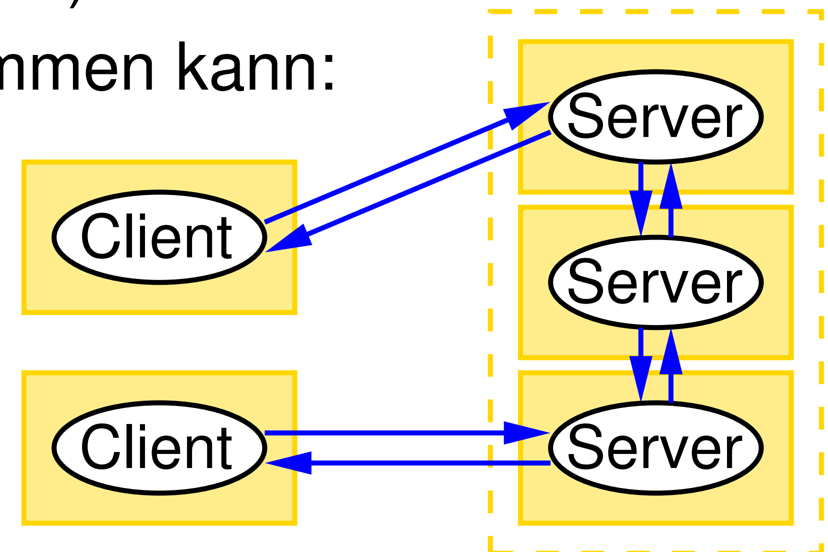
- I.A. nebenläufige Anfragen mehrerer Client-Prozesse an den Server-Prozeß



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

Varianten des Client/Server-Modells

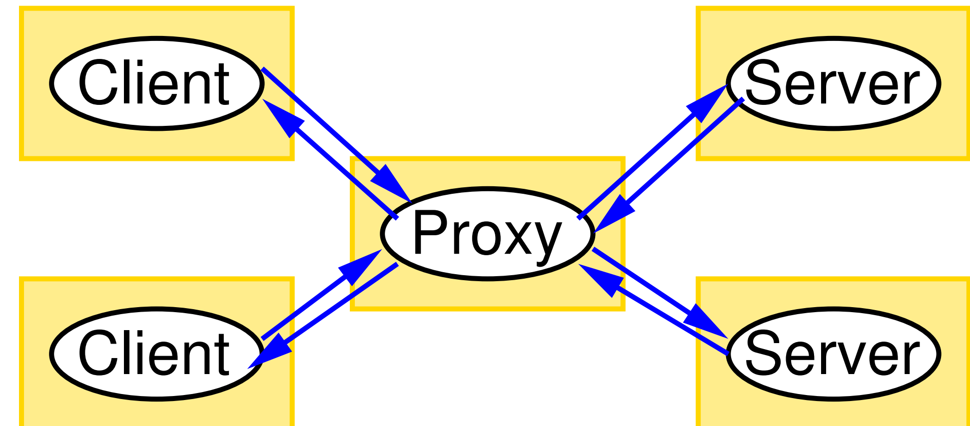
- Kooperierende Server
  - Verbund von Servern bearbeitet transparent einen Aufruf
  - Beispiel: *Domain Name Server* (DNS)
    - falls Server Adresse nicht bestimmen kann: Aufruf wird transparent an anderen Server weitergereicht



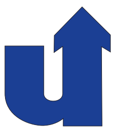
- Replizierte Server
  - Replikate der Serverprozesse werden bereitgestellt
    - transparente Replikate (oft in Clustern)
      - Anfragen werden automatisch auf die Server verteilt
    - öffentliche Replikate (z.B. *Mirror Server*)
  - Ziele: bessere Performance, Ausfallsicherheit

Varianten des Client/Server-Modells ...

- Proxy-Server / Caches
  - Proxy ist Stellvertreter für Server
  - Aufgabe oft Caching von Daten / Ergebnissen
  - z.B. Web-Proxy

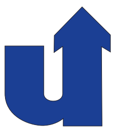


- Mobiler Code
  - ausführbarer Servercode wandert auf Anfrage zum Client
  - Ausführung des Codes durch Client
  - bekanntestes Beispiel: JavaScript / Java Applets im WWW
- Mobile Agenten
  - Agent enthält Code und Daten, wandert durch das Netzwerk und führt Aktionen auf lokalen Ressourcen durch



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

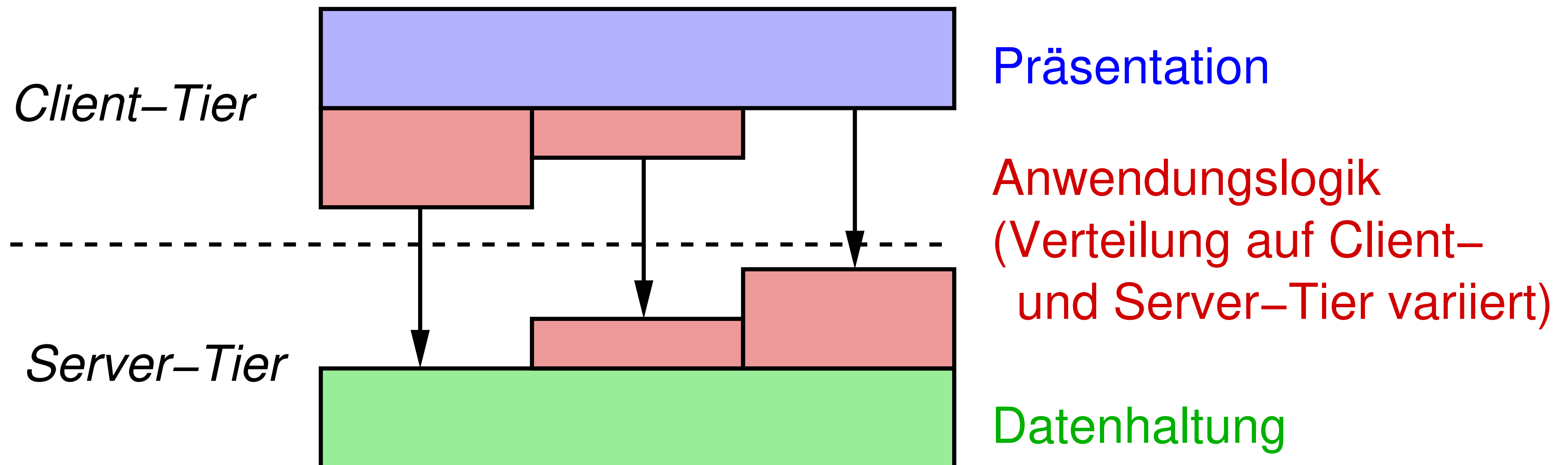


Das *Tier*-Modell

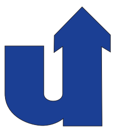
- Typische Aufgaben in einem Informationssystem:
  - Präsentation – Schnittstelle zum Anwender
  - Anwendungslogik – eigentliche Funktionalität
  - Datenhaltung – Speicherung der Daten in einer Datenbank
- *Tier*-Modell bestimmt:
  - Zuordnung der Aufgaben zu Anwendungskomponenten
  - Verteilung der Anwendungskomponenten auf *Tiers*
- Architekturen:
  - 2-*Tier*-Architekturen
  - 3-*Tier*-Architekturen
  - 4- und mehr-*Tier*-Architekturen

2-*Tier*-Architektur

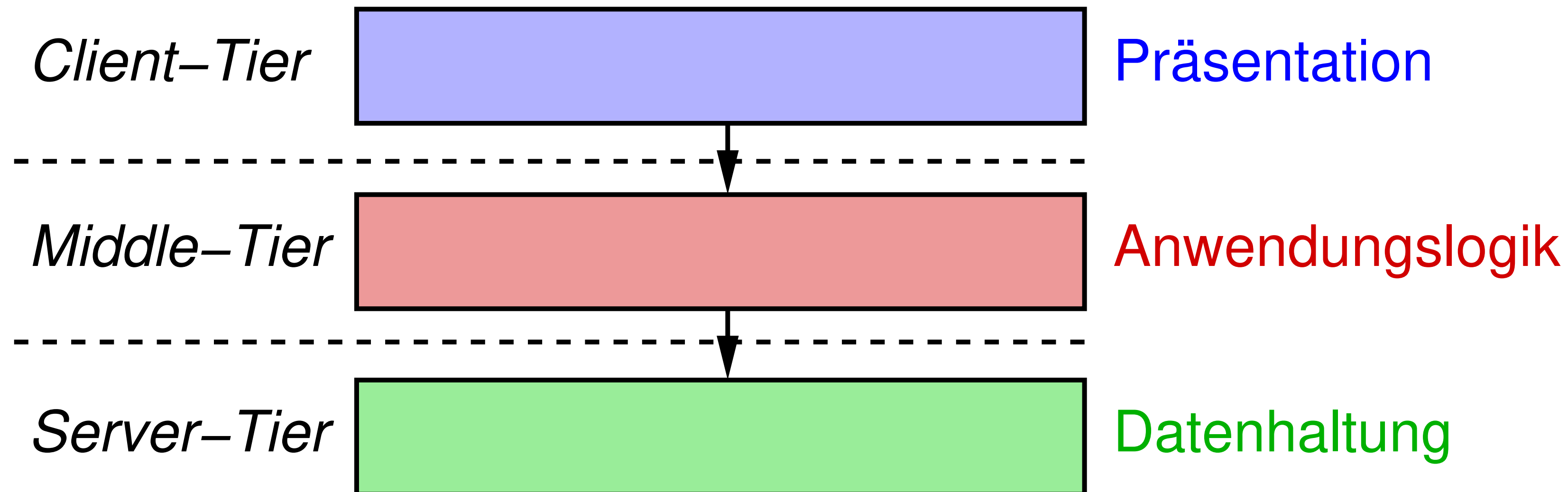
- Client- und Server-*Tier*
- Keine eigene *Tier* für die Anwendungslogik



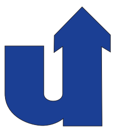
- Vorteil: einfach, performant
- Nachteil: schwer wartbar, schlecht skalierbar



3-*Tier*-Architektur



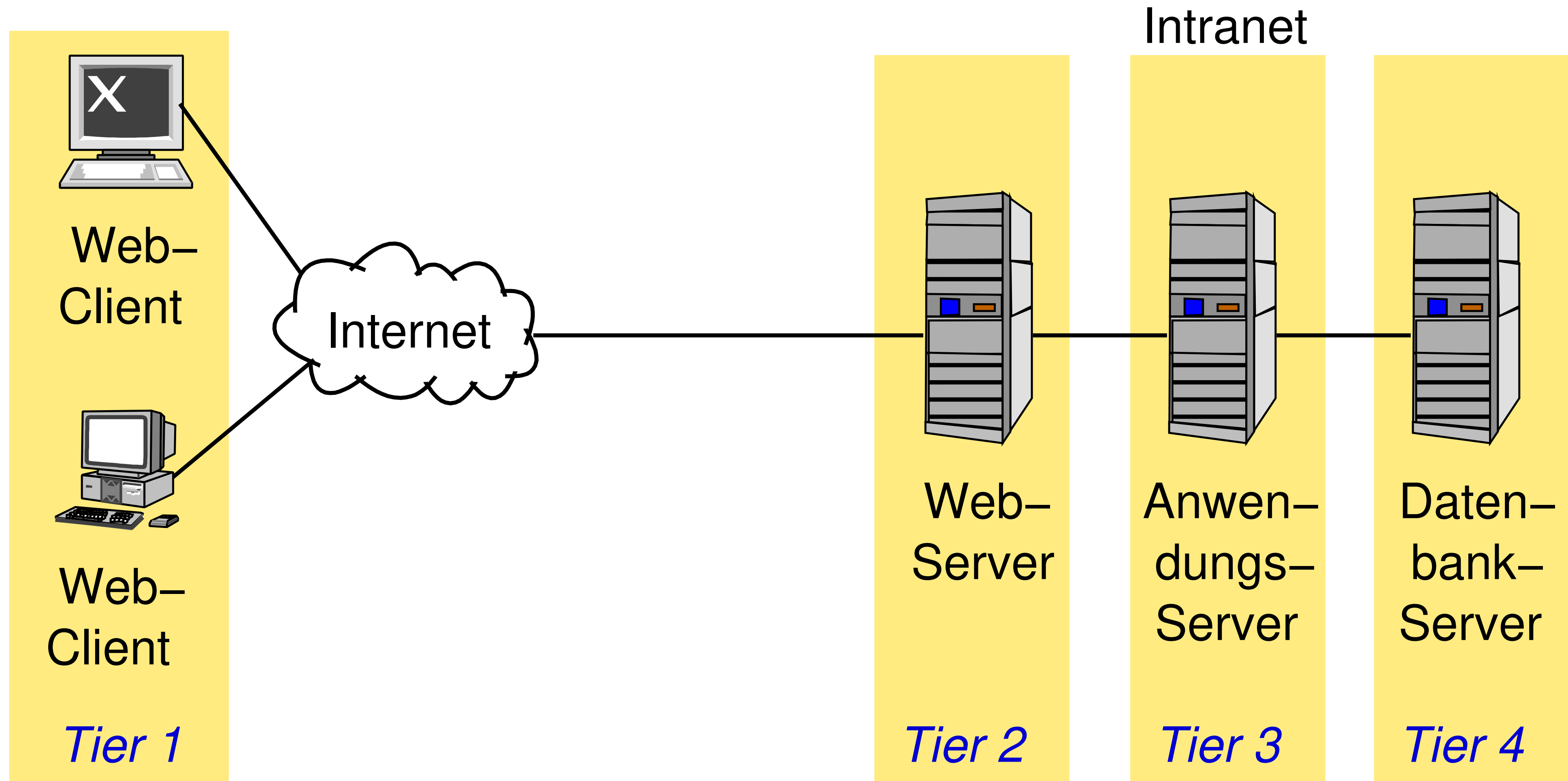
- Standard-Verteilungsmodell für einfache Web-Anwendungen:
  - *Client-Tier*: Web-Browser zur Anzeige
  - *Middle-Tier*: Web-Server mit Servlets / JSP / ASP
  - *Server-Tier*: Datenbank-Server
- Vorteile: Anwendungslogik zentral administrierbar, skalierbar



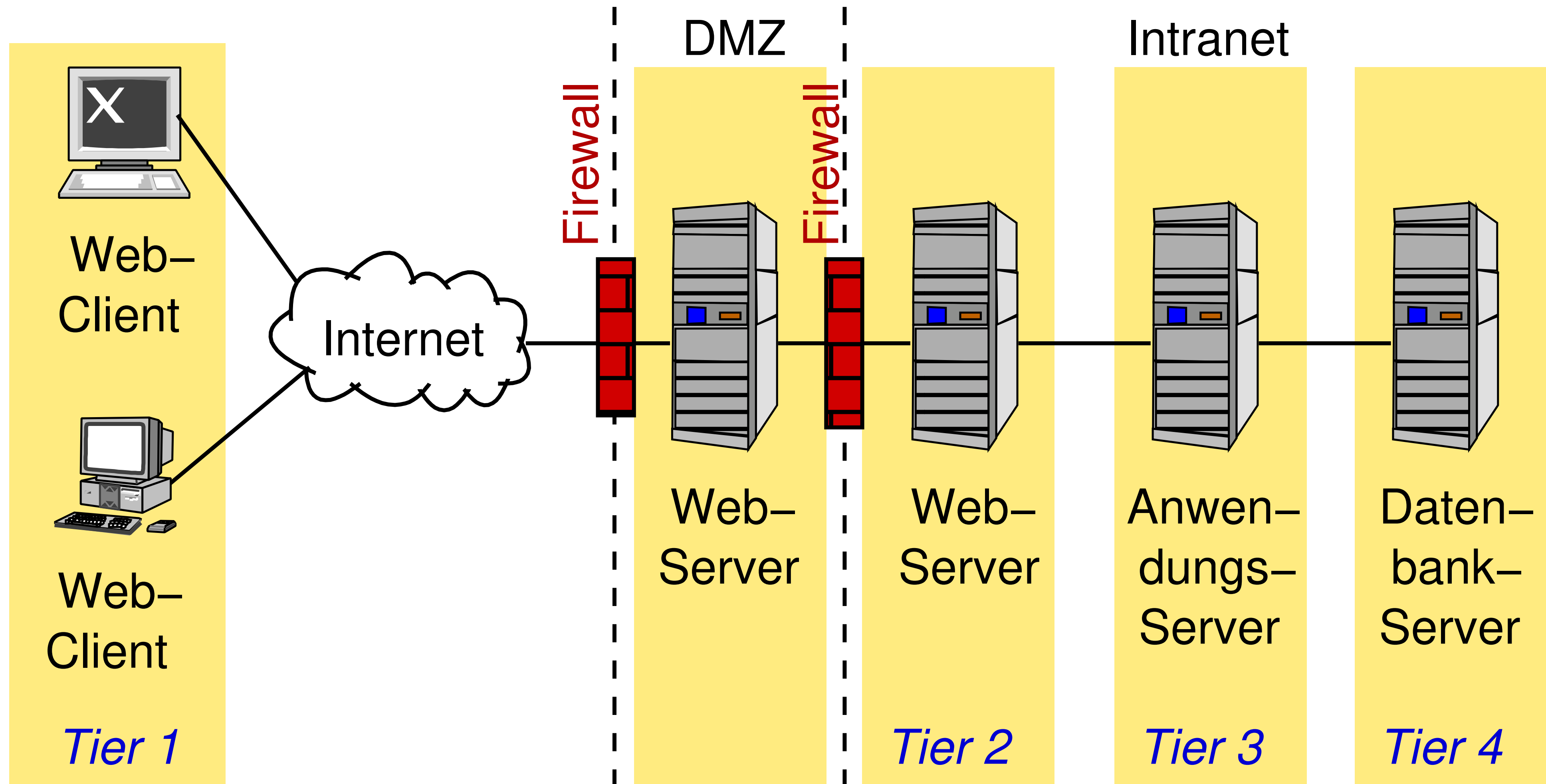
4- und mehr-*Tier*-Architekturen

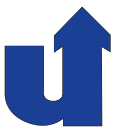
- Unterschied zu 3-*Tier*-Architektur:
  - Anwendungslogik auf mehere *Tiers* verteilt
- Motivation:
  - Minimierung der Komplexität (*Divide and Conquer*)
  - Besserer Schutz einzelner Anwendungsteile
  - Wiederverwendbarkeit von Komponenten
- Viele verteilte Informationssysteme haben 4- und mehr-*Tier*-Architekturen

Beispiel: typische Internet-Anwendung



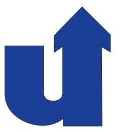
Beispiel: typische Internet-Anwendung





Thin- und *Fat-Clients*

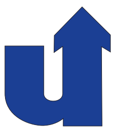
- Charakterisiert Komplexität der Anwendungskomponente auf der *Client-Tier*
- *Ultra-Thin-Client*
  - *Client-Tier* nur Präsentation: reine Anzeige von Dialogen
  - Päsentationskomponenten: Web-Browser
  - nur bei 3- und mehr-*Tier*-Architekturen möglich
- *Thin-Client*
  - *Client-Tier* nur Präsentation: Anzeige von Dialogen, Aufbereitung der Daten zur Anzeige
- *Fat-Client*
  - Teile der Anwendungslogik auf *Client-Tier*
  - i.d.R. 2-*Tier*-Architektur

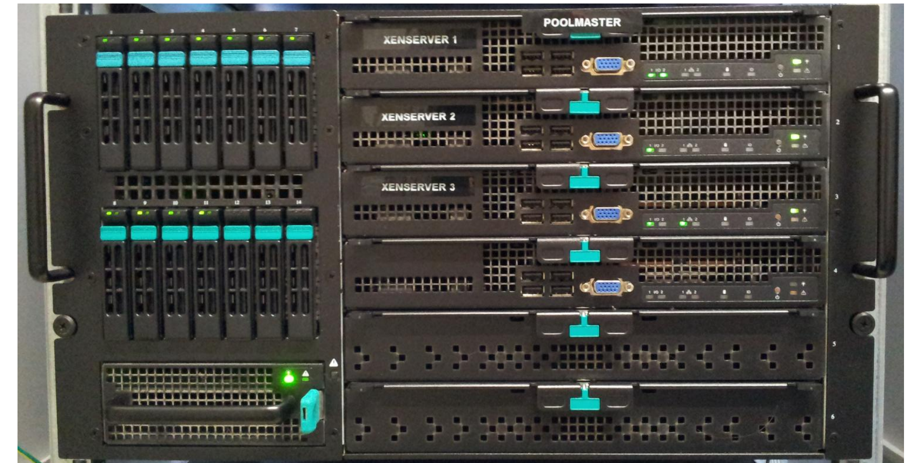


Abgrenzung zu *Enterprise Application Integration* (EAI)

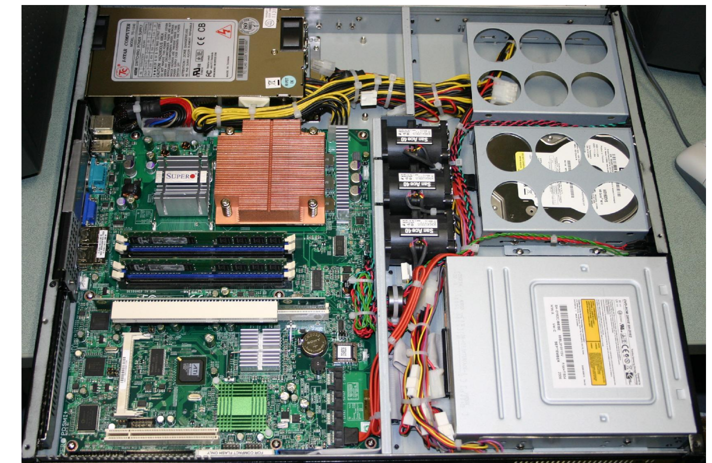
- EAI: Integration unterschiedlicher Anwendungen
  - Kommunikation, Austausch von Daten
- Ziele ähnlich wie bei verteilten Anwendungen / Middleware
  - Middleware wird oft auch für EAI eingesetzt
- Unterschiede:
  - verteilte Anwendungen: Anwendungskomponenten, hoher Kopplungsgrad, meist wenig Heterogenität
  - EAI: vollständige Anwendungen, geringer Kopplungsgrad, meist große Heterogenität (unterschiedliche Technologien, Systeme, Programmiersprachen, ...)

1.6 Cluster

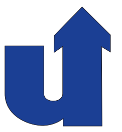




- Cluster: Gruppe vernetzter Rechner, die als einheitliche Rechenressource wirkt
  - d.h. Multicomputer-System
  - Knoten i.a. Standard-PCs oder *Blade*-Server
- Anwendung v.a. als Hochleistungs-Server
- Motivation:
  - (schrittweise) Skalierbarkeit
  - hohe Verfügbarkeit
  - gutes Preis-/Leistungsverhältnis

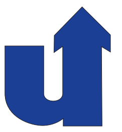


[Stallings, 13.4]



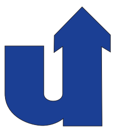
Verwendungszwecke für Cluster

- Hochverfügbarkeits-Cluster (*High Availability*, HA)
  - verbesserte Ausfallsicherheit
  - bei Fehler auf einem Knoten: Dienste werden auf andere Knoten migriert (*Failover*)
- *Load Balancing*-Cluster
  - eingehende Aufträge werden auf verschiedene Knoten des Clusters verteilt
    - i.a. durch (redundante) zentrale Instanz
  - häufig bei WWW bzw. Email-Servern
- *High Performance Computing*-Cluster
  - Cluster als Parallelrechner



Cluster-Konfigurationen

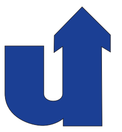
- Passives Standby (kein eigentlicher Cluster)
  - Bearbeitung aller Anfragen durch primären Server
  - sekundärer Server übernimmt Aufgaben (nur) bei Ausfall
- Aktives Standby
  - alle Server bearbeiten Aufträge
  - ermöglicht Lastausgleich und Ausfallsicherheit
  - Problem: Zugriff auf Daten anderer / ausgefallener Server
  - Alternativen:
    - Replikation der Daten (viel Kommunikation)
    - gemeinsames Festplattensystem (i.d.R. gespiegelte Platten bzw. RAID-System wegen Ausfallsicherheit)



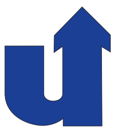
Konfigurationen für Aktives Standby

- Separate Server mit Replikation der Daten
  - getrennte Platten, Daten werden laufend auf sekundären Server kopiert
- Server mit gemeinsamen Festplatten
  - *Shared Nothing* Cluster
    - getrennte Partitionen für jeden Server
    - bei Serverausfall: Umkonfiguration der Partitionen
  - *Shared Disk* Cluster
    - gleichzeitige Nutzung durch alle Server
    - benötigt *Lock Manager* Software zum Sperren von Dateien bzw. Datensätzen

1.7 Zusammenfassung



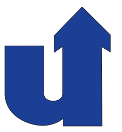
- Verteiltes System
  - HW- und SW-Komponenten auf vernetzten Rechnern
  - kein gemeinsamer Speicher, keine globale Zeit
  - Motivation: Nutzung verteilter Ressourcen
- Herausforderungen
  - Heterogenität, Offenheit, Sicherheit, Skalierbarkeit
  - Fehlerverarbeitung, Nebenläufigkeit, Transparenz
- Software-Architektur: Middleware
- Architekturmodelle:
  - *Peer-to-Peer*, Client/Server
  - *n-Tier* Modelle
- Cluster: Hochverfügbarkeit, *Load Balancing*



Verteilte Systeme

SoSe 2018

2 Middleware

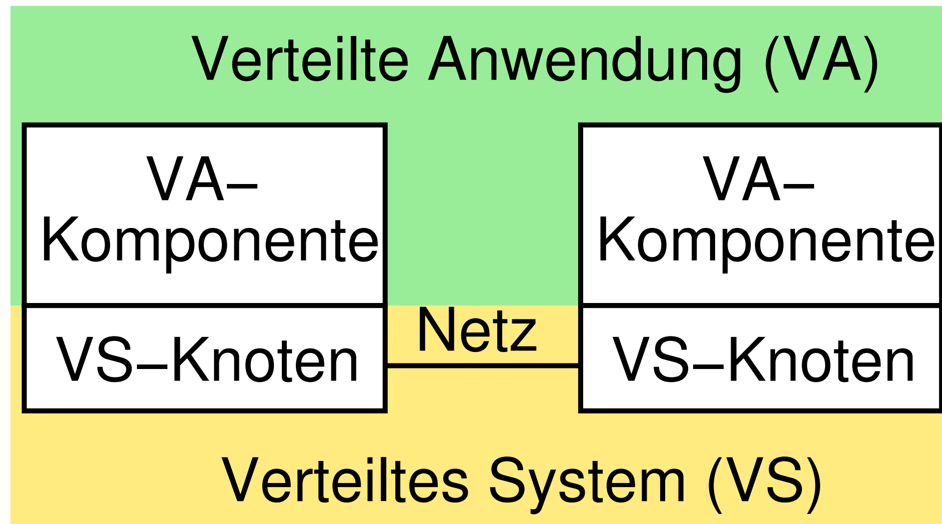


Inhalt

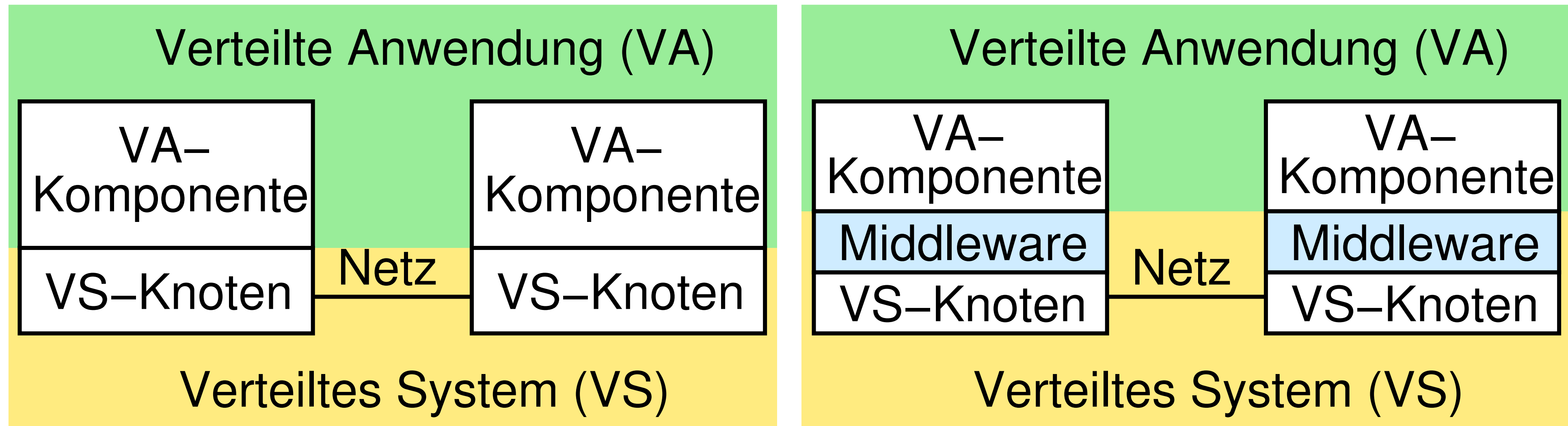
- Kommunikation in verteilten Systemen
- Kommunikationsorientierte Middleware
- Anwendungsorientierte Middleware

Literatur

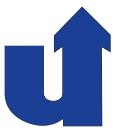
- Hammerschall: Kap. 2, 6
- Tanenbaum, van Steen: Kap. 2
- Colouris, Dollimore, Kindberg: Kap 4.4



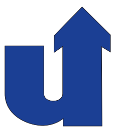
- VA nutzt VS für Kommunikation zwischen ihren Komponenten
- VSe bieten i.a. nur einfache Kommunikationsdienste an
  - direkte Nutzung: **Netzwerkprogrammierung**
- **Middleware** bietet intelligentere Schnittstellen
  - verbirgt Details der Netzwerkprogrammierung



- 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
  - Transparenz (☞ **1.3**)
- Middleware kann auch Zusatzdienste für Anwendungen bieten
  - starke Unterschiede bei existierender Middleware
- Unterscheidung:
  - **kommunikationsorientierte Middleware** (☞ **2.2**)
    - (nur) Abstraktion von der Netzwerkprogrammierung
  - **anwendungsorientierte Middleware** (☞ **2.3**)
    - neben Kommunikation steht Unterstützung verteilter Anwendungen im Mittelpunkt

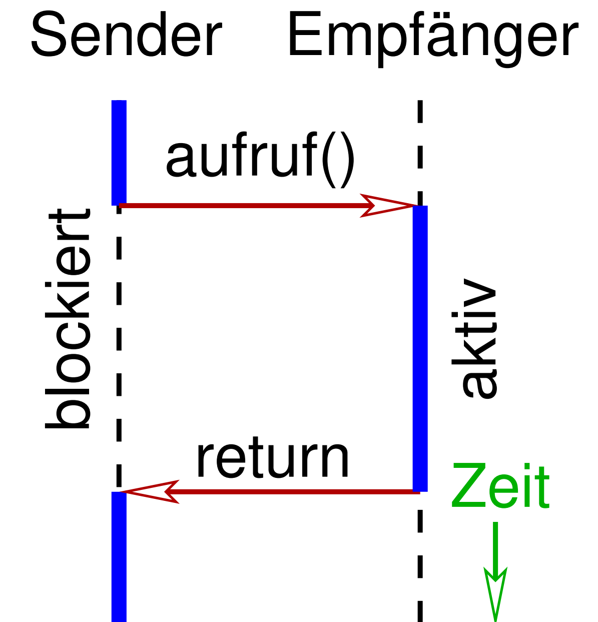


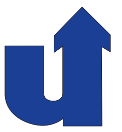
2.1 Kommunikation in verteilten Systemen

- Basis: **Interprozeßkommunikation (IPC)**
  - Austausch von Nachrichten zwischen Prozessen (☞ **BS_I: 3.2**)
    - auf demselben oder auf verschiedenen Knoten
    - z.B. über Ports, Mailboxen, Ströme, ...
- Zur Verteilung: Netzwerkprotokolle (☞ **RN_I**)
  - relevante Themen u.a.: Adressierung, Zuverlässigkeit, Reihenfolgeerhaltung, *Timeouts*, Bestätigungen, *Marshalling*
- Schnittstelle zur Netzwerkprogrammierung: Sockets (☞ **RN_II**)
  - Datagramme (UDP) und Ströme (TCP)

Synchrone Kommunikation

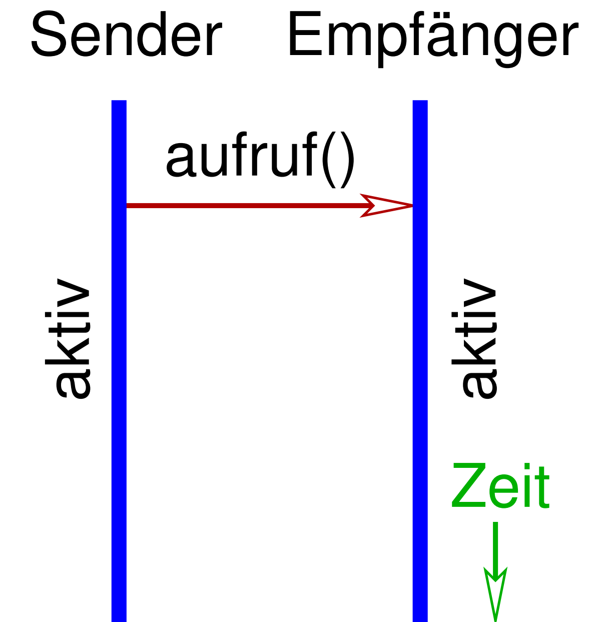
- Sender und Empfänger blockieren beim Aufruf der Sende- bzw. Empfangs-Operation
  - Empfänger wartet auf Aufruf
  - Sender wartet auf Ergebnis des Aufrufs
- Enge Kopplung zwischen Sender und Empfänger
  - Vorteil: einfach zu verstehendes Modell
  - Nachteil: hohe Abhängigkeit, insbes. im Fehlerfall
- Voraussetzungen:
  - zuverlässige und schnelle Netzwerk-Verbindung
  - Empfängerprozeß ist verfügbar





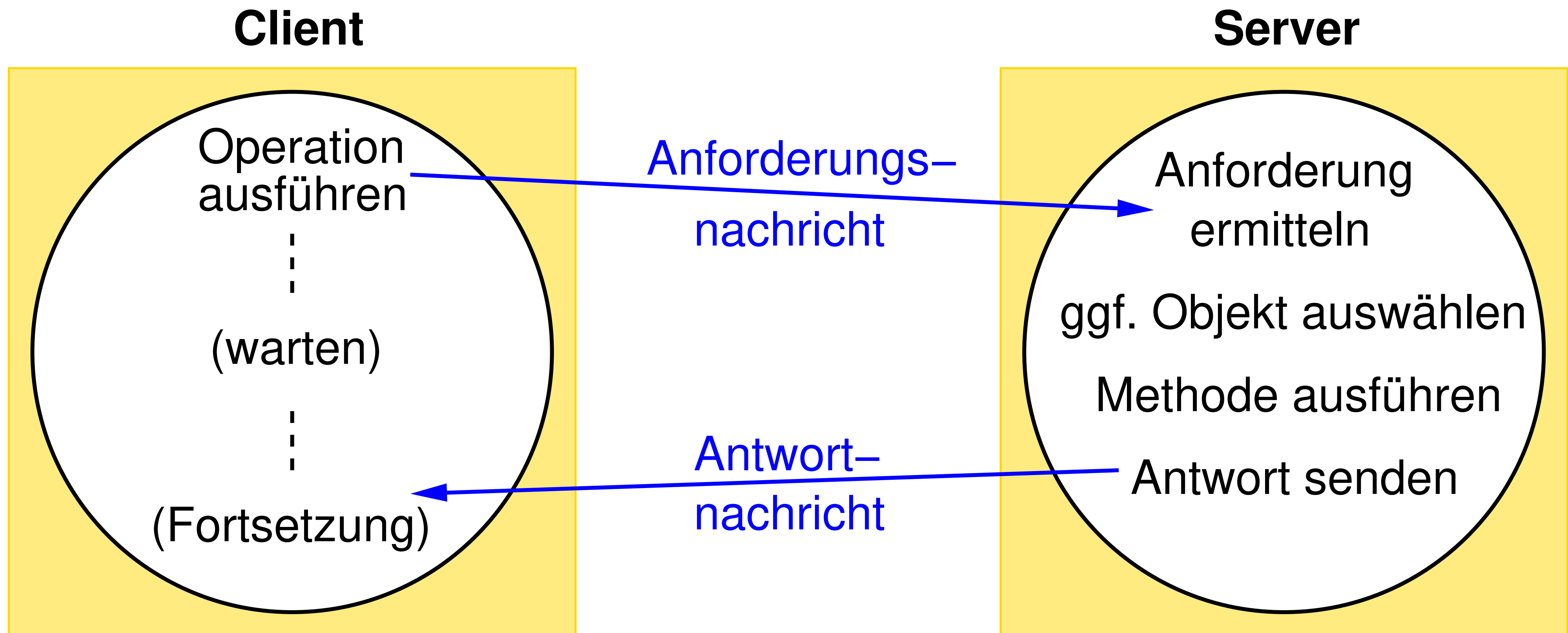
Asynchrone Kommunikation

- Sender wird nicht blockiert, kann nach dem Senden der Nachricht sofort weiterarbeiten
- Eingehende Nachrichten werden beim Empfänger gepuffert
- Antworten sind optional
  - Empfänger kann Antwort asynchron an Sender schicken



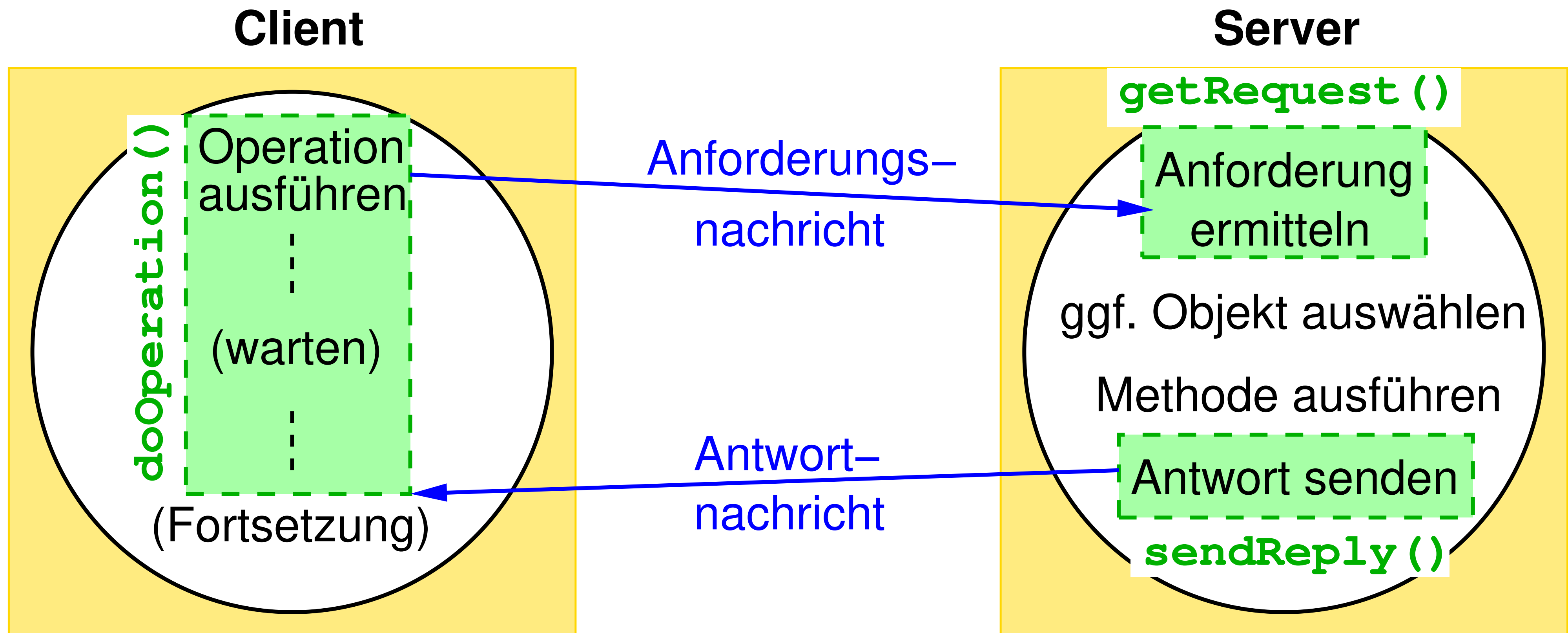
- Komplexere Implementierung und Verwendung als synchrone Kommunikation, aber meist effizienter
- Nur lose Kopplung zwischen den Prozessen
  - Empfänger muß nicht empfangsbereit sein
  - geringere Fehlerabhängigkeit

Client/Server-Kommunikation

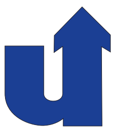


- Meist synchron: Client blockiert, bis Antwort eintrifft
- Varianten: asynchron (nicht blockierend), *one way* (ohne Antwort)

Client/Server-Kommunikation



- Meist synchron: Client blockiert, bis Antwort eintrifft
- Varianten: asynchron (nicht blockierend), *one way* (ohne Antwort)

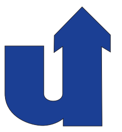


Client/Server-Kommunikation: Anfrage-/Antwort-Protokoll

- Typische Operationen:
  - `doOperation()` – sende Anfrage und warte auf Ergebnis
  - `getRequest()` – warte auf Anfrage
  - `sendReply()` – sende Ergebnis
- Typische Nachrichtenstruktur:

Feld	Bedeutung
messageType	Anfrage / Antwort ?
requestID	eindeutige ID der Anfrage (i.a. int)
objectReference	Referenz auf entfernes Objekt (ggf.)
methodID	aufzurufende Methode (int / String)
arguments	Argumente (i.a. als Byte–Array)

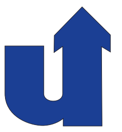
  - Request-ID + Sender-ID ergeben eindeutige Nachrichten-ID
    - z.B. um Antwort einer Anfrage zuzuordnen



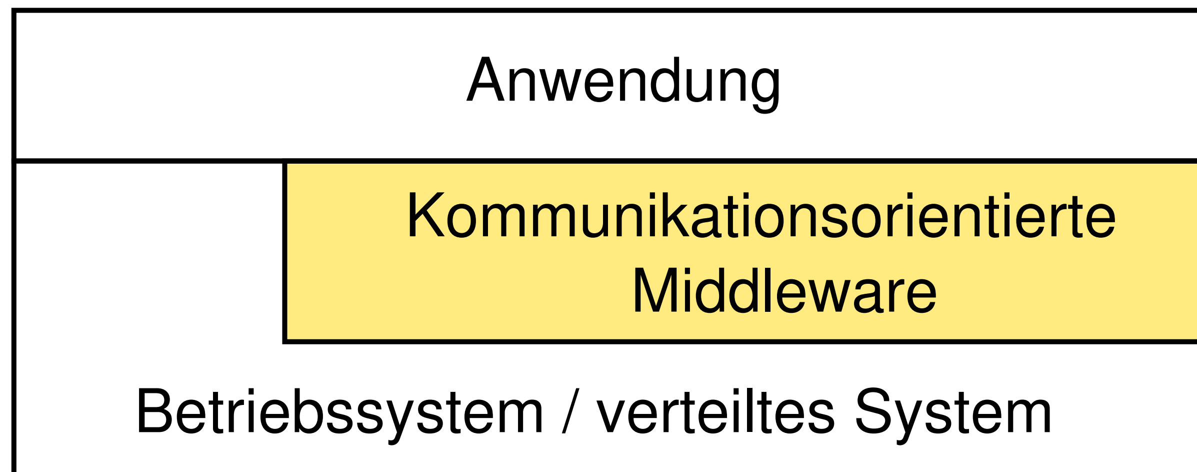
Client/Server-Kommunikation: Fehlerbehandlung

- Anfrage- und Antwort-Nachrichten können ggf. verloren gehen
- Client setzt *Timeout* bei Anfrage
  - nach Ablauf wird Anfrage i.a. erneut gesendet
  - nach einigen Wiederholungen: Abbruch mit Ausnahme
- Server verwirft doppelte Anfragen, falls Anfrage bereits / noch bearbeitet wird
- Bei verlorenen Antwort-Nachrichten:
  - idempotente Operationen können erneut ausgeführt werden
  - sonst: Ergebnisse der Operationen in *History* speichern
    - bei wiederholter Anfrage: nur Ergebnis neu senden
    - Löschen von *History*-Einträgen, wenn nächste Anfrage eintrifft; ggf. auch Bestätigungen für Ergebnisse

2.2 Kommunikationsorientierte Middleware



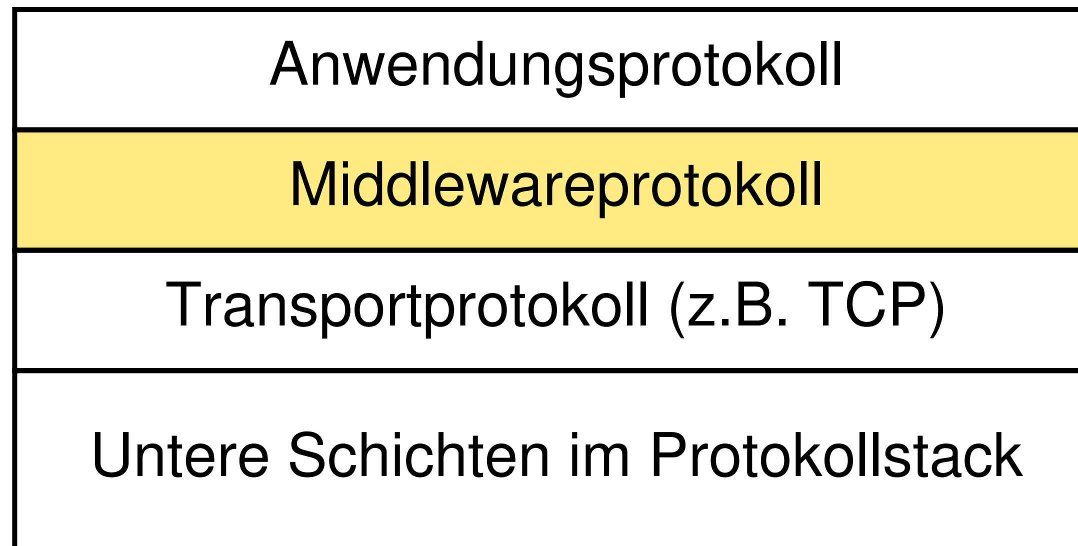
- Fokus: Bereitstellung einer Kommunikationsinfrastruktur für verteilte Anwendungen
- Aufgaben:
  - Kommunikation
  - Behandlung der Heterogenität
  - Fehlerbehandlung

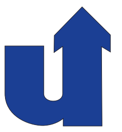


2.2.1 Aufgaben der Middleware

Kommunikation

- Bereitstellung eines Middleware-Protokolls
- Lokalisierung und Identifikation der Kommunikationspartner
- Integration mit Prozeß- und Threadverwaltung

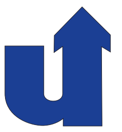




Heterogenität

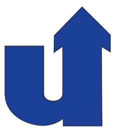
- Problem bei der Datenübertragung:
  - Heterogenität in verteilten Systemen
- Heterogene Hardware und Betriebssysteme
  - unterschiedliche Byte-Reihenfolge
    - *Little Endian* / *Big Endian*
  - unterschiedliche Zeichencodierung
    - z.B. ASCII / Unicode / UTF-8 / EBCDIC (IBM Mainframes)
- Heterogene Programmiersprachen
  - unterschiedliche Darstellung von einfachen und komplexen Datentypen im Hauptspeicher

2.2.1 Aufgaben der Middleware ...



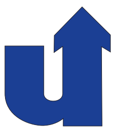
Heterogenität: Lösungen (☞ RN_I)

- Verwendung übergeordneter, einheitlicher Datenformate
  - allen Kommunikationspartnern und Middleware bekannt
  - plattformspezifische Formate für eine Middleware (z.B. CDR bei CORBA) oder externe Formate, z.B. XML
- Heterogenität von Hardware und Betriebssystem
  - wird transparent für die Anwendungen von der Middleware behandelt
- Heterogenität der Programmiersprachen
  - Anwendungen müssen Daten in übergeordnetes Format und zurück konvertieren (***Marshalling*** / ***Unmarshalling***)
    - notwendiger Code wird i.d.R. automatisch generiert
      - *Client-Stub / Server-Skeleton*



Fehlerbehandlung

- Mögliche Fehler durch Verteilung
  - Fehlerhafte Übertragung (inkl. Verlust von Nachrichten)
    - Behandlung durch Protokolle des verteiltes Systems:
      - Prüfsummen, CRC
      - Neuübertragung von Paketen (z.B. bei TCP)
  - Ausfall von Komponenten (Netz, Hardware, Software)
    - Behandlung durch Middleware oder Anwendung:
      - Akzeptanz des Fehlers
      - Neusenden von Nachrichten
      - Replikation von Komponenten (Fehlervermeidung)
      - kontrolliertes Beenden der Anwendung

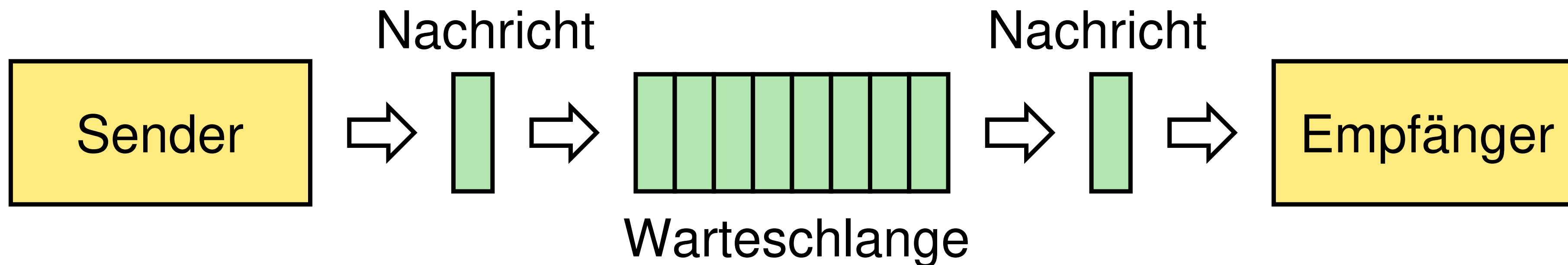


2.2.2 Programmiermodelle

- Programmiermodell legt zwei Konzepte fest:
  - Kommunikationsmodell
    - synchron vs. asynchron
  - Programmierparadigma
    - objektorientiert vs. prozedural
- Drei verbreitete Programmiermodelle bei Middleware:
  - nachrichtenorientiertes Modell (asynchron / beliebig)
  - entfernte Prozeduraufrufe (synchron / prozedural)
  - entfernte Methodenaufrufe (synchron / objektorientiert)

Nachrichtenorientiertes Modell

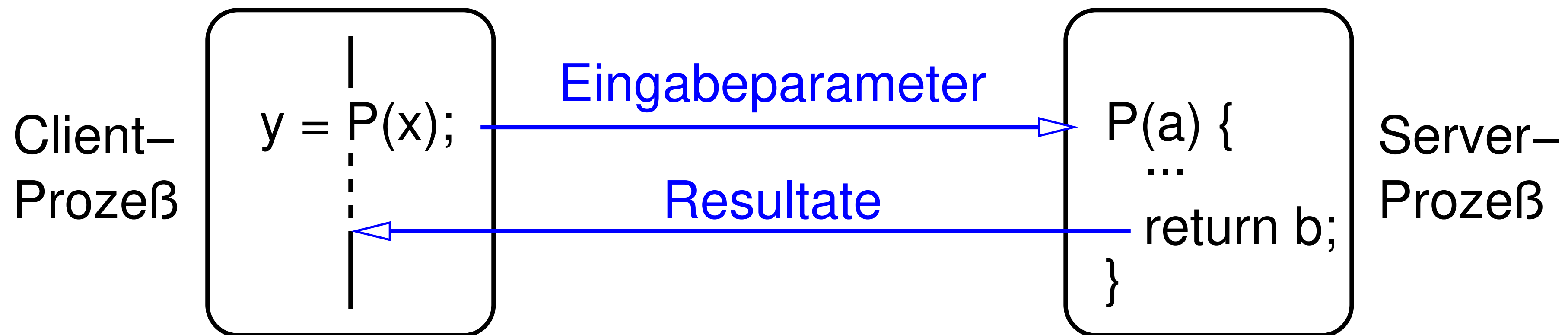
- Sender stellt Nachricht in Warteschlange des Empfängers



- Empfänger nimmt Nachricht an, sobald er bereit ist
- Weitgehende Entkopplung von Sender und Empfänger
- Keine Methoden- oder Prozeduraufrufe
  - Daten werden von der Anwendung verpackt und verschickt
  - keine automatische Antwort-Nachricht

Entfernter Prozeduraufruf (RPC, *Remote Procedure Call*)

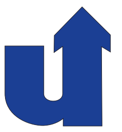
- Ermöglicht einem Client den Aufruf einer Prozedur in einem entfernten Server-Prozeß



- Kommunikation nach Anfrage / Antwort-Prinzip

Entfernter Methodenaufruf (RMI, *Remote Method Invocation*)

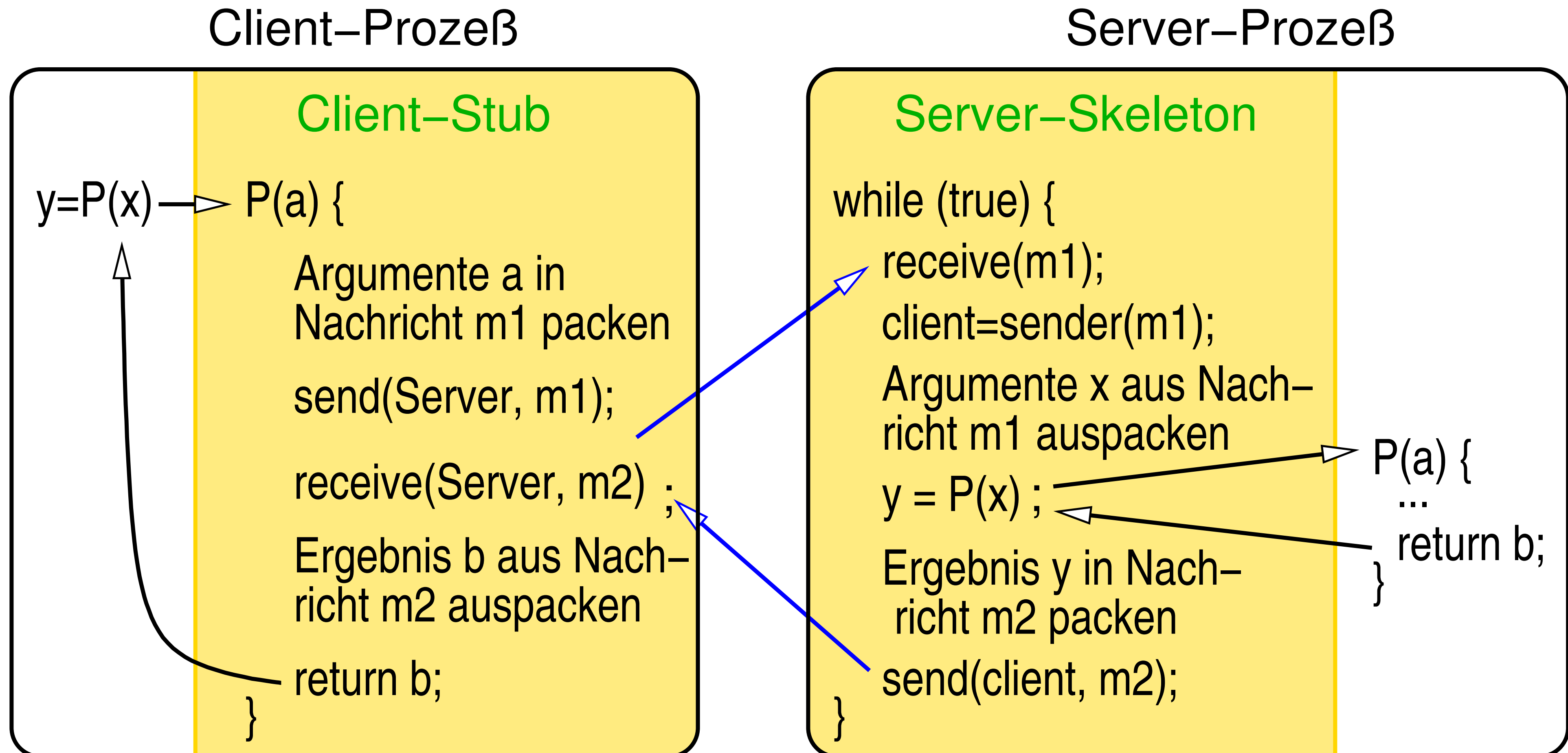
- Ermöglicht einem Objekt, Methoden eines entfernten Objekts aufzurufen
- Prinzipiell sehr ähnlich zu RPC



Gemeinsame Grundkonzepte entfernter Aufrufe

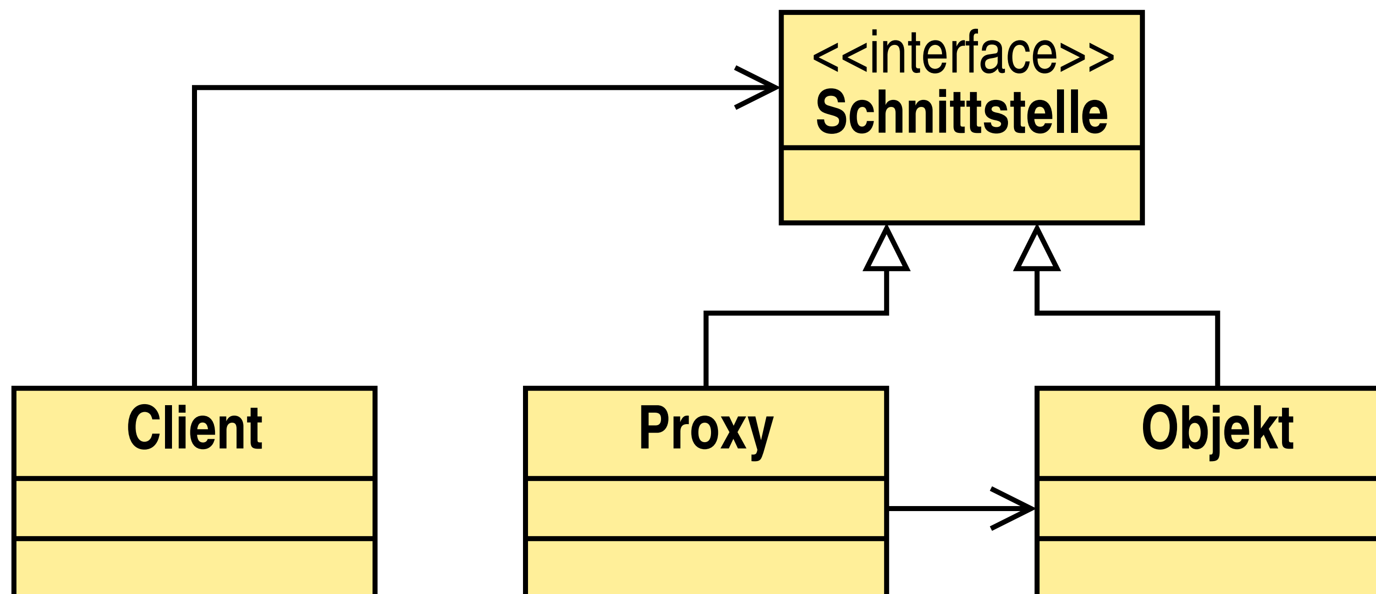
- Client und Server werden durch Schnittstellendefinition entkoppelt
  - legt Namen der Aufrufe, Parameter und Rückgabewerte fest
- Einführung von ***Client-Stubs*** und ***Server-Skeletons*** (bzw. -Stubs) als Zugriffsschnittstelle
  - werden automatisch aus Schnittstellendefinition generiert
    - 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)

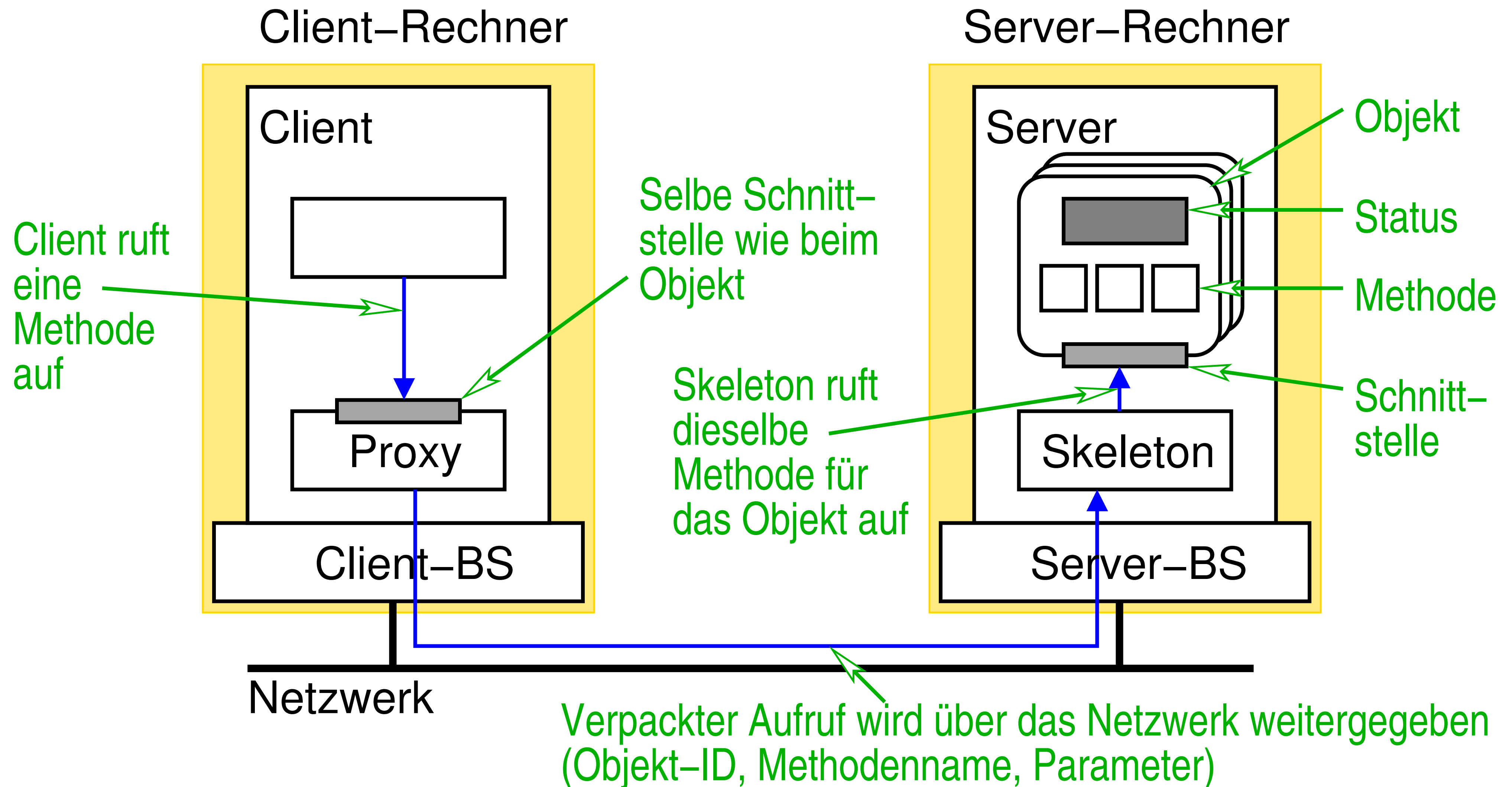


Basis von RMI: Das Proxy-Pattern

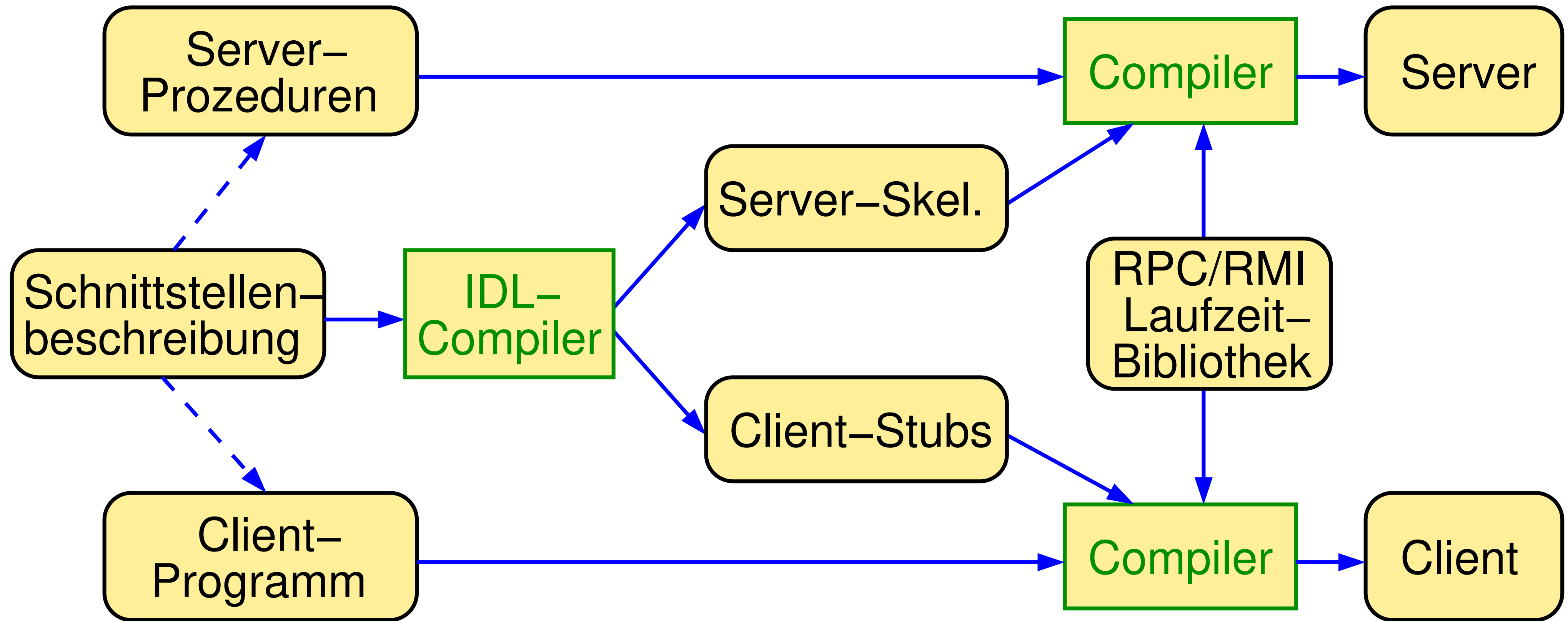
- Client arbeitet mit Stellvertreterobjekt (**Proxy**) des eigentlichen Serverobjekts
  - Proxy und Serverobjekt implementieren dieselbe Schnittstelle
  - Client kennt / nutzt lediglich diese Schnittstelle



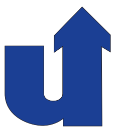
Ablauf eines entfernten Methodenaufrufs



Erstellung eines Client/Server-Programms



- Gilt prinzipiell für alle Realisierungen entfernten Aufrufe

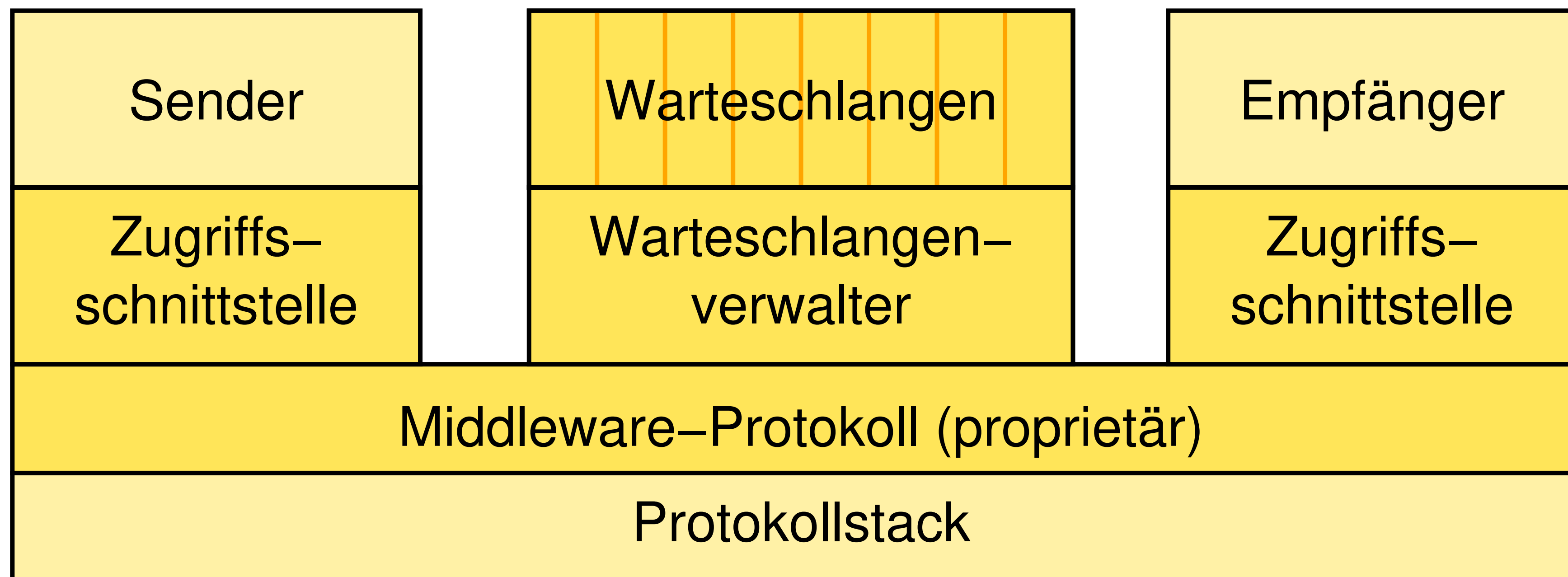


2.2.3 Middleware-Technologien

- Realisieren (mindestens) eines der Programmiermodelle
  - setzen auf offene Standards / standardisierte Schnittstellen
- Entfernter Prozeduraufruf
  - SUN RPC, DCE RPC, Web Services (☞ **CSP: 7**), ...
- Entfernter Methodenaufruf
  - Java RMI (☞ 3), CORBA (☞ **CSP: 3**), ...
- Nachrichtenorientierte Middleware-Technologien
  - MOM: *Message Oriented Middleware*, *Messaging Systeme*
  - vorwiegend für EAI
  - Java Message Service, WebSphereMQ (MQSeries), ...

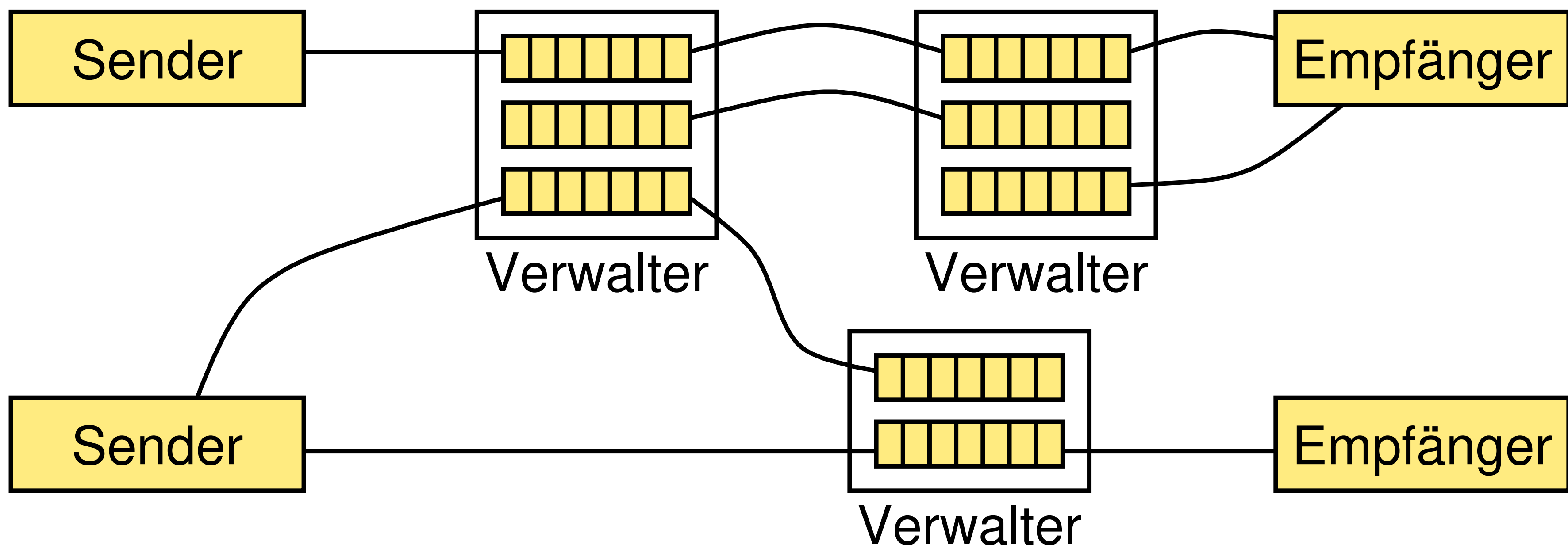
2.2.4 *Message Oriented Middleware* (MOM)

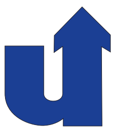
- Middleware-Technologie zum nachrichtenorientierten Modell
- Neben Nachrichtenübermittlung weitere Dienste, v.a. Warteschlangenverwaltung



Wateschlangen-Infrastruktur

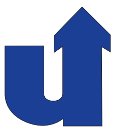
- Zugriff auf Warteschlangen nur lokal möglich
  - lokal: selber Rechner oder selbes Subnetz
- Transport von Nachrichten über Subnetzgrenzen hinweg durch Warteschlangenverwalter (Router)





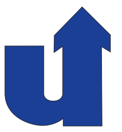
Variante des Nachrichtenaustauschs

- Punkt-zu-Punkt-Kommunikation (*Point-to-Point*)
  - Kommunikation zwischen zwei festgelegten Prozessen
  - einfachstes Modell: asynchrone Kommunikation
  - Erweiterung: *Request-Reply*-Modell
    - ermöglicht synchrone Kommunikation über asynchrone Middleware
- Broadcast-Kommunikation
  - Nachricht wird an alle erreichbaren Empfänger versendet
  - eine Umsetzung: *Publish-Subscribe*-Modell
    - *Publisher* veröffentlichen Nachrichten zu einem Thema
    - *Subscriber* abonnieren bestimmte Themen
    - Vermittlung durch *Broker*



Beispiel: Java Message Service

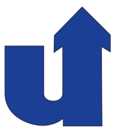
- Teil der Java Enterprise Edition (Java EE)
- Einheitliche Java-Schnittstelle für Dienste von MOM-Servern
- Unterscheidet zwei Rollen:
  - JMS-Provider: der jeweilige MOM-Server
  - JMS-Client: Sender bzw. Empfänger von Nachrichten
- JMS unterstützt:
  - asynchrone Punkt-zu-Punkt-Kommunikation
  - *Request-Reply*-Modell
  - *Publish-Subscribe*-Modell
- JMS definiert jeweils Zugriffsobjekte und Methoden



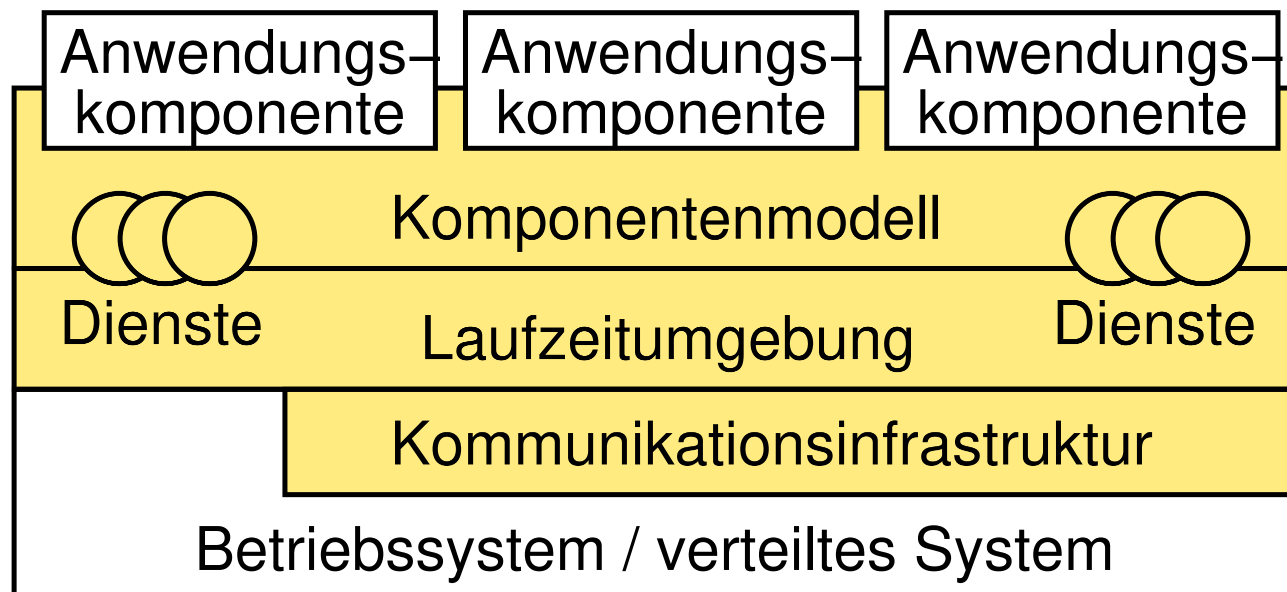
2.2.5 Zusammenfassung

- Aufgaben: Kommunikation, Behandlung der Heterogenität, Fehlerbehandlung
- Programmiermodelle:
  - nachrichtenorientiertes Modell (asynchron)
    - Basis: Nachrichtenwarteschlangen
    - Verfeinerungen:
      - *Request-Reply*-Modell (synchron)
      - *Publish-Subscribe*-Modell (Broadcast)
  - entfernte Prozedur- bzw. Methodenaufrufe
    - synchron: Anfrage und Antwort
    - generierte Stubs für *(Un-)Marshalling*

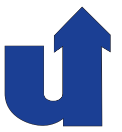
2.3 Anwendungsorientierte Middleware



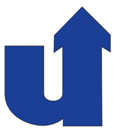
- Setzt auf kommunikationsorientierter Middleware auf
- Erweitert diese um:
  - Laufzeitumgebung
  - Dienste
  - Komponentenmodell



2.3.1 Laufzeitumgebung

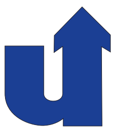


- Setzt auf Knoten-Betriebssystemen des verteilten Systems auf
  - Betriebssystem (BS) verwaltet Prozesse, Speicher, E/A, ...
  - liefert Basisfunktionalität
    - Starten / Beenden von Prozessen, Scheduling, ...
    - Interprozeßkommunikation, Synchronisation, ...
- Laufzeitumgebung erweitert Funktionalität des BS um:
  - verbesserte Ressourcenverwaltung
    - u.a. Nebenläufigkeit, Verbindungsverwaltung
  - Verbesserung der Verfügbarkeit
  - Verbesserte Sicherheitsmechanismen



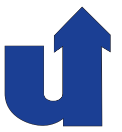
Ressourcenverwaltung

- Geht bei Middleware über die einfache BS-Funktionalität hinaus
  - z.B. unabhängig verwaltete Hauptspeicherbereiche mit individuellen Sicherheitskriterien
  - *Pooling* von Prozessen, Threads, Verbindungen
    - werden auf Vorrat angelegt und bei Bedarf zur Verfügung gestellt
  - möglich, da Middleware spezifisch für bestimmte Klasse von Anwendungen ist
- Ziel: verbesserte Performance, Skalierbarkeit und Verfügbarkeit



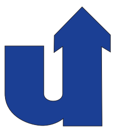
Nebenläufigkeit

- Nebenläufigkeit in diesem Zusammenhang:
  - isolierte parallele Bearbeitung von Aufträgen
- Nebenläufigkeit realisierbar durch Prozesse oder Threads
  - Thread (leichtgewichtige Prozesse): Ablauf„fäden“ innerhalb von Prozessen
    - Threads im selben Prozeß teilen sich alle Ressourcen
  - Vor- und Nachteile:
    - Prozesse: hoher Ressourcenbedarf, nicht gut skalierbar, guter Schutz, bei geringer Nebenläufigkeit
    - Threads: gut skalierbar, kein gegenseitiger Schutz, bei hoher Nebenläufigkeit



Nebenläufigkeit ...

- Middleware übernimmt automatische Erzeugung / Verwaltung von Threads bei nebenläufigen Aufträgen, z.B.
  - *single-threaded*
    - nur ein Thread, sequentielle Abarbeitung
  - *thread-per-request*
    - für jede Anfrage wird ein neuer Thread erzeugt
  - *thread-per-session*
    - für jede Sitzung (Client) wird ein neuer Thread erzeugt
  - *thread pool*
    - feste Anzahl von Threads, eingehende Anfragen werden automatisch verteilt
      - spart Kosten der Threaderzeugung ein
      - beschränkt Ressourcenverbrauch

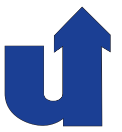


Verbindungsverwaltung

- Verbindungen hier: Endpunkte von Kommunikationskanälen
  - treten an *Tier*-Grenzen (zwischen Prozeßräumen) auf
    - z.B. Client-Server-Schnittstelle, Datenbankzugriff
  - sind im aktiven Zustand einem Prozeß / Thread zugeordnet
  - benötigen Ressourcen (Speicher, Prozessorzeit)
  - Auf- und Abbau ist aufwendig
- Zur Schonung der Ressourcen: *Pooling* von Verbindungen
  - Verbindungen werden auf Vorrat initialisiert und in Pool gestellt
  - jeder Thread / Prozeß erhält bei Bedarf eine Verbindung
  - nach Verwendung: zurückstellen in Pool

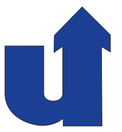
Verfügbarkeit

- Anforderung an die Anwendung,
  Umsetzung aber vor allem durch die Umgebung
- Ausfallzeiten entstehen durch
  - Ausfall einer Hard- oder Softwarekomponente
  - Überlastung einer Hard- oder Softwarekomponente
  - Wartung einer Hard- oder Softwarekomponente
- Häufige Technik zur Sicherung der Verfügbarkeit: Cluster
  - Replikation von Hard- und Software
  - Cluster tritt nach außen als eine Einheit auf
  - zwei Arten: Fail-over Cluster / Load-balancing Cluster



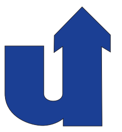
Sicherheit

- Verteilte Anwendungen sind durch ihre Verteiltheit angreifbar
- Middleware unterstützt unterschiedliche Sicherheitsmodelle
- Sicherheitsanforderungen:
  - **Authentifizierung**:
    - stellt Identität des Anwenders / einer Komponente sicher
    - z.B. durch Paßwort-Abfrage (bei Benutzer) oder kryptographische Techniken u. Zertifikate (bei Komponenten)
  - **Autorisierung**
    - Festlegung von Zugriffsrechten für Benutzer auf konkrete Dienste
      - bzw. auch feiner: Methoden und Attribute
    - Überprüfung setzt sichere Authentifizierung voraus



Sicherheit ...

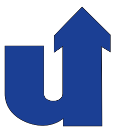
- Sicherheitsanforderungen ...:
  - **Vertraulichkeit**
    - Abhören von Daten während der Übertragung im Netz ist nicht möglich
    - Technik: Verschlüsselung
  - **Integrität**
    - Übertragene Daten können nicht unbemerkt verändert werden
    - Techniken: kryptographische Prüfsumme (*Message Digest, Fingerprint*), digitale Signatur
      - digitale Signatur stellt auch Authentizität des Absenders sicher



Sicherheit ...

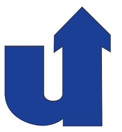
- Sicherheitsmechanismen:
  - Verschlüsselung
    - symmetrisch (z.B. IDEA, AES)
      - selber Schlüssel zum Ver- und Entschlüsseln
    - asymmetrisch (*Public-Key*-Verfahren, z.B. RSA)
      - öffentlicher Schlüssel zum Verschlüsseln
      - privater Schlüssel zum Entschlüsseln
  - Digitale Signatur
    - sichert Integrität einer Nachricht und Authentizität des Senders sowie Verbindlichkeit
  - Zertifikat
    - beglaubigt Zusammengehörigkeit von öffentlichem Schlüssel und Person (bzw. Komponente)

2.3.2 Dienste



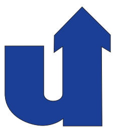
Namensdienst (Verzeichnisdienst) (☞ 4)

- Veröffentlichung von verfügbaren Diensten
  - im Intranet oder Internet
- Zuordnung von Namen zu Referenzen (Adressen)
  - Name dient als eindeutiger / unveränderlicher Identifikator
  - Client kann über den Namen die Adresse eines Servers erfragen
    - Adresse kann sich z.B. bei Neustart ändern
  - Ziel: Entkopplung von Client und Server
- Beispiele: JNDI, RMI Registry, CORBA Interoperable Naming Service, UDDI Registry, LDAP Server, ...



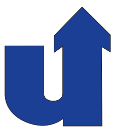
Sitzungsverwaltung

- In interaktiven Systemen: jeder Instanz eines Clients wird eine eigene **Sitzung** (***Session***) zugeordnet
  - gelöscht beim Beenden des Clients oder beim Abmelden
- Sitzung speichert alle relevanten Daten (im Hauptspeicher)
  - z.B. Kennung des Anwenders, Browsertyp, „Warenkorb“, ...
  - Speicherung im Server oder im Client
  - transiente Daten: werden am Ende der Sitzung gelöscht
  - persistente Daten: werden am Ende der Sitzung auf Datenträger geschrieben (Datenbank)
- Middleware realisiert / unterstützt die Zuordnung von Anfragen zu Sitzungen (oft transparent)
  - z.B. Cookies, HTTPSessions, Session Beans, ...



Transaktionsverwaltung (☞ 7.4)

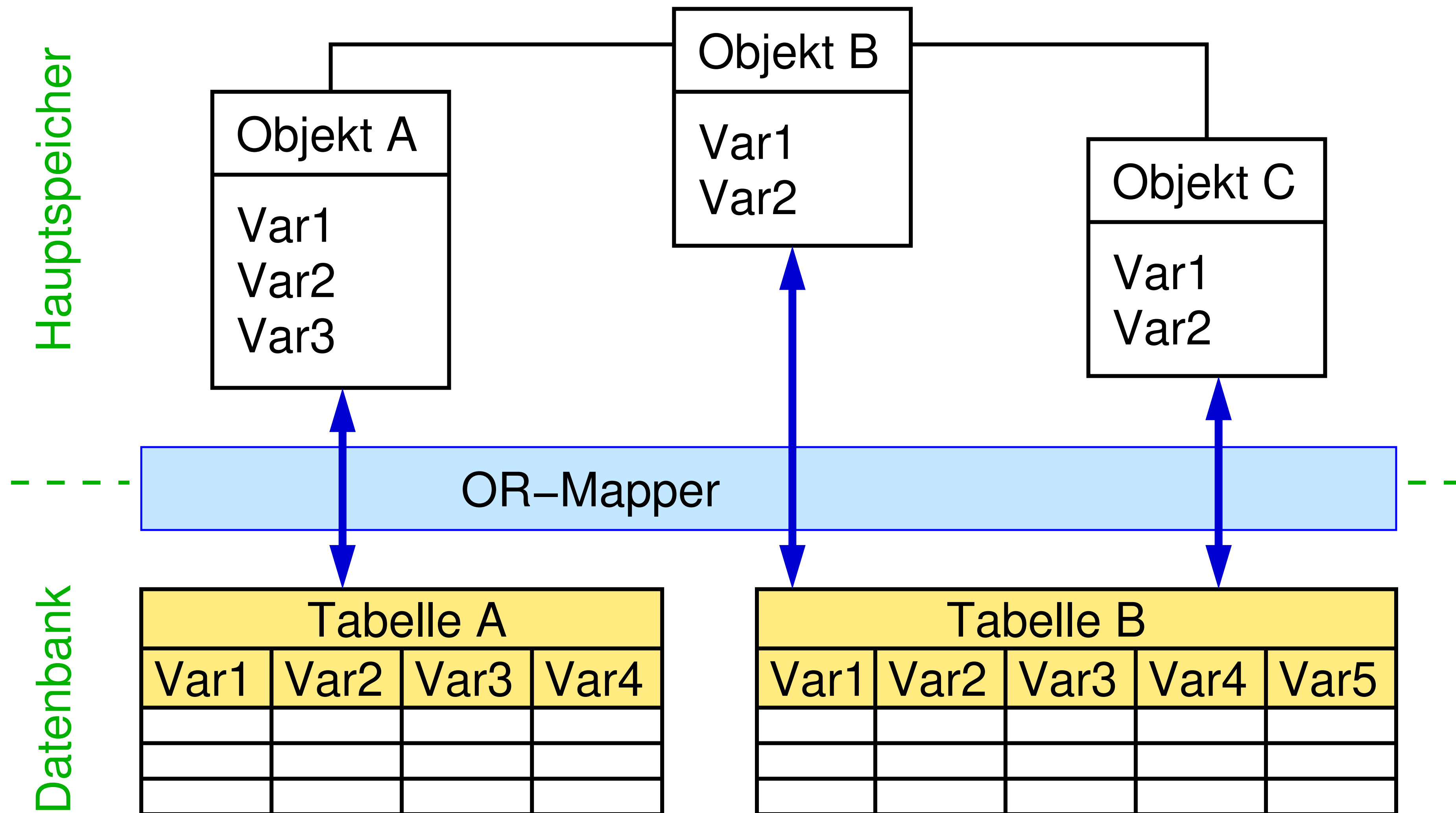
- Dienst für interaktive, datenzentrierte Anwendungen
  - Konsistenz / Integrität der Daten ist wichtig
  - d.h. gesamter (ggf. verteilter) Datenbestand muß einen in sich gültigen Zustand repräsentieren
- Typischer Ablauf in Anwendungen:
  1. Client fordert Daten an
  2. Client verändert die Daten
  3. Client fordert das Rückschreiben der Daten an
  - Problem: die Schritte 1 - 3 könnten von zwei Clients genau gleichzeitig durchgeführt werden
- Transaktionsverwaltung erlaubt Durchführung einer Folge von Aktionen als atomare Einheit



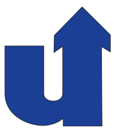
Persistenzdienst

- Persistenz: Gesamtheit aller Maßnahmen zur dauerhaften Speicherung von Hauptspeicher-Daten
- Persistenzdienst: intelligente Schnittstelle zur Datenbank
  - in Middleware integriert oder als eigenständige Komponente
  - neben Transaktionsverwaltung wichtigster Dienst für daten-zentrierte Anwendungen
- Häufigste Art: objektrelationaler Mapper (OR-Mapper)
  - bildet Objekte im Hauptspeicher auf Tabellen in relationaler Datenbank ab
  - Regeln werden von Anwendungsentwickler vorgegeben

Persistenzdienst ...

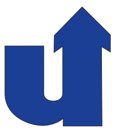


2.3.3 Komponentenmodell



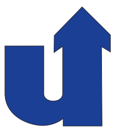
- Komponenten: „große“ Objekte zur Strukturierung von Anwendungen
- Ein Komponentenmodell definiert:
  - Komponentenbegriff
    - Struktur und Eigenschaften der Komponenten
    - vorgeschriebene und optionale Schnittstellen
  - Schnittstellenverträge
    - wie interagieren Komponenten untereinander und mit der Laufzeitumgebung?
  - Komponenten-Laufzeitumgebung
    - Verwaltung des Lebenszyklus der Komponenten
    - implizite Bereitstellung von Diensten: Komponente teilt nur Anforderungen mit (z.B. Persistenz)

2.3.4 Middleware-Technologien



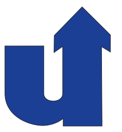
- *Object Request Broker* (ORB)
  - verteilte Objekte, entfernte Methodenaufrufe
  - Vielzahl an Diensten, nur grundlegende Laufzeitumgebung
  - Beispiel: CORBA
- *Application Server*
  - Fokus: Unterstützung der Anwendungslogik (*Middle-Tier*)
  - Dienste, Laufzeitumgebung und Komponentenmodell
  - heute nur noch als Teil einer Middleware-Plattform
- Middleware-Plattformen
  - Erweiterung von Appl. Servern: Unterstützung aller *Tiers*
    - neben verteilten Anwendungen auch EAI
  - Beispiele: Java EE / EJB, .NET / COM, CORBA 3.0 / CCM

2.3.5 Zusammenfassung



Anwendungsorientierte Middleware

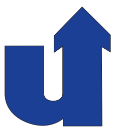
- Laufzeitumgebung
  - Ressourcenverwaltung, Verfügbarkeit, Sicherheit
- Dienste
  - Namensdienst, Sitzungsverwaltung, Transaktionsverwaltung, Persistenzdienst
- Komponentenmodell
  - Komponentenbegriff, Schnittstellenverträge, Laufzeitumgebung



Verteilte Systeme

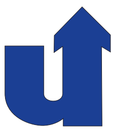
SoSe 2018

3 Verteilte Programmierung mit Java RMI



Inhalt

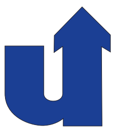
- Einführung
- *Hello World* mit RMI
- RMI im Detail
  - Klassen und Interfaces, Stubs, Namensdienst, Parameterübergabe, *Factories*, *Callbacks*, ...
- *Deployment*: Laden entfernter Klassen
  - Java *Remote Class Loader* und *Security Manager*



Literatur

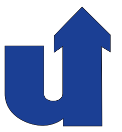
- WWW-Dokumentation und Tutorials von Oracle
  - `http://docs.oracle.com/javase/6/docs/api/index.html?java/rmi/package-summary.html`
  - `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`
- Hammerschall: Kap. 5.2
- Farley, Crawford, Flanagan: Kap. 3
- Horstmann, Cornell: Kap. 5
- Orfali, Harkey: Kap. 13
- Peter Ziesche: Nebenläufige & verteilte Programmierung, W3L-Verlag, 2005. Kap. 8

3.1 Einführung



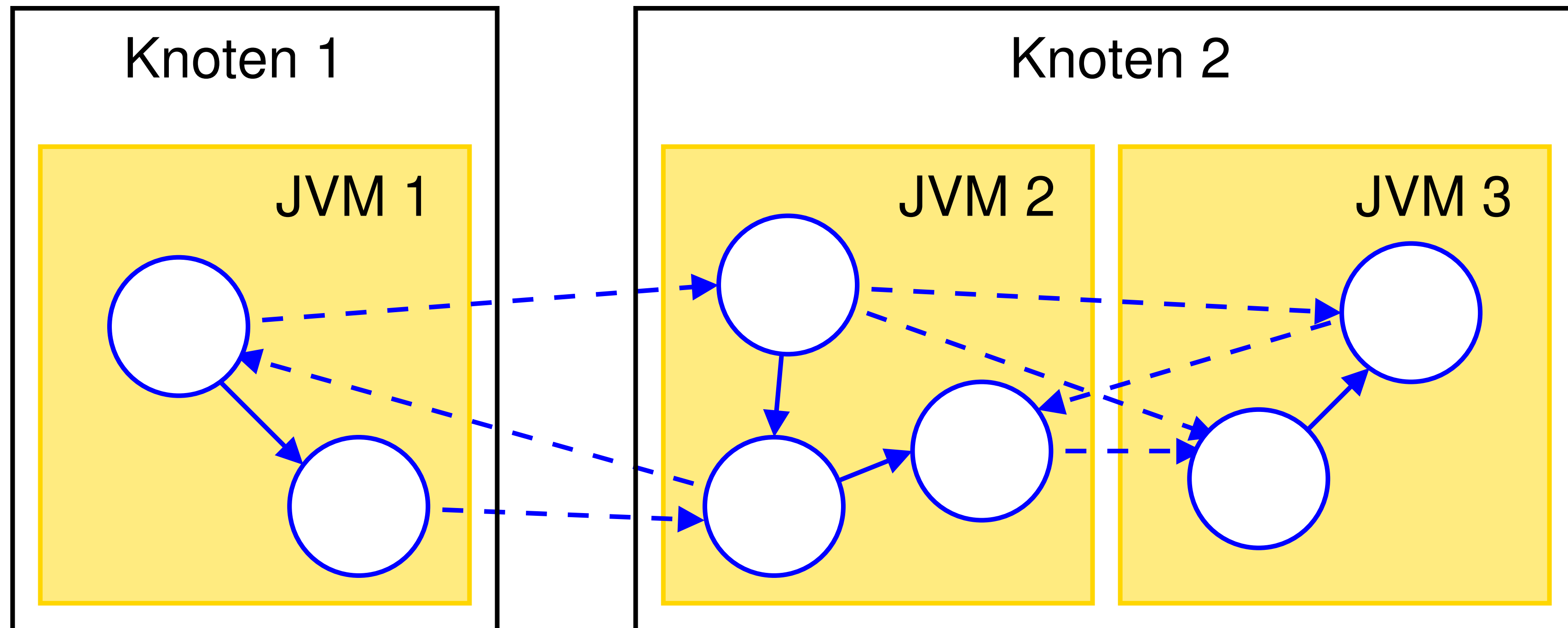
- Java RMI ist fester Bestandteil von Java
  - erlaubt Nutzung entfernter Objekte
- Elemente von Java RMI:
  - entfernte Objektimplementierungen
  - Client-Schnittstellen (Stubs) zu entfernten Objekten
  - Namensdienst, um Objekte im Netz ausfindig zu machen
  - Dienst zum automatischen Erzeugen (Aktivieren) von Objekten
  - Kommunikationsprotokoll
- Java-Schnittstellen für die ersten vier Elemente
  - im Paket `java.rmi` und dessen Unterpaketen

3.1 Einführung ...

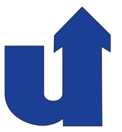


- Java RMI setzt voraus, daß alle Objekte (d.h. Client und Server) in Java programmiert sind
  - im Unterschied z.B. zu CORBA
- Vorteil: nahtlose Integration in die Sprache
  - entfernte Objekte sind (fast!) genauso wie lokale zu verwenden
  - incl. verteilter *Garbage Collection*
- Integration von Objekten in anderen Programmiersprachen:
  - „Einwickeln" in Java-Code über Java Native Interface (JNI)
  - Nutzung von RMI/IIOP: Interoperabilität mit CORBA
    - direkte Kommunikation von RMI- und CORBA-Objekten

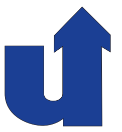
Verteilte Objekte



- *Remote*-Referenzen genau wie lokale Referenzen verwendbar
- Objekte können in Client- und Server-Rolle auftreten

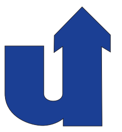


3.1.1 RMI-Architektur



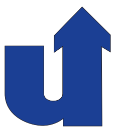
Stub/Skeleton - Schicht

- Stub: lokales Proxy-Objekt für das entfernte Objekt
- Skeleton: nimmt Aufrufe entgegen und leitet sie an das richtige Objekt weiter
- Stub- und Skeleton-Klassen werden automatisch aus Schnittstellendefinition (Java Interface) generiert
- Ab JDK 1.2: Skeleton-Klasse ist generisch
  - Skeleton nutzt *Reflection*-Mechanismus von Java, um Methoden des Server-Objekts aufzurufen
  - *Reflection* erlaubt Abfrage der Methodendefinitionen einer Klasse und generischen Methodenaufruf zur Laufzeit
- Ab JDK 1.5: Stub-Klassen werden zur Laufzeit erzeugt
  - durch die Java-Klasse `Proxy`



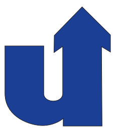
Remote Reference - Schicht

- Definiert Aufrufsemantik von RMI
  - In JDK 1.1: nur Unicast, Punkt-zu-Punkt
    - Aufruf wird an genau ein existierendes Objekt geleitet
  - Ab JDK 1.2 auch aktivierbare Objekte
    - Objekt wird ggf. erst (re-)aktiviert
      - neues Objekt, Zustand wird von Festplatte restauriert
  - Auch möglich: Multicast-Semantik
    - Proxy sendet Anfrage an Menge von Objekten und gibt erste Antwort zurück
- Außerdem: Verbindungsmanagement, verteilte *Garbage-Collection*



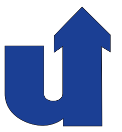
Transportschicht

- Verbindungen zwischen JVMs
  - Basis: TCP/IP - Ströme
- Eigenes Protokoll: *Java Remote Method Protocol* (JRMP)
  - erlaubt Tunneln der Verbindung über HTTP (wg. *Firewalls*)
  - erlaubt Definition eigener *Socket-Factory*, z.B. zur Verwendung von *Transport Layer Security* (TLS bzw. SSL)
- Ab JKD 1.3 auch RMI-IIOP
  - benutzt IIOP (*Internet Inter-ORB Protocol*) von CORBA
  - damit: direkte Interoperabilität mit CORBA-Objekten

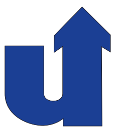


3.1.2 RMI-Dienste

- **Namensdienst: RMI *Registry***
  - registriert *Remote*-Referenzen auf RMI-Objekte unter frei wählbaren eindeutigen Namen
  - ein Client kann sich dann zu einem Namen die zugehörige Referenz holen
    - technisch: *Registry* sendet serialisiertes Proxy-Objekt (Client-Stub) an den Client
    - ggf. wird auch der Ort der benötigten `class`-Dateien mitübertragen (siehe **3.4.1**)
  - RMI kann auch mit anderen Namensdiensten verwendet werden, z.B. über JNDI (Java Naming and Directory Interface)

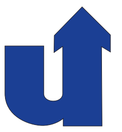


- ***Object Activation Service***
  - normalerweise: *Remote*-Referenz auf RMI-Objekt ist nur solange gültig, wie das Objekt existiert
    - bei Absturz des Servers oder der Server-JVM: Objekt-Referenzen werden ungültig
      - Referenzen ändern sich beim Neustart!
  - RMI *Activation Service* eingeführt mit JDK 1.2
  - startet Server-Objekte auf Anfrage eines Clients
    - Server-Objekt muß Aktivierungsmethode beim RMI *Activation-Daemon* registrieren



Verteilte *Garbage-Collection*

- automatische *Garbage-Collection* von Java funktioniert auch für *Remote*-Objekte
- serverseitige JVM verwaltet Liste von *Remote*-Referenzen auf Objekte
- Referenzen werden auf Zeit „geleast“
- Referenzzähler des Objekts wird erniedrigt, wenn
  - Client die Referenz löscht (z.B. Ende der Lebensdauer der Referenzvariable), oder
  - Client den *Lease* nicht rechtzeitig verlängert
    - Grund: *Remote Reference*-Schicht kann Objekt nicht explizit „abmelden“, wenn Client abstürzt
    - Standardeinstellung: 10 Min.



Struktur:

Client–JVM

Server–JVM

Interface

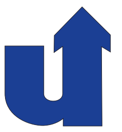
```
interface Hello {
   String sayHello();
}
```

Client–Klasse

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

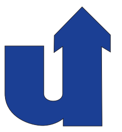
Server–Klasse

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



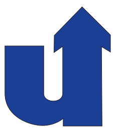
Ablauf der Entwicklung:

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!)

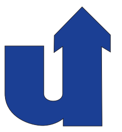


Hello-World Interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
String sayHello() throws RemoteException;
}
```

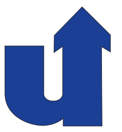
Marker–Schnittstelle , enthält keine Methoden, markiert Interface als RMI–Schnittstelle

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



Implementierung der Server-Klasse

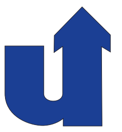
- Eine Klasse, die *remote* nutzbar sein soll, muß:
  - ein festgelegtes *Remote*-Interface implementieren
  - i.d.R. von `java.rmi.server.UnicastRemoteObject` abgeleitet werden
    - definiert Aufrufsemantik: Punkt-zu-Punkt
  - einen Konstruktor besitzen, der `RemoteException` werfen kann
    - Erzeugung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



Hello-World Server (1)

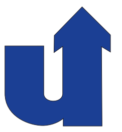
```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

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

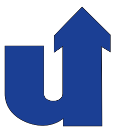


Hello-World Server (2)

```
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

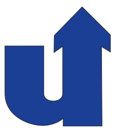
Registrieren des Server–Objekts unter dem Namen "Hello–Server" beim Name–Server (RMI–Registry, lokaler Rechner, Port 1099)



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

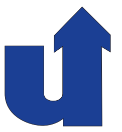


Hello-World Client

```
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) {
...
}
}
}
```

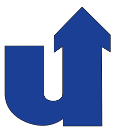
Objektreferenz vom Name–Server holen

Aufruf der Methode des entfernten Objekts



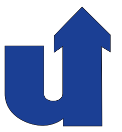
Übersetzen und Starten des Systems

- 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`



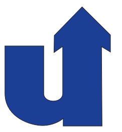
Übersetzen und Starten des Systems ...

HelloClient.java
Hello.java
HelloServer.java
javac
javac
HelloClient.class
Hello.class
Hello.class
HelloServer.class
rmic
HelloServer_Stub.class
bis JDK 1.4
Client–Seite
Server–Seite



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`



Ablauf des Beispiels

Client–
Rechner

Server–
Rechner

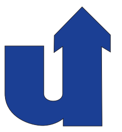
rmiregistry

Test	
Foo	

HelloClient

HelloServer

Remote
Objekt



Ablauf des Beispiels

Client–Rechner | Server–Rechner

rmiregistry

Test

Foo

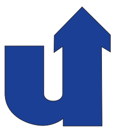
Name und Stub–Objekt (serialisiert)

HelloClient

HelloServer

Remote Objekt

`Naming.rebind(...)`



Ablauf des Beispiels

Client–Rechner | Server–Rechner

rmiregistry

Test	
Foo	
Hello–Server	

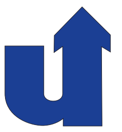
HelloClient

Name und Stub–Objekt (serialisiert)

HelloServer

Remote Objekt

`Naming.rebind(...)`



Ablauf des Beispiels

Client–
Rechner

Server–
Rechner

rmiregistry

Test	
Foo	
Hello–Server	

Name

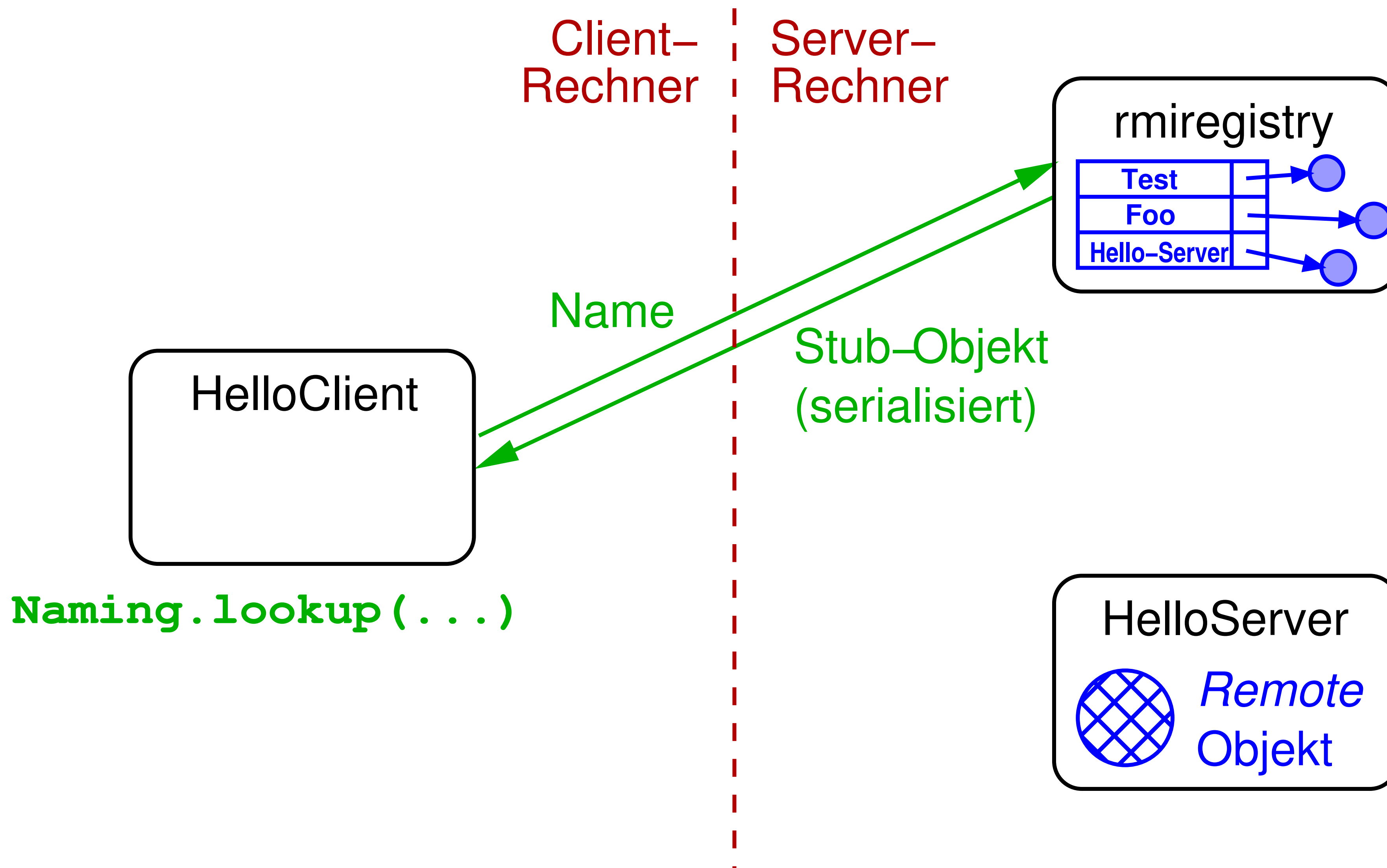
HelloClient

`Naming.lookup(...)`

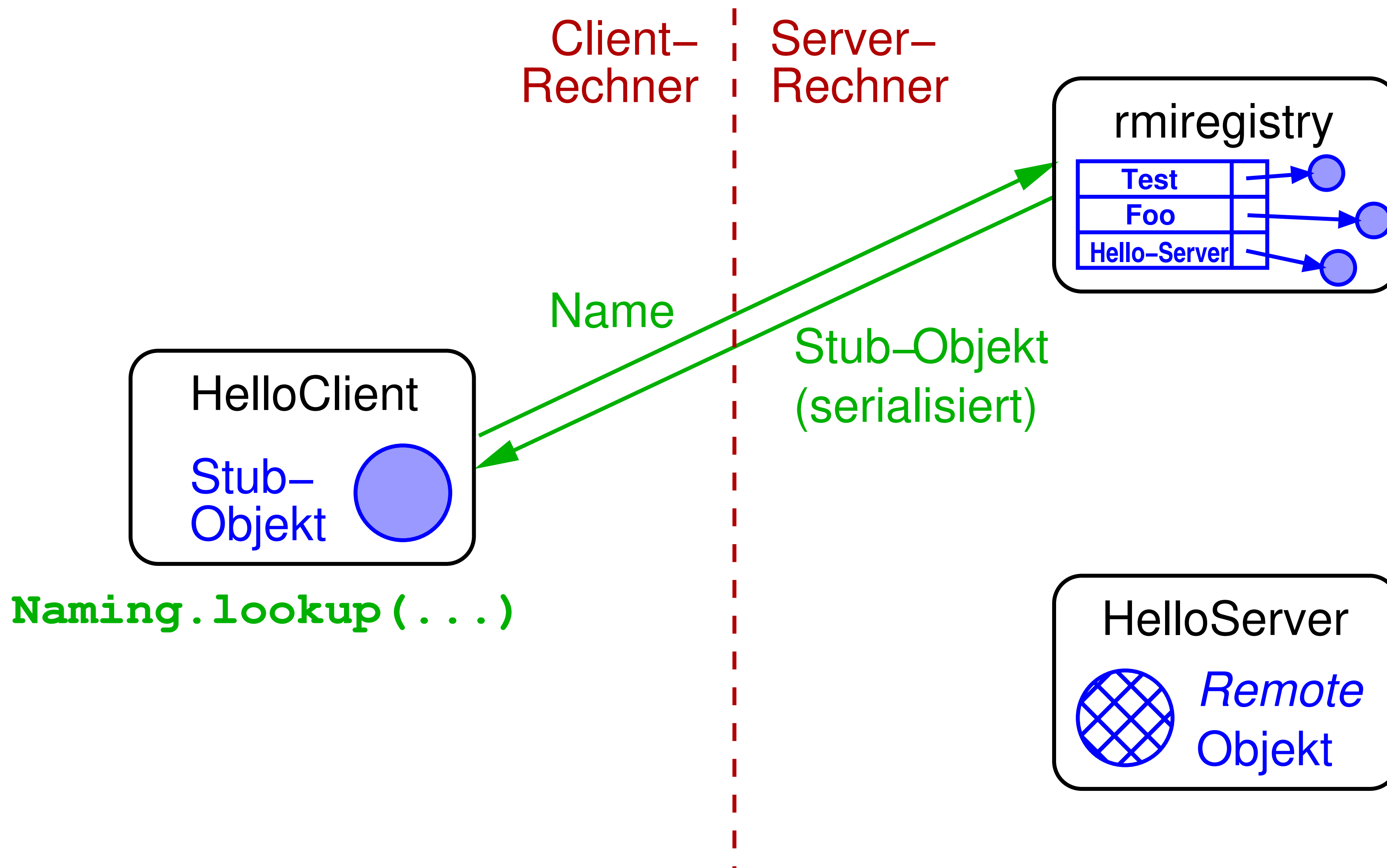
HelloServer

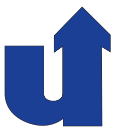
Remote Objekt

Ablauf des Beispiels



Ablauf des Beispiels





Ablauf des Beispiels

Client–
Rechner

Server–
Rechner

rmiregistry

Test	
Foo	
Hello–Server	

HelloClient

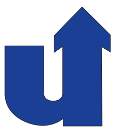
Stub–
Objekt

`obj.sayHello()`

HelloServer

Remote
Objekt

3.3 RMI im Detail



3.3.1 Klassen und Interfaces

java.lang

Object

java.io

IOException

<<interface>>
Remote

RemoteException

Naming

RemoteObject

RemoteServer

RemoteStub

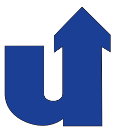
UnicastRemoteObject

java.rmi
java.rmi.server

uses

HelloClient

HelloServer

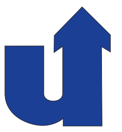


Schnittstelle `Remote`

- Jedes *Remote*-Objekt muß diese Schnittstelle implementieren
- Bietet keine Methoden, dient nur als Markierung

Klasse `RemoteException`

- Superklasse für alle Exceptions, die vom RMI-System ausgelöst werden können, z.B. bei
  - Kommunikationsfehlern (Server nicht erreichbar, ...)
  - Fehler beim *(Un-)Marshalling*
  - Protokollfehler
- Jede *Remote*-Methode muß `RemoteException` (oder Basisklasse davon) in `throws`-Klausel spezifizieren

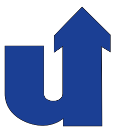


Klasse `RemoteObject`

- Basisklasse für alle *Remote*-Objekte
- Definiert die Methoden `equals`, `hashCode` und `toString` neu
- `toStub()` gibt Referenz auf Stub-Objekt zurück
- `getRef()` liefert *Remote*-Referenz (= Java-Klasse)
  - wird vom Stub zum Aufruf von Methoden genutzt

Klasse `RemoteServer`

- Basisklasse für alle Server-Implementierungen
  - `UnicastRemoteObject`, `Activatable`
- Methode `getClientHost()`: Hostadresse des Clients des momentanen RMI-Aufrufs
- `setLog()` und `getLog()`: Protokollierung von RMI-Aufrufen

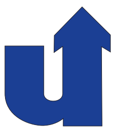


Klasse `UnicastRemoteObject`

- realisiert *Remote*-Objekt mit folgenden Eigenschaften:
  - Referenzen zum Objekt nur gültig, solange Server-Prozeß (JVM) noch läuft
  - Client-Aufruf wird an genau ein Objekt geleitet (über TCP-Verbindung), keine Replikation
- Konstruktor erlaubt Festlegung von Port und *Socket-Factories*
  - damit z.B. Verbindungen über TLS/SSL realisierbar
- Klassenmethode `exportObject()` macht Objekt per RMI verfügbar
- Klassenmethode `unexportObject()` hebt Verfügbarkeit auf

Klasse `RemoteStub`

- Basisklasse für alle Client-Stubs

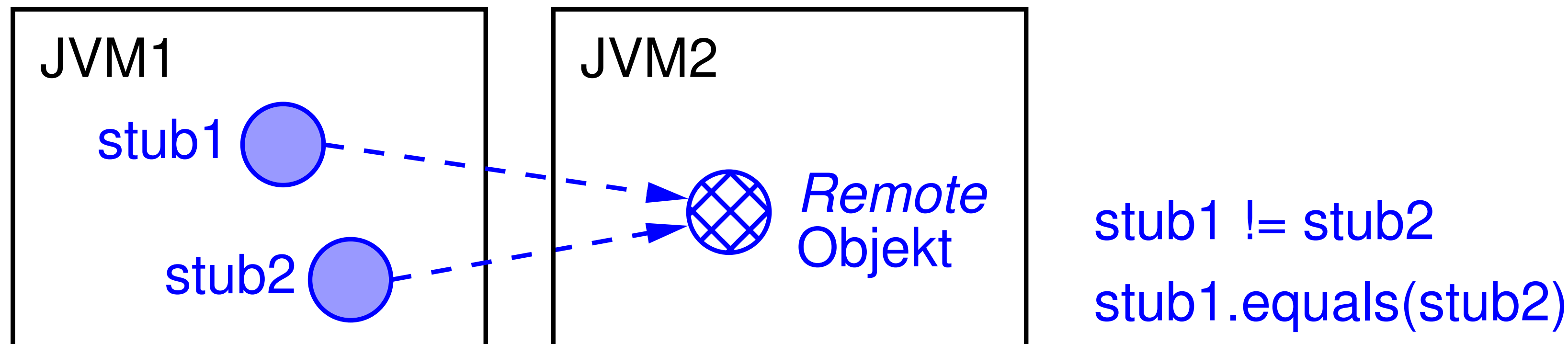


Klasse `Naming`

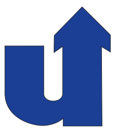
- Erlaubt einfachen Zugriff auf RMI-*Registry*
- Wichtige Methoden:
  - `bind()` / `rebind()`: registriert Objekt unter gegebenem Namen
  - `lookup()`: liefert Objektreferenz zu einem Namen
- Namen sind im URL-Format angegeben
  - legen auch Host und Port der RMI-*Registry* fest
  - Aufbau der URL:

3.3.2 Besonderheiten von *Remote*-Klassen

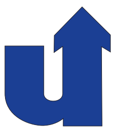
- Vergleich von entfernten Objekten
  - Vergleich mit `==` bezieht sich ausschließlich auf die Stub-Objekte
    - Ergebnis ist `false`, selbst wenn beide Stubs auf dasselbe *Remote*-Objekt verweisen
  - Vergleich mit `equals()` liefert `true` genau dann, wenn beide Stubs auf dasselbe *Remote*-Objekt verweisen



3.3.2 Besonderheiten von *Remote*-Klassen ...



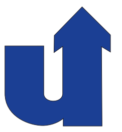
- Methode `hashCode()`
  - u.a. von *Container*-Klassen `HashMap`, `HashSet` verwendet
  - Hash-Code wird nur aus Objektidentifikator des *Remote*-Objekts berechnet
    - selbes *Remote*-Objekt ⇒ selber Hash-Code
    - Inhalt des Objekts bleibt unberücksichtigt
  - konsistent mit Verhalten von `equals()`
- Klonen von Objekten
  - klonen des *Remote*-Objekts ist über den Aufruf von `clone()` des Stubs nicht möglich
  - klonen von Stubs ist nicht notwendig / sinnvoll



3.3.3 Parameterübergabe

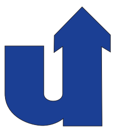
- Übergabe von Parametern an *Remote*-Methoden erfolgt
  - entweder über *call-by-value*
  - oder über *call-by-reference*
- Genutzter Mechanismus ist abhängig vom Typ des Parameters
- Entscheidung wird z.T. erst zur Laufzeit getroffen!

- Rückgabe des Ergebnisses folgt selben Regeln wie Parameterübergabe



Parameterübergabe bei lokalen Methoden

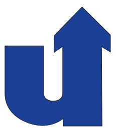
- Java kennt zwei Arten von Typen:
  - **Werttypen**: einfache Datentypen
    - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`
    - werden an lokale Methoden *by value* übergeben
    - d.h. Methode erhält eine Kopie des Werts
  - **Referenztypen**: Klassen (incl. `String` und Arrays)
    - werden an lokale Methoden *by reference* übergeben
    - d.h. Methode arbeitet auf dem Originalobjekt, kann Objekt ggf. auch verändern



Parameterübergabe bei *Remote*-Methoden

- Werttypen: werden immer *by value* übergeben
- Referenztypen: abhängig vom konkreten Objekt
  - Objekt ist serialisierbar: *call-by-value*
  - Objekt gehört zu Klasse, die `Remote` Schnittstelle implementiert: *call-by-reference*
  - keines von beiden: Fehler (`java.rmi.MarshalException`)
  - beides: ??! (dieser Fall ist zu vermeiden!)

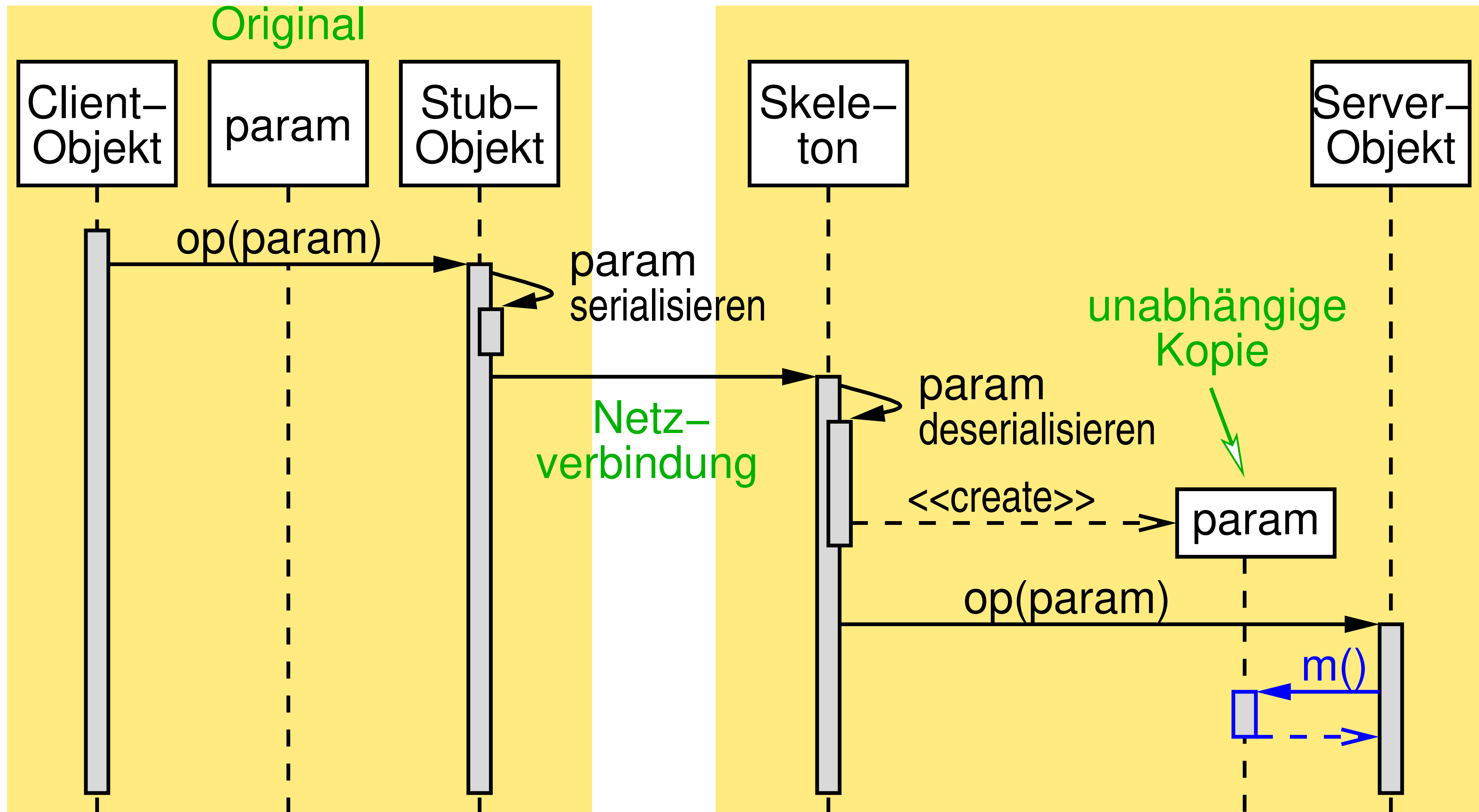
  - Entscheidung erfolgt erst zur Laufzeit

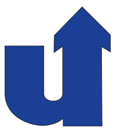


Serialisierbare Objekte

- Klasse muß Schnittstelle `java.io.Serializable` implementieren
- Serialisierbare Objekte können über ein Netz übertragen werden
  - nur die Daten werden übertragen, der Code (`class`-Datei) muß beim Empfänger zur Verfügung stehen!
- *Default*-Serialisierung von Java:
  - alle Attribute des Objekts werden serialisiert und übertragen
  - rekursives Verfahren!
  - Voraussetzung: alle Attribute und alle Basisklassen sind serialisierbar
- Eigene Serialisierung ist realisierbar:
  - Implementierung der Methoden `writeObject` und `readObject`

Übergabe eines serialisierbaren Objekts

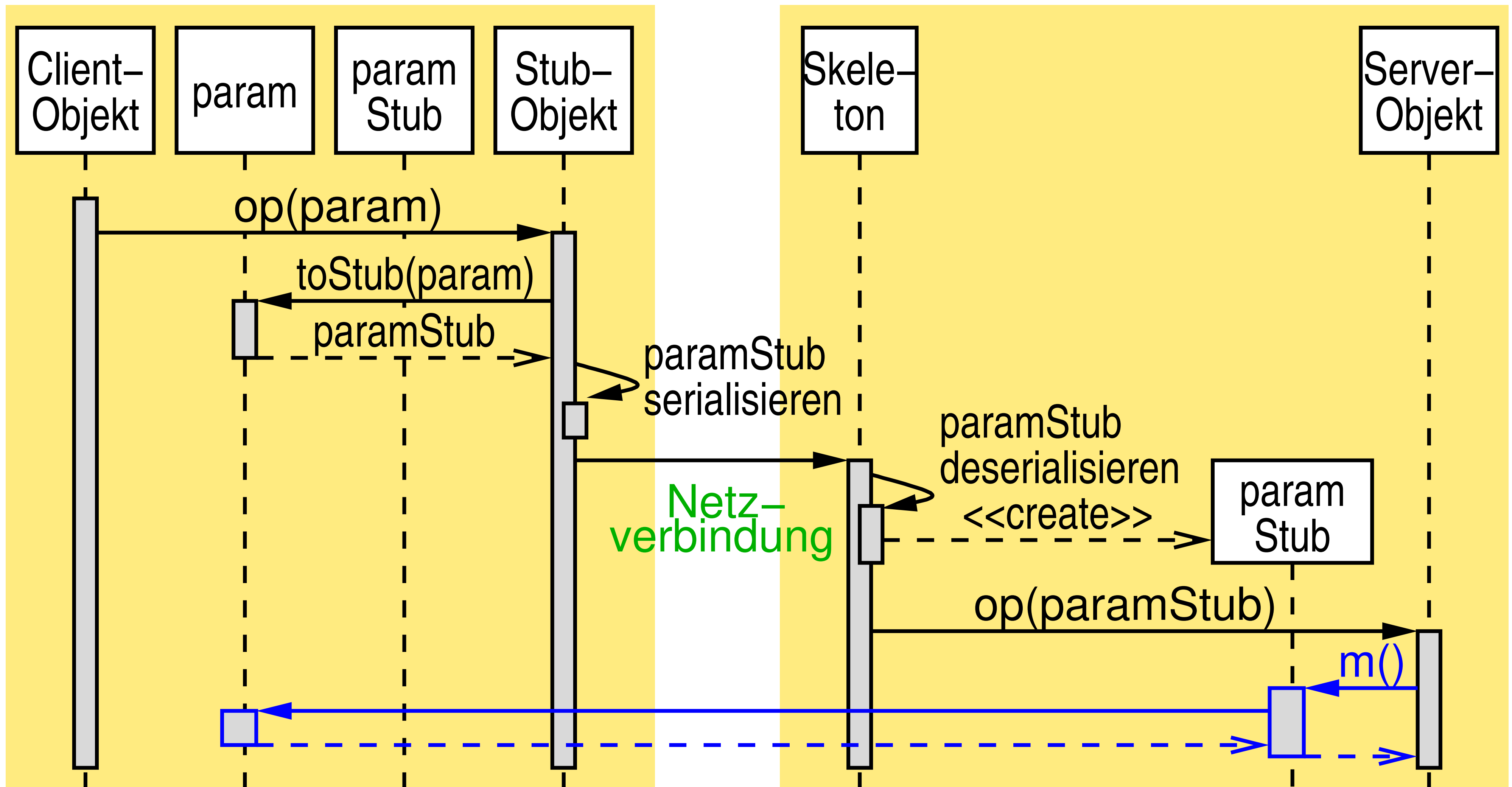


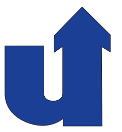


Remote Objekte

- Klasse des Parameter-Objekts muß eine Schnittstelle implementieren, die von `Remote` erbt
  - Parametertyp muß diese Schnittstelle sein
  - Klasse typisch von `UnicastRemoteObject` abgeleitet
- Übertragen wird ein serialisiertes Stub-Objekt
  - bis JDK 1.4 muß Stub-Klasse durch `rmic` generiert werden und beim Server zur Verfügung stehen
  - ab JDK 1.5: dynamische Erzeugung der Stub-Klasse
- Falls Server Methoden des Parameter-Objekts aufruft:
  - Aufrufe werden per RMI an das Originalobjekt geleitet

Übergabe eines *Remote*-Objekts

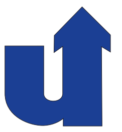




Beispiele

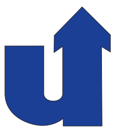
- Siehe WWW:
  - *Hello-World* mit *call-by-value* Parameter
  - *Hello-World* mit *call-by-reference* Parameter

3.3.3 Parameterübergabe ...



Arrays und Container-Objekte

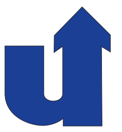
- Arrays und Container-Objekte (aus dem *Java Collection Framework*, `java.util`) sind serialisierbar
  - werden also beim Empfänger neu erzeugt
- Für die Elemente des Arrays / Containers gelten dieselben Regeln wie für einfache Parameter
  - bei gemischtem Inhalt: Elemente werden je nachdem *by value* oder *by reference* übergeben



3.3.4 *Factories*

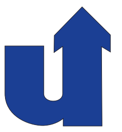
- Häufig: Client erhält über RMI-*Registry* Referenz eines entfernten Objekts, das Referenzen auf weitere Objekte liefert
  - bzw. diese Objekte neu erzeugt

  Bezeichnung: *Factory*-Objekt bzw. *Factory*-Klasse
- Beispiel: Server für Bankkonten
  - Registrierung aller Konto-Objekte bei RMI-*Registry* nicht sinnvoll
  - stattdessen: Registrierung einer *Factory*, die Referenz auf Konto-Objekt zu gegebener Kontonummer liefert
    - ggf. Erzeugung eines neuen Objekts (aus Datenbank)
- Anmerkung: RMI erlaubt keine entfernte Objekterzeugung
  - Client kann kein Objekt auf Serverseite erzeugen



3.3.5 *Client Callbacks*

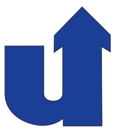
- Häufig: Server möchte Aufrufe im Client durchführen
  - z.B. Fortschrittsanzeige, Rückfragen, ...
- Dazu: Client-Objekt muß RMI-Objekt sein
  - Übergabe der Referenz `this` an Server-Methode
- In einigen Fällen ist Ableiten von `UnicastRemoteObject` nicht möglich, z.B. bei Applets
  - dann: Export des Objekts durch
    `UnicastRemoteObject.exportObject(obj,0);`
  - Achtung: beim Aufruf `exportObject(obj)` wird auch ab JDK 1.5 kein dynamischer Stub erzeugt
- Beispielcode: siehe WWW (*Hello-World* mit Callback)



3.3.6 RMI und Threads

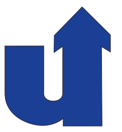
- RMI macht keine Angaben darüber, wieviele Threads serverseitig für Methodenaufrufe bereitgestellt werden
  - nur ein Thread, ein Thread pro Aufruf, ...
- D.h. prinzipiell können mehrere Server-Methoden gleichzeitig aktiv sein
  - korrekte Synchronisation erforderlich (`synchronized`)!
- Clientseitiges Sperren eines *Remote*-Objekts durch einen `synchronized`-Block ist nicht möglich
  - gesperrt wird nur lokaler Stub
  - Sperre muß ggf. über Methoden des *Remote*-Objekts realisiert werden

3.4 *Deployment*



- *Deployment:* Verteilung, Übertragung und Installation der Bestandteile einer verteilten Anwendung
  - konkret bei RMI: welche `class`-Datei muß wohin?

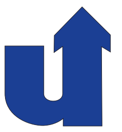
- **Server**, **RMI-*Registry*** und **Client** benötigen die `class`-Dateien für:
  - das *Remote*-Interface des Servers
  - alle Klassen bzw. Interfaces, die im Server-Interface (rekursiv) benutzt werden
  - bis JDK 1.4 auch die Stub-Klassen für alle benutzten *Remote*-Interfaces



- **Client** und **Server** benötigen zusätzlich die `class`-Dateien für:
  - ihre eigene Implementierung
  - alle Klassen von serialisierbaren Objekten, die sie empfangen
    - als Parameter oder Ergebnis von Methodenaufrufen
  - bis JDK 1.4 auch die Stub-Klassen für alle *Remote*-Objekte, die sie empfangen

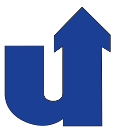
- Probleme bei statischer Installation der `class`-Dateien für Stubs und serialisierte Objekte:
  - Abhängigkeit zwischen Client und Server
    - Aufrufparameter, Ergebnisobjekte
  - Änderung der Klassen erfodert Neuinstallation
    - macht einen Vorteil verteilter Anwendungen zunichte

3.4.1 Entferntes Klassenladen bei RMI



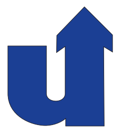
Klassenlader

- Klassenlader (*class loader*) dienen zum Nachladen von Klassen (und Schnittstellen) zur Laufzeit
  - genauer eigentlich: von `class`-Dateien
- Jede Klasse wird nur einmal geladen
- Klassenlader sind selbst Java-Objekte
  - Basisklasse: `java.lang.ClassLoader`
- RMI verwendet einen eigenen Klassenlader
  - `java.rmi.RMIClassLoader`



Entferntes Laden von Klassen

- `RMIClassLoader` erlaubt, Klassen auch von entfernten Rechnern zu laden
  - über HTTP (Web Server) oder FTP
  - URL wird beim Start der JVM über `codebase`-*Property* definiert
- Damit: zentrale Ablage der nötigen Dateien
  - „automatisches“ *Deployment*
- Einschränkungen:
  - alle Klassen, die im Client-Code namentlich genannt werden, müssen lokal vorhanden sein
  - Client muß eigenen *Security Manager* definieren



Beispiel

BankClient
main()

<<interface>>
BankServer
{Remote}
getKonto(...): Konto

<<interface>>
Konto
{Remote}
getBuchungen(...): Buchung

<<interface>>
Buchung
getDatum(): ...
getBetrag(): ...
getText(): ...

Serializable

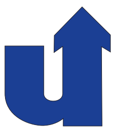
Unicast RemoteObject

BuchungImpl

KontoImpl

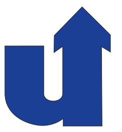
BankServerImpl

- Die Klassen `BankServer`, `Konto` und `Buchung` müssen lokal beim Client (`BankClient`) vorhanden sein
- `BuchungImpl` kann vom Client nachgeladen werden



Lokale und entfernt ladbare Klassen

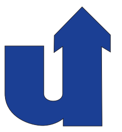
- Lokal ladbar (über `CLASSPATH`) müssen (für Client und Server) sein:
  - alle im Client-Code namentlich genannte Klassen,
    alle in diesen Klassen namentlich genannte Klassen, ...
  - d.h. alles, was auch zum Kompilieren gebraucht wird
- *Remote* ladbar sind:
  - Stub-Klassen von *Remote*-Objekten
  - Unterklassen, auf die nur mittels Polymorphie zugegriffen wird
    - d.h. Client-Code verwendet nur Oberklasse oder Schnittstelle
- Die *RMI-Registry* kann alle benötigten Klassen *remote* laden



Beispiel: *Hello-World* mit Callback und Ergebnisobjekt

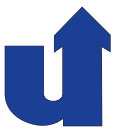
- Schnittstellen (siehe WWW):

```
public interface Hello extends Remote
{
  HelloObj getHello(AskUser ask) throws RemoteException;
}

public interface AskUser extends Remote
{
  boolean ask(String question) throws RemoteException;
}

public interface HelloObj
{
  void sayIt();
}
```



Beispiel: Wie werden die Klassen geladen?

- Schnittstellen `Hello.class`, `AskUser.class`, `HelloObj.class`
  - müssen beim Client lokal vorhanden sein
  - können von der *RMI-Registry* auch *remote* geladen werden
- Implementierung `HelloObjImpl.class` von `HelloObj`
  - kann vom Client *remote* geladen werden
  - wird von *RMI-Registry* nicht benötigt
- Stub-Klassen der beiden `Remote`-Schnittstellen
  - werden ab JDK 1.5 i.d.R. dynamisch erzeugt (nicht geladen)
  - können aber auch *remote* geladen werden



Beispiel: Notwendige Änderungen im Client

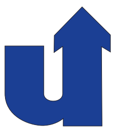
- Verwendung des RMI *Security Managers*:

```
public static void main(String args[]) {
  System.setSecurityManager(new RMISecurityManager());
```

- Definition der Sicherheitsrichtlinien: *Policy*-Datei:

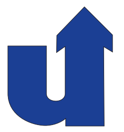
```
grant {
  permission java.net.SocketPermission "myserver:1024-",
             "connect,accept";
  permission java.net.SocketPermission "www.bsvs.de:80",
             "connect";
};
```

  - erlaubt lokalen Klassen (Client!)
    - Verbindung zu/von `myserver` auf nichtprivilegierten Ports:
      - RMI-*Registry* (1099), Server- und Callback-Objekt (dyn.)
    - Verbindung zum Web-Server

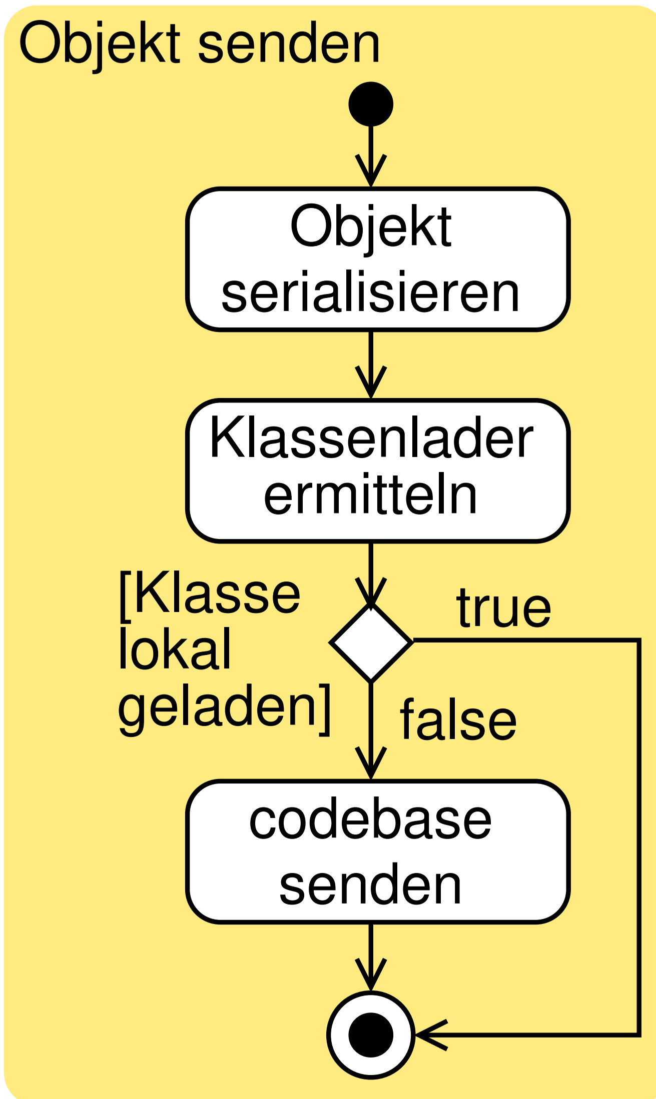


Beispiel: *Deployment*

- Alle *remote* zu ladenden Klassen werden in ein Archiv gepackt
- Archiv wird über einen Web-Server zugreifbar gemacht
- Start des Servers mit Angabe einer `codebase`, z.B.:
  - `java -Djava.rmi.server.codebase="http://www.bsvs.de/jars/HelloServer.jar" HelloServer`
  - die `codebase`-*Property* gibt der JVM die URL an, unter der die Klassen zu laden sind
  - Server gibt `codebase` beim Registrieren des Server-Objekts an die RMI-*Registry* weiter
  - RMI-*Registry* gibt `codebase` an Client weiter
- Start des Clients mit Angabe der *Policy*-Datei, z.B.:
  - `java -Djava.security.policy=policy HelloClient`



Ablauf bei der Übertragung von Objekten

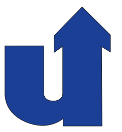


serialisiertes Objekt empfangen

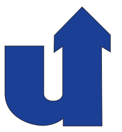
[Klasse im Speicher]
true
geladene Klasse verwenden
false
[Klasse lokal verfügbar]
true
Klasse lokal laden
false
[codebase vorhanden]
true
Klasse von codebase* laden
false
ClassNotFound Exception
Objekt deserialisieren

* ab JDK 1.7 explizit fest– zulegen

3.4.2 Java Sicherheitsmanager



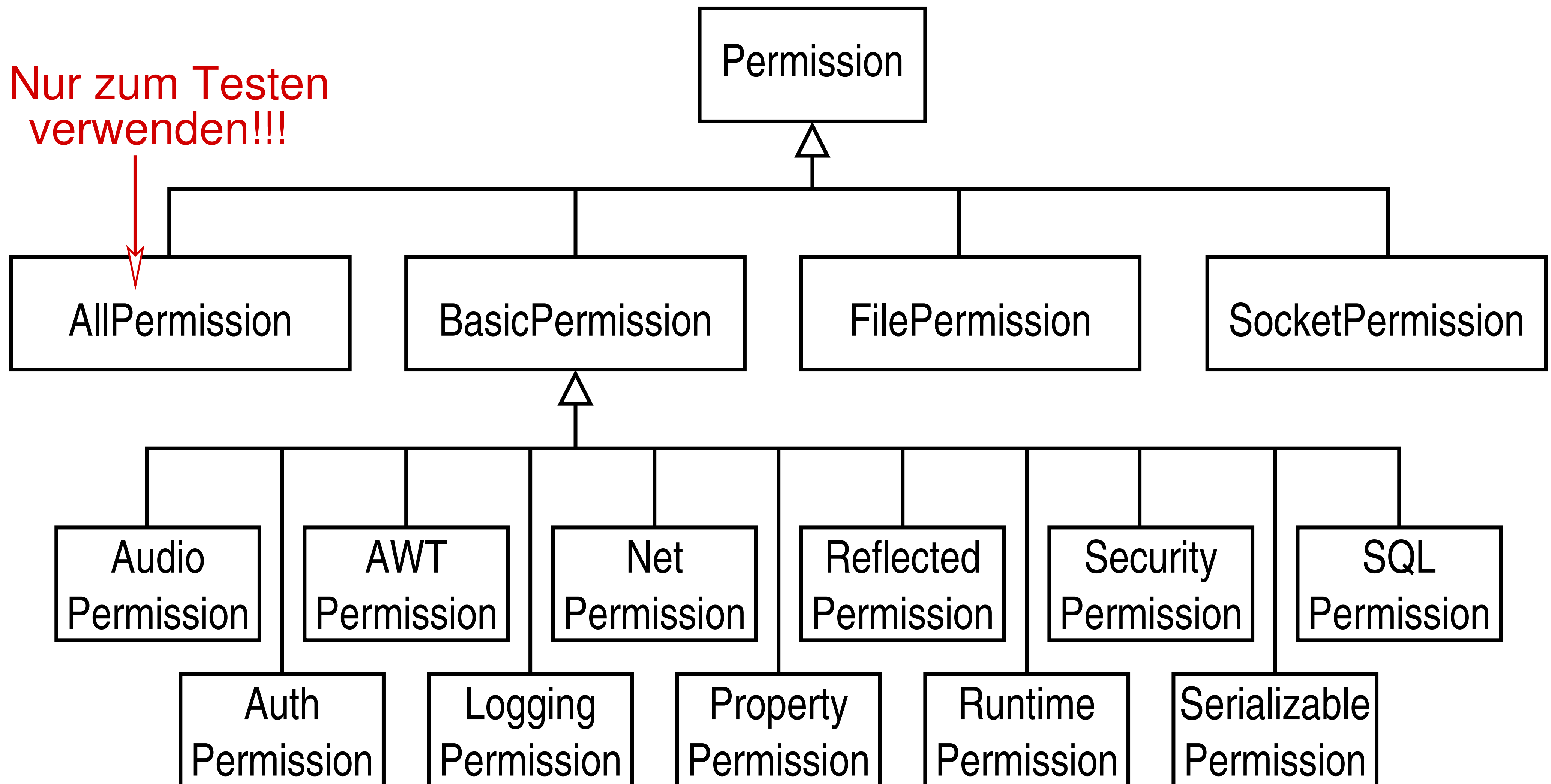
- JVM kann bei Bedarf mit Sicherheitsmanager ausgerüstet werden
  - bei Java-Anwendungen: `System.setSecurityManager()`
  - bei Java-Applets: standardmäßig
- Sicherheitsmanager prüft u.a. ob die Anwendung
  - auf lokale Datei zugreifen darf,
  - Netzverbindung aufbauen darf,
  - JVM anhalten darf,
  - Klassenlader erzeugen darf,
  - AWT-Ereignisse lesen darf, ...
- Vorgaben durch Sicherheitsrichtlinie (*Security Policy*)
  - bei Verletzung der Vorgaben: Exception

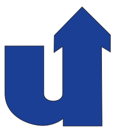


Sicherheitsrichtlinien

- Weisen Berechtigungen an Codes bestimmter Quellen zu
- Codequelle kann durch zwei Eigenschaften beschrieben werden:
  - Codestandort: URL, von wo der Code geladen wurde
  - Zertifikate (bei signiertem Code)
- Berechtigungen erlauben Zugriffe auf bestimmte Ressourcen
  - Berechtigungen werden durch Objekte modelliert, sind meist aber in Berechtigungsdatei spezifiziert
  - z.B.
    ```
    FilePermission p =
            new FilePermission("/tmp/*", "read,write");
    ```
  - bzw.
    ```
    permission java.io.FilePermission "/tmp/*",
            "read,write";
    ```

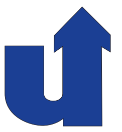
Hierarchie der Berechtigungsklassen in JDK 1.2





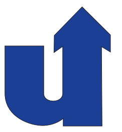
Sicherheitsrichtlinien-Datei (*Policy File*)

```
grant {
  permission java.net.SocketPermission "www.bsvs.de:80",
    "connect";
};

grant codebase "file:" {
  permission java.io.FilePermission "/home/tom/-",
    "read, write";
  permission java.io.FilePermission "/bin/*", "execute";
};

grant codebase "http://www.bsvs.de/jars/HelloServer.jar" {
  permission java.net.SocketPermission "localhost:1024-",
    "listen, accept, connect";
};
```



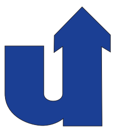
Sicherheitsrichtlinien-Datei (*Policy File*) ...

- Alle Klassen dürfen:
  - Verbindung zu `www.bsvs.de`, Port 80 aufbauen
- Lokal geladene Klassen dürfen:
  - Dateien in `/home/tom` oder (rekursiv) einem Unterverzeichnis davon lesen und schreiben
  - Dateien im Verzeichnis `/bin` ausführen
- Klassen, die aus `http://www.bsvs.de/jars/HelloServer.jar` geladen wurden, dürfen:
  - Netzverbindungen auf dem / zum lokalen Rechner auf nichtprivilegierten Ports (ab 1024) annehmen / aufbauen



Weitere Dokumentation

- Generelles zu Sicherheitsrichtlinien:
  `http://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html`
- Überblick über die Berechtigungsklassen:
  `http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html`
- Java API Dokumentation:
  `http://docs.oracle.com/javase/8/docs/api/`



3.5 Zusammenfassung

- RMI erlaubt Zugriff auf entfernte Objekte
  - transparent, über Proxy-Objekte
  - Klassen der Proxy-Objekte
    - werden i.d.R. dynamisch erzeugt
- Parameterübergabemechanismen
  - per Wert, falls Parameter-Objekt serialisierbar
  - per Referenz, falls Parameter-Objekt RMI-Objekt
- Klassen können auch *remote* geladen werden (*Security Manager*!)
- Namensdienst: RMI-*Registry*
- Sicherheit: RMI über SSL möglich, aber nicht ideal

Ankündigung

Jahresgespräch Informatik

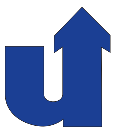
wann: Mi. 23.05.2018, 14:00 (s.t.)

wo: H-F 114

wer: Professoren und Studierende der Informatik (Bachelor, Master)

warum: Wir brauchen Ihr Feedback!

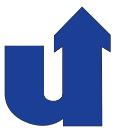
- Diskussion von Stärken und Schwächen des Studiengangs
- Diskussion zur Weiterentwicklung des Studiengangs



Verteilte Systeme

SoSe 2018

4 Namensdienste

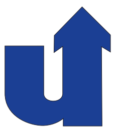


Inhalt

- Grundlagen
- Beispiel: JNDI

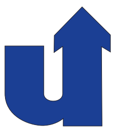
Literatur

- Tanenbaum, van Steen: Kap. 4.1
- Farley, Crawford, Flanagan: Kap. 7
- http://docs.oracle.com/javase/tutorial/jndi/overview



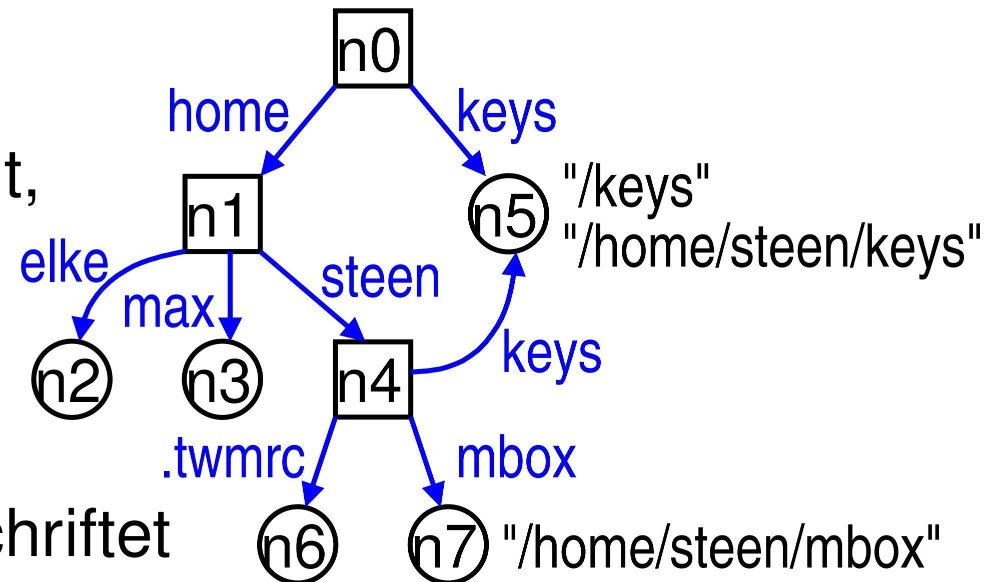
Namen, Adressen und IDs

- **Name**: Zeichen- oder Bitfolge, die auf eine Einheit verweist
  - Einheit: z.B. Rechner, Drucker, Datei, Benutzer, Webseite, ...
- **Adresse**: Name des Zugangspunkts einer Einheit
  - Zugangspunkt ermöglicht Zugriff auf die Einheit
  - mehrere Zugangspunkte pro Einheit sind möglich
  - Zugangspunkt kann sich im Lauf der Zeit ändern
- Ein **positionsunabhängiger Name** bezeichnet eine Einheit unabhängig von ihrem Zugangspunkt
- **ID**: Name mit folgenden Eigenschaften:
  - ID verweist auf max. eine Einheit, Einheit hat max. eine ID
  - ID verweist immer auf dieselbe Einheit (nicht wiederverwendet)

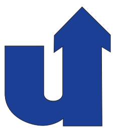


Namensräume

- Dargestellt durch gerichteten, beschrifteten Graph
  - Blattknoten: benannte Einheit, ggf. mit Information / Status
  - innerer Knoten: Verzeichnis-knoten
  - Kanten sind mit Namen beschriftet

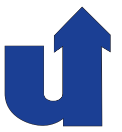


- Einheiten werden durch Pfade im Graphen benannt:
  Startknoten: < Beschriftung-1, Beschriftung-2, ...>
  - absoluter Pfad: von Wurzel (des Namensraums) ausgehend
  - relativer Pfad: von beliebigem Knoten ausgehend
- Beispiel: Namen im UNIX-Dateisystem



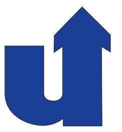
Linken und Verknüpfen

- **Alias**: alternativer Name für dieselbe Einheit
- Möglichkeiten zur Realisierung von Aliasen:
  - erlaube mehrere absolute Pfadnamen für eine Einheit
    - z.B. *Hard Link* in Unix
  - (spezieller) Blattknoten speichert Pfadnamen der Einheit
    - z.B. *Symbolic Link* in Unix
- Transparente Verknüpfung verschiedener Namensräume:
  - (spezieller) Verzeichnisknoten speichert ID eines Verzeichnisknotens in einem anderen Namensraum
    - z.B. „gemountetes“ Dateisystem in Unix



Namensauflösung

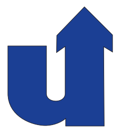
- Finden des Knotens (bzw. der Information), die einem Namen zugeordnet ist
  - beginne beim Startknoten
  - schlage in Verzeichnistabelle erste Beschriftung nach
    $\Rightarrow$ ID des nächsten Knotens
  - usw., bis Pfad vollständig bearbeitet
- **Schlußmechanismus**: Bestimmung des Startknotens
  - meist implizit
- **globale Namen**: Auflösung unabhängig von speziellem Kontext
- **lokale Namen**: Auflösung ist kontextabhängig
  - z.B. Pfadname relativ zum Arbeits-Verzeichnis in Unix



Implementierung von Namensdiensten

- Typische Operationen:
  - bind(Name, Adresse, Attribute)
  - lookup(Name, Attribute) $\Rightarrow$ Adresse, Attribute
  - unbind(Name, Adresse)
- In verteilten Systemen:
  - Namensraum wird verteilt gespeichert (i.a. hierarchisch)
  - für hohe Verfügbarkeit: zusätzlich replizierte Speicherung
- Namensauflösung kann iterativ oder rekursiv erfolgen
  - iterativ: Server antwortet mit Adresse des nächsten Servers
  - rekursiv: Server fragt selbst bei nächstem Server an
- Beispiel: *Domain Name System* (☞ **RN_I, 11.1**)

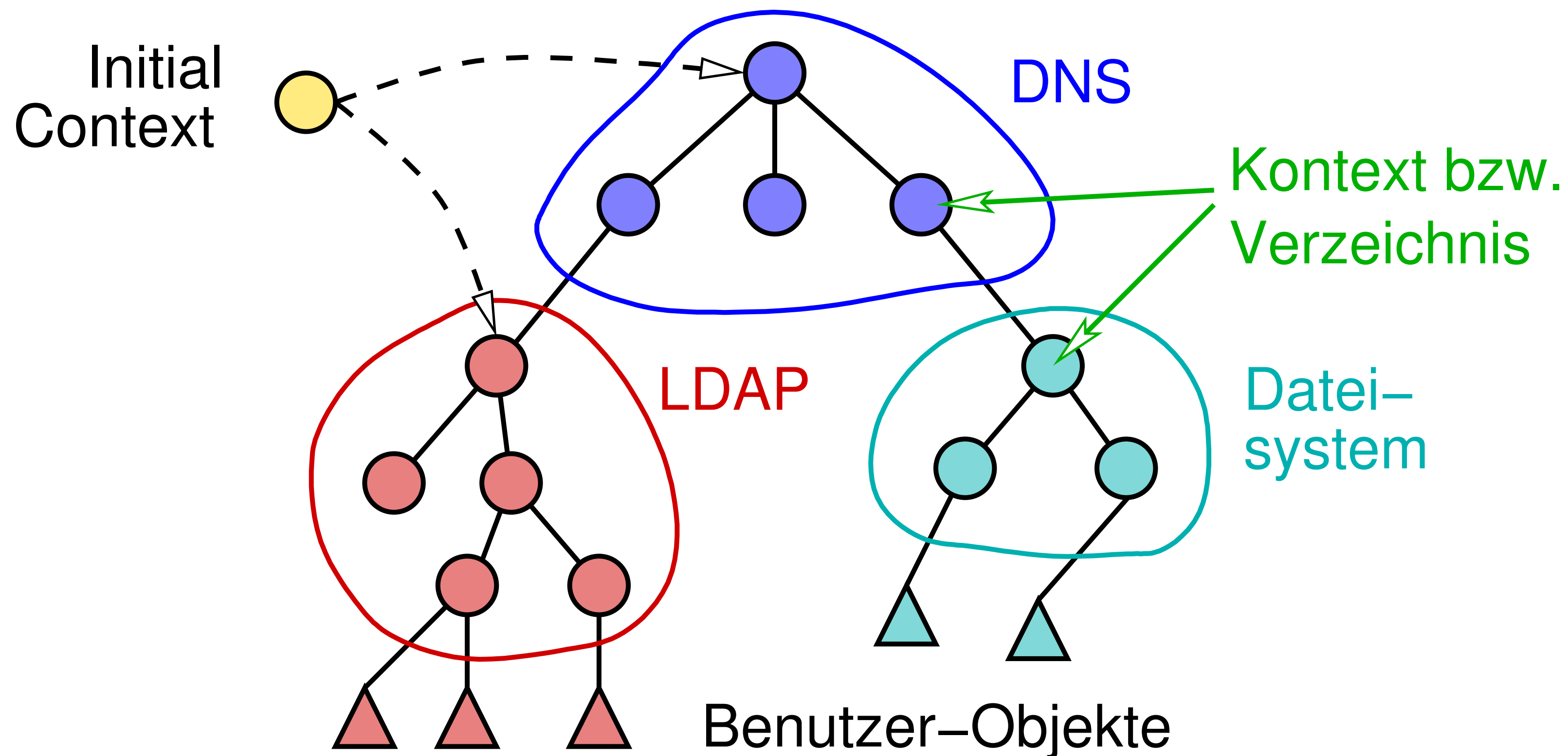
4.2 Beispiel: JNDI



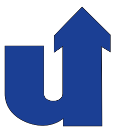
- JNDI: *Java Naming and Directory Interface*
- API zum Zugriff auf verschiedene Namens- und Verzeichnis-Dienste
  - Verzeichnisdienst speichert auch Attribute zu Objekten

4.2 Beispiel: JNDI ...

- JDNI unterstützt zusammengesetzte Namensräume
  - von verschiedenen Namens-/Verzichnisdiensten verwaltet

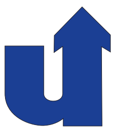


- Verzeichnisse werden als „Kontext“ bezeichnet
  - Objekte werden innerhalb eines Kontexts an Namen gebunden



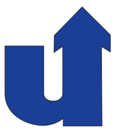
Die Schnittstelle `javax.naming.Context` für Namens-Kontexte

- Wichtige Methoden:
  - `bind()`, `rebind()` : Binden von Objekten an Namen
    - `bind()` wirft Exception, falls Name schon existiert
  - `unbind()` : Namen entfernen
  - `rename()` : Umbenennen
  - `lookup()` : Namen in Objekt auflösen
  - `listBindings()` : Liste aller Bindungen
  - `createSubcontext()` : Unter-Kontext erzeugen
  - `destroySubcontext()` : Unter-Kontext löschen



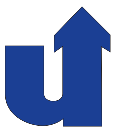
Die Schnittstelle `javax.naming.Context` für Namens-Kontexte ...

- Implementierungsklasse `InitialContext`
  - für initialen Kontext (abhängig vom konkreten Namensdienst)
    - `Context iC = new InitialContext(properties);`
  - Konfiguration über `Properties`-Objekt (`Hashtable`), u.a.:
    - `"java.naming.factory.initial"`
      - *Factory* für `InitialContext`
    - `"java.naming.provider.url"`
      - Kontaktinformation für *Service Provider*
    - `"java.naming.security.principal"` bzw. `"java.naming.security.credentials"`
      - Benutzername bzw. Paßwort für Authentifizierung



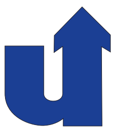
Beispiel: Zugriff auf RMI-*Registry*

```
import javax.naming.*;
...

    Properties props = new Properties();
    props.put("java.naming.factory.initial",
      "com.sun.jndi.rmi.registry.RegistryContextFactory");
    props.put("java.naming.provider.url",
      "rmi://localhost:1099");
    Context ctx = new InitialContext(props);

    obj = (Hello)ctx.lookup("Hello-Server");

    message = obj.sayHello();
```



Beispiel: Zugriff auf lokales Dateisystem

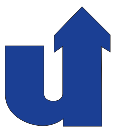
```
import javax.naming.*;
...

    Properties props = new Properties();
    props.put("java.naming.factory.initial",
        "com.sun.jndi.fscontext.RefFSContextFactory");
    Context ctx = new InitialContext(props);

    for (int i=0; i<args.length-1; i++)
      ctx = (Context)ctx.lookup(args[i]);
    NamingEnumeration<Binding> list
      = ctx.listBindings(args[args.length-1]);
    while (list.hasMore()) {
      Binding b = list.next();
      System.out.println(b.getName()+": "+b.getClassName());
    }
```



Verteilte Systeme

SoSe 2018

5 Prozeß-Management

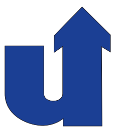


Inhalt

- Verteiltes Prozeß-Scheduling
- Codemigration

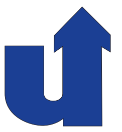
Literatur

- Tanenbaum, van Steen: Kap. 3
- Stallings: Kap 14.1



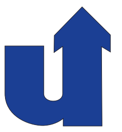
5.1 Verteiltes Prozeß-Scheduling

- Typisch: Middleware-Komponente, die
  - entscheidet auf welchem Knoten ein Prozeß ausgeführt wird
  - und Prozesse ggf. zwischen Knoten migriert
- Ziele:
  - Lastausgleich zwischen Knoten
  - Maximierung der Systemleistung (mittlere Antwortzeit)
    - auch: Minimierung der Kommunikation zwischen Knoten
  - Erfüllung spezieller Anforderungen an Hardware / Ressourcen
- Last: typisch Länge der Prozeß-Warteschlange (*ready queue*)
  - ggf. zusätzlich Berücksichtigung von Ressourcenverbrauch und Kommunikationsvolumen



Ansätze zum verteilten Scheduling

- Statisches Scheduling
  - Abbildung von Prozessen auf Knoten wird vor der Ausführung festgelegt
  - NP-vollständig, daher heuristische Verfahren
- Dynamischer Lastausgleich, zwei Varianten:
  - Ausführungsort eines Prozesses wird bei der Erzeugung festgelegt und später nicht mehr geändert
  - Ausführungsort eines Prozesses kann zur Laufzeit (ggf. mehrfach) geändert werden
    - präemptiv dynamischer Lastausgleich, **Prozeßmigration**



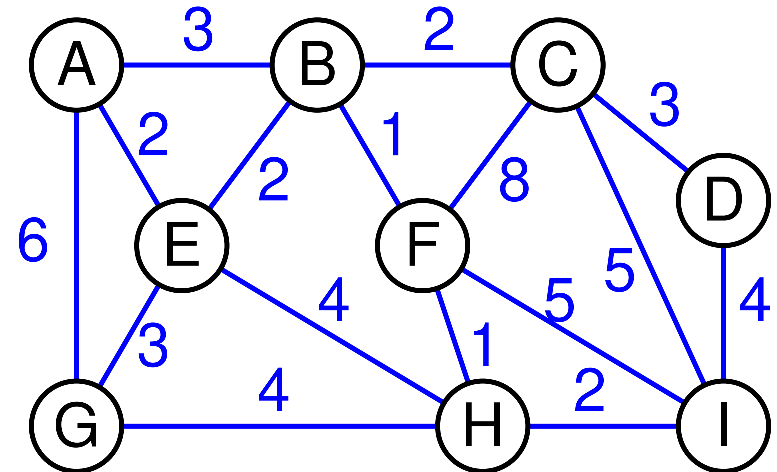
5.1.1 Statisches Scheduling

- Verfahren abhängig von Struktur / Modellierung eines Jobs
  - Jobs bestehen aus mehreren Prozessen
  - Unterschiedliche Kommunikationsstruktur
- Beispiele:
  - kommunizierende Prozesse: Graphpartitionierung
  - nicht-kommunizierende Tasks mit Abhängigkeiten: List-Scheduling

Scheduling durch Graphpartitionierung

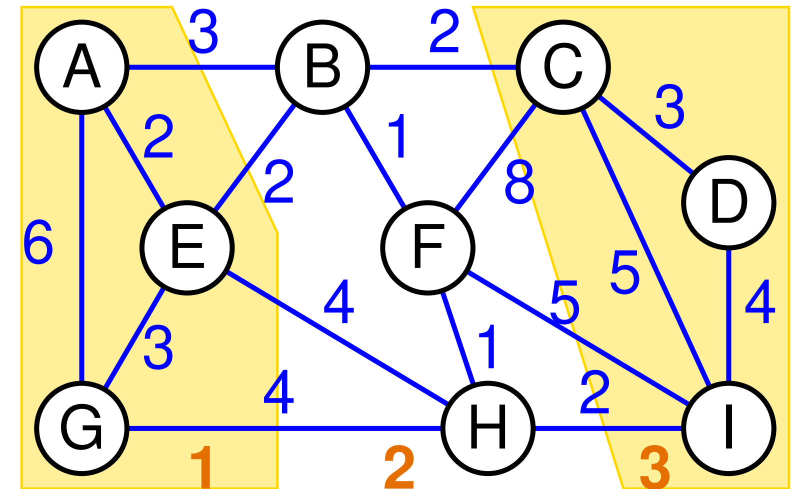
- Gegeben: Prozeßsystem mit
  - CPU- / Speicheranforderungen
  - Angabe der Kommunikationslast zwischen je 2 Prozessen

  i.a. dargestellt als Graph



- Gesucht: Aufteilung (Partitionierung) des Graphen so, daß
  - CPU- und Speicheranforderungen für jeden Knoten erfüllt
  - Partitionen in etwa gleich groß (Lastausgleich)
  - Gewichte-Summe der geschnittenen Kanten minimal
    - d.h. möglichst wenig Kommunikation zwischen Knoten
- NP-vollständig, daher viele heuristische Verfahren

Scheduling durch Graphpartitionierung



- Gegeben: Prozeßsystem mit
  - CPU- / Speicheranforderungen
  - Angabe der Kommunikationslast zwischen je 2 Prozessen

  i.a. dargestellt als Graph

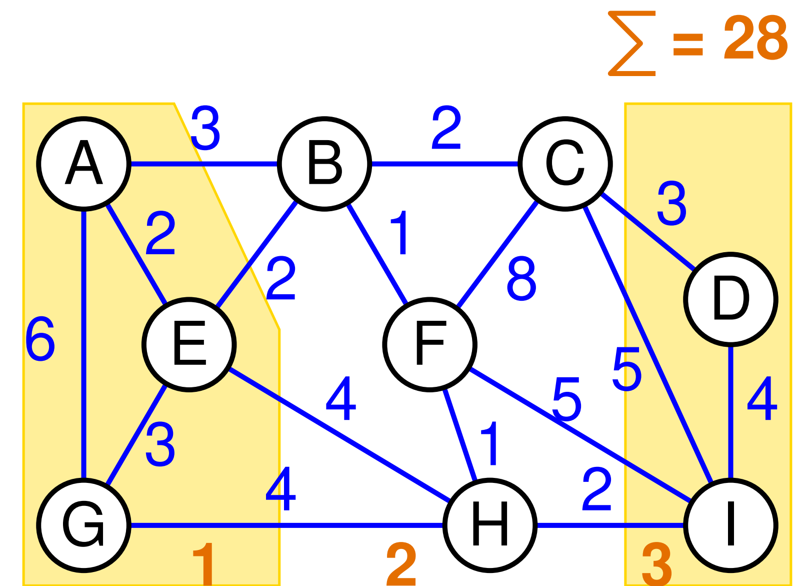
- Gesucht: Aufteilung (Partitionierung) des Graphen so, daß
  - CPU- und Speicheranforderungen für jeden Knoten erfüllt
  - Partitionen in etwa gleich groß (Lastausgleich)
  - Gewichte-Summe der geschnittenen Kanten minimal
    - d.h. möglichst wenig Kommunikation zwischen Knoten

- NP-vollständig, daher viele heuristische Verfahren

Scheduling durch Graphpartitionierung

- Gegeben: Prozeßsystem mit
  - CPU- / Speicheranforderungen
  - Angabe der Kommunikationslast zwischen je 2 Prozessen

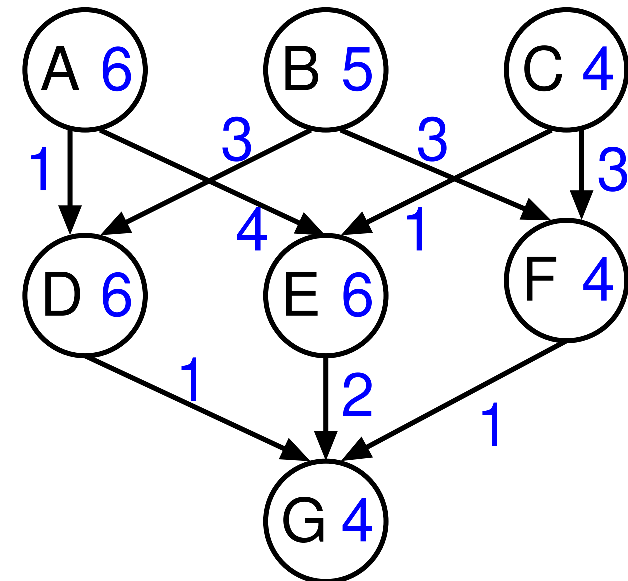
  i.a. dargestellt als Graph

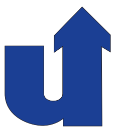


- Gesucht: Aufteilung (Partitionierung) des Graphen so, daß
  - CPU- und Speicheranforderungen für jeden Knoten erfüllt
  - Partitionen in etwa gleich groß (Lastausgleich)
  - Gewichte-Summe der geschnittenen Kanten minimal
    - d.h. möglichst wenig Kommunikation zwischen Knoten
- NP-vollständig, daher viele heuristische Verfahren

List-Scheduling

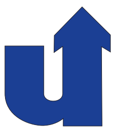
- Tasks mit Abhängigkeiten, aber ohne Kommunikation während der Ausführung
  - Tasks arbeiten auf Ergebnissen anderer Tasks
- Modellierung
  - Programm als DAG dargestellt
  - Knoten: Tasks mit Ausführungs- zeiten
  - Kanten: Kommunikation mit Über- tragungsdauer



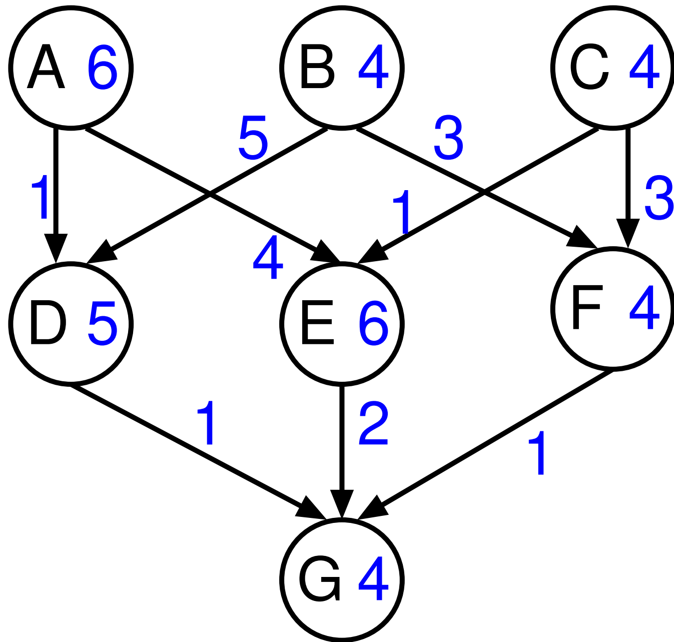


Vorgehensweise

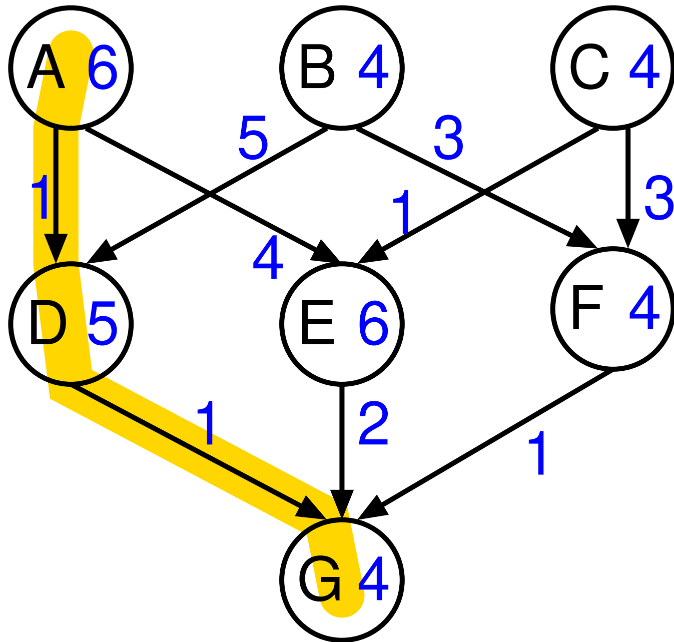
- Erstelle priorisierte Liste mit allen Tasks
  - viele unterschiedliche Heuristiken zur Bestimmung der Prioritäten, z.B. nach:
    - Länge des längsten Pfades (ohne Kommunikation) vom Knoten bis zum Ende des DAGs (*High Level First with Estimated Time*, HLFET)
    - frühest möglicher Startzeitpunkt (*Earliest Task First*, ETF)
- Arbeite die Liste wie folgt ab:
  - weise ersten Task an den Knoten zu, der frühesten Startzeitpunkt erlaubt
  - entferne Task aus der Liste
- Erstellung und Abarbeitung der Liste ggf. auch verzahnt



Beispiel: List-Scheduling mit HLFET

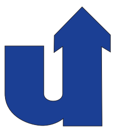


Beispiel: List-Scheduling mit HLFET

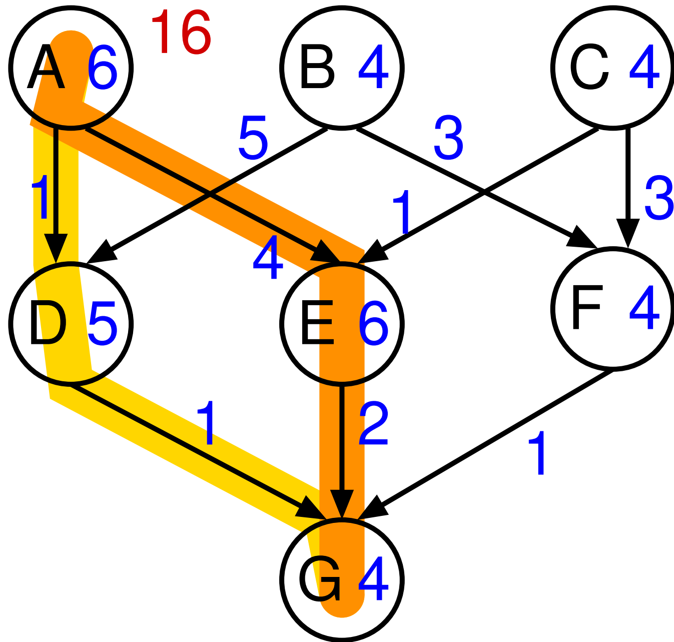


Static Level (ohne Komm.):

6+5+4 = 15

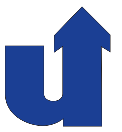


Beispiel: List-Scheduling mit HLFET

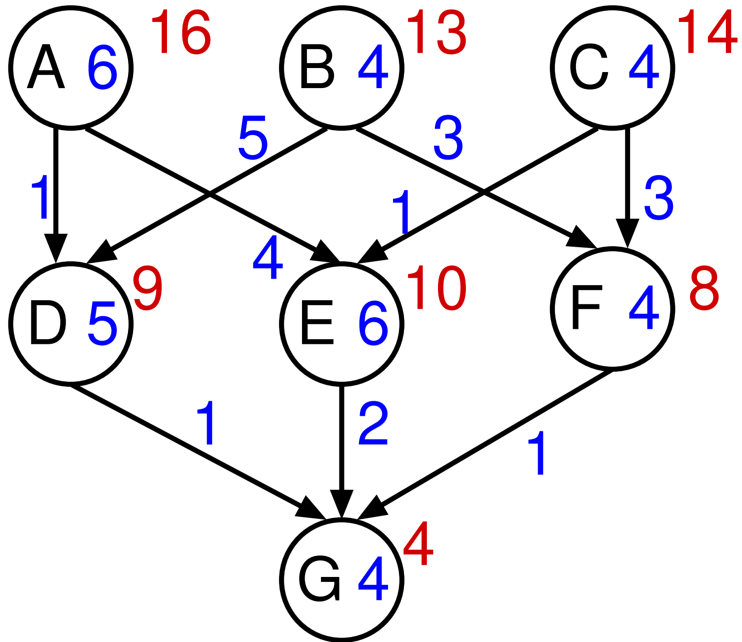


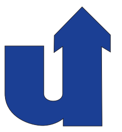
Static Level (ohne Komm.):

6+5+4 = 15   6+6+4 = 16

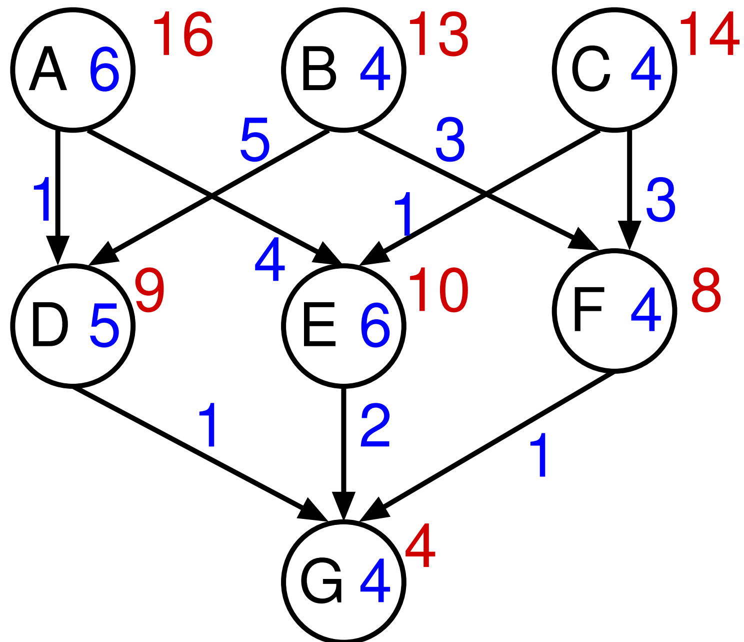


Beispiel: List-Scheduling mit HLFET

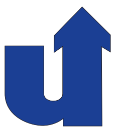




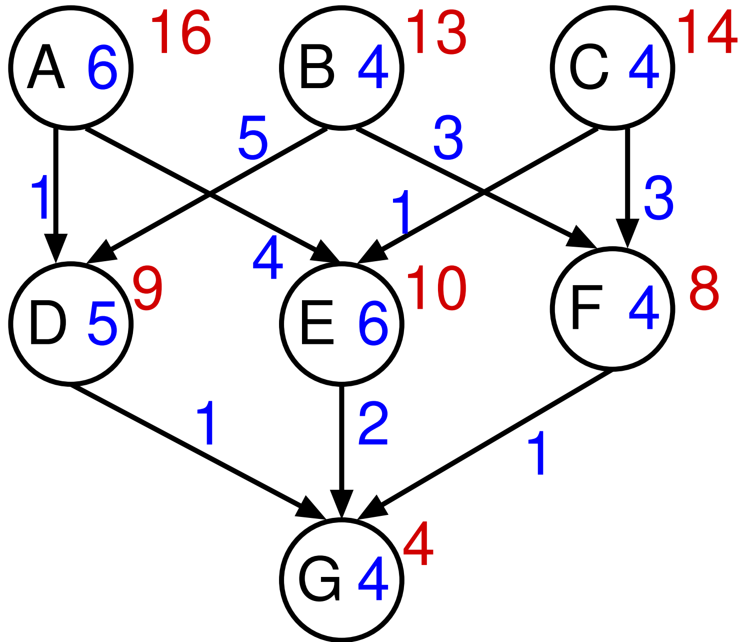
Beispiel: List-Scheduling mit HLFET



Liste:	A	C	B	E	D	F	G



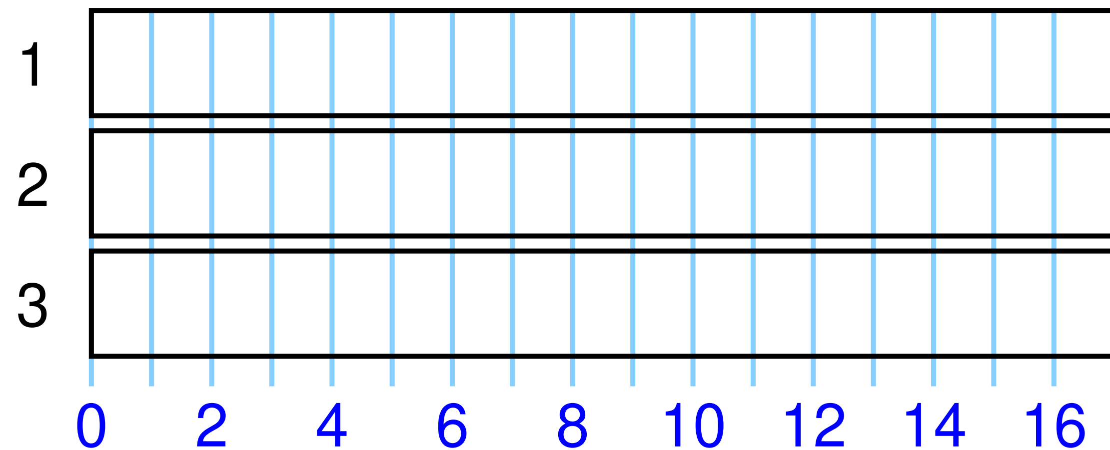
Beispiel: List-Scheduling mit HLFET

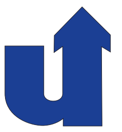


Liste:

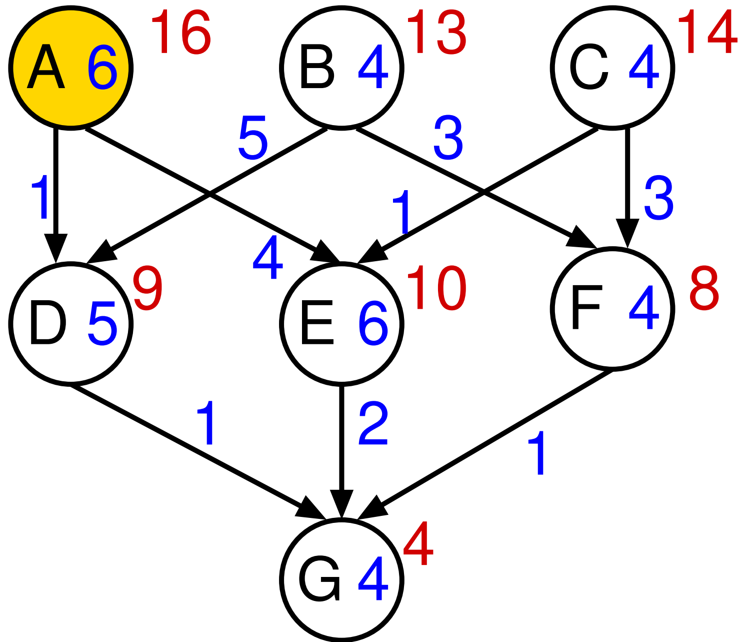
A	C	B	E	D	F	G

Schedule mit 3 Knoten:

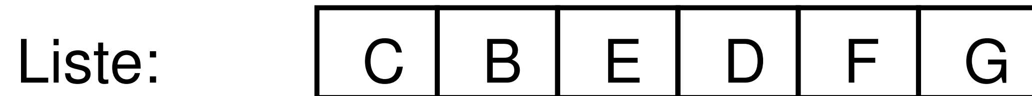




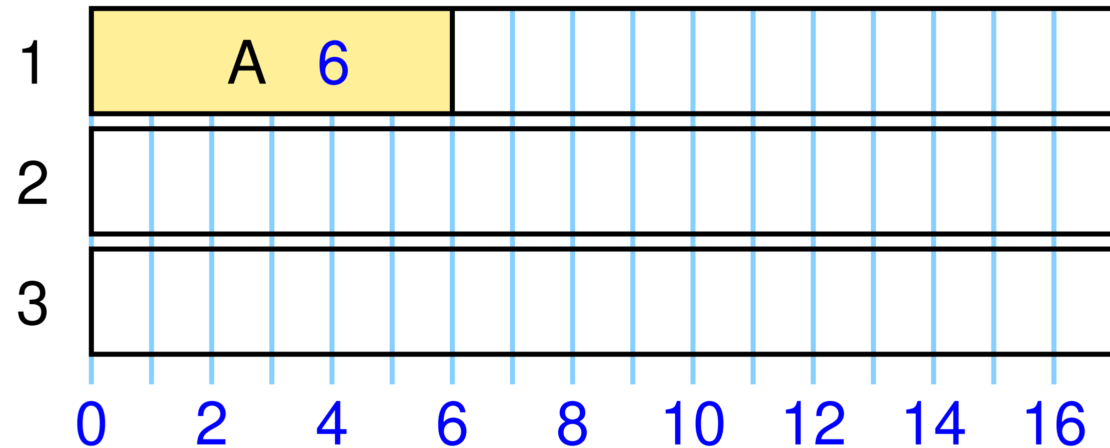
Beispiel: List-Scheduling mit HLFET



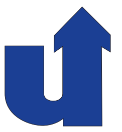
Liste:



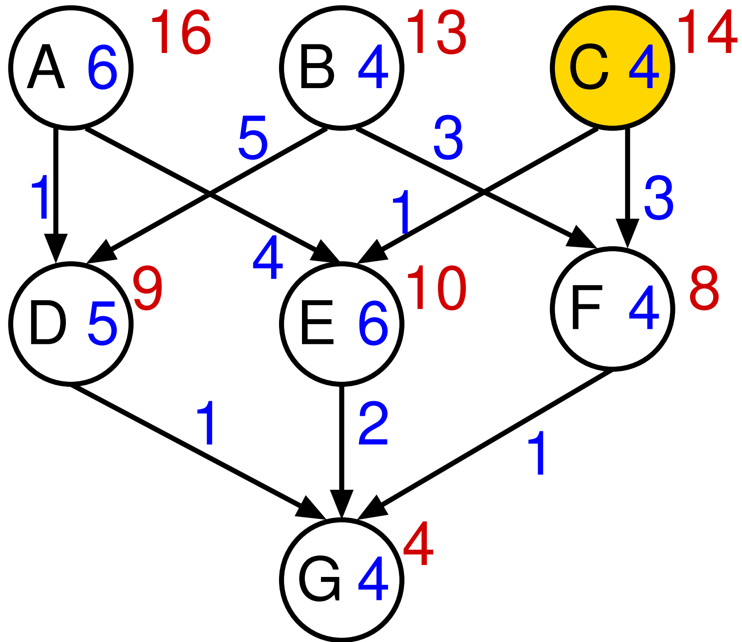
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit



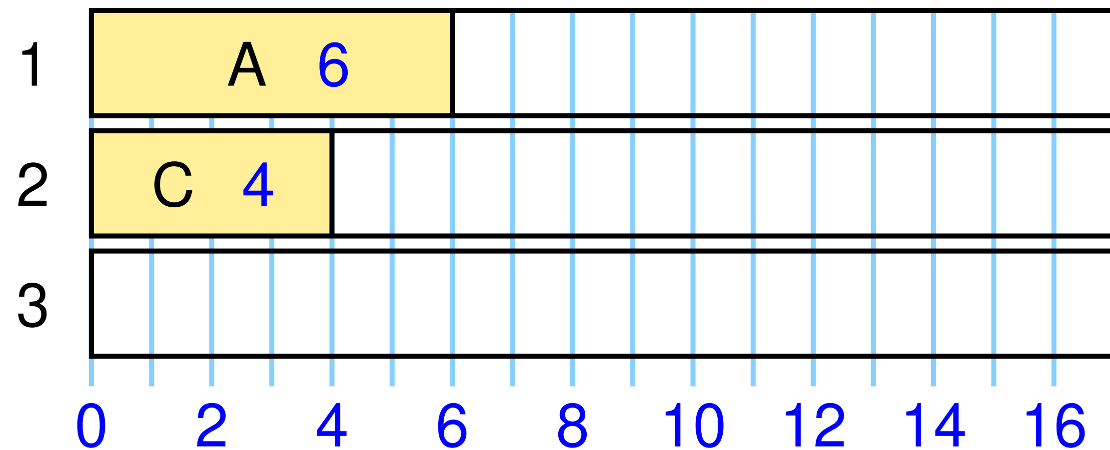
Beispiel: List-Scheduling mit HLFET



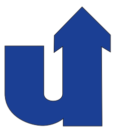
Liste:

B	E	D	F	G

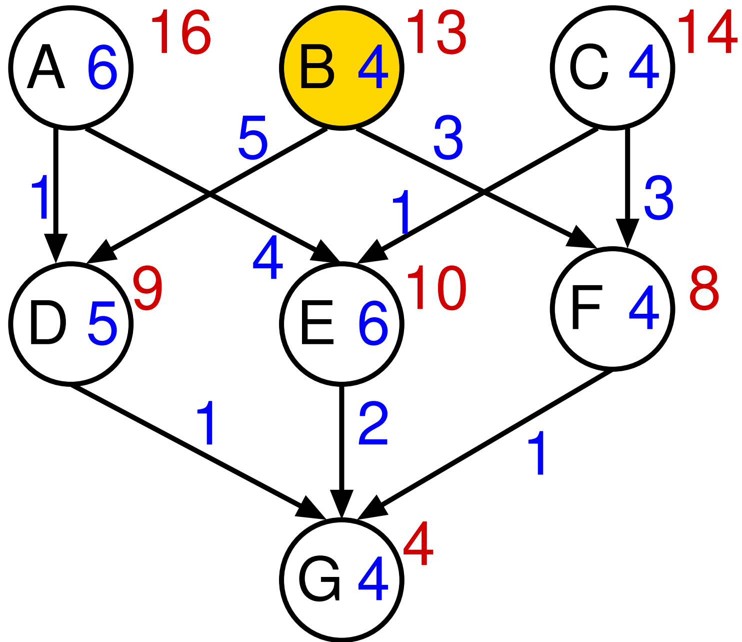
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit

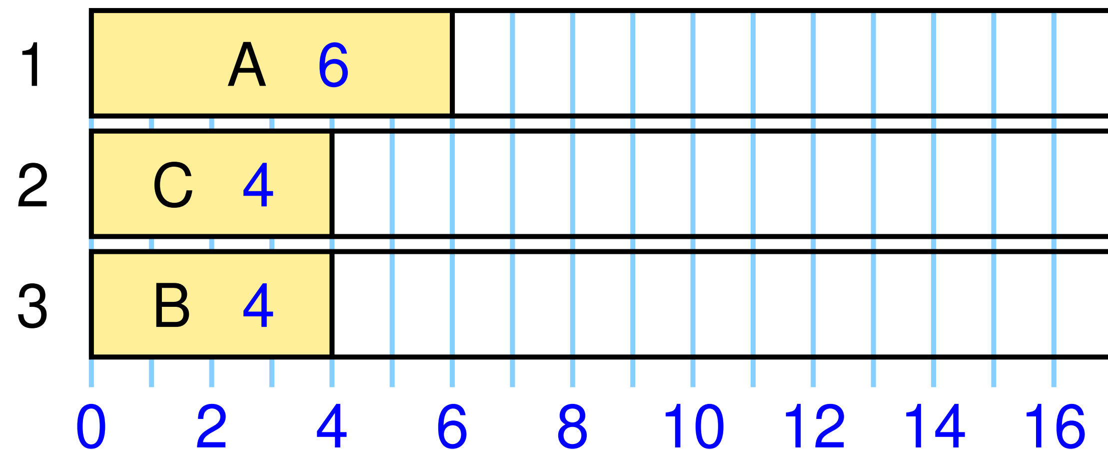


Beispiel: List-Scheduling mit HLFET



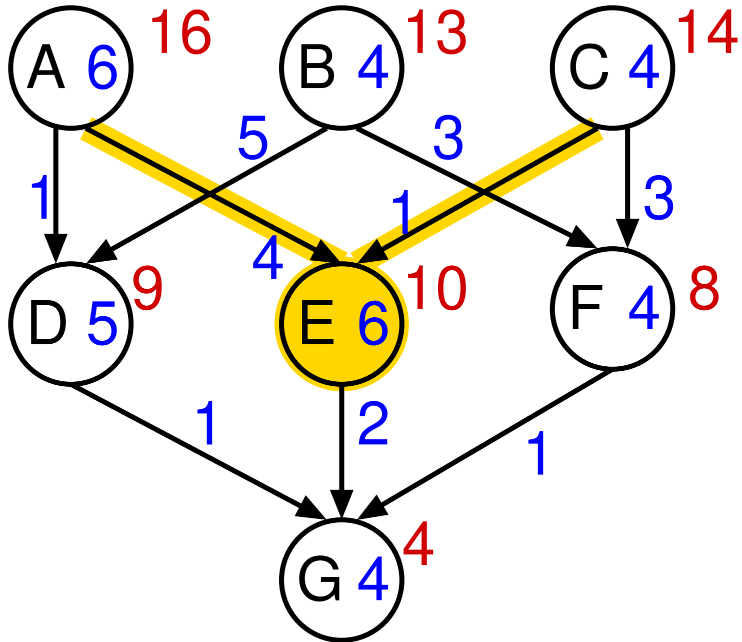
Liste:

Schedule mit 3 Knoten:



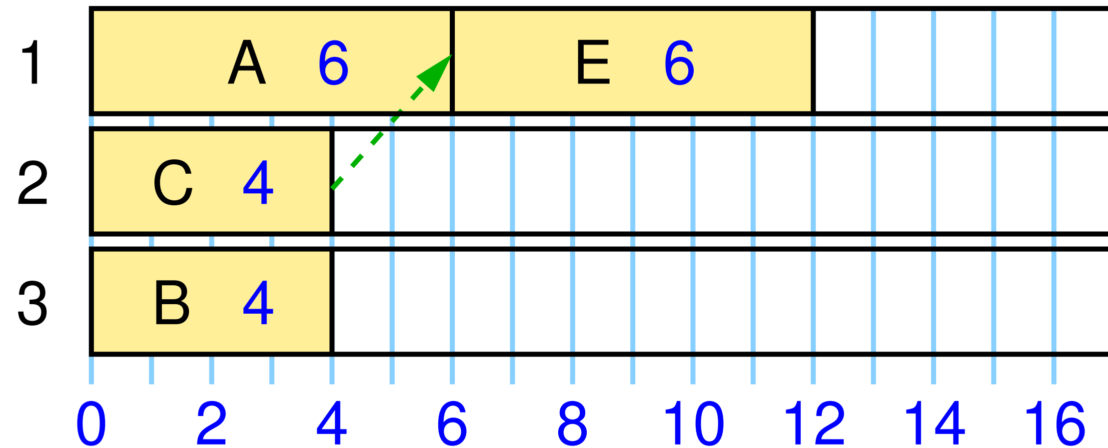
- Annahme: lokale Kommunikation kostet keine Zeit

Beispiel: List-Scheduling mit HLFET



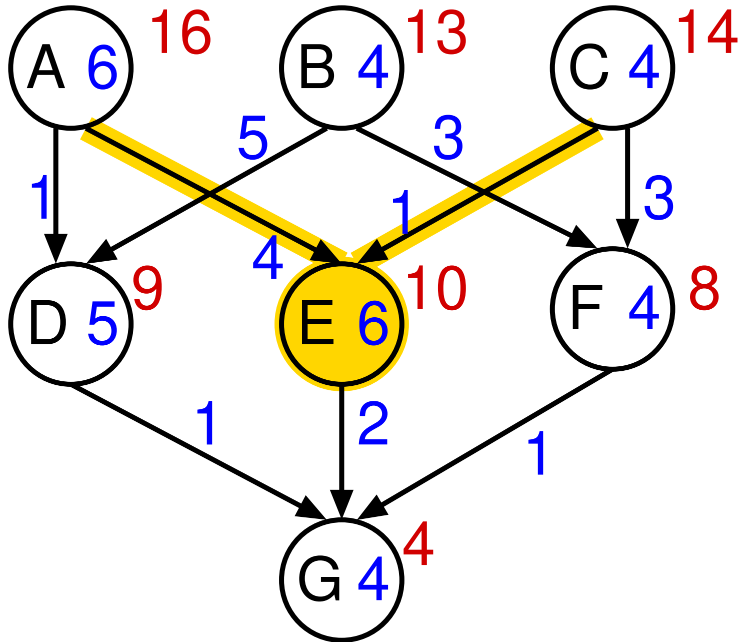
Liste:

Schedule mit 3 Knoten:



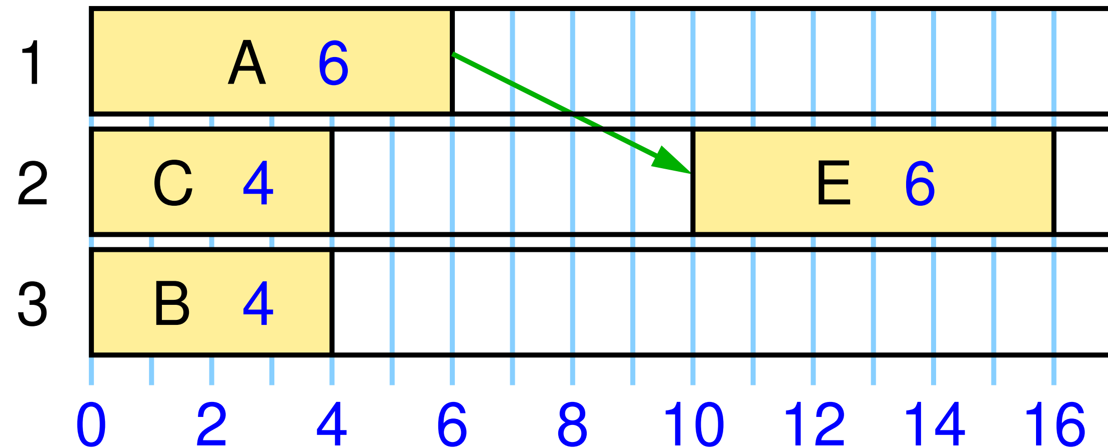
- Annahme: lokale Kommunikation kostet keine Zeit

Beispiel: List-Scheduling mit HLFET

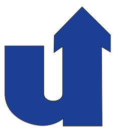


Liste:

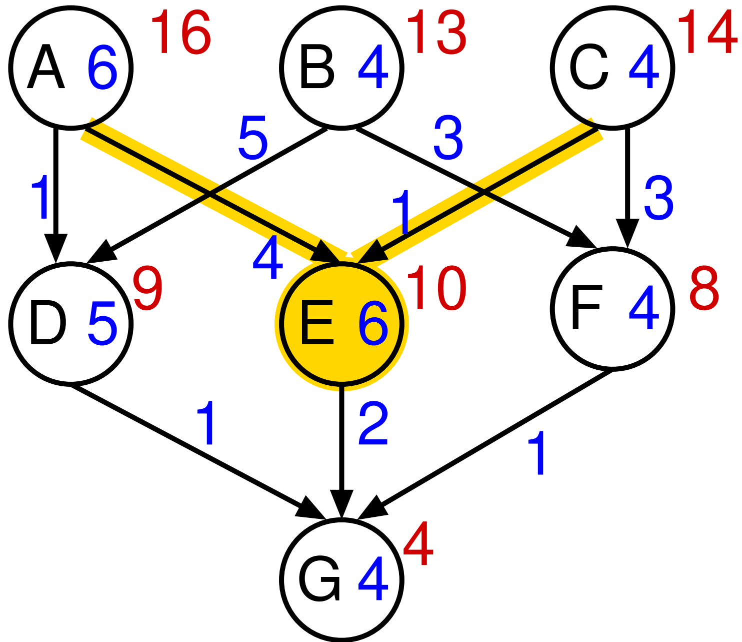
Schedule mit 3 Knoten:



- Annahme: lokale Kommunikation kostet keine Zeit

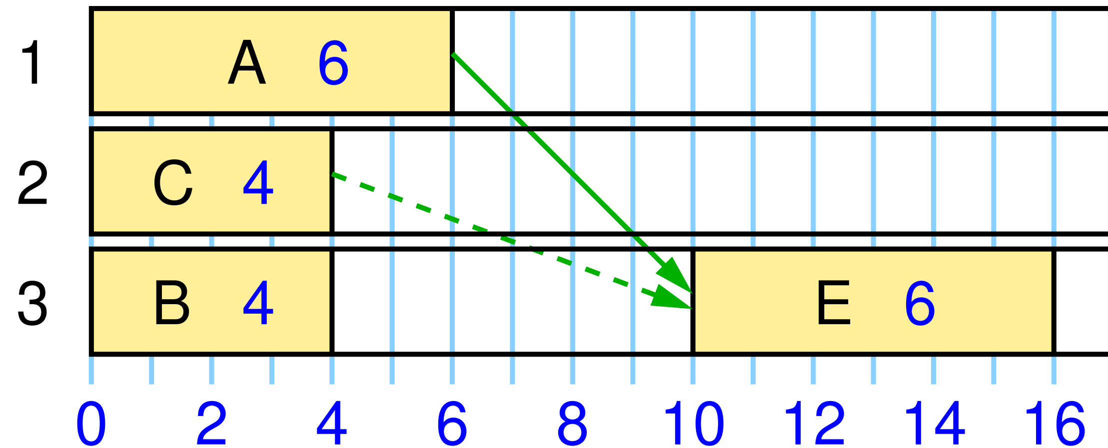


Beispiel: List-Scheduling mit HLFET

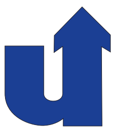


Liste:

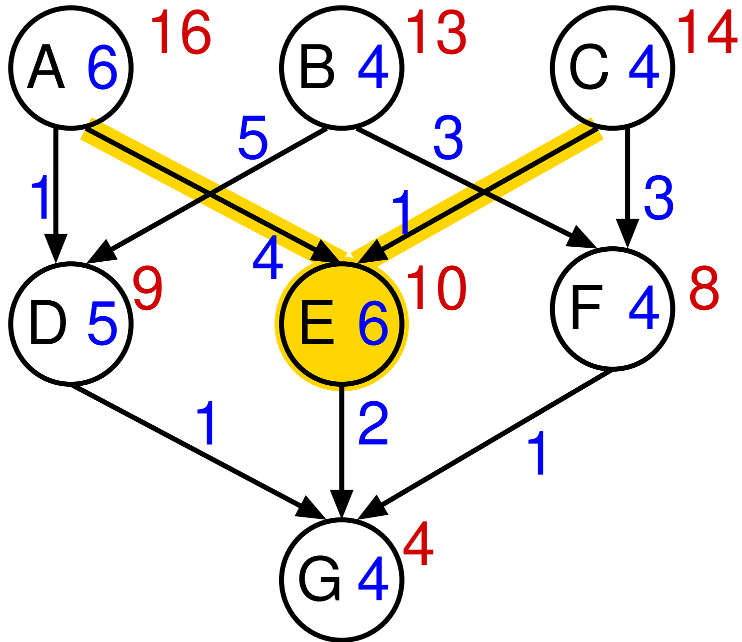
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit

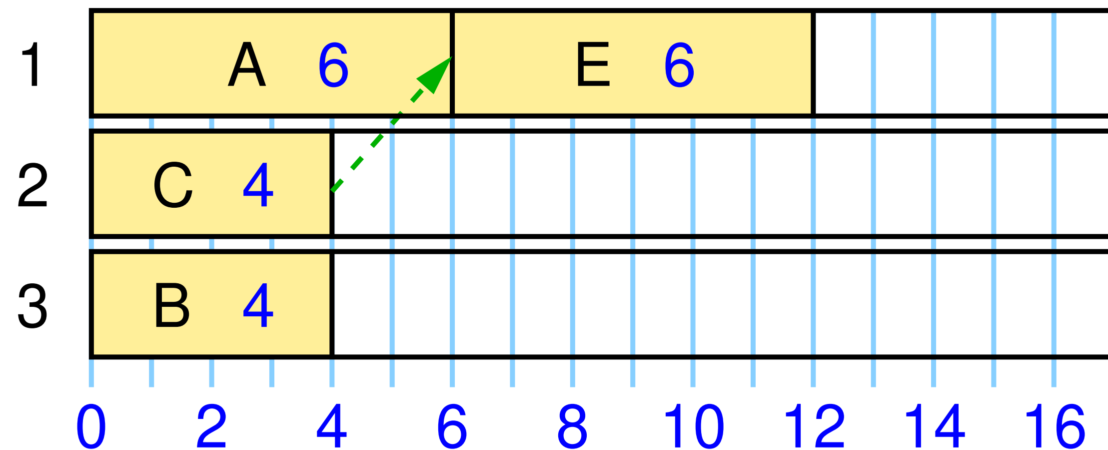


Beispiel: List-Scheduling mit HLFET

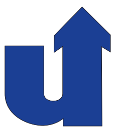


Liste:

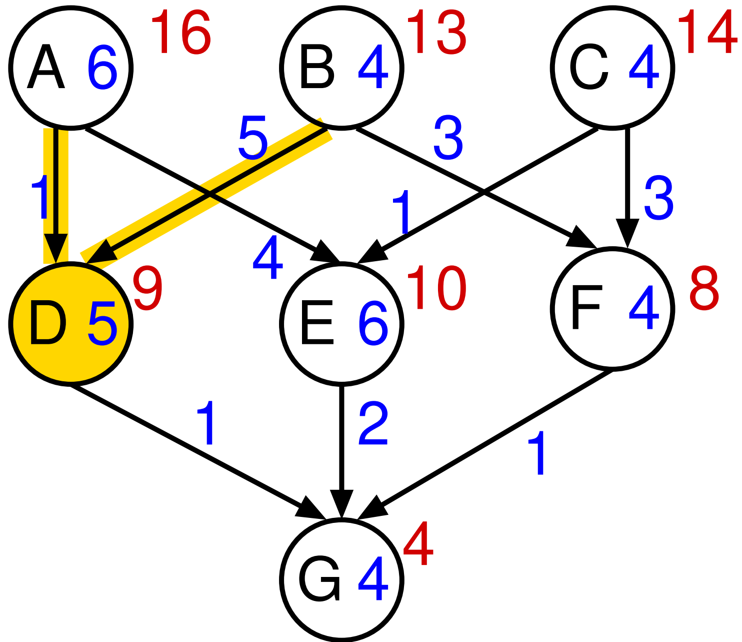
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit

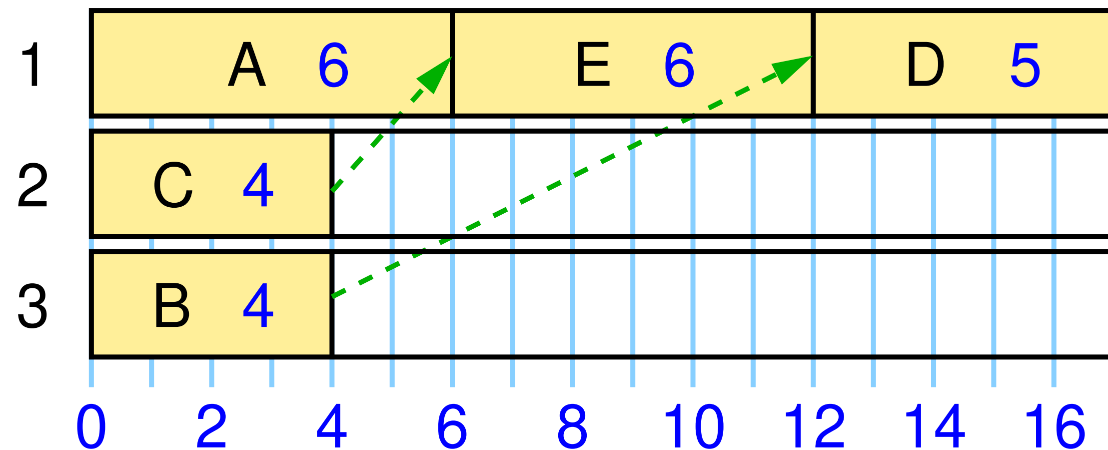


Beispiel: List-Scheduling mit HLFET

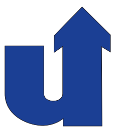


Liste:

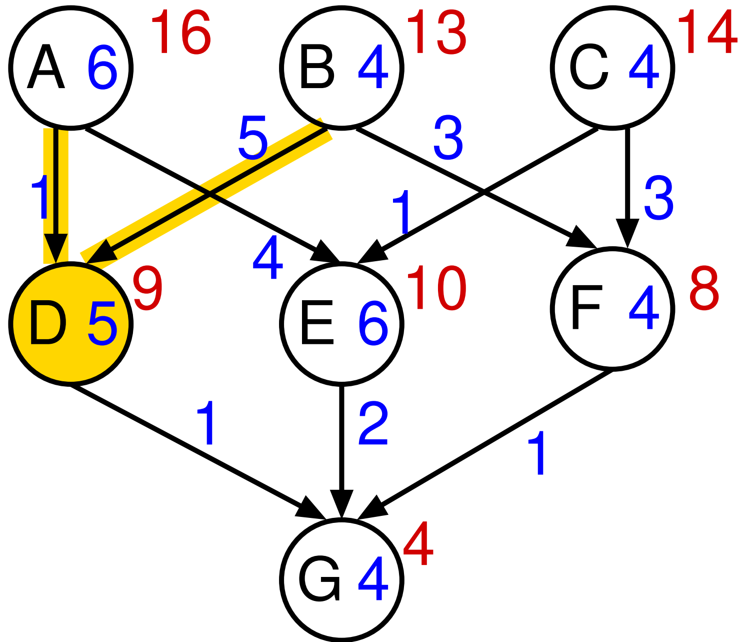
Schedule mit 3 Knoten:



- Annahme: lokale Kommunikation kostet keine Zeit

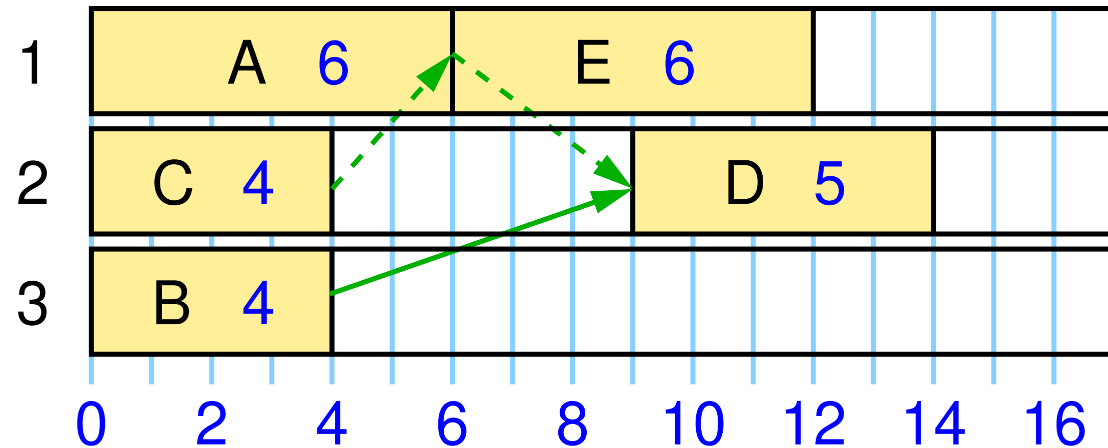


Beispiel: List-Scheduling mit HLFET

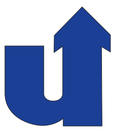


Liste:

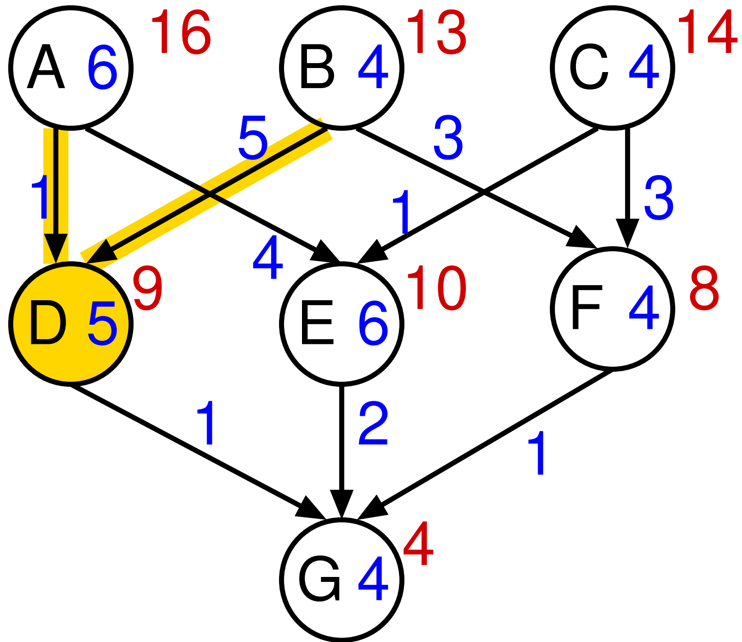
Schedule mit 3 Knoten:



- Annahme: lokale Kommunikation kostet keine Zeit

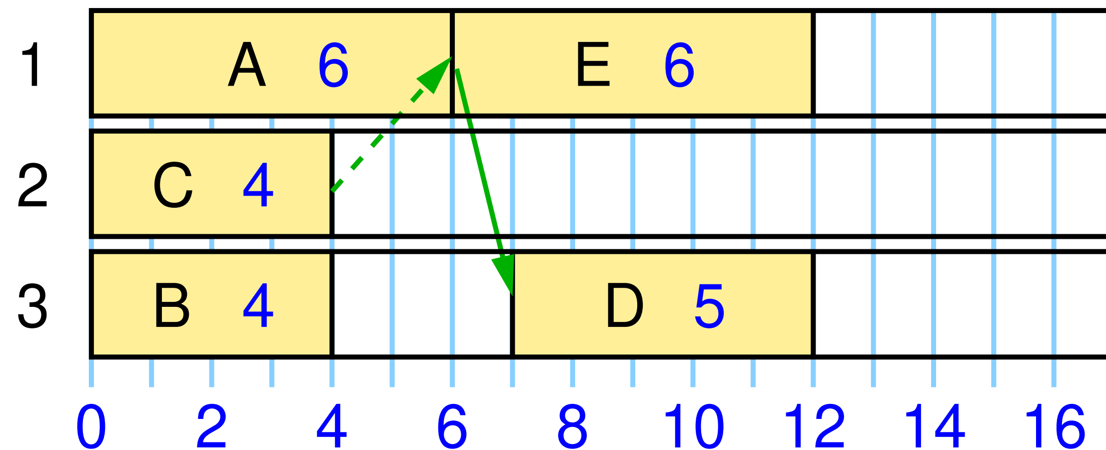


Beispiel: List-Scheduling mit HLFET

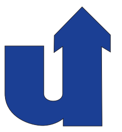


Liste:

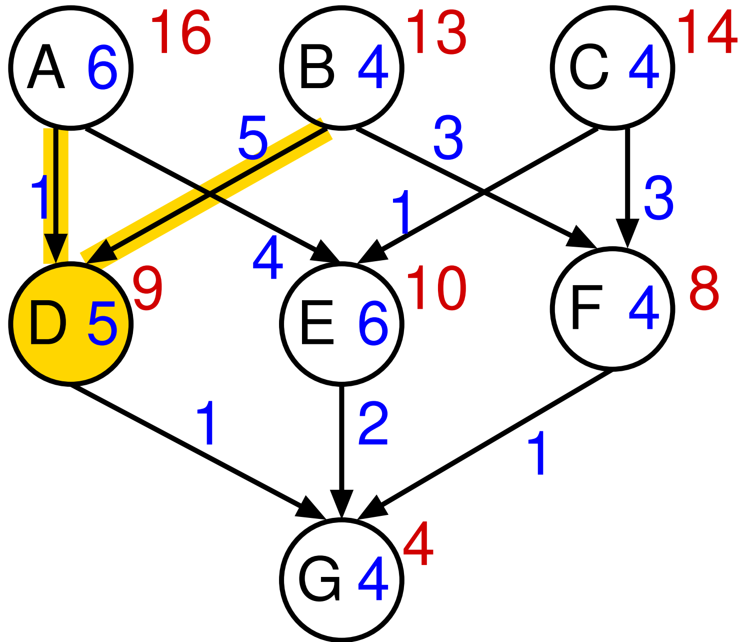
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit

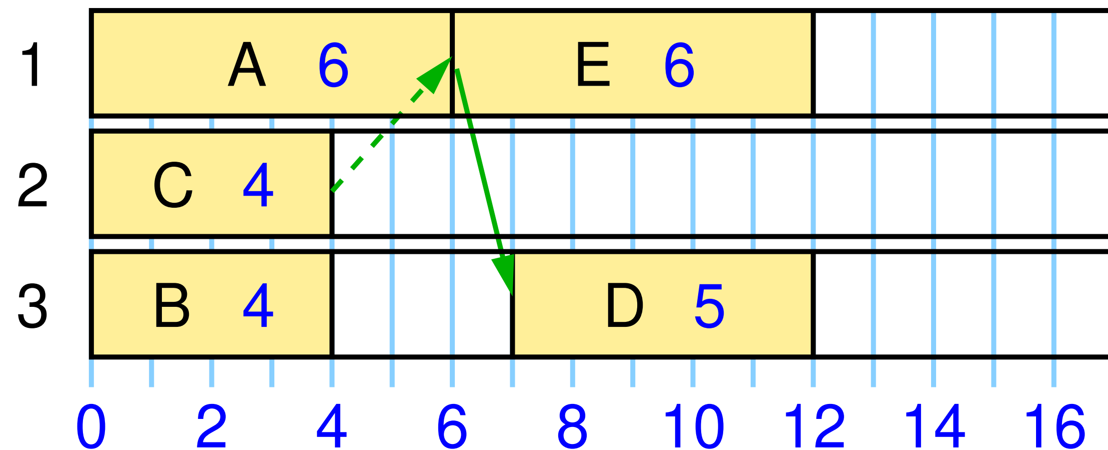


Beispiel: List-Scheduling mit HLFET

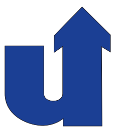


Liste:

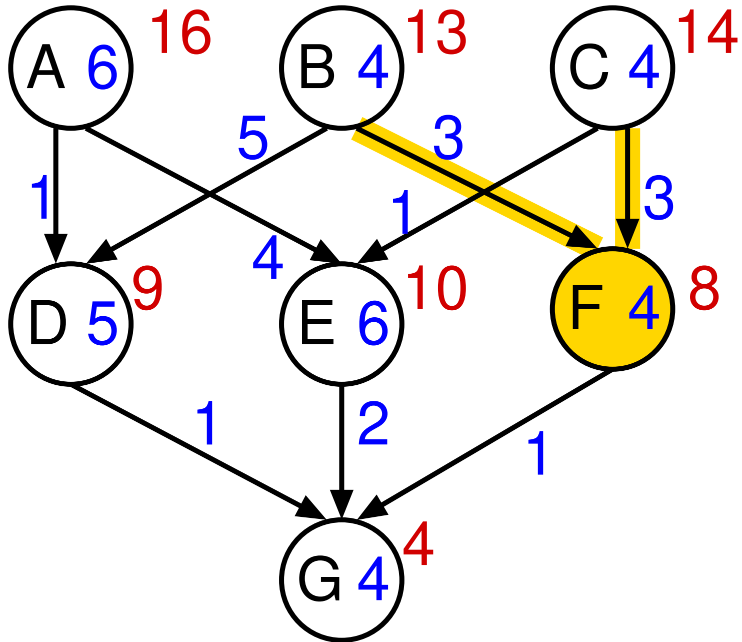
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit

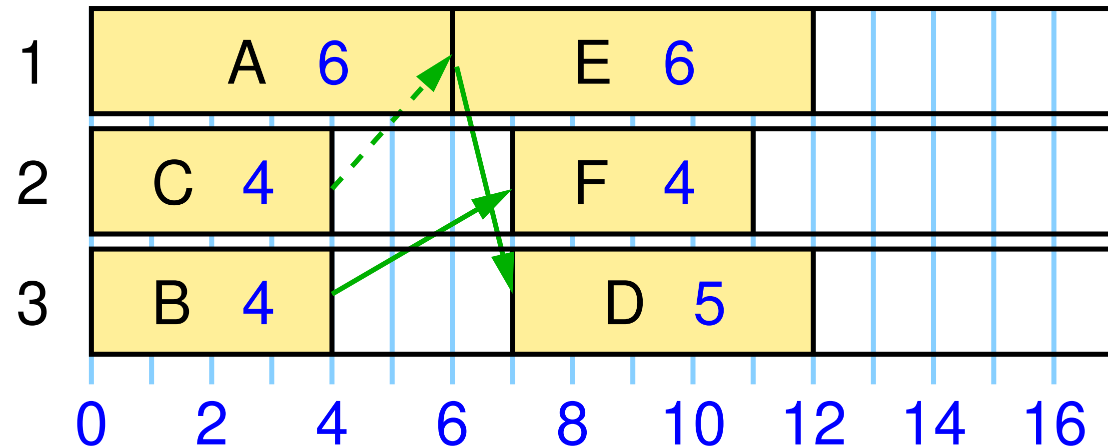


Beispiel: List-Scheduling mit HLFET

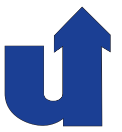


Liste:

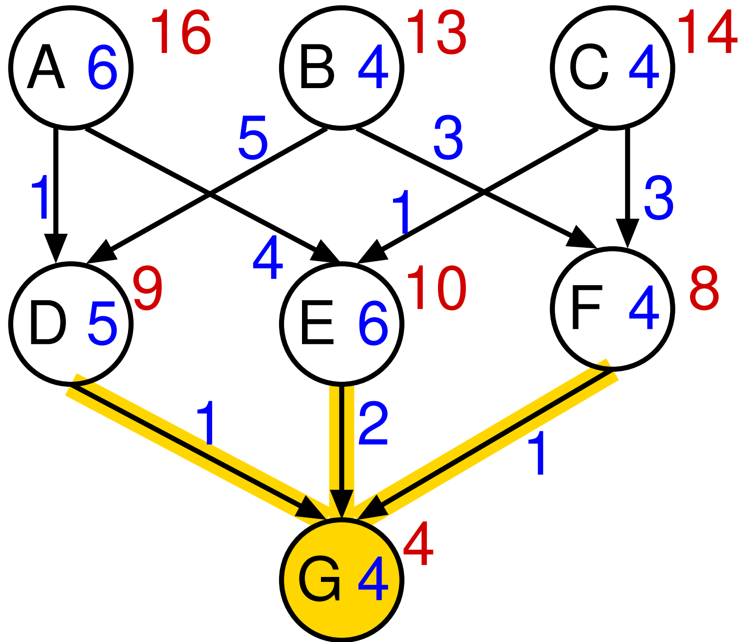
Schedule mit 3 Knoten:



➥ Annahme: lokale Kommunikation kostet keine Zeit

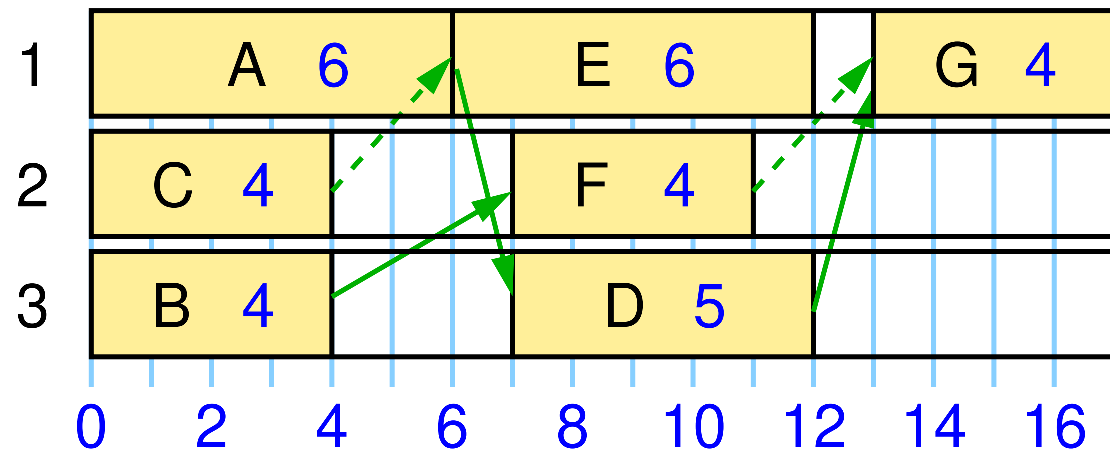


Beispiel: List-Scheduling mit HLFET

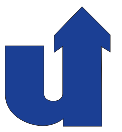


Liste:

Schedule mit 3 Knoten:

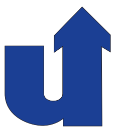


- Annahme: lokale Kommunikation kostet keine Zeit



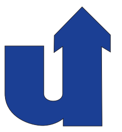
5.1.2 Dynamischer Lastausgleich

- Bestandteile eines Lastausgleichs-Systems
  - *Information policy* – wann wird der Lastausgleich angestoßen?
    - bei Bedarf, periodisch, bei Zustandsänderungen, ...
  - *Transfer policy* – unter welcher Bedingung wird Last verschoben?
    - oft: Entscheidung mit Hilfe von Schwellwerten
  - *Location policy* – wie wird der Empfänger (bzw. Sender) gefunden?
    - Polling einiger Knoten, Broadcast, ...
  - *Selection policy* – welche Tasks werden verschoben?
    - neue Tasks, lange Tasks, ortsunabhängige Tasks, ...



Typische Ansätze zum dynamischen Lastausgleich

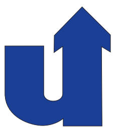
- Senderinitiierte Lastverteilung
  - neuer Prozeß i.a. auf lokalem Knoten gestartet
  - falls Knoten überlastet: Last anderer Knoten ermitteln und Prozeß auf niedrig belastetem Knoten starten
    - z.B. frage zufällig ausgewählten Knoten nach Last, sende Prozeß falls Last $\leq$ Schwellwert, sonst: nächster Knoten
  - Nachteil: zusätzliche Arbeit für ohnehin überlasteten Knoten!
- Empfängerinitiierte Lastverteilung
  - bei Terminierung eines Prozesses: prüfe ob noch genügend Arbeit (Prozesse) vorhanden sind
  - falls nicht: frage andere Knoten nach Arbeit
- Analog auch für präemptiven dynamischen Lastausgleich



5.2 Codemigration

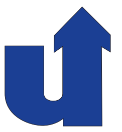
[Tanenbaum/Steen, 3.4]

- In verteilten Systemen neben Übertragung von Daten häufig auch Übertragung von Programmen
  - teilweise auch während deren Ausführung
- Motivation: Leistung und Flexibilität
  - präemptiv dynamischer Lastausgleich
  - Optimierung der Kommunikation (schiebe Code zu den Daten bzw. stark interaktiven Code zum Client)
  - Erhöhung der Verfügbarkeit (Migration vor Systemwartung)
  - Nutzung spezieller HW- oder SW-Ressourcen
  - Nutzung / Räumung ungenutzter Arbeitsplatzrechner
  - Vermeidung von Code-Installation auf Client-Rechnern (dynamisches Laden des Codes vom Server)



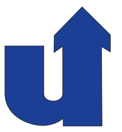
Modelle für die Code-Migration

- Modellvorstellung: ein Prozeß besteht aus drei „Segmenten“:
  - Codesegment
    - der ausführbare Programmcode des Prozesses
  - Ausführungssegment
    - gesamter Ausführungsstatus des Prozesses
      - virtueller Adreßraum (Daten, Heap, Stack)
      - Prozessor-Register (incl. Befehlszähler)
      - Prozeß-Kontrollblock
  - Ressourcensegment
    - enthält Verweise auf externe Ressourcen, die der Prozeß benötigt
      - z.B. Dateien, Geräte, andere Prozesse, Mailboxen, ...



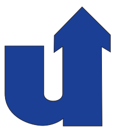
Modelle für die Code-Migration ...

- **Schwache Mobilität**
  - übertragen wird nur das Codesegment
    - ggf. mit Initialisierungsdaten
  - Programm wird immer von Ausgangszustand aus gestartet
  - Beispiele: Java-Applets, Laden entfernter Klassen in Java
- **Starke Mobilität**
  - übertragen werden Code- und Ausführungssegment
  - Verschiebung eines Prozesses in Ausführung
  - Beispiele: Prozeßmigration, Agenten
- Sender- bzw. empfänger-initiierte Migration



Probleme und Lösungen bei der Code-Migration

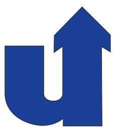
- Sicherheit: Zielrechner führt unbekannten Code aus (z.B. Applet)
  - eingeschränkte Umgebung (Sandbox)
  - signierter Code
- Heterogenität: Code- und Ausführungssegment sind abhängig von CPU und Betriebssystem
  - Einsatz virtueller Maschinen (z.B. JVM, XEN)
  - Migrationspunkte, an denen Zustand portabel gespeichert und gelesen werden kann (ggf. compilerunterstützt)
- Zugriff auf (lokale) Ressourcen
  - entfernter Zugriff mit globaler Referenz
  - Ressource verschieben oder kopieren
  - neue Bindung an Ressource desselben Typs



Prozeßmigration

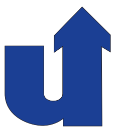
[Stallings, 14.1]

- Verschiebung eines bereits laufenden Prozesses
  - angestoßen durch BS oder den Prozeß selbst
  - meist zum dynamischen Lastausgleich
- Teilweise kombiniert mit *Checkpoint*/*Restart*-Funktion
  - statt den Zustand des Prozesses zu übertragen, kann dieser auch persistent gespeichert werden
- Entwurfsziele von Migrationsverfahren:
  - geringer Kommunikationsaufwand
  - nur kurze Blockierung des migrierten Prozesses
  - nach Migration keine Abhängigkeit vom Quellrechner mehr



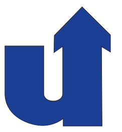
Ablauf einer Prozeßmigration

- Erzeugen eines neuen Prozesses auf dem Zielsystem
- Übertragen des Code- u. Ausführungssegments (Prozeßadreß-raum, Prozeßkontrollblock), Initialisierung des Zielprozesses
  - nötig: identische CPU und BS bzw. virtuelle Maschine
- Aktualisierung aller Verbindungen zu anderen Prozessen
  - Kommunikationsverbindungen, Signale, ...
  - während der Migration: Zwischenspeicherung bei Quelle
  - anschließend: Weiterleitung an Zielrechner
- Löschen des ursprünglichen Prozesses
  - ggf. Zurückbehalten eines „Schattenprozesses“ für umgeleite-te Systemaufrufe, z.B. Dateizugriffe



Übertragung des Prozeßadreßraums

- ***Eager (all)***: Übertrage gesamten Adreßraum
  - auf Quellknoten verbleiben keine Spuren des Prozesses
  - bei großem Adreßraum sehr teuer (v.a., wenn nicht alle Seiten genutzt werden)
  - häufig zusammen mit *Checkpoint*/*Restart*-Funktion
- ***Precopy***: Prozeß läuft während der Übertragung auf dem Quellknoten weiter
  - um Zeit zu minimieren, in der Prozeß blockiert ist
  - während der Übertragung modifizierte Seiten müssen nochmal gesendet werden



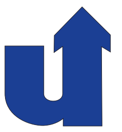
Übertragung des Prozeßadreßraums ...

- ***Eager (dirty)***: Übertrage nur modifizierte Seiten, die im Hauptspeicher sind
  - alle anderen Seiten werden erst bei Zugriff übertragen
    - Integration mit virtueller Speicherverwaltung
  - Motivation: Hauptspeicher des Quellknotens schnell „leeren“
  - Quellknoten bleibt ggf. bis Prozeßende involviert
- ***Copy-on-reference***: Übertrage jede Seite erst bei Zugriff
  - Variation von *Eager (dirty)*
  - geringste Anfangskosten
- ***Flushing***: Verdränge vor der Migration alle Seiten auf Festplatte
  - danach: *Copy-on-reference*
    - Vorteil: Hauptspeicher des Quellknotens wird entlastet

Verteilte Systeme

SoSe 2018

6 Zeit und globaler Zustand

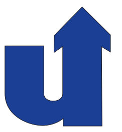


Inhalt

- Synchronisation physischer Uhren
- Lamport’sche Kausalitätsrelation
- Logische Uhren
- Globaler Zustand

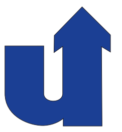
Literatur

- Tanenbaum, van Steen: Kap. 5.1-5.3
- Colouris, Dollimore, Kindberg: Kap. 10
- Stallings: Kap 14.2



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



Nebenläufigkeit vs. (echte) Parallelität

Beispiel: 4 Prozesse

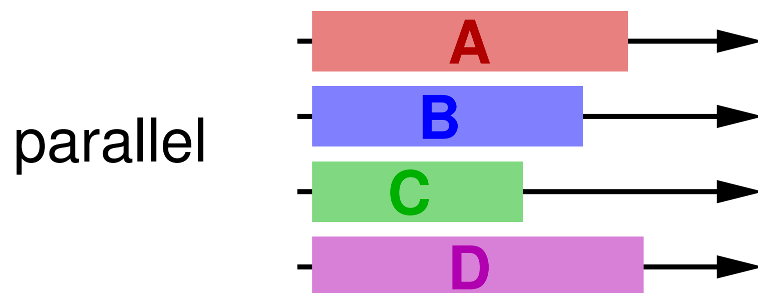
sequentiell

Ein Zeitstrahl, Prozesse werden nicht unterbrochen.

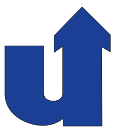
nebenläufig

Ein Zeitstrahl, Prozesse können jederzeit durch andere unterbrochen werden: verzahnte Ausführung.

parallel



Jeder Knoten / Prozeß hat seinen eigenen Zeitstrahl! Ereignisse in verschiedenen Prozessen können echt gleichzeitig stattfinden.

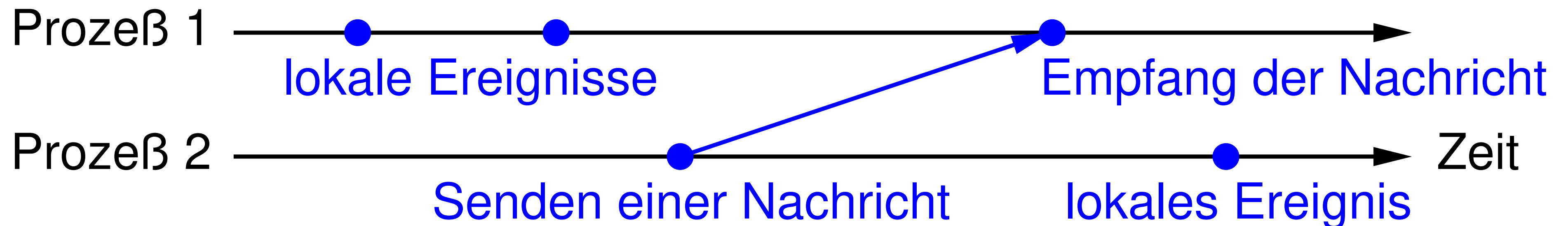


Globale Zeit

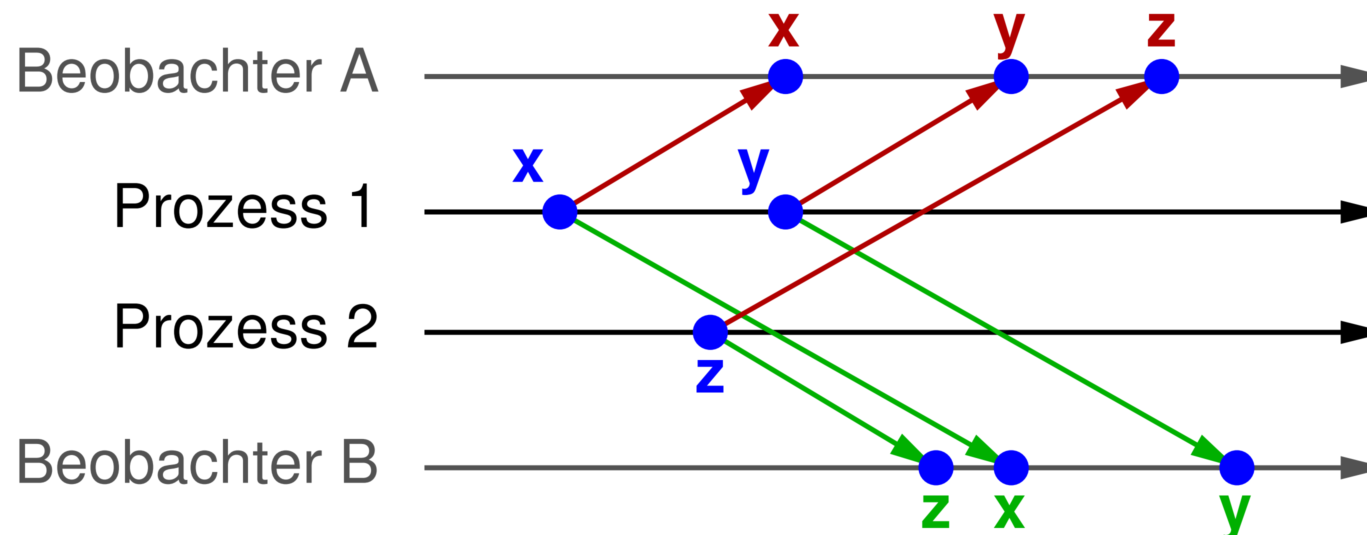
- Auf Ein-/Mehrprozessorsystem
  - jedem Ereignis kann (zumindest theoretisch) ein eindeutiger Zeitstempel derselben lokalen Uhr zugeordnet werden
  - bei Mehrprozessorsystemen: Synchronisation am gemeinsamen Speicher
- In verteilten Systemen:
  - viele lokale Uhren (eine pro Knoten)
  - exakte Synchronisation der Uhren (prinzipiell!) nicht möglich
  - $\Rightarrow$ Reihenfolge von Ereignissen auf verschiedenen Knoten nicht (immer) eindeutig zu ermitteln
  - (vgl. spezielle Relativitätstheorie)

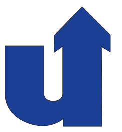
Eine Auswirkung der Verteiltheit

- Vorbemerkung: Ereignisse in verteilten Systemen



- Szenario: zwei Prozesse beobachten zwei andere Prozesse





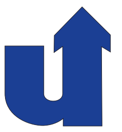
Eine Auswirkung der Verteiltheit ...

- 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!
- Auch aus Zeitstempeln der (lokalen) Uhren ist die Reihenfolge von Ereignissen nicht sinnvoll zu ermitteln
- Daher in solchen Fällen:
  - Ereignise mit Zeitstempeln **logischer Uhren** (☞ **6.3**)
  - logische Uhren erlauben Aussagen über kausale Reihenfolge

[Coulouris, 10.3]

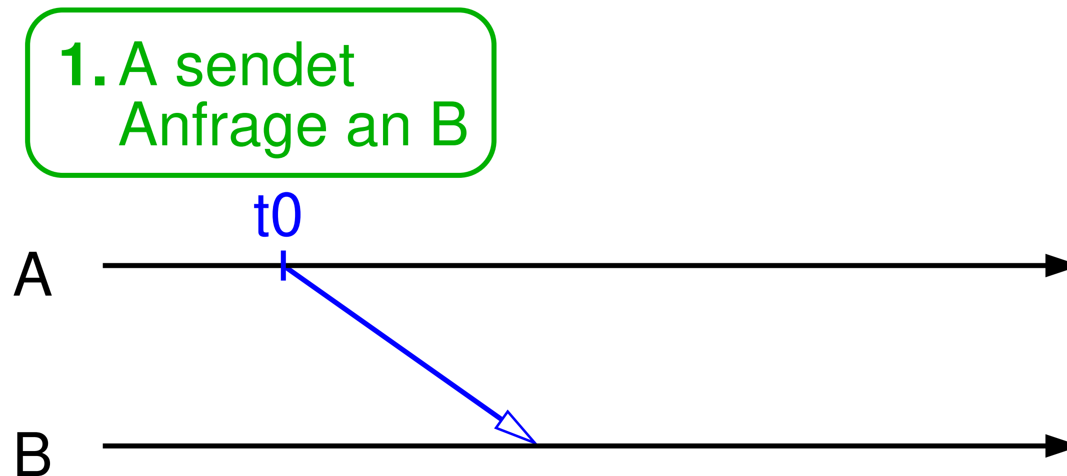
6.1 Synchronisation physischer Uhren

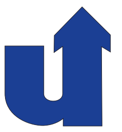
- Physische Uhr zeigt 'reale' Zeit
  - basierend auf UTC (*Universal Time Coordinated*)
- Jeder Rechner hat seine eigene (physische) Uhr
  - Quarzoszillator mit Zähler in HW und ggf. in SW
- Uhren weichen i.d.R. voneinander ab (***Offset***)
  - Offset ändert sich im Lauf der Zeit: ***Clock Drift***
    - typ. $\mathbf{10^{-6}}$ für Quarze, $\mathbf{10^{-13}}$ für Atomuhren
- Ziel der Uhrensynchronisation:
  - halte den Offset der Uhren unter einer festgelegten Schranke
  - ***Clock Skew***: maximal erlaubte Abweichung



Cristians Methode

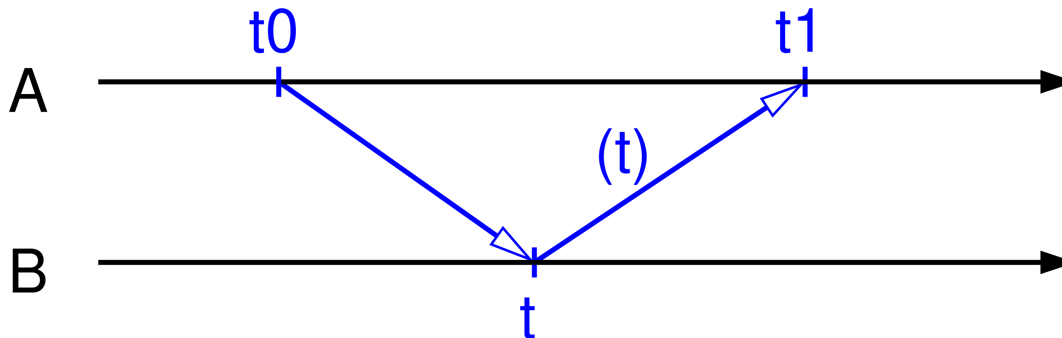
- Annahme: A und B wollen ihre Uhren miteinander synchronisieren
  - B kann auch ein Zeitserver sein (z.B. mit GPS-Uhr)
- Vorgehensweise:





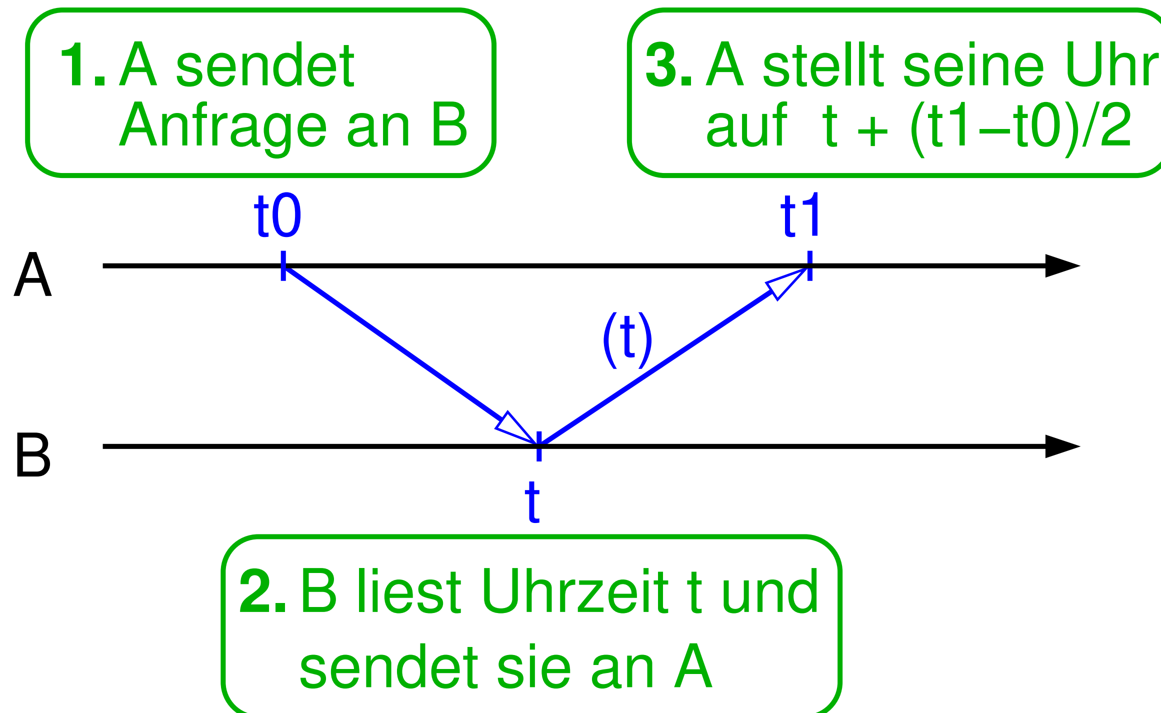
Cristians Methode

- Annahme: A und B wollen ihre Uhren miteinander synchronisieren
  - B kann auch ein Zeitserver sein (z.B. mit GPS-Uhr)
- Vorgehensweise:



Cristians Methode

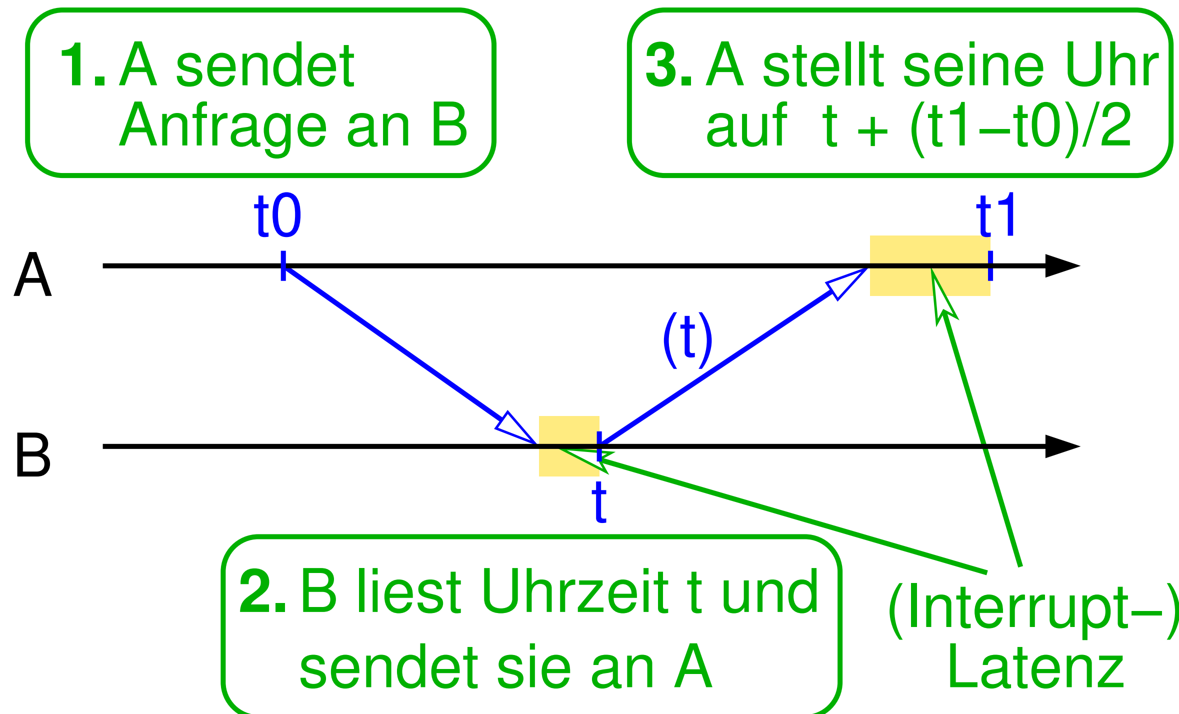
- Annahme: A und B wollen ihre Uhren miteinander synchronisieren
  - B kann auch ein Zeitserver sein (z.B. mit GPS-Uhr)
- Vorgehensweise:



- A muß Laufzeit der Antwort-Nachricht berücksichtigen
- Schätzung: Laufzeit = halbe *Round-Trip*-Zeit = $(t1 - t0)/2$

Cristians Methode

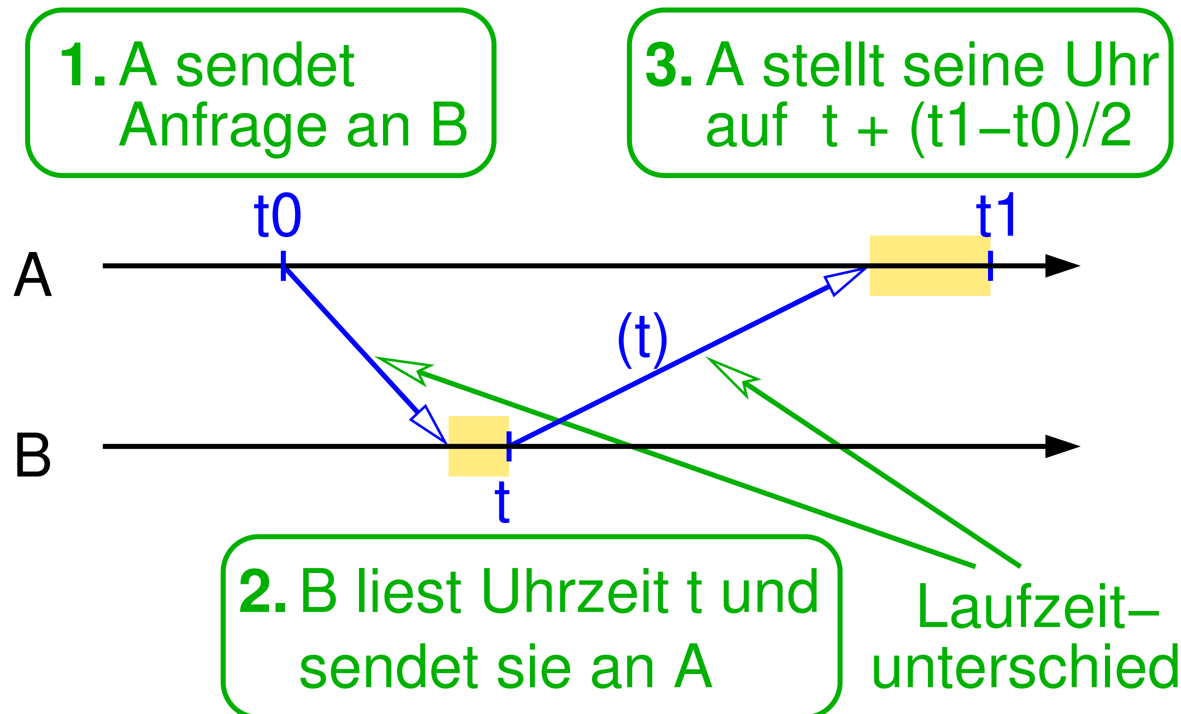
- Annahme: A und B wollen ihre Uhren miteinander synchronisieren
  - B kann auch ein Zeitserver sein (z.B. mit GPS-Uhr)
- Vorgehensweise:



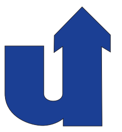
- A muß Laufzeit der Antwort-Nachricht berücksichtigen
- Schätzung: Laufzeit = halbe *Round-Trip*-Zeit = $(t1 - t0)/2$

Cristians Methode

- Annahme: A und B wollen ihre Uhren miteinander synchronisieren
  - B kann auch ein Zeitserver sein (z.B. mit GPS-Uhr)
- Vorgehensweise:

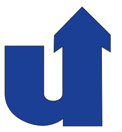


- A muß Laufzeit der Antwort-Nachricht berücksichtigen
- Schätzung: Laufzeit = halbe *Round-Trip*-Zeit = $(t1 - t0)/2$



Cristians Methode: Diskussion

- Problem: Laufzeiten beider Nachrichten sind u.U. verschieden
  - systematische Unterschiede (verschiedene Wege / Latenzen)
  - statistische Schwankungen der Laufzeit
- Genauigkeitsschätzung, falls minimale Laufzeit (min) bekannt:
  - B kann t frühestens zur Zeit $t0 + min$, spätestens zur Zeit $t1 - min$ bestimmt haben (gemessen mit der Uhr von A)
  - damit Genauigkeit $\pm\ ((t1 - t0)/2 - min)$
- Verbesserung der Genauigkeit:
  - mehrfache Durchführung des Nachrichtenaustauschs
  - verwende denjenigen mit minimaler *Round-Trip*-Zeit

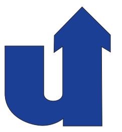


Umstellen der Uhr

- Zurückdrehen ist problematisch
  - Reihenfolge / Eindeutigkeit von Zeitstempeln
- Nichtmonotones „Springen" der Uhrzeit ebenfalls problematisch
- Daher: Uhr wird i.a. langsam angepaßt
  - läuft schneller / langsamer, bis Gangunterschied ausgeglichen

Weitere Protokolle

- Berkeley-Algorithmus: Server berechnet Mittelwert aller Uhren
- NTP (*Network Time Protocol*): Hierarchie von Zeit-Servern im Internet mit periodischem Abgleich
- IEEE 1588: Uhrensynchronisation für Automatisierungssysteme

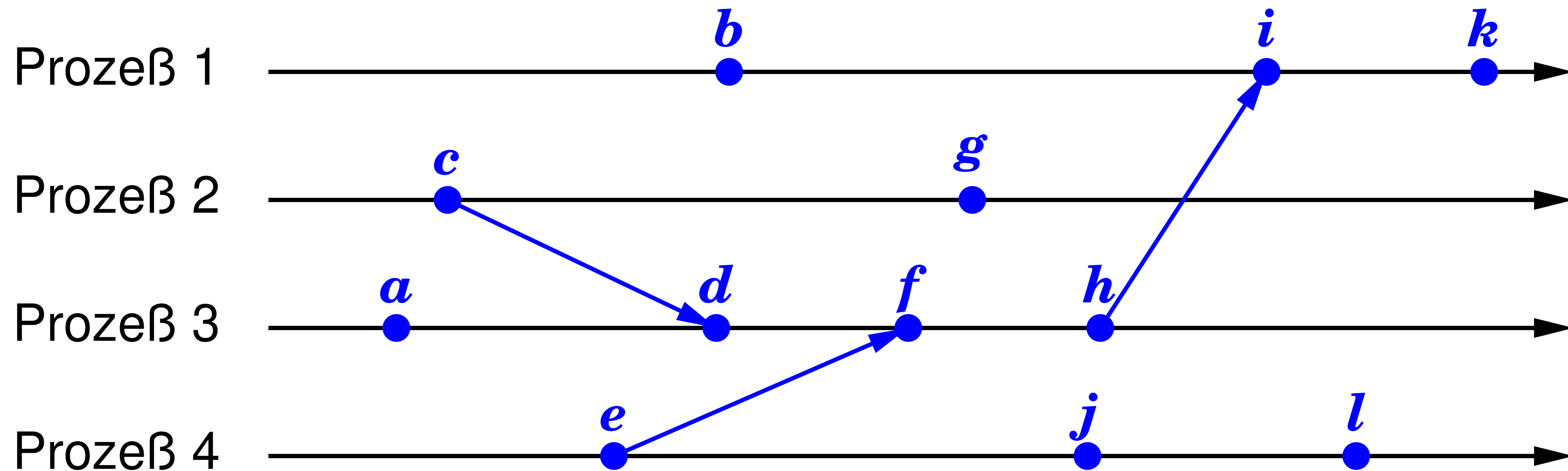


[Coulouris, 10.4]

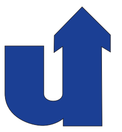
6.2 Die Lamport'sche Kausalitätsrelation

- In zwei Fällen kann die Reihenfolge von Ereignissen auch ohne globale Uhr bestimmt werden:
  - falls die Ereignisse im selben Prozeß sind, reicht lokale Uhr
  - das Senden einer Nachricht ist immer **vor** deren Empfang
- Definition der Kausalitätsrelation $\rightarrow$ (*happened before*)
  - falls Ereignisse a, b im selben Prozeß i sind und $t_i(a) < t_i(b)$ (t_i: Zeitstempel mit Uhr von i), so gilt $a \rightarrow b$
  - falls a das Senden einer Nachricht und b deren Empfang ist, so gilt $a \rightarrow b$
  - falls $a \rightarrow b$ und $b \rightarrow c$, so gilt auch $a \rightarrow c$ (Transitivität)
- $a \rightarrow b$ bedeutet, daß b kausal von a abhängen **kann**

Beispiele



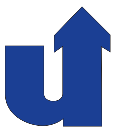
- Hier gilt u.a.:
  - $b \rightarrow i$ und $a \rightarrow h$ (Ereignisse im selben Prozeß)
  - $c \rightarrow d$ und $e \rightarrow f$ (Senden / Empfang einer Nachricht)
  - $c \rightarrow k$ und $a \rightarrow i$ (Transitivität)
  - $g \not\rightarrow l$ und $l \not\rightarrow g$: l und g sind **nebenläufig** (***concurrent***)



6.3 Logische Uhren

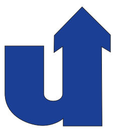
[Coulouris, 10.4]

- Physische Uhren können nicht exakt synchronisiert werden
  - daher: ungeeignet, um die **Reihenfolge** zu bestimmen, in der Ereignisse aufgetreten sind
- Logische Uhren
  - nehmen Bezug zur kausalen Ordnung von Ereignissen
  - kein fester Bezug zur realen Zeit
- Im folgenden:
  - Lamport-Zeitstempel
    - sind konsistent mit der kausalen Ordnung der Ereignisse
  - Vektor-Zeitstempel
    - erlauben kausale Sortierung von Ereignissen

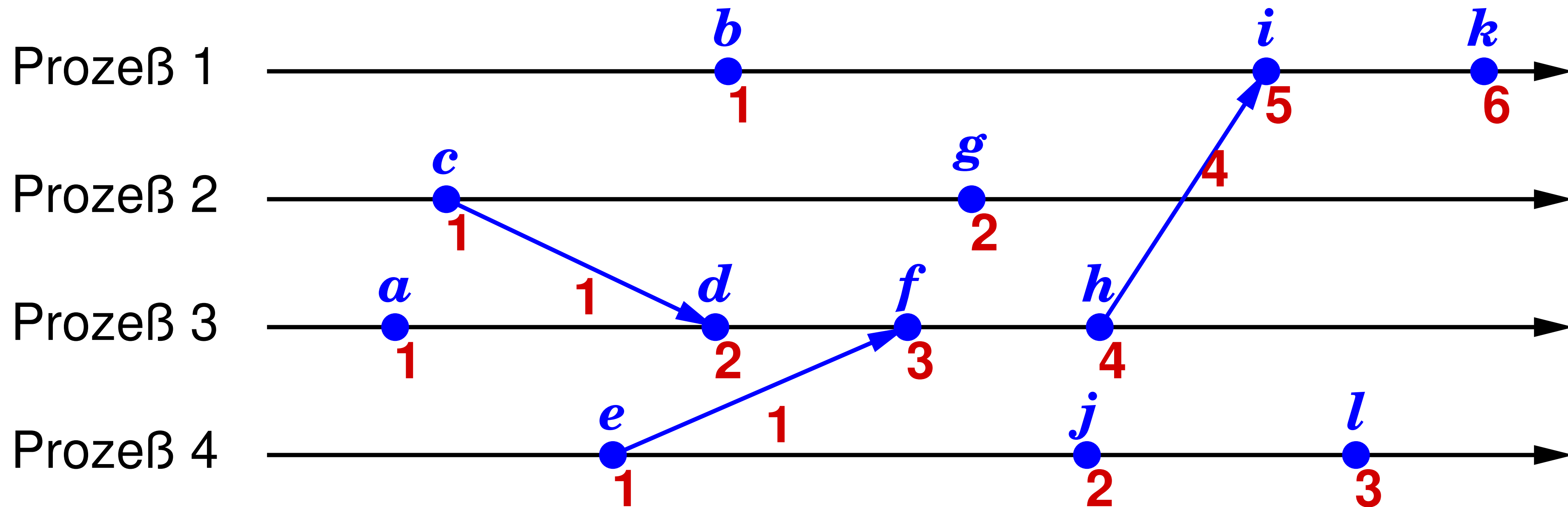


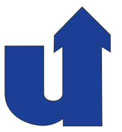
Lamport-Zeitstempel

- Lamport-Zeitstempel sind natürliche Zahlen
- Jeder Prozess i hat einen lokalen Zähler L_i, der wie folgt aktualisiert wird:
  - bei (genauer: vor) jedem lokalen Ereignis: $L_i = L_i + 1$
  - in jeder Nachricht wird auch der Zeitstempel L_i des Sendeereignisses mitgesendet
  - bei (ganauer: nach) Empfang einer Nachricht mit Zeitstempel t: $L_i = \max(L_i, t+1)$
- Lamport-Zeitstempel sind mit der Kausalität konsistent:
  - $a \rightarrow b \;\Rightarrow\; L(a) < L(b)$, wobei L der Lamport-Zeitstempel im jeweiligen Prozess ist
  - die Umkehrung gilt aber nicht!

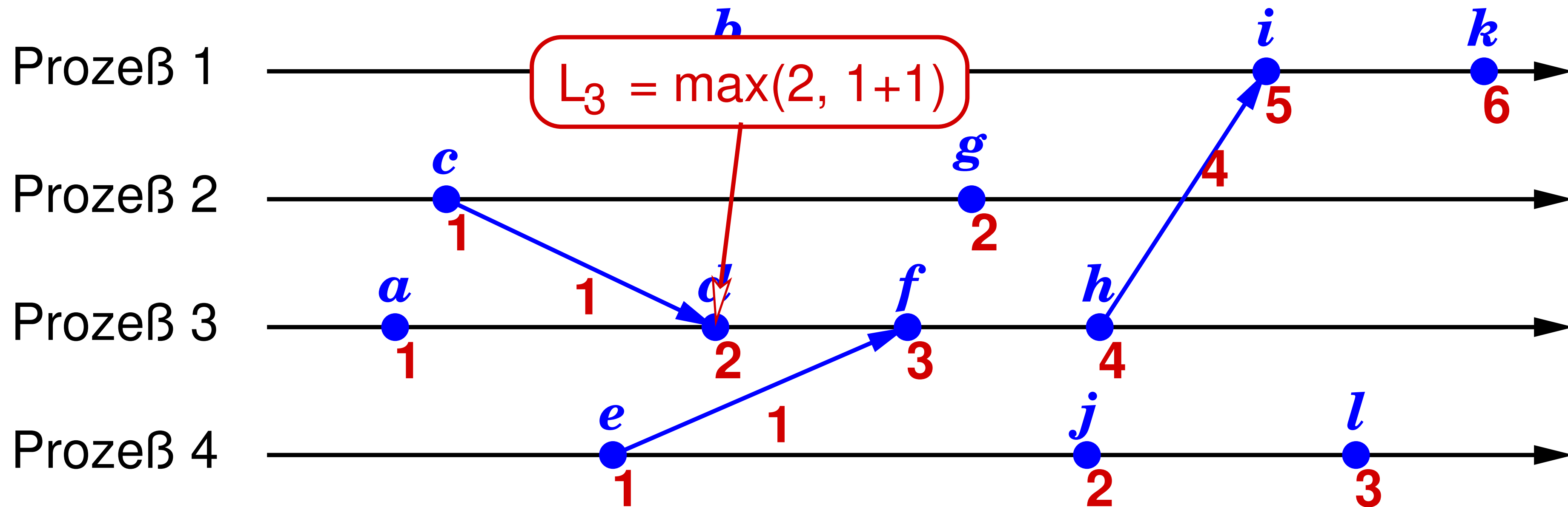


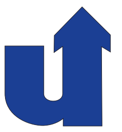
Lamport-Zeitstempel: Beispiel



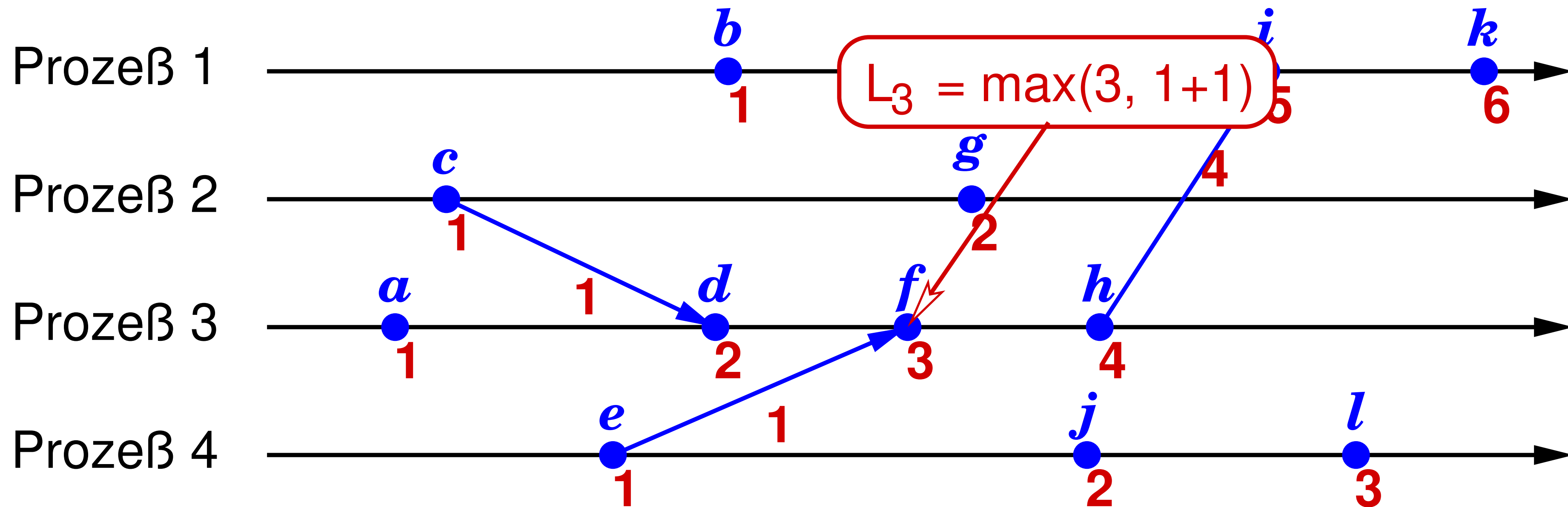


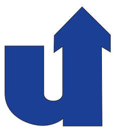
Lamport-Zeitstempel: Beispiel



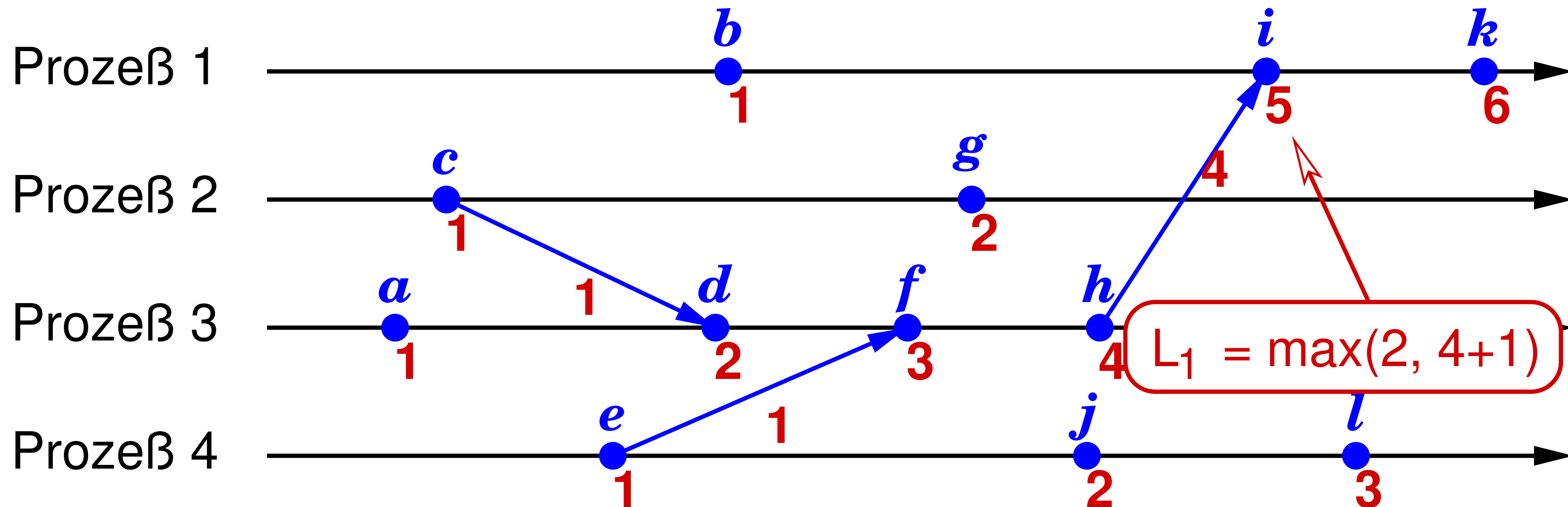


Lamport-Zeitstempel: Beispiel

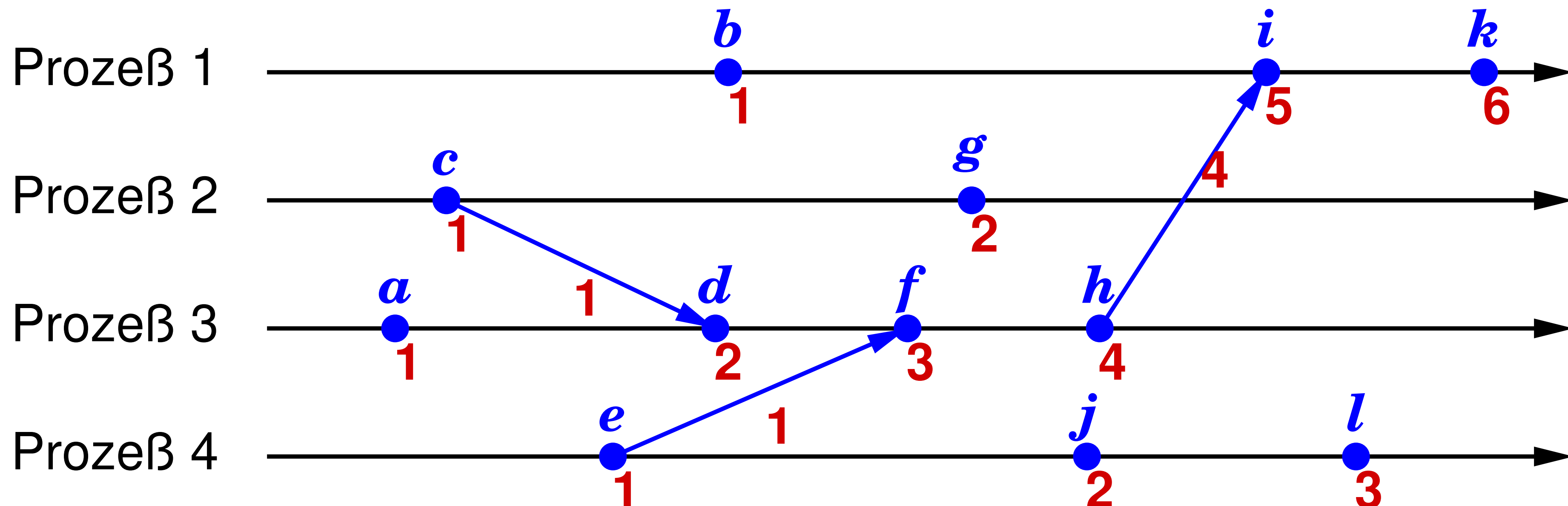




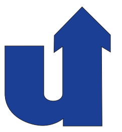
Lamport-Zeitstempel: Beispiel



Lamport-Zeitstempel: Beispiel

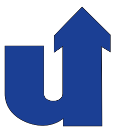


- Hier gilt u.a.:
  - $c \rightarrow k$ und $L(c) < L(k)$
  - $g \not\rightarrow j$ und $L(g) \not< L(j)$
  - $g \not\rightarrow l$, aber trotzdem $L(g) < L(l)$



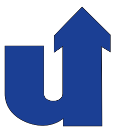
Vektor-Zeitstempel

- Ziel: Zeitstempel, die die Kausalität **charakterisieren**
  - $a \rightarrow b \;\Leftrightarrow\; V(a) < V(b)$, wobei V der Vektor-Zeitstempel im jeweiligen Prozess ist
- Eine Vektoruhr in einem System mit N Prozessen ist ein Vektor von N ganzen Zahlen
  - jeder Prozeß hat seinen eigenen Vektor V_i
  - $V_i[i]$: Zahl der bisher in Prozeß i aufgetretenen Ereignisse
  - $V_i[j], j \neq i$: Zahl der Ereignisse in Prozeß j, von denen i weiß
    - d.h. von denen er kausal beeinflußt worden sein könnte

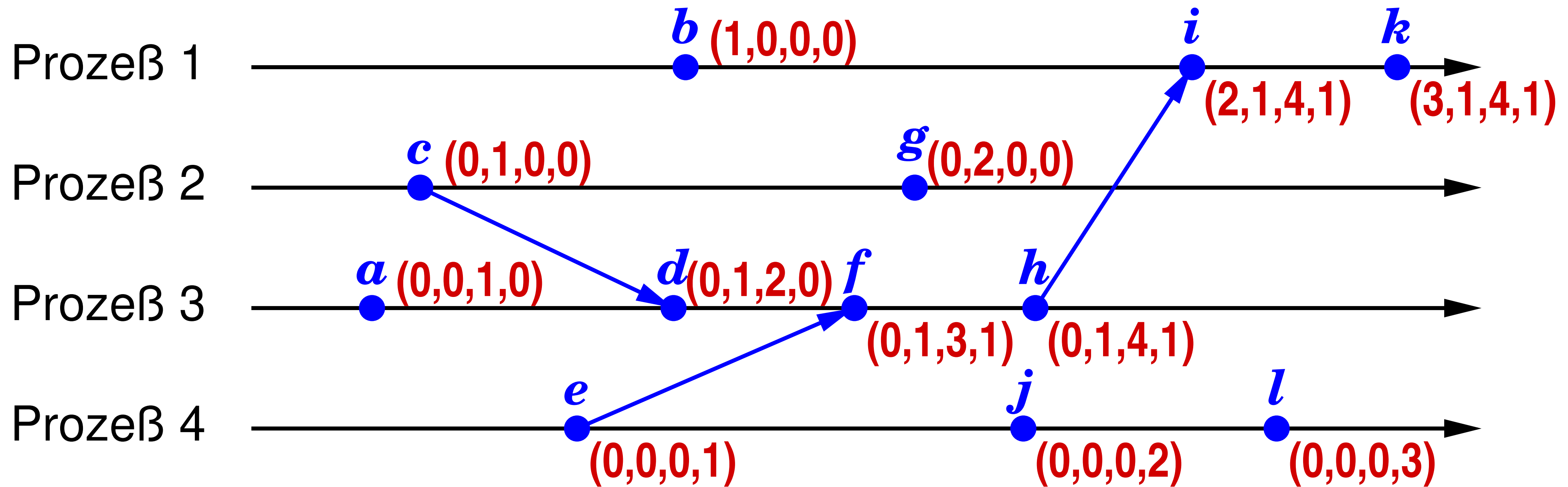


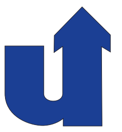
Vektor-Zeitstempel ...

- Aktualisierung von V_i in Prozeß i:
  - vor jedem lokalen Ereignis: $V_i[i] = V_i[i] + 1$
  - V_i wird in jeder Nachricht mitgesendet
  - nach Empfang einer Nachricht mit Zeitstempel t:
    $V_i[j] = \max(V_i[j], t[j])$ für alle $j = 1, 2, \ldots, N$
- Vergleich von Vektor-Zeitstempeln:
  - $V = V' \Leftrightarrow V[j] = V'[j]$ für alle $j = 1, 2, \ldots, N$
  - $V \leq V' \Leftrightarrow V[j] \leq V'[j]$ für alle $j = 1, 2, \ldots, N$
  - $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
  - die Relation $<$ definiert eine **partielle** Ordnung

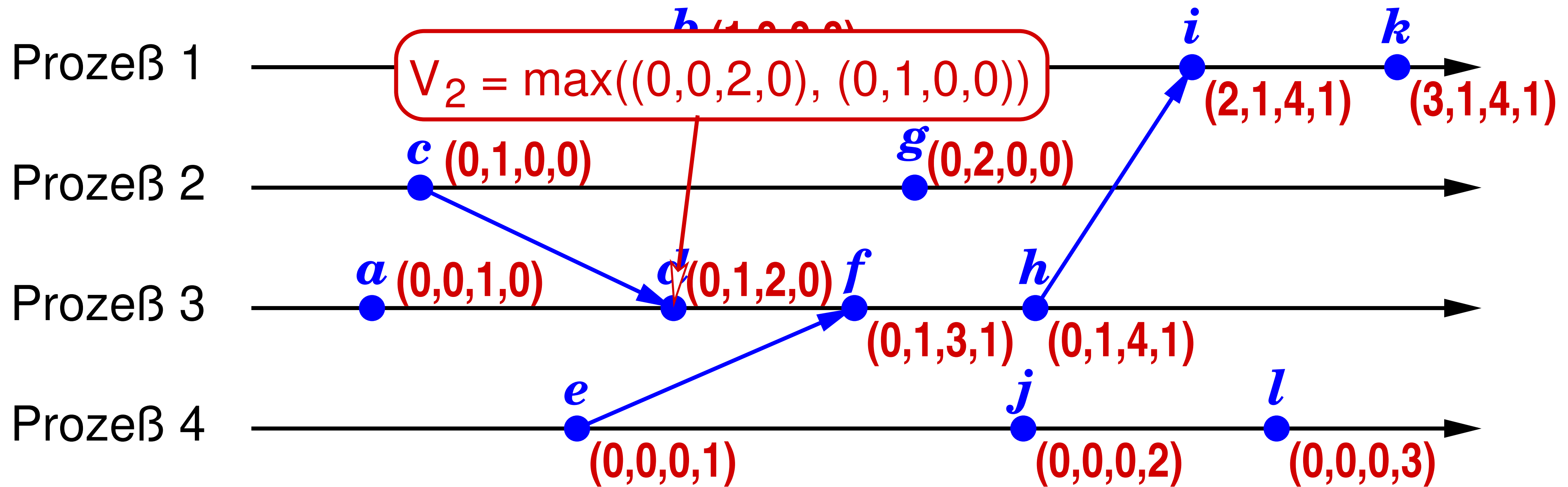


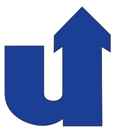
Vektor-Zeitstempel: Beispiel



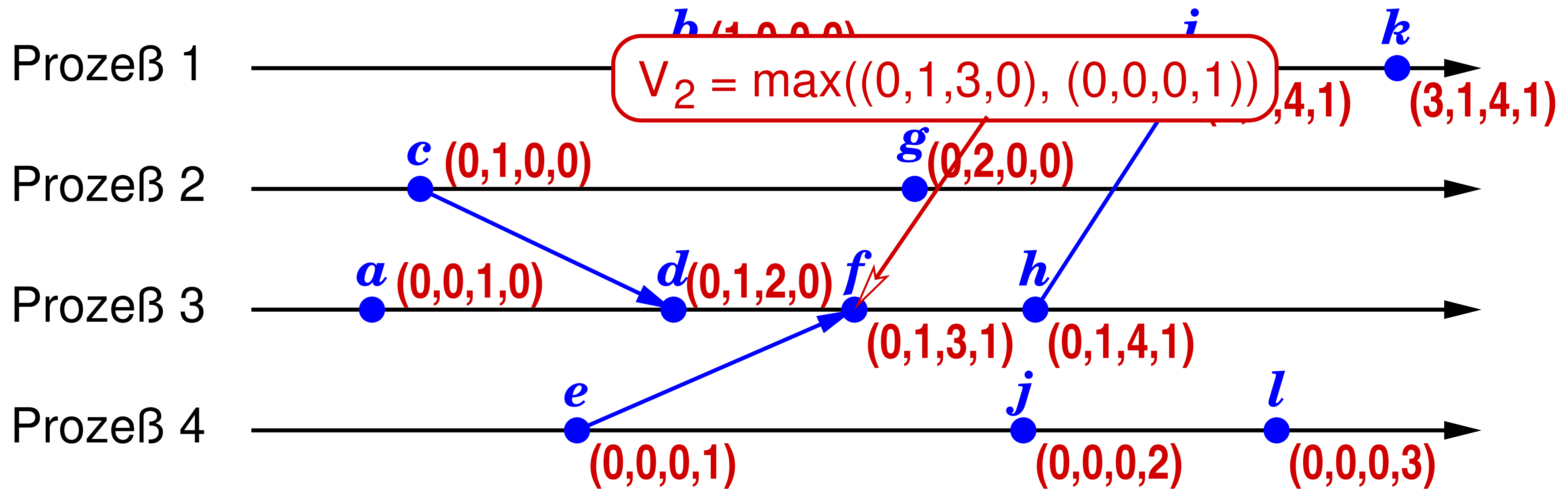


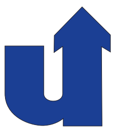
Vektor-Zeitstempel: Beispiel



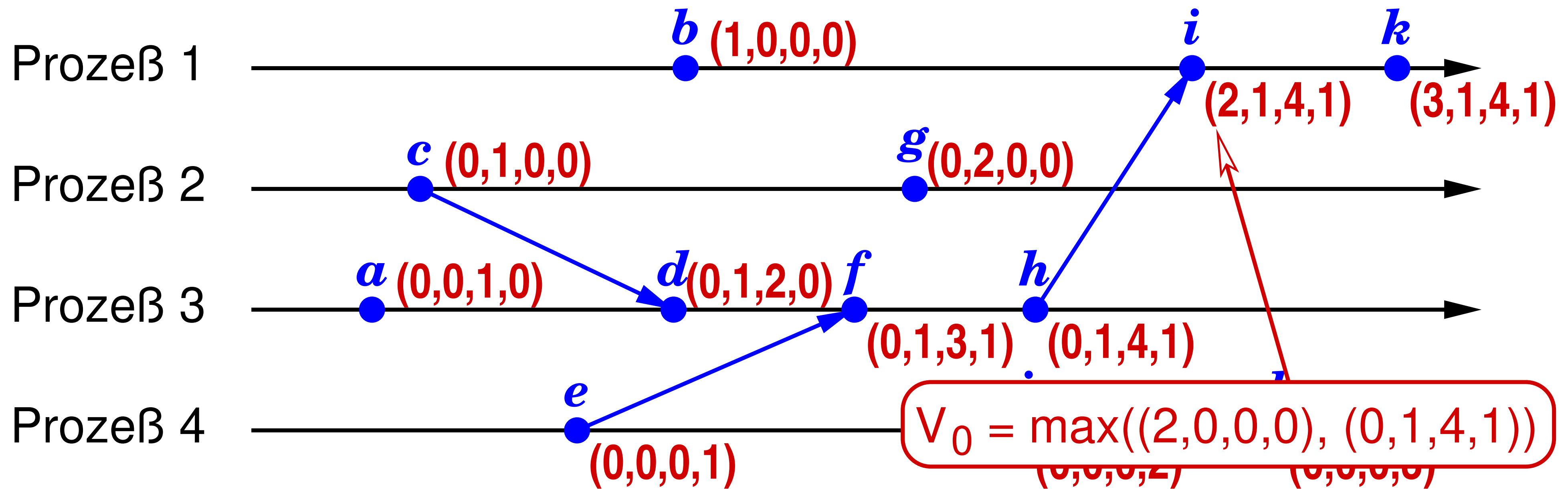


Vektor-Zeitstempel: Beispiel

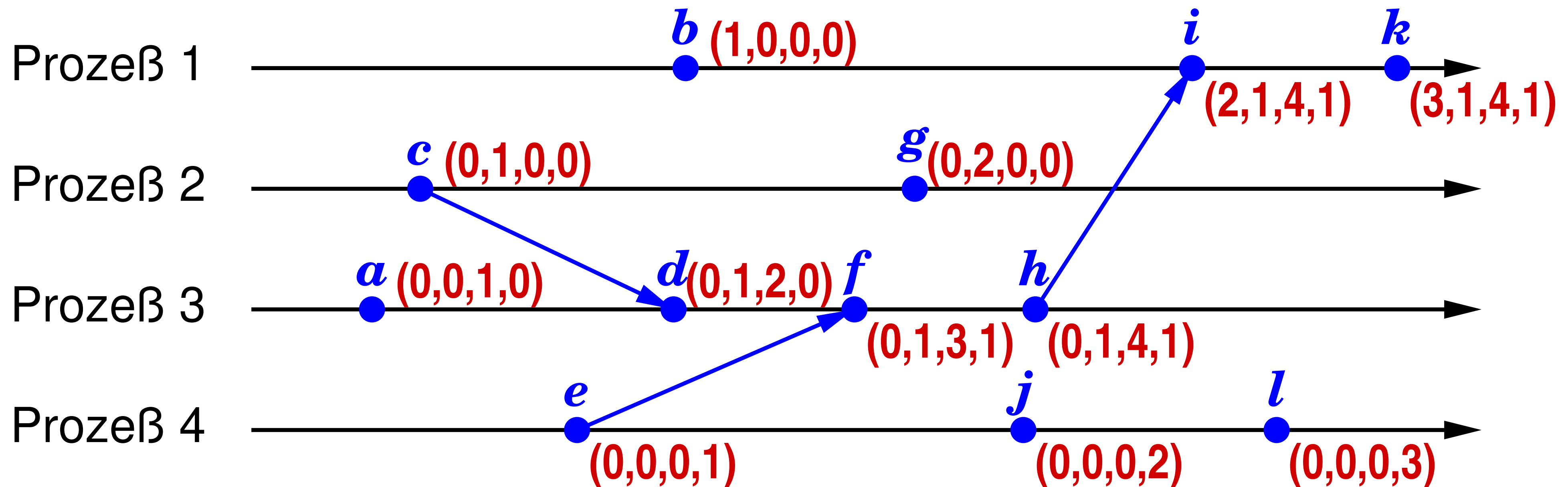




Vektor-Zeitstempel: Beispiel



Vektor-Zeitstempel: Beispiel



- Hier gilt u.a.:
  - $c \rightarrow k$ und $V(c) < V(k)$
  - $g \not\rightarrow l$ und $V(g) \not< V(l)$, sowie $l \not\rightarrow g$ und $V(l) \not< V(g)$
    - $V(l)$ und $V(g)$ nicht vergleichbar $\Leftrightarrow$ l und g nebenläufig

Ein Beispiel zur Motivation

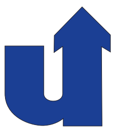
- Szenario: *Peer-to-Peer*-Anwendung, Prozesse senden sich gegenseitig Aufträge
- Frage: wann kann die Anwendung terminieren?
- Antwort: wenn kein Prozeß mehr einen Auftrag bearbeitet

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, ...

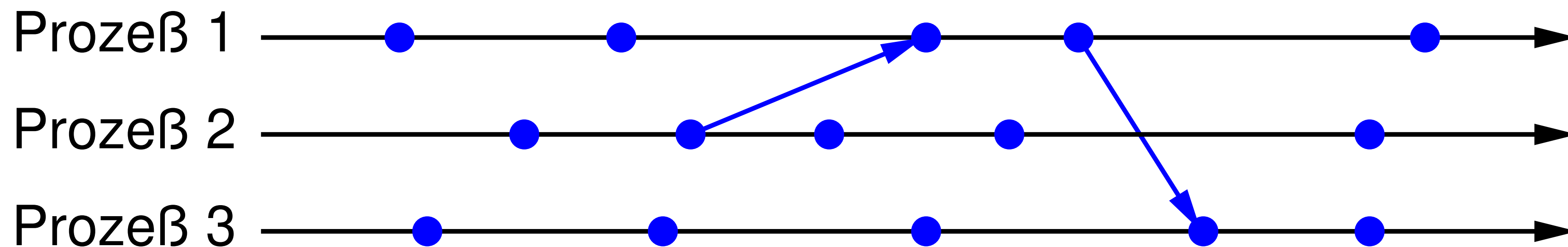
6.4 Globaler Zustand ...



- Wie bestimmt sich der Gesamtzustand eines verteilten Prozeß-systems?
  - 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? $\Rightarrow$ **konsistente Schnitte**

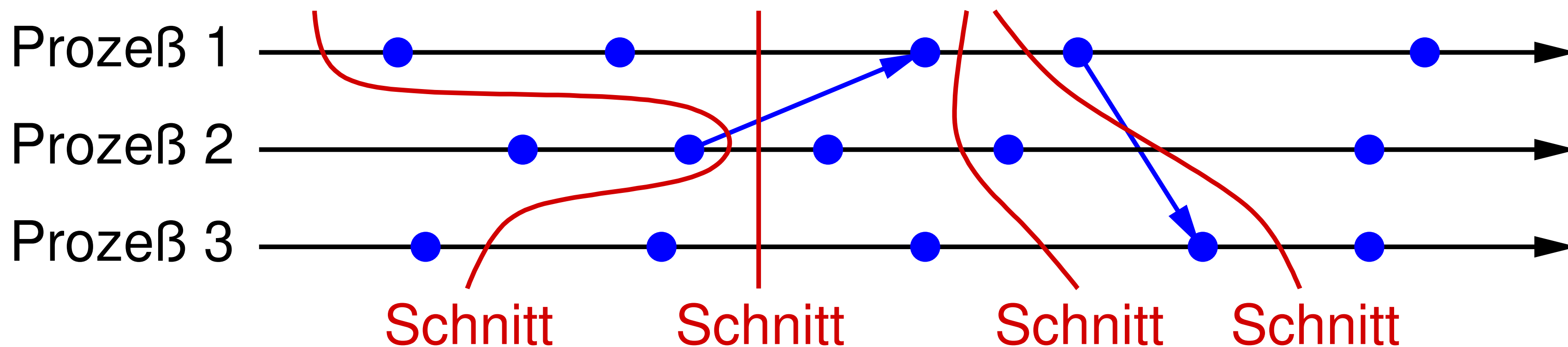
Konsistente Schnitte

- Ziel: bilde aus (nicht zeitgleich ermittelten) lokalen Zuständen einen sinnvollen globalen Zustand
- Prozesse modelliert durch Folgen von Ereignissen:



Konsistente Schnitte

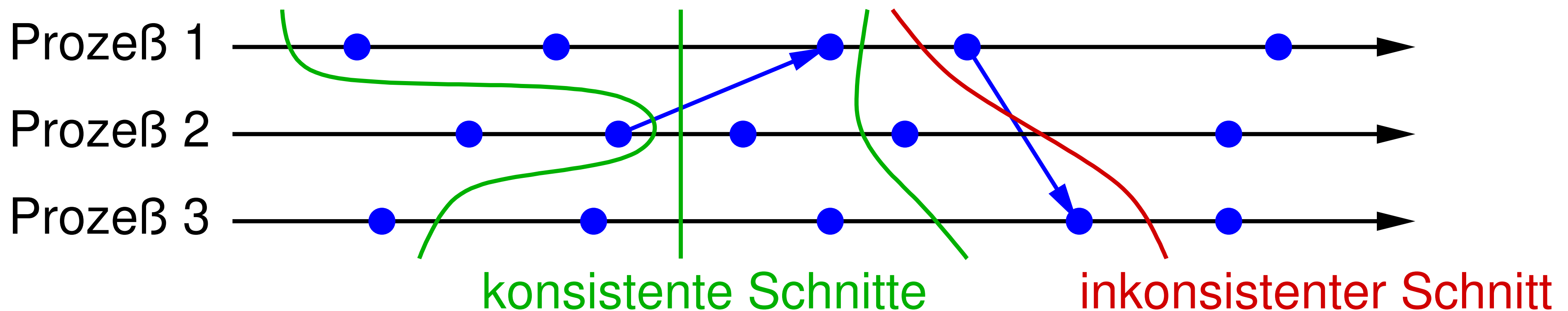
- Ziel: bilde aus (nicht zeitgleich ermittelten) lokalen Zuständen einen sinnvollen globalen Zustand
- Prozesse modelliert durch Folgen von Ereignissen:



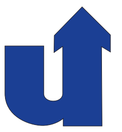
- **Schnitt**: betrachte in jedem Prozeß ein **Präfix** der Ereignisfolge

Konsistente Schnitte

- Ziel: bilde aus (nicht zeitgleich ermittelten) lokalen Zuständen einen sinnvollen globalen Zustand
- Prozesse modelliert durch Folgen von Ereignissen:

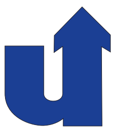


- **Schnitt**: betrachte in jedem Prozeß ein **Präfix** der Ereignisfolge
- **Konsistenter Schnitt**:
  - falls der Schnitt den Empfang einer Nachricht beinhaltet, so beinhaltet er auch das Senden dieser Nachricht



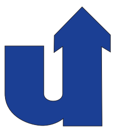
Der Schnappschuß-Algorithmus von Chandy und Lamport

- Ermittelt online einen „Schnappschuß“ des globalen Zustands
  - d.h.: einen konsistenten Schnitt
- Der globale Zustand besteht aus:
  - den lokalen Zuständen aller Prozesse
  - dem Zustand aller Kommunikationsverbindungen
    - d.h. den Nachrichten in Übertragung
- Annahmen / Eigenschaften:
  - zuverlässige Nachrichtenkanäle mit Reihenfolgeerhaltung
  - Prozeßgraph ist stark zusammenhängend
  - jeder Prozeß kann jederzeit einen Schnappschuß auslösen
  - die Prozesse werden während des Algorithmus nicht blockiert



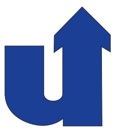
Der Schnappschuß-Algorithmus von Chandy und Lamport ...

- Wenn ein Prozeß einen Schnappschuß initiieren will:
  - Prozeß zeichnet zunächst seinen lokalen Zustand auf
  - dann sendet er eine Marker-Nachricht über jeden ausgehenden Kanal
- Wenn ein Prozeß eine Marker-Nachricht empfängt:
  - falls er seinen lokalen Zustand noch nicht gespeichert hat:
    - Prozeß zeichnet seinen lokalen Zustand auf
    - und sendet Marker über jeden ausgehenden Kanal
  - sonst:
    - Prozeß zeichnet Zustand des (Empfangs-)Kanals auf
      - d.h., alle Nachrichten, die seit Speicherung des lokalen Zustands eingetroffen sind

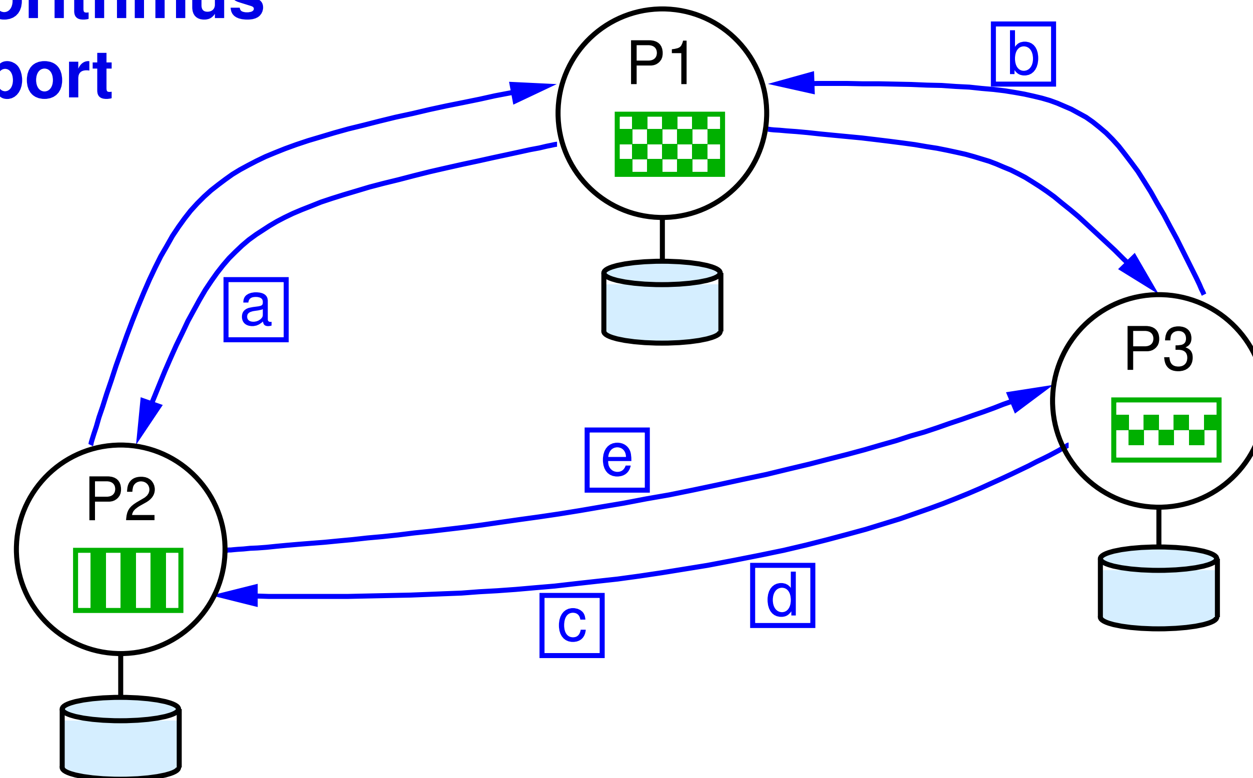


Der Schnappschuß-Algorithmus von Chandy und Lamport ...

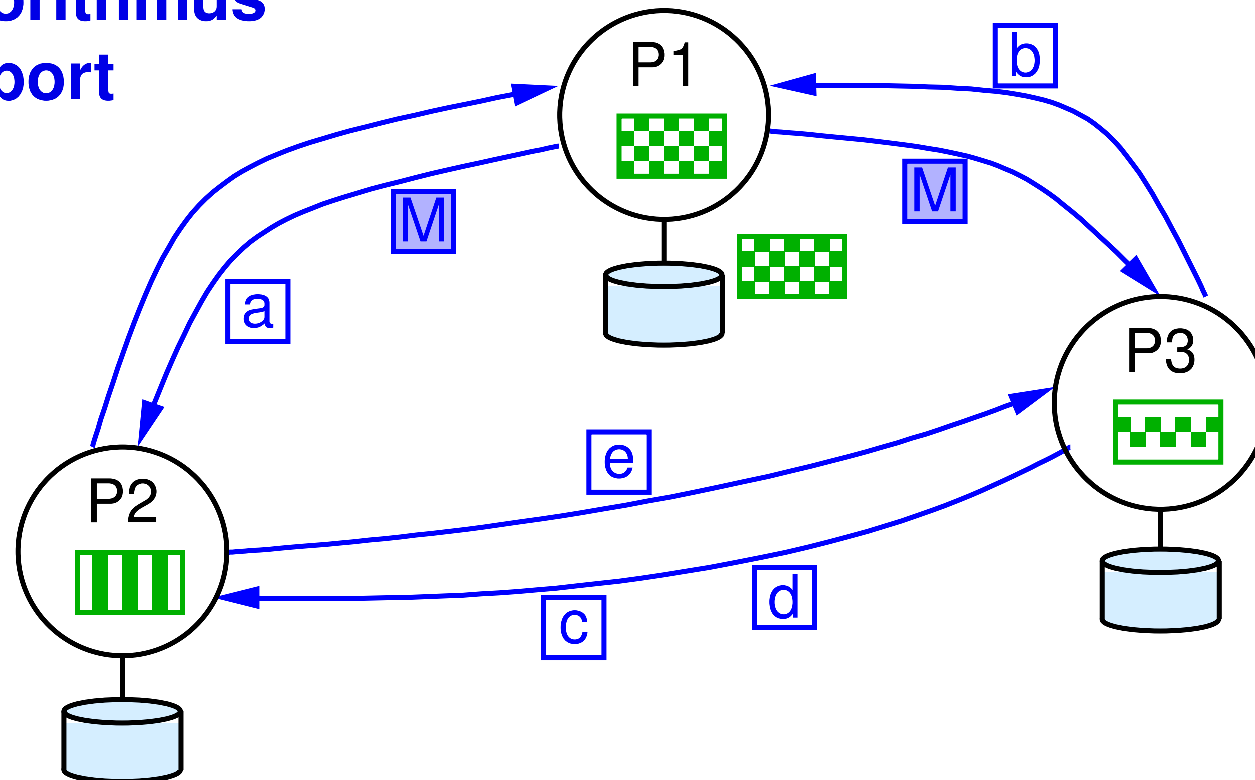
- Der Algorithmus ist beendet, wenn jeder Prozeß über jeden Kanal eine Marker-Nachricht erhalten hat
  - der ermittelte konsistente Schnitt ist dann (zunächst) verteilt gespeichert



Beispiel zum Algorithmus von Chandy/Lamport

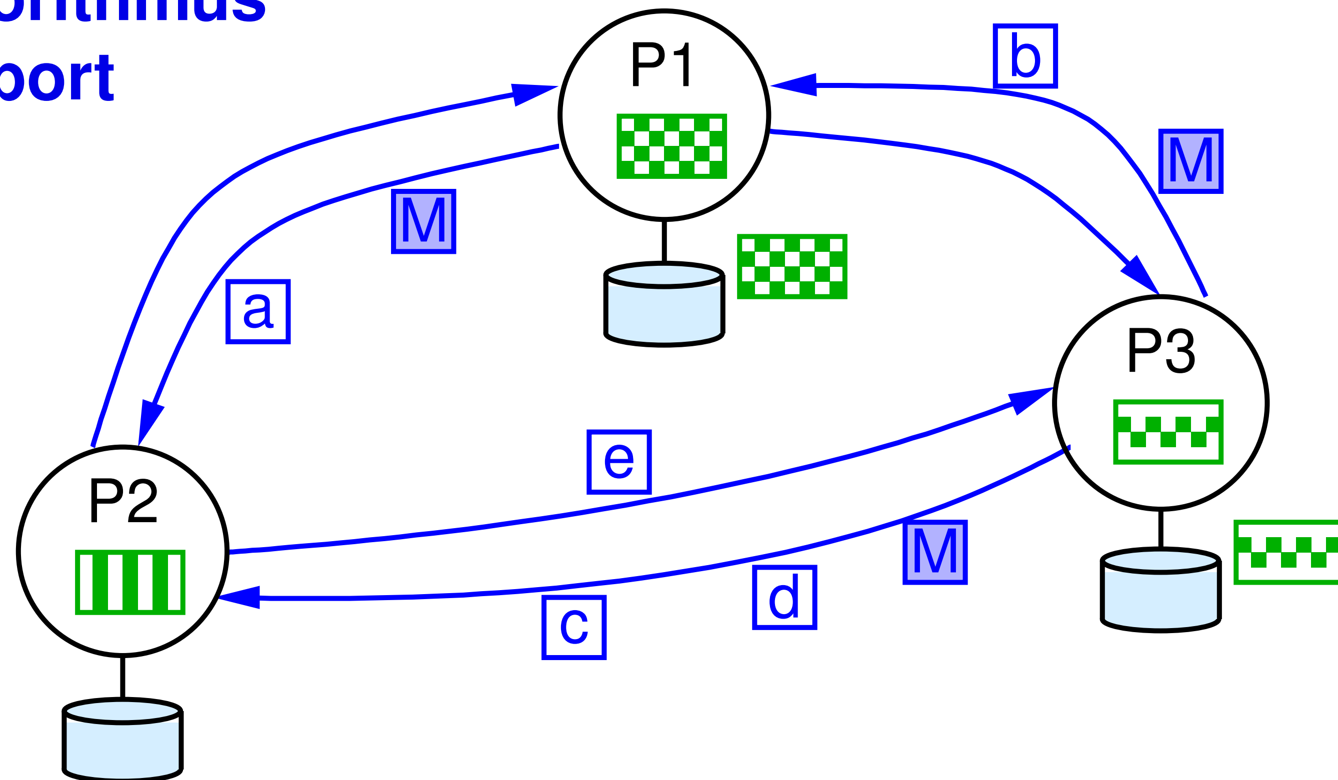


Beispiel zum Algorithmus von Chandy/Lamport



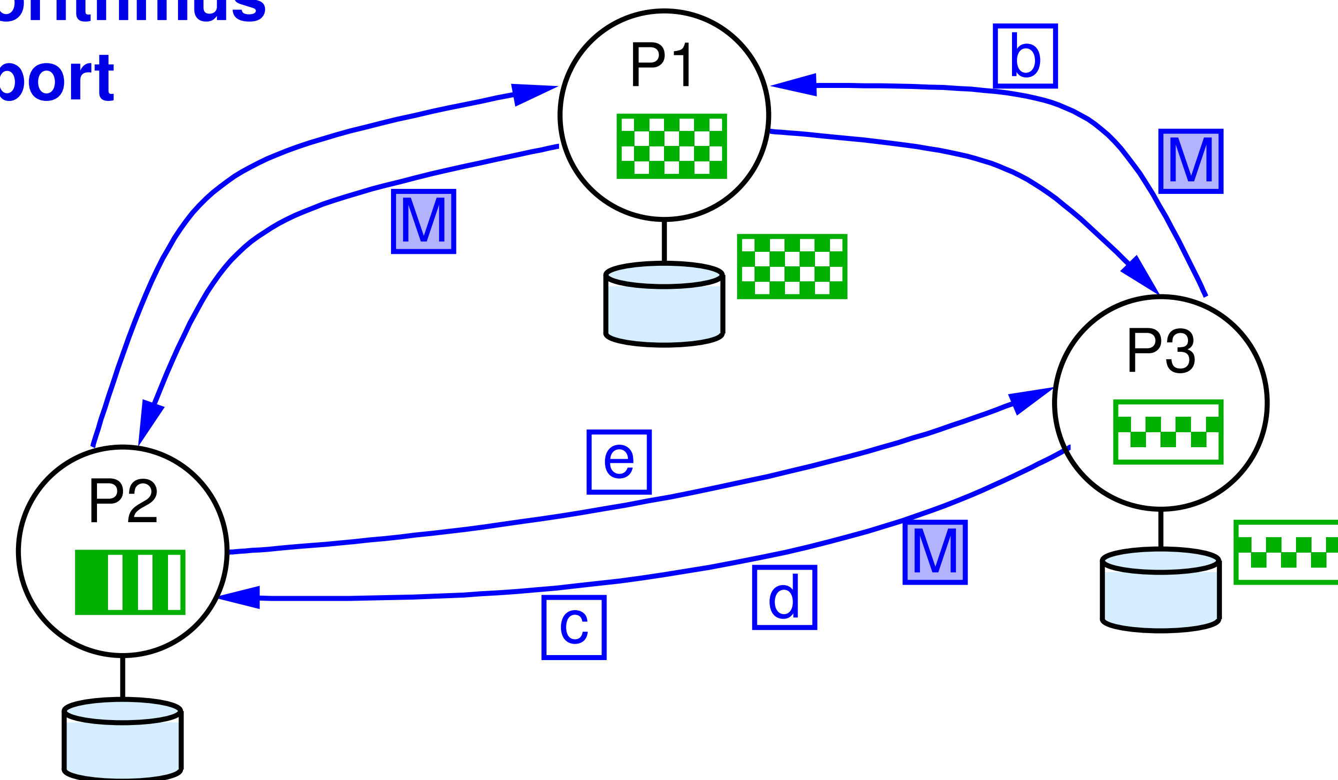
1. P1 initiiert Schnappschuß, sichert Zustand, sendet Marker

Beispiel zum Algorithmus von Chandy/Lamport



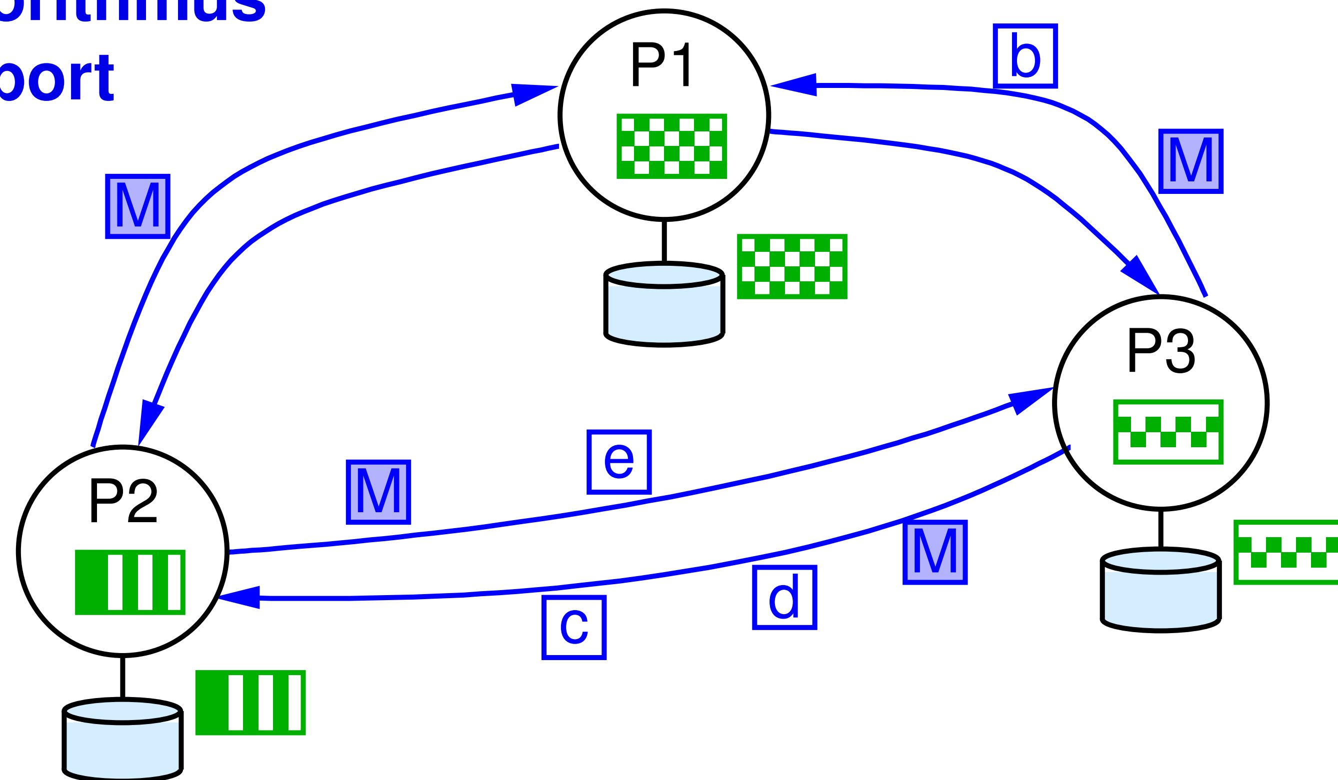
1. P1 initiiert Schnappschuß, sichert Zustand, sendet Marker
2. P3 empfängt Marker von P1, sichert Zustand, sendet Marker

Beispiel zum Algorithmus von Chandy/Lamport



1. P1 initiiert Schnappschuß, sichert Zustand, sendet Marker
2. P3 empfängt Marker von P1, sichert Zustand, sendet Marker
3. P2 empfängt und verarbeitet a

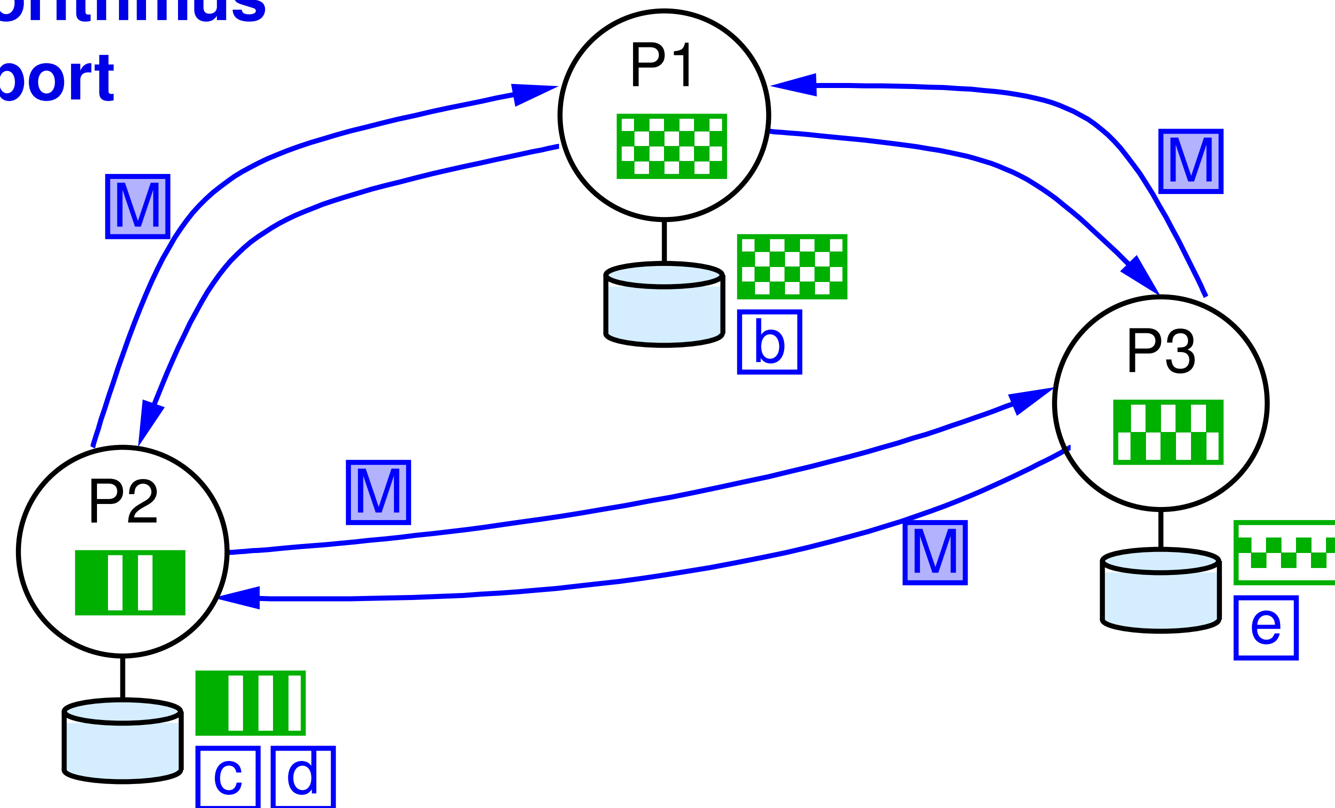
Beispiel zum Algorithmus von Chandy/Lamport



1. P1 initiiert Schnappschuß, sichert Zustand, sendet Marker
2. P3 empfängt Marker von P1, sichert Zustand, sendet Marker
3. P2 empfängt und verarbeitet a

   P2 empfängt Marker von P1, sichert Zustand, sendet Marker

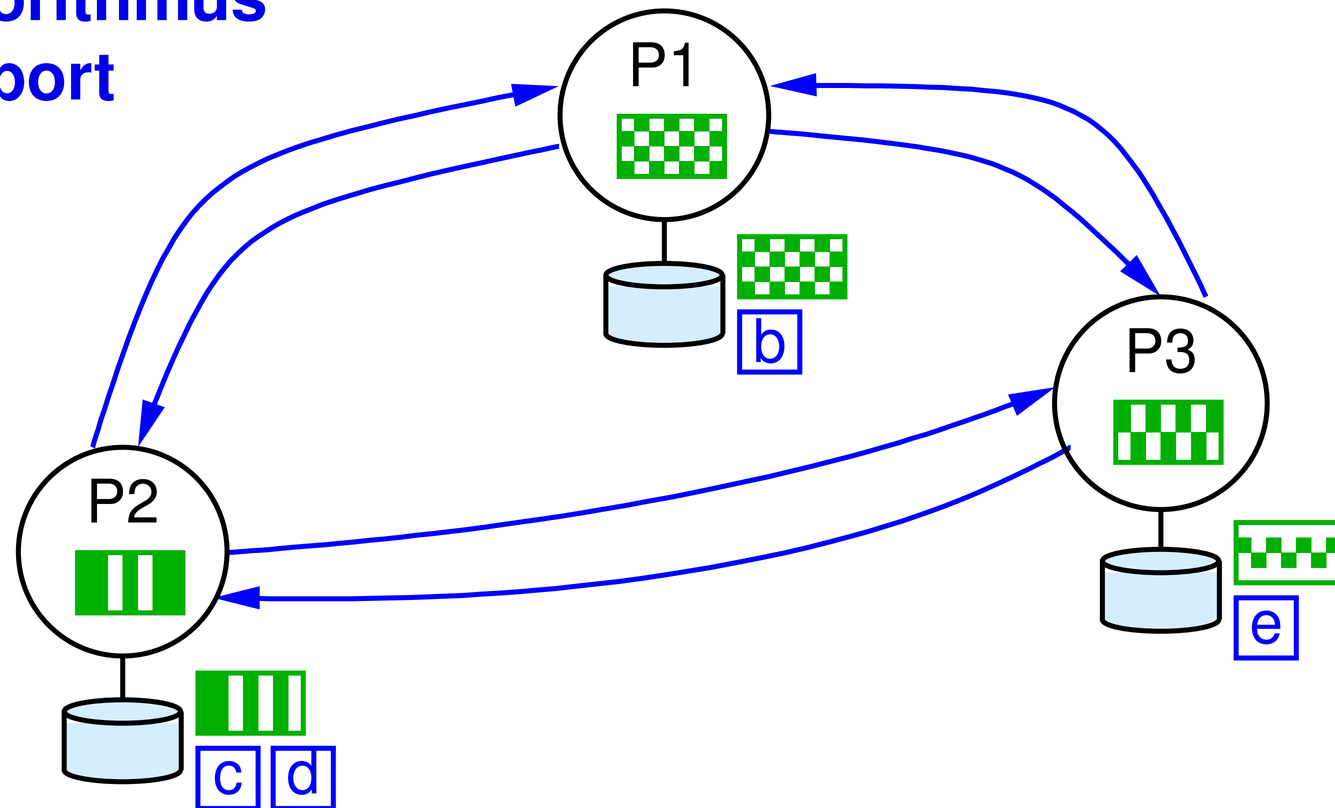
Beispiel zum Algorithmus von Chandy/Lamport



1. P1 initiiert Schnappschuß, sichert Zustand, sendet Marker
2. P3 empfängt Marker von P1, sichert Zustand, sendet Marker
3. P2 empfängt und verarbeitet a

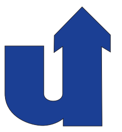
   P2 empfängt Marker von P1, sichert Zustand, sendet Marker
4. P1, P2, P3 sichern eintreffende Nachrichten, bis alle Marker empfangen

Beispiel zum Algorithmus von Chandy/Lamport

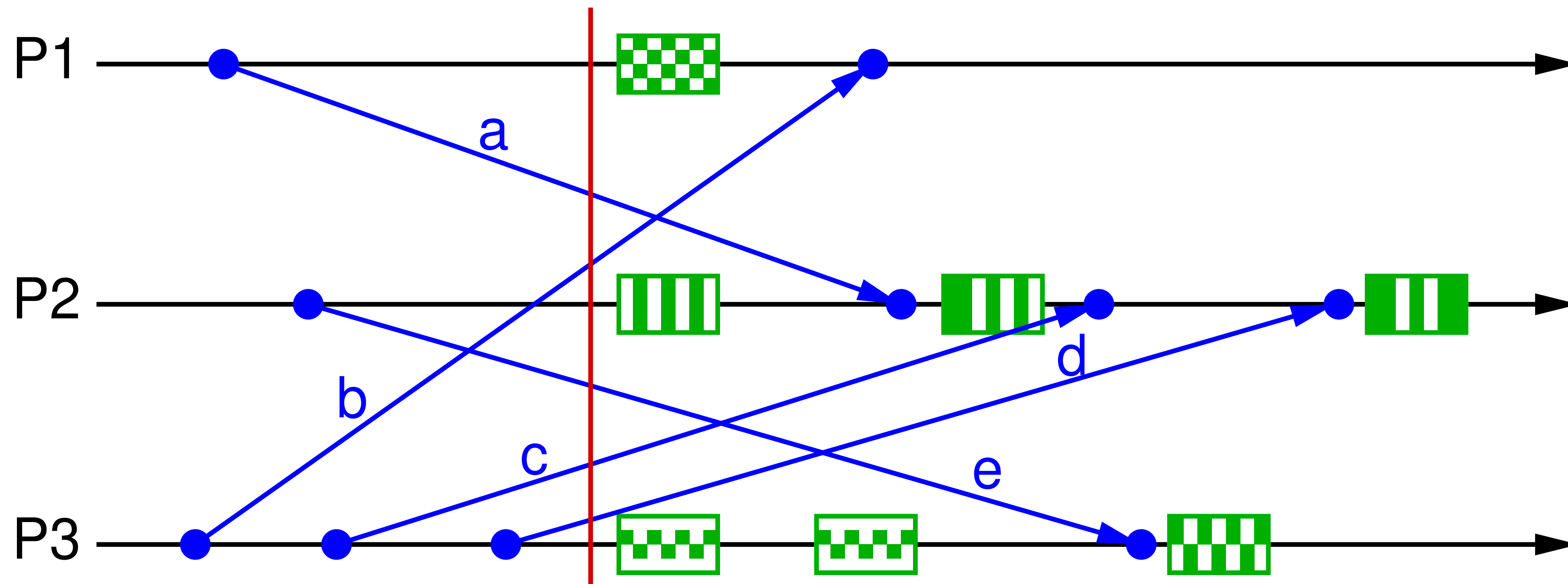


1. P1 initiiert Schnappschuß, sichert Zustand, sendet Marker
2. P3 empfängt Marker von P1, sichert Zustand, sendet Marker
3. P2 empfängt und verarbeitet a

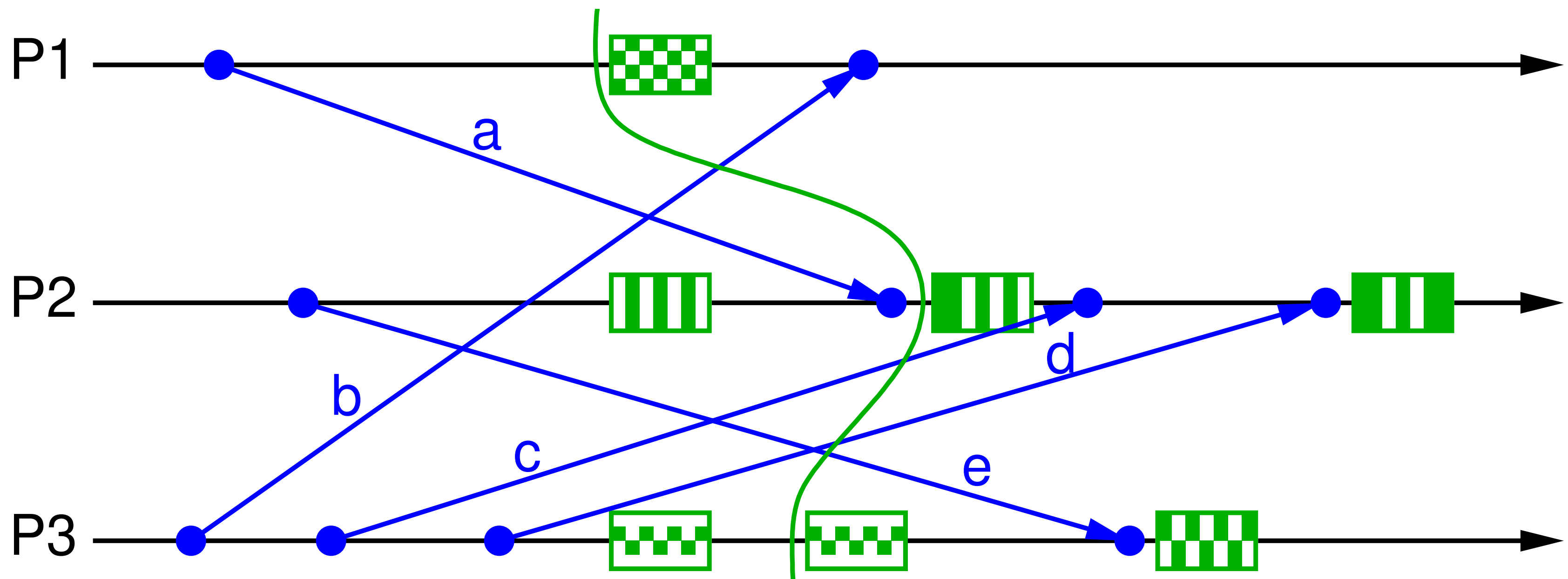
   P2 empfängt Marker von P1, sichert Zustand, sendet Marker
4. P1, P2, P3 sichern eintreffende Nachrichten, bis alle Marker empfangen



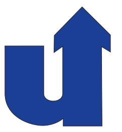
Ablauf im Beispiel und gewählter Schnitt



Ablauf im Beispiel und gewählter Schnitt



- Der Schnitt besteht aus den lokalen Zuständen von P1, P2, P3 und den Nachrichten b, c, d, e



Verteilte Systeme

SoSe 2018

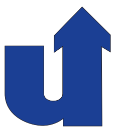
7 Koordination

Inhalt

- Wahl-Algorithmen
- Wechselseitiger Ausschluß
- Gruppen-Kommunikation (Multicast)
- Transaktionen

Literatur

- Tanenbaum, van Steen: Kap. 5.4-5.6
- Colouris, Dollimore, Kindberg: Kap. 11, 12
- Stallings: Kap 14.3



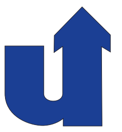
7.1 Wahl-Algorithmen

[Tanenbaum/Steen, 5.4]

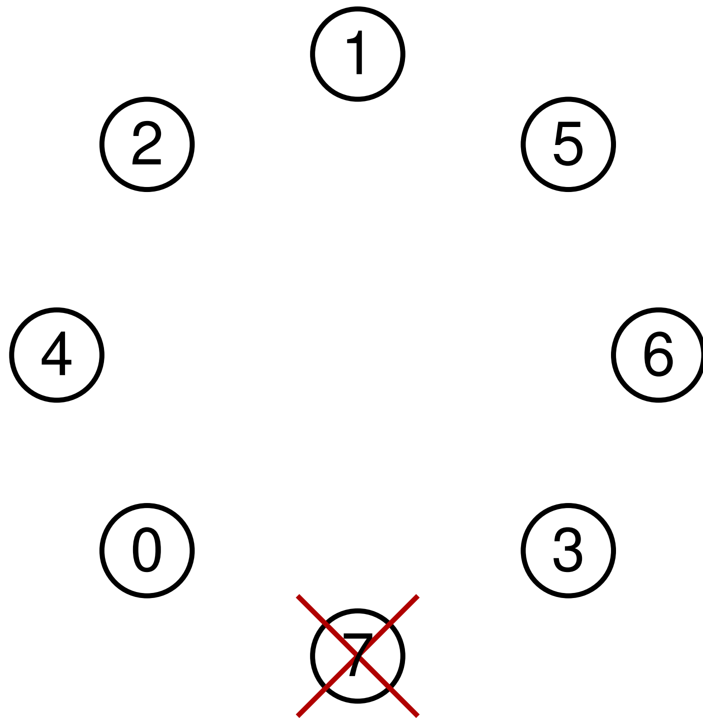
- In vielen verteilten Algorithmen muß **ein** beliebiger Prozeß eine ausgezeichnete Rolle übernehmen
  - z.B. zentraler Koordinator, Initiator, ...
- Frage: wie wählt man diesen Prozeß eindeutig?
  - Prozesse müssen unterscheidbar sein, z.B. über eindeutige ID
  - wähle dann z.B. den Prozess mit der höchsten ID
- Voraussetzungen / Anforderungen:
  - Wahl kann von mehreren Prozessen gleichzeitig initiiert werden
    - z.B. nach Ausfall bzw. Wiederherstellung eines Prozesses
  - nach der Wahl haben alle Prozesse dasselbe Ergebnis
  - jeder Prozess kennt die IDs aller anderen Prozesse, weiß aber nicht, ob diese laufen

Der Bully-Algorithmus

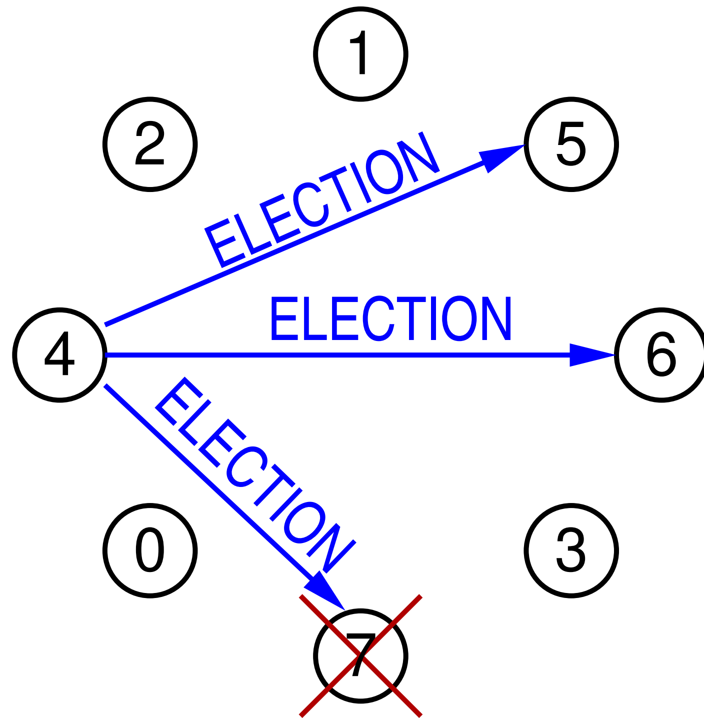
- Ein Prozeß P führt wie folgt eine Wahl durch:
  - P sendet eine ELECTION-Nachricht an alle Prozesse mit größerer ID
  - falls keiner der Prozesse reagiert, gewinnt P die Wahl
  - falls ein Prozeß antwortet: P verliert die Wahl
- Wenn ein Prozeß eine ELECTION-Nachricht empfängt:
  - (Nachricht kommt von einem Prozeß mit niedrigerer ID)
  - sende eine OK-Nachricht zurück
  - halte selbst eine Wahl ab
- Irgendwann bleibt nur noch ein Prozess übrig
  - dieser gewinnt die Wahl, sendet Ergebnis an alle anderen



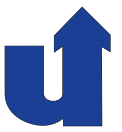
Bully-Algorithmus: Beispiel



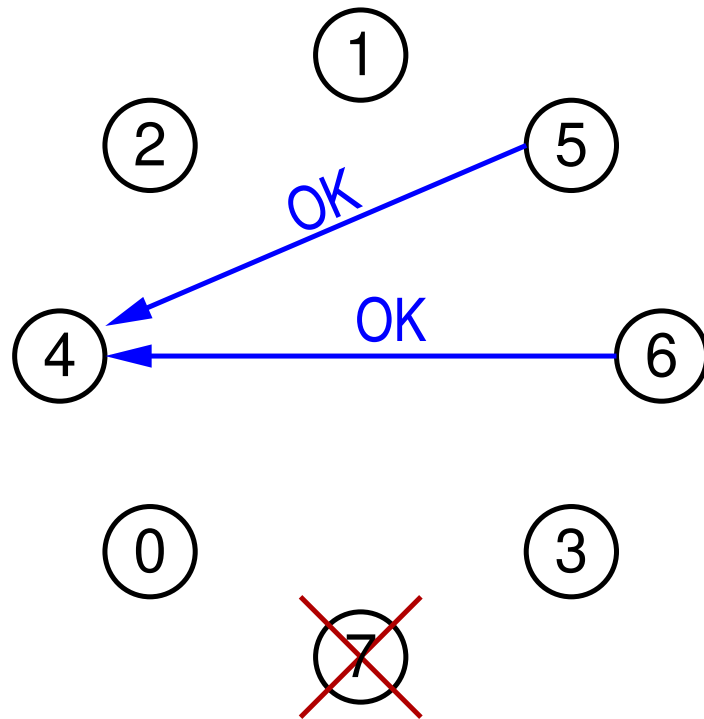
Bully-Algorithmus: Beispiel



Prozeß 4 hält eine Wahl ab



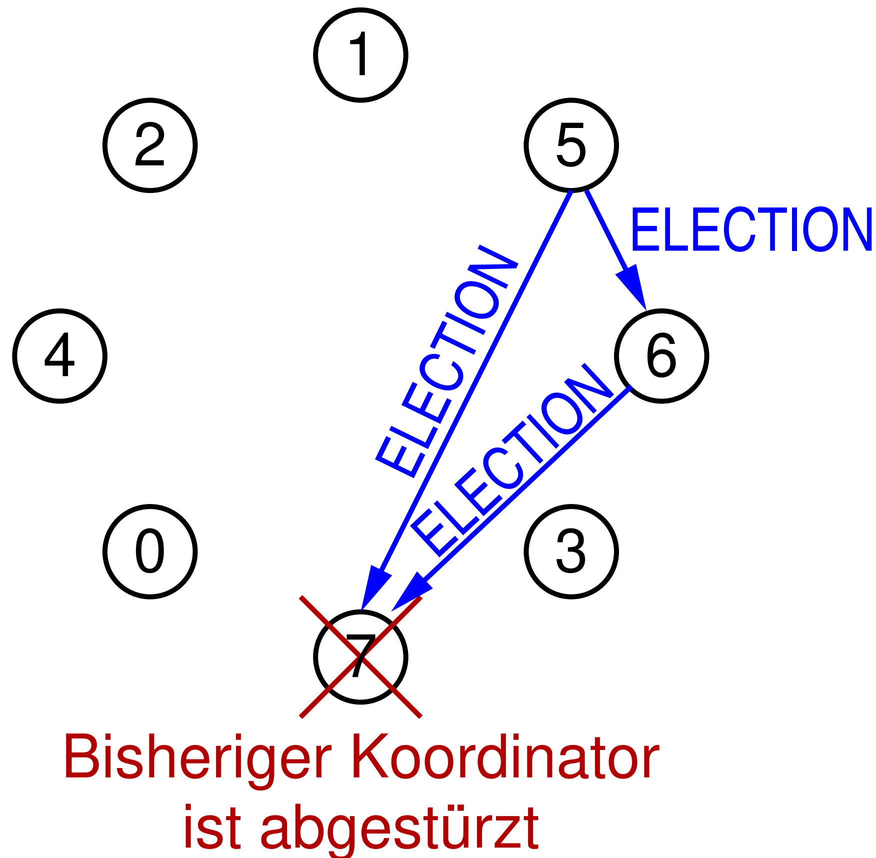
Bully-Algorithmus: Beispiel



Prozeß 4 hält eine Wahl ab

Prozesse 5 und 6 reagieren,
Prozeß 4 beendet seine Wahl

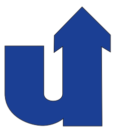
Bully-Algorithmus: Beispiel



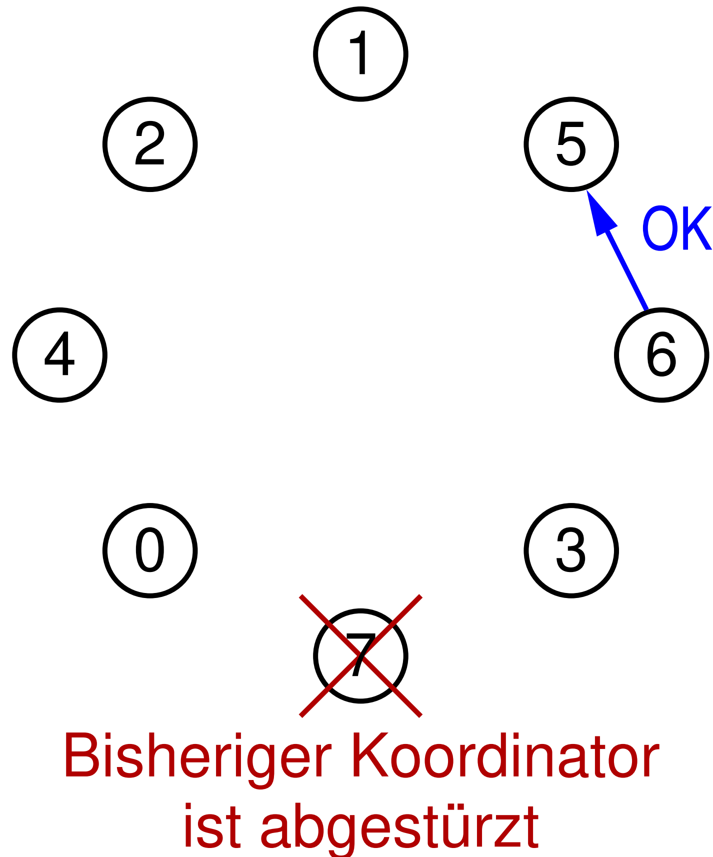
Prozeß 4 hält eine Wahl ab

Prozesse 5 und 6 reagieren,
Prozeß 4 beendet seine Wahl

Prozesse 5 und 6 halten
gleichzeitig je eine Wahl ab



Bully-Algorithmus: Beispiel



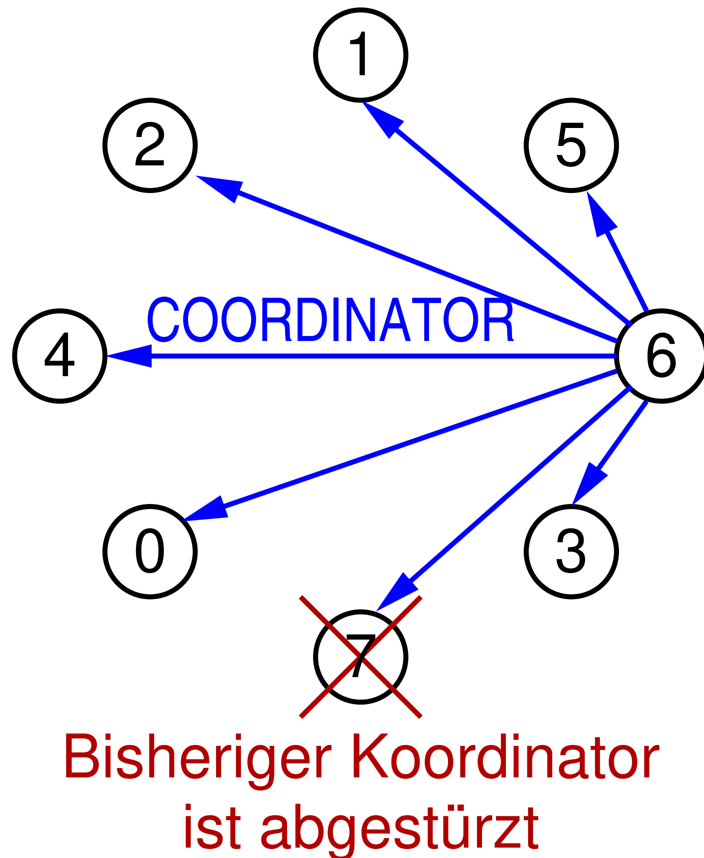
Prozeß 4 hält eine Wahl ab

Prozesse 5 und 6 reagieren,
Prozeß 4 beendet seine Wahl

Prozesse 5 und 6 halten
gleichzeitig je eine Wahl ab

Prozeß 6 antwortet an 5
Prozeß 5 beendet seine Wahl

Bully-Algorithmus: Beispiel



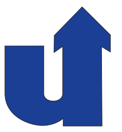
Prozeß 4 hält eine Wahl ab

Prozesse 5 und 6 reagieren,
Prozeß 4 beendet seine Wahl

Prozesse 5 und 6 halten
gleichzeitig je eine Wahl ab

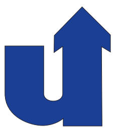
Prozeß 6 antwortet an 5
Prozeß 5 beendet seine Wahl

Niemand hat auf die Wahl von
Prozeß 6 reagiert, damit ge–
winnt er die Wahl und teilt das
allen anderen mit



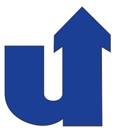
Ein Ring-Algorithmus

- Annahme: Prozesse formen einen logischen Ring, d.h. jeder Prozess kennt seine Nachfolger im Ring
- Nachrichten werden wie folgt entlang des Ringes gesendet:
  - Prozeß versucht, die Nachricht an seinen direkten Nachfolger zu senden
  - falls dieser nicht aktiv ist wird Nachricht zum nächsten Prozeß im Ring gesendet, usw.
- ELECTION-Nachrichten enthalten eine Liste von Prozeß-IDs

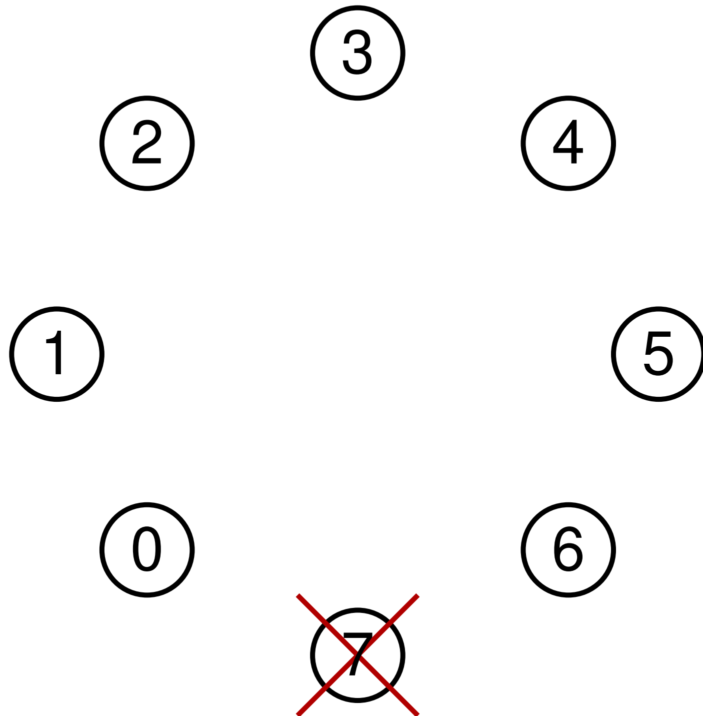


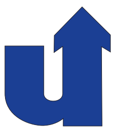
Ein Ring-Algorithmus ...

- Ein Prozeß, der die Wahl initiiert, sendet ELECTION-Nachricht mit eigener ID entlang des Rings
- Bei Empfang einer ELECTION-Nachricht durch einen Prozeß:
  - falls eigene ID nicht in der Liste der IDs ist:
    - hänge eigene ID an die Liste an
    - sende Nachricht weiter entlang des Rings
  - sonst (Nachricht kam wieder zum Initiator zurück):
    - bestimme höchste ID in der Liste
    - sende diese ID in COORDINATOR-Nachricht entlang des Rings

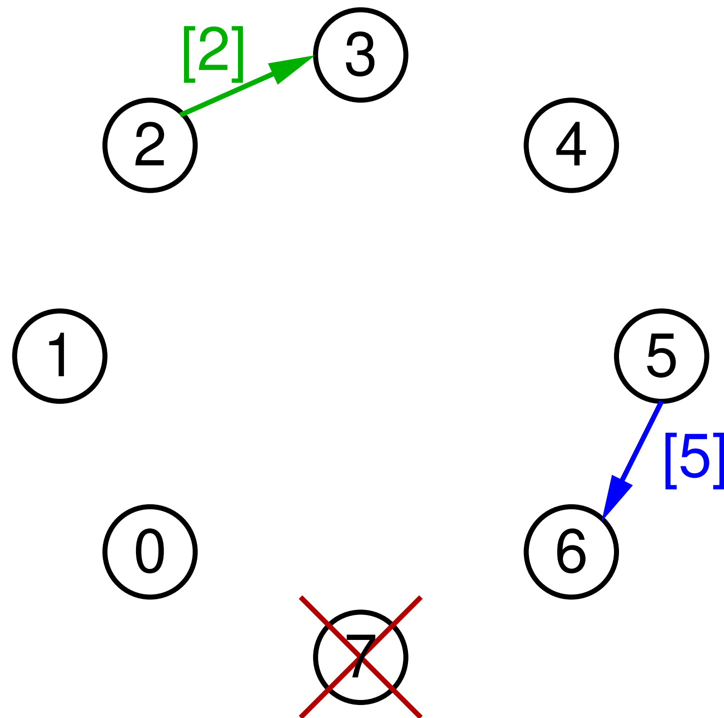


Ring-Algorithmus: Beispiel

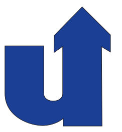




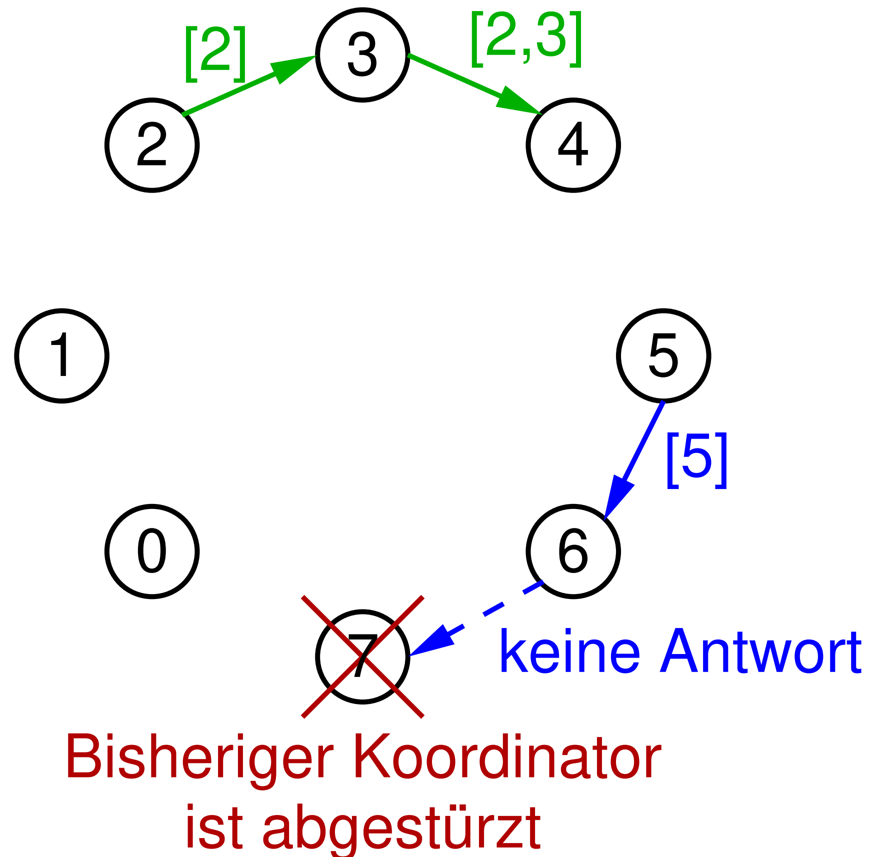
Ring-Algorithmus: Beispiel



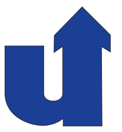
Prozesse 2 und 5 starten gleichzeitig eine Wahl



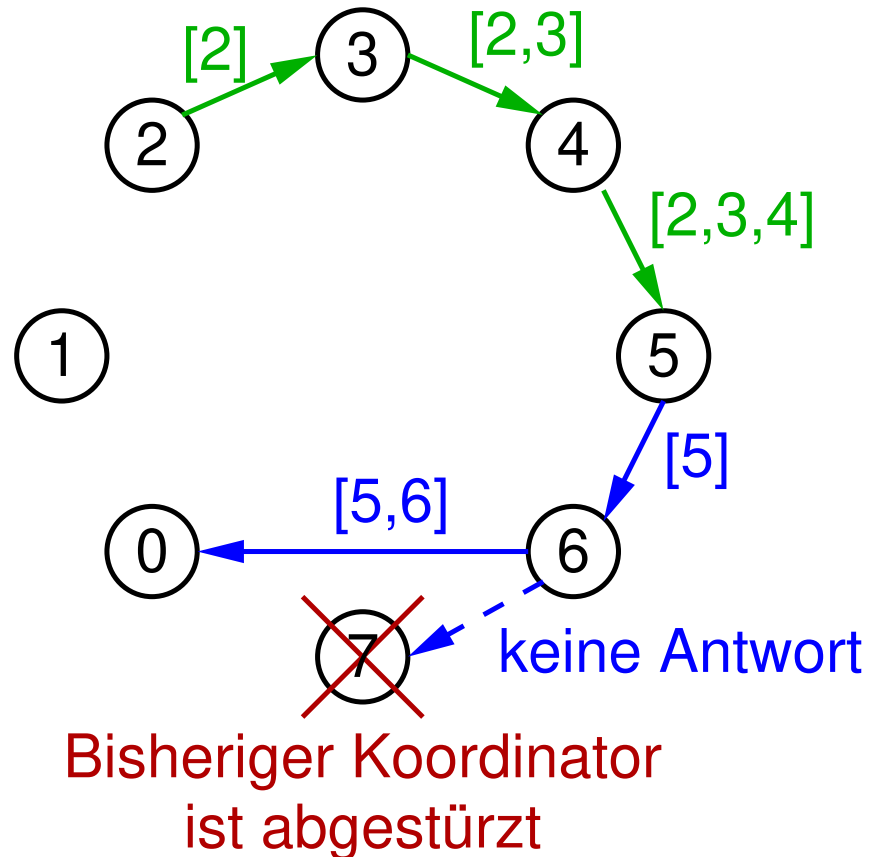
Ring-Algorithmus: Beispiel



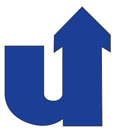
Prozesse 2 und 5 starten gleichzeitig eine Wahl



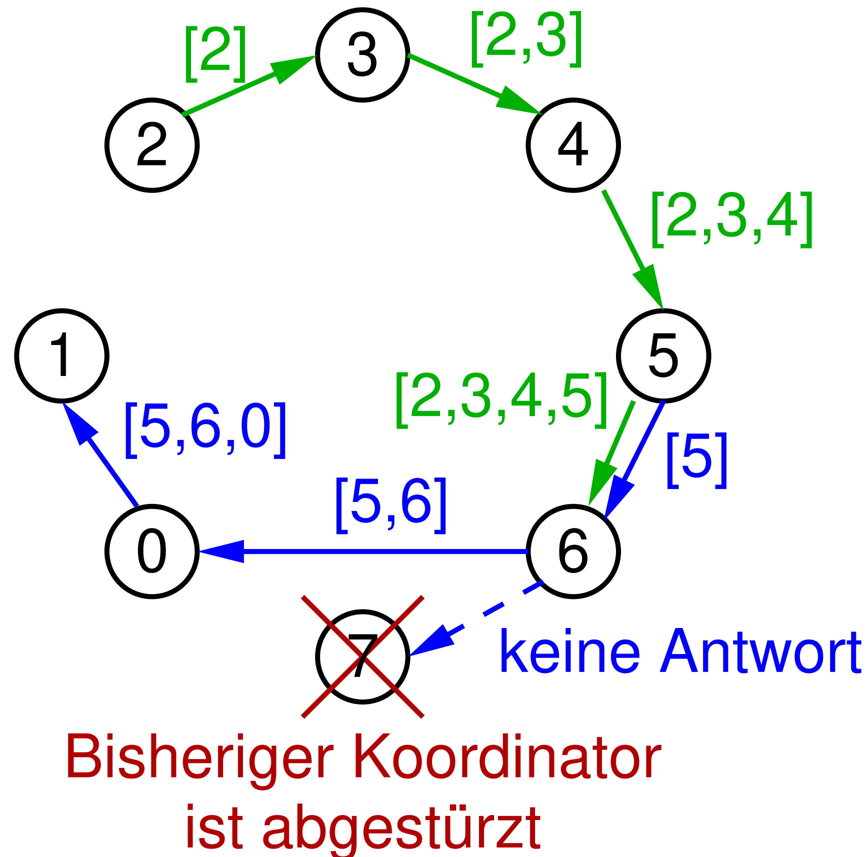
Ring-Algorithmus: Beispiel



Prozesse 2 und 5 starten gleichzeitig eine Wahl



Ring-Algorithmus: Beispiel

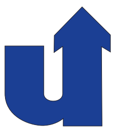


Prozesse 2 und 5 starten gleichzeitig eine Wahl

Irgendwann erhalten beide ihre ELECTION–Nachrichten wieder zurück und versenden eine COORDINATOR–Nach– richt (mit demselben Inhalt!)

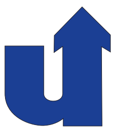
7.2 Wechselseitiger Ausschluß

- Hier vorwiegend: Nutzung / Zuteilung exklusiver Ressourcen
- Anforderungen:
  - Sicherheit: zu jeder Zeit kann nur ein Prozeß die Ressource nutzen
  - Lebendigkeit: jeder Prozeß, der die Ressource anfragt, bekommt sie irgendwann auch
  - Fairness: Zugang zu Ressourcen in „FIFO"-Reihenfolge
- Lösungsansätze:
  - zentralisierter Server
  - verteilter Algorithmus mit Lamport-Uhr
  - Token-Ring-Algorithmus



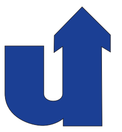
Zentralisierter Server

- Ein ausgezeichneter Koordinator-Prozeß verwaltet die Ressource und eine Warteschlange für wartende Prozesse
  - Bestimmung z.B. über Wahl-Algorithmen
- Anforderung einer Ressource durch Nachricht an Koordinator
  - falls Ressource frei: Koordinator antwortet mit OK
  - sonst: Koordinator antwortet nicht
    - anfragender Prozeß ist blockiert (wartet auf Antwort)
- Freigabe der Ressource durch Nachricht an Koordinator
  - falls Prozesse warten: Koordinator sendet einem davon ein OK
- Problem: Prozesse können Koordinator-Ausfall nicht erkennen
  - dazu ggf. negative Antworten und Polling



Ein verteilter Algorithmus (Ricart / Agrawala)

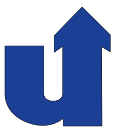
- Idee: ein Prozeß, der eine Ressource haben will, fragt alle anderen Prozesse nach ihrem OK
  - ein Prozeß erteilt sein OK, wenn er
    - die Ressource nicht will, oder
    - er die Ressource will, der andere Prozeß sie aber „früher“ angefordert hat
- Benötigt **vollständige** Ordnung der Anforderungs-Ereignisse
  - Ordnung muß konsistent mit Kausalität sein
  - realisierbar z.B. über Zeitstempel (Lamport-Zeit, Prozeß-ID) mit lexikographischer Ordnung
    - ergibt im Beispiel von Folie 207 die Ereignis-Reihenfolge: $b, c, a, e, g, d, j, f, l, h, i, k$



Ein verteilter Algorithmus (Ricart / Agrawala) ...

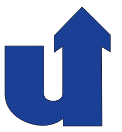
- Um eine Ressource anzufordern, sendet ein Prozeß folgende Nachricht an alle anderen Prozesse:
  - ID der Ressource
  - Zeitstempel T der Anforderung
    - Paar: (aktuelle Lamport-Zeit, eigene Prozeß-ID)

  (Die Nachricht muß zuverlässig zugestellt werden)
- Der Prozeß wartet dann, bis er von jedem anderen Prozeß eine OK-Nachricht erhält
- Danach kann er die Ressource (exklusiv) benutzen



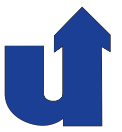
Ein verteilter Algorithmus (Ricart / Agrawala) ...

- Jeder Prozeß reagiert auf Anfrage-Nachrichten wie folgt:
  - Ressource nicht selbst genutzt und auch nicht angefordert:
    - sende OK-Nachricht zurück
  - Ressource selbst genutzt:
    - sende keine Antwort
    - stelle die Anforderung in eine Warteschlange
  - Ressource nicht genutzt, aber angefordert:
    - falls T(eingehende Nachricht) $<$ T(eigene Anforderung):
      - sende OK-Nachricht zurück
    - sonst:
      - sende keine Antwort
      - stelle die Anforderung in eine Warteschlange

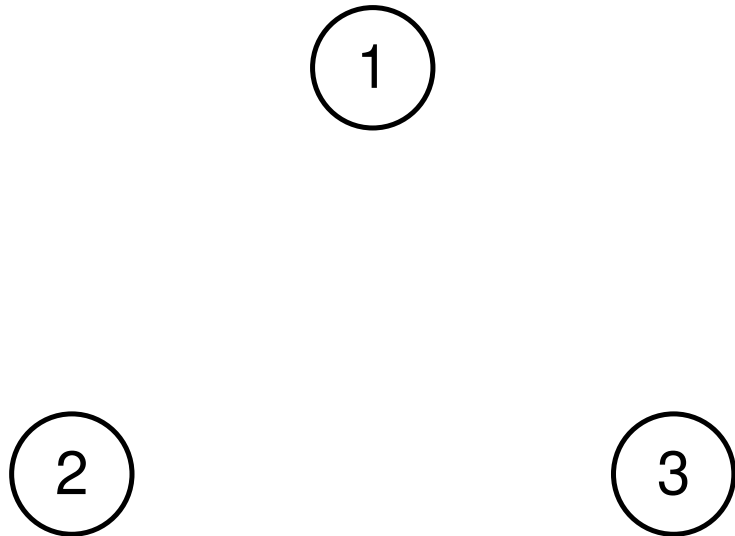


Ein verteilter Algorithmus (Ricart / Agrawala) ...

- Wenn ein Prozeß die Ressource freigibt:
  - sende OK-Nachricht an **alle** Prozesse der Warteschlange
  - lösche die Warteschlange



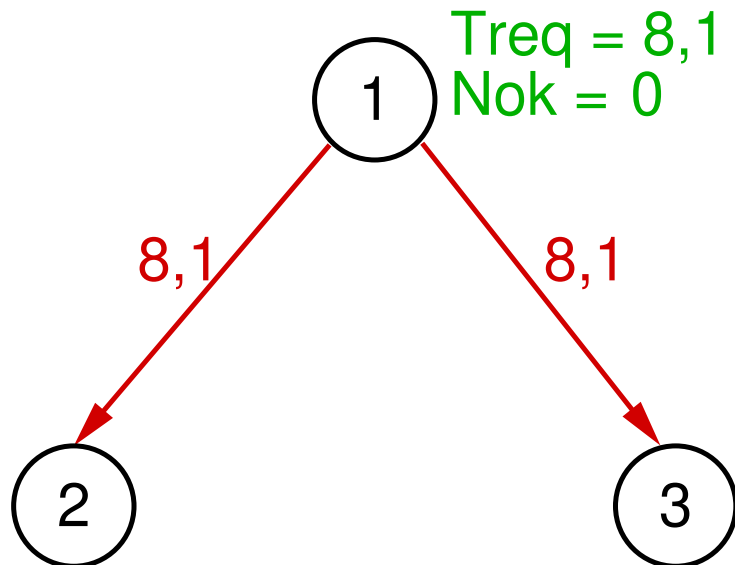
Beispiel zum Algorithmus von Ricart / Agrawala



P1 und P3 wollen beide
die Ressource

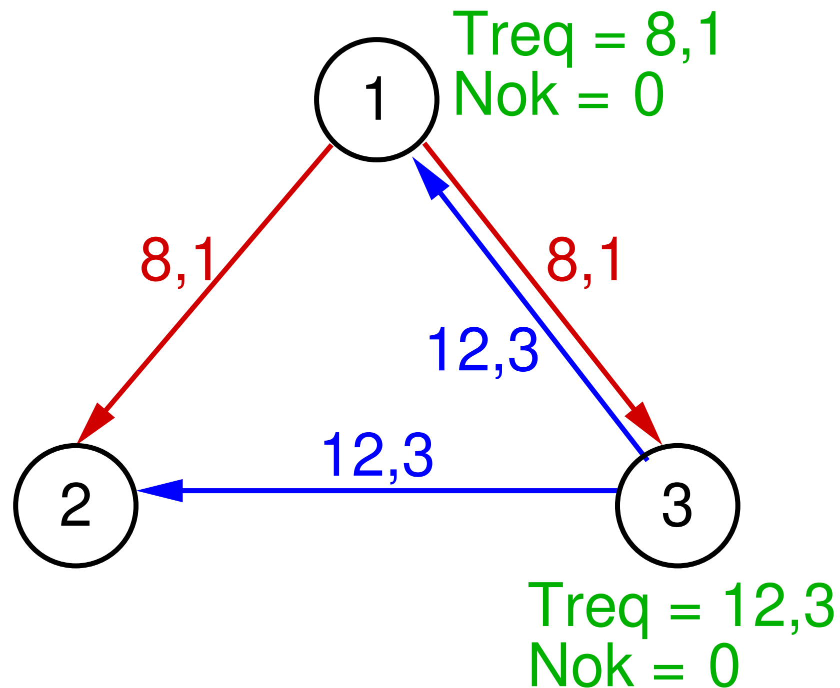
Beispiel zum Algorithmus von Ricart / Agrawala

1. P1 sendet Anfrage an alle



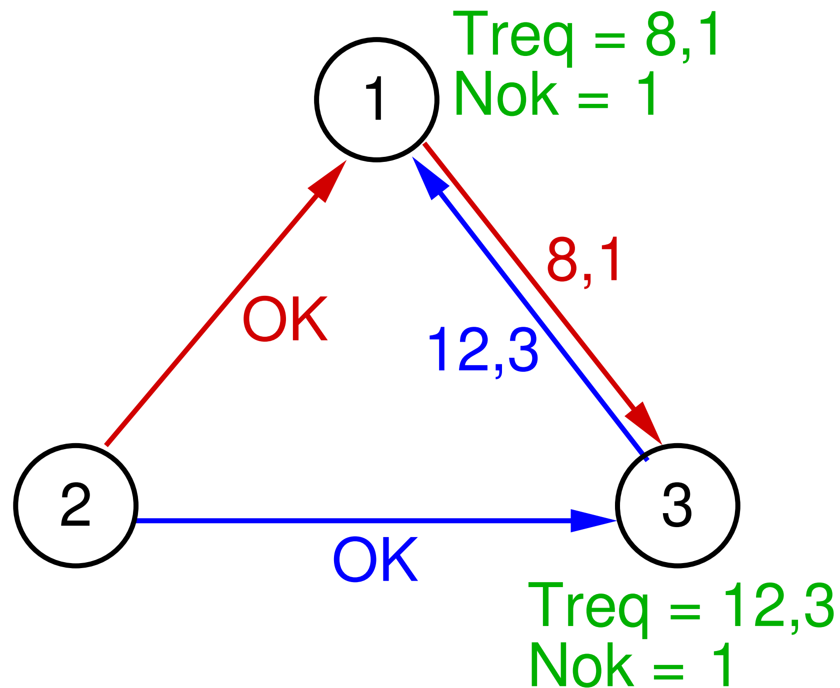
P1 und P3 wollen beide die Ressource

Beispiel zum Algorithmus von Ricart / Agrawala



1. P1 sendet Anfrage an alle
2. P3 sendet Anfrage an alle

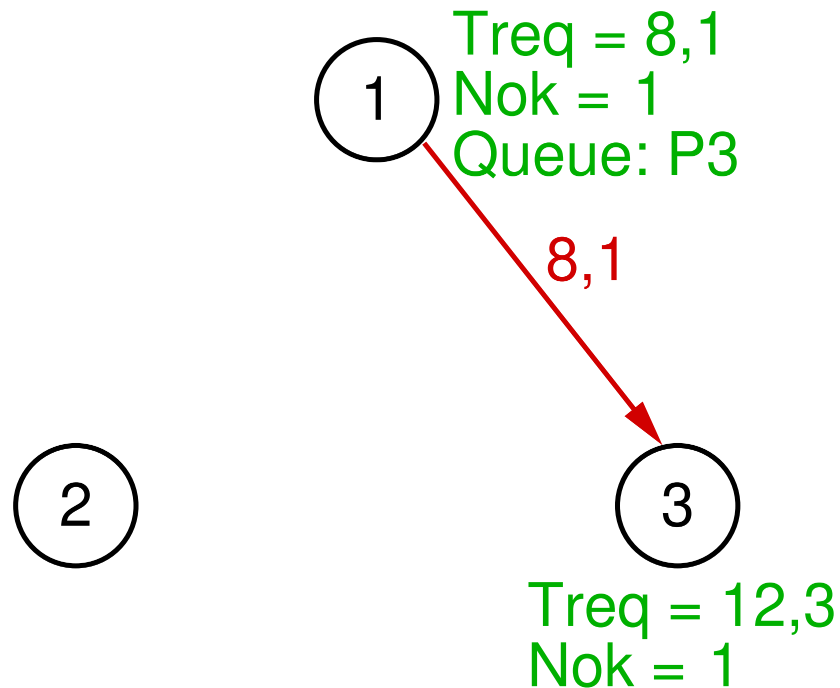
Beispiel zum Algorithmus von Ricart / Agrawala



P1 und P3 wollen beide die Ressource

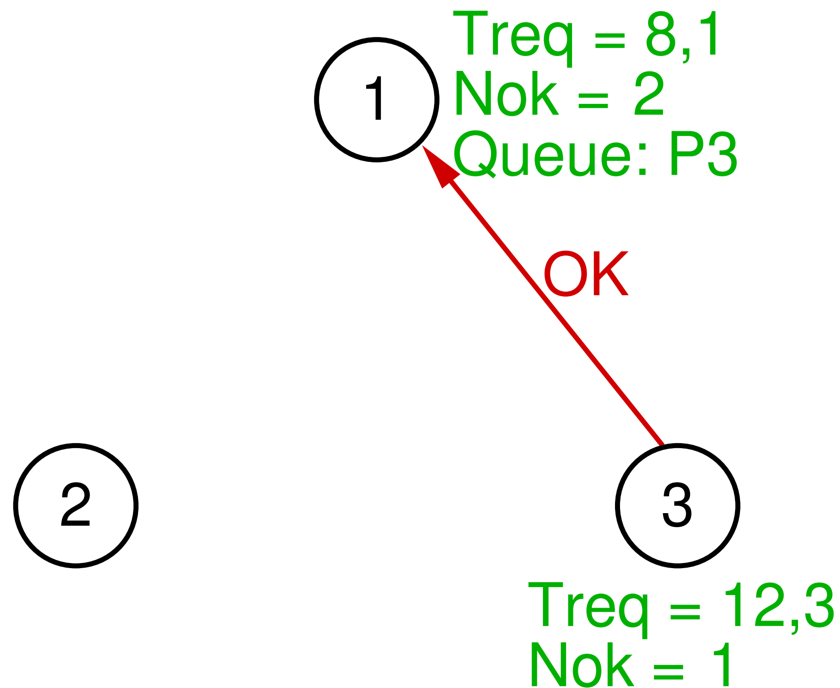
1. P1 sendet Anfrage an alle
2. P3 sendet Anfrage an alle
3. P2 sendet OK an P1 und P3, da er die Ressource nicht will

Beispiel zum Algorithmus von Ricart / Agrawala



1. P1 sendet Anfrage an alle
2. P3 sendet Anfrage an alle
3. P2 sendet OK an P1 und P3, da er die Ressource nicht will
4. P1 sendet kein OK an P3, da (12,3) > (8,1). P1 stellt P3 in Warteschlange

Beispiel zum Algorithmus von Ricart / Agrawala



1. P1 sendet Anfrage an alle
2. P3 sendet Anfrage an alle
3. P2 sendet OK an P1 und P3, da er die Ressource nicht will
4. P1 sendet kein OK an P3, da (12,3) > (8,1). P1 stellt P3 in Warteschlange
5. P3 sendet OK an P1, da (8,1) < (12,3)

Beispiel zum Algorithmus von Ricart / Agrawala

P1 besitzt Ressource

1: Treq = 8,1
Nok = 2
Queue: P3

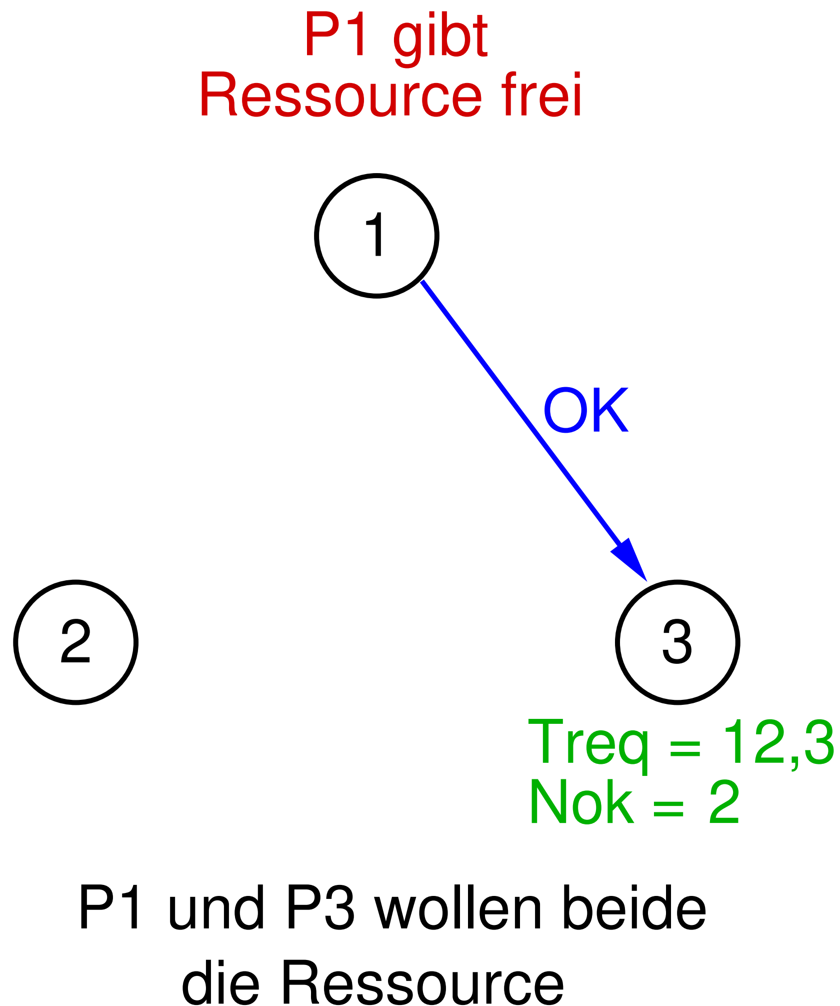
2

3: Treq = 12,3
Nok = 1

P1 und P3 wollen beide die Ressource

1. P1 sendet Anfrage an alle
2. P3 sendet Anfrage an alle
3. P2 sendet OK an P1 und P3, da er die Ressource nicht will
4. P1 sendet kein OK an P3, da (12,3) > (8,1). P1 stellt P3 in Warteschlange
5. P3 sendet OK an P1, da (8,1) < (12,3)
6. P1 hat alle OKs erhalten und nutzt die Ressource

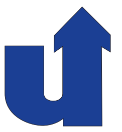
Beispiel zum Algorithmus von Ricart / Agrawala



1. P1 sendet Anfrage an alle
2. P3 sendet Anfrage an alle
3. P2 sendet OK an P1 und P3, da er die Ressource nicht will
4. P1 sendet kein OK an P3, da (12,3) > (8,1). P1 stellt P3 in Warteschlange
5. P3 sendet OK an P1, da (8,1) < (12,3)
6. P1 hat alle OKs erhalten und nutzt die Ressource
7. P1 gibt Ressource frei und sendet sein OK an P3

Ein Token-Ring-Algorithmus

- Die Prozesse formen einen logischen Ring
- Im Ring kreist ein **Token**
  - Berechtigung zur (exklusiven) Nutzung der Ressource
  - Token wird initial von einem der Prozesse erzeugt
- Bei Ankunft des Tokens: Prozeß prüft, ob er die Ressource will
  - falls ja:
    - nutze die Ressource
    - nach Freigabe der Ressource:
      - gib Token an Nachfolger im Ring weiter
  - falls nein:
    - gib Token sofort an Nachfolger im Ring weiter



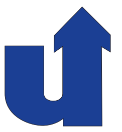
Vergleich der Algorithmen

- Zentralisierter Server:
  - Server ist *Single Point of Failure* und ggf. Leistungsengpaß
  - Clients können (ohne Zusatzmaßnahmen) nicht zwischen Serverausfall und belegter Ressource unterscheiden
  - nur wenig Kommunikation nötig
- Verteilter Algorithmus:
  - Ausfall **jedes** Knotens ist problematisch
  - jeder Knoten kann zum Leistungsengpaß werden
  - hoher Kommunikationsaufwand
  - Nachweis: verteilter, symmetrischer Algorithmus ist möglich

Vergleich der Algorithmen ...

- Token-Ring-Algorithmus:
  - Problem: Verlust des Tokens (Erkennung, Neu-Erzeugung)
  - Ausfall von Knoten ist problematisch
  - Kommunikation, auch wenn Ressource nicht genutzt wird

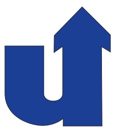
Algorithmus	Nachrichten pro Zuteilung	Verzögerung vor Zuteilung	Probleme
Zentralisiert	3	2	Ausfall des Servers
Verteilt	$2(n-1)$	$2(n-1)$	Ausfall eines beliebigen Prozesses
Token-Ring	$1 \ldots \infty$	$0 \ldots n-1$	verlorenes Token, Prozeß-Ausfall



[Coulouris, 4.5, 11.4]

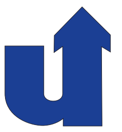
7.3 Gruppen-Kommunikation (Multicast)

- In verteilten Systemen ist oft auch Kommunikation mit einer **Gruppe** von Prozessen (**Multicast**) wichtig, z.B. für:
  - Fehlertoleranz basierend auf replizierten Diensten
    - Dienst realisiert durch Gruppe von Servern
    - alle Server empfangen und bearbeiten die Anfragen
  - Finden von Diensten (insbes. *Discovery*/Namens-Dienste)
    - Multicast ist ein möglicher Ansatz dafür
  - Bessere Leistung durch replizierte Daten
    - Änderungen müssen an alle Kopien gesandt werden
  - Versenden von Ereignisbenachrichtigungen
    - alle Abonnenten erhalten das Ereignis



Fragestellungen / Probleme

- Adressierung der Empfänger
  - explizite Auflistung aller Empfänger
  - Adressierung einer Prozeß-Gruppe
    - statische / dynamische Gruppen
- Zuverlässigkeit
  - angemessene Garantien, daß Nachrichten bei den Empfängern ankommen
- Reihenfolge
  - angemessene Garantien bezüglich der Reihenfolge, in der Multicast-Nachrichten bei den verschiedenen Empfängern eintreffen

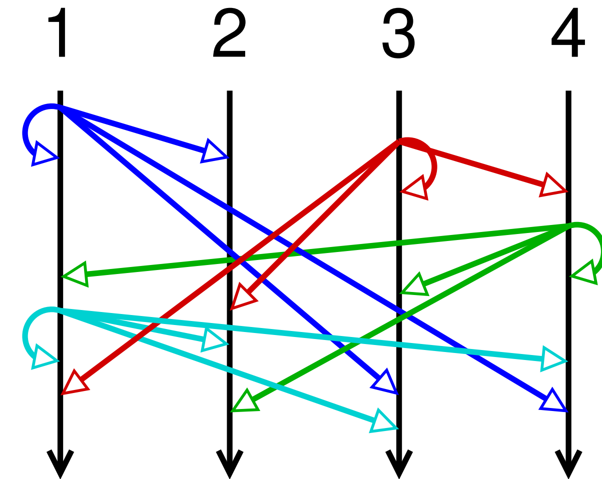


Zuverlässigkeit

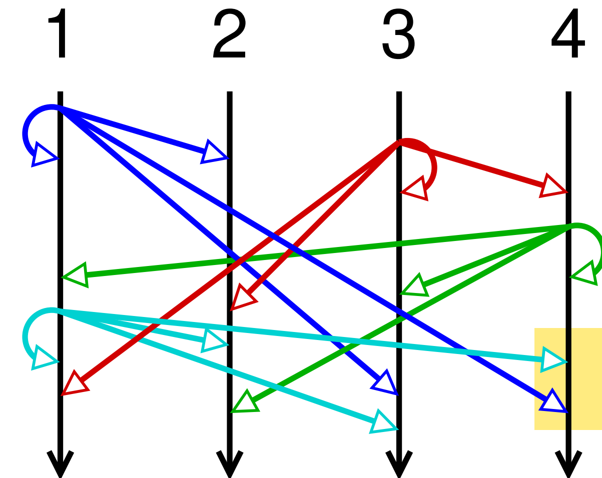
- **Unzuverlässiger Multicast**:
  - es kann vorkommen, daß einige Prozesse die Nachricht nicht erhalten (z.B. bei Paketverlust)
- **Zuverlässiger Multicast**:
  - abgesehen von Netzwerk- und Prozeßausfällen wird die Nachricht an alle Prozesse der Gruppe ausgeliefert
- **Atomarer Multicast**:
  - die Nachricht wird (unter allen Umständen) entweder von **allen** Prozessen der Gruppe empfangen oder von **keinem**
  - notwendig, wenn alle Prozesse der Gruppe konsistent gehalten werden müssen (z.B. Operationen auf replizierten Daten)

Reihenfolge

- **Unsortiert**
  - Empfangsreihenfolge ist unbestimmt und kann in verschiedenen Prozessen unterschiedlich sein

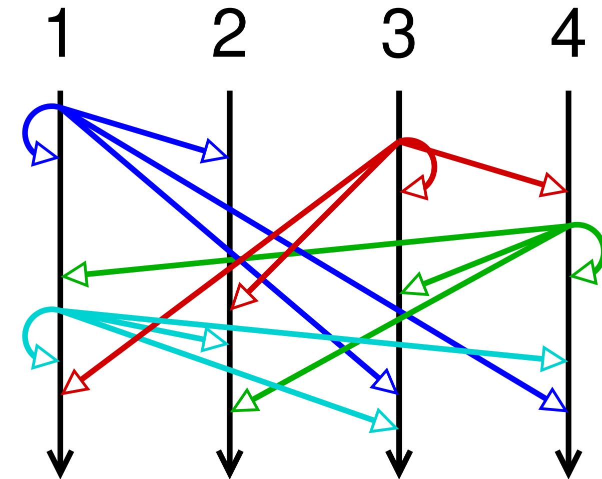


- **FIFO-Sortierung**
  - Nachrichten **desselben** Senders werden von allen Prozessen in FIFO-Reihenfolge empfangen
  - d.h. Einführung von **senderlokalen** Sequenznummern

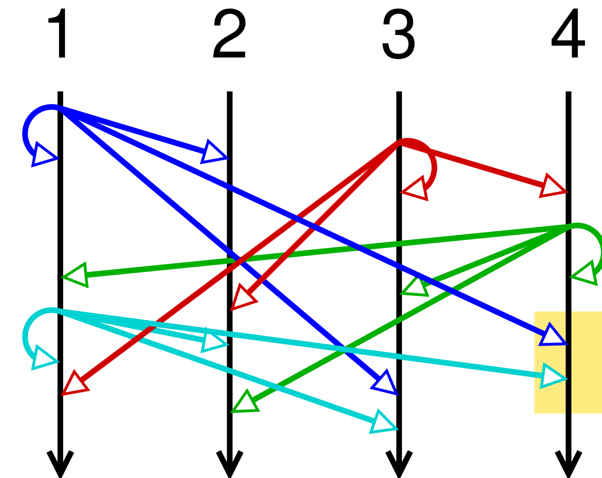


Reihenfolge

- **Unsortiert**
  - Empfangsreihenfolge ist unbestimmt und kann in verschiedenen Prozessen unterschiedlich sein

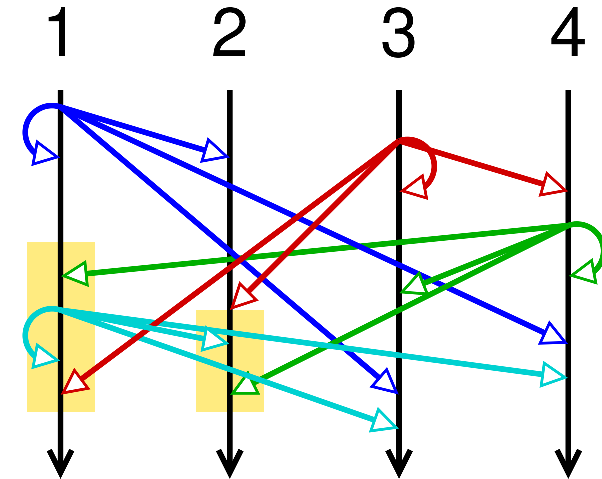


- **FIFO-Sortierung**
  - Nachrichten **desselben** Senders werden von allen Prozessen in FIFO-Reihenfolge empfangen
  - d.h. Einführung von **senderlokalen** Sequenznummern

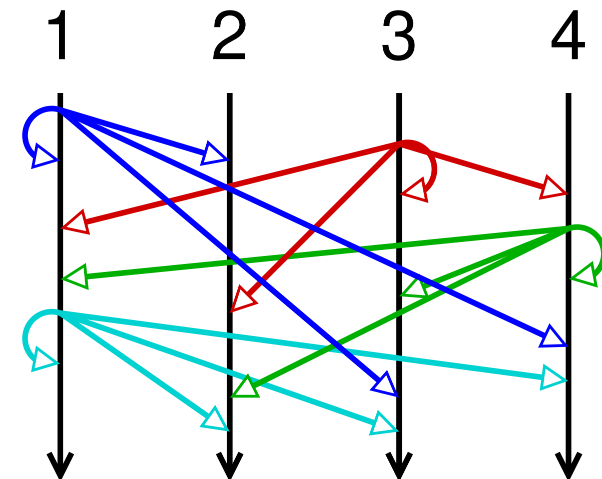


Reihenfolge ...

- **Kausale Sortierung**
  - wenn Nachricht m' kausal von m abhängen kann, dann empfangen alle Prozesse m vor m'
  - d.h. Einführung von Vektorzeit-stempeln

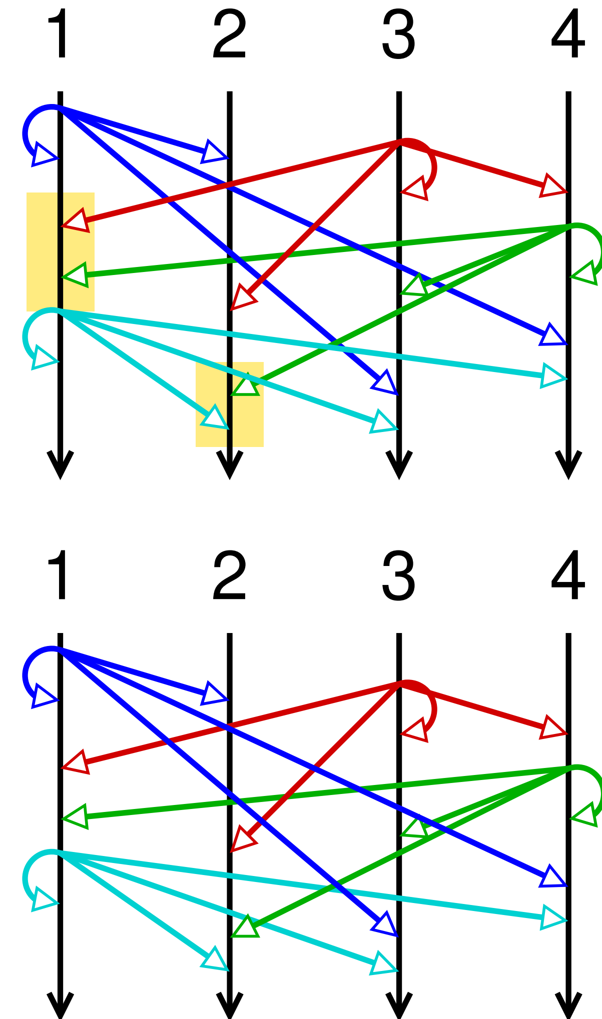


- **Vollständige Sortierung**
  - **alle** Nachrichten werden von allen Prozessen in derselben Reihen-folge empfangen
  - d.h. Einführung von **globalen** Sequenznummern



Reihenfolge ...

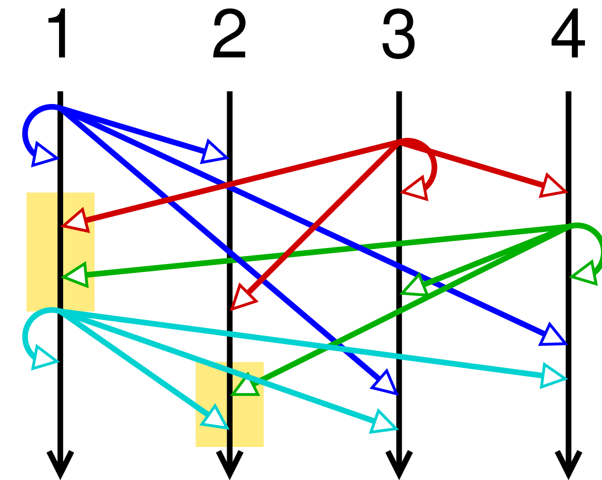
- **Kausale Sortierung**
  - wenn Nachricht m' kausal von m abhängen kann, dann empfangen alle Prozesse m vor m'
  - d.h. Einführung von Vektorzeitstempeln



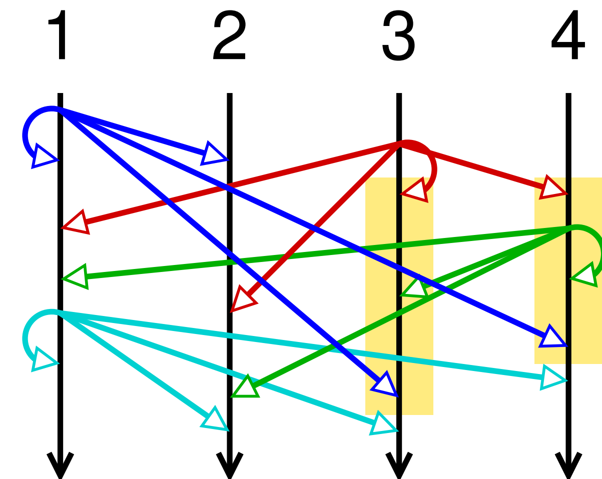
- **Vollständige Sortierung**
  - **alle** Nachrichten werden von allen Prozessen in derselben Reihenfolge empfangen
  - d.h. Einführung von **globalen** Sequenznummern

Reihenfolge ...

- **Kausale Sortierung**
  - wenn Nachricht m' kausal von m abhängen kann, dann empfangen alle Prozesse m vor m'
  - d.h. Einführung von Vektorzeit-stempeln

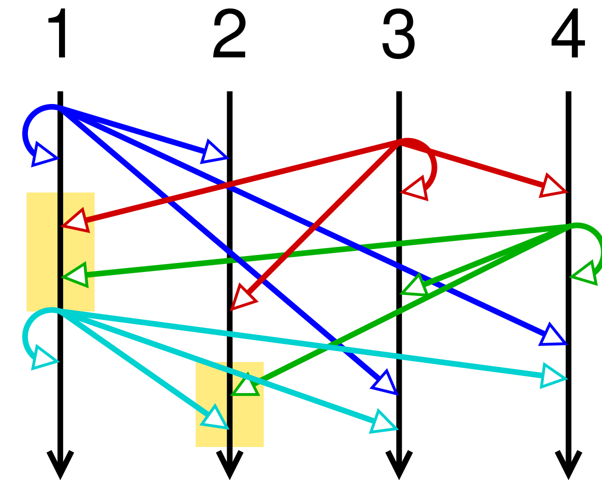


- **Vollständige Sortierung**
  - **alle** Nachrichten werden von allen Prozessen in derselben Reihen-folge empfangen
  - d.h. Einführung von **globalen** Sequenznummern

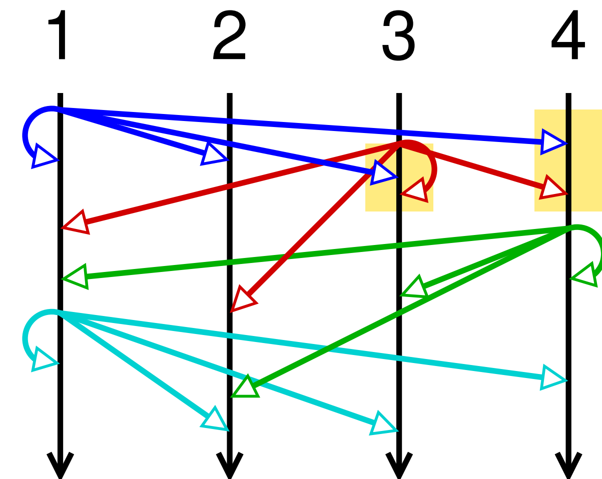


Reihenfolge ...

- **Kausale Sortierung**
  - wenn Nachricht m' kausal von m abhängen kann, dann empfangen alle Prozesse m vor m'
  - d.h. Einführung von Vektorzeitstempeln

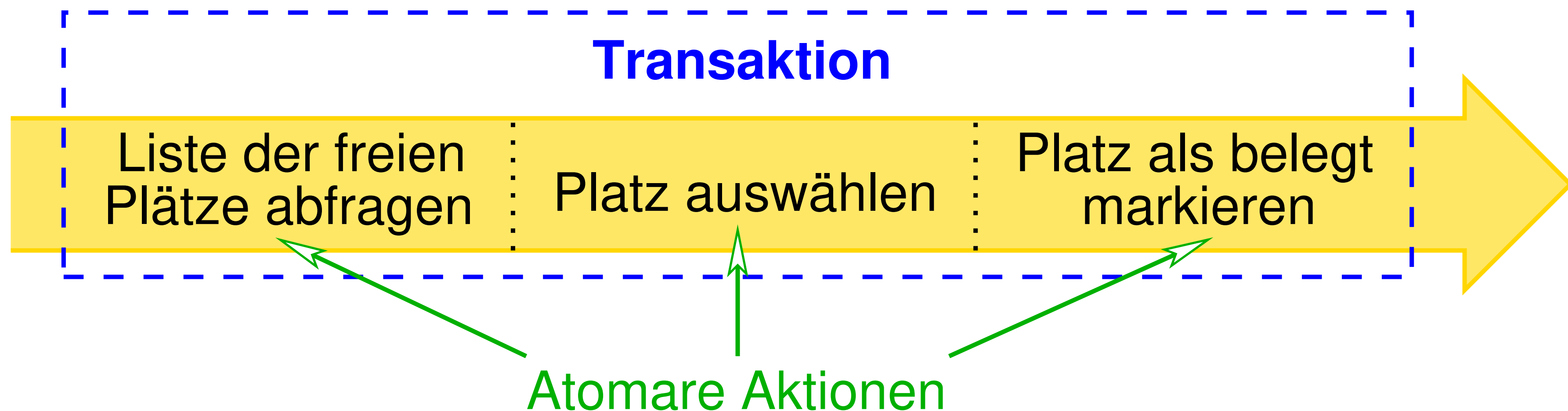


- **Vollständige Sortierung**
  - **alle** Nachrichten werden von allen Prozessen in derselben Reihenfolge empfangen
  - d.h. Einführung von **globalen** Sequenznummern

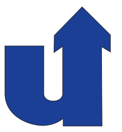


7.4 Transaktionen

- Zusammenfassung einer Folge von atomaren Aktionen zu einer Einheit
  - atomare Aktionen: Lesen, Ändern, Schreiben von Daten
- Beispiel: Sitzplatz-Reservierung



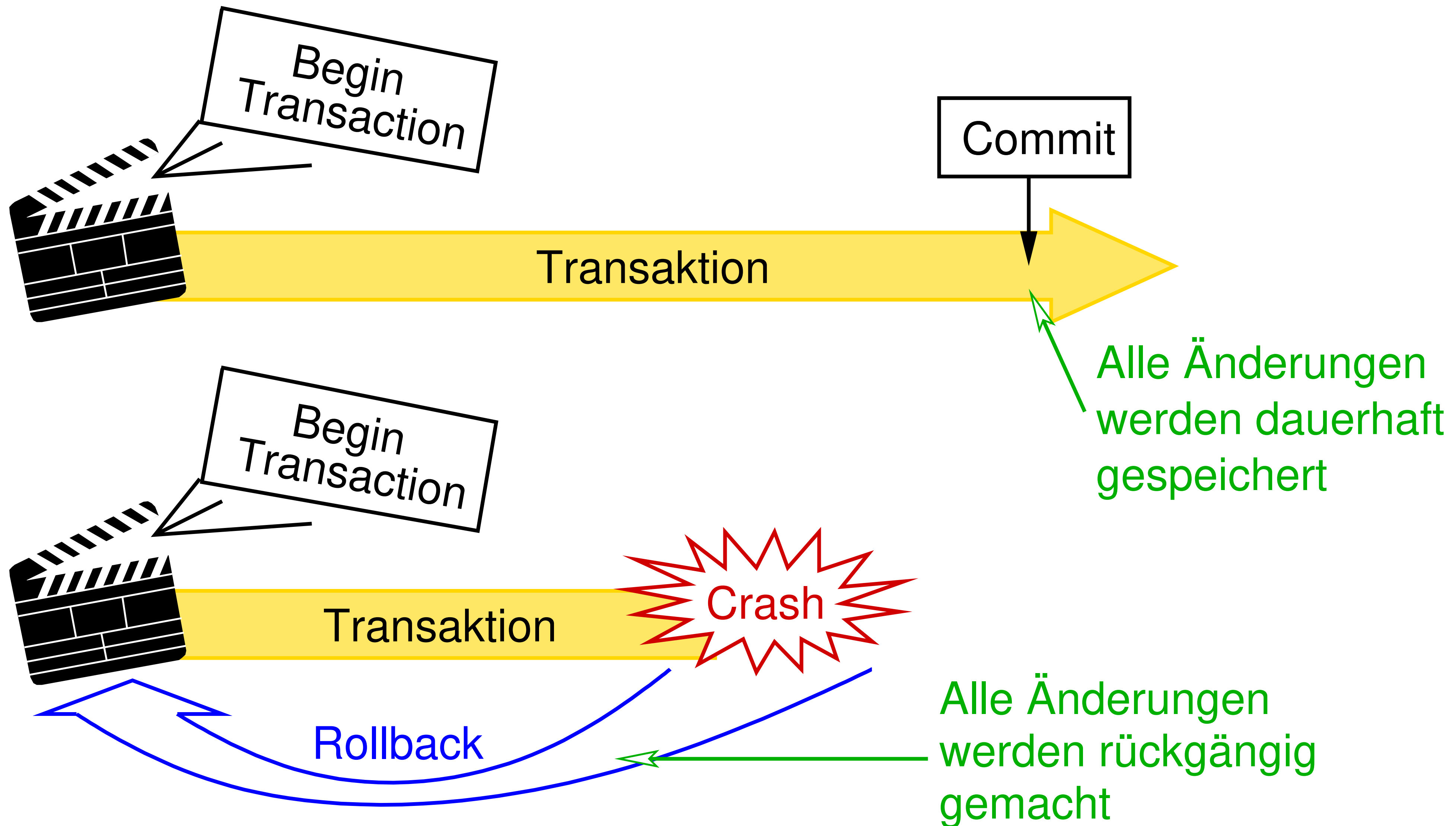
- Einsatz nicht nur in Datenbanksystemen



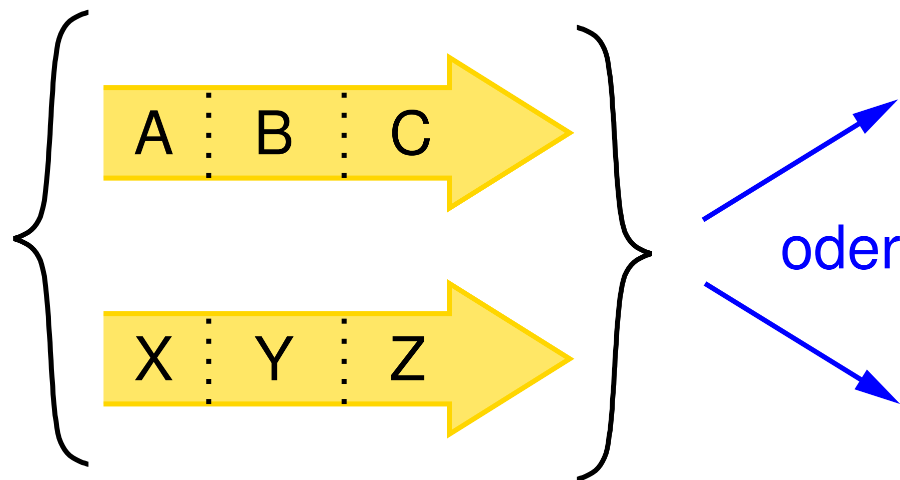
Eigenschaften von Transaktionen: ACID

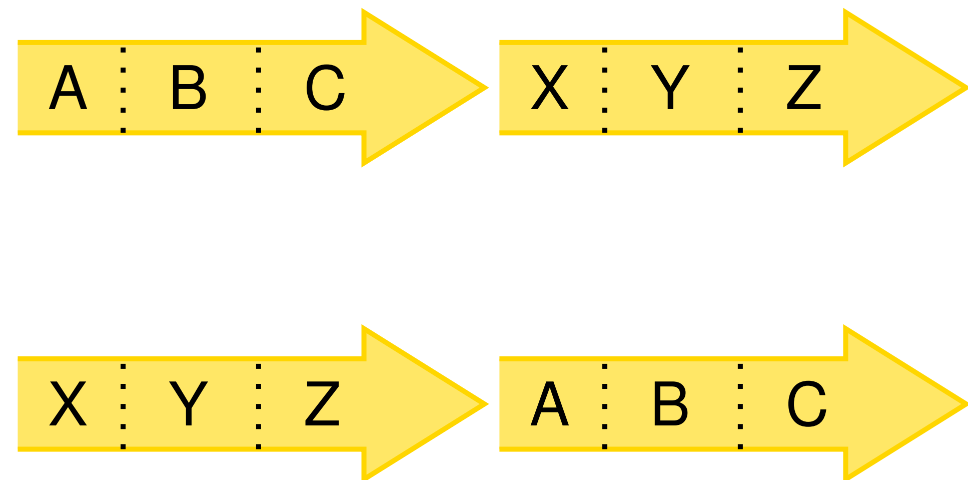
- ***A**tomicity* (Atomarität)
  - Alles-oder-Nichts-Prinzip: entweder werden alle atomaren Aktionen (korrekt) ausgeführt oder gar keine
- ***C**onsistency* (Konsistenz)
  - konsistenter Zustand wird durch Transaktion wieder in konsistenten Zustand überführt
- ***I**solation*
  - nebenläufige Transaktionen beeinflussen sich nicht; das Ergebnis ist dasselbe wie bei sequentieller Ausführung
- ***D**urability* (Dauerhaftigkeit)
  - am (erfolgreichen) Ende der Transaktion sind alle Änderungen dauerhaft gespeichert

Atomarität

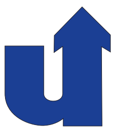


Isolation





➥ Ergebnis der nebenläufigen Transaktionen entspricht einer der beiden Serialisierungen



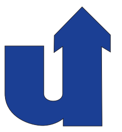
Isolationsebenen

- Vollständige Isolation von (Datenbank-)Transaktionen oft zu einschränkend / zu wenig performant
- Daher: SQL99-Standard definiert vier Isolationsebenen
- Ziel: Vermeidung ungewollter Phänomäne
  - ***dirty reads:*** Transaktion kann Daten einer Transaktion lesen, bevor sie festgeschrieben („*committed*“) wurden
  - ***unrepeatable reads:*** Transaktion sieht bei wiederholtem Lesen festgeschriebene Änderungen anderer Transaktionen
  - ***phantom reads:*** Transaktion sieht bei wiederholtem Lesen, daß andere Transaktionen Datensätze hinzugefügt oder gelöscht haben

Isolationsebenen nach ANSI/ISO-SQL99

Isolations–ebene \ Phänomen	Dirty Reads	Unrepeatable Reads	Phantom Reads
Read Uncommitted	möglich	möglich	möglich
Read Committed	nicht möglich	möglich	möglich
Repeatable Read	nicht möglich	nicht möglich	möglich
Serializable	nicht möglich	nicht möglich	nicht möglich

- *Serializable* entspricht vollständiger Isolation

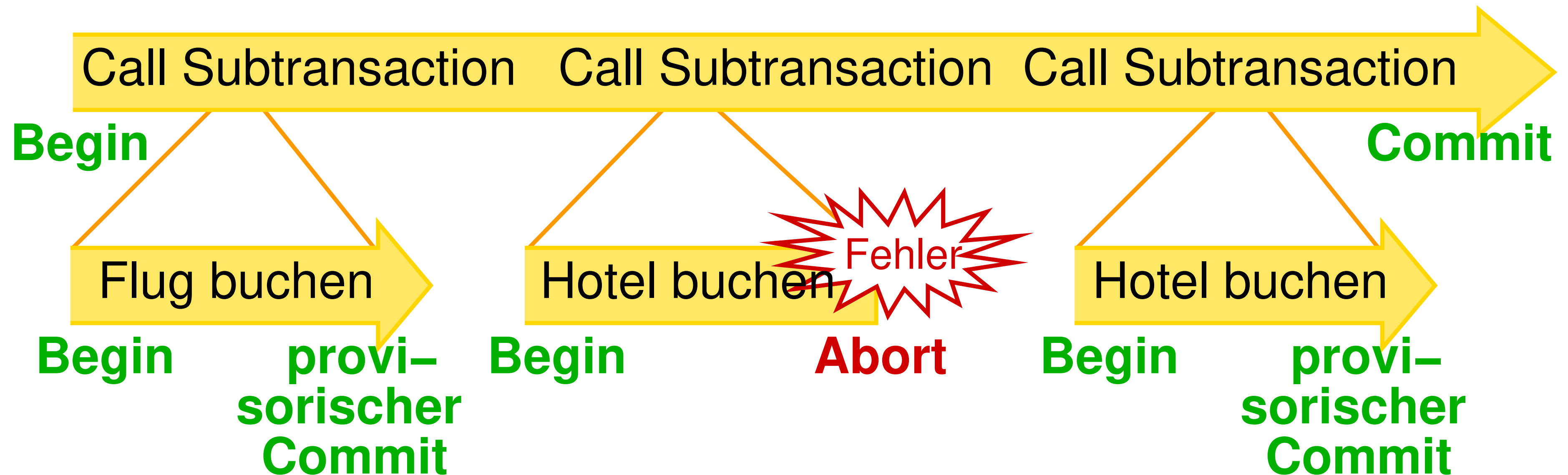


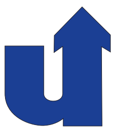
Verschachtelte Transaktionen

- Innerhalb einer Transaktion finden Sub-Transaktionen statt
- Übergeordnete Transaktion kann erfolgreich zu Ende laufen, auch wenn Subtransaktion mit Fehler abgebrochen wurde
- Abbruch der übergeordneten Transaktion führt zur Rückgängigmachung aller Subtransaktionen
- Beispiel: Buchung von Flug und Hotel
  - Buchung des Flugs soll festgehalten werden, auch wenn Hotelbuchung (im ersten Versuch) fehlschlägt
- Verschachtelte Transaktion werden nur von wenigen Transaktionsdiensten unterstützt

Flache Transaktion

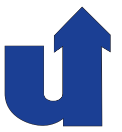
Verschachtelte Transaktionen





Verteilte Transaktionen

- Bisher: Daten sind an genau einer Stelle gespeichert
- Verteilte Transaktionen: Daten sind verteilt gespeichert
- Realisierung von Transaktionen auf den einzelnen Datenressourcen (Datenbanken) nicht mehr ausreichend
  - verteilte Transaktionsverwaltung wird notwendig
- Es gibt ein allgemein anerkanntes Modell der Open Group für die Verwaltung verteilter Transaktionen
  - wird von den meisten Transaktionsdiensten umgesetzt
  - wichtigstes Merkmal: **2-Phasen-Commit**



Modell zur Verwaltung verteilter Transaktionen

Verwaltung verteilter Transaktionen
über Ressourcengrenzen hinweg

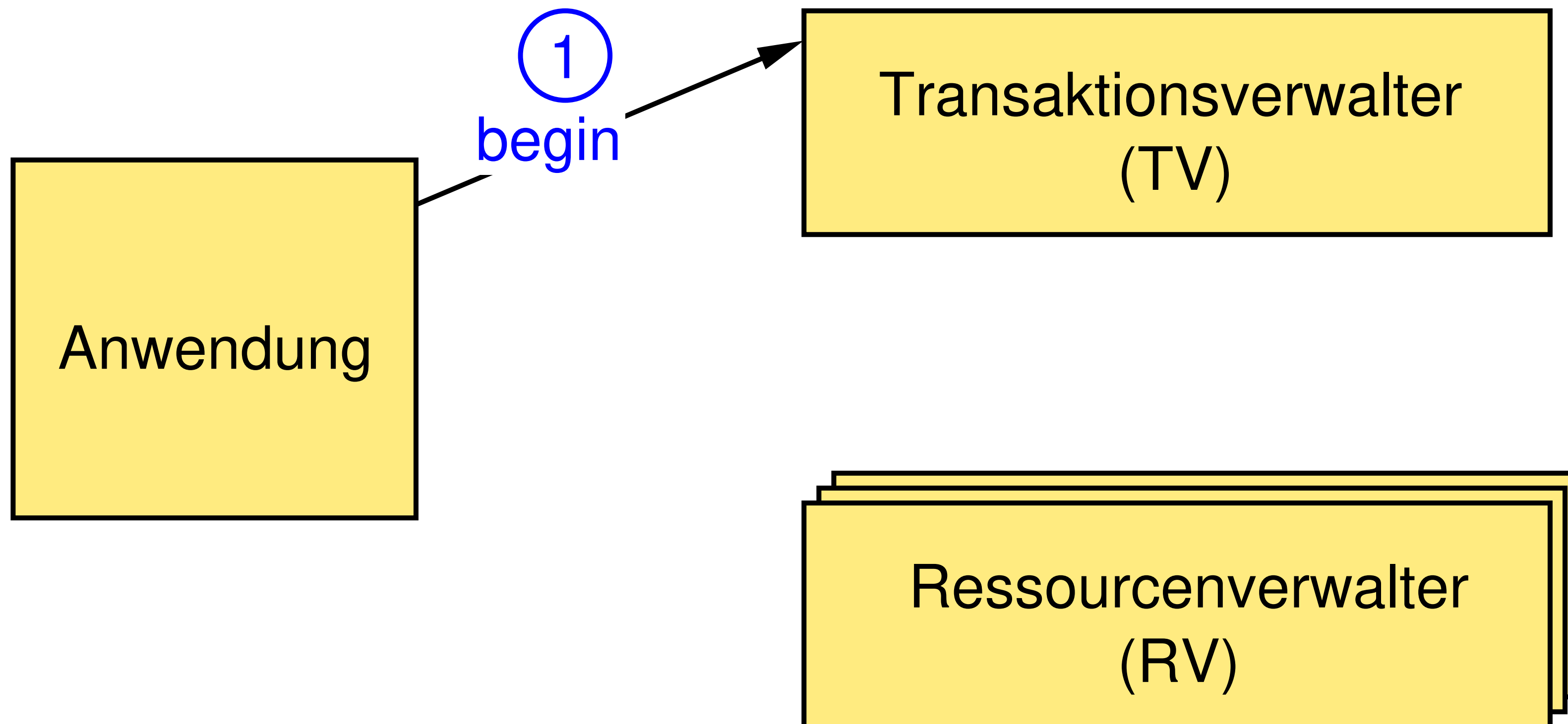
Transaktionsverwalter
(TV)

Anwendung

Ressourcenverwalter
(RV)

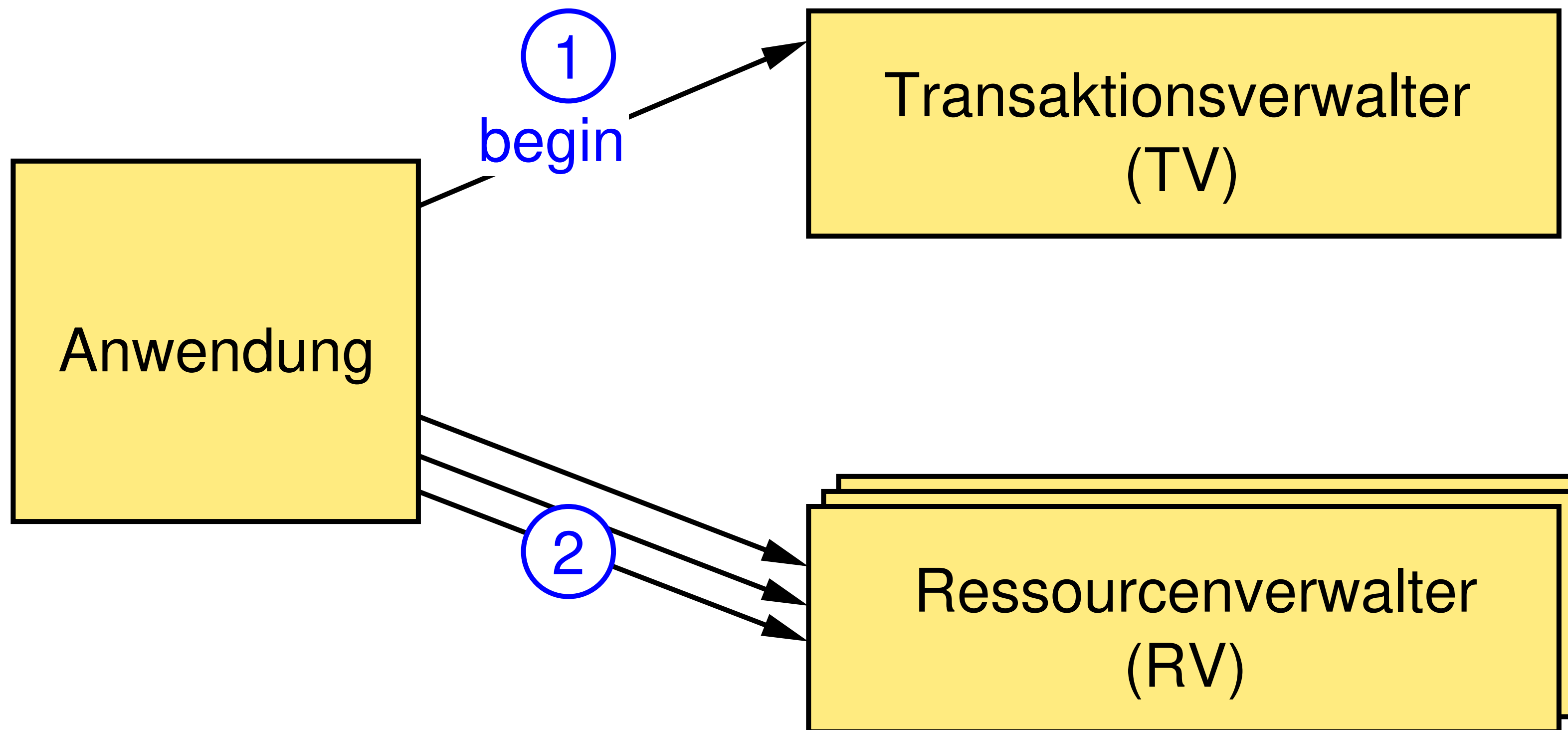
Transaktionsverwaltung innerhalb der
einzelnen Datenressourcen

Modell zur Verwaltung verteilter Transaktionen



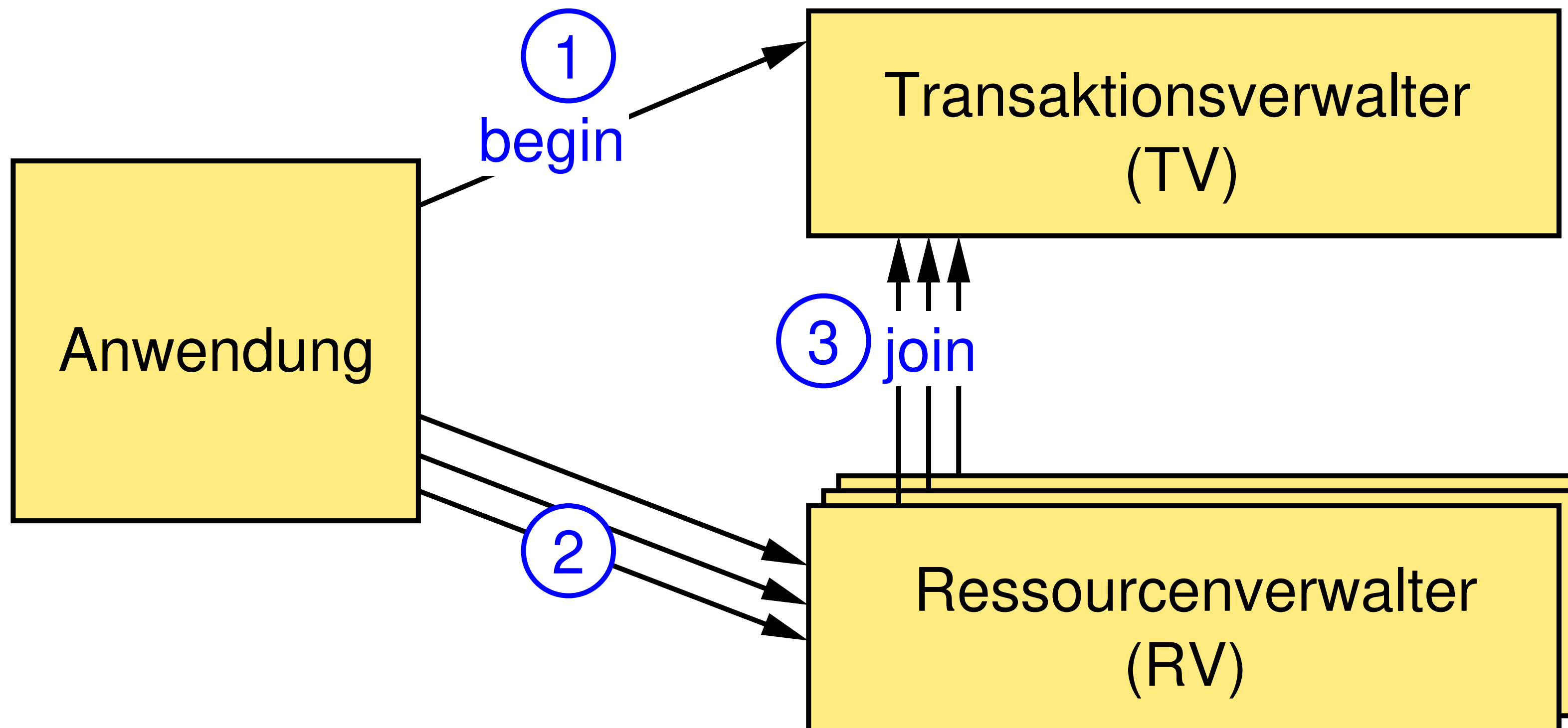
1. Anwendung fordert Start einer neuen Transaktion an.
   TV initialisiert intern neue Transaktion.

Modell zur Verwaltung verteilter Transaktionen



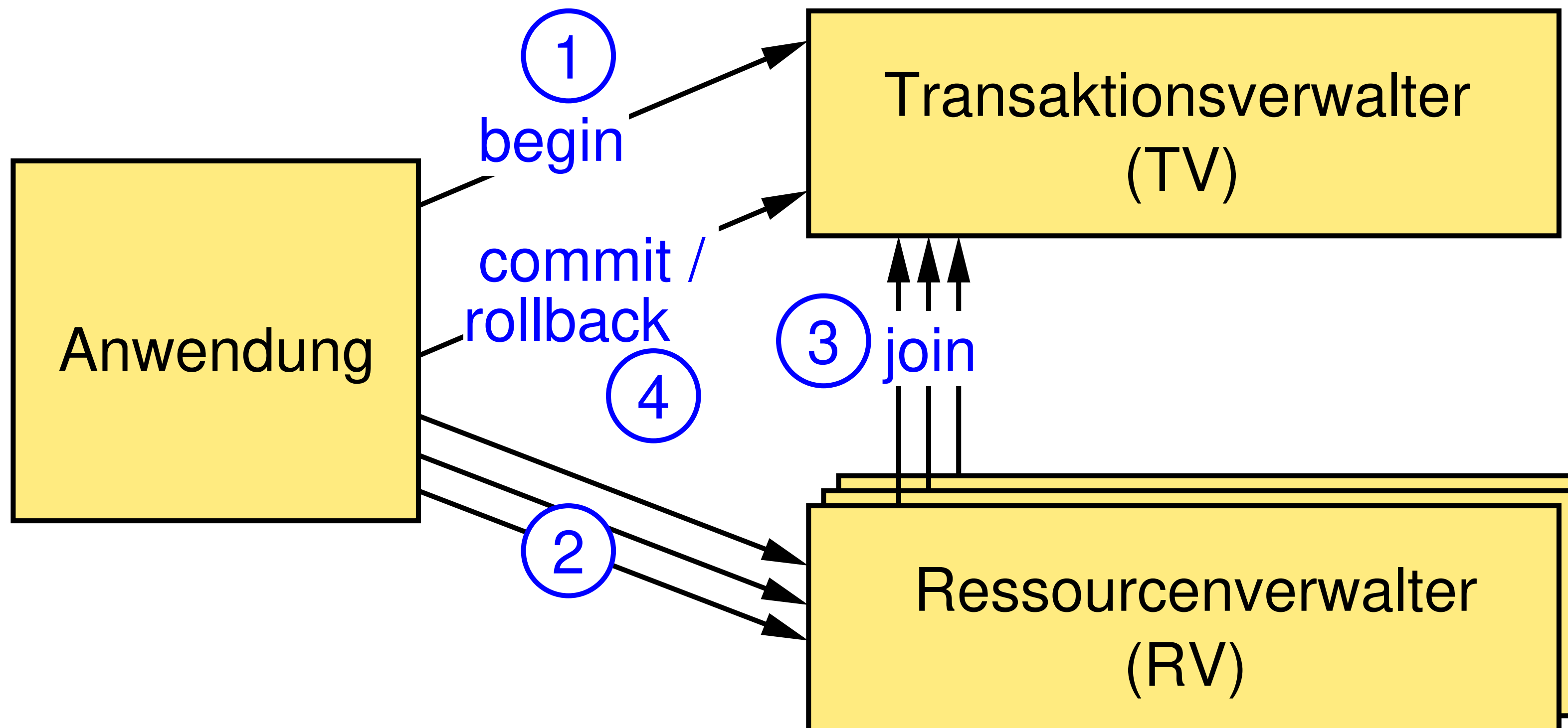
2. Transaktion ist aktiv.
   Anwendung kann auf Ressourcen zugreifen.

Modell zur Verwaltung verteilter Transaktionen



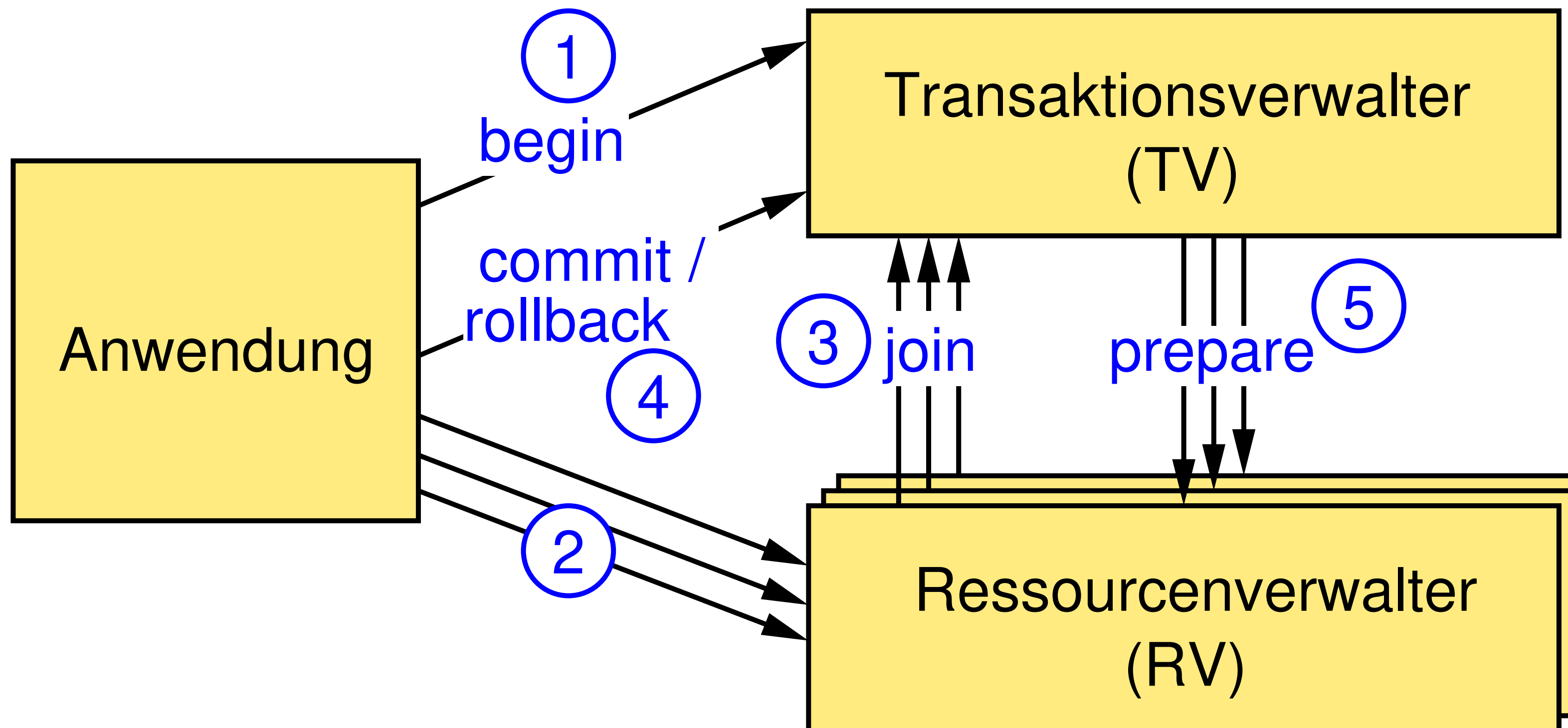
3. Jeder RV, der von Anwendung genutzt wird, meldet sich beim TV für Transaktion an

Modell zur Verwaltung verteilter Transaktionen



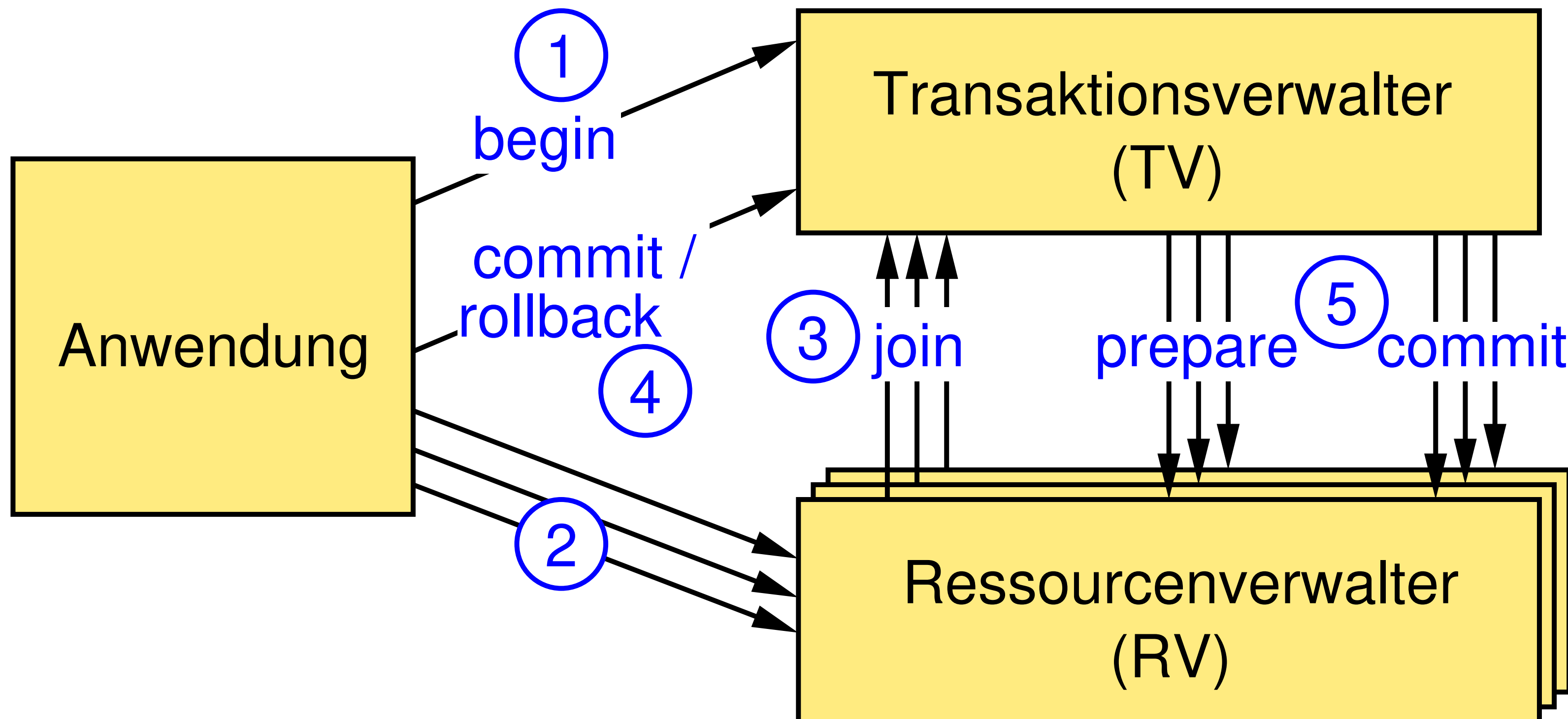
4. Anwendung fordert Festschreiben oder Abbruch der Transaktion

Modell zur Verwaltung verteilter Transaktionen

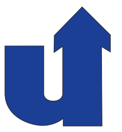


5. TV fordert RV auf, die Änderungen festzuschreiben:
   2–Phasen–Commit

Modell zur Verwaltung verteilter Transaktionen



5. TV fordert RV auf, die Änderungen festzuschreiben:
   2–Phasen–Commit



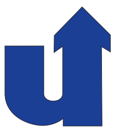
2-Phasen-Commit

- Phase 1 (Abstimmphase)
  - TV fragt bei allen beteiligten RV an, ob *Commit* erfolgreich verlaufen würde („*Prepare*“)
  - RV, der mit „Ja“ antwortet, bereitet Festschreiben vor
- Phase 2 (Abschluß)
  - Falls alle RV mit „Ja“ gestimmt haben:
    - TV sendet *Commit*-Befehl an alle RV
    - RV schreibt Daten endgültig fest und bestätigt an TV
  - sonst:
    - TV sendet *Abort*-Befehl an alle RV

Verteilte Systeme

SoSe 2018

8 Replikation und Konsistenz

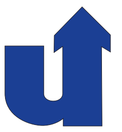


Inhalt

- Einführung, Motivation
- Daten-zentrierte Konsistenzmodelle
- Client-zentrierte Konsistenzmodelle
- Verteilungsprotokolle
- Konsistenzprotokolle

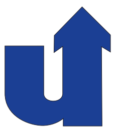
Literatur

- Tanenbaum, van Steen: Kap. 6



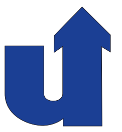
8.1 Einführung und Motivation

- **Replikation**: im verteilten System werden mehrere (identische) Kopien von Datenobjekten vorgehalten
  - Prozesse können auf eine beliebige Kopie zugreifen
- Gründe für die Replikation:
  - Erhöhung von Verfügbarkeit und Zuverlässigkeit
    - falls ein Replikat nicht verfügbar ist, verwende ein anderes
    - Lesen mehrerer Replikate mit Mehrheitsentscheid
  - Steigerung der Leistung
    - bei großen Systemen: nebenläufige (Lese-)Zugriffe auf verschiedene Replikate
    - bei über einen großen Bereich verteilten Systemen: Zugriff auf ein Replikat in der Nähe



Zentrales Problem der Replikation: Konsistenz

- Bei Änderungen der Daten müssen **alle** Replikate konsistent gehalten werden
- Einfachste Möglichkeit: alle Aktualisierungen erfolgen über vollständig sortierten, atomaren Multicast
  - hoher Overhead, wenn häufig Aktualisierungen erfolgen
    - in einigen Repliken werden diese evtl. nie gelesen
  - vollständig sortierter, atomarer Multicast sehr teuer bei vielen / weit verstreuten Repliken
- Strikte Konsistenzerhaltung der Repliken verschlechtert immer die Leistung und Skalierbarkeit
- Lösung: abgeschwächte Konsistenzforderungen
  - oft nur sehr schwache Forderungen, z.B. News, Web, ...

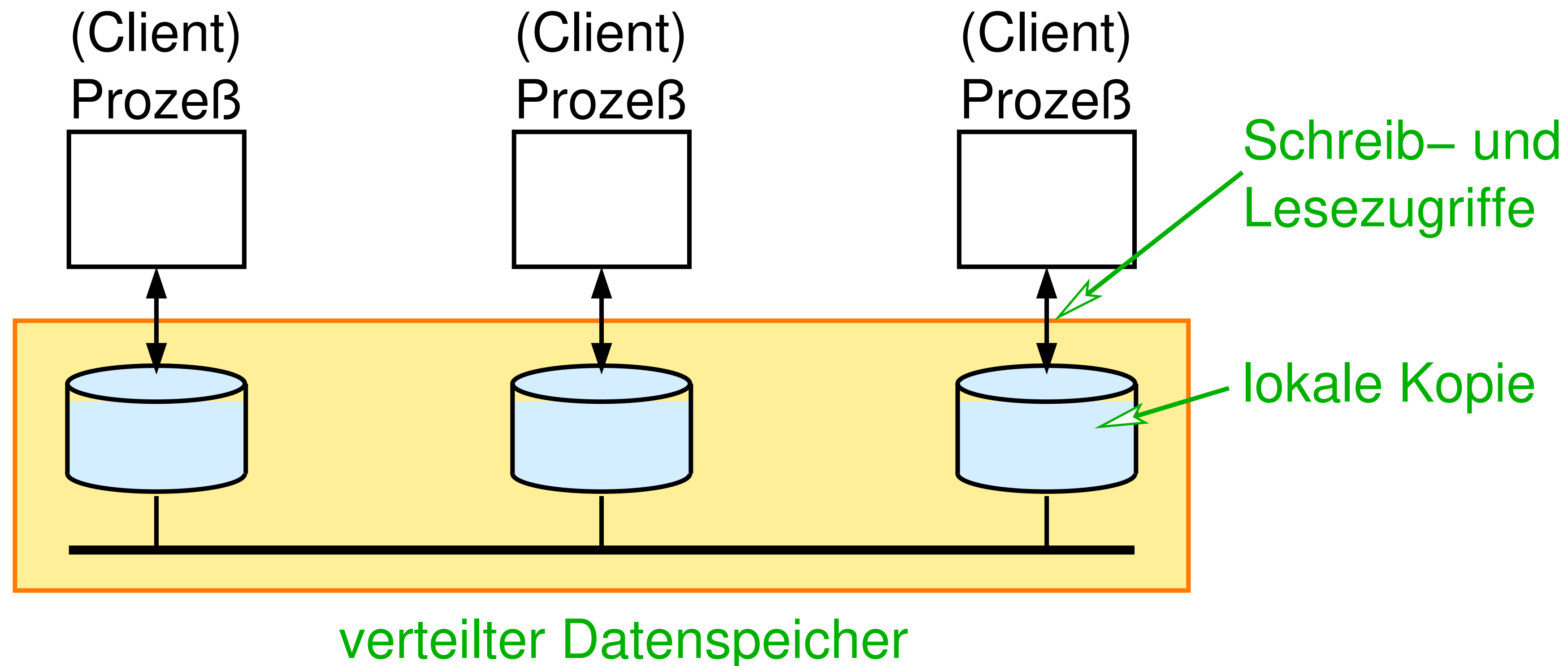


Konsistenzmodelle

- Ein Konsistenzmodell legt fest, in welcher Reihenfolge die Schreiboperationen (Aktualisierungen) der Prozesse von den anderen Prozessen „gesehen“ werden
- Intuitive Erwartung: eine Leseoperation gibt immer das Ergebnis der letzten Schreiboperation zurück (**strenge Konsistenz**)
  - Problem: es gibt keine globale Zeit
    - sinnlos, von der „letzten“ Schreiboperation zu sprechen
  - daher: andere Konsistenzmodelle notwendig
- **Daten-zentrierte Konsistenzmodelle**: Sicht des Datenspeichers
- **Client-zentrierte Konsistenzmodelle**: Sicht **eines** Prozesses
  - Annahme: (i.W.) keine Aktualisierung durch mehrere Prozesse

8.2 Daten-zentrierte Konsistenzmodelle

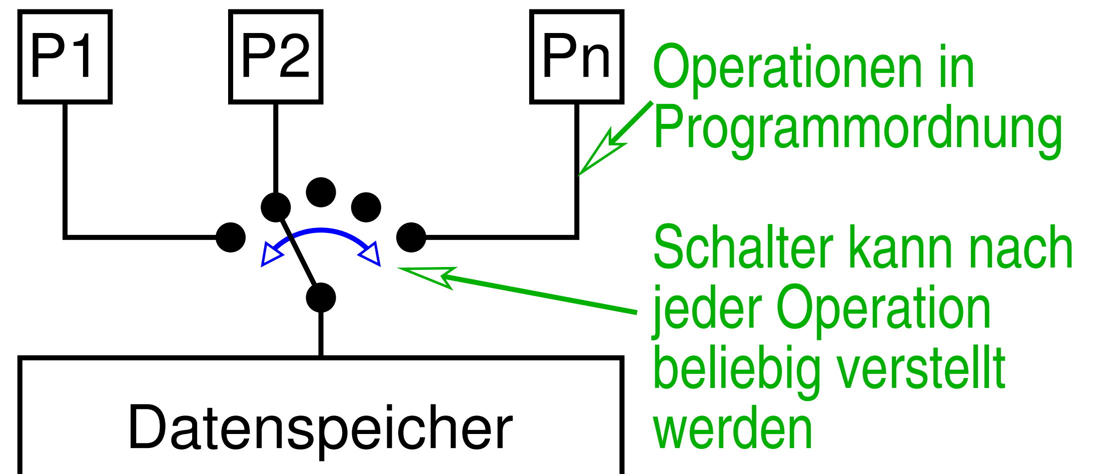
- Modell eines verteilten Datenspeichers:



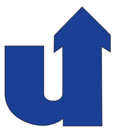
  - logischer, gemeinsam genutzter Datenspeicher
  - physisch über mehrere Knoten verteilt und repliziert

Sequentielle Konsistenz

- Ein Datenspeicher ist **sequentiell konsistent**, wenn das Ergebnis jeder Programmausführung so ist, also ob:
  - die (Schreib-/Lese-)Operationen aller Prozesse in einer (beliebigen) sequentiellen Reihenfolge ausgeführt werden,
  - in der Operationen jedes einzelnen Prozesses in der vom Programm vorgegebenen Reihenfolge erscheinen.
- D.h. die Ausführung der Operationen der einzelnen Prozesse kann beliebig verzahnt werden
- Unabhängig von Zeit bzw. Uhren



- Alle Prozesse sehen die Zugriffe in derselben Reihenfolge



Sequentielle Konsistenz: Beispiele

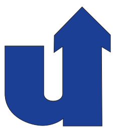
erlaubter Ablauf:

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)b	R(x)a

verbotener Ablauf:

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

- Bezeichnungen:
  - W(x)a : in die Variable ’x’ wird der Wert ’a’ geschrieben
  - R(x)a : Variable ’x’ wird gelesen, Ergebnis ist ’a’
- Eine mögliche sequentielle Reihenfolge des linken Ablaufs:
  - W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a



Sequentielle Konsistenz: Beispiele

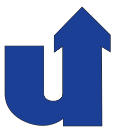
erlaubter Ablauf:

P1:				W(x)a		
P2:	W(x)b					
P3:		R(x)b			R(x)a	
P4:			R(x)b			R(x)a

verbotener Ablauf:

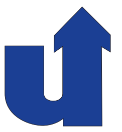
P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

- Bezeichnungen:
  - W(x)a : in die Variable ’x’ wird der Wert ’a’ geschrieben
  - R(x)a : Variable ’x’ wird gelesen, Ergebnis ist ’a’
- Eine mögliche sequentielle Reihenfolge des linken Ablaufs:
  - W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a



Linearisierbarkeit

- Stärker als sequentielle Konsistenz
- Annahme: die Knoten (Prozesse) besitzen synchonisierte Uhren
  - d.h. Approximation einer globalen Zeit
- Operationen besitzen Zeitstempel auf Basis dieser Uhren
- Im Vergleich mit sequentieller Konsistenz zusätzlich gefordert:
  - die sequentielle Reihenfolge der Operationen ist konsistent mit deren Zeitstempeln
- Aufwendige Implementierung
- Verwendet zur formalen Verifikation nebenläufiger Algorithmen



Kausale Konsistenz

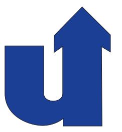
- Abschwächung der sequentiellen Konsistenz
- (Nur) Schreiboperationen, die potentiell kausal abhängig sind, müssen für alle Prozesse in derselben Reihenfolge sichtbar sein

Kausal, aber nicht seq. konsistent:

P1:	W(x)a		W(x)c		
P2:		R(x)a W(x)b			
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)a		R(x)b	R(x)c

nicht kausal konsistent:

P1:	W(x)a			
P2:		R(x)a W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b



Schwache Konsistenz

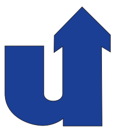
- In der Praxis: Zugriffe auf gemeinsame Ressourcen werden über Synchronisationsvariablen (SV) koordiniert
- Dann: schwächere Konsistenzforderungen ausreichend:
  - Zugriffe auf SV sind sequentiell konsistent
  - keine Operation auf SV erlaubt, bis alle vorhergehenden Schreibzugriffe auf Daten überall abgeschlossen sind
  - keine Operationen auf Daten, bevor alle vorhergehenden Operationen auf SV abgeschlossen sind

Erlaubte Ereignisabfolge:

P1:	W(x)a	W(x)b	S		
P2:			R(x)a	R(x)b	S
P3:			R(x)b	R(x)a	S

Ungültige Ereignisabfolge:

P1:	W(x)a	W(x)b	S		
P2:				S	R(x)a

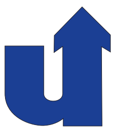


Freigabe-Konsistenz (*Release Consistency*)

- Idee wie bei schwacher Konsistenz, aber Unterscheidung von *acquire* und *release*-Operationen (wechselseitiger Ausschluß!)
  - vor einer Operation auf den Daten müssen alle *acquire*-Operationen des Prozesses abgeschlossen sein
  - vor Ende einer *release*-Operation müssen alle Operationen des Prozesses auf den Daten abgeschlossen sein
  - *acquire* / *release*-Operationen eines Prozesses werden überall in der selben Reihenfolge gesehen

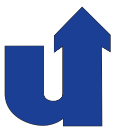
Erlaubte Ereignisabfolge:

P1:	acq(L)	W(x)a	W(x)b	rel(L)				
P2:					acq(L)	R(x)b	rel(L)	
P3:								R(x)a



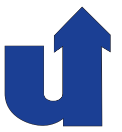
Vergleich der Modelle

Streng	Absolute Zeitreihenfolge aller gemeinsamen Zugriffe (physikalisch nicht sinnvoll!)
Linearisierbarkeit	Alle Prozesse sehen alle Zugriffe in derselben Reihenfolge. Zugriffe sind nach einem (nicht eindeutigen) globalem Zeitstempel sortiert.
Sequentiell	Alle Prozesse sehen alle Zugriffe in derselben Reihenfolge. Zugriffe nicht der Zeit nach sortiert
Kausal	Alle Prozesse sehen kausal verknüpfte Zugriffe in derselben Reihenfolge
Schwach	Daten nur dann verläßlich konsistent, nachdem eine Synchronisierung vorgenommen wurde
Freigabe	Daten werden beim Verlassen des kritischen Bereichs konsistent gemacht

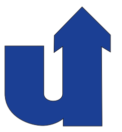


8.3 Client-zentrierte Konsistenzmodelle

- In der Praxis:
  - Clients sind i.d.R. unabhängig voneinander
  - Änderungen an den Daten sind meist selten
  - wegen Partitionierung oft keine Schreib-/Schreib-Konflikte
    - z.B. DNS, WWW (Caches), ...
- **Eventuelle Konsistenz**: alle Repliken werden irgendwann konsistent, wenn längere Zeit keine Aktualisierungen stattfinden
- Problem, falls ein Client die Replik wechselt, auf die er zugreift
  - Aktualisierungen sind dort evtl. noch nicht angekommen
  - Client stellt inkonsistentes Verhalten fest
- Lösung: Client-zentrierte Konsistenzmodelle
  - garantieren Konsistenz für einen **einzelnen** Client
  - aber nicht für nebenläufige Zugriffe durch mehrere Clients



Veranschaulichung des Problems



Monotones Lesen

- Beispiel für client-zentriertes Konsistenzmodell
  - weitere: siehe Tanenbaum / van Steen, Kap. 6.3
- Regel: Wenn ein Prozeß den Wert eines Datums x liest, liefert jede nachfolgende Leseoperation für x denselben oder einen aktuelleren Wert
- Beispiel: Zugriff auf Mailbox an verschiedenen Orten

Mit monotonem Lesen

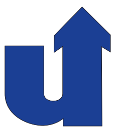
L1:	$WS(x_1)$		$R(x_1)$	
L2:		$WS(x_1;x_2)$		$R(x_2)$

L1/L2: lokale Kopien
WS(...) Menge der Schreiboperationen

Ohne monotones Lesen:

L1:	$WS(x_1)$		$R(x_1)$	
L2:		$WS(x_2)$	$R(x_2)$	$WS(x_1;x_2)$

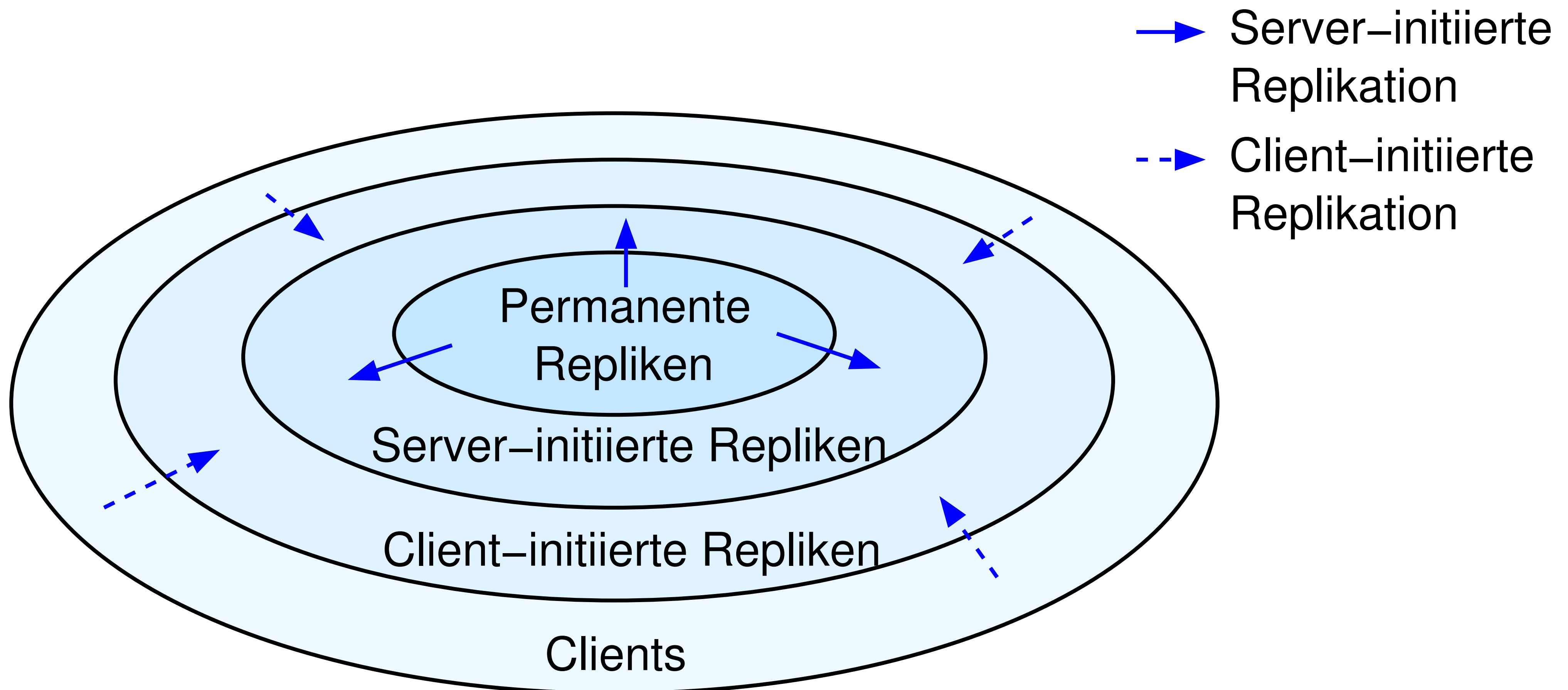
Schreiboperationen auf x in L1 werden hier auf x in L2 ausgeführt



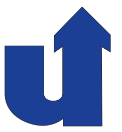
8.4 Verteilungsprotokolle

- Frage: wo, wann und von wem werden Replikate plaziert?
  - permanente Repliken
  - server-initiierte Repliken
  - client-initiierte Repliken
- Frage: wie werden Aktualisierungen verteilt (unabhängig vom Konsistenzprotokoll, ☞ **8.5**)?
  - Versenden von Invalidierungen, Status oder Operationen
  - Pull- oder Push-Protokolle
  - Unicast oder Multicast

Plazierung der Replikate



➥ Alle drei Arten können gleichzeitig auftreten

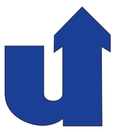


Permanente Repliken

- Ausgangsmenge der Repliken, statisch angelegt, meist klein
- Beispiele:
  - replizierte Web-Site (transparent für Client)
  - Spiegelung (*Mirroring*, Client sucht bewußt ein Replikat aus)

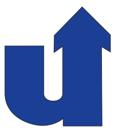
Server-initiierte Repliken

- Server erzeugt bei Bedarf weitere Replikate (*Push-Cache*)
  - z.B. bei Web-Hosting-Diensten
- Schwierig: Entscheidung, wann und wo Replikate erzeugt werden
  - i.a. Zugriffszähler für jede Datei, zusätzlich Information über Herkunft der Anfragen ($\rightarrow$ nächstgelegener Server)



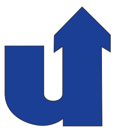
Client-initiierte Repliken

- Andere Bezeichung: ***Client Cache***
- Client Cache speichert (häufig) benutzte Daten lokal zwischen
- Ziel: Verbesserung der Zugriffszeit
- Management des Caches ist komplett dem Client überlassen
  - Server kümmert sich nicht um Konsistenzerhaltung
- Daten werden i.a. nur für begrenzte Zeit im Cache gehalten
  - verhindert Verwendung extrem veralteter Daten
- Cache meist auf Client-Rechner plaziert, oder gemeinsamer Cache für mehrere Clients in deren Nähe
  - z.B. Web-Proxy-Caches



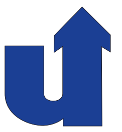
Weitergabe von Aktualisierungen: was wird versendet?

- Der neue Wert des Datenobjekts
  - gut bei hohem Lese-/Aktualisierungsverhältnis
- Die Aktualisierungsoperation (aktive Replikation)
  - spart Bandbreite (Operation mit Parametern i.a. klein)
  - dafür aber mehr Rechenleistung erforderlich
- Nur eine Benachrichtigung (Invalidierungsprotokolle)
  - Benachrichtigung macht Kopie des Datenobjekts ungültig
    - beim nächsten Zugriff wird neue Kopie angefordert
  - benötigt sehr wenig Netzwerkbandbreite
  - gut bei niedrigem Lese-/Aktualisierungsverhältnis



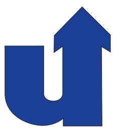
Pull- und Push-Protokolle

- **Push**: Aktualisierungen werden auf Initiative des Server verteilt, der die Änderung vorgenommen hat
  - Repliken müssen keine Aktualisierungen anfordern
  - häufig bei permanenten und server-initiierten Repliken
  - wenn relativ hoher Grad an Konsistenz gefordert ist
  - bei hohem Lese-/Aktualisierungsverhältnis
  - Problem: Server muß alle Repliken kennen
- **Pull**: Repliken fordern die Aktualisierungen der Daten aktiv an
  - häufig bei Client Caches
  - bei niedrigem Lese-/Aktualisierungsverhältnis
  - Nachteil: höhere Antwortzeit bei Cache-Zugriff
- ***Leases***: Mischform: erst für einige Zeit Push, später dann Pull



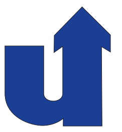
Unicast vs. Multicast

- Unicast: sende Aktualisierung einzeln an jeden Replikat-Server
- Multicast: sende eine Nachricht und überlasse dem Netzwerk die Verteilung (z.B. IP-Multicast)
  - oft wesentlich effizienter
  - vor allem im LAN: Hardware-Broadcast möglich
- Multicast sinnvoll bei Push-Protokollen
- Unicast besser bei Pull-Protokollen
  - nur ein einzelner Client/Server fordert Aktualisierung an



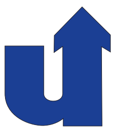
8.5 Konsistenzprotokolle

- Beschreiben, wie sich Replikat-Server untereinander abstimmen, um ein bestimmtes Konsistenzmodell zu implementieren
- Hier speziell betrachtet:
  - Konsistenzmodelle, die Operationen global serialisieren
  - z.B. sequentielle, schwache und Freigabe-Konsistenz
- Zwei grundlegende Vorgehensweisen:
  - *primary-based* (primärbasierte) Protokolle
    - Schreib-Operationen werden immer auf einer speziellen Kopie ausgeführt (**primäre Kopie**)
  - *replicated-write* (repliziertes-Schreiben) Protokolle
    - Schreib-Operationen gehen an mehrere Kopien

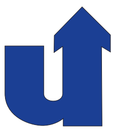


Primary-based-Protokolle

- Lese-Operationen auf beliebigen (lokalen) Kopien
- Schreib-Operationen nur auf der primären Kopie
  - zur Realisierung einer sequentiellen Konsistenz:
    - primäre Kopie aktualisiert alle anderen Kopien und wartet auf Bestätigungen, erst dann Antwort an Client
    - Problem: Performance
- *Remote-write*-Protokolle
  - Schreiber leitet Operation an feste Primärkopie weiter
- *Local-write*-Protokolle
  - Schreiber muß sich vor Aktualisierung primäre Kopie holen
    - d.h. primäre Kopie migriert zwischen den Servern
  - gutes Modell auch für mobile Benutzer

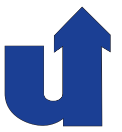


Remote-write-Protokoll: Ablauf (sequentielle Konsistenz)



Remote-write-Protokoll: Ablauf (sequentielle Konsistenz)

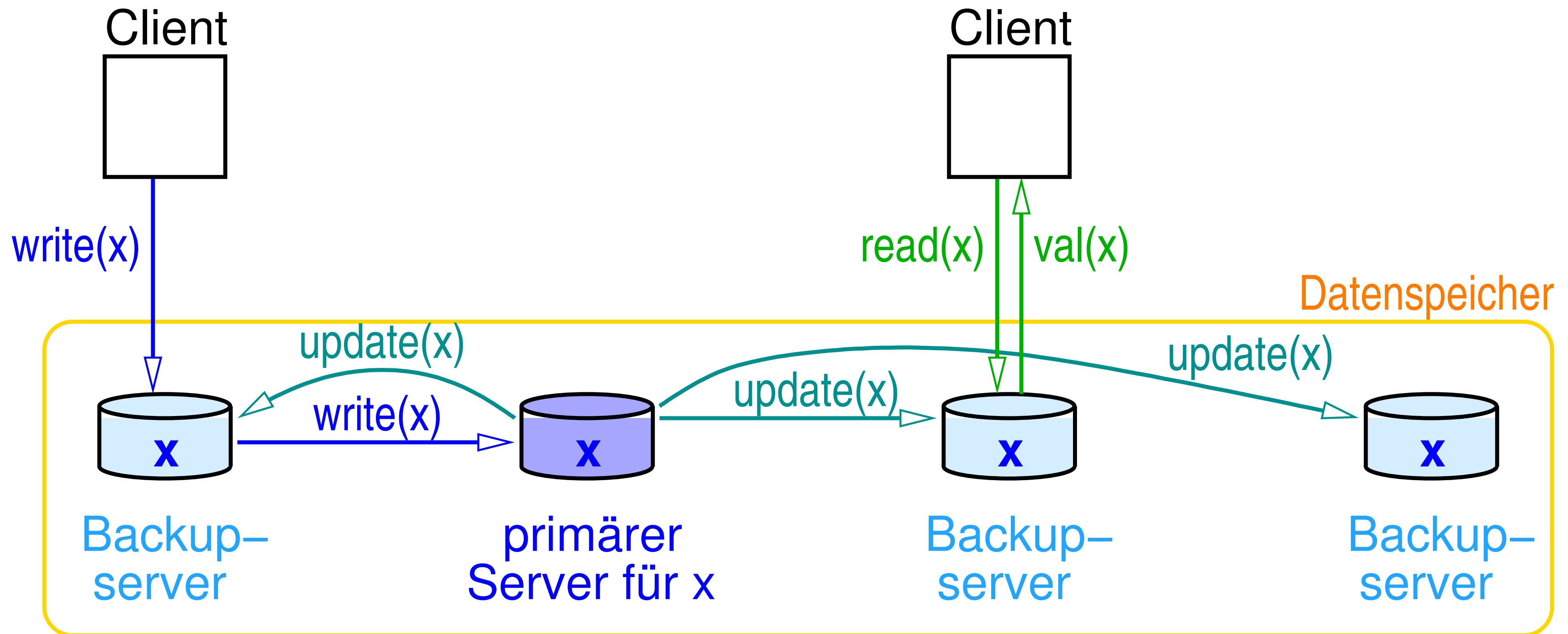
(1) Schreibanforderung;



Remote-write-Protokoll: Ablauf (sequentielle Konsistenz)

(1) Schreibanforderung; Weitergabe an primären Server

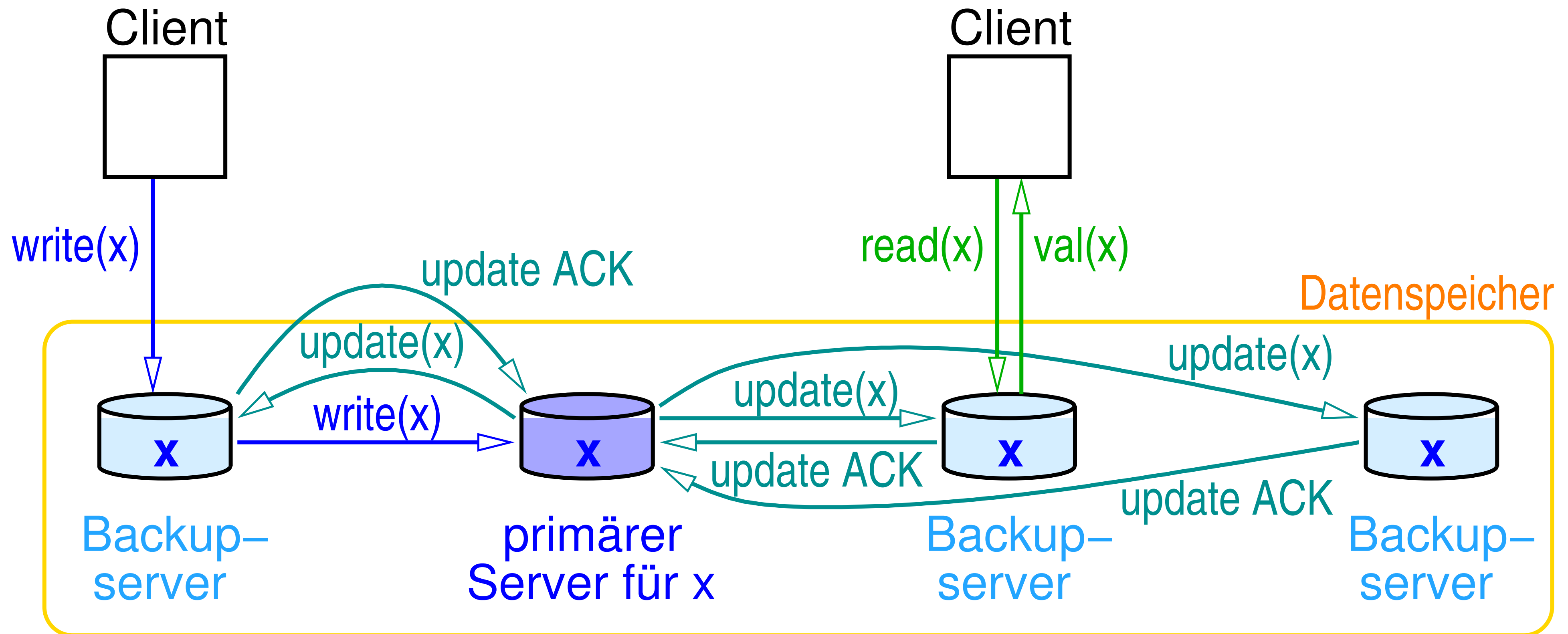
Remote-write-Protokoll: Ablauf (sequentielle Konsistenz)



(1) Schreibanforderung; Weitergabe an primären Server

(2) primärer Server aktualisiert alle Backups

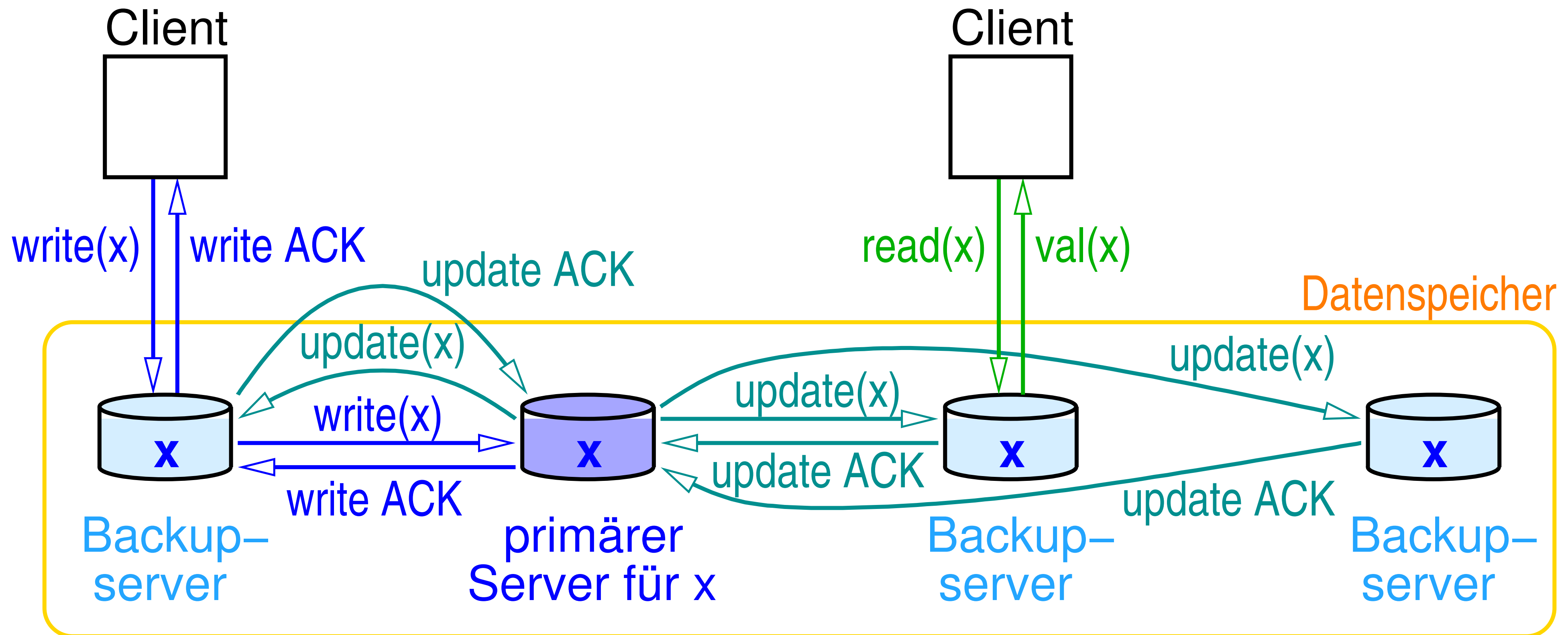
Remote-write-Protokoll: Ablauf (sequentielle Konsistenz)



(1) Schreibanforderung; Weitergabe an primären Server

(2) primärer Server aktualisiert alle Backups und wartet auf Bestätigung

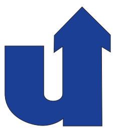
Remote-write-Protokoll: Ablauf (sequentielle Konsistenz)



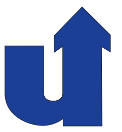
(1) Schreibanforderung; Weitergabe an primären Server

(2) primärer Server aktualisiert alle Backups und wartet auf Bestätigung

(3) Ende der Schreiboperation wird bestätigt

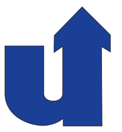


Local-write-Protokoll: Ablauf (Freigabekonsistenz)



Local-write-Protokoll: Ablauf (Freigabekonsistenz)

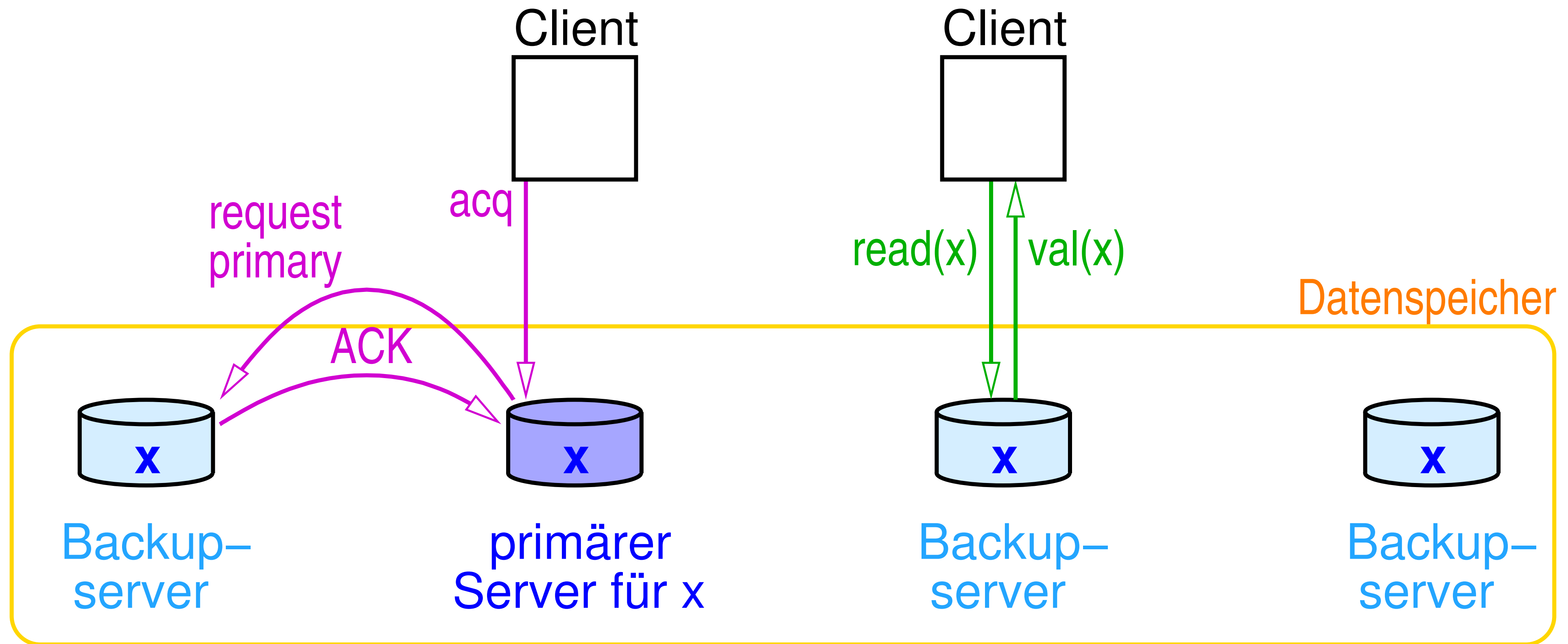
(1) Sperranforderung;



Local-write-Protokoll: Ablauf (Freigabekonsistenz)

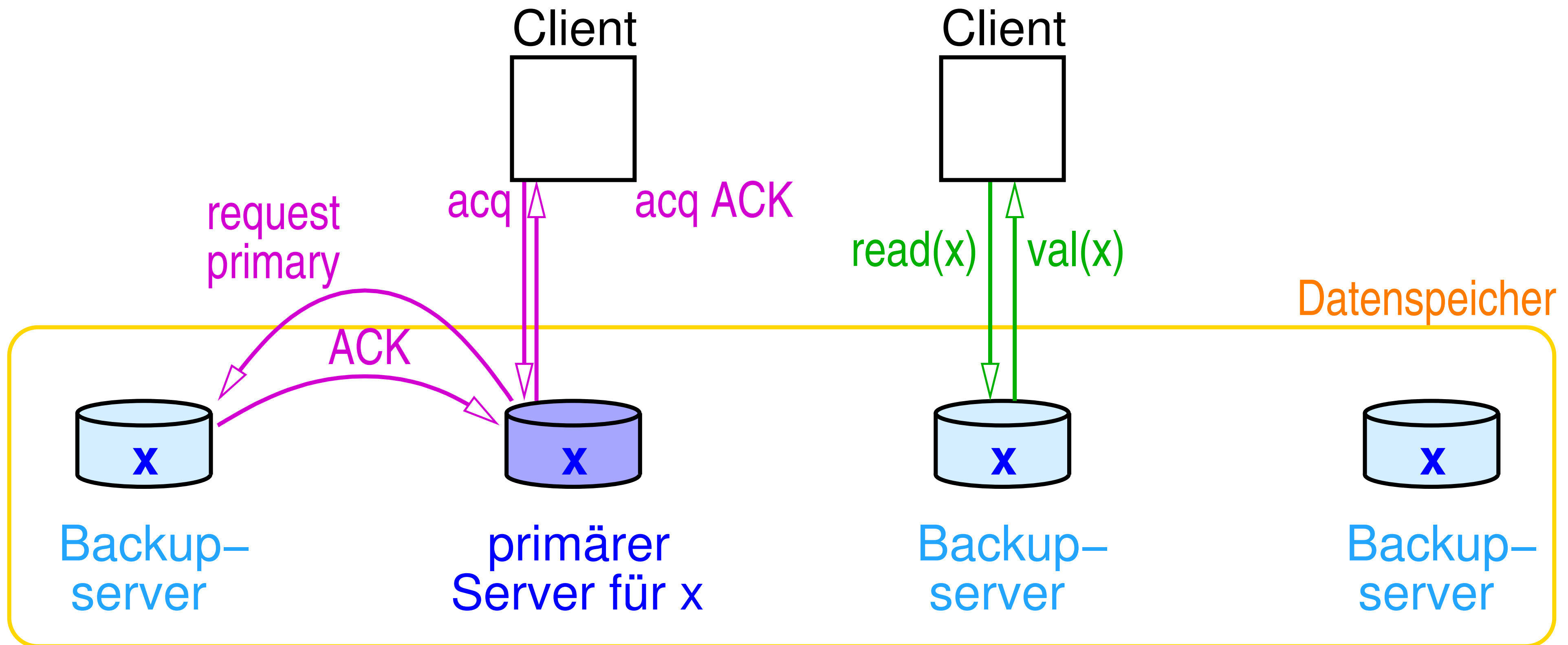
(1) Sperranforderung; Primärkopie auf neuen Server "verschieben"

Local-write-Protokoll: Ablauf (Freigabekonsistenz)



(1) Sperranforderung; Primärkopie auf neuen Server "verschieben"

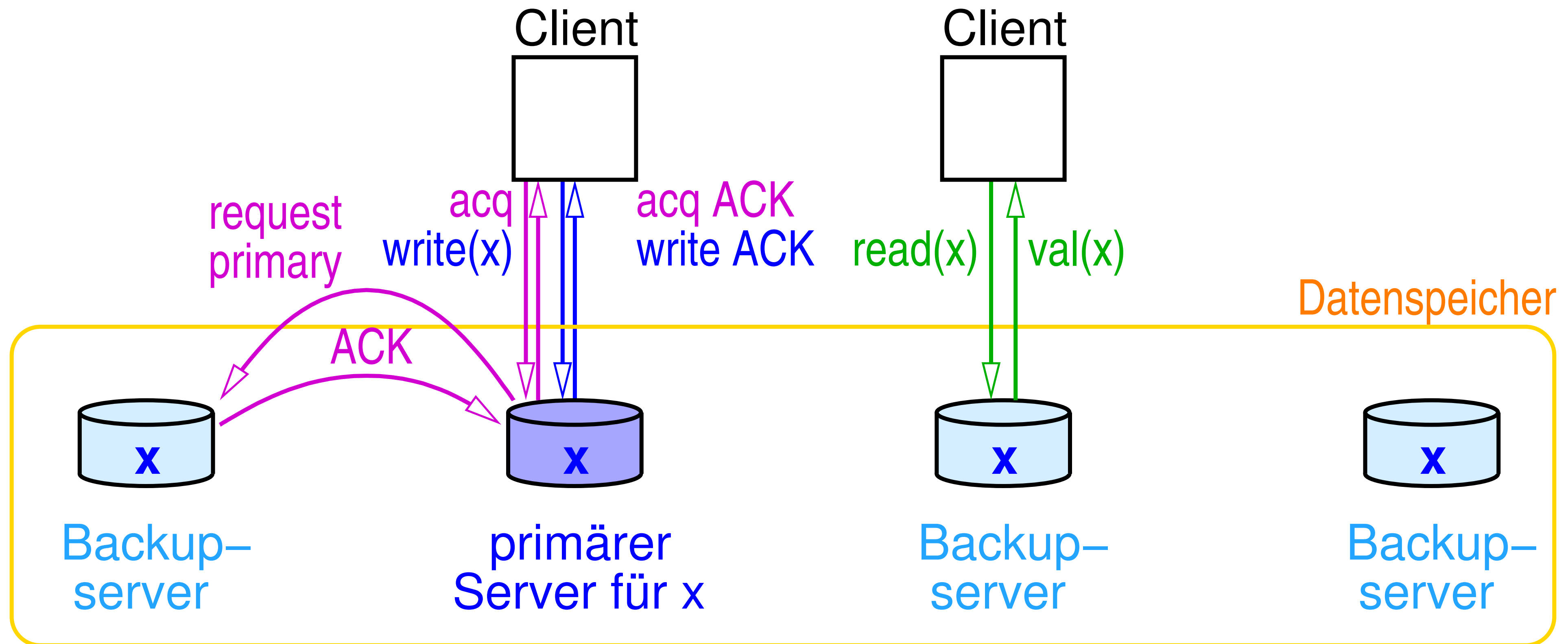
Local-write-Protokoll: Ablauf (Freigabekonsistenz)



(1) Sperranforderung; Primärkopie auf neuen Server "verschieben"

(2) Ende der Sperroperation wird bestätigt

Local-write-Protokoll: Ablauf (Freigabekonsistenz)



(1) Sperranforderung; Primärkopie auf neuen Server "verschieben"

(2) Ende der Sperroperation wird bestätigt

(3) Schreiboperationen werden (nur) auf lokalem Server durchgeführt

Local-write-Protokoll: Ablauf (Freigabekonsistenz)

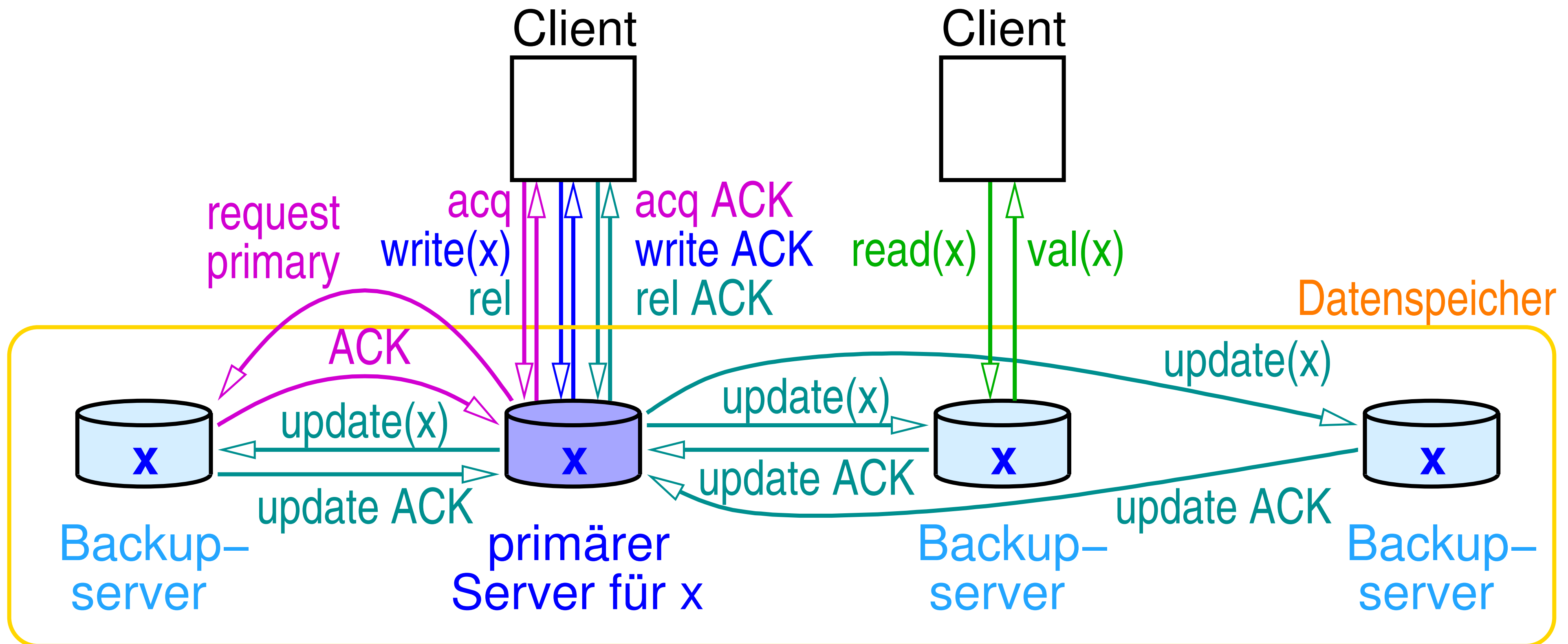
(1) Sperranforderung; Primärkopie auf neuen Server "verschieben"

(2) Ende der Sperroperation wird bestätigt

(3) Schreiboperationen werden (nur) auf lokalem Server durchgeführt

(4) neuer primärer Server aktualisiert Backups

Local-write-Protokoll: Ablauf (Freigabekonsistenz)

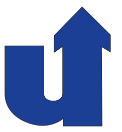


(1) Sperranforderung; Primärkopie auf neuen Server "verschieben"

(2) Ende der Sperroperation wird bestätigt

(3) Schreiboperationen werden (nur) auf lokalem Server durchgeführt

(4) neuer primärer Server aktualisiert Backups und wartet auf Bestätigung

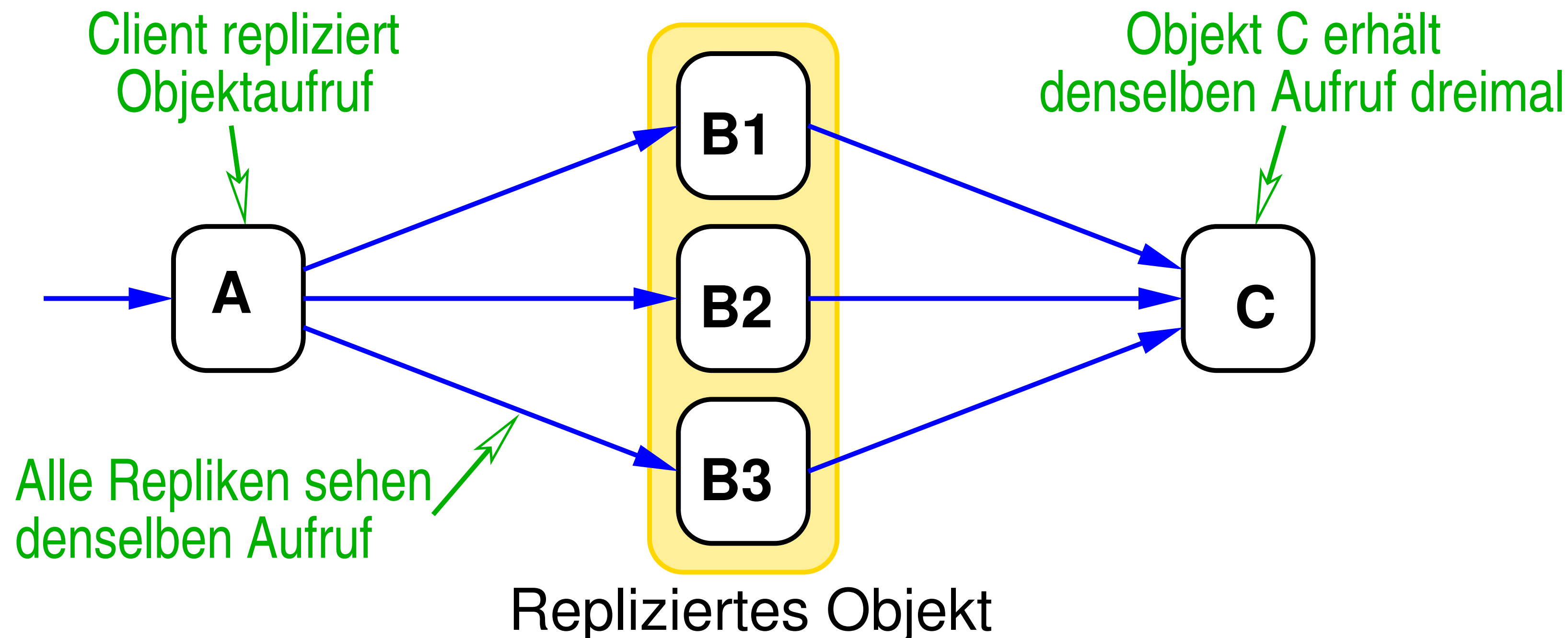


Replicated-Write-Protokolle

- Erlauben die Ausführung von Schreiboperationen auf (mehreren) beliebigen Replikaten
- Zwei Ansätze:
  - aktive Replikation
    - Schreiboperationen werden an alle Kopien weitergegeben
    - Forderung: global eindeutige Reihenfolge der Operationen
      - über vollständig sortierten Multicast
      - oder über zentralen *Sequencer*-Prozeß
  - Quorum-basierte Protokolle
    - nur ein Teil der Replikate wird geändert
    - dafür müssen auch mehrere Kopien gelesen werden

Problem bei replizierten Objektaufrufen

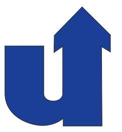
- Was passiert, wenn ein repliziertes Objekt ein anderes aufruft?



- Lösung: Middelware, die sich der Replikation bewußt ist
  - Koordinator von B sorgt dafür, daß nur ein Aufruf an C geht und Ergebnis an alle Repliken von B verteilt wird

Quorum-basierte Protokolle

- Clients benötigen Erlaubnis mehrerer Server zum Schreiben **und** zum Lesen
- Beim Schreiben: Anfrage an (mindestens) N_W Kopien
  - deren Server müssen der Änderung zustimmen
  - Daten bekommen bei Änderung eine neue Versionsnummer
  - Bedingung: $N_W > N/2$ (N = Gesamtanzahl der Kopien)
    - verhindert Schreib-/Schreib-Konflikte
- Beim Lesen: Anfrage an (mindestens) N_R Kopien
  - Client wählt die neueste Version (höchste Versionsnummer)
  - Bedingung: $N_R + N_W > N$
    - stellt sicher, daß immer neueste Version gelesen wird



Quorum-basierte Protokolle: Beispiele

Lesequorum

A	B	C	D
E	F	G	H
I	J	K	L

A	B	C	D
E	F	G	H
I	J	K	L

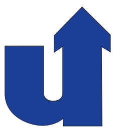
A	B	C	D
E	F	G	H
I	J	K	L

Schreibquorum

$N_R = 6 \quad N_W = 7$ — korrekt

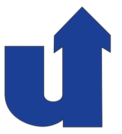
$N_R = 7 \quad N_W = 6$ — Schreib–/Schreib–Konflikte möglich ($N_W \leqslant N/2$)

$N_R = 1 \quad N_W = 12$ — korrekt



8.6 Zusammenfassung

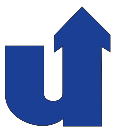
- Replikation wegen Verfügbarkeit und Leistung
- Problem: Konsistenzerhaltung der Kopien
  - strengstes Modell: sequentielle Konsistenz
  - Abschwächungen: kausale K., schwache K, Freigabe-K.
  - Client-zentrische Konsistenzmodelle
- Implementierung von Replikation und Konsistenz:
  - Replikationsschema: statisch, server-initiiert, client-initiiert
  - Verteilungsprotokolle
    - Art der Aktualisierung, Push / Pull, Unicast / Multicast
  - Konsistenzprotokolle
    - *primary based- / replicated write*-Protokolle



Verteilte Systeme

SoSe 2018

9 Verteilte Dateisysteme

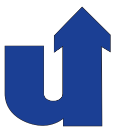


Inhalt

- Allgemeines
- Fallstudie: NFS

Literatur

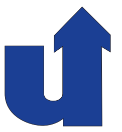
- Tanenbaum, van Steen: Kap. 10
- Colouris, Dollimore, Kindberg: Kap. 8



[Coulouris, 8.1-8.3]

9.1 Allgemeines

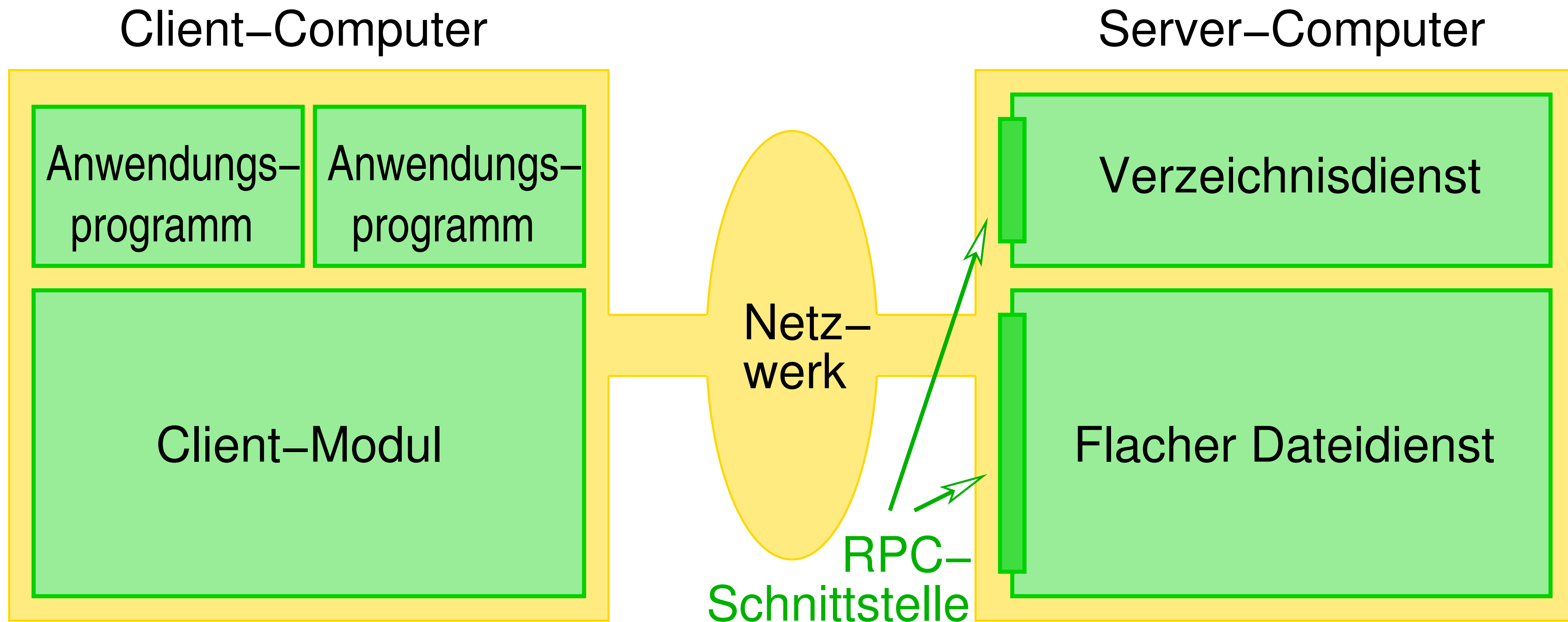
- Ziel: Unterstützung der gemeinsamen Nutzung von Information (Dateien) im **Intra**net
  - im Internet: WWW
- Erlaubt Anwendungen, auf entfernte Dateien genauso zuzugreifen wie auf lokale
  - ähnliche (z.T. sogar bessere) Leistung und Zuverlässigkeit
- Ermöglicht Betrieb von plattenlosen Knoten (*diskless nodes*)
- Beispiele:
  - NFS (Standard im UNIX-Bereich)
  - AFS (Ziel: Skalierbarkeit), CIFS (Windows), CODA, xFS, ...



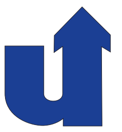
Anforderungen

- Transparenz: Zugriffs-, Orts-, Mobilitäts-, Leistungs- und Skalierungs-Transparenz
- Nebenläufige Dateiaktualisierungen (z.B. Sperren)
- Dateireplikation (häufig: lokales Caching)
- Heterogenität von Hardware und Betriebssystem
- Fehlertoleranz (inbesondere bei Server-Ausfall)
  - oft: *at-least-once* Semantik + idempotente Operationen
  - vorteilhaft: zustandslose Server (einfacher Neustart)
- Konsistenz (☞ **8**)
- Sicherheit (Zugriffskontrolle, Authentifizierung, Verschlüsselung)
- Effizienz

Modell-Architektur eines verteilten Dateisystems



- Aufgaben des Client-Moduls:
  - Emulation der Dateischnittstelle des lokalen BSs
  - ggf. Caching von Dateien bzw. Dateiabschnitten

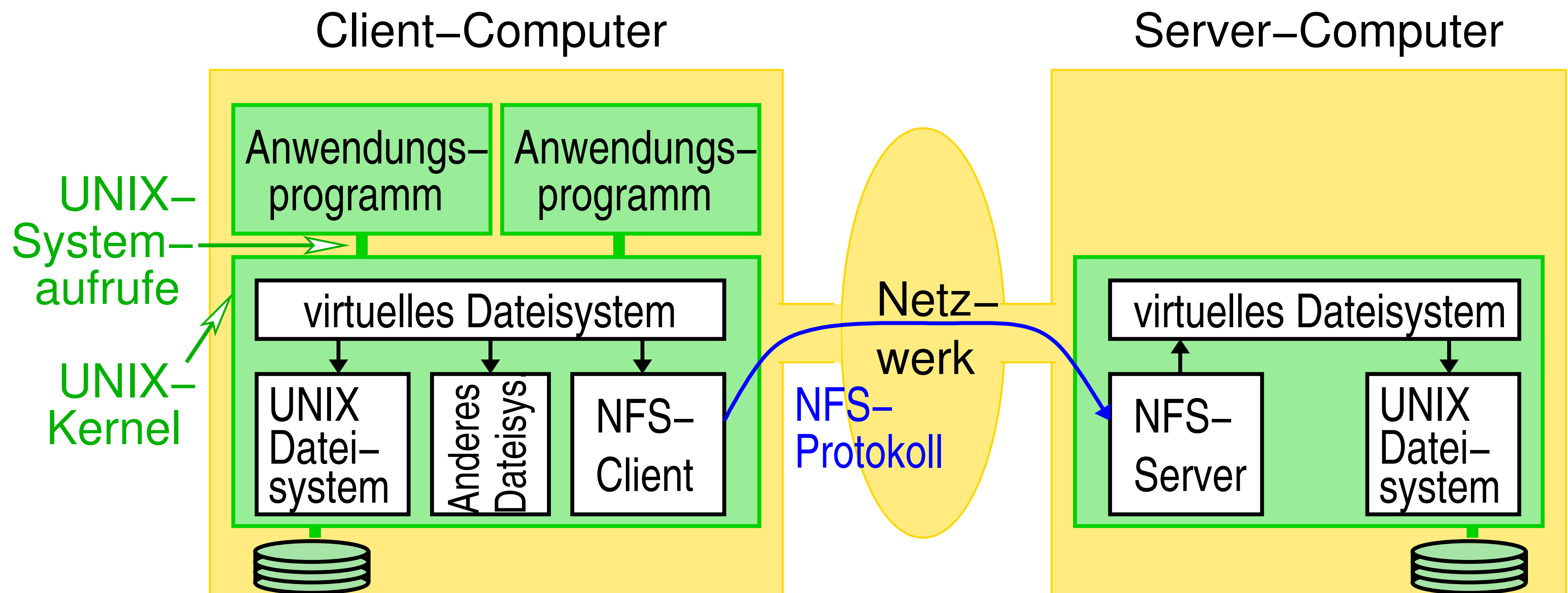


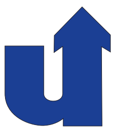
Modell-Architektur eines verteilten Dateisystems ...

- Flacher Dateidienst:
  - bietet idempotente Zugriffsoperationen auf Dateien
    - z.B. *read*, *write*, *create*, *remove*, *getAttributes*, *setAttributes*
    - kein *open* / *close*, kein impliziter Dateizeiger
  - Dateien werden durch UFIDs (*Unique File IDs*) identifiziert
    - (lange) ganzzahlige IDs, können als *Capabilities* dienen
- Verzeichnisdienst:
  - bildet Datei- bzw. Pfadnamen auf UFIDs ab
    - ggf. erst Authentifizierung und Prüfung der Zugriffsrechte
  - Dienste zum Erzeugen, Löschen und Modifizieren von Verzeichnissen

9.2 Fallstudie: NFS

- Eingeführt 1984 von Sun
- Offenes, BS-unabhängiges Protokoll
- Architektur:



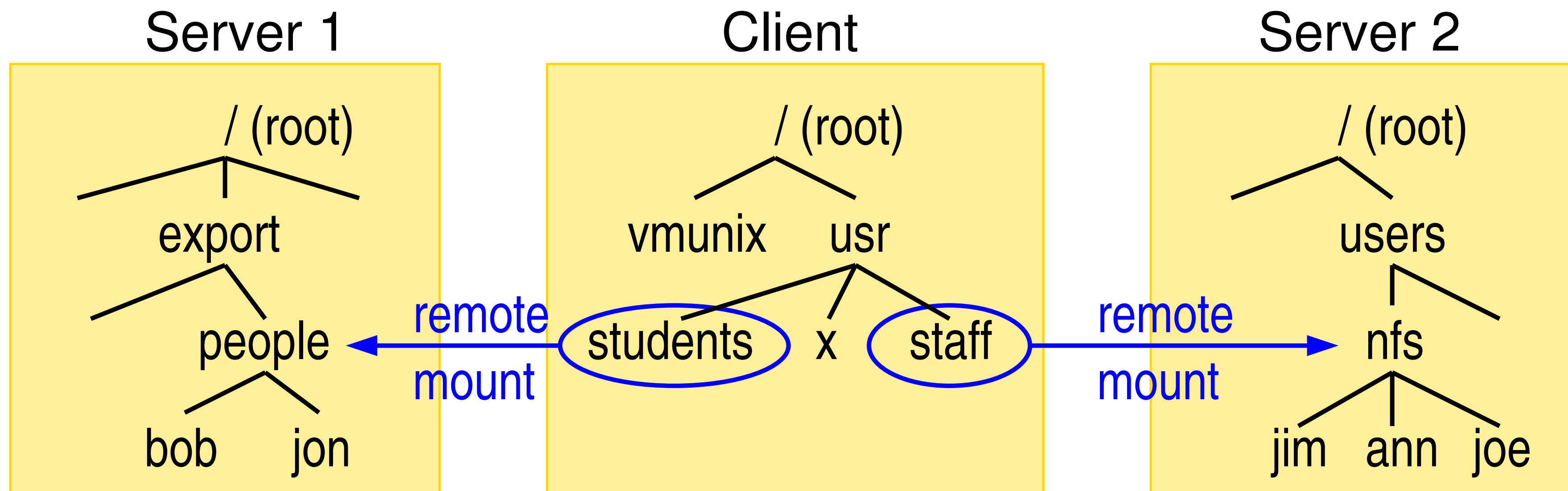


Zugriffskontrolle und Authentifizierung

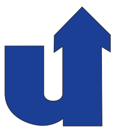
- NFS-Server ist zustandslos (bis einschl. NFS3)
- UFID (Datei-Handle): i.w. nur Dateisystem-ID und I-Node
  - keine *Capability*
- Überprüfung der Zugriffsrechte daher bei jeder Anforderung
  - durch RPC-Protokoll
- Authentifizierung meist nur über Benutzer- und Gruppen-ID
  - extrem unsicher!
- Weitere Möglichkeiten in NFS3:
  - Diffie-Hellman-Schlüsselaustausch (unsicher)
  - Kerberos
- NFS4: sicherer RPC (RPCSEC_GSS)

Mount-Dienst

- NFS-Dateisystem kann in lokalen Verzeichnisbaum eingehängt werden (*mount*)



- Zusammenarbeit des `mount`-Kommandos im Client mit Mount-Dienst des NFS-Servers
  - Mount-Dienst stellt auf Anfrage Datei-Handles der exportierten Verzeichnisse zur Verfügung (für Namensauflösung)

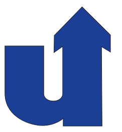


Übersetzung von Pfadnamen

- Iterativ (NFS3): für jedes Verzeichnis Anfrage an NFS-Server
  - notwendig, da Pfad Mount-Punkte überschreiten kann
  - Ineffizienz wird durch Client-Caching gemildert

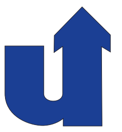
Automounter

- Ziel: Einrichtung eines NFS-Mounts erst beim Zugriff
  - bessere Fehlertoleranz, Lastausgleich möglich
- Automounter ist lokaler NFS-Server
  - dadurch sieht er *lookup()*-Anforderungen des Clients
- bei Anforderung: Einrichten des NFS-Mounts und eines symbolischen Links zum Mount-Punkt
- nach längerer Inaktivität: Aufheben des Mounts



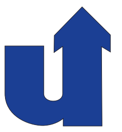
Server-Caching

- Traditionelles File-Caching in UNIX:
  - Puffer im Hauptspeicher für zuletzt verwendete Plattenblöcke
  - *Read-ahead*: Folgeblöcke werden vorab in Cache geladen
  - *Delayed-write*: modifizierte Blöcke erst zurückgeschrieben, wenn Platz benötigt wird; zusätzlich alle 30s durch `sync`
- Server-Caching in NFS: zwei Modi
  - *Write-through*: Schreibanfragen werden im Server-Cache und sofort auch auf Festplatte ausgeführt
    - Vorteil: kein Datenverlust bei Serverabsturz
  - *Delayed-write*: modifizierte Daten bleiben im Cache, bis *commit*-Operation ausgeführt (d.h. Datei geschlossen) wird
    - Vorteil: bessere Leistung, falls viele Schreiboperationen



Client-Caching

- NFS-Client speichert Ergebnisse von (u.a.) *read* / *write* und *lookup*-Operationen in lokalem Cache zwischen
  - führt zu Konsistenzproblem, da jetzt mehrere Kopien
- Client ist für Konsistenzerhaltung verantwortlich
- Aktualität des Cache-Eintrags wird bei jedem Zugriff geprüft
  - dazu: Vergleich, ob Modifikations-Zeitstempel im Cache mit Modifikations-Zeitstempel auf Server übereinstimmt
  - bei negativer Validierung: Cache-Eintrag wird gelöscht
  - bei erfolgreicher Validierung: Cache-Eintrag wird eine gewisse Zeit (3 - 30 s) ohne weitere Prüfungen als aktuell erachtet
    - d.h. Änderungen erst nach einigen Sekunden sichtbar
    - Kompromiß zwischen Konsistenz und Effizienz



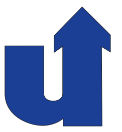
Client-Caching ...

- Behandlung von Schreiboperationen:
  - Dateiblock wird im Cache als *dirty* markiert
  - markierte Blöcke werden asynchron an Server gesendet:
    - beim Schließen der Datei
    - bei `sync`-Operation auf Client-Rechner
    - ggf. öfter durch *Block-Input/Output* (Bio)-Dämonen
- Bio-Dämonen realisieren asynchrone Operationen für *Read-ahead* und *Delayed-write*
  - zur Leistungsoptimierung
- NFS garantiert keine wirkliche Konsistenz der Client-Caches

Verteilte Systeme

SoSe 2018

10 Verteilter Gemeinsamer Speicher

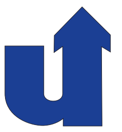


Inhalt

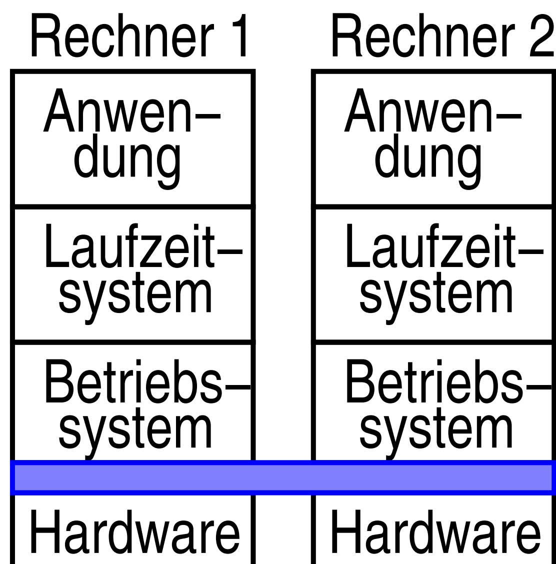
- Einführung
- Designalternativen

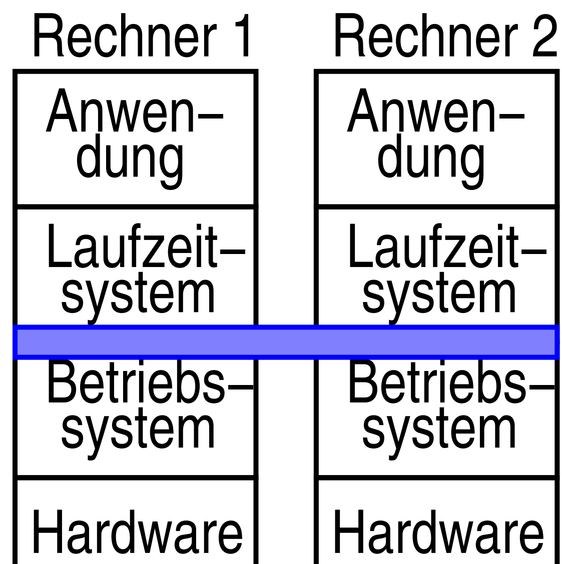
Literatur

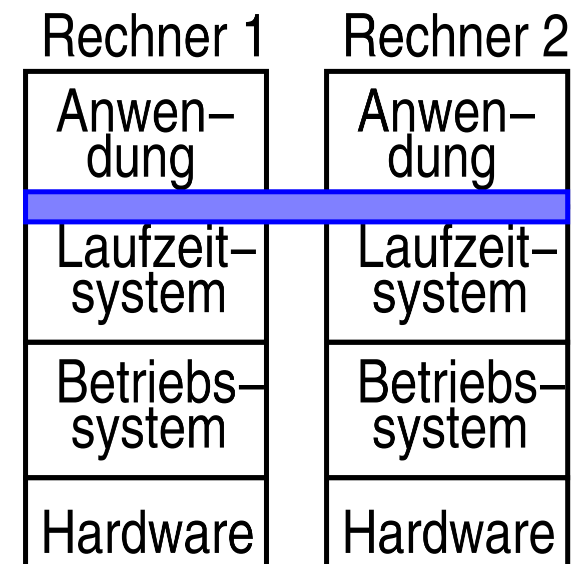
- Colouris, Dollimore, Kindberg: Kap. 16.1-16.3

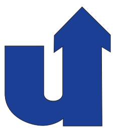


- Ziel: gemeinsamer Speicher in verteilten Systemen
- Hier betrachtete, grundlegende Technik:
  - seitenbasierte Speicherverwaltung auf den Knoten
  - bei Bedarf: Nachladen von Seiten über das Netzwerk
  - ggf. Replikation von Seiten zur Leistungssteigerung
- Zur Abgrenzung:



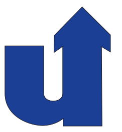






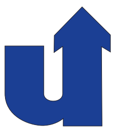
Designalternativen

- Struktur des gemeinsamen Speichers:
  - byteorientiert (gemeinsame, verteilte Speicherseiten)
  - objektorientiert (gemeinsame, verteilte Objekte)
    - z.B. Orca
  - unveränderliche Daten (gemeinsamer, verteilter Container)
    - Operationen: lesen, hinzufügen, entfernen
    - z.B. Linda *Tuple Space*, JavaSpaces
- Granularität (bei seitenbasierten Verfahren):
  - bei Änderung eines Bytes: Übertragung der gesamten Seite
  - bei großen Seiten: effizientere Kommunikation, weniger Verwaltungsaufwand, mehr *False Sharing*



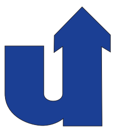
Designalternativen ...

- Konsistenzmodell: meist sequentielle oder Freigabe-Konsistenz
- Konsistenzprotokoll: typ. *Local write*-Protokoll
  - d.h. Speicherseite migriert zum zugreifenden Prozeß
  - mit oder ohne Replikation bei Lesezugriffen
    - client-initiierte Replikation, d.h. Leser fordert Kopie an
  - i.a. nur jeweils ein Schreiber für eine Seite
  - meist Invalidierungsprotokolle (mit Push-Modell)
  - Aktualisierungsprotokolle nur, wenn Schreibzugriffe gepuffert werden können (z.B. bei Freigabekonsistenz)



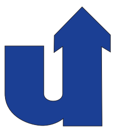
Designalternativen ...

- Verwaltung von Kopien
  - meist: jederzeit entweder mehrere Leser oder ein Schreiber
  - jede Seite hat einen Eigentümer
    - Schreiber oder einer der Leser (letzter Schreiber)
    - verwaltet Liste der Prozesse mit Kopien der Seite
  - vor Schreibzugriff: Prozeß fordert aktuelle Kopie an
- Finden des Eigentümers einer Seite:
  - zentraler Manager
    - verwaltet Eigentümer, gibt Anforderungen weiter
  - feste Verteilung
    - feste Abbildung Seite $\rightarrow$ Manager



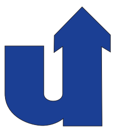
Designalternativen ...

- Finden des Eigentümers einer Seite ...:
  - Multicast statt Manager
    - Problem: gleichzeitige Anforderungen
    - Lösung: vollständig sortierter Multicast, Vektorzeitstempel
  - dynamisch verteilter Manager
    - jeder Prozeß kennt wahrscheinlichen Eigentümer
    - dieser gibt Anfrage ggf. weiter
    - wahrscheinlicher Eigentümer wird aktualisiert,
      - wenn ein Prozeß das Eigentumsrecht überträgt
      - bei Erhalt einer Invalidierungsnachricht
      - bei Erhalt einer angeforderten Seite
      - bei Weitergabe einer Anfrage (auf Anfragenden)



Designalternativen ...

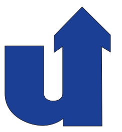
- Probleme: z.B. *Thrashing*, insbes. durch *False Sharing*
  - einfache Abhilfe:
    - erneute Migration der Seite erst nach bestimmter Zeit
  - TreadMarks: *Multiple Writer*-Protokoll
    - Freigabekonsistenz; bei Freigabe werden nur geänderte Teile der Seite übertragen
    - Änderungen werden dann in „zusammengemischt“
    - bei Konflikten: Ergebnis ist nichtdeterministisch



Verteilte Systeme

SoSe 2018

11 Fehlertoleranz



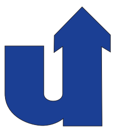
Inhalt

- Einführung
- Prozeßelastizität
- Zuverlässige Kommunikation
- Wiederherstellung

Literatur

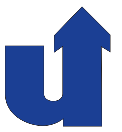
- Tanenbaum, van Steen: Kap. 7

11.1 Einführung



Begriffe

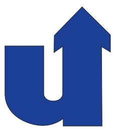
- **Ausfall**: System kann seine Zusagen nicht mehr einhalten
- **Fehler**: Ursache des Ausfalls
  - Fehler kann transient, periodisch oder permanent sein
- **Fehlertoleranz**: System fällt trotz Fehler nicht aus
- Anforderungen an verlässliche Systeme:
  - **Verfügbarkeit**: p(System funktioniert zum Zeitpunkt t)
  - **Zuverlässigkeit**: p(System funktioniert im Zeitintervall Δt)
  - **Sicherheit**: kein größerer Schaden, wenn System ausfällt
  - **Wartbarkeit**: Aufwand zur „Reparatur“ nach einem Ausfall



Fehlermodelle

Absturzfehler	Server wird unterbrochen
Auslassungsfehler	Server reagiert nicht auf Anfragen
Empfangsauslassung	Server erhält keine eingehenden Anfragen
Sendeauslassung	Server sendet keine Nachrichten
Timing-Fehler	Antwortzeit ausserhalb der Spezifikation
Antwortfehler	Antwort des Servers ist falsch
Wertfehler	Nur der Wert der Antwort ist falsch
Zustandsübergangsf.	Inkorrekter Steuerfluss im Server
Byzantinischer Fehler	Zufällige Antworten zu beliebigen Zeiten

- Weitere Unterscheidung: kann der Client den Fehler erkennen oder nicht?

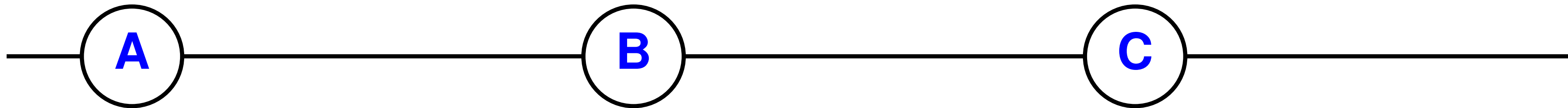


Fehlermaskierung durch Redundanz

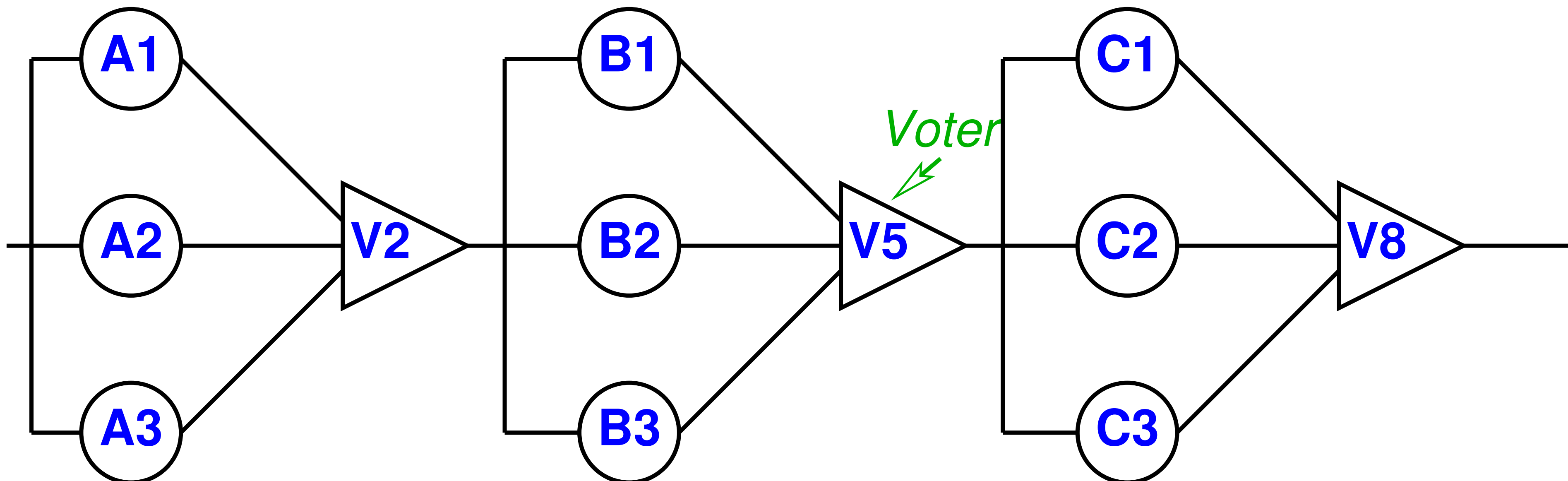
- Fehlertolerantes System muß Fehler vor anderen Prozessen verbergen
- Wichtigste Technik dazu: **Redundanz**
  - **Informationsredundanz**: zusätzliche „Prüfbits“ (z.B. CRC)
  - **zeitliche Redundanz**: Wiederholung fehlerhafter Aktionen
  - **physische Redundanz**: mehrfaches Vorhalten wichtiger Komponenten
- Beispiel: **TMR, *Triple Modular Redundancy***
  - Komponenten werden dreifach repliziert
  - Mehrheitsentscheid der Ergebnisse
  - schützt vor Ausfall einer replizierten Komponente

Beispiel zu TMR

Ohne Redundanz

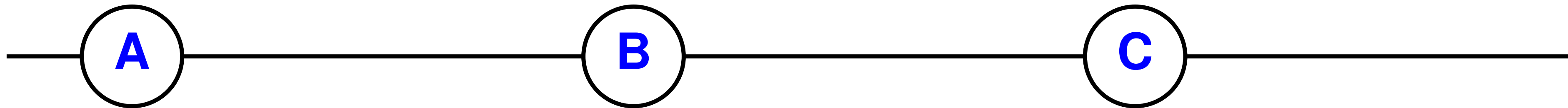


Mit TMR

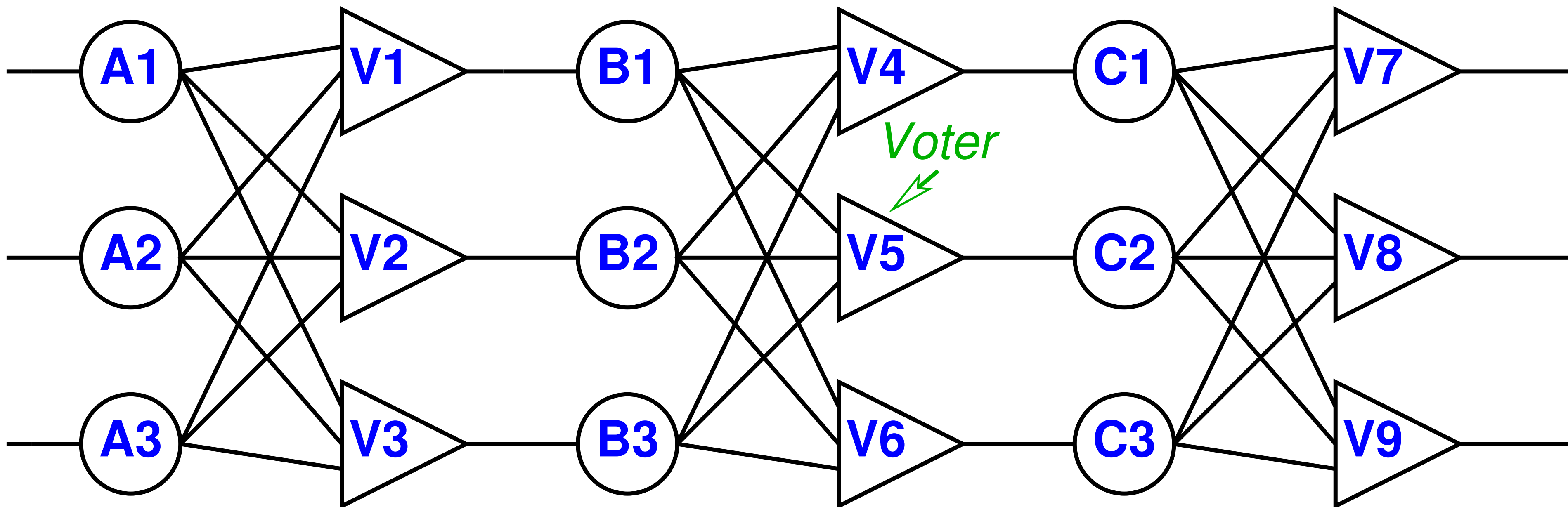


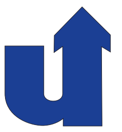
Beispiel zu TMR

Ohne Redundanz



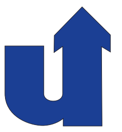
Mit TMR





Ziel: Schutz gegen Prozeßfehler

- Durch Replikation von Prozessen in Gruppen
  - Nachricht an die Gruppe wird von allen Mitgliedern empfangen
    - i.a. vollständig sortierter Multicast
- Fragstellungen:
  - Organisation der Gruppen?
    - flach (symmetrisch) vs. hierarchisch (zentraler Koordinator)
    - Gruppenverwaltung, synchroner Beitritt / Austritt
  - notwendige Anzahl der Replikate?
    - **k-fehlertolerant**: Ausfall von k Prozessen kompensierbar
    - bei stillschweigendem Ausfall: $\geq k+1$ Prozesse
    - bei byzantinischen Fehlern: $\geq 2k+1$ Prozesse
  - Einigung in fehlerhaften Systemen?



Einigung in fehlerhaften Systemen

- Einigung ist mit unzuverlässiger Kommunikation unmöglich
  - Zwei-Armeen-Problem
- Einigung fehlerhafter Prozesse bei zuverlässiger Kommunikation
  - Problem der byzantinischen Generäle
  - Einigung nur möglich, wenn $> \frac{2}{3}$ der Prozesse korrekt arbeiten

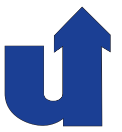
1. Sende Informationen

2. Empfangene Information

3. Sende empfangene Information an alle

2. Empfangene Information		1 erhält:	2 erhält:	4 erhält:
1: (1, 2, x, 4)	von 1:		(1, 2, x ,4)	(1, 2, x ,4)
2: (1, 2, y, 4)	von 2:	(1, 2, y ,4)		(1, 2, y ,4)
3: (1, 2, 3, 4)	von 3:	(a, b, c, d)	(e, f, g, h)	(i, j, k, l)
4: (1, 2, z, 4)	von 4:	(1, 2, z ,4)	(1, 2, z ,4)	

11.3 Zuverlässige Kommunikation

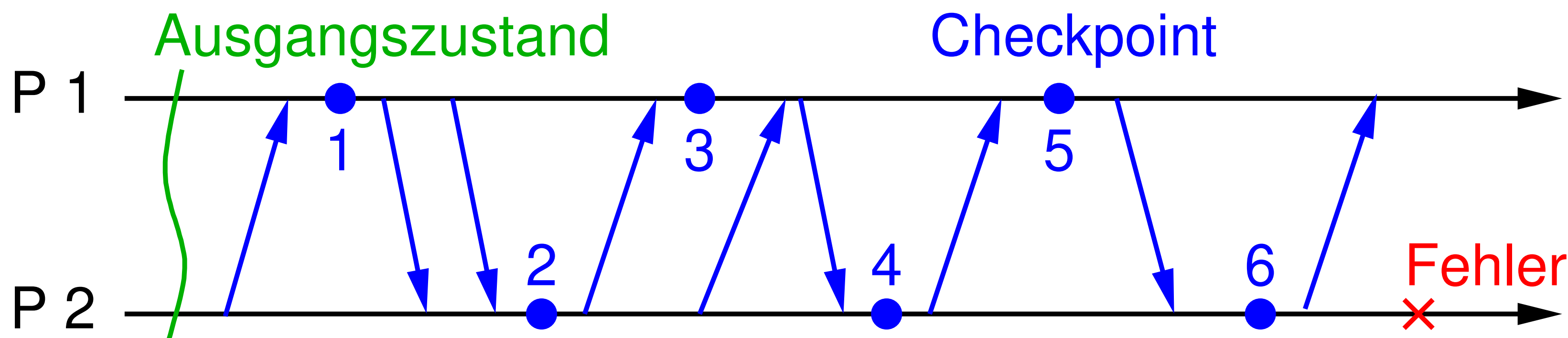


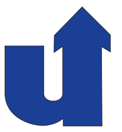
Ziel: Schutz gegen Kommunikationsfehler

- Punkt-zu-Punkt-Kommunikation (☞ **RN_I**)
  - TCP maskiert Auslassungsfehler, aber keine Absturzfehler
- Client/Server-Kommunikation (☞ **2.1**)
  - mögliche Fehler:
    - Server nicht gefunden
    - verlorene Anfrage
    - Server-Absturz während der Bearbeitung der Anfrage
    - verlorene Antwort
    - Client-Absturz nach Senden der Anfrage
- Gruppenkommunikation (☞ **7.3**)
- Verteilter Commit (☞ **7.4**)

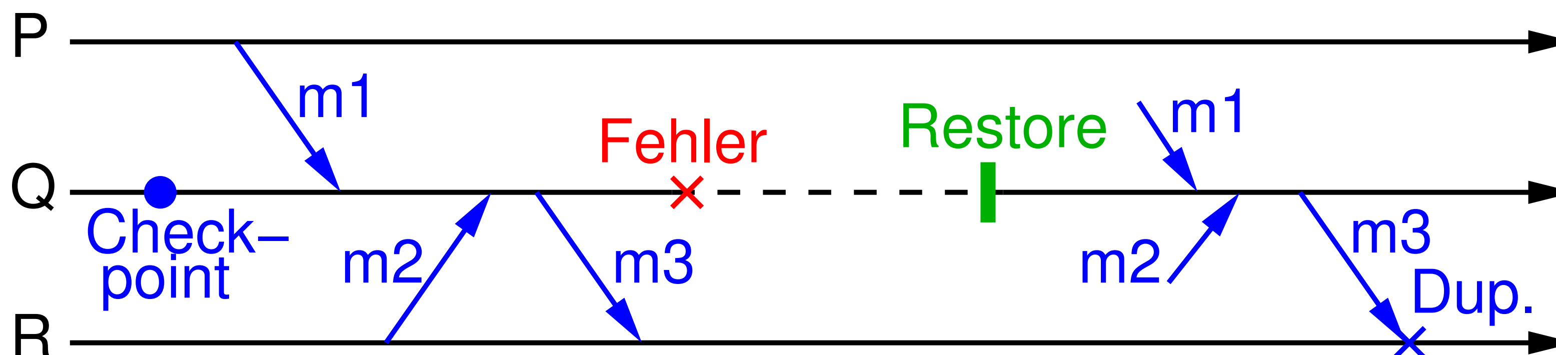
Ziel: Wiederherstellung des Systems nach einem Fehler

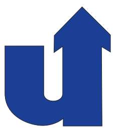
- Vorwärts-Wiederherstellung: gehe in korrekten neuen Zustand
- Rückwärts-Wiederherstellung: gehe in korrekten früheren Zustand
  - d.h. Rücksetzen auf konsistenten Schnitt
  - regelmäßiges Sichern in stabilen Speicher (*Checkpointing*)
- Unabhängige Checkpoints
  - Prozesse sichern Zustand unabhängig voneinander
  - Problem: Domino-Effekt





- Koordinierte Checkpoints
  - Chandy-Lamport-Algorithmus (☞ **6.4**)
  - Alternativ: blockierendes 2-Phasen-Protokoll
  - Problem: Rücksetzen aller Prozesse nötig
- Lokale Checkpoints mit Nachrichtenprotokollierung
  - Ziel: abgestürzten Prozeß in Zustand wiederherstellen, der konsistent mit aktuellem Zustand der anderen Prozesse ist
  - Rücksetzen auf letzten Sicherungspunkt und Wiedereinspielen empfangener Nachrichten



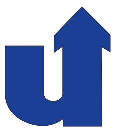


Verteilte Systeme

SoSe 2018

12 Zusammenfassung, wichtige Themen

12 Zusammenfassung, wichtige Themen ...



1. Einführung

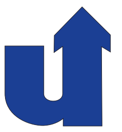
- Definition eines verteilten Systems
- Eigenschaften / Herausforderungen verteilter Systeme
- Architekturmodelle: Client/Server, n-Tier

2. Middleware

- Aufgaben der Middleware
- Kommunikationsorientierte und anwendungsorientierte Middleware
- Realisierung entfernter Aufrufe (Proxy-Muster)

3. Verteilte Programmierung mit Java RMI

- Vorgehensweise bei der Erstellung einer RMI-Anwendung
- Programmierung von Server und Client



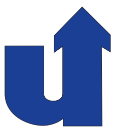
4. Namensdienste

5. Prozeß-Management

- Graphpartitionierung, List-Scheduling, Codemigration

6. Zeit und globaler Zustand

- Synchronisation physischer Uhren
- Lamport'sche Kausalitätsrelation
- Lamport- und Vektor-Uhren
- Konsistente Schnitte, Chandy/Lamport-Algorithmus

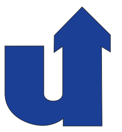


7. Koordination

- Wahl-Algorithmen
- Wechselseitiger Ausschluss (zentral, Ricart/Agrawala, Ring)
- Multicast (Zuverlässigkeit, Reihenfolge)
- Transaktionen

8. Replikation und Konsistenz

- Konsistenzbegriff
- Sequentielle Konsistenz, Freigabe-Konsistenz
- Konsistenzprotokolle (Primary-based, Quorum-basiert)



9. Verteilte Dateisysteme

10. Verteilter Gemeinsamer Speicher

11. Fehlertoleranz

- Fehlermodelle
- Physische Redundanz, Einigung
- Wiederherstellung