Verteilte Systeme

Jürgen Kleinöder

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.cs.fau.de

Sommersemester 2013

http://www4.cs.fau.de/Lehre/SS13/V_VS

Überblick

Kontakt

Dozenten

- Jürgen Kleinöder
- Tobias Distler

Übungsbetreuung

- Tobias Distler
- Christopher Eibel
- Timo Hönig
- Klaus Stengel

Zielsetzung

- Beantwortung der Frage: „Was macht ein verteiltes System eigentlich zu einem Verteilten System?“
  - Ein Verteiltes System ist mehr als die Vernetzung von Rechnern
  - Rechnernetze sind nicht Thema der Veranstaltung
- Vermittlung der Grundlagen und der elementaren Problemstellungen Verteilter Systeme
- Verteilte Systeme aus „Systemsicht“ (Abstraktionen, Ressourcen, ...)
  - Erweiterung des Betriebssystembegriffs in Richtung Verteilte Systeme

Lernziele

- Verständnis der grundlegenden Problemstellungen und ihrer Lösungsansätze
- Einordnung der verschiedenen Kommunikationskonzepte und -mechanismen
- Kennenlernen gängiger Systemarchitekturen
- Tieferes Verständnis für ausgewählte Problemstellungen in Verteilten Systemen, z.B.:
  - Behandlung von Zeit
  - Synchronisation in Verteilten Systemen
  - Unterschiedliche Sichtweise von Prozessen auf den „aktuellen“ Systemzustand
  - Umgang mit Fehlern
  - Replikation

... Lernziele

- Praktische Erfahrungen mit der Lösung ausgewählter Problemstellungen
  - Entwicklung eines Fernaufrufsystems von Grund auf
  - Realisierung ausgewählter Algorithmen für
    - verteilte und
    - fehlertolerante Systeme

Inhalt – Vorlesung (Teil A: Grundlagen von VS)

- Bestandsaufnahme, Beispiele Verteilter Systeme, Problembereiche
- Eigenschaften
  - Physikalische/logische Verteiltheit
  - Heterogenität, Nebenläufigkeit, Fehlerverarbeitung
  - Sicherheit, Offenheit, Skalierbarkeit, Transparenz
- Architekturen Verteilter Systeme
- Interprozesskommunikation und Fernaufrufe
  - Nachrichtenaustausch
  - IPC-Semantiken und -varianten
  - Fernaufrufe – Kommunikation und Semantikaspekte
  - Fernaufrufe – Parameterübergabe, Nachrichtenerstellung, Realisierungsaspekte

... (Teil B: Middleware und Verteilte Algorithmen)

- Verteilte Anwendungen und Middleware
- Zeit in Verteilten Systemen
  - Logische Uhren
  - Uhrensynchronisation
- Verteilte Algorithmen
  - Synchronisation und gegenseitiger Ausschluss
  - Wahlverfahren
  - Multicast Kommunikation
- FT-CORBA
  - Middleware und Replikationskonzepte
- Verteilte Algorithmen für fehlertolerante Programme
  - Unzuverlässige und zuverlässige Verbindungen
  - Ausfallerkennung
  - Synchrone/asynchrone Systeme

Inhalt – Übung

- Teil A: Fernaufrufsystem

  Implementierung eines Java-RMI-ähnlichen Systems

  - RMI als Anwender ausprobieren
  - Serialisierung in Java
  - Threads und Synchronisierung in Java
  - (Dynamische) Generierung von Proxies
  - Rückruf/Callback
  - RPC-Semantiken

- Teil B: Verteilte Algorithmen
  - Basisabstraktionen für verteilte Algorithmen
  - Implementierung einfacher verteilter Algorithmen

Vorlesungsbetrieb

- Vorlesungstermin
  - Montag 12 - 14 oder Mittwoch 12 - 14 (noch festzulegen)
  - Ort: 0.031-113
- Foliensatz
  - Ausdrucke werden in der Vorlesung zur Verfügung gestellt
  - außerdem über die WWW-Seite der Veranstaltung abrufbar

Vorlesungsbetrieb

- Rückmeldungen und Fragen
  - Geben Sie uns Rückmeldungen über den Stoff.
    Nur so kann eine gute Vorlesung entstehen und gut bleiben.
  - Stellen Sie Fragen!
  - Machen Sie uns auf Fehler aufmerksam!
  - Nutzen Sie auch außerhalb der Vorlesung die Möglichkeit,
    uns anzusprechen:
    persönlich (Zimmer 0.041 / 0.043 im RRZE-Gebäude, Martensstr. 1)
    E-Mail {distler, jk}@cs.fau.de

Übung

- Übungstermin
  - Übungsbeginn ist in der Woche ab **22.04.2013**
  - Tafelübung: Dienstag, 12:30 - 14:00 Uhr, Raum 0.031-113
  - Rechnerübung: Dienstag, 14:00 - 16:00 Uhr, Raum 02.155-113
- Inhalt der Tafelübungen
  - Ergänzende und vertiefende Informationen zur Vorlesung
  - Hilfestellungen zu den Übungsaufgaben
  - Klärung von Fragen
  - Anmeldung zu den Übungen: Web-Anmeldesystem Waffel
    `https://waffel.informatik.uni-erlangen.de`

Prüfungsmöglichkeiten

- Bachelor und Master Informatik
  - 5 ECTS- oder 7,5 ECTS-Modul in der Vertiefung Verteilte Systeme und Betriebssysteme
- Bachelor IuK
  - 5 ECTS-Modul als „Wahlpflichtmodul aus Katalog für IuK“
- Master IuK
  - 5 ECTS- oder 7,5 ECTS-Modul als „Wahlpflichtmodul aus INF“ in den Schwerpunkten
    - Eingebettete Systeme
    - Kommunikationsnetze
    - Realisierung von Informations- und Kommunikationssystemen
    - Übertragung und Mobilkommunikation
- Bachelor und Master Mechatronik
  - 5-ECTS-Modul in der Modulgruppe „(Verteilte) Eingebettete Systeme“
- Wahlmodul in verschiedenen anderen Studienfächern

Prüfungsmöglichkeiten – Modulvarianten

- 5 ECTS: Vorlesung + Übung
  - erfolgreiche Bearbeitung der abzugebenden Übungsaufgaben
  - mündliche Prüfung über Vorlesungs- und Übungsstoff
- 7,5 ECTS: Vorlesung + erweiterte Übung
  - erfolgreiche Bearbeitung der abzugebenden Übungsaufgaben
  - erfolgreiche Bearbeitung der Zusatzaufgaben
  - mündliche Prüfung über Vorlesungs- und Übungsstoff

Literatur

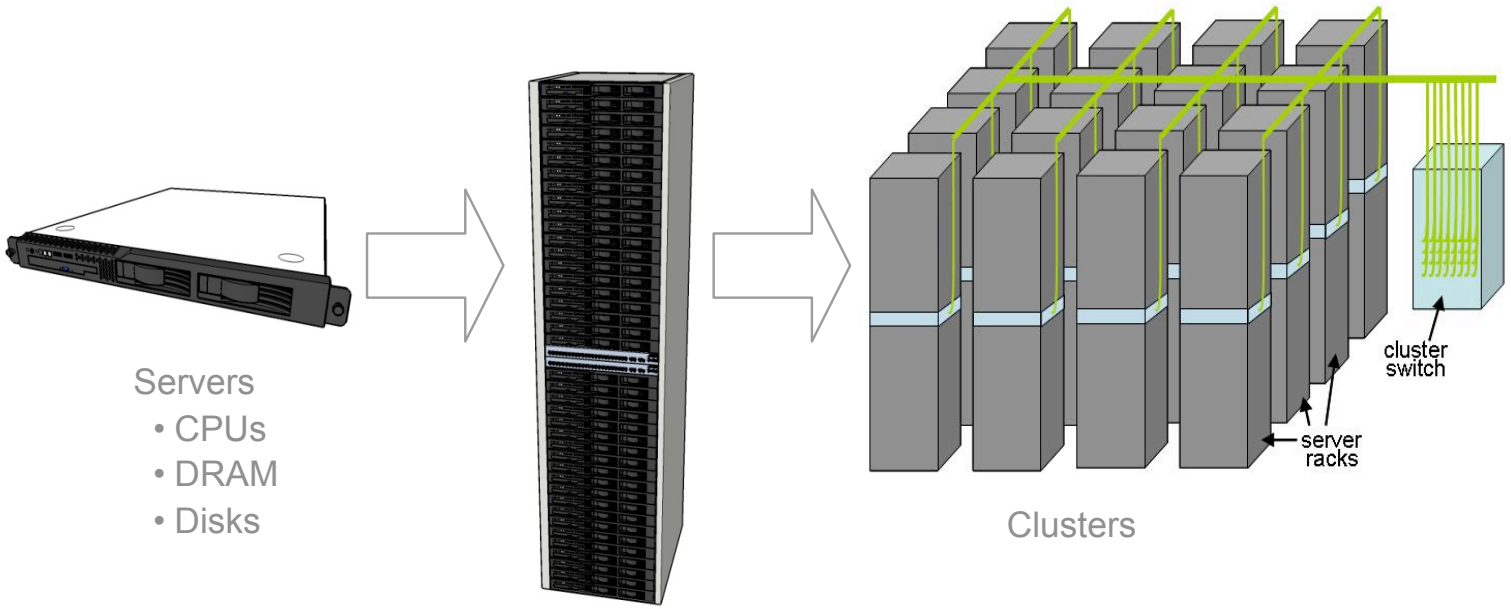
[1] W. M. Gentleman.
Message Passing between Sequential Processes: The Reply Primitive and the Administrator Concept.
Software Practice and Experience, 11(5):435–466, May 1981.

[2] J. H. Saltzer, D. P. Reed, and D. D. Clark.
End-To-End Arguments in System Design.
Transactions on Computer Systems, 2(4):277–288, Nov. 1984.

Überblick

Prozessorfarm @Google

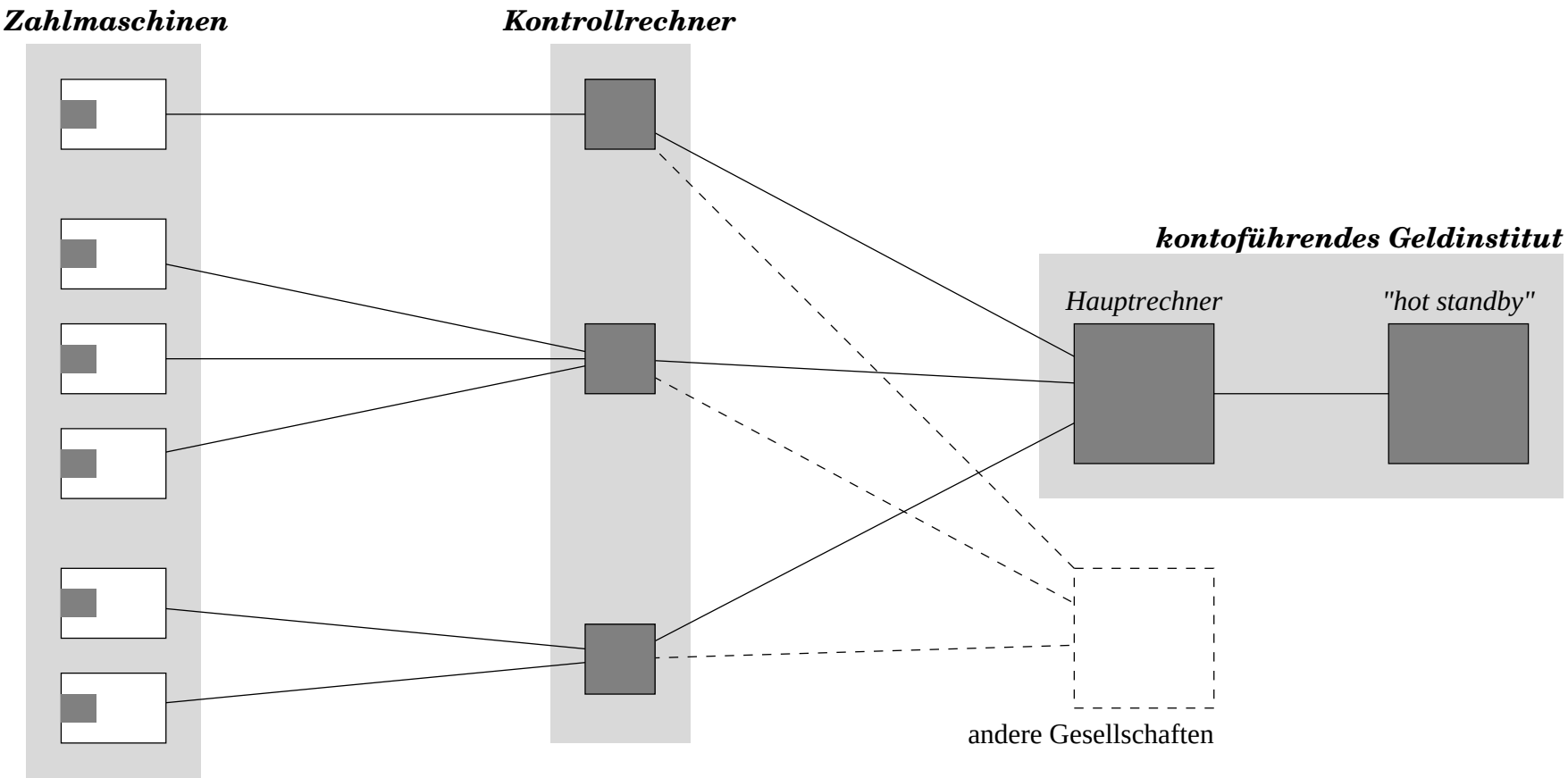
Prozessorfarm @Google

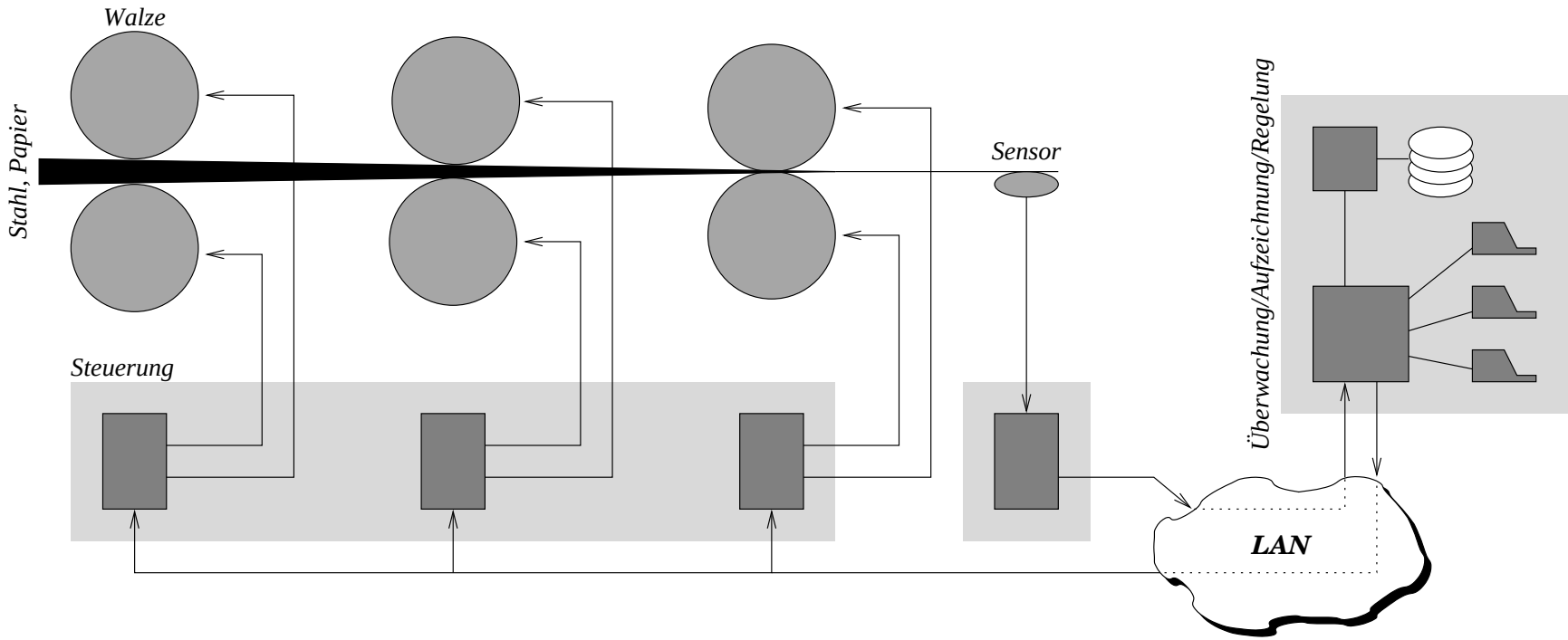


Servers
- CPUs
- DRAM
- Disks

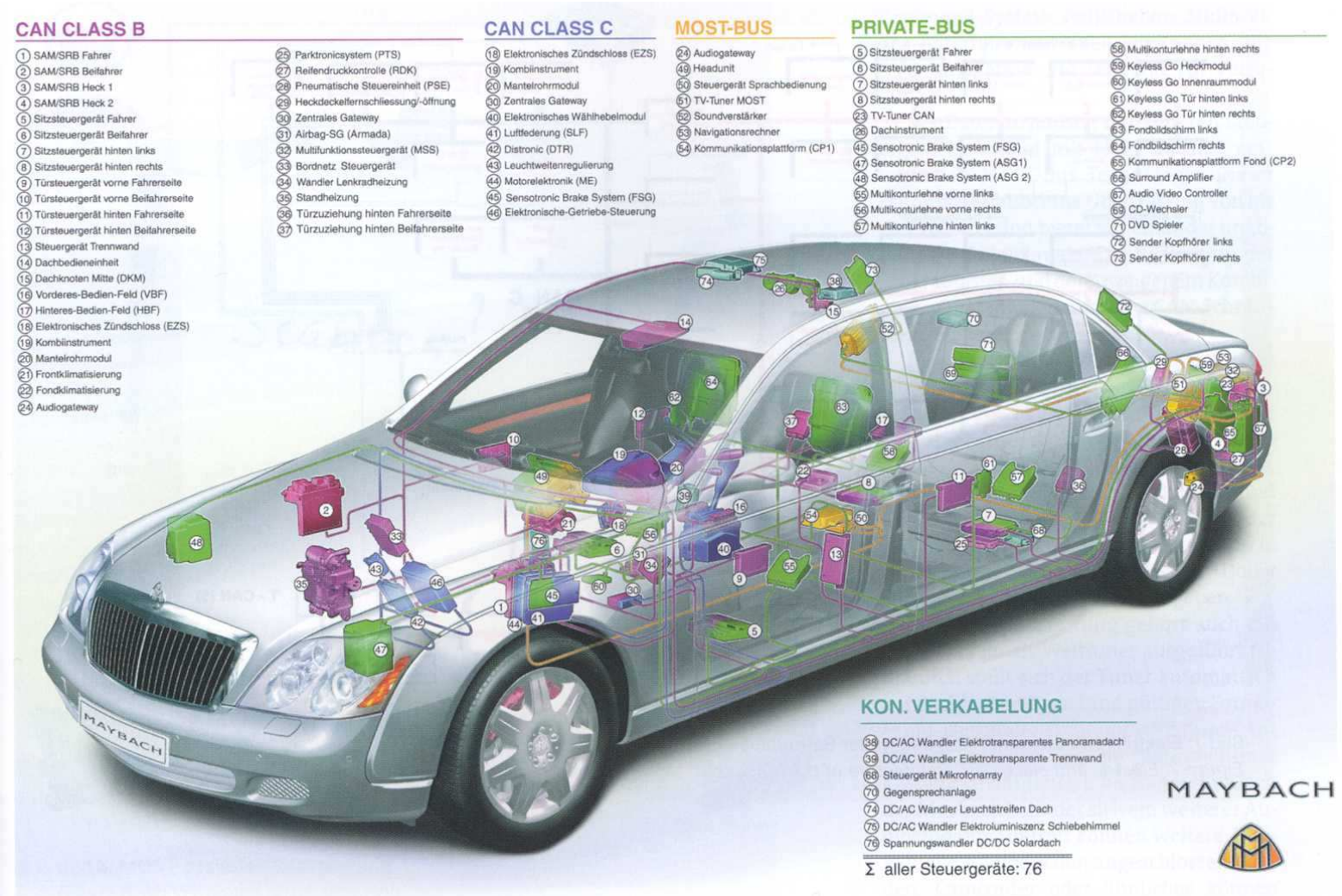
Racks
- 40-80 servers
- Ethernet switch

Clusters


Zahlmaschinen
Kontrollrechner
kontoführendes Geldinstitut
Hauptrechner
"hot standby"
andere Gesellschaften

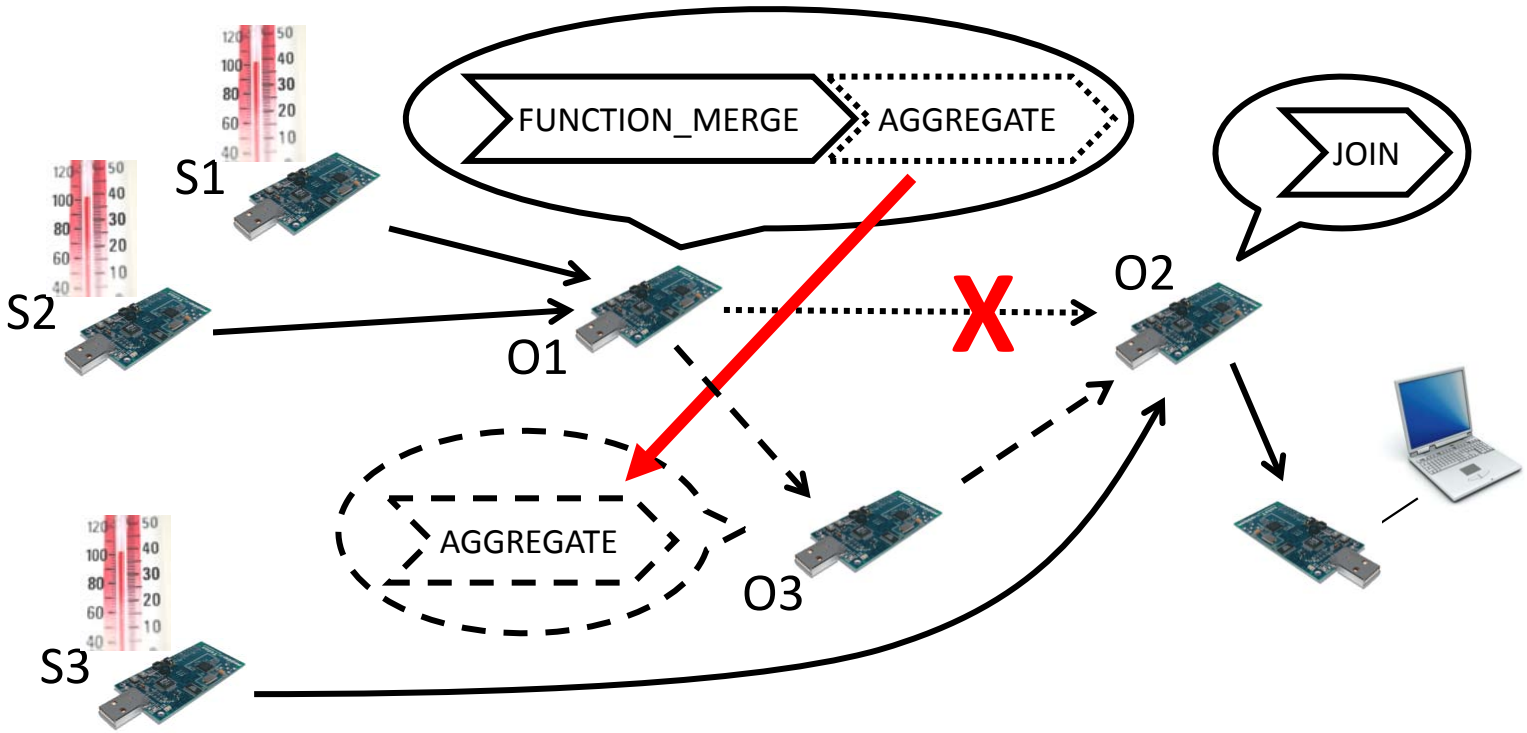

Walze
Stahl, Papier
Sensor
Steuerung
Überwachung/Aufzeichnung/Regelung
LAN

Kraftfahrzeuge



Quelle: [2]

Sensornetzwerke



Anwendungsszenarien (1)

- Ein Benutzer an einem Ort
  - hoch-parallele Anwendung
    – effizentes Rechnen auf vielen Rechnern
  - Beispiele: Simulationen, meteorologische oder aerodynamische Berechnungen
- Ein Benutzer an mehreren Orten
  - Verwendung verschiedener Rechner
  - Wunsch nach homogener Arbeitsumgebung
  - Beispiele: zentrale Datenhaltung (Fileserver), zentraler Terminkalender

Anwendungsszenarien (2)

- Viele Benutzer an vielen Orten
  - Effizienz, Lastverteilung, Verfügbarkeit
  - Überwindung der Orts- und Zeitgrenzen
  - Beispiele:
    - Virtuelle Welten, Mehrbenutzer-Spiele
    - Chat, E-Mail, Videokonferenz
    - E-Commerce, CSCW, weltweite Produktentwicklung

Verteilte Systeme: Merkmale

- Mehrere, unabhängige Rechner
  - können unabhängig voneinander ausfallen
- Verbunden durch ein Netzwerk
  - Interaktion nur durch Nachrichtenaustausch möglich
  - Netzwerk unzuverlässig, mit variablen Nachrichtenverzögerungen, moderate Übertragungsgeschwindigkeit im Vergleich zu Multiprozessor-/Multicoresystemen

  $\Rightarrow$ Unterschied zu Parallelrechnern
- Kooperation der Knoten
  - Beteiligte Knoten interagieren, um gemeinsam eine Aufgabe zu lösen oder einen Dienst anzubieten

  $\Rightarrow$ Unterschied zu einem Rechnernetz

Vorteile

- Rechenleistung vor Ort
  - persönlicher Rechner statt Anschluss an Zentralrechner
  - inhärent verteilte Anwendungen
- Effizienz / Rechenleistung / Skalierbarkeit
  - einfacher Einsatz mehrerer Rechner
  - gutes Verhältnis Kosten zu Effizienz
- Verfügbarkeit und Zuverlässigkeit
  - redundante Auslegung von Komponenten
  - Gesamtsystem bleibt auch bei Ausfall einzelner Komponenten verfügbar

Problembereiche (1)

lokal $\Rightarrow$ entfernt

- Im Falle entfernt ausgelegter Interaktionen sind mehr Fehlerarten möglich als im Falle nur lokal ausgelegter Interaktionen.

direkte $\Rightarrow$ indirekte Bindung

- Konfigurierung wird zu einem dynamischen Vorgang und erfordert Bindungsunterstützung zur Laufzeit.

sequentielle $\Rightarrow$ nebenläufige Ausführung

- Nebenläufigkeit durch Parallelität erfordert Mechanismen zur Koordinierung der Aktivitäten.

Problembereiche (2)

synchrone ⇒ asynchrone Interaktion

- Verzögerungen durch die Kommunikation erfordern Unterstützung für asynchrone Interaktionen und zur Fließbandverarbeitung (*pipelining*).

homogene ⇒ heterogene Umgebung

- Interaktionen zwischen entfernten Systemen erfordern eine gemeinsame Datenrepräsentation.

einzelne Instanz ⇒ replizierte Gruppe

- Replikation kann Verfügbarkeit (*availability*) und/oder Zuverlässigkeit (*dependability*) bereitstellen, erfordert aber auch Maßnahmen zur Konsistenzwahrung.

Problembereiche (3)

fester Platz ⇒ Wanderung

- Die Lage entfernter Schnittstellen (zu Funktionen, Objekten, Komponenten) kann sich zur Laufzeit ändern.

einheitlicher ⇒ zusammengeschlossener Namensraum

- Die Namensauflösung muss (ggf. bestehende) Verwaltungsgrenzen zwischen verschiedenen entfernten Systemen reflektieren.

gemeinsamer ⇒ zusammenhangloser Speicher

- Mechanismen des gemeinsamen Speichers sind nicht (oder nur sehr eingeschränkt) im großen Maßstab anwendbar.

Verteilte Systeme: Anmerkungen und Definition

Leslie Lamport

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Paulo Veríssimo

If you do not need a distributed system, do not distribute.

$\Rightarrow$ Fehlertoleranz von verteilten Systemen ist eine sehr wichtige Eigenschaft, die auch heute noch in vielen Systemen fehlt!

Definition von Andrew Tanenbaum

Ein verteiltes System ist eine Kollektion unabhängiger Computer, die den Benutzern als ein Einzelcomputer erscheinen.

Referenzen

[1] Architecture Projects Management Ltd.
ANSA: An Engineer's Introduction to the Architecture.
Technical Report TR.03.02, Castle Hill, Cambridge, UK, November 1989.
http://www.ansa.co.uk/ANSATech/89/TR0302.pdf.

[2] DaimlerChrysler AG.
Der neue Maybach.
ATZ/MTZ Sonderheft, page 125, September 2002.

[3] Tony King.
Pandora: An Experiment in Distributed Multimedia.
In *Proceedings of Eurographics '92*, Cambridge, UK, September 1992.
http://www.uk.research.att.com/pandora.html.

Überblick

Überblick

- Verteiltheit:
  - physikalisch
  - logisch
- charakteristische Eigenschaften:
  - Heterogenität
  - Nebenläufigkeit
  - Fehlerbehandlung
- wünschenswerte Eigenschaften:
  - Sicherheit
  - Offenheit
  - Skalierbarkeit
  - Transparenz

Physikalische Verteiltheit

. . . der Hardware

bezieht sich auf das verteilte System als Kollektion *unabhängiger, entfernter* Rechner, die über direkte *Leitungen* beliebiger Art oder über *Transportsysteme* zu einem Netz verbunden sind.

- Unabhängigkeit und Entfernung sind wesentliche Ursache für die speziellen *Eigenschaften* eines Verteilten Systems
- Transportsysteme bestehen ihrerseits aus Rechnern und Leitungen
- die Transportsystemrechner dienen der Datenweitergabe (*Vermittlung*)

. . . der Software

spiegelt sich in den Prozessen wider, die auf Grundlage des Rechnernetzes zur Ausführung kommen und eröffnet wichtige Vorteile durch:

- dezentrale Informationsverarbeitung
- gemeinsame Nutzung von Betriebsmitteln
- Erhöhung der Zuverlässigkeit

Logische Verteiltheit

So naheliegend es ist, die physikalische Verteiltheit als Kennzeichen eines verteilten Systems anzusehen, so unklar ist es, wann man ein System als physikalisch verteilt betrachtet und wann nicht. Es drängt sich unwillkürlich die Frage auf, ab welcher Entfernung von Komponenten die Bezeichnung als verteiltes System gerechtfertigt ist. [2]

- Technologiefortschritt bei der Hardware lässt Distanzen schrumpfen
  - gestern noch Rechnernetz, heute ein „*system on chip*“
  - die Aufteilung auf eigenständige Komponenten bleibt
- auch schon eine Verteilung auf mehrere Prozesse auf einem einzelnen Rechner führt zur einer Reihe von typischen Eigenschaften eines Verteilten Systems
- physikalische Verteilung ganz ausser acht zu lassen, wäre jedoch zu voreilig

(Beispielsweise erfordert Zuverlässigkeit physikalisch voneinander getrennte bzw. entfernte Komponenten.)

Gemeinsame Nutzung von Betriebsmitteln

- der Zugriff auf Betriebsmittel kann aus der Ferne erfolgen
  - Betriebsmittel allen Prozessen als *Dienstleistung* zugänglich
    - {Druck, Datenbank, Datei, Web, . . . , Namens}dienst
  - Betriebsmittelverwalter sind dabei selbst Prozesse → **Server**
- die gesamte Betriebsmittelmenge könnte gemeinschaftlich verwaltet werden

  So wie es in einem konventionellen System für die Durchführung eines Prozesses z.B. unerheblich ist, welchen Speicherplatz er zugewiesen bekommt, kann es nun vollkommen egal sein, auf welchem Rechner er ausgeführt wird.

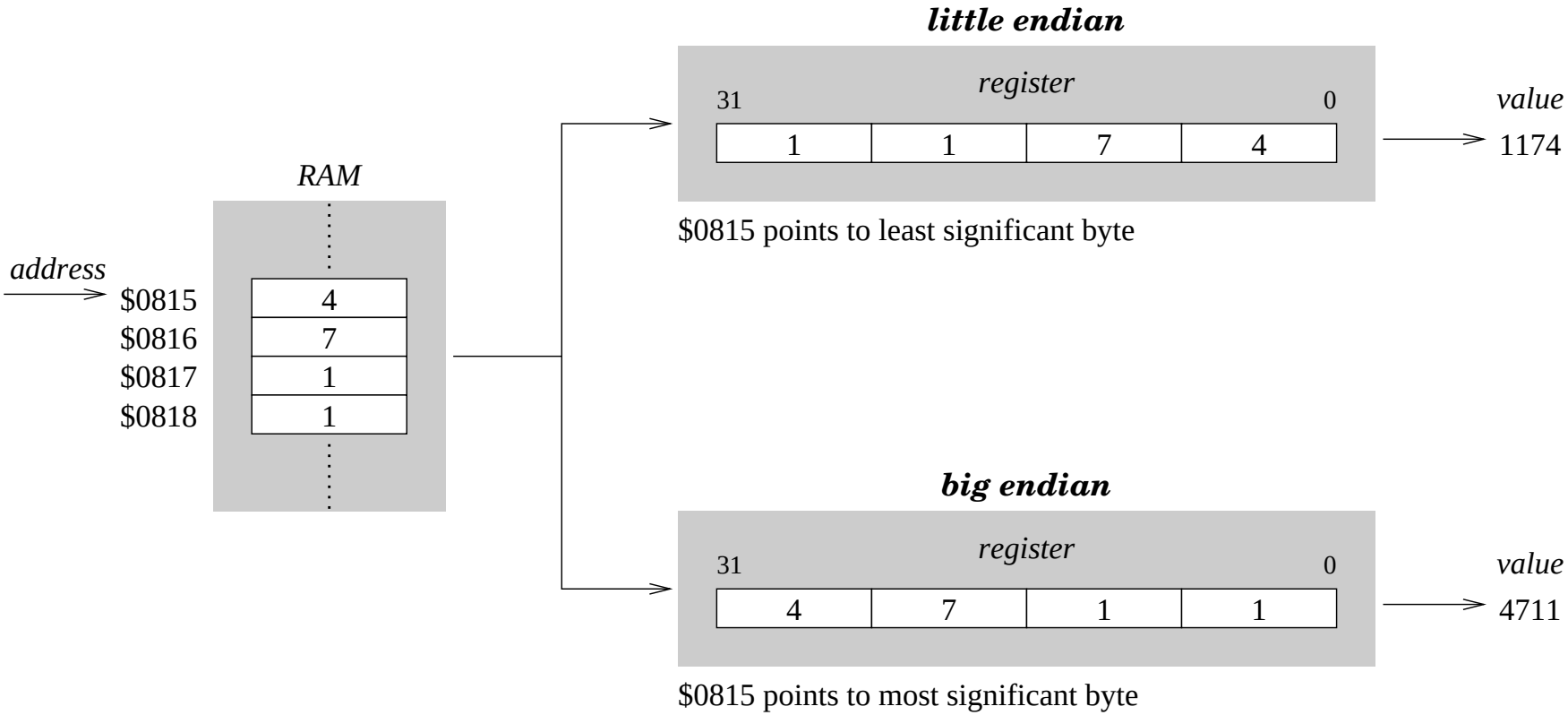
- die Betriebsmittelvergabe erweitert sich um eine geographische Komponente

Heterogenität

Heterogenität ist in folgenden Bereichen vorzufinden:

- Netzwerke . . . . . Anschlusstyp, Medium, Technik, Topographie
- Rechner-Hardware . . . . . Peripherie, Speicher, CPU . . .
- Prozessoren . . . . . Informationsdarstellung, „Byte Sex“
- Betriebssysteme . . . . . Ausführungsumgebung, API
- Programmiersprachen . . . . . Semantik, Pragmatik
- Implementierungen durch verschiedene Personen . . . . . Standards

Heterogenität „Byte Sex“



Heterogenität

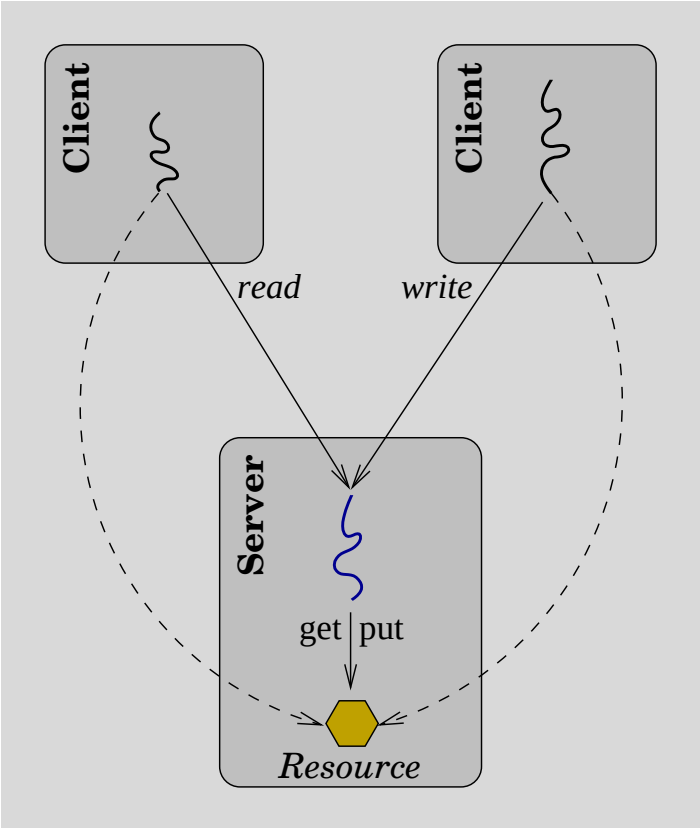
- Softwareschicht zur Abstraktion von den jeweiligen Systemeigenheiten
  - Programmiersprachen $\begin{cases} \text{unabhängig} & \rightarrow \text{CORBA / Web Service} \\ \text{abhängig} & \rightarrow \text{Java RMI} \end{cases}$
- einheitliches Programmiermodell zur Entwicklung verteilter Software
  - Prozedurfernaufruf (*remote procedure call*, RPC [3])
  - Objektfernaufruf (*remote method invocation*, RMI)
  - entfernte Ereignisbenachrichtigung oder SQL-Zugriffe
  - verteilte Transaktionsverarbeitung
- grundlegende Bausteine bilden **Prozesse** und **Botschaftenaustausch**

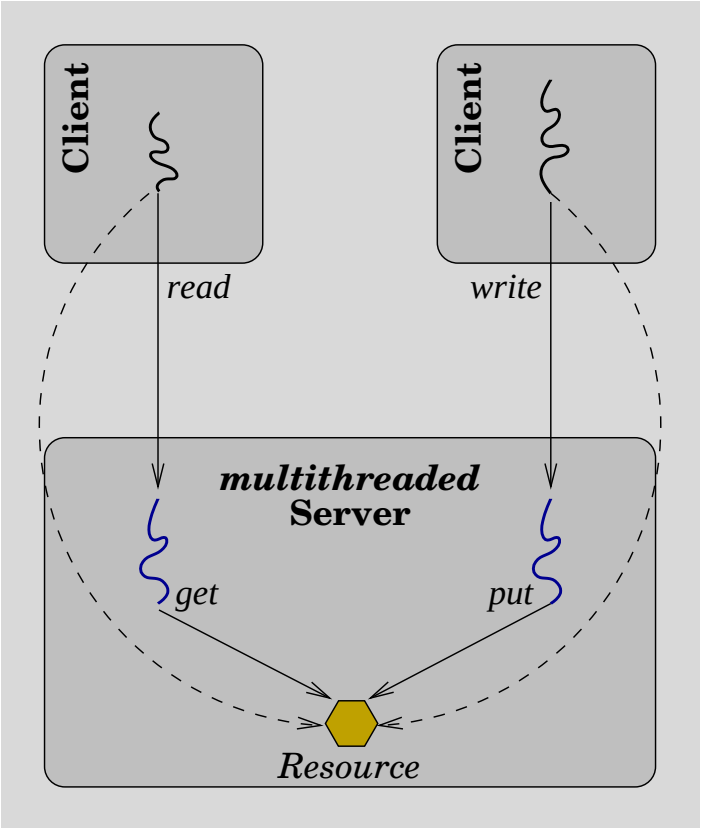
Heterogenität Mobiler Code / Virtuelle Maschine

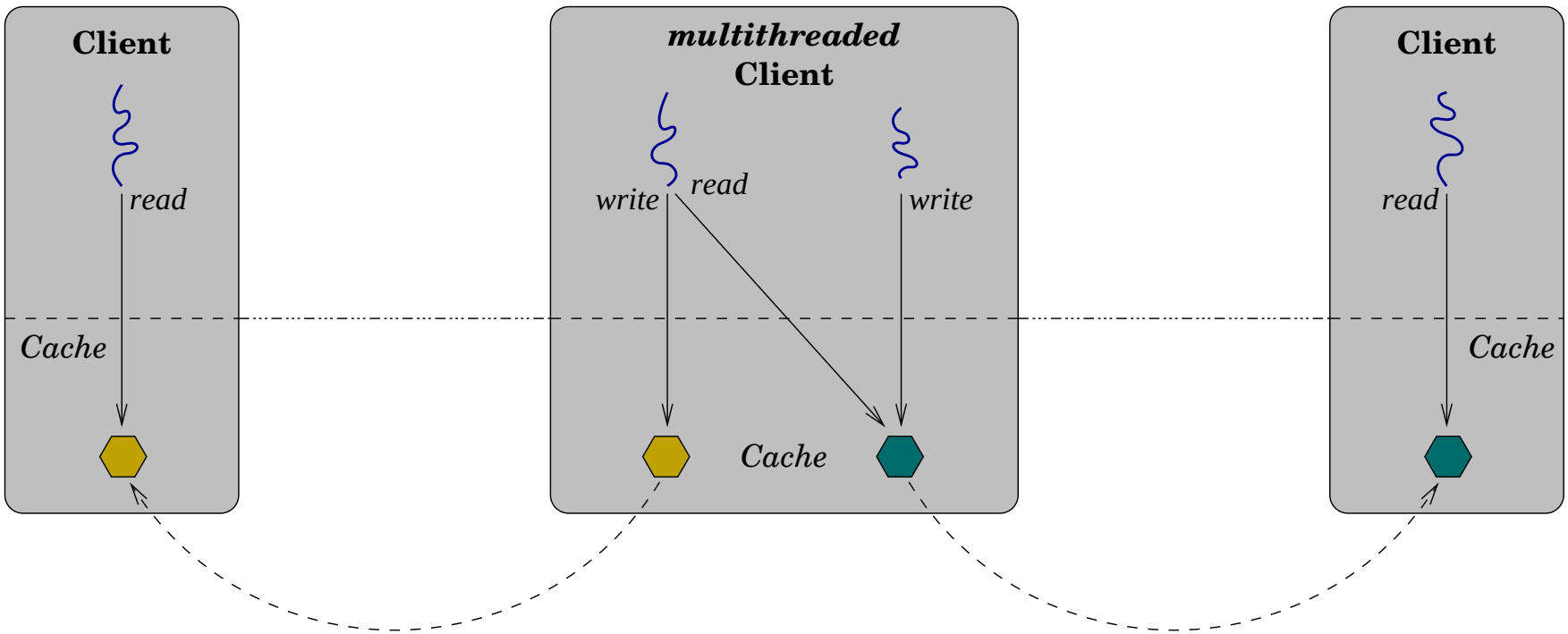
- der Begriff „*mobiler Code*“ bezieht sich auf übertragbaren Maschinencode
  - beispielsweise Java Bytekode
  - oder auch in Form von Schadsoftware (z.B. per Email)
- Anweisungsfolgen übertragbaren Maschinencodes sind Hardware-unabhängig
  - ein Übersetzer erzeugt Zwischencode für eine virtuelle Maschine (VM)
  - die VM ist für jeden Typ Hardware nur einmal implementiert
  - ein Interpreter (d.h. die VM) führt den Zwischenkode aus, nicht die CPU
  - ggf. erfolgt auch eine „*just in time*“ Übersetzung einzelner Komponenten
- der Ansatz ist i.A. abhängig von der Programmiersprache
  Beispiele: Java, C#, ¬ C++

Nebenläufigkeit

- Dienste und Anwendungen stellen im Verteilten System *Betriebsmittel* zur gemeinsamen Nutzung zur Verfügung
  - „gleichzeitige“, sich überlappende Betriebsmittelzugriffe sind hierbei sehr wahrscheinlich
- die nebenläufigen Zugriffe finden auf verschiedenen Ebenen statt:
  1. mehrere Prozesse ($\rightarrow$ **Clients**) benutzen einen Server zum selben Zeitpunkt,
  2. der Server ist mehrfädig ausgelegt, d.h. bedient mehrere Klienten gleichzeitig
  3. und/oder die Betriebsmittel liegen klientenseitig als Replikate vor (*Caching*)
- *Kooperation* der Zugriffe geht (weit) über klassische Semaphorverfahren hinaus
  - Konsistenzwahrung erfordert den Einsatz verteilt arbeitender Algorithmen


Client
Client
read
write
Server
get
put
Resource


Client
Client
read
write
multithreaded
Server
get
put
Resource


Client
read
Cache
multithreaded
Client
write
read
write
Cache
Client
read
Cache

Fehlerbehandlung

- die Wahrscheinlichkeit von Fehlern (in technischen Systemen) ist niemals 0
  - fehlerbedingte Ausfälle in verteilten Systemen sind partiell
    - d.h., einige Komponenten fallen aus, während andere noch funktionieren
  - zur Verarbeitung von Fehlern kommen verschiedene Techniken zum Einsatz:
    - Fehler erkennen, maskieren, tolerieren
    - Wiederherstellung nach Fehlern
    - Redundanz
  - verteilte Systeme bieten einen (relativ) hohen Grad an *Verfügbarkeit*[1]
- „*5-nines*“ (99.999 %) Fehlertoleranz ist eine der großen Herausforderungen

- **Fault (Fehlerursache)** – unerwünschter Zustand, der zu einem Fehler führen kann
- **Error (Fehler)** – Systemzustand, der nicht den Spezifikationen entspricht
- **Failure (Funktionsausfall)** – Diensterbringung ist nicht mehr möglich

Singal/Shivaratri

An error is a manifestation of a fault in a system,
which could lead to system failure.

- Gutmütige Fehler (benign faults)
  - **Crash Stop:** Ein Knoten fällt komplett aus
    - **Fail Stop**: Jeder korrekte Knoten erfährt innerhalb endlicher Zeit vom Ausfall eines Knoten
    - **Fail Silent**: Keine perfekte Ausfallerkennung möglich (asynchrones oder partiell synchrones Modell)
  - **Crash Recovery**: Ein korrekter Knoten kann endlich oft ausfallen und wiederanlaufen
- Bösartige Fehler (malicious faults)
  - Häufig als **byzantinische** Fehler bezeichnet
  - Fehlerhafte Prozesse können beliebige Aktionen ausführen, und dabei auch untereinander kooperieren
  - Modell, das alle beliebigen Arten von Fehler umfasst, z.B. auch gezielte Angriffe von außen auf das System

Fehlerbehandlung

~ erkennen Einige Fehler sind erkennbar (z.B. durch Prüfsummen), andere sind unmöglich zu erkennen (z.B. ein Server-Ausfall). Die Herausforderung ist, mit nicht erkennaber vermutbaren Fehlern umzugehen.

~ maskieren Einige Fehler, die erkannt wurden, können verborgen werden (z.B. verlorene Nachrichten wiederholen, Dateien auf mehrere Datenträger sichern) oder abgeschwächt werden (z.B. fehlerhafte Nachrichten verwerfen). Probleme bereitet der nicht ganz ausschließbare „schlimmste Fall“[2].

~ tolerieren Einige Fehler, die nicht erkannt oder maskiert werden konnten, sind hinzunehmen und ggf. bis hinauf zur Anwendungsebene „hochzureichen“. Software verteilter Systeme soll „fehlergewahr“ sein. Redundanz hilft dabei.

- Rechnerabstürze bzw. Komponentenausfälle zeigen i.A. typische Fehlermuster:
  - Berechnungen von Programmen sind unvollständig
  - permanente Daten befinden sich möglichweise im inkonsistenten Zustand
- grundsätzlich wird dabei zwischen zwei Fehlerarten unterschieden:
  - **transiente Fehler** werden durch ∼smaßnahmen behoben, die zum Ziel haben, einen konsistenten Systemzustand (wieder) zu erreichen
    - *checkpointing*, (*forward*/*backward*) *recovery*, Transaktionen
  - **permanente Fehler** werden durch Reparatur behoben, indem die fehlerhafte Komponente ersetzt oder umgangen wird
- Maßnahmen zur Wiederherstellung sind im Softwareentwurf zu berücksichtigen

- Fehlertoleranz bedeutet „über das Notwendige hinausgehen“ zu müssen:
  - mehr als eine
    { Route zwischen zwei Punkten im Netz vorsehen
    { Serverinstanz (Prozess/Rechner) derselben Art einsetzen
- kritische „Funktionsgruppen“ liegen dazu oft *funktional repliziert* vor
  - redundante Implementierung ein und derselben Einheit
  - realisiert durch Einsatz unabhängiger Entwicklungsteams
- beträchtlicher Mehraufwand steht den (hoffentlich) seltenen Ausfällen gegenüber
  - Fernaufrufe an Servergruppen absetzen — die „richtige“ Antwort auswählen
  - Daten mehrfach speichern — Aktualisierungen überall nachvollziehen

Sicherheit

- den Eigenwert von Informationen für ihre Benutzer zu sichern bedeutet:

Schutz vor	Offenlegung gegenüber Unbefugten	$\rightarrow$	**Vertraulichkeit**
	Veränderung oder Beschädigung	$\rightarrow$	**Integrität**
	Störungen des Betriebsmittelzugriffs	$\rightarrow$	**Verfügbarkeit**

- sensible Informationen sind sicher über ein Netzwerk zu übertragen
  - dabei reicht es nicht, nur den Inhalt der Nachrichten zu verbergen
  - die Identität des Absenders (Benutzer, Agent) ist sicherzustellen
- Verschlüsselungsverfahren helfen, die Authentizität von Klienten zu bestimmen

Offenheit

- offene Systeme
  - zeichen sich durch die Veröffentlichung (der Spezifikation und Dokumentation) der Schnittstellen ihrer „Schlüsselkomponenten“ aus.
- offene verteilte Systeme
  - basieren auf dem Vorhandensein eines einheitlichen Kommunikationsmechanismus und veröffentlichten Schnittstellen für den Zugriff auf gemeinsam genutzte Betriebsmittel;
  - können aus heterogener Hard- und Software aufgebaut sein, die insbesondere auch von unterschiedlichen Herstellern stammen können.

Die Herausforderung ist, mit der Komplexität von Systemen zurechtzukommen, die aus vielen Komponenten (unterschiedlicher Herkunft) bestehen.

Skalierbarkeit

Ein System, das als skalierbar bezeichnet wird, bleibt auch dann effektiv, wenn die Anzahl der Ressourcen und die Anzahl der Benutzer wesentlich steigt. *[1]*

- daraus leiten sich folgende Problemfelder für Entwurf und Implementierung ab:
  - Kontrolle { der Kosten für die physischen Betriebsmittel / des Leistungsverlusts
  - Vermeidung
    { von Betriebsmittelerschöpfung (32/128-Bit Internetadressen) / von Leistungsengpässen (Replikation, *Caching*)
- im Idealfall sollte der Zuwachs „transparent“ sein für System und Anwendungen

- Kostenkontrolle
  - Soll ein System mit n Benutzern skalierbar sein, so sollte die Anzahl der physischen Ressourcen für ihre Unterstützung mindestens proportional zu n sein, d.h. $O(n)$.
- Verlustkontrolle
  - Suchalgorithmen (um z.B. Einträge im *Domain Name System*, DNS, zu finden und aufzulösen), die hierarchische Strukturen verwenden, skalieren besser als solche mit linearen Strukturen. Gleichwohl resultiert der Größenanstieg in einen gewissen Leistungsverlust:
    - die Zeit, die für den Zugriff auf eine hierarchische Struktur benötigt wird, ist $O(log\ n)$, wobei n die Größe der Datenmenge darstellt.

    Damit ein System skalierbar ist, sollte der maximale Leistungsverlust nicht höher sein.

Transparenz (1)

- Zugriffstransparenz
  - ermöglicht den Zugriff auf lokale und globale (d.h. entfernte) Betriebsmittel unter Verwendung identischer Operationen. Das betreffende API macht keinen Unterschied zwischen lokalen und entfernten Operationen.
- Ortstransparenz (auch Positions∼)
  - erlaubt den Zugriff auf Betriebsmittel, ohne ihre Position/ihren Ort kennen zu müssen. Beispielsweise erfolgt der Zugriff nicht direkt über Internetadressen, sondern indirekt über Domänennamen (DNS).

Beide werden unter dem Begriff **Netzwerktransparenz** zusammengefasst. So ist z.B. `wosch@informatik.uni-erlangen.de` Netzwerk-transparent. [Warum?]

Transparenz (2)

- Nebenläufigkeits∼
  - erlaubt mehreren Prozessen gleichzeitiges und konfliktfreies Arbeiten mit denselben gemeinsam genutzten Betriebsmitteln.
- Replikations∼
  - erlaubt die Verwendung mehrerer Betriebsmittelinstanzen (d.h. *Replikaten*), um Zuverlässigkeit und Leistung zu verbessern.
- Fehler∼
  - erlaubt den kontinuierlichen Rechnereinsatz trotz des möglichen Ausfalls von Hard- und Software-Komponenten (*non-stop computing*).

Transparenz (3)

- Migrations∼
  - (auch Mobilitäts∼) erlaubt das Verschieben bzw. Wandern von Betriebsmitteln und Prozessen innerhalb eines Systems ohne Beeinträchtigung der laufenden Arbeit von Benutzern bzw. Programmen.
- Leistungs∼
  - erlaubt die Last abhängige Neukonfigurierung des Systems zum Zwecke der Leistungssteigerung.
- Skalierungs∼
  - erlaubt die Vergrößerung (ggf. auch Verkleinerung) des Systems ohne Auswirkungen auf die realisierte Struktur und die zum Einsatz gebrachten Algorithmen.

Unschärfeprinzip

Es ist die Eigenart verteilter Systeme, dass es auf Dauer keine zwei Prozesse gibt, die zur gleichen Zeit die gleiche, zutreffende Sicht des Systems haben. Ein Prozess innerhalb des Systems verfügt entweder über **unvollständige**, **aktuelle** *oder über* **vollständige**, **überholte** *Zustandsinformationen.* *[2]*

Verteilte Systeme: Definition (die 2.) (V. Garg)

Aus Sicht der Fehlertoleranz: [6]

- Keine gemeinsame Uhr
  - Uhrensynchronisation schwierig aufgrund nicht bekannter Verzögerungszeiten bei der Kommunikation, physikalische Uhren werden daher nur selten zur Koordinierung verwendet
- Kein gemeinsamer Speicher
  - Kein einzelner Knoten kennt den vollständigen globalen Zustand. Es ist daher schwierig, globale Eigenschaften des Systems zu beobachten
- Keine akkurate Ausfallerkennung
  - Es ist in einem asynchronen verteilten System (d.h. keine obere Schranke für Kommunikationszeiten bekannt) unmöglich, langsame von ausgefallenen Prozessen zu unterscheiden

Ausklang. . .

Everything should be made as simple as possible, but no simpler.
(Albert Einstein)

You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.
(Antoine de Saint Exupery)

Referenzen

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair.
Distributed Systems: Concepts and Design.
Addison Wesle, fifth edition, 2011.

[2] R. G. Herrtwich and G. Hommel.
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989.
ISBN 3-540-51701-4.

[3] B. J. Nelson.
Remote Procedure Call.
Technical Report CMU–81–119, Carnegie-Mellon University, 1982.

[4] B. Randell, P. A. Lee, and P. C. Treleaven.
Reliability Issues in Computing System Design.
ACM Computing Surveys, 10(2):123–165, June 1978.

[5] D. P. Siewiorek and R. S. Swarz.
Reliable Computer Systems: Design and Evaluation.
A K Peters Ltd., 3rd edition, 1998.
ISBN 15-688-1092-X.

[6] P. Vijay K. Garg.
Elements of distributed computing.
John Wiley & Sons, Inc., New York, NY, USA, 2002.

Überblick

Überblick

- Architekturelemente
  - Kommunikationsteilnehmer
  - Kommunikationsparadigma
  - Rollen und Verantwortlichkeiten
  - Orte
- Architekturmuster
  - Layering (Ebenen)
  - Tiered Architecture (Stufen)
  - Thin Clients
  - Proxies
  - Brocker
  - Reflection

Architekturelemente: Kommunikationsteilnehmer

System-orientierte und Problem-orientierte Sichtweise

- Prozesse: intuitive Systemsicht, führt zu der weit verbreiteten Darstellung von verteilten Systemen als Menge von Prozessen, die über IPC-Mechanismen interagieren
  - ? sehr kleine Systemen ohne ein Prozess-Konzept
  - ? Threads — eigentlich sind nicht die Prozesse, sondern die Threads die Kommunikationsteilnehmer
- Objekte: Problem-orientierte Sicht — in einem verteilten Objekt-basierten System bestehen Anwendungen aus einer Menge verteilter, interagierender Objekte
  - ? Objekte mit mehreren Threads
- Komponenten: Weiterentwicklung des Objekt-Konzepts hin zu Software-Bausteinen
- Web Services: ähnlich zu Komponenten, aber explizit auf das WWW als Kommunikationsplattform ausgerichtet

Architekturelemente: Kommunikationsparadigmen

direkte Kommunikation

- Interprozesskommunikation
  - *low-level*-Mechanismen für Nachrichtenübertragung
- Fernaufruf
  - *Request-Reply-Protokolle*
  - Prozedur-Fernaufruf
  - Methoden-Fernaufruf
  - Details in Kapitel 5

Architekturelemente: Kommunikationsparadigmen

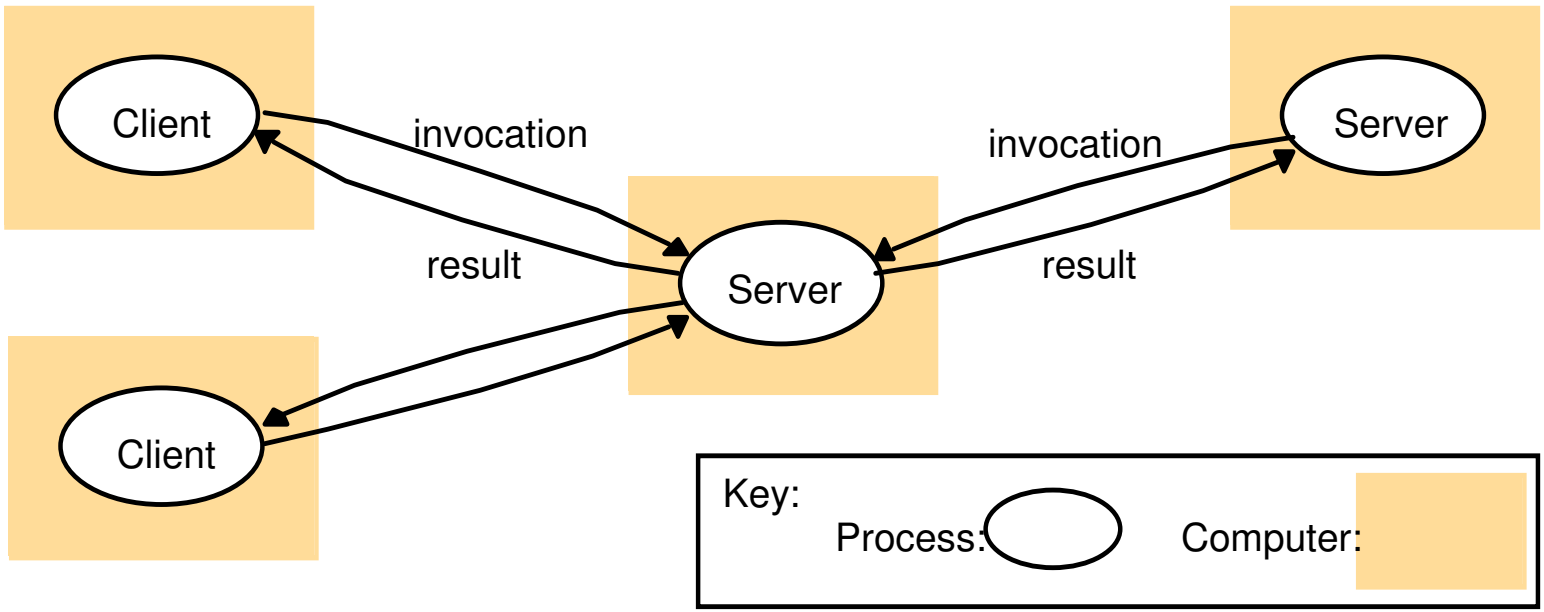
indirekte Kommunikation

- Gruppenkommunikation
  - Abstraktionsmechanismus für mehrere Nachrichtenempfänger
  - Sender kennt nur die *Gruppe*, nicht aber die konkreten Empfänger
  - Basismechanismus für fehlertolerante Systeme
- Publish-subscribe Mechanismen
  - Viele Produzenten (*Publisher*) verteilen Informationseinheiten (*Events*) an eine große Menge von Empfängern (*Subscribers*).
- Message Queues
  - Indirekte 1:1-Kommunikation, entkoppelt über *Briefkasten*
- Tuple Spaces
  - Prozesse platzieren strukturierte Dateneinheiten (*Tuples*) in einem persistenten Speicherbereich (*Tuple Space*), andere Prozesse können die Daten dort lesen oder entnehmen.
  - Leser und Schreiben mussen nicht gleichzeitig existieren.
- Distributed Shared Memory
  - Abstraktion für gemeinsamen Speicher im Verteilten System.

Architekturelemente: Rollen

Client-Server

- üblichste Kommunikationsart in Verteilten Systemen



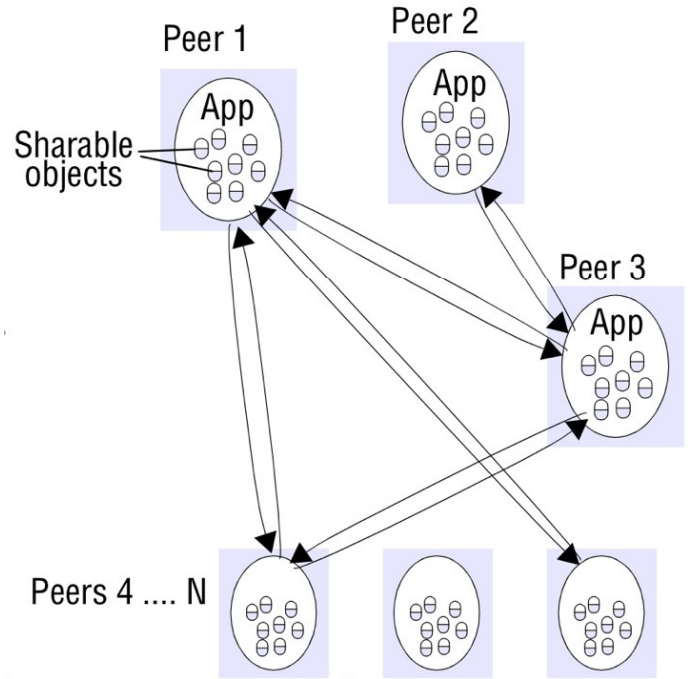
Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

- Prozesse nehmen *Client*- oder *Server*-Rolle ein.
- Server können zur Erbringung ihres Dienstes auch die Dienste weiterer Server in Anspruch nehmen — und werden damit zu Klienten.
- Beispiele: Web-Server, Mail-Server, File-Server, ...

Architekturelemente: Rollen

Peer-to-Peer

- Interaktion gleichberechtigter Prozesse im Netzwerk



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

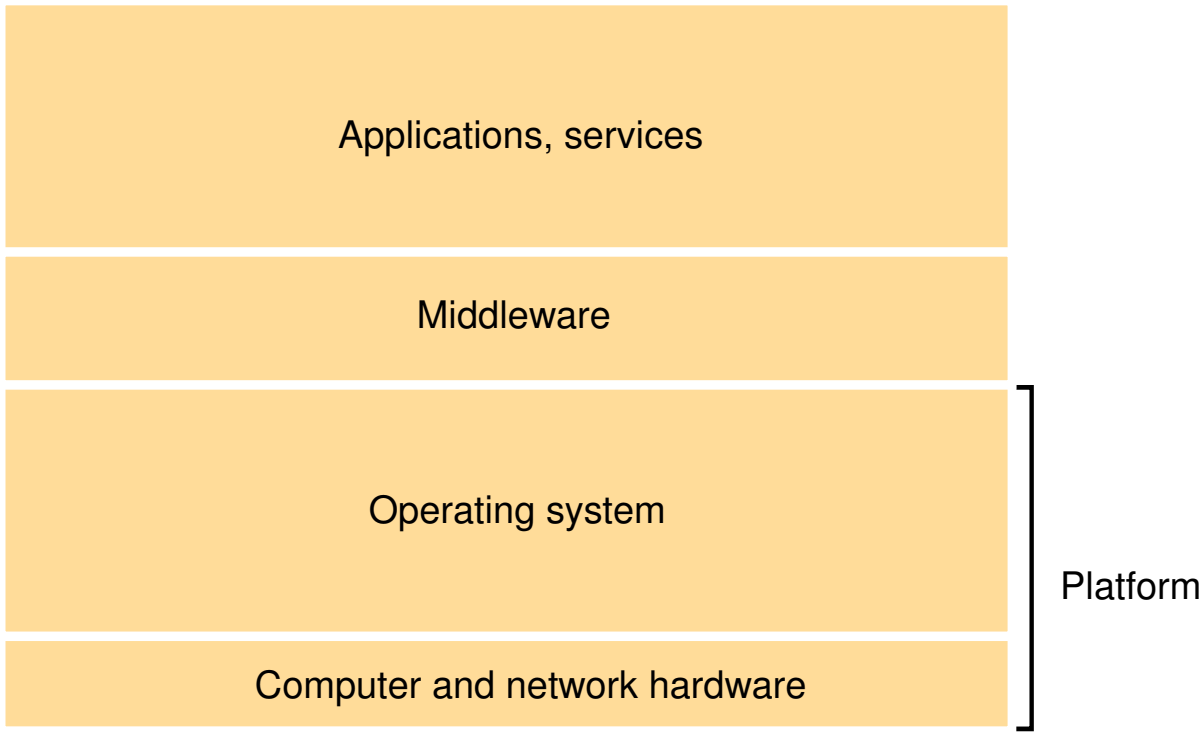
- zentralisierte P2P-Systeme: Server zur Verwaltung vorhanden
- Beispiele: Datei-Tauschbörsen

Architekturelemente: Orte

- Dienste auf mehrere Server abbilden
  - Prozesse auf mehreren Rechnern erbringen gemeinsam einen Dienst
  - Teile des Dienstes können auf verschiedene Server platziert werden
  - Lastverteilung zwischen gleichartigen Servern
- Caching
  - kürzlich benötigte Objekte werden näher am Klienten gespeichert
  - bei Anfragen wird die Aktualität des Objekts überprüft und nur bei Bedarf vom eigentlichen server neu übertragen
- Mobiler Code
  - Code wird vom Server geladen und lokal ausgeführt
  - Beispiele: *Applets*, Javascript
- Mobile Agenten
  - Programme wandern zwischen Rechnern und führen Aufgaben aus
  - Beispiel: Installation und Wartung von Software in einem Firmennetzwerk

Architekturmuster: Ebenen *(Layers)*

- Grundlegendes Abstraktionskonzept
  - jede Ebene nutzt die Dienste der darunterliegenden Ebene
  - jede Ebene bietet nach oben eine Software-Abstraktion an (und verbirgt damit Implementierungsdetails)



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

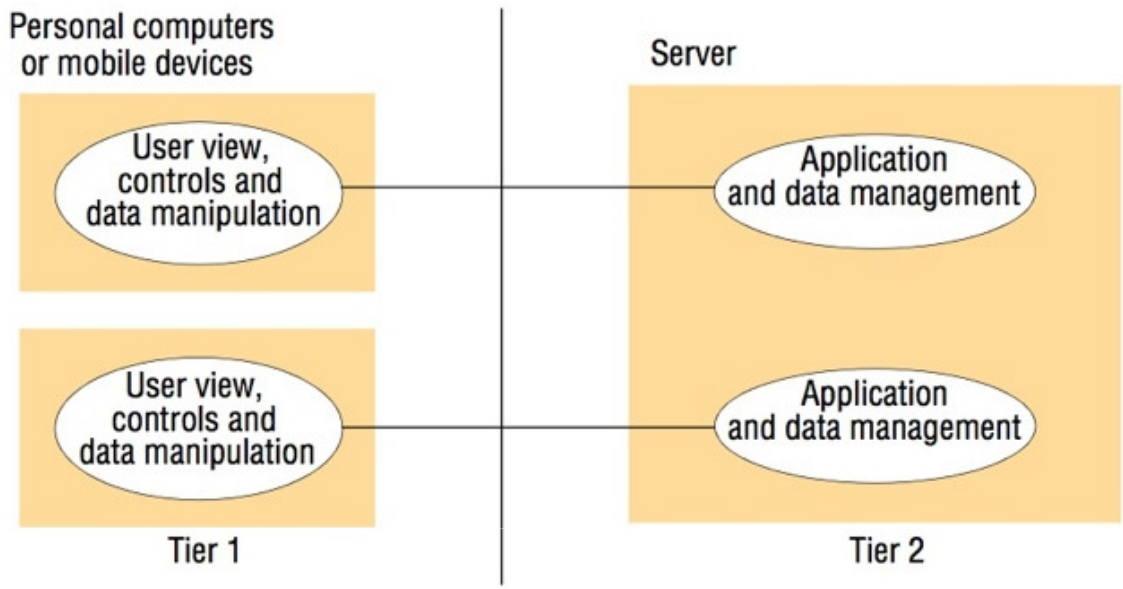
Architekturmuster: Ebenen *(Layers)* (2)

- Plattform für Verteilte Systeme:
  - umfasst die grundlegenden Hardware- und Software-Ebenen
    z. B. Intel x86 / Windows, ARM / Linux
  - bietet einfache Mechanismen zur Kommunikation und Koordination von Prozessen im Verteilten System
    z. B. TCP/IP-Implementierung mit Socket-Schnittstelle

- Middleware
  - verbirgt Heterogenität und bietet Programmierparadigma-gerechte Schnittstellen
  - Plattform für verteilte interagierende Prozesse oder Objekte
  - unterstützt verteilte Anwendungen durch geeignete Abstraktionen und Dienste
    z.B. Nameservice, verteilte Transaktionen, Objektmigration, ...
  - Beispiel: CORBA

Architekturmuster: Stufen *(Tiers)*

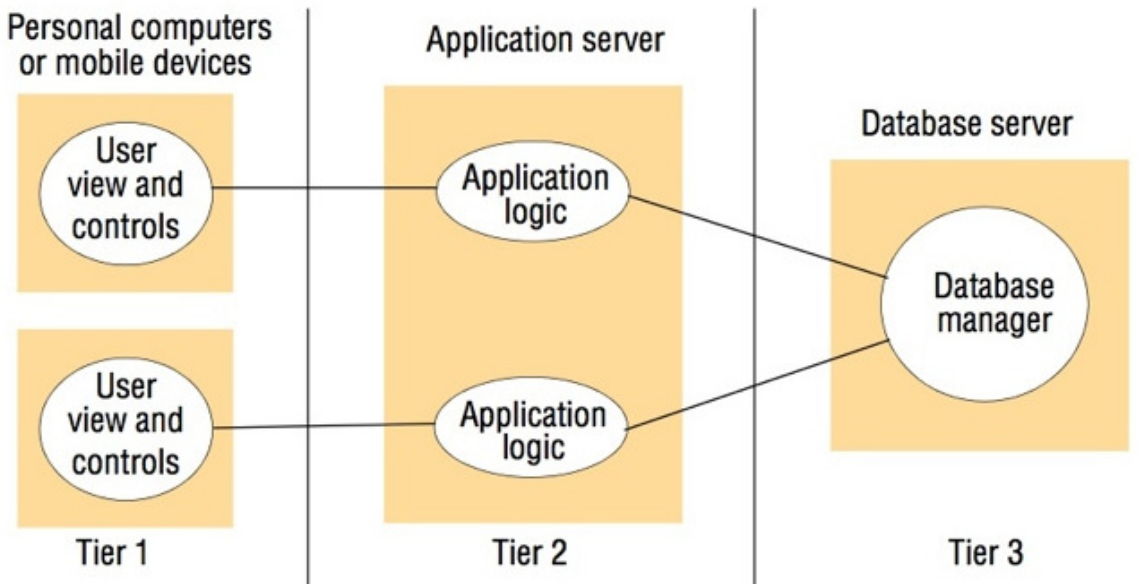
- Komplementär zu dem Konzept der Ebenen
  - Ebenen dienen der vertikalen Strukturierung eines Dienstes in Abstraktionsebenen
  - Organisation in Stufen *(Tiering)* strukturiert die Funktionalität einer Ebene in verschiedene Server, ggf. auf verschiedenen Knoten im Netz



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

Architekturmuster: Stufen *(Tiers)* (2)

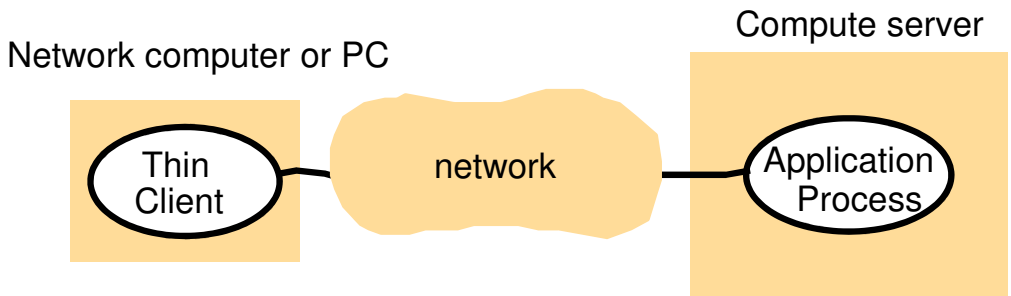
- Typische Struktur einer *three-tier*-Lösung:
  - *Presentation logic*: Benutzerinteraktion und Darstellung der Anwendung
  - *Application Logic (Business Logic)*: Anwendungsspezifische Verarbeitung der Daten (Berechnungen/Algorithmen, Strategien)
  - *Data logic*: Persistente Speicherung der Anwendungsdaten - z. B. in einer Datenbank



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

Architekturmuster: *Thin Clients*

- Verlagerung der Komplexität vom Endgerät des Benutzers auf Dienste im Internet
  - wird vor allem im Bereich *Cloud Computing* deutlich
  - auch in gestuften Architekturen erkennbar



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Architekturmuster: *Thin Clients* (2)

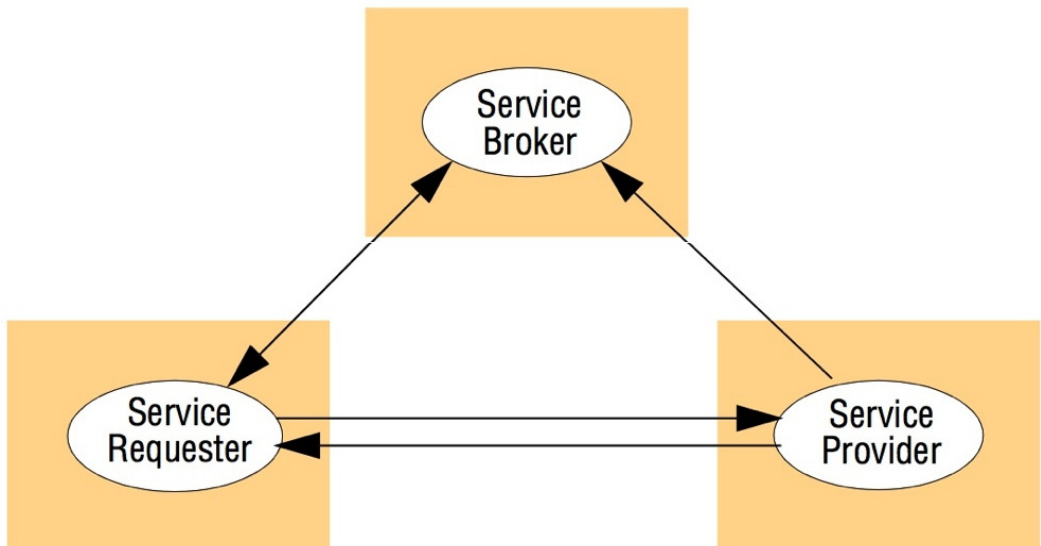
- Vorteil: Einsatz sehr einfacher Nutzer-Endgeräte
  - kostengünstig
  - einfache Installation und Wartung der Software
  - mehrere Abstufungen möglich: kleiner PC, SunRay, VNC-Software
- Funktionalität des Endgeräts wird durch eine Vielzahl von Diensten im Netz erweitert
- Problem: rechenintensive oder graphisch aufwändige Anwendungen (z. B. CAD-Systeme)
- Virtual Network Computing (VNC)
  - lokales System wird nur zur Anzeige der Grafik (auf FrameBuffer-Ebene) genutzt, jede Mausbewegung und Tastatureingabe wird an den Server übertragen

Architekturmuster: *Proxies*

- Grundlegendes Softwaremuster in verteilten Anwendungen, um Ortstransparenz zu erreichen.
  - lokaler Proxy agiert als Stellvertreter für das entfernte Objekt
  - Proxy bietet die gleiche Schnittstelle wie das entfernte Objekt
  - Verteiltheit wird vor dem Aufrufer verborgen
    (nicht ganz: anderes Zeit- und Fehlerverhalten möglich!)
  - Details siehe Kapitel 5
- weitere Anwendungsmöglichkeiten für Proxies
  - Zwischenspeicherung (Caching) von entfernten Objekten
    (Reduzierung der Netzlast, schnellere Antwort)
  - Verbergen von replizierten Objekten
  - spezielle Zugriffskontrollmechanismen

Architekturmuster: *Broker*

- Unterstützung von Interoperabilität in komplexen verteilten Infrastrukturen
  - wie findet ein Dienst-Nutzer den Dienst-Anbieter?
- typische Beispiele:
  - Naming Service in der CORBA-Middleware
  - Registry in Java RMI



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

Architekturmuster: *Reflection*

Reflection unterstützt zwei Konzepte:

- Introspection
  - die Möglichkeit, die Eigenschaften (z.B. Schnittstellen) eines Systems dynamisch (zur Laufzeit) zu untersuchen
  - dynamische Erzeugung von Proxies aus den Schnittstelleninformationen
  - Vermittlung von Aufrufen durch einen generischen Server an die passenden Schnittstellen hintergelagerter Diensterbringer
- Intercession / Interception
  - die Möglichkeit, dynamisch Strukturen und Verhalten zu verändern
  - Abfangen von Aufrufen (für zusätzliche Sicherheitsprüfungen, Optimierungen, Replikation, ...

Überblick

IPC-Grundlagen – Überblick

- Netzwerke – Aspekte für Verteilte Systeme
- Netzwerkgrundlagen
  - Paketvermittlung
  - Daten-Streaming
  - Vermittlungsschemata
  - Protokolle
  - Routing
  - Internetworking
  - Internet-Protokolle

IPC-Grundlagen – Überblick (2)

- TCP/IP-Anwendungsschnittstelle
  - Client-Server-Modell
  - lokale/entfernte Kommunikation
  - Adressierung
  - Kommunikationsarten
  - Datendarstellung
  - Socket-Schnittstelle
  - DNS-Anfragen
  - Datenaustausch

Netzwerke — Aspekte für Verteilte Systeme

Netzwerke sind die Grundlage für Verteilte Systeme.
Ihre Eigenschaften und Fähigkeiten bestimmen wesentlich, wie ein Verteiltes System beschaffen ist (sein kann).

- Leistung
  - Interaktion in Verteilten System basiert im Wesentlichen auf Nachrichtenaustausch
  - welche Parameter beeinflussen die Geschwindigkeit von Nachrichtenzustellungen?
    - Latenz
      Zeit von der Sende-Operation bis erste Daten am Ziel ankommen
    - Transferrate (Bits pro Sekunde)

  => *Nachrichtenübertragungszeit = Latenz + Nachrichtenlänge/Transferrate*

- Skalierbarkeit

Netzwerke — Aspekte für Verteilte Systeme(2)

- Zuverlässigkeit
  - wesentliche Grundlage für Strategien in verteilten Anwendungen
  - ist der Fehlerfall die seltene Ausnahme oder der zu erwartende Normalfall?
- Sicherheit
  - worauf kann sich die Anwendung verlassen, was muss sie selbst absichern?
- Mobilität
  - mobile Geräte sind eine stark wachsende Komponente in Verteilten Systemen
  - Folge: wechselnde Erreichbarkeit verbunden mit Schwankungen bei Leistung, Zuverlässigkeit, ...
  - was kann das transparent für die Anwendung machen, was ist inhärent sichtbar?

Netzwerke — Aspekte für Verteilte Systeme(3)

- Dienstgüte (Quality of Service)
  - erlaubt das Netzwerk die Formulierung von QoS-Anforderungen?
  - wie kann eine Anwendung ihren Bedarf ermitteln und ausdrücken
- Multicast
  - Gruppenkommunikation ist die Basis für viele Mechanismen in Verteilten Systemen (-> Fehlertoleranz)
  - welche Mechanismen für Broadcast oder Multicast unterstützt bereits das Netzwerk?

Netzwerkgrundlagen

Grundlegende Technik in Rechnernetzen ist *Paketvermittlung*. Kommunikation erfolgt *asynchron* – d. h. Nachrichten erreichen ihr Ziel nach einer (variierenden) Verzögerungszeit

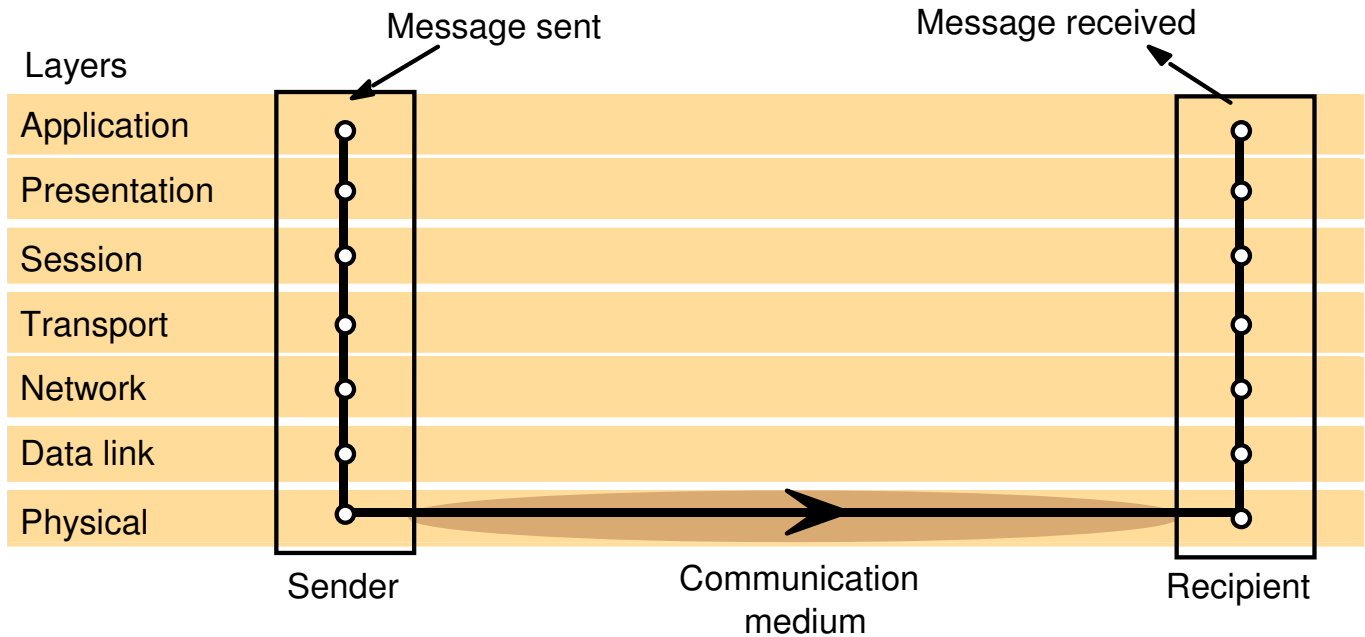
- Paketvermittlung
  - Nachrichten werden Datenpakete begrenzte Länge zerlegt
  - Pakete werden mit Adressinformationen versehen und über dasNetz übertragen
- Daten-Streaming
  - nicht alle Anwendungen in Verteilten Systemen basieren auf dem Austausch von Nachrichten
  - wichtigste Ausnahmen: Audio- und Video-Daten
  - spezielle QoS-Anforderungen: Durchsatz, Jitter

Netzwerkgrundlagen (2)

- Vermittlungsschemata
  - Broadcast
  - Leitungsvermittlung (Circuit Switching)
  - Paketvermittlung
  - Frame Relay

Netzwerkgrundlagen (3)

- Protokolle
  - Regeln und Formate für den Austausch von Nachrichten
  - Protokollschichten
    - Aufteilung der verschiedenen Aufgaben, die bei einer Nachrichtenübertragung anfallen
    - Schichten können aus Effizienzgründen in der Implementierung verschmolzen werden



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

Netzwerkgrundlagen (4)

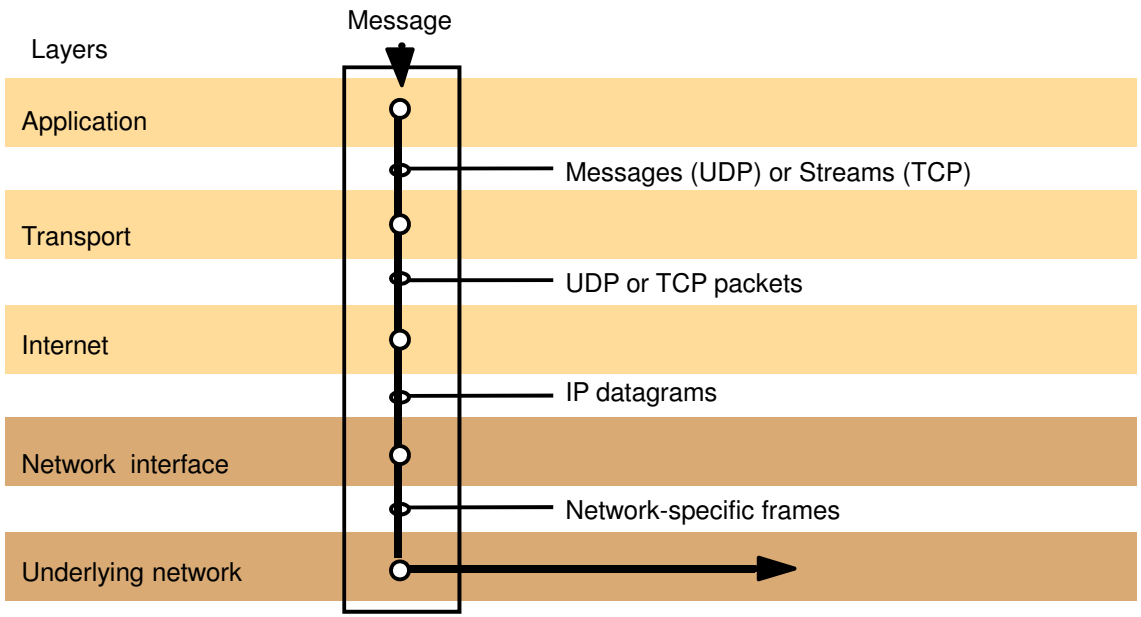
- ... Protokolle
  - Protokollstack
    - Vollständige Menge von Protokollschichten
    - Beispiel: CORBA - TCP/IP - Ethernet
  - Paketzusammenstellung
    - Paketgrößen in verschiedenen Protokollschichten / Teilnetzen ggf. unterschiedlich
    - Pakete einer Nachricht können zerteilt und verschmolzen werden
  - Adressierung
    - erfolgt auf verschiedenen Protokollschichten unterschiedlich: Anwendungs-spezifische Adressen, Prozesse, Rechner, Vermittlungsknoten
    - Abbildungsmechanismen innerhalb des Protokollstacks (z. B. IP-Adresse -> Ethernet-Adresse)
  - Paketzustellung
    - Datagramme – analog zu *Telegrammen*, jedes Datagramm enthält alle notwendigen Adressdaten und wird für sich eigenständig zugestellt
    - Virtuelle Verbindungen – analog zu *Telefonverbindungen* zunächst ein Verbindungsaufbau, die Datenpakete werden dann über die Verbindung übertragen.

Netzwerkgrundlagen (5)

- Routing
  - Zustellung eines Datenpakets durch Übertragung über mehrere Zwischen-/Vermittlungsrechner
  - grundlegendes Konzept in allen Netzen (außer sehr einfachen lokalen Netzen)
  - spezielle Protokolle legen die Übertragungswege fest
- Internetworking
  - Aufbau eines Netzwerks durch den Zusammenschluss vieler Netzwerke
  - lokale Netze, Funknetze, Weitverkehrsnetze
  - unterschiedliche Basistechnologien, unterschiedliche Vermittlungsknote und -Technologien (Router, Switches, Bridges, Hubs, Tunnel, ...)

Netzwerkgrundlagen (6)

- Internet-Protokolle
  - Basis des heutigen Internet
  - erste Entwicklungen in den 1960er Jahren im ARPAnet (NCP - *Network Control Protocol*)
  - Entwicklung der heutigen Kommunikationsprotokolle ab 1980 (TCP/IP - *Transmission Control Protocol / Internet Protocol*)
  - zunächst einfache Anwendungsprokolle für Dateitransfer (TFP), Rechnerzugang (rlogin), Mail (SMTP), ...



TCP/IP Grundlagen

- TCP/IP hat sich in den letzten 20 Jahren als der de-facto-Kommunikationsstandard etabliert
- API ursprünglich in der UNIX-Version der Univ. Berkeley entstanden
- Grundlegendes Kommunikationsparadigma: Client-Server-Modell
- Verbindungsorientierte Kommunikation: TCP/IP
- Paketorientierte Kommunikation: UDP/IP
- Schnittstelle zu Anwendungsprogrammen: Sockets = Kommunikationsendpunkte
- In UNIX-Systemen weitgehende Integration in die Datei-Ein/Ausgabe-Schnittstelle

Client-Server-Modell

Ein **Server** ist ein Programm, das einen **Dienst** (*Service*) anbietet, der über einen Kommunikationsmechanismus erreichbar ist.

Server

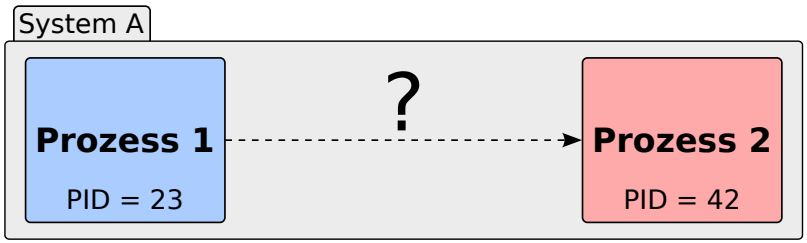
- **Akzeptiert Anforderungen**, die von außen kommen
- **Führt** einen angebotenen **Dienst aus**
- **Schickt** das **Ergebnis zurück** zum Sender der Anforderung
- Ist in der Regel als normaler Benutzerprozess realisiert

Client

- Schickt eine **Anforderung an einen Server**
- Wartet auf eine Antwort

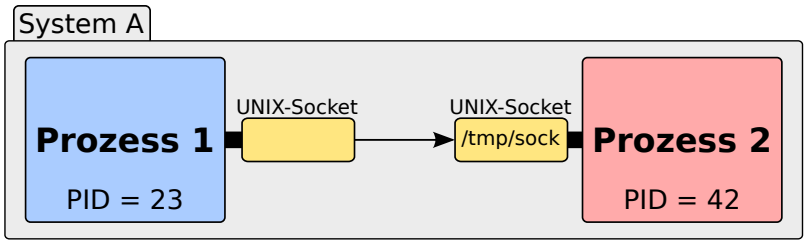
? **Wie findet man seinen gewünschten Kommunikationspartner?**

- Intuitiv: über dessen Prozess-ID



  - **Problem:** Prozesse werden dynamisch erzeugt/beendet; PID ändert sich

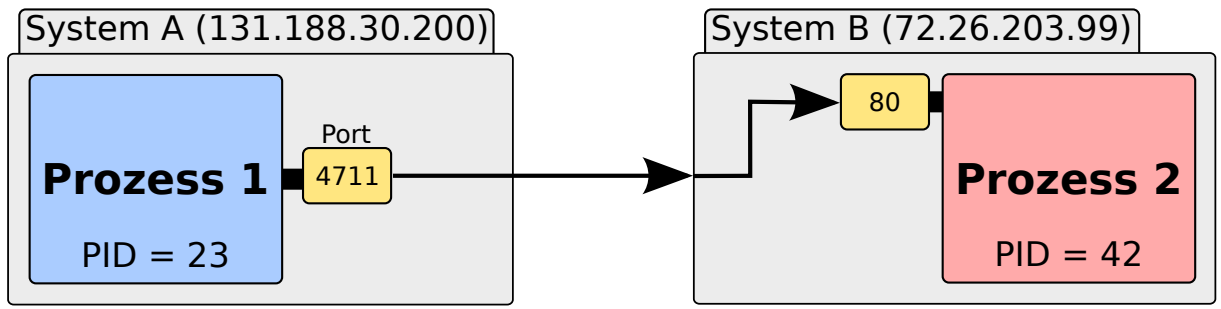
- **Lösung:** Verwendung eines abstrakten „Namens“



  - Prozess 2 ist so über einen speziellen Eintrag im Dateisystem erreichbar

? **Wie findet man nun seinen gewünschten Kommunikationspartner?**

- Wieder über einen Socket...
- ... diesmal aber mit zweistufig aufgebautem „Namen“:
  1. Identifikation des Systems innerhalb des Netzwerks
  2. Identifikation des Prozesses innerhalb des Systems
- Beispiel TCP/IP: eindeutige Kombination aus
  1. IP-Adresse
  2. Port-Nummer



Adressierung in IP-Netzwerken

Internet Protocol

- Netzwerkprotokoll zur Bildung eines virtuellen Netzwerkes auf der Basis mehrerer physischer Netze (Routing)
- Unzuverlässige Datenübertragung
  - Für Zuverlässigkeit ist darüberliegende Transportschicht zuständig

IPv4

- 32-Bit-Adressraum ($\approx$ 4 Milliarden Adressen)
- Notation: 4 mit `.` getrennte Byte-Werte in Dezimaldarstellung
  - z. B. `131.188.30.200`
- Nicht zukunftsfähig wegen des zu kleinen Adressraums
  - Alle Adressen in Asien/Pazifikraum (seit April 2011) und Europa/Nahost (seit September 2012) bereits vergeben!

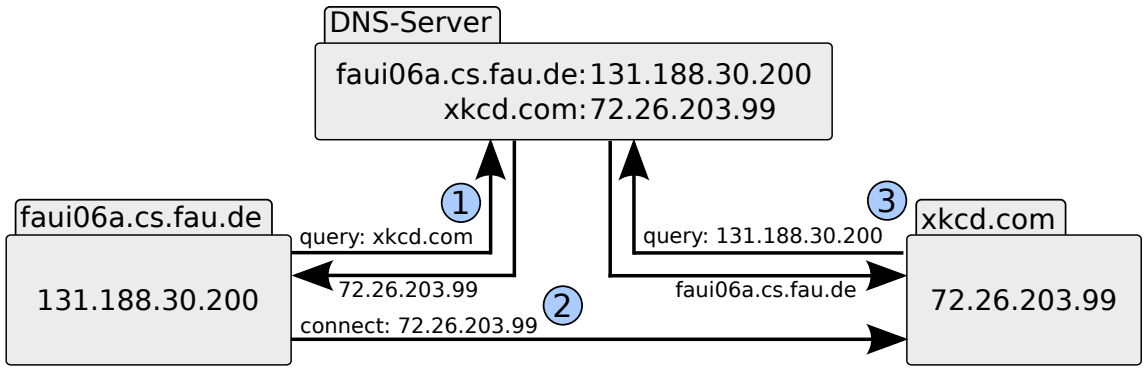
Adressierung in IP-Netzwerken

IPv6

- 128-Bit-Adressraum ($\approx 3{,}4 \cdot 10^{38}$ Adressen)
- Notation: 8 mit `:` getrennte 2-Byte-Werte in Hexadezimaldarstellung
  - z. B. `2001:638:a00:1e:219:99ff:fe33:8e75`
- In der Adresse kann einmalig `::` als Kurzschreibweise einer Nullfolge verwendet werden
  - z. B. *localhost*-Adresse: `0:0:0:0:0:0:0:1 = ::1`
- Abwärtskompatibilität durch transparente IPv4-in-IPv6-Unterstützung
- Bereits 1998 als Standard verabschiedet, aber seither nur schleppende Einführung

Domain Name System (DNS)

- IP-Adressen sind nicht leicht zu merken
- ... und ändern sich, wenn man einen Rechner in ein anderes Rechenzentrum umzieht
- **Lösung:** zusätzliche Abstraktion durchs DNS-Protokoll



1. *Forward lookup*: Rechnername → IP-Adresse
2. Kommunikationsaufbau
3. *Reverse lookup* (im Beispiel optional): IP-Adresse → Rechnername

Portnummer

- Zur Identifikation eines Prozesses innerhalb eines Systems
- 16-Bit-Zahl, d. h. kleiner als 65536
- Portnummern < 1024: *well-known ports*
  - Können nur von Prozessen gebunden werden, die mit speziellen Privilegien gestartet wurden (Ausführung als *root*)
  - z. B. ssh = 22, smtp = 25, http = 80

Kommunikationsarten

Verbindungsorientiert (Datenstrom)

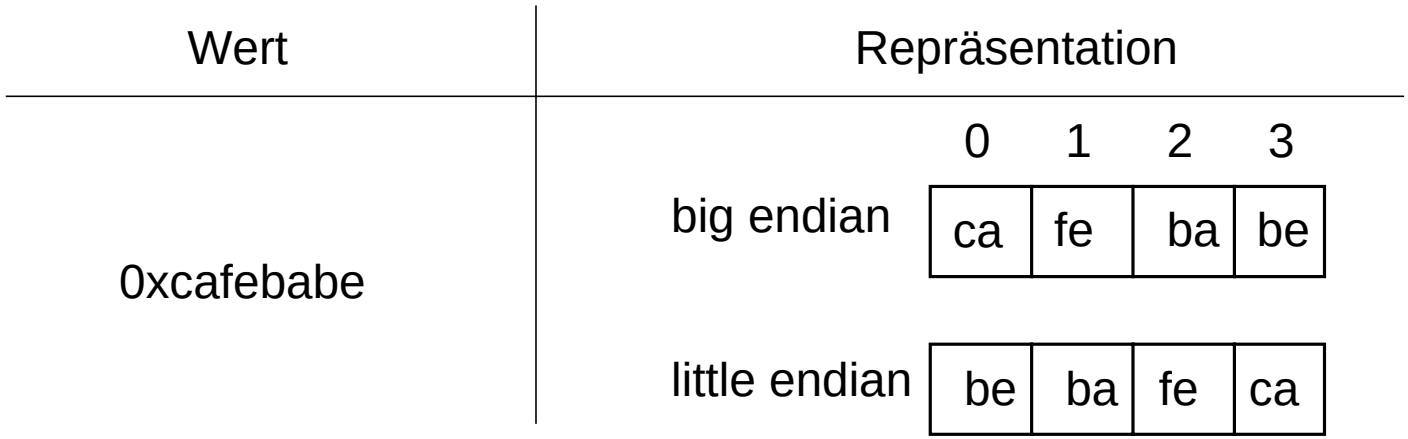
- Gesichert gegen Verlust und Duplizierung von Daten
- Reihenfolge der gesendeten Daten bleibt erhalten
- Vergleichbar mit einer Pipe – allerdings bidirektional
- Implementierung: Transmission Control Protocol (TCP)

Paketorientiert

- Schutz vor Bitfehlern – nicht vor Paketverlust oder -duplizierung
- Datenpakete können eventuell in falscher Reihenfolge ankommen
- Grenzen von Datenpaketen bleiben erhalten
- Implementierung: User Datagram Protocol (UDP)

Datendarstellung auf einem Rechner

- Beim Austausch von binären Datenwörtern ist die Reihenfolge der Einzelbytes zur richtigen Interpretation relevant
- Kommunikation zwischen Rechnern verschiedener Architekturen – z. B. x86 (*little endian*) und SPARC (*big endian*) – setzt Konsens über die verwendete Byteorder voraus
- Beispiel:



- **Definierter Standard:** Netzwerk-Byteorder = *big endian*

TCP/IP - Anwendungsschnittstelle — Sockets

- Generischer Mechanismus zur Interprozesskommunikation
- Verwendung im Programm ist unabhängig von der Kommunikations-Domäne
  - ... egal, ob der Kommunikationspartner ein Prozess auf dem selben Rechner ist oder ob er tausende von Kilometern entfernt ist
- Betriebssystemseitige Implementierung ist abhängig von der jeweiligen Kommunikations-Domäne
  - Innerhalb des selben Systems: z. B. UNIX-Socket
    $\rightarrow$ Adressierung über Dateinamen, Kommunikation über gemeinsamen Speicher, keine Sicherungsmechanismen notwendig
  - Über Rechnergrenzen hinweg: z. B. TCP/UDP-Socket
    $\rightarrow$ Adressierung über IP-Adresse + Port, nachrichtenbasierte Kommunikation, Sicherungsmechanismen bei TCP

Erzeugen eines Sockets

- Sockets werden mit dem Systemaufruf `socket(2)` angelegt:

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

  - `domain`, z. B.:
    - `PF_UNIX`: UNIX-Domäne
    - `PF_INET`: IPv4-Domäne
    - `PF_INET6`: IPv6-Domäne
  - `type` innerhalb der gewählten Domäne:
    - `SOCK_STREAM`: Stream-Socket
    - `SOCK_DGRAM`: Datagramm-Socket
  - `protocol`:
    - `0`: Standard-Protokoll für gewählte Kombination (z. B. TCP/IP bei `PF_INET(6)` + `SOCK_STREAM`)

- Ergebnis ist ein numerischer Socket-Deskriptor
  - Entspricht einem Datei-Deskriptor und unterstützt (bei Stream-Sockets) die selben Operationen: `read(2)`, `write(2)`, `close(2)`, ...

Binden eines Sockets an eine Adresse

- Nach seiner Erzeugung muss ein Socket zunächst an eine Adresse *gebunden* werden, bevor er verwendet werden kann
- Der Systemaufruf `bind(2)` stellt eine generische Schnittstelle zum Binden von Sockets in unterschiedlichen Domänen bereit:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

  - `sockfd`: Socket-Deskriptor
  - `addr`: protokollspezifische Adresse
    - Socket-Interface (`<sys/socket.h>`) ist zunächst protokollunabhängig:

```
struct sockaddr {
    sa_family_t sa_family; // Adressfamilie
    char sa_data[14];      // Platzhalter-Bytes für die Adresse
};
```

  - `addrlen`: Länge der konkret übergebenen Struktur in Bytes

Binden eines IPv4-Sockets

- Name durch IP-Adresse und Port-Nummer definiert:

```
struct sockaddr_in {
    sa_family_t    sin_family; // = AF_INET
    in_port_t      sin_port;   // Port
    struct in_addr sin_addr;   // Internet-Adresse
};
```

  - `sin_port`: Port-Nummer
  - `sin_addr`: IPv4-Adresse
    - `INADDR_ANY`: wenn Socket auf allen lokalen Adressen (z. B. allen Netzwerkschnittstellen) Verbindungen akzeptieren soll

- `sin_port` und `sin_addr` müssen in Netzwerk-Byteorder vorliegen!
  - Umwandlung mittels `htons(3)`, `htonl(3)`: konvertiert Datenwort von Host-spezifischer Byteordnung in Netzwerk-Byteordnung (*big endian*) – bzw. zurück:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Binden eines IPv6-Sockets

TCP/IP API

- Name durch IP-Adresse und Port-Nummer definiert:

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   // = AF_INET6
    in_port_t       sin6_port;     // Port-Nummer
    uint32_t        sin6_flowinfo; // = 0
    struct in6_addr sin6_addr;     // IPv6-Adresse
    uint32_t        sin6_scope_id; // = 0
};

struct in6_addr {
    unsigned char   s6_addr[16];
};
```

  - `sin6_addr`: IPv6-Adresse
    - `in6addr_any` / `IN6ADDR_ANY_INIT`: auf allen lokalen Adressen Verbindungen akzeptieren

- Die Werte für `in6addr_any` bzw. `IN6ADDR_ANY_INIT` liegen bereits in Netzwerk-Byteorder vor

Verbindungsaufbau durch Client

- `connect(2)` meldet Verbindungswunsch an Server:

```
int connect(int sockfd, const struct sockaddr *addr,
                                        socklen_t addrlen);
```

  - `sockfd`: Socket, über den die Kommunikation erfolgen soll
  - `addr`: Beinhaltet abstrakten „Namen“ (bei uns: IP-Adresse und Port) des Servers
  - `addrlen`: Länge der `addr`-Struktur
- `connect()` blockiert solange, bis der Server die Verbindung annimmt oder zurückweist
- Falls der Socket noch nicht lokal gebunden ist, wird automatisch eine lokale Bindung hergestellt (Port-Nummer wird vom System gewählt)
- Socket ist anschließend bereit zur Kommunikation mit dem Server

DNS-Anfragen

- Zum Ermitteln der Werte für die `sockaddr`-Struktur kann das DNS-Protokoll verwendet werden
- `getaddrinfo(3)` liefert die nötigen Werte:

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

  - `node` gibt den DNS-Namen des Hosts an (oder die IP-Adresse als String)
  - `service` gibt entweder den numerischen Port als String (z. B. `"25"` oder den Dienstnamen (z. B. `"smtp"`, `getservbyname(3)`) an
  - Mit `hints` kann die Adressauswahl eingeschränkt werden (z. B. auf IPv4-Sockets). Nicht verwendete Felder auf `0` bzw. `NULL` setzen.
  - Ergebnis ist eine verkettete Liste von Socket-Namen; ein Zeiger auf das Kopfelement wird in `*res` gespeichert
- Freigabe der Ergebnisliste nach Verwendung mit `freeaddrinfo(3)`

DNS-Anfragen

```
struct addrinfo {
  int              ai_flags;     // Flags zur Auswahl (hints)
  int              ai_family;    // z. B. PF_INET6
  int              ai_socktype;  // z. B. SOCK_STREAM
  int              ai_protocol;  // Protokollnummer
  socklen_t        ai_addrlen;   // Größe von ai_addr
  struct sockaddr *ai_addr;      // Adresse fuer bind()/connect()
  char            *ai_canonname; // Offizieller Hostname (FQDN)
  struct addrinfo *ai_next;      // Nächstes Listenelement oder NULL
};
```

- `ai_flags` relevant zur Anfrage von Auswahlkriterien (`hints`)
  - `AI_ADDRCONFIG`: Auswahl von Adresstypen, für die auch ein lokales Interface existiert (z. B. werden keine IPv6-Adressen geliefert, wenn der aktuelle Rechner gar keine IPv6-Adresse hat)
- `ai_family`, `ai_socktype`, `ai_protocol` für `socket(2)` verwendbar
- `ai_addr`, `ai_addrlen` für `bind(2)` und `connect(2)` verwendbar

DNS-Anfragen: Beispiel

```
struct addrinfo hints = {
    .ai_socktype = SOCK_STREAM,  // Nur TCP-Sockets
    .ai_family   = PF_UNSPEC,    // Beliebige Protokollfamilie
    .ai_flags    = AI_ADDRCONFIG // Nur lokal verfuegbare Adresstypen
}; // C99: alle anderen Elemente der Struktur werden implizit genullt

struct addrinfo *head;
int error = getaddrinfo("xkcd.com", "80", &hints, &head);
if (error != 0) {
    // Fehler! Behandlung siehe Man-Page
}

// Liste der Adressen durchtesten
int sock;
struct addrinfo *curr;
for (curr = head; curr != NULL; curr = curr->ai_next) {
    sock = socket(curr->ai_family, curr->ai_socktype, curr->ai_protocol);
    if (connect(sock, curr->ai_addr, curr->ai_addrlen) == 0)
        break;
    close(sock);
}
if (curr == NULL) {
    // Fehler!
}

freeaddrinfo(head);
```

Verbindungsannahme durch Server

- **Ausgangssituation:** Socket wurde bereits erstellt (`socket(2)`) und an einen Namen gebunden (`bind(2)`)
- Verbindungsannahme vorbereiten mit `listen(2)`:

```
int listen(int sockfd, int backlog);
```

  - `backlog`: Unverbindliche Größe der Warteschlange, in der alle eingehenden Verbindungswünsche zwischengepuffert werden
    - Bei voller Warteschlange werden Verbindungsanfragen zurückgewiesen
    - Maximal mögliche Größe: `SOMAXCONN`

Verbindungsannahme durch Server (2)

- Verbindung annehmen mit `accept(2)`:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

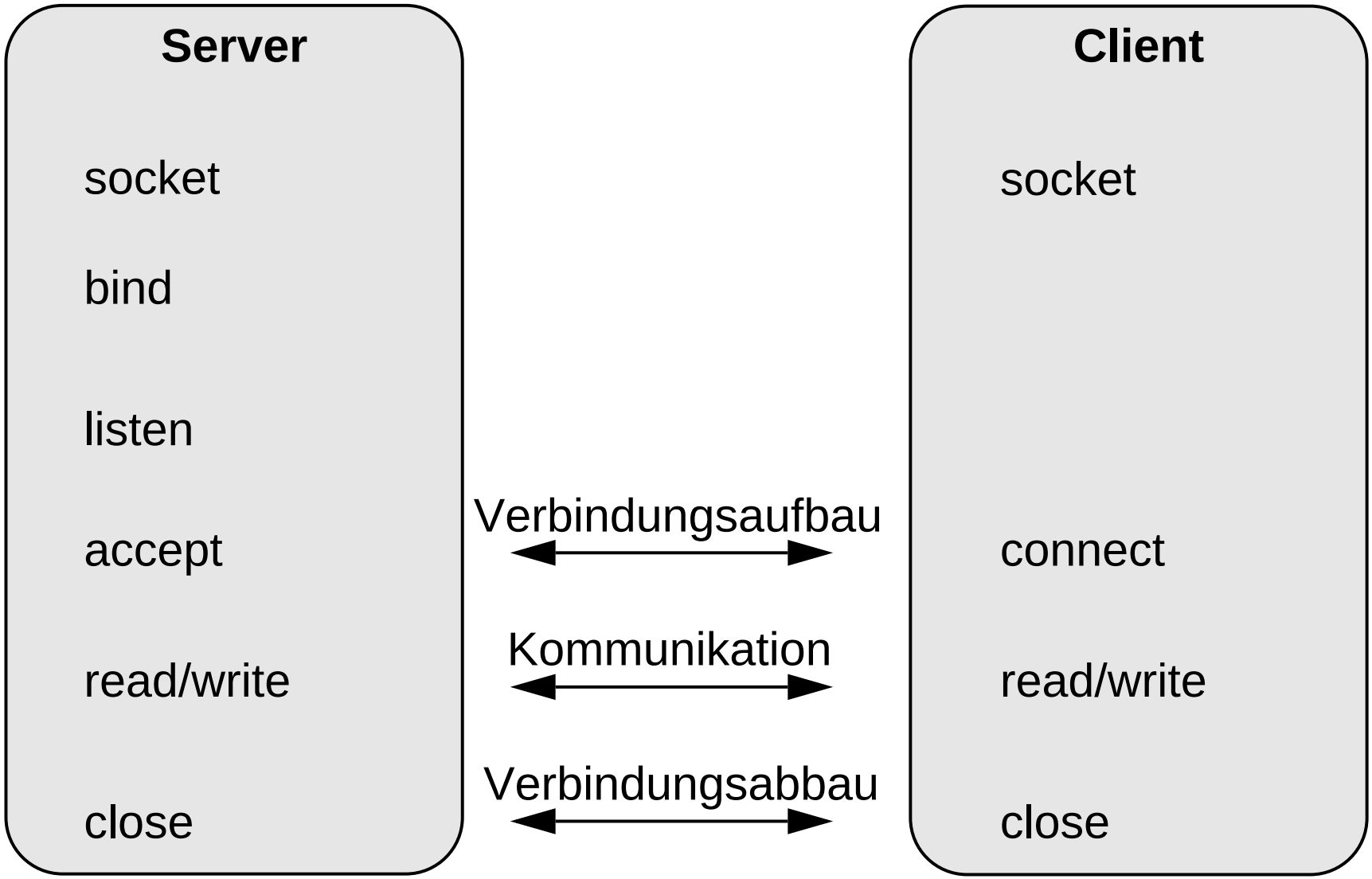
  - `addr`, `addrlen`: Ausgabeparameter zum Ermitteln der Adresse des Clients
    - Bei Desinteresse zweimal `NULL` übergeben
- Entnimmt die vorderste Verbindungsanfrage aus der Warteschlange
  - Blockiert bei leerer Warteschlange
- Erzeugt einen neuen Socket und liefert ihn als Rückgabewert
  - Kommunikation mit dem Client über diesen neuen Socket
  - Annahme weiterer Verbindungen über den ursprünglichen Socket

Verbindungsannahme durch Server (3)

- Nach Beendigung des Server-Prozesses erlaubt das Betriebssystem kein sofortiges `bind(2)` an den selben Port
  - Erst nach Timeout erneut möglich
  - Grund: es könnten sich noch Datenpakete für den alten Prozess auf der Leitung befinden
- Testen und Debuggen eines Server-Programms dadurch stark erschwert
- Lösungsmöglichkeiten:
  1. Bei jedem Start einen anderen Port verwenden
  2. Sofortige Wiederverwendung des Ports forcieren:

```
int sock = socket(...);
...
int flag = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &flag, sizeof(flag));
...
bind(sock, ...);
```

TCP/IP-Sockets: Zusammenfassung



Ein-/Ausgabemechanismen

- Nach dem Verbindungsaufbau lässt sich ein Stream-Socket nach dem selben Schema benutzen wie eine geöffnete Datei
- Für Ein- und Ausgabe stehen verschiedene Funktionen zur Verfügung:
  - Ebene 2: POSIX-Systemaufrufe
    - arbeiten mit Filedeskriptoren (`int`)
  - Ebene 3: Bibliotheksfunktionen
    - greifen intern auf die Systemaufrufe zurück
    - wesentlich flexibler einsetzbar
    - arbeiten mit File-Pointern (`FILE *`)

Ebene	Variante	Ein-/Ausgabedaten	Funktionen
2	blockorientiert	Puffer, Länge	`read()`, `write()`
3	blockorientiert	Array, Elementgröße, -anzahl	`fread()`, `fwrite()`
	zeichenorientiert	Einzelbyte	`getc()`, `putc()`
	zeilenorientiert	`'\0'`-terminierter String	`fgets()`, `fputs()`
	formatiert	Formatstring, beliebige Variablen	`fscanf()`, `fprintf()`

Ein-/Ausgabemechanismen

- Auf Grund ihrer Flexibilität eignen sich `FILE *` für String-basierte Ein- und Ausgabe wesentlich besser
- Konvertierung von Dateideskriptor nach `FILE *`:

```
FILE *fdopen(int fd, const char *mode);
```

  - `mode` kann sein: `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"` (`fd` muss entsprechend geöffnet sein)
  - Sockets sollten mit `"a+"` geöffnet werden
- Schließen des erzeugten `FILE *`:

```
int fclose(FILE *fp);
```

  - Darunterliegender Filedeskriptor wird dabei geschlossen
  - Erneutes `close(2)` nicht notwendig

Beispiel: einfacher Echo-Server

! Fehlerabfragen nicht vergessen

```
int listenSock = socket(PF_INET6, SOCK_STREAM, 0);

struct sockaddr_in6 name = {
    .sin6_family = AF_INET6,
    .sin6_port   = htons(1112),
    .sin6_addr   = in6addr_any
};
bind(listenSock, (struct sockaddr *) &name, sizeof(name));

listen(listenSock, SOMAXCONN);

for (;;) {
    int clientSock = accept(listenSock, NULL, NULL);
    char buf[1024];
    ssize_t n;
    while ((n = read(clientSock, buf, sizeof(buf))) > 0) {
        write(clientSock, buf, n);
    }
    close(clientSock);
}
```

Beispiel: einfacher Echo-Server

```
int clientSock;
while ((clientSock = accept(listenSock, NULL, NULL)) != -1) {
    char buf[1024];
    ssize_t n;
    while ((n = read(clientSock, buf, sizeof(buf))) > 0) {
        write(clientSock, buf, n);
    }
    close(clientSock);
}
```

- Limitierungen:
  - Neue eingehende Verbindung kann erst nach vollständiger Abarbeitung der vorherigen Anfrage angenommen werden
  - Monopolisierung des Dienstes möglich (*Denial of Service*)!
- Mögliche Ansätze zur Abhilfe:

1. Mehrere Prozesse
   - Anfrage wird durch Kindprozess bearbeitet
2. Mehrere Threads
   - Anfrage wird durch einen Thread im gleichen Prozess bearbeitet

Überblick

Fernaufrufe – Überblick

- IPC-Semantiken
- IPC und Fernaufrufe
- Prozeduraufruf und aktionsorientierte Kommunikation
- Semantikaspekte
  - Parameterarten, Parameterübergabe, Gültigkeitsbereiche, Speicheradressen
- Nachrichten zusammenstellen/auseinandernehmen
- Zustellungsgarantien, Aufrufsemantiken, Fehlermodell, Waisen

IPC-Semantiken

Grundkonzepte für die Kommunikation zwischen Prozessen durch Nachrichtenaustauasch

- **no-wait send** der Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist
  - *Pufferung* oder *Signalisierung* (dass der übergebene Puffer wieder frei ist)
- **synchronization send** der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist
  - *Rendezvous* zwischen Sende- und Empfangsprozess
- **remote-invocation send** der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist
  - *Fernaufruf* einer vom Empfangsprozess auszuführenden Funktion

IPC und Fernaufrufe

- IPC bildet die Grundlage für den Austausch von Nachrichten zwischen Prozessen
- die Nutzung der TCP/IP-Protokolle bietet zwei Alternativen
  - Verpacken von Nachrichten in UDP-Paketen
    - Ein Paket pro Nachricht
    - Nachrichtenzustellung ist zunächst genauso unzuverlässig wie UDP
  - Versenden von Nachrichten über TCP-Verbindungen
    - Nachrichten müssen im Datenstrom der Verbindung kodiert werden (Anfang und Ende kenntlich machen)
    - Nachrichtenübertragung ist zuverlässig – *so lange die Verbindung steht!*

IPC und Fernaufrufe (2)

- IPC bildet die Basis, in der „darüberliegenden“ Ebene steht die *Bedeutung* der Nachrichten im Mittelpunkt
  - die Bedeutung kann sich *implizit* durch den Verarbeitungsalgorithmus (das Programm) ergeben
  - oder
  - die Prozesse machen sie sich gegenseitig *explizit* über „Anweisungen“ bekannt
- die Nachrichten enthalten (problemspezifische) Daten und/oder Code:
  function shipping der Empfangsprozess interpretiert Programme
  - mobiler Code (Java Bytecode, PostScript) ggf. mit Daten unterfüttert

  data shipping der Empfangsprozess interpretiert Daten
- im „Normalfall“ bewirken Nachrichten die Ausführung entfernter Routinen
  - die aufzurufenden Prozeduren/Funktionen sind implizit oder explizit kodiert

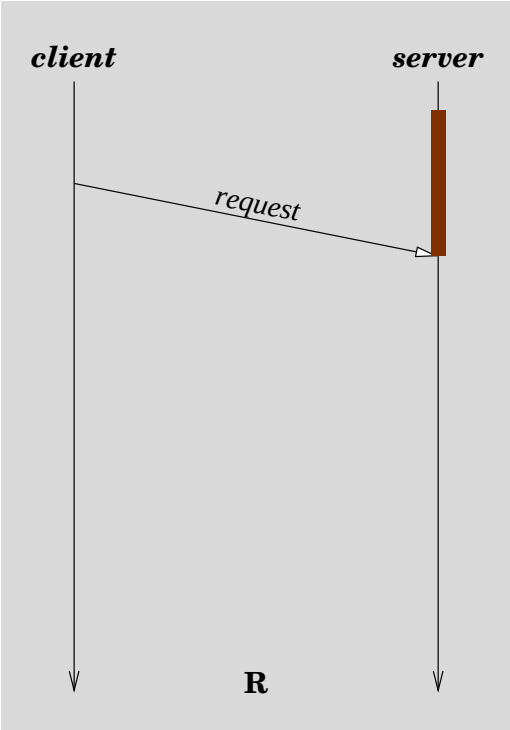
IPC-Protokolle für Fernaufrufe

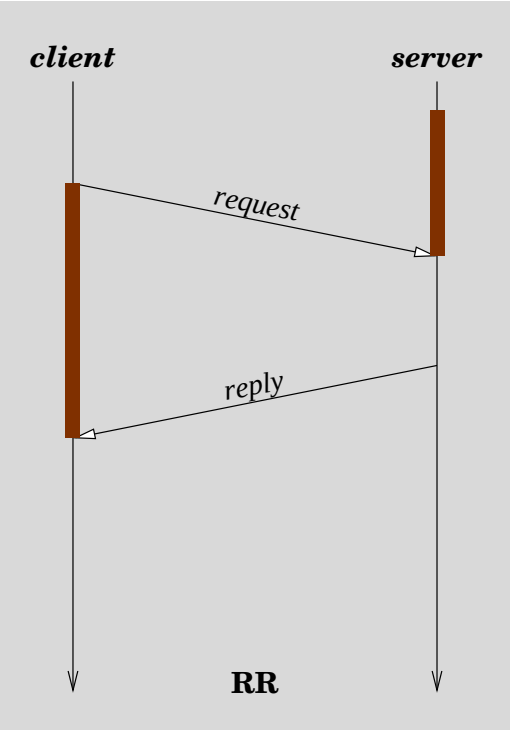
request (**R**) kann genutzt werden, wenn die entfernte Prozedur/Funktion keinen Rückgabewert liefert und der Sendeprozess keine Bestätigung für die erfolgte Ausführung benötigt . . . . . 1 Nachricht

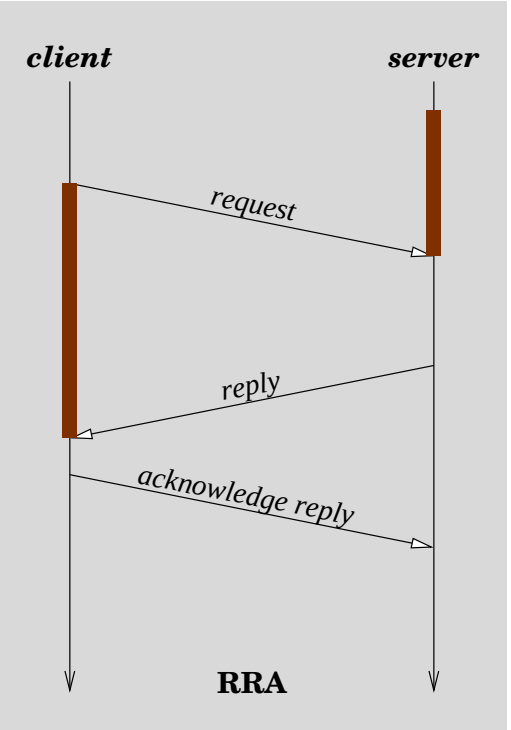
request-reply (**RR**) ist das geläufige Verfahren, da die Antwortnachricht implizit die Anforderungsnachricht bestätigt und dadurch explizite Bestätigungen entfallen . . . . . 2 Nachrichten

request-reply-acknowledge reply (**RRA**) gestattet es, die zum Zwecke der *Fehlermaskierung* (beim Server) gespeicherten Antwortnachrichten zu verwerfen, wenn (vom Client) keine weitere Anforderungsnachricht gesendet wird . . . . . 3 Nachrichten

IPC-Protokolle für Fernaufrufe (2)







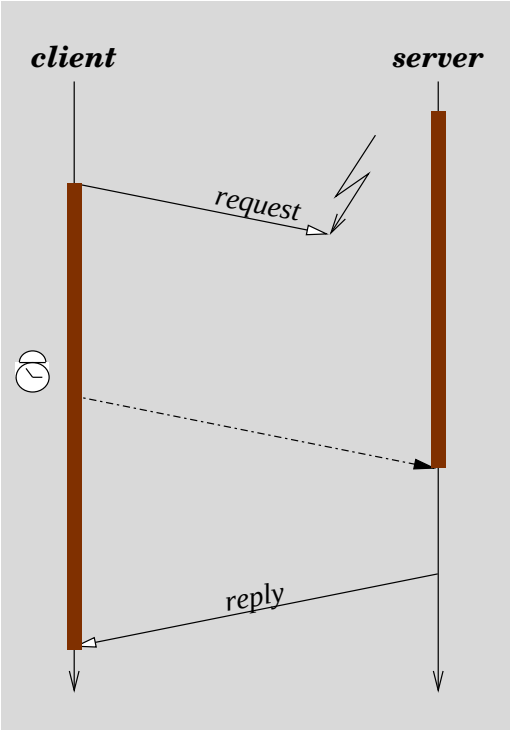
Zustellungsfehler

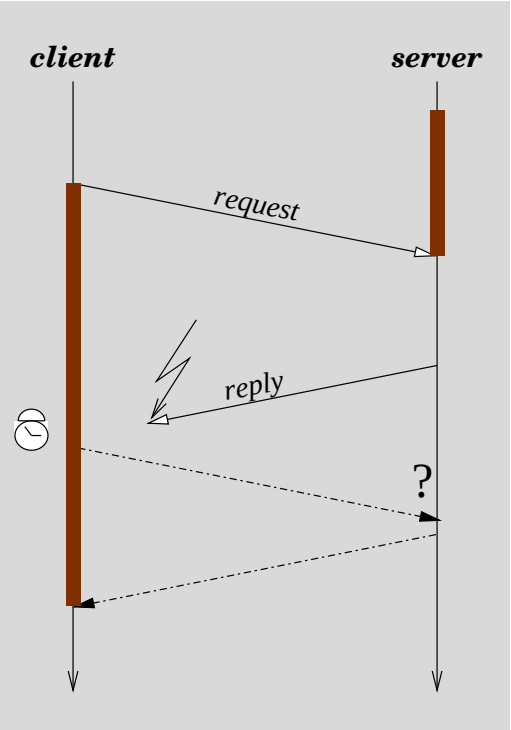
- Nachrichtenaustausch unterliegt bestimmten (typischen) *Fehlerannahmen*:
  1. Nachrichten können verloren gehen
     - beim Sender, beim Empfänger oder im Netz
  2. es können Netzwerk-Partitionierungen auftreten
     - ein oder mehrere Rechner (Knoten) werden „abgetrennt“
  3. Prozesse können scheitern (d.h. „abstürzen“)
     - Prozess-, Rechner- oder Netzwerkausfälle sind nicht unterscheidbar
  4. Daten können verfälscht werden
- als Folge sind unterschiedliche (typische) *Protokollvarianten* entstanden

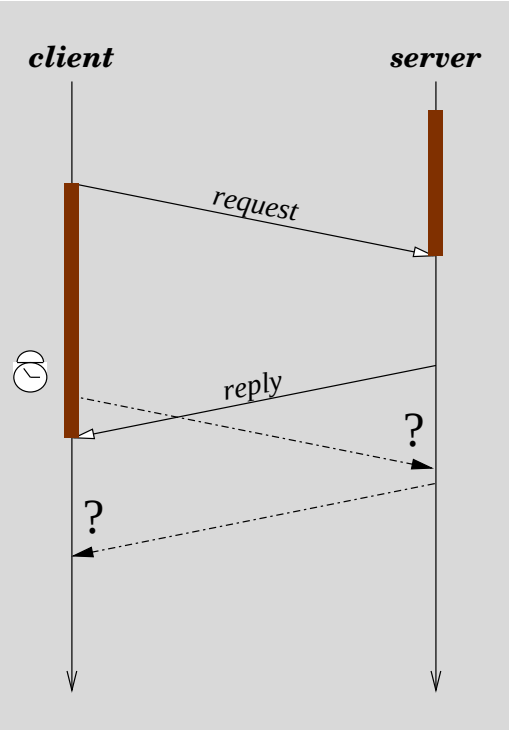
Fehlermaskierung

- Anforderungs- und ggf. auch Antwortnachrichten wiederholen
  - nach einer Pause (*time-out*) werden die Nachrichten erneut versendet
  - die „optimale“ Länge der Pause zu bestimmen ist äußerst schwierig
- eingetroffene Nachrichtenduplikate sind zu erkennen und zu ignorieren
  - ggf. bereits versandte Antwortnachrichten wiederholt versenden
  - auf Client- bzw. Server-Seite ist ggf. „*duplicate supression*“ anzuwenden
- *idempotente Operationen*/Zustandsfreiheit tolerieren Anforderungsduplikate

Fehlermaskierung (2)







Aktionsorientierte/datenorientierte Kommunikation

- Bei klassischer IPC steht der Austausch von Daten (Nachrichten, Datenströme) im Vordergrund
- Alternative: Beauftragen einer *Aktivität* bei dem anderen Prozess: **aktionsorientierte Kommunikation**.

  Grundschema der Interaktion zwischen Auftraggeber (AG) und Auftragnehmer (AN):

1. AG sendet die Anforderung zur Diensterbringung, die ein AN empfängt
2. währenddessen wartet der AG auf eine Rückmeldung
3. die Rückmeldung sendet der AN nach erfolgter Diensterbringung
4. mit Zustellung der Rückmeldung kann der AG weiterarbeiten

Eine aktionsorientierte Kommunikation erfordert damit zwei Vorgänge **datenorientierter Kommunikation**:
(*request* und *reply*)

Kommunikationsmittel „Prozedur“

- Gemeinsamkeit von Prozeduraufruf und aktionsorientierter Kommunikation:
  - der AG entspricht der Routine, die den Prozeduraufruf tätigt — *Client*
  - der AN entspricht der aufgerufenen Prozedur — *Server*
  - request/reply entsprechen Aufruf/Rücksprung
- Unterschied: beim Prozeduraufruf ist der die Prozedur aufrufende *Prozess* mit dem die Prozedur ausführenden identisch
- *aktionsorientierte Kommunikation* ist der Aufruf einer entfernten Prozedur
  - „entfernt“ heißt „anderer Faden, Adressraum, Prozess und/oder Rechner“
  - $\Rightarrow$ **Prozedurfernaufruf**

Synchrone Kommunikation

- die *remote-invocation send*-Semantik unterstützt Prozedurfernaufrufe, die *ein Ergebnis liefern*

  send gibt die Anforderung ab, blockiert den AG und deblockiert ggf. den AN

  receive vom AN nimmt die Anforderung entgegen

  reply übermittelt die Rückmeldung des AN und deblockiert den AG

- die *synchronization send*-Semantik unterstützt alle *sonstigen* Prozedurfernaufrufe:

  send gibt die Anforderung ab, blockiert den AG und deblockiert ggf. den AN

  receive vom AN nimmt die Anforderung entgegen und deblockiert den AG

Asynchrone Variante — *Promises*

- asynchroner Prozedurfernaufruf kehrt *sofort* zurück und liefert als Ergebnis ein *promise*-Objekt (ein „Versprechen“ auf das Ergebnis, ohne jedoch den Verfügbarkeitszeitpunkt festzulegen)
  - das *promise*-Objekt dient der Aufnahme des ihm zugeordneten Resultats
  - der *promise*-Zustand kann (mittels `ready()`) abgefragt werden:

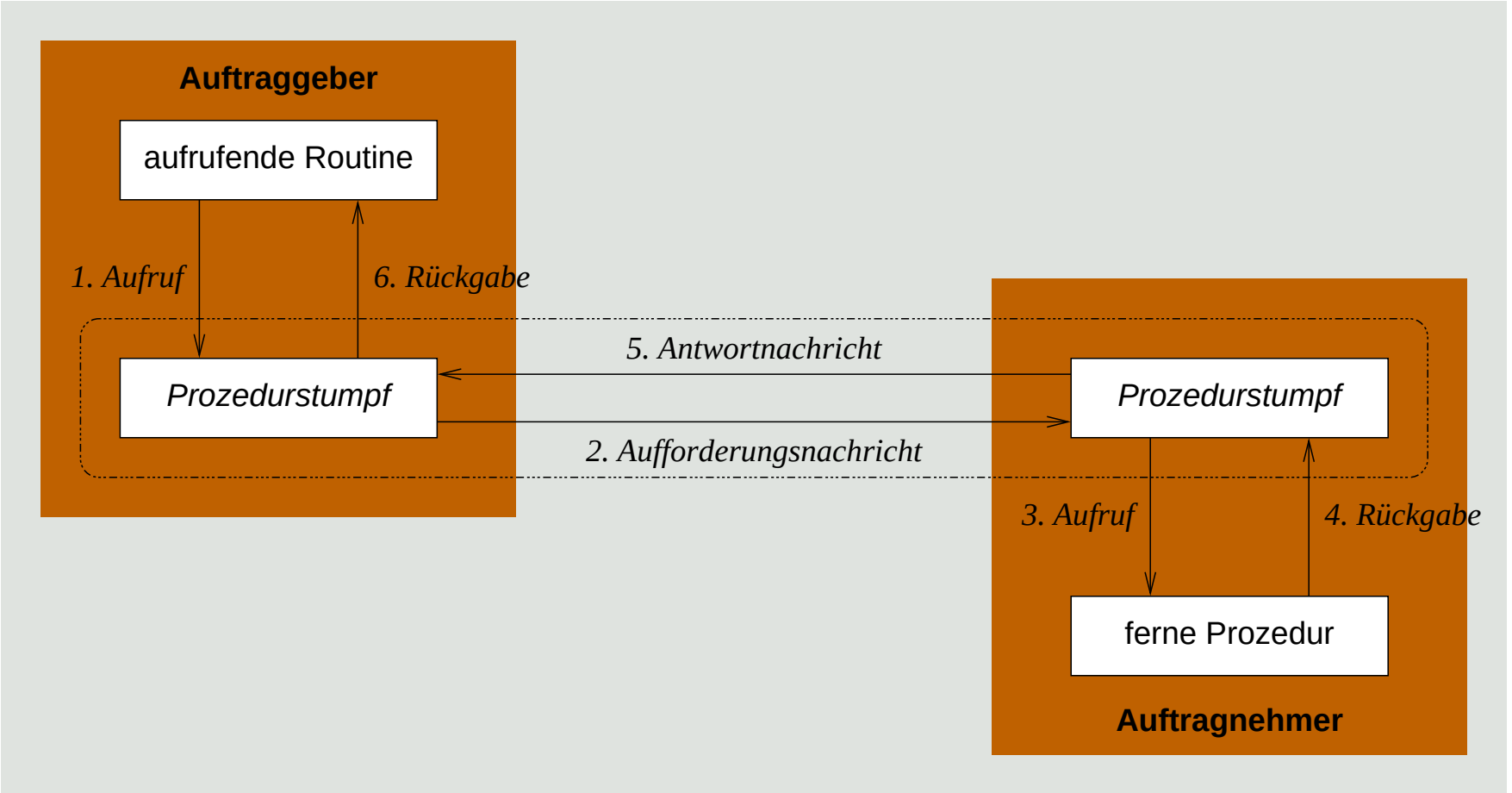
    blockiert ⇒ der Aufruf läuft, ein Resultat liegt noch nicht vor
    - ein `claim()`-Aufruf würde blockieren bis das Ergebnis vorliegt

    bereit ⇒ der Aufruf ist beendet, das Resultat liegt vor
    - in `claim()` ggf. blockierte Aufrufer werden deblockiert
    - der `claim()`-Aufruf liefert das Ergebnis zurürck
  - eine sprachliche Unterstützung im objektorientierten Sinn erleichtert die Anwendung
  - Alternative für Sprachunterstützung: asynchrone Aufrufe mit *wait by necessity*
    - keine expliziten Promise-Objekte sondern implizites Blockieren bei Benutzung des Prozedurergebnisses
  - mit *no-wait send* wird die *promise*-Implementierung ideal unterstützt

Prozeduraufruf ⇒ Nachrichtenaustausch

- Client Stub: Prozedurstumpf auf Seite des Auftraggebers
  - abstrahiert von der Örtlichkeit der entfernten, aufgerufenen Prozedur
  - setzt den Prozeduraufruf in einen Nachrichtenaustausch um
  - verpackt die tatsächlichen Aufrufparameter und entpackt Rückgabewerte
- Server Stub: Prozedurstumpf auf Seite des Auftragnehmers
  - abstrahiert von der Örtlichkeit der entfernten, aufrufenden Prozedur
  - setzt die Prozedurrückkehr in einen Nachrichtenaustausch um
  - entpackt die Aufrufparameter und verpackt Rückgabewerte

Prozedurstümpfe beim Prozedurfernaufruf



Konventioneller Aufruf vs. Fernaufruf

- Ziel eines Fernaufrufmechanismus ist es, die bekannte Semantik konventioneller Prozeduraufrufe aufrecht zu erhalten, obwohl sich die Ausführungsumgebung radikal anders gestaltet:
  - aufrufende und aufgerufene Seite sind örtlich voneinander getrennt
    - Code und Daten teilen nicht denselben (physikalischen) Arbeitsspeicher
  - beide Seiten arbeiten weitestgehend autonom
    - sie werden von verschiedenen Prozessen/Prozessoren ausgeführt
  - beide Seiten können unabhängig voneinander ausfallen

Die Semantik konventioneller, lokaler Prozeduraufrufe ist nur teilweise erreichbar

Semantikaspekte

- Parameterarten sind zu unterscheiden, um Aufwand zu minimieren
  - Eingabe- vs. Ausgabe- vs. Ein-/Ausgabeparameter
- Parameterübergabe ist ggf. explizit zu spezifizieren
  - *call-by-reference* vs. *call-by-value/result*
- Gültigkeits-/Sichtbarkeitsbereiche sind ggf. massiv eingeschränkt
  - Variablen des entfernten umfassenden *Scopes* sind meist nicht gültig/sichtbar
- Speicheradressen sind im Regelfall nicht systemweit eindeutig
  - Zeiger in Nachrichten zu versenden, ist (nahezu) sinnlos

Semantikaspekte (2) — Parameterarten

- die Auslegung der (formalen) Parameter bestimmt u.a. den IPC Mehraufwand:

  Eingabeparameter sind **nur** Bestandteil der *Anforderungsnachricht*

  Ausgabeparameter sind **nur** Bestandteil der *Antwortnachricht*

  Ein-/Ausgabeparameter sind Bestandteil **beider** Nachrichten

- nicht immer liefern Programmiersprachen passende Auslegungshinweise, z.B.

  C/C++ { *Zeiger* `char*`, `struct Foo*`; *Referenz* `struct Foo&`; *Feld* `char foo[4]` } welche Parameterart?

- die Art eines jeden Parameters müsste in der Schnittstelle spezifiziert sein

- *call-by-value/result* sind „geradlinig“ und einfach zu behandeln
  - die tatsächlichen Parameter werden in die/aus den jeweiligen Nachrichten kopiert
- *call-by-reference* wird je nach Parameterart abgebildet wie folgt:

Einabeparameter	→	*call-by-value*	} dereferenzieren
Ausgabeparameter	→	*call-by-result*	
Ein-/Ausgabeparameter	→	*call-by-value-result*	
unspezifiziert	→	*call-by-value*	Speicheradresse

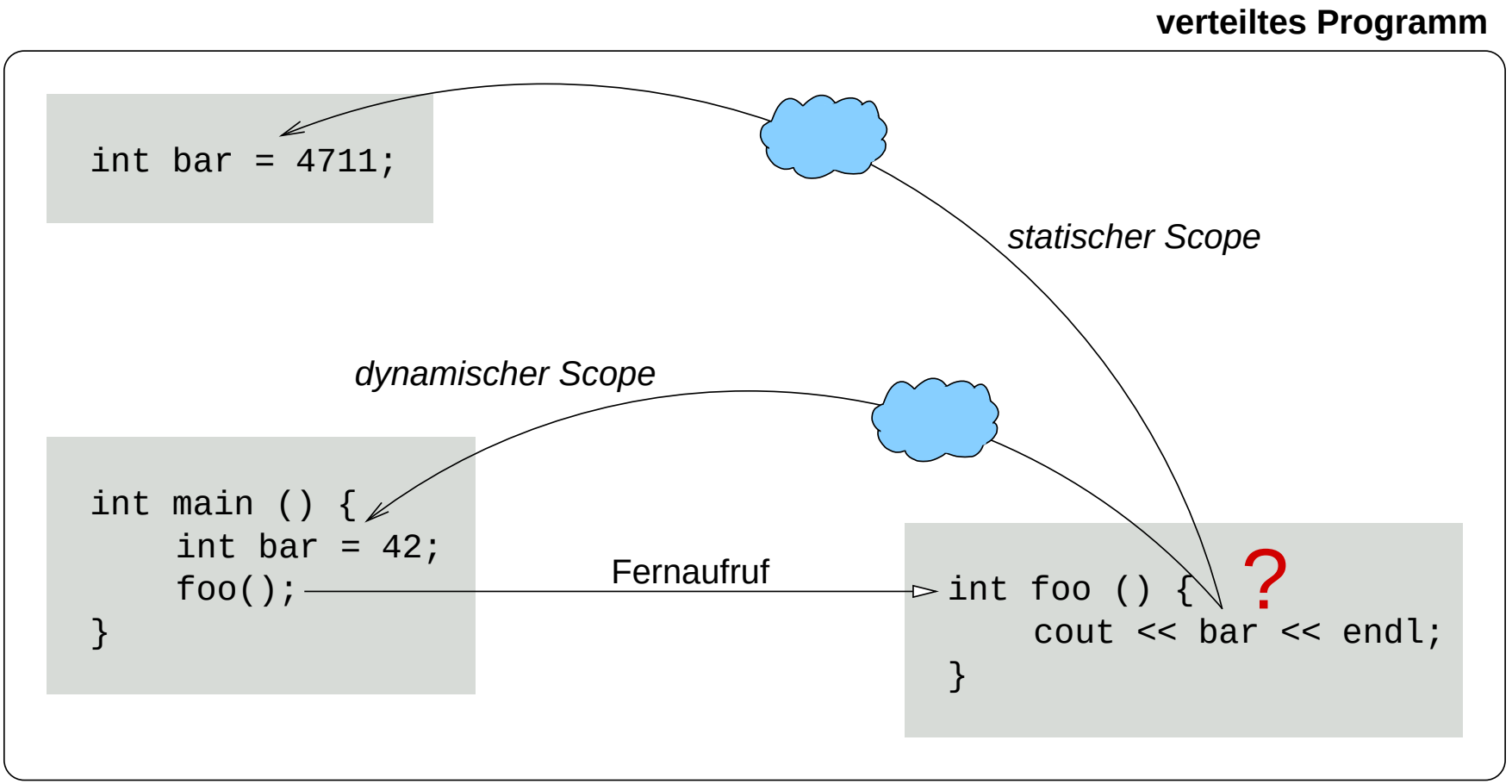
- *call-by-name* ggf. nur auf Basis von *function shipping* –
  - eine Funktion wird mitgeliefert, die den tatsächlichen Parameter berechnet

- jeder Name (einer Variablen) ist mit einem Platzhalter assoziiert
  - die Variable belegt einen Speicherplatz an einer bestimmten Adresse
  - den Namen/Platzhaltern sind Gültigkeitsbereiche („Blöcke“) zugeordnet
- die Bindung eines Namens an seinen Block (*Scope*) ist statisch oder dynamisch:

  statischer Scope ist bereits zur *Übersetzungszeit* bekannt
  - ändert sich nur bei Quelltextänderungen am Programm

  dynamischer Scope ist erst zur *Laufzeit* bekannt
  - ändert sich mit Eintritt in/Verlassen von Prozeduren bzw. Funktionen
- der „umfassende Block“ ist für eine entfernte Prozedur nicht oder nur bedingt zugänglich

Gültigkeitsbereiche von Programmvariablen



C/C++ bindet statisch, wie die meisten anderen Programmiersprachen auch. Das löst das Problem aber nicht.

- Programmadressen sind (im Regelfall) nicht systemweit eindeutig:
  - die fragliche Adresse kann beim Dienstanbieter bereits vergeben sein
  - nur im *single programm, multiple data* (SPMD) Modell wäre sie eindeutig (In dem Fall liegt auf den Rechnern des verteilten Systems dasselbe Programm. Demzufolge sind auf allen betrachteten Rechnern die Adressen der Variablen aber auch der Prozeduren/Funktionen identisch.)
  - im „Normalfall“ ist die Eindeutigkeit einer solchen Adresse eher zufällig
- allgemein sind Adressen abhängig von dem Kontext, in dem sie definiert sind
  - sie beziehen sich auf ein Programm in einem Adressraum auf einem Rechner
  - Adressen verteilter Programme besitzen eine **geographische Komponente**
- ferne Adressen entsprechen logischen Adressen — sie sind (ggf.) abbildbar

Verteilter gemeinsamer Speicher

- Multiprozessor
  - eng gekoppelte Prozessoren
  - gemeinsamer und kohährenter Speicher
- Multicomputer und Verteilte Systeme
  - lose gekoppelte Prozessoren
  - nur privater Speicher verfügbar
- verteilter gemeinsamer Speicher
  - Illusion eines gemeinsamen Speichers

Verteilter gemeinsamer Speicher (2)

- Mögliche Realisierung
  - seitenbasierte Speicherverwaltung
  - Seitenadressierung, unterstützt durch MMU
  - entspricht logischem/virtuellem Speicher
  - ein virtueller Adressraum über alle beteiligten Rechner
  - eine Seite residiert zu einem Zeitpunkt immer nur auf genau einem Rechner
- Lokaler Zugriff
  - Seite ist in der lokalen Seitentabelle eingetragen, Präsenzbit ist gesetzt $\Rightarrow$ Zugriff ist möglich
  - Zugriff erfolgt auf lokalen Speicher

Verteilter gemeinsamer Speicher (3)

- Entfernter Zugriff
  - Seite ist in lokaler Seitentabelle nicht eingetragen, Präsenzbit ist nicht gesetzt
  - lokaler Zugriff löste einen Seitenfehler (page fault) aus
    $\Rightarrow$ Unterbrechungsbehandlung durch das Betriebssystem
  - Betriebssystem holt die Seite von dem entfernten Rechner
    (hierbei wird die Seite dort ausgetragen!)
  - Seite wird in lokale Seitentabelle eingetragen, Präsenzbit wird gesetzt
  - Speicherzugriff wird wiederholt

- Kohärenter Speicher
  (Lese-Operation liefert immer den zuletzt geschriebenen Wert)
  - falls Hole- und Weitergabe-Operationen für Seiten atomar
  - falls keine Ausfällte auftreten

Verteilter gemeinsamer Speicher (4)

- Nebenläufige Zugriffe auf eine Seite sind sehr ineffizient
  - Seitenflattern zwischen Rechnern
- False-Sharing
  - zwei Datenobjekte liegen in der selben Seite, werden aber von unterschiedlichen Programmbereichen benutzt
  - die unterschiedlichen Programmbereiche werden parallel auf verschiedenen Rechnern ausgeführt
  - Ergebnis: gegenseitiges „Stehlen“ der Seite bei Datenzugriffen
- Maßnahmen zur Effizienzsteigerung erforderlich
  - ausgefeiltere Kohärenzprotokolle
  - paralleles Lesen erlauben
  - Abschwächung der Speicherkonsistenz-Anforderungen

Verteilter gemeinsamer Speicher (5)

- Resumee
  - Verteilter gemeinsamer Speicher nur bei sehr enger Rechnerkopplung sinnvoll
    - spezielle Hardware-Unterstützung
    - kurze Latenzen
    - schnelle Datenübertragung

    $\Rightarrow$ Höchstleistungsrechner
  - In lose gekoppelten Systemen ist reine Nachrichtenübertragung vorzuziehen
    - Kommunikation nicht über gemeinsamen Speicher, sondern über Prozeduraufrufe

Nachrichten zusammenstellen / auspacken

- *marshalling*

  to arrange (troops, things, ideas, etc.) in order; array; dispose

  - die einzelnen Datenelemente in einen Nachrichtenpuffer gepackt anordnen:
    1. für die **Serialisierung** verstreut vorliegender Daten sorgen
    2. die vereinbarte **Repräsentation** (Typ) der Daten gewährleisten
  - je nach Herangehensweise sind Referenzen aufzulösen und umzuwandeln
    - wenn die referenzierten Strukturen *call-by-value* zu übertragen sind
  - die so zusammengestellte Nachricht geht per IPC an den Empfangsprozess

- *unmarshalling*
  - die *inverse Funktion* auf Empfangsseite
  - Nachricht entpacken und die ursprünglichen Datenelemente wieder herstellen
  - hierbei ggf. eine andersartige Repräsentation auf der Zielseite berücksichtigen

Serialisierung/Deserialisierung

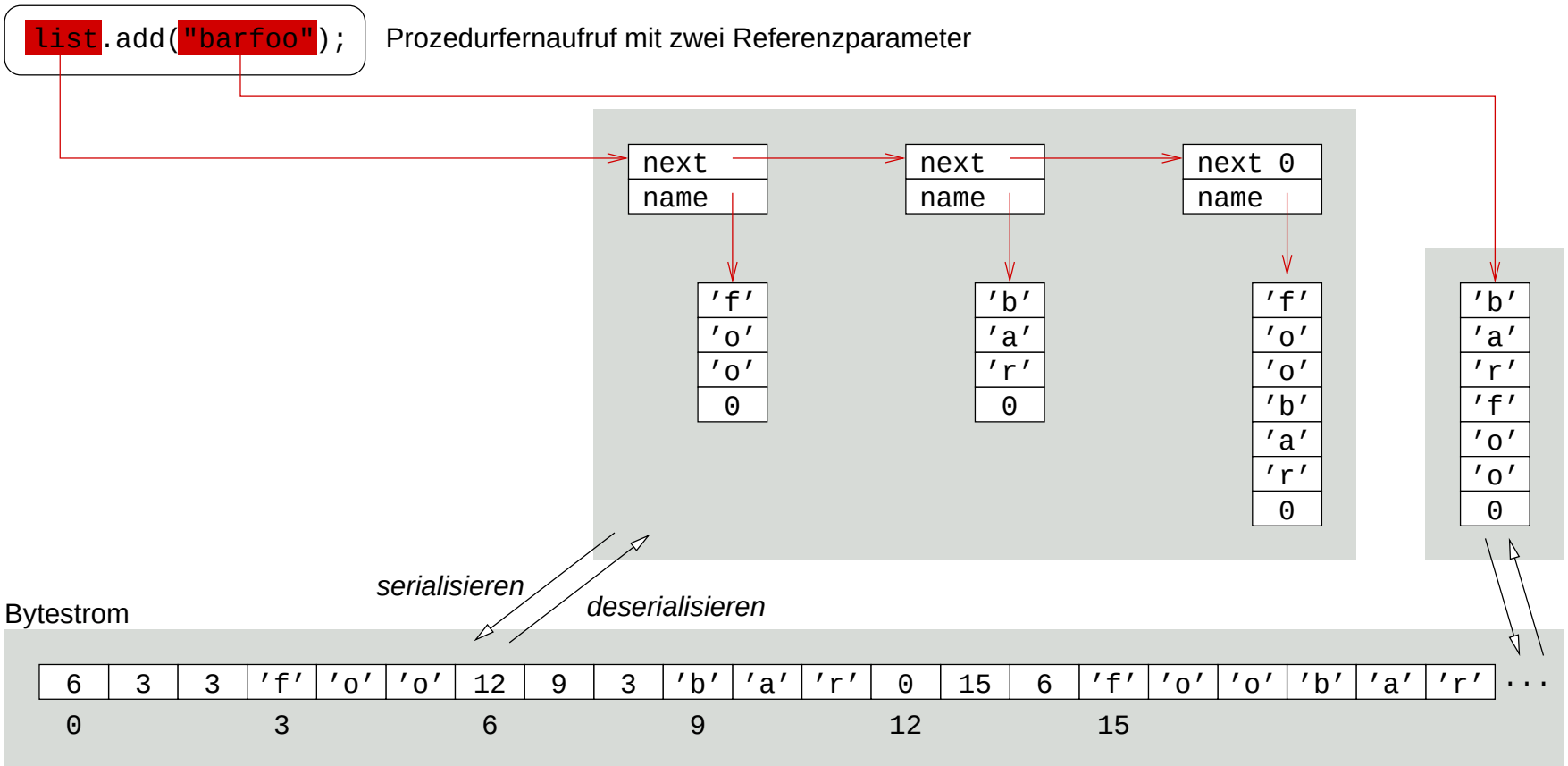
- geläufig ist, **verzeigerte Datenstrukturen** *call-by-value* (*-result*) zu übergeben
  - zum Marshalling wird eine Beschreibung des logischen Aufbaus strukturierter/dynamischer Daten benötigt (= Typbeschreibung)
  - alle Referenzen werden aufgelöst und die referenzierten Objekte kopiert
  - nach dem Empfang werden die Datenstrukturen wieder rekonstruiert

- zwei Sorten von Zeigern sind dabei zu unterscheiden:
  innere Referenzen: die Verkettungszeiger rekursiver Datenstrukturen
  - sind vergleichsweise unproblematisch: „logische Zeiger“ vergeben

  äußere Referenzen: Zeiger von außen hinein auf einzelne Verbundelemente
  - die relative Position der Verbundelemente kann sich ändern: Relokation

- der Aufwand kann je nach Art/Aufbau der Datenstrukturen beträchtlich sein

Packen/Entpacken strukturierter Daten



Objektorientierung „erschwert“ Automatisierung

- Referenzen vom Typ einer Oberklasse zeigen ggf. auf Instanzen von Unterklassen
  - eine *abstrakte Oberklasse* sagt wenig/nichts aus über die Objektstruktur

    $\Rightarrow$ in der Fernaufrufschnittstelle ist die konkrete Ausprägung des zu übertragenden Objekts unbekannt
  - erst die spezialisierende Unterklasse enthält die erforderlichen Strukturinformationen
- der *tatsächliche Typ* des Parameters ist erst zur Laufzeit bekannt
  - automatische Generierung des Stubs mit der Marshalling-Funktion ist zur Übersetzungszeit nicht möglich
  - statt dessen „spezialisierte Zusammenstellung“ der Fernaufrufnachrichten
    - erfordert eine vollständige Analyse des (zu verteilenden) Quellprogramms
  - Alternative: Analyse der Parametertypen zum Zeitpunkt des Aufrufs — *Reflection*

Datenrepräsentation

Heterogenität – die einzelnen in Nachrichten übertragenen Elemente können Werte unterschiedlichster (elementarer) Datentypen repräsentieren.

- *Speicherung* wie auch *Darstellung* von Instanzen dieser Typen ist nicht in allen Rechnern identisch:
  - natürliche/ganze Zahlen: vorzeichenbehaftet, {1,2}er-Komplement
  - Fließkommazahlen: Basis, Mantisse, Exponent
  - Zeichensätze: ISO-8859-Familie (ASCII), BCD, EBCDIC, Unicode
  - Speicherreihenfolge: *big endian* vs. *little endian*

$\Longrightarrow$ Daten sind ggf. zu konvertieren, damit kooperierende Prozesse funktionieren!

Konvertierungsoptionen

beidseitig *external data representation* (XDR)

- zu sendende Daten in eine **kanonische Darstellung** umkodieren *und*
- empfangene („kanonische“) Daten in die lokale Darstellung umkodieren
- Problem: nutzloser Mehraufwand im Falle „gleichartiger“ Rechner

sendeseitig „*sender makes it right*“

- zu sendende Daten in die empfangsseitige Darstellung ggf. umkodieren
- Problem: Mehrteilnehmerkommunikation, Weiterleiten von Nachrichten

empfangsseitig „*receiver makes it right*“

- empfangene Daten in die lokale Darstellung ggf. umkodieren

→ *endian tag*

External Data Representation — XDR

entwickelt von Sun Microsystems, *RFC 1014*, 1987

- sprachbasierter Standard zur Beschreibung und Kodierung von Daten
  - der ISO/OSI **Präsentationsschicht** (*presentation layer*, 6) zugeordnet
  - **implizite Typung**, nicht explizit wie bei ASN.1
    - die Typen der einzelnen Daten ind er nachricht ergeben sich implizit aus dem Typ der aufgerufenen Schnittstelle
    - bei ASN.1 werden sie explizit in der Nachricht mitgeschickt
  - Annahme: Bytes bzw. Oktets (d.h. Einheiten von 8 Bits) sind portabel
- Daten werden als „Vielfaches von vier Bytes“ (32 Bits) repräsentiert (*Tradeoff* „Vier“ — Groß genug als effiziente Lösung für die meisten Maschinen, mit Ausnahme von 64-Bit Architekturen, und klein genug als vertretbarer Mehraufwand zur Repräsentation der kodierten Daten.)
  - Füllbytes (0 – 3) ergänzen den Datenstrom immer zum Vielfachen von 4
- die Reihenfolge (der „Byte-Sex“) ist *big endian*

XDR „*Considered Harmful*“?

- Virtualisierung:
  Jede Maschine, deren physikalische Darstellung mit der durch XDR vorgegebenen virtuellen übereinstimmt, wird effizienter arbeiten als jene, bei der die physikalische von der virtuellen Darstellungsform stark abweicht:
  - Speicherreihenfolge („Byte-Sex“)

	endian		endian		
Sun	*big*	$\leftrightarrow$	*big*	m68K	„optimal“
IBM 370	*big*	$\leftrightarrow$	*little*	Z80	
Alpha	*little*	$\leftrightarrow$	*big*	R10000	
VAX	*little*	$\leftrightarrow$	*little*	x86	„suboptimal“

  - Verschnitt („interne Fragmentierung“)

Sprachintegration von Fernaufrufen

- Fernaufrufsysteme lassen sich in zwei Hauptkategorien einteilen:
  1. in einer Programmiersprache **integriertes Konzept** . . . . . Argus
     - internes Wissen über Datentyp- und Laufzeitmodell ist verfügbar
     - der Übersetzer agiert gleichzeitig als *Stub-Generator* (*stub generator*)
     - umfassende Analysen, Optimierungen und Automatismen werden möglich
  2. von einer Programmiersprache **separiertes Konzept** . . . . . CORBA
     - das Schnittstellenverhalten entfernter Prozeduren wird in einer eigenen „*Interface Definition Language*“ explizit beschrieben (IDL)
     - fehlendes internes Wissen schränkt Analysen, Optimierungen etc. ein
  3. **teilweise integriertes Konzept** . . . . . Java RMI
     - der Compiler kennt Konzepte des Fernaufrufs (Remote-Schnittstellen, etc.) nicht
     - in den Java-Standards sind die Konzepte definiert
     - Unterstützung für RMI integraler Bestandteil der Laufzeitumgebung
- nur das integrierte Konzept ($\rightarrow$) definiert eine einheitliche Semantik

Entlastung von Routineaufgaben

- die Stubs sorgen für eine **lose Kopplung** zweier Ausführungsumgebungen:

  client stub: den Ort des Dienstabieters (beim Namensdienst) erfragen

  1. *Marshalling* und versenden der Anforderungsnachricht

  ⋮ die Durchführung der angeforderten Operation abwarten

  2. empfangen und *Unmarshalling* der Antwortnachricht

  server stub: den Ort des Dienstabieters (dem Namensdienst) angeben

  1. empfangen und *Unmarshalling* der Anforderungsnachricht
  2. durchführen der angeforderten Operation (lokaler Aufruf)
  3. *Marshalling* und versenden der Antwortnachricht

- **Stub-Generatoren** erzeugen die dafür notwendigen Programmsequenzen

Transparenz von Fernaufrufen

- Verteilungstransparenz ist nur bedingt erreichbar, trotz integriertem Konzept:
  - syntaktische Unterschiede (in der Schnittstelle) werden vermieden
  - Parameterübergabe, *Marshalling* und *Unmarshalling* wird verborgen
  - Nachrichtenpuffer und Fäden werden erzeugt, verwaltet und entsorgt
  - Stubs werden automatisch erzeugt — was ist denn sonst noch nötig?
- Fernaufrufe sind fehleranfälliger als konventionelle lokale Prozeduraufrufe

entfernt	ein Netzwerk, ein anderer Rechner und ein anderer Prozess
¬ entfernt	kein Netzwerk, kein anderer Rechner und kein anderer Prozess

- verteilte Systeme unterliegen einem radikal anderem **Fehlermodell**

Zustellungsgarantien

- *request-reply*-Protokolle eröffnen Optionen für Fehlertoleranzmaßnahmen:

1. Wiederholung der Anforderungsnachricht . . . . . *request retry*
   - bis die Antwort eintrifft oder ein Anbieterausfall angenommen wird
2. Filterung der Anforderungsduplikate . . . . . *duplicate suppression*
   - wenn die Anforderung empfangen wurde und noch in Arbeit ist
3. Wiederholung der Antwortnachricht . . . . . *reply retry*
   - bis die nächste Anforderung oder eine Bestätigung eintrifft

- Kombinationen dieser Optionen begründen verschiedene Aufrufsemantiken

Fehlertoleranzmaßnahme vs. Aufrufsemantik

Option			**Semantik**	
request retry	*dupl. supr.*	*re-execute reply retry*		
Nein	—	—	„kann sein“	*maybe*
Ja	Nein	*re-execute*	„wenigstens einmal“	*at-least-once*
Ja	Ja	*reply retry*	„höchstens einmal“	*at-most-once*

Aufrufsemantiken (1)

- *maybe*: kann sein, der Aufruf wurde ausgeführt oder auch nicht
  - der Klient hat im Fehlerfall keine Gewissheit über die korrekte Ausführung
- *at-least-once*: mindestens einmal, die mehrfache Ausführung ist möglich
  - der Klient erwartet die Antwort innerhalb einer vorgegebenen Zeitspanne
    - jeder Fernaufruf wird mit einem *Timeout* versehen
    - nach der dadurch definierten Pause erfolgt die Aufrufwiederholung
    - die Anzahl der Wiederholungen ist (üblicherweise) begrenzt
    - mit Erreichen der max. Anzahl tritt eine *Ausnahmesitutation* ein
  - der Anbieter muss **idempotente Operationen** exportieren

Aufrufsemantiken (2)

at-most-once: höchstens einmal, mehrfache Ausführung ist ausgeschlossen

- vergleichsweise aufwändiges, speicherintensives Verfahren:
  - jedem neuen Aufruf wird eine eindeutige *Aufrufkennung* gegeben
  - Wiederholungen kommen mit derselben Kennung und werden verworfen
    - Möglichkeit der Ausnahmesituation entsprechend *at-least-once*
  - Antworten werden gespeichert und bei Wiederholungen zurückgeliefert
  - ein neuer Aufruf ist Anlass, gespeicherte Antworten zu entsorgen

    (Ansonsten muss der Anbieter hin und wieder beim Klienten nachfragen, ob dieser noch „lebt“, um so ein Kriterium für die Entsorgung gespeicherter Antwortnachrichten zu gewinnen.)
- die Ausführung erfolgt nur, wenn keine Rechnerausfälle vorliegen

Aufrufsemantiken (3)

- *last-of-many*: akzeptiert nur die Antwort zum jüngst zurückliegenden Aufruf
  - jeder Aufruf, auch der wiederholte, erhält eine eindeutige Kennung
  - jede Antwort trägt die Kennung des zugehörigen Aufrufs
    - der Klient verwirft Antworten mit nicht mehr aktueller Aufrufkennung
    - eine Variante von *at-least-once*, falls keine Idempotenz vorliegt
  - die wiederholte Ausführung der Operationen ist ggf. weiterhin problematisch
  - zumindest können die Klienten aber mit „aktuellen“ Daten weiterarbeiten

    (Im Gegensatz dazu liefert *at-most-once* ggf. „veraltete“ Daten, die zur ersten Ausführung wiederholt abgesetzter Aufrufe korrespondieren. Dieser Fall kann vorliegen, wenn die vom Anbieter verwalteten Daten von mehreren Klienten be- und verarbeitet werden (*information sharing*).)

Aufrufsemantiken (4)

- *exactly-once*: genau einmal, entspricht der Semantik lokaler Aufrufe
  - erfordert Transaktionskonzepte mit Wiederanlauf von Komponenten
    - gibt damit (eine gewisse) Garantie bei Systemfehlern bzw. -ausfällen
  - nach Wiederanlauf sind Aussagen zur Operationsdurchführung „unscharf“:

    Imagine, for example, a chocolate factory, in which vats of liquid chocolate are filled by having a computer set a bit in some device register to open a valve. After recovering from a crash, there is no way for the chocolate server to see if the crash happened one microsecond before or after the bit was set. (Tanenbaum, Computer Networks, 1989)

  - wünschenswerte, ideale Semantik, die in „Reinform“ jedoch unerreichbar ist

Idempotente Operationen

- Idempotenz „idem“ (*lat.*) dasselbe; wenn die wiederholte Ausführung derselben Operation (mit denselben Parameterwerten) durch den Dienstanbieter immer den Effekt einer einmaligen Ausführung besitzt
  - wiederholte Ausführungen sind möglich im Falle ungefilterter Duplikate
    - kritisch sind z.B. Schreiboperationen auf Dateien (`write(2)`)
    - ebenso Operationen, die eine Folge (Liste) um Elemente erweitern[3]
  - ein **zustandsfreier Dienstanbieter** (*stateless server*) ist gefordert
    - bekannter Vertreter ist das Sun *Network File System* (NFS)
  - bei zustandsbehafteten Dienstanbietern ist Duplikatelimination notwendig

Fehlersemantiken

Aufrufsemantik	**Fehlerart**							
	fehlerfreier Ablauf		Verlust von Nachrichten		zusätzlicher Ausfall von Anbieter		Klient	
	Aus.	Erg.	Aus.	Erg.	Aus.	Erg.	Aus.	Erg.
maybe	1	1	0/1	0/1	0/1	0/1	0/1	0/1
at-least-once	1	1	≥ 1	≥ 1	≥ 0	≥ 0	≥ 0	0
at-most-once	1	1	1	1	0/1	0/1	0/1	0
exactly-once	1	1	1	1	1	1	1	1

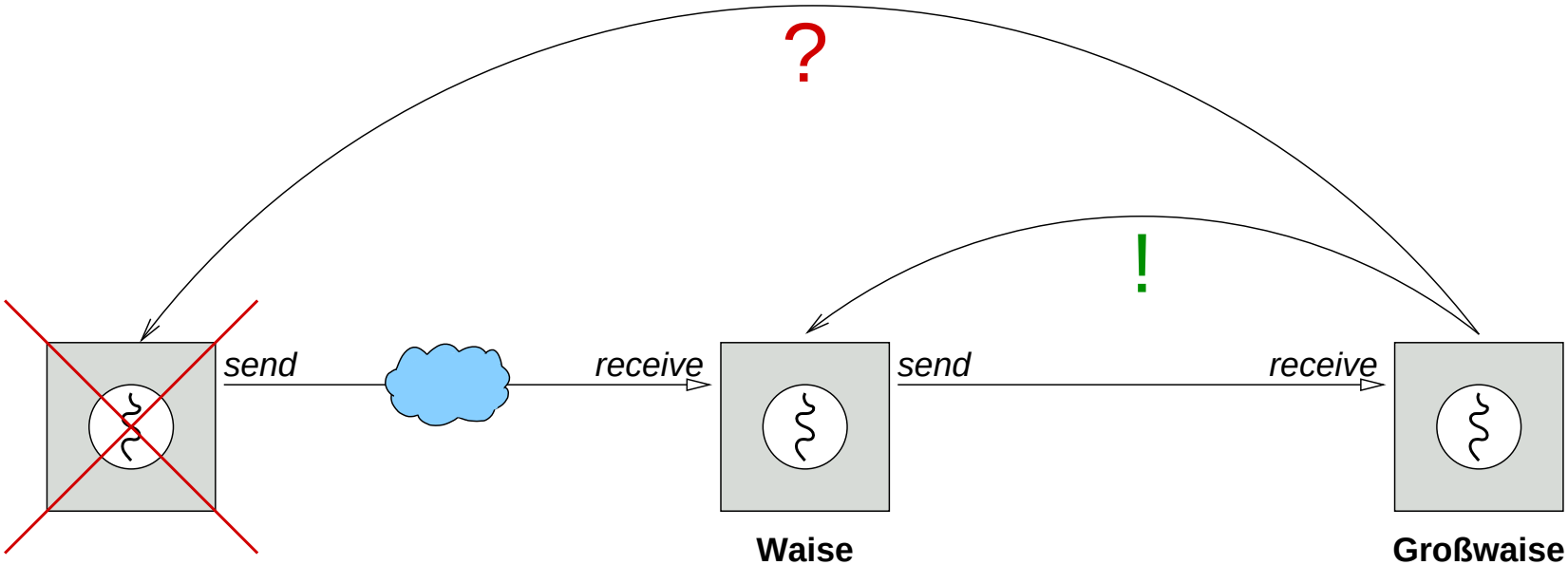
Aus.= Ausführung, Erg.= Ergebnis; die jeweilige Anzahl ist angegeben

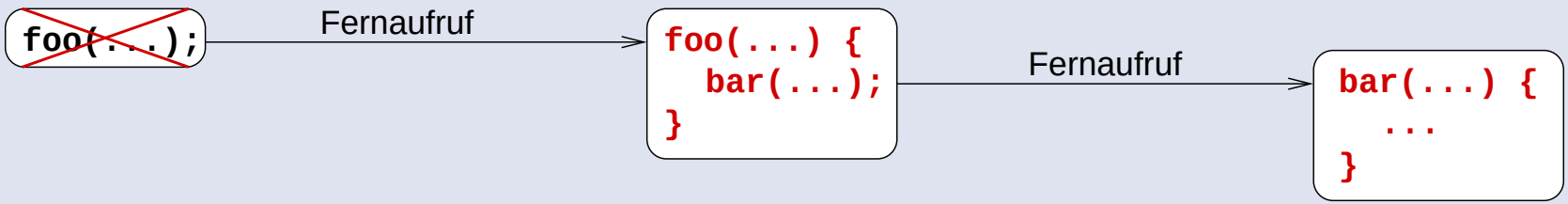
Verwaister Aufruf (1)

- *Orphan* ein Klient, der den Aufruf (z.B. wegen eines ggf. zu kurzen *Timeout*) abgebrochen hat und nicht mehr am Ergebnis interessiert ist oder der mittlerweile überhaupt nicht mehr zur Verfügung steht

- Maßnahmen, um unnötige Arbeit des Dienstanbieters zu vermeiden:
  1. zusätzliche Zeitüberwachung des Klienten durch den Dienstanbieter
     - die *Timeout*-Wahl ist klientenabhängig ($T_S \gg T_C$)
  2. pro Klienten einen Ausfallzähler beim Dienstanbieter verwalten
     - Abbruch aller alten Aufrufe nach Ausfall und Wiederanlauf des Klienten
  3. zusätzliche direkte Statusanfragen an den Klienten senden
     - schnelle Waisenerkennung, sofern der Ausfall diagnostizierbar ist
- besondere Schwierigkeiten bereiten „Fernaufrufketten“: **Transitivität**

Verwaister Aufruf (2)





Waisenbehandlung (1)

- *extermination* (Vertilgung, Ausrottung, Wegschaffung)
  - nach Wiederanlauf wird (beim Klienten) auf ausstehende Fernaufrufe geprüft
    - den betreffenden Anbietern gehen sodann Abbruchanforderungen zu
    - rekursives Vorgehen, wegen der Möglichkeit von „Großwaisen“
  - Klientenstümpfe müssen „Buch“ führen über alle Fernaufrufe
    - *Log* anlegen vor der Anforderung und zerstören nach Empfang der Antwort

- *expiration* (Verscheiden, Ablauf)
  - der Anbieter gibt dem (Fern-) Aufruf ein *Zeitquantum* zur Ausführung
    - mit Ablauf erbittet der Anbieter ein weiteres Zeitquantum vom Klienten
    - im Fehlerfall bricht der Anbieter den Aufruf ab bzw. terminiert er
  - nach Wiederanlauf ist der erste Fernaufruf um ein Zeitquantum zu verzögern
  - *Leasing*: zeitlich begrenzt gültige Referenzen - erstmalig realisiert in *Jini*

Waisenbehandlung (2)

- *reincarnation* (Wiederverkörperung, Wiedergeburt)
  - bei Wiederanlauf eines Klients wird eine neue *Epoche* eröffnet
    (Epochen entstehen dadurch, dass die Zeit in sequentiell nummerierte Abschnitte aufgeteilt wird.)
    - der Beginn einer Epoche wird allen Maschinen mitgeteilt (*broadcast*)
    - auf den Maschinen werden daraufhin alle Anbieter (Prozesse) terminiert
  - jede Anforderungs- und Antwortnachricht enthält eine Epochenkennung
    - damit können unerwartete Antworten von Waisen herausgefiltert werden

- *gentle reincarnation* (sanfte Wiedergeburt)
  - zum Epochenbeginn versucht der Anbieter „seinen“ Klienten zu lokalisieren
  - ist der Klient nicht lokalisierbar, werden die Prozesse terminiert

Ausnahmebehandlung — *Exception Handling*

- volle Verteilungstransparenz erreichen zu können, ist reine Illusion
  - auch *exactly-once* kann nur die erkennbaren Fehler maskieren
  - in den anderen Fällen ist die Rückkehr vom Aufruf keinesfalls garantiert (Es ist der Fernaufrufmechanismus selbst, der einen Klienten z.B. bleibend verklemmen kann. Sicherlich kann auch die Implementierung des per Fernaufruf in Anspruch genommenen Dienstes Verklemmungsursache sein, was aber hier nicht zur Debatte steht.)
- die Stubs (auf beiden Seiten) sollten Ausnahmesituationen anzeigen
  - Ausnahmen der Anbieterseite werden als Rückruf zum Klienten propagiert
  - Ausnahmen der Klientenseite werden konventionellerweise „hochgereicht“
- die Anwendungen sollten Maßnahmen zur Fehlertoleranz beinhalten

Zusammenfassung

- Fernaufrufe sind konventionellen Prozeduraufrufen nur ähnlich, nicht gleich:
  - Parameter{arten, Übergabe}, Gültigkeitsbereiche und Speicheradressen
  - Fehlermodell, Zustellungsgarantien, Aufrufsemantiken, verwaiste Aufrufe
  - die Latenz entfernter Aufrufe ist um Größenordnungen höher: *Promise*
- Verteilungstransparenz kann nicht wirklich (durchgängig) erzielt werden
  - *exactly-once* ist nicht (bzw. nur mit Abstrichen) zu verwirklichen
  - Ausnahmen sind zu behandeln, die nur in verteilten Systemen auftreten
  - Fernaufrufe machen fehlertolerante Anwendungssoftware keinesfalls obsolet
- der Unterschied zu konventionellen Aufrufen ist (doch) explizit zu machen

Überblick

Verteilte Anwendungen, Interaktionsformen

- Klassifikation von Interaktionsformen
  - explizit
  - implizit
  - orthogonal
  - nicht-orthogonal
  - uniform
  - nicht-uniform
  - transparent
  - nicht-transparent

explizite, orthogonale Interaktion

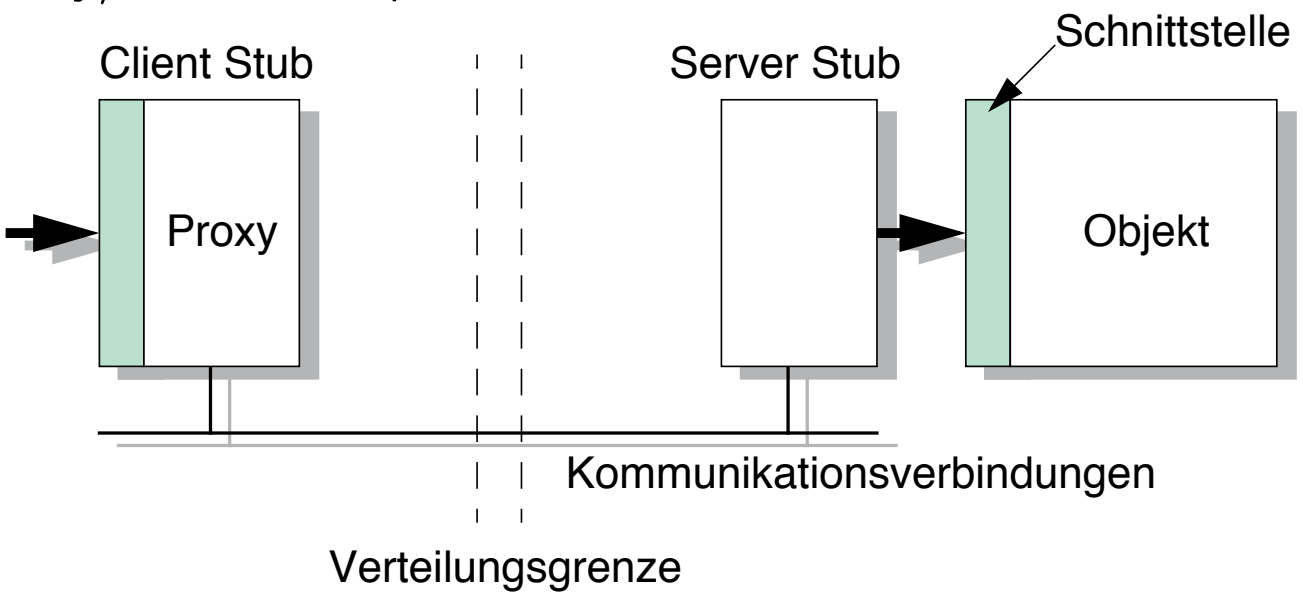
- weit verbreitete Vorgehensweise
- von klassischen Interprozesskommunikations-Mechanismen geprägt
  - Nachrichten (Datagramm-Sockets, Messages, -)
  - Verbindungen (Stream-Sockets, Pipes, -)

\+ Vorteile
- meist weit verbreitete, etablierte Infrastruktur vorhanden
- kein „unsichtbarer“ Overhead

– Nachteile
- Programmierung aufwändig
- Bruch im Programmierparadigma (vor allem bei Objektorientierung)
  - Serialisierung von Parametern, Verlust von Typ-Information
- Verteilung wird durch die Programmierung „fest verdrahtet“
  - Software sehr unflexibel in Bezug auf Änderungen

implizite, nicht-orthogonale Interaktion

- Interaktion zwischen lokalen und verteilten Funktionen oder Objekten unterscheidet sich prinzipiell nicht
  - nur ein Interaktionsmechanismus: Funktions-/Methodenaufruf
- grundlegendes Konzept: Remote Procedure Call
  - Verteilung wird durch Vermittler- oder Stellvertreterobjekte vor den Kommunikationspartnern (weitgehend) verborgen
  - Proxy/Stub-Prinzip



uniforme / nicht-uniforme Interaktion

- nicht-uniforme Interaktion
  - unterschiedliche Methodenaufrufe für lokale und remote-Referenzen
  - unterschiedliche Semantik bei der Parameterübergabe
    - by reference, by value
    - Problem: Übergabe von lokalen Objektreferenzen
- uniforme Interaktion
  - keinerlei Unterschied zwischen lokalen und remote-Aufrufen

transparente / nicht-transparente Verteilung

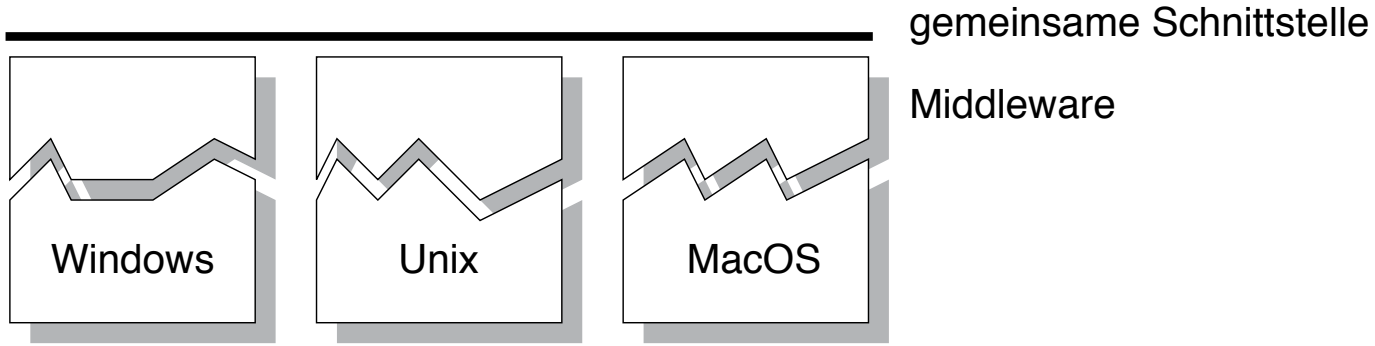
- volle Transparenz:
  Anwendungsentwickler sieht keinerlei Unterschied zwischen lokalen und verteilten Objekten
- Probleme:
  - im verteilten Fall können spezielle Fehler auftreten
    - unabhängiger Objektausfall → Remote Exception
  - verteilte Interaktion ist implizit signifikant teurer
    - Transparenz kann zu Ineffizienz führen
  - Verteilung ist häufig ein Entwurfskriterium
    - Verbergen der Verteilung in der Implementierung ist unsinnig
- Fazit:
  - bei der Programmierung sollte zwischen potentiell verteilten und definitiv lokalen Objekten unterschieden werden können

Herausforderungen

- Überwindung heterogener Hardware- und Softwarestrukturen
  - verschiedene Hardware
  - verschiedene Betriebssysteme
  - verschiedene Programmiersprachen
- Ortstransparenz
  - statische Konfiguration
  - Objektmigration
- Globale Dienste
  - z.B. Namensdienste, Transaktionsdienst, Persistenz, Kontrolle der Nebenläufigkeit

Middleware für verteiltes Programmieren

- Middleware - Software zwischen Betriebssystem und Anwendung



  - Bereitstellung von Abstraktionen und Diensten für verteilte Anwendungen
  - „Betriebssystemschnittstelle“ für ein verteiltes System

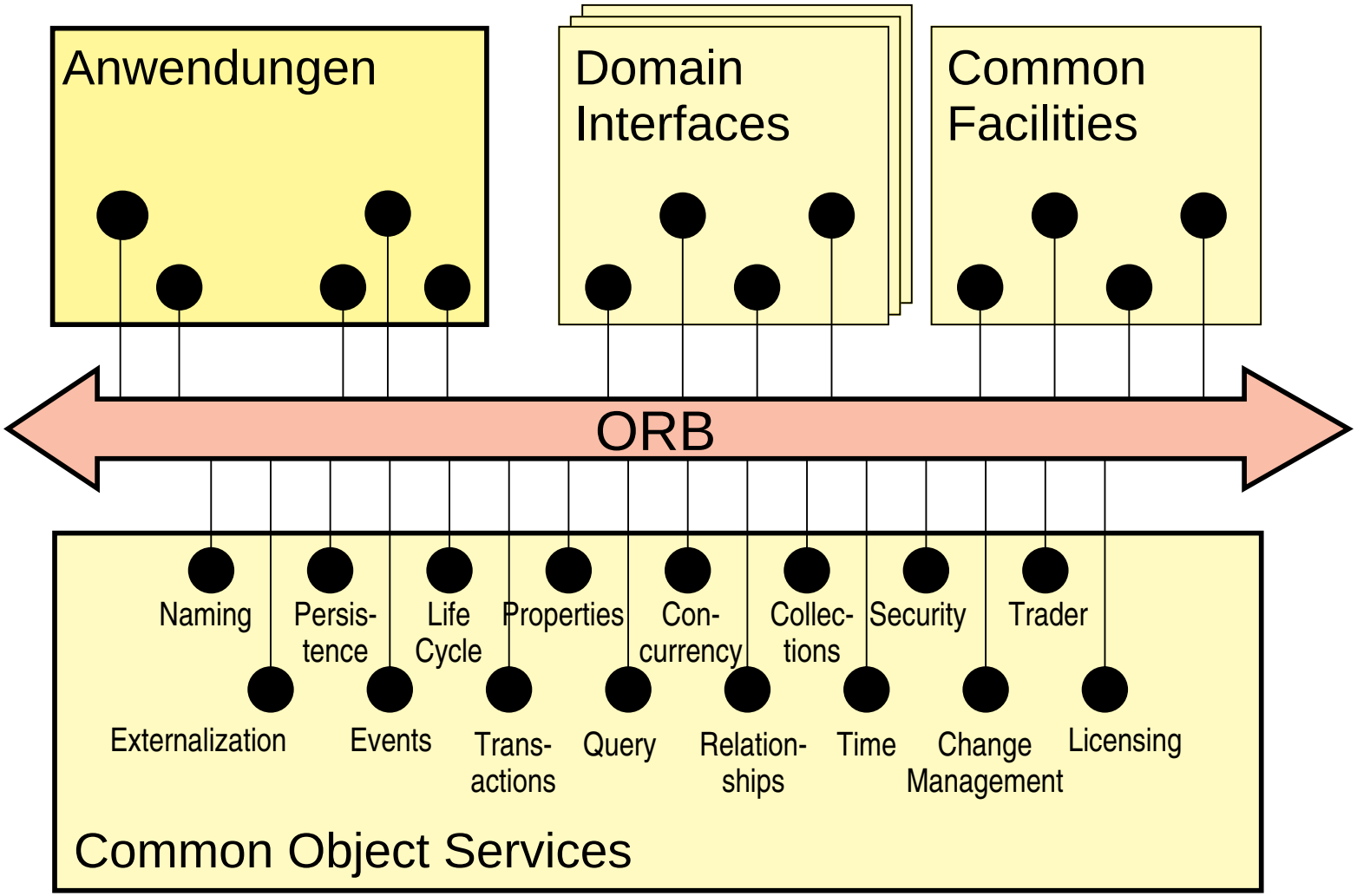
- CORBA
  - plattformunabhängige Middleware-Architektur für verteilte Objekte
  - Standard der OMG (Object Management Group)
  - erste umfangreiche Normierung von Middleware-Konzepten

Motivation für CORBA

- Verteilte objektbasierte Programmierung
  - verteilte Objekte mit definierter Schnittstelle
- Heterogenität in Verteilten Systemen
  - verschiedene Hardware-Architekturen
  - verschiedene Betriebssysteme und Betriebssystem-Architekturen
  - verschiedene Programmiersprachen

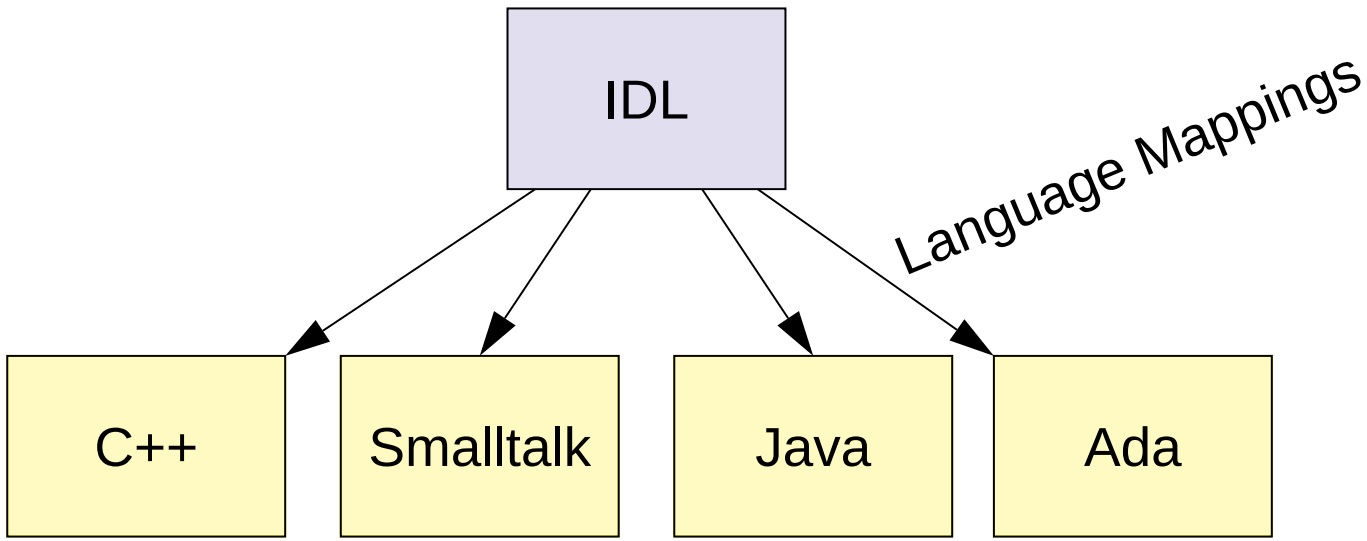
  $\Rightarrow$ Transparenz der Heterogenität
- Dienste im verteilten System
  - Namensdienst, Zeitdienst,...
- Was ist CORBA?
  - **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
    - plattformunabhängige Middleware-Architektur für verteilte Objekte
    - Sammlung von Standard-Dokumenten verwaltete durch die OMG
    - `http://www.omg.org/spec/CORBA/3.1`

OMA – Object Management Architecture



Interface Definition Language (IDL)

- Sprache zur Beschreibung von Objekt-Schnittstellen
  - unabhängig von der Implementierungssprache des Objekts
  - Sprachabbildung (Language-Mapping) definiert, wie IDL-Konstrukte in die Konzepte einer bestimmten Programmiersprache abgebildet werden
  - Language-Mapping ist Teil des CORBA standards
  - Language mappings sind festgelegt für:
    - Ada, C, C++, COBOL, Java, Lisp, PL/I, Python, Smalltalk
    - weitere inoffizielle Language-Mappings existieren,z.B. für Perl



Interface-Definition-Language

- IDL angelehnt an C++
  - geringer Lernaufwand
- eigene Datentypen
  - Basistypen
  - Aufzählungstyp
  - zusammengesetzte Typen
- Module
  - definieren hierarchischen Namensraum für Anwendungsschnittstellen und -typen
- Schnittstellen
  - beschreiben einen von außen wahrnehmbaren Objekttyp
- Exceptions
  - beschreiben Ausnahmebedingungen

Interface-Definition-Language

- Umsetzung von IDL in Sprachkonstrukte (z.B. Java)

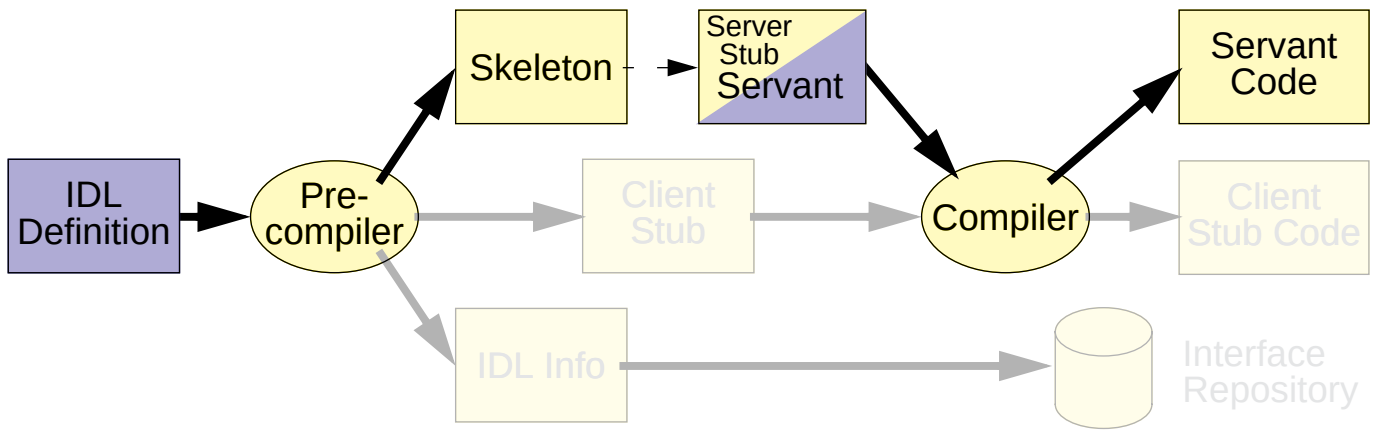
```
// IDL
module MyModule{
 interface MyInterface{
   attribute long lines;
   void printLine( in string toPrint );
 };
};
```

```
//Java
package MyModule;

public interface MyInterface extends ... {
 public int lines();
    public void lines( int lines );
 public void printLine( java.lang.String toPrint );
 ...
};
```

Objekterzeugung

- Ablauf auf Serverseite
  - Beschreibung der Objektschnittstelle in IDL
  - Programmierung des Server-Objekts in der Implementierungssprache
- Stub/Skeleton-Erzeugung

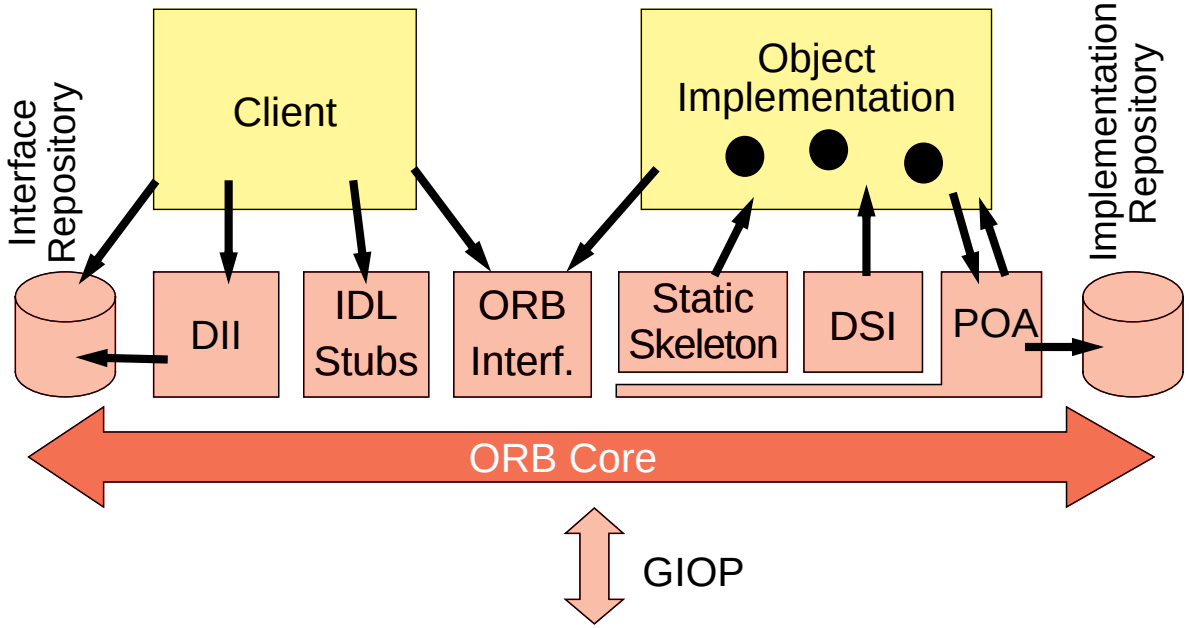


Objekte Binden

- Ablauf auf Serverseite
  - Registrierung des Objekts am **ORB**
    (bzw. dem Portable-Object-Adaptor (POA))
    - der ORB erzeugt eine Interoperable Object Reference (IOR)
- Binden des Clients an das Server-Objekt
  - Referenz auf Server-Objekt besorgen
    - Rückgabewert eines Methodenaufrufs
    - Ergebnis einer Namensdienst-Anfrage
    - von 'ausserhalb' des Systems: Benutzer kennt Referenz als String (IORs können in Strings konvertiert werden und umgekehrt)
- Erzeugung des Client-Stub & Methodenaufruf über Client-Stub

ORB Architektur

- **O**bject **R**equest **B**roker – ORB
  - ist das Rückgrat einer CORBA-Implementierung
  - vermittelt Methodenaufrufe von einem zum anderen Objekt
    - ... innerhalb/zwischen Adressräumen/Prozessen von (verschiedenen) ORBs



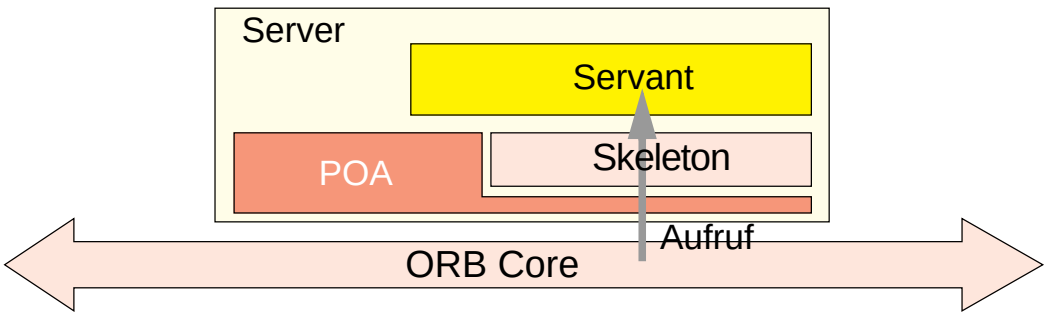
- Zentrale Komponenten eines ORB
  - mehrere interne Komponenten (teilweise nicht für Anwender sichtbar)

Portable-Object-Adaptor (POA)

- Aufgaben des Objektadapters
  - Generierung der Objektreferenzen
  - Entgegennahme eingehender Methodenaufruf-Anfragen
  - Weiterleitung von Methodenaufrufen an den entsprechenden Servant
  - Authentisierung des Aufrufers (Sicherheitsfunktionalität)
  - Aktivierung und Deaktivierung von Servants
- Portabilität von Objektimpl. zwischen verschiedenen ORBs
- Trennung von Objekt (CORBA-Objekt mit seiner Identität) und Objektimplementierung
  - eine Objektimplementierung kann **mehrere** CORBA-Objekte realisieren
  - Objektimplementierung kann bei Bedarf **dynamisch aktiviert** werden
  - persistente CORBA-Objekte (Objekte, die Laufzeit eines Servers überdauern)
  - eine Object Reference beim Client steht für ein bestimmtes CORBA Objekt – **nicht unbedingt** für einen bestimmten Servant!

Portable-Object-Adaptor (POA)

- Jeder Servant kennt seine POA-Instanz
  - POA-Instanz kennt die an ihm angemeldeten Objekte
  - POA stellt Kommunikationsplattform bereit und nimmt Aufrufe entgegen (für seine Objekte)



- Mehrere POA-Instanzen (mit unterschiedlichen Strategien) innerhalb eines Servers möglich
- Beispiel: Lifespan policy
  - `TRANSIENT` für Objekte, die Servant nicht überleben
  - `PERSISTENT` für Objekte, die POA und Servant überleben können

Überblick

Fault-Tolerant CORBA – Überblick

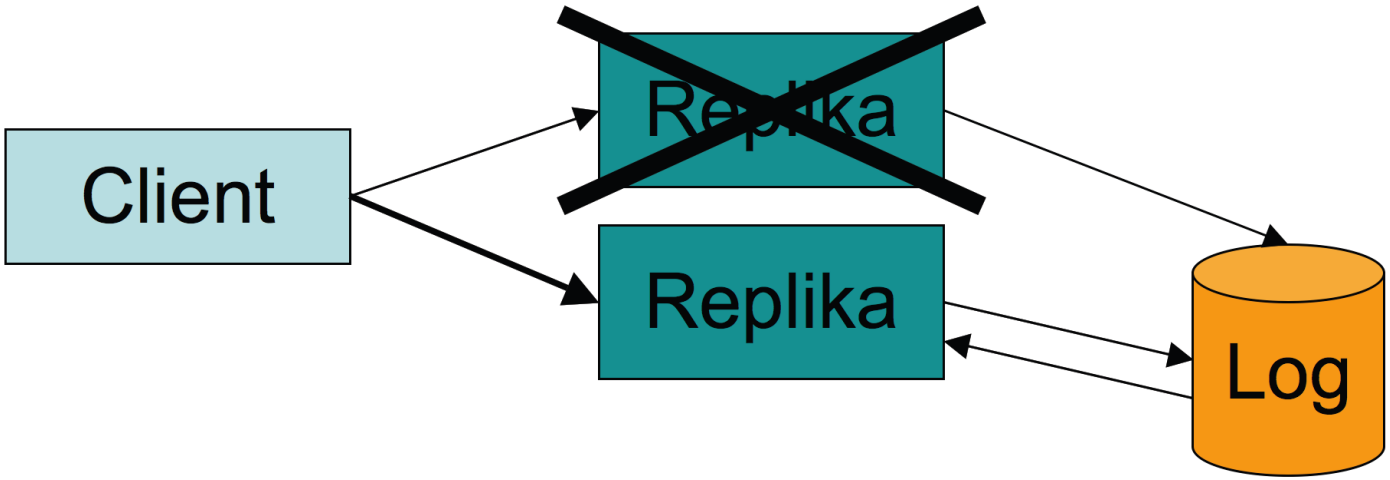
- Grundlagen bzw. Basismechanismen
- Einstellung von Fehlertoleranzeigenschaften
- Verwaltung von Replikaten
- Fehlermanagement
- Recoverymanagement
- Implementierung

Von CORBA zu FT-CORBA

- CORBA
  - CORBA spezifiziert keine Mechanismen für Redundanz
  - Zuverlässige Verbindungen basierend auf TCP/IP ermöglichen beschränkte Erkennung von Ausfällen
- Ziele von FT-CORBA
  - minimale Modifikation von Applikationen
    - (sowohl auf Client- wie auch auf Serverseite)
  - Unterstützung für Fehlertoleranz gegenüber von **fail stop** Fehlern
    - verschiedene Fehlertoleranz-Anforderungen
    - verschiedene Applikationen
    - verschiedene Mechanismen zur Fehlererkennung und -behandlung

Was muss eine Fehlertoleranzinfrastruktur leisten?

- Replikation von Objekten
- Erkennung von Ausfällen
- Logging von Anfragen und Zustandssicherung von Objektzuständen
- Zustandstransfer zur Initialisierung und Reinitialisierung von Objekten
- Transparente Verschattung von Ausfällen



Grundlagen: Redundanz

- Objekte bilden die Einheit der Replikation
- Starke Konsistenz
  - Alle Replikate haben den gleichen Zustand
  - Ermöglicht einfache Anwendungsentwicklung
    - Client interagiert mit Objekten
  - Stellt hohe Anforderungen an die Mechanismen der Infrastruktur
    - Alle Mechanismen zur Fehlertoleranz werden durch die Middleware bereitgestellt

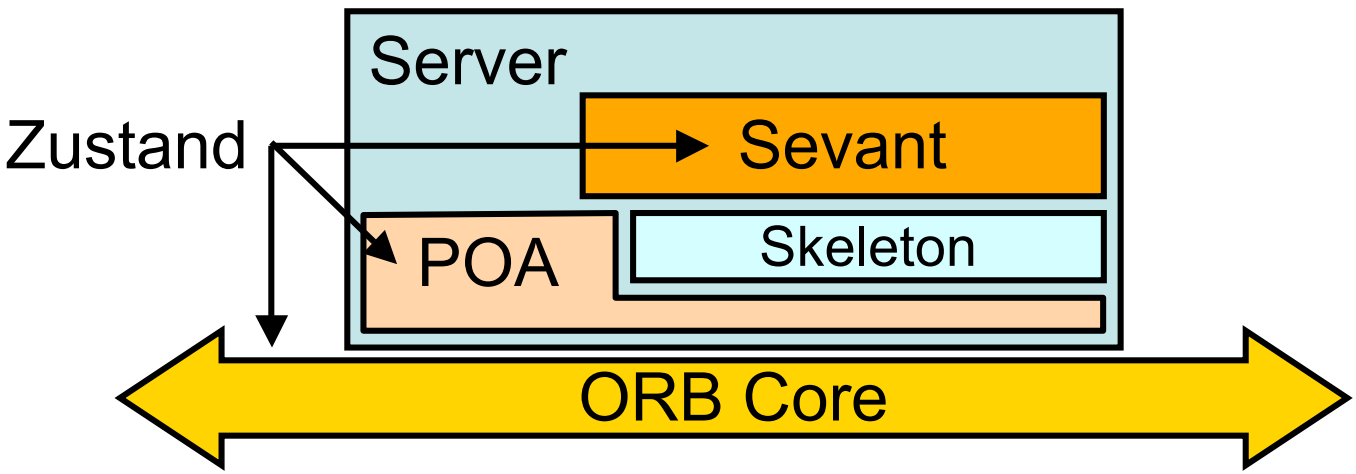
Grundlagen: Redundanz

- Objektgruppe
  - Menge aller Replikate eines Objektes bilden eine Objektgruppe
  - Jede Gruppe verfügt über eine **Object Group Reference (IOR)**
  - Zielsetzung
    - Replikationstransparenz
    - Fehlertransparenz

- Identität
  - CORBA-Objekte werden durch ihren Ausführungsort identifiziert
    - schwache Form von Identität
  - FT-CORBA benötigt eindeutige und ortsunabhängige Identifikation
    - Objektgruppen werden durch `FTDomainId` und `ObjectGroupId` identifiziert
    - einzelne Replikate durch `FTDomainId`, `ObjectGroupId` und Ort

Grundlagen: Zustand

- Zustand bildet die Menge an Informationen die zur Erhaltung von konsistenten Replikaten nötig ist
- Zustand ergibt sich durch das replizierte Objekt und die Infrastruktur (POA und ORB)



Grundlagen: Determinismus

- Replikation von Objekten erfordert deterministisches Verhalten
  - Aufrufe werden an alle Replikate in gleicher Reihenfolge zugestellt
  - Ausgehend von identischen Ausgangszuständen werden identische Endzustände erreicht
  - Objekt entspricht somit einem Zustandsautomat
- Problemfälle
  - objektexterne Informationen
    - z.B. Zeit, Zufallszahlen oder Aufruf an externen Informationsquellen
  - Koordinierung wenn parallel mehrere Threads ein Objekt verändern
    - z.B. Anforderung von Locks

Grundlagen: Art der Replikation

- **Active** Replication
  - Replikate bearbeiten alle Anforderungen
- **Passive** Replication
  - Es gibt ein aktives Replikat (*master*) welches Aufrufe bearbeitet
  - Alle anderen Replikate sind in Wartestellung

- Anwendungen besitzen unterschiedliche Fehlertoleranzanforderungen
  - Active Replication
    - Sehr kurze Verzögerungen im Fehlerfall
    - Hoher Aufwand unter anderem durch die erforderliche Gruppenkommunikation
  - Passive Replication
    - Längere Verzögerung bei Ausfällen
    - Geringer Aufwand zur Laufzeit
    - Schnellere Antwortzeit im Normalbetrieb

Grundlagen: Zuständigkeiten Serverseite

- Objektreplikation
- Verwaltung der System- und Fehlertoleranzanforderungen pro Objektgruppe
  - `Property Manager`-Schnittstelle
- Erzeugen von replizierten Objekten
  - `Generic Factory`-Schnittstelle
  - `Replication Manager`-Schnittstelle
  - Zustandstransfer
- Erkennung von Ausfällen

Grundlagen: Zuständigkeiten Clientseite

- Failover
  - Falls ein Server nicht antwortet wird es entweder erneut versucht oder ein anderer Server wird kontaktiert
  - Aufrufe werden nur einmal durch das replizierte Objekt ausgeführt (muss durch die Serverseite unterstützt werden)
- Adressierung
  - Veraltete Referenzen werden in Kooperation mit dem Server ersetzt
    - Server liefert aktuelle Version
- Ausfall der Verbindung
  - z. B. kein Replikat ist erreichbar
  - Anwendung wird benachrichtig

Grundlagen: Kontrolle und Verwaltung

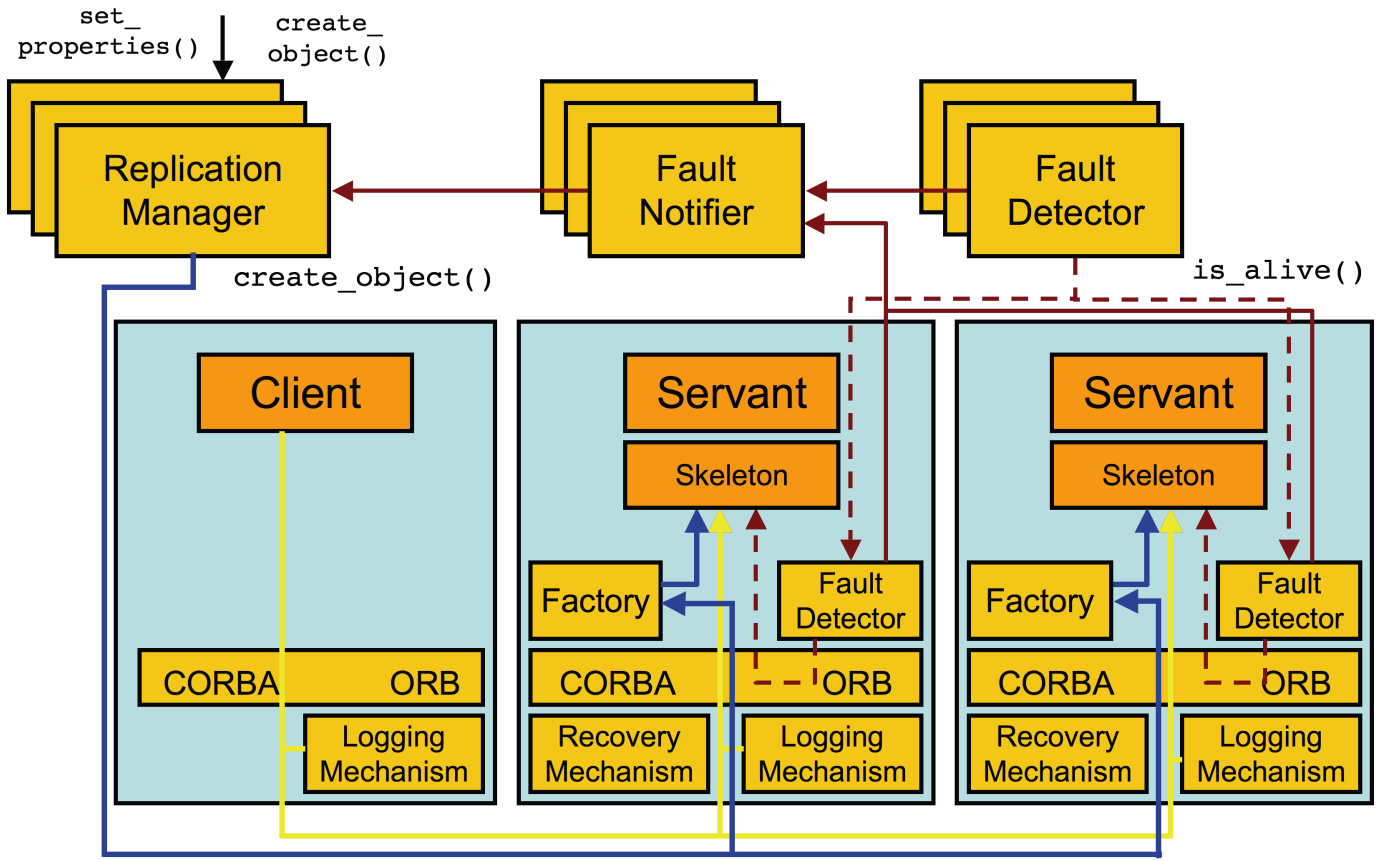
- Infrastruktur kontrolliert Fehlertoleranz
  - Automatische Erzeugung von Replikaten
  - Automatische Erhaltung der Konsistenz
- Applikation kontrolliert Fehlertoleranz (nur in Spezialfällen nötig!)
  - Applikation lenkt die Erzeugung und Platzierung von Replikaten
  - Applikation gewährleistet die Konsistenz
- Zuständigkeit: Fault Tolerance Domain
  - Menge von Rechnern zur Bereitstellung von fehlertoleranten Applikationen
  - Zentrale Komponente bildet der Replication Manager
    - steuert Erzeugung und Platzierung von Replikaten

Limitationen des FT-CORBA Standards

- Determinismus bleibt Aufgabe des Anwendungsentwicklers
- Keine explizite Unterstützung für
  - Behandlung von Netzwerkpartitionierungen
  - bösartige Fehler
  - Softwarefehler und Designfehler
- Interoperabilität
  - Alle Replikate eines replizierten Objektes müssen die gleiche Infrastruktur nutzen

Es ist Aufgabe der Hersteller hier individuelle Lösungen anzubieten!

Architekturübersicht

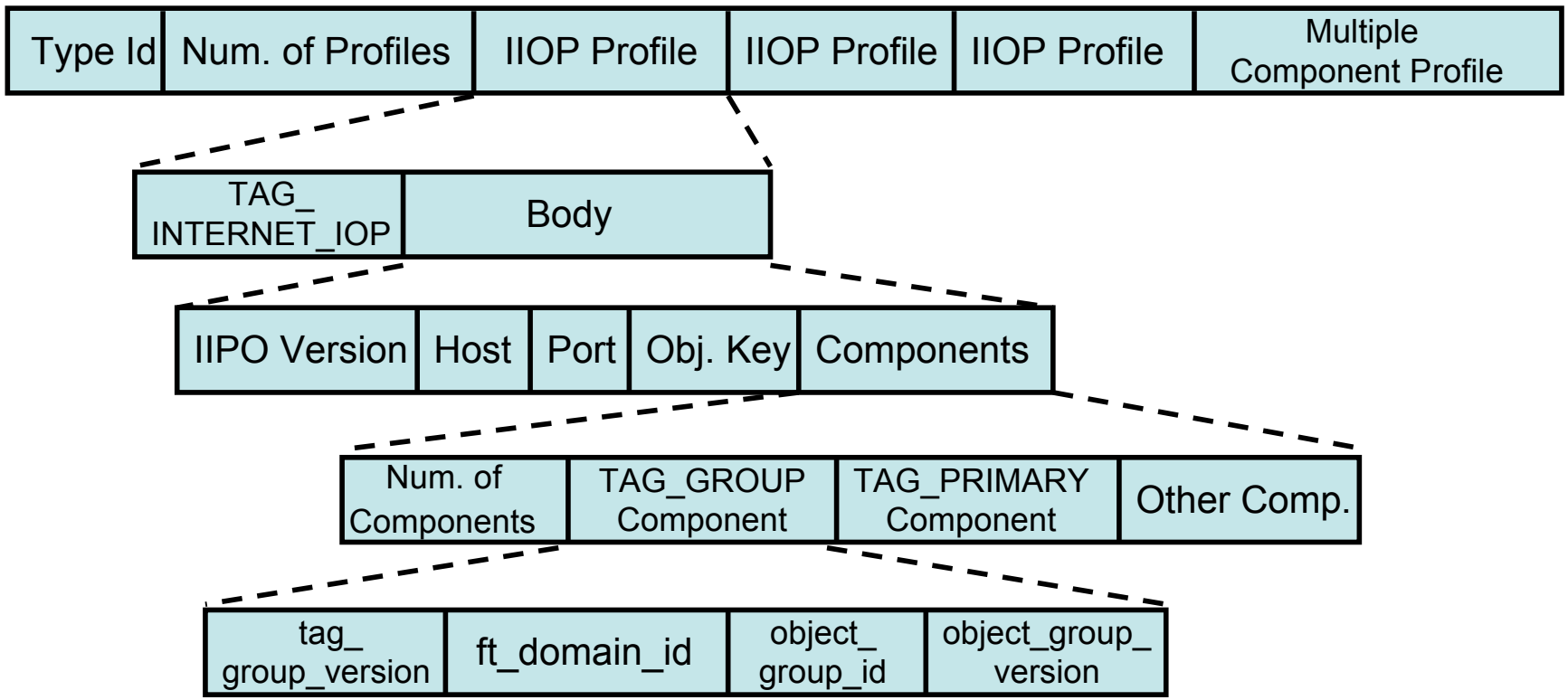


Adressierung

- Interoperable Object Group Reference (IOGR)
  - Eine IGOR besitzt mehrere IOR Profile
  - Jedes Profile setzt enthält Informationen zur Identität des Objektes
    - `FTDomainId` identifiziert die Domäne
    - `ObjectGroupId` identifiziert das replizierte Objekt (innerhalb der Domäne)
    - `ObjectGroupRefVersion` legt die Version der Referenz fest
  - Maximal ein Profil enthält die Komponente `TAG_PRIMARY` zur Identifikation des primären Replikats.
    - Achtung, muss nicht aktuell sein!
- Was wird adressiert?
  - direkte Addresssierung der Replikate
  - Addessierung von Gateways

Adressierung

Interoperable Object Group Reference (IOGR)



Adressierung

- Aktualität von IOGR
  - Problem: Objektreferenzen können veralten
    - z.B einzelne Replikate fallen aus, oder es kommen neue Replikate hinzu
  - Lösung: Version der Client-Referenz wird bei Anfragen übertragen
    1. Versionsinformation aus der Referenz wird als `GROUP_VERSION Service Context` der Anfrage beigefügt
    2. Server entnimmt Anfragen die Versionsinformation
    3. Ist die Version des Clients aktuell wird die Anfrage verarbeitet
    4. Ist die Version des Clients veraltet wird eine `LOCATE_FORWARD_PERM` Exception erzeugt
    5. Ist die Version des Servers (anscheinend) veraltet wird der Replication Manager nach der aktuellen Referenz gefragt

Verhalten bei Ausfällen

- Wiederholung von Aufrufen bei Ausfällen von Replikaten
- Auf ORB-Transportebene gibt es eine der folgenden Exceptions
  - `COMM_FAILURE, TRANSIENT, NO_RESPONSE, OBJ_ADAPTER`
  - Status ist `COMPLETED_MAYBE`
- Problem
  - Ohne weitere Maßnahmen kann eine Verletzung der at-most-once Semantik vorliegen
- Lösung
  - Eindeutige Identifizierung von Anfragen durch `REQUEST Service Context`
    - `Client Id`, identifiziert den Client
    - `Retention Id`, identifiziert den Aufruf
    - `Expiration Time`, legt fest wie lange Aufrufergebnisse aufbewahrt werden
  - Server ORB erkennt so die Wiederholungen von Anfragen und sendet das bereits ermittelte Ergebnis

Verhalten bei Ausfällen

- Problem
  - Es kommt zu einem Serverausfall oder Verbindungsproblem während eines Aufrufs
  - Die TCP/IP Verbindung wird nicht ordnungsgemäß beendet und der Client bekommt den Abbruch deshalb nicht mit
- Lösung
  - Periodische Testaufrufe (heartbeat messages)
  - Client fragt diese via Policy pro Verbindung an und spezifiziert
    - Aufruffrequenz
    - Timeout
  - Client-ORB ruft `_FT_HB()` beim Server-ORB auf
  - Operation wird vom Skeleton bereitgestellt
  - Server-ORB muss dies explizit erlauben
    - Kontrolle von erzeugtem Netzwerkverkehr

Einstellung von Fehlertoleranzeigenschaften

- Konfigurierbare Eigenschaften
  - Replikation
  - Mitgliedschaft
  - Konsistenz
  - Monitoring
    - Intervall und Timeout
  - Erzeugung replizierter Objekte mittels Factories
  - Weitere Konfigurationseigenschaften
    - Replikatanzahl (initiale bzw. minimale Anzahl)
    - Sicherungspunktintervall

Replikation

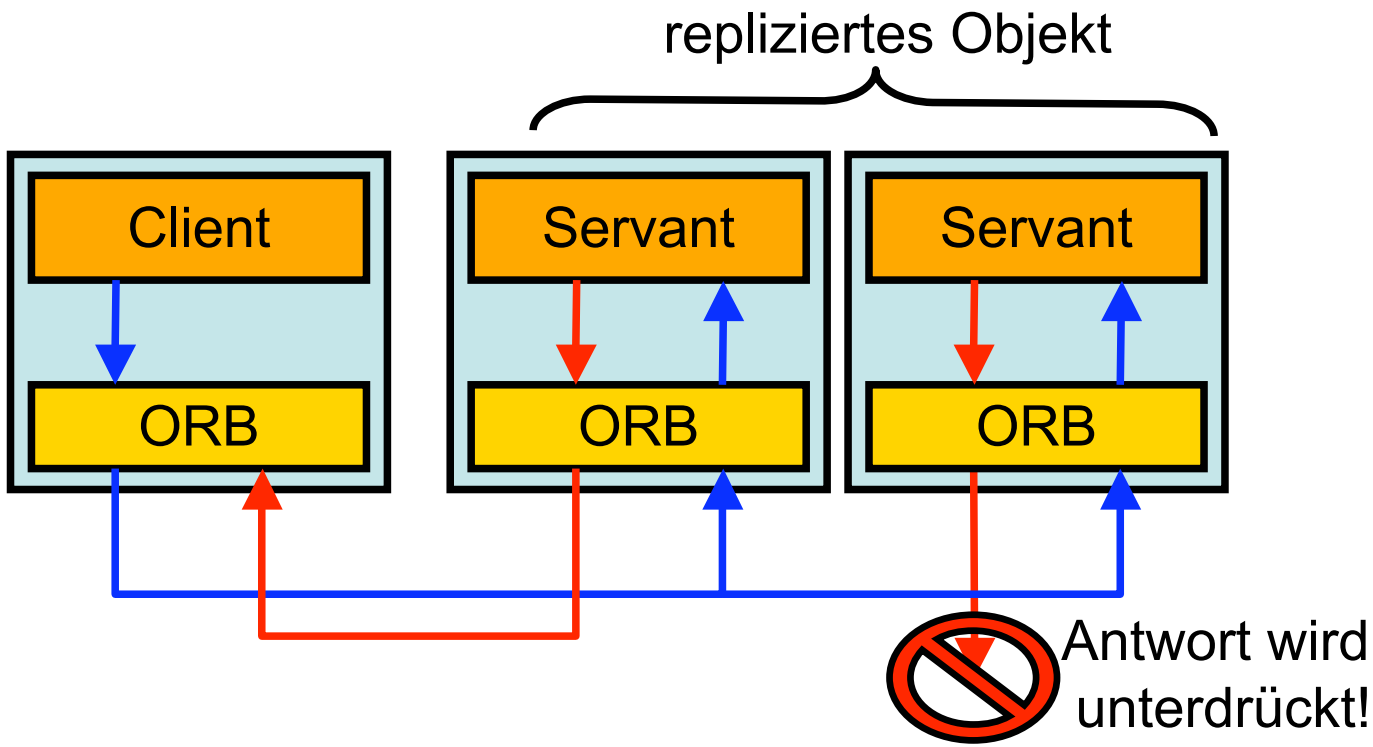
- Stateless
  - statische Daten und nur lesender Zugriff
- Cold Passive Replication
  - Wiederherstellung wird durch Sicherungspunkt und Protokollierung von Aufrufen erreicht
  - langsame Kompensation von Ausfällen
- Warm Passive Replication
  - Aktueller Zustand des primären Replikats wird periodisch an alle anderen Replikate übertragen
  - Protokollierung von Aufrufen
  - schnelleres Recovery im Vergleich zu cold passive
- Active Replication
  - alle Replikate bearbeiten Aufrufe
  - sehr geringe Verzögerung bei Ausfällen

Active Replication

- Normalbetrieb
  - Wenn replizierte Objekte von anderen replizierten Objekten aufgerufen werden, müssen duplizierte Aufrufe und Antworten unterdrückt werden
- Behandlung von Ausfällen
  - Infrastruktur verschattet Ausfälle transparent da mindestens ein Replikat auf Anfragen antwortet
- Behandlung von Beitritten
  - Zustandstransfer zur Integration eines neuen oder temporär ausgefallenen Replikats nötig

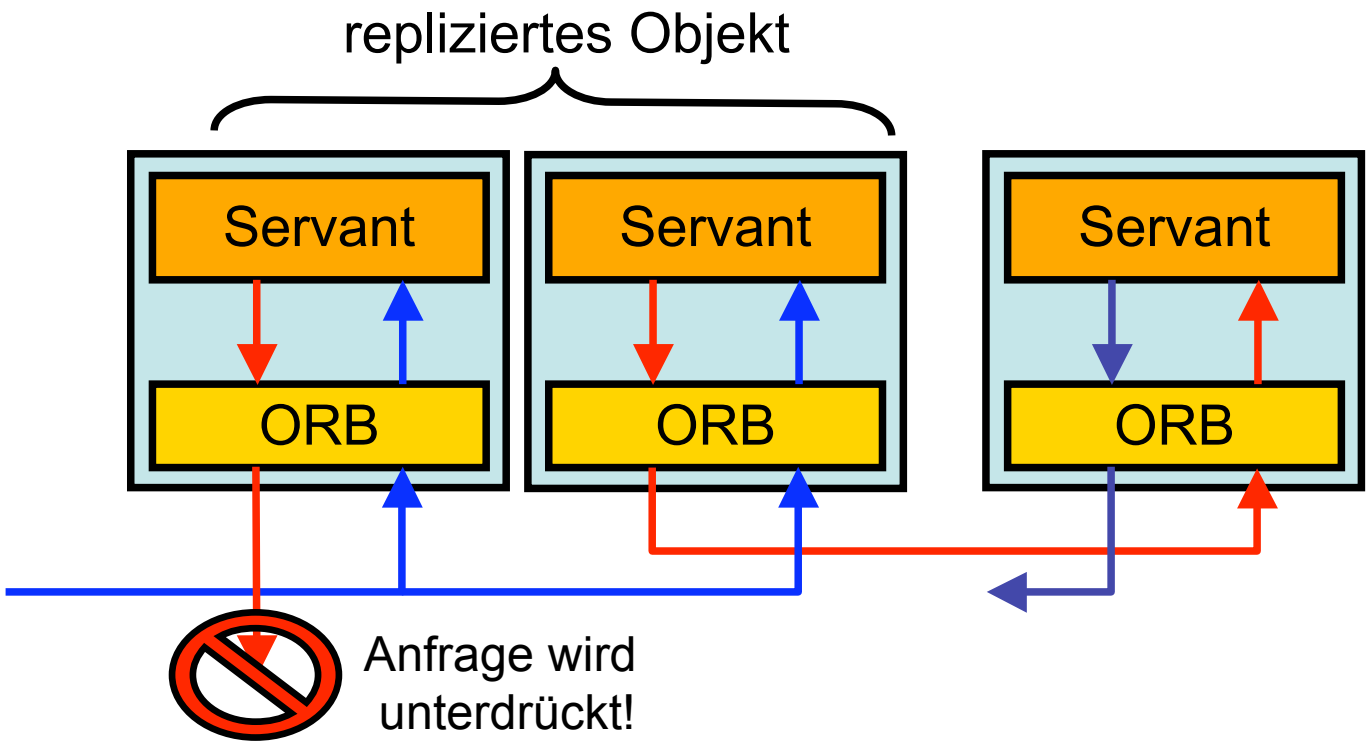
Active Replication

- Unterdrückung von duplizierten Ergebnissen
  - Kann im Prinzip auf Seite des Aufrufers oder der Replikate erfolgen



Active Replication

- Unterdrückung von duplizierten Aufrufen
  - Nötig wenn ein repliziertes Objekt andere Objekte aufruft

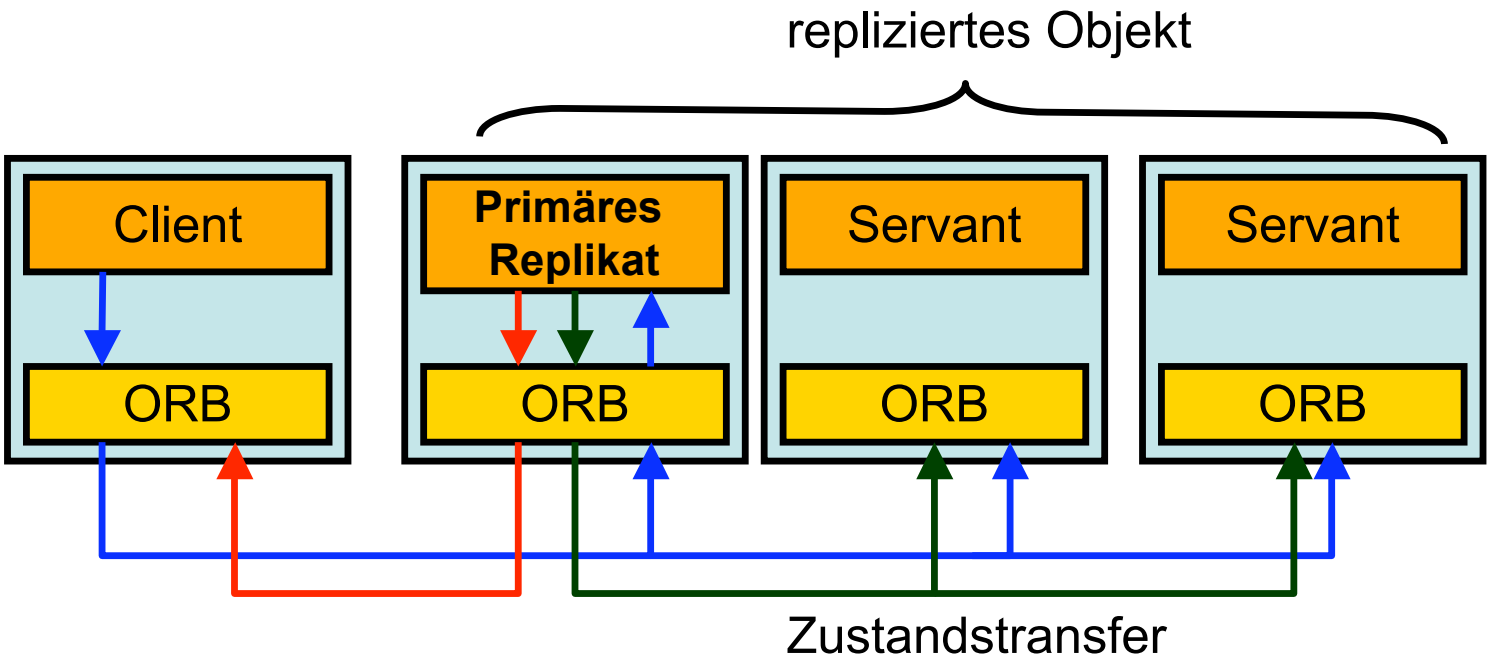


Passive Replication

- Normalbetrieb
  - periodischer Zustandstransfer zu allen Replikaten (nur bei warm passive nötig)
  - Logging von Aufrufen und Sicherungspunkten
- Behandlung von Ausfällen
  - Ausfall des primären Replikats erfordern die Wahl eines neuen Replikats
  - Initialisierung des neuen primären Replikats (Zustandstransfer bei cold passive Replication)
  - Erkennung von duplizieren Anfragen
- Behandlung von Beitritten
  - Zustandstransfer zu neuem oder wiederhergestelltem Replikat bei warm passive Replication

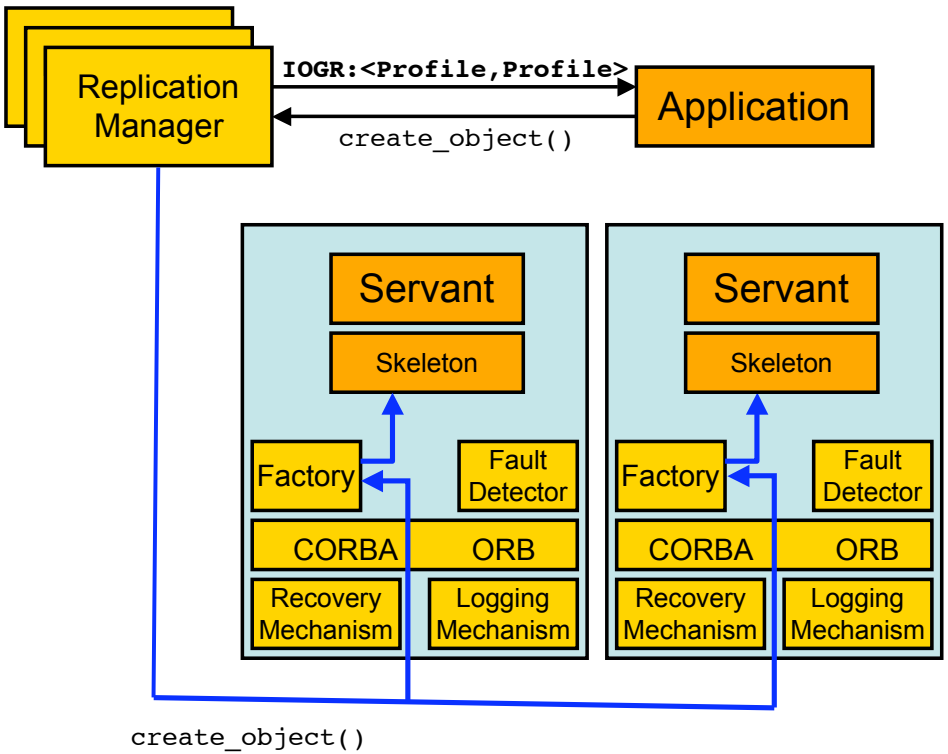
Warm Passive Replication

- Protokollierung von Anfragen
  (durch passive Replikate bzw. die Middleware)
- Periodischer Zustandstransfer



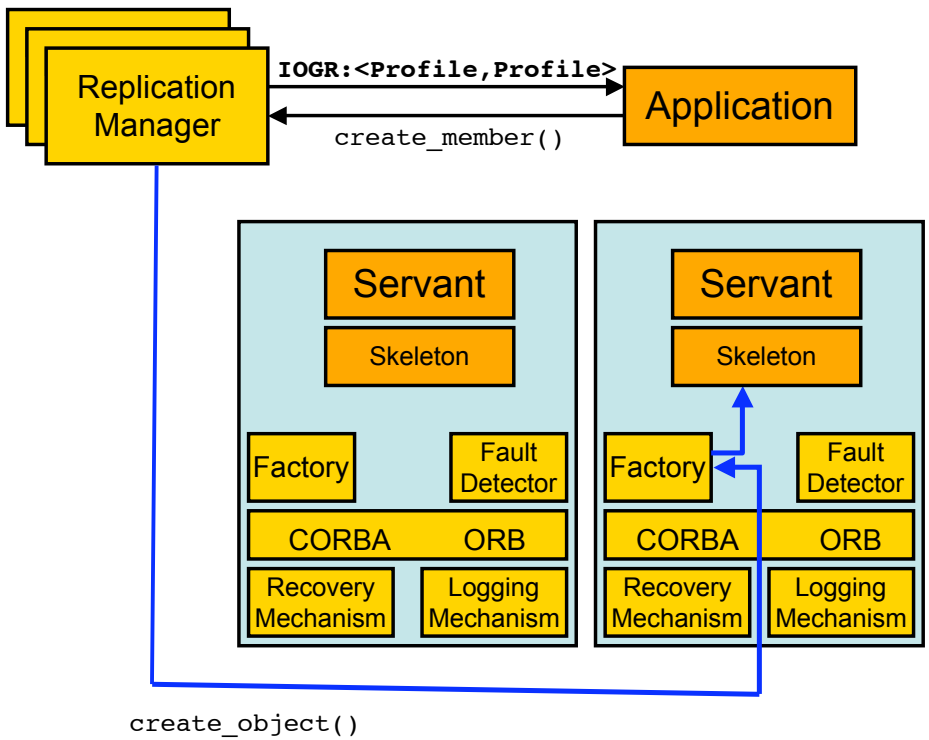
Mitgliedschaft

- Infrastruktur kontrolliert Replikaterzeugung
  - Infrastruktur erzeugt Replikate und platziert sie in der Domäne
  - Anwendung ruft hierzu die `create_object()`-Methode des Replication Managers auf



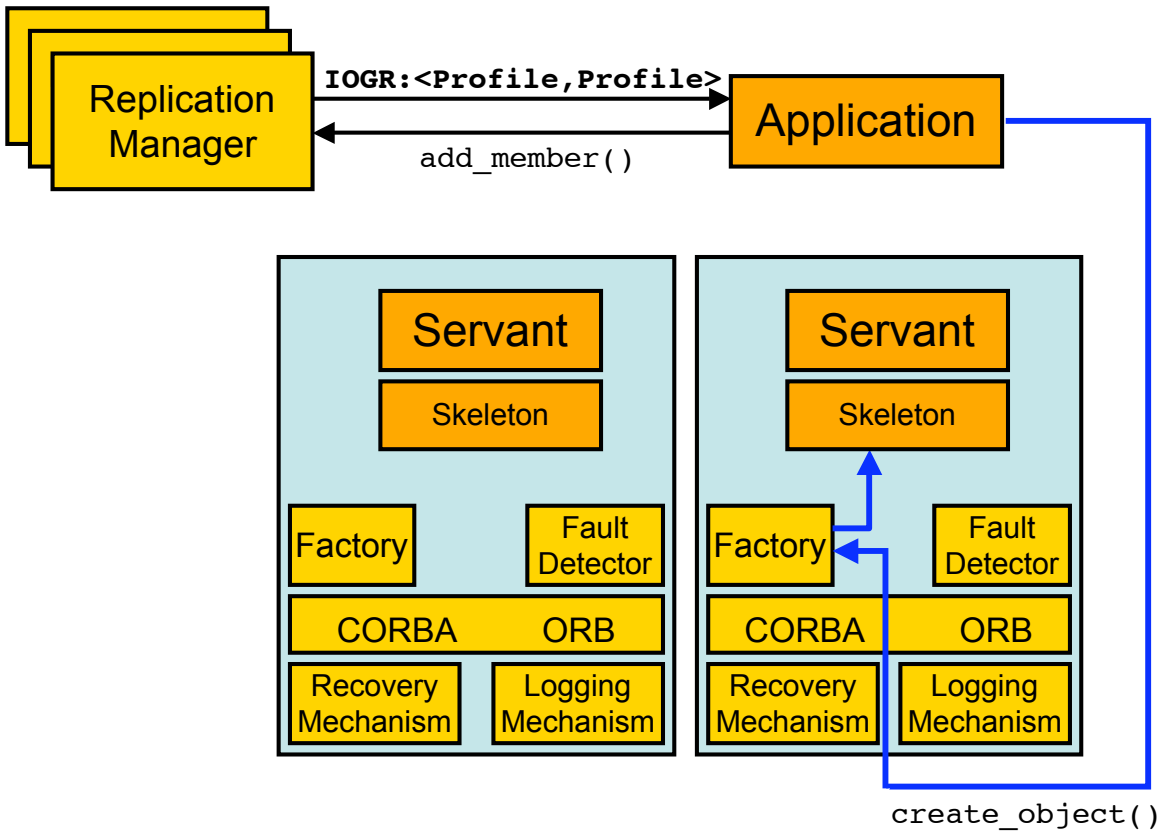
Mitgliedschaft

- Anwendung kontrolliert Replikaterzeugung
  - Anwendung erzeugt und platziert Replikate entsprechend ihrer Erfordernisse
  - Anwendung beauftragt den Replication Manager zur Erzeugung einzelner Replikate



Mitgliedschaft

- Anwendung kontrolliert Replikaterzeugung
  - Anwendung erzeugt Replikate eigenständig und nutzt den Replication Manager zur Verwaltung



Konsistenz

- Infrastruktur kontrolliert Konsistenz durch Mechanismen wie Logging, Zustandssicherung und Recovery
  - Active Replication
    - Nach jedem Aufruf haben alle Replikate den gleichen Zustand
    - Erfordert eine Gruppenkommunikation und total geordnete Zustellung von Nachrichten
  - Passive Replication
    - Nach jedem Zustandstransfer haben alle Replikate den gleichen Zustand
- Anwendung kontrolliert Konsistenz
  - Anwendung stellte alle Mechanismen zur Konsistenzerhaltung bereit

Monitoring

- Granularität der Ausfallüberwachung
  - **Mitglieder**
    - Es wird jedes Mitglied einer Objektgruppe beobachtet
  - **Ort**
    - Es wird ein Objekt als Repräsentant für den Ort beobachtet
    - Wird das beobachtete Replikat beendet wird ein anderes Objekt ausgewählt
    - Fällt das Objekt aus wird der Rechner als ausgefallen betrachtet
  - **Ort und Typ**
    - Es wird ein Objekt pro Ort und Typ kontrolliert
    - Wird das beobachtete Replikat beendet wird ein anderes Objekt vom gleichen Typ ausgewählt
    - Fällt das Objekt aus werden alle Objekte des Typs an diesem Ort als ausgefallen betrachtet

Factories

- Erzeugung von replizierten Objekten wird durch eine Sequenz von `FactoryInfo` Strukturen konfiguriert
- Aufbau der `FactoryInfo` Struktur
  - Referenz auf Fabrik
  - Ort der Fabrik
  - Information zur Konfiguration bzw. Eigenschaften der Fabrik (`Criteria`)

Verwaltung von Replikaten

- Aufgaben des Replication Manager
  - Verwaltung von Objektgruppen und ihren Eigenschaften
    - Replikationsart
    - Mitgliedschaft
    - Konsistenz
    - usw.
- Technische Realisierung
  - Methoden zur Benachrichtigung über Ausfälle
    - `register_fault_notifier()`
    - `get_fault_notifier()`
  - Erbt von
    - `Property Manager` – Management von Fehlertoleranzeigenschaften
    - `Object Group Manager` – Verwaltung von Objektgruppen
    - `Generic Factory` – Erzeugung von replizierten Objekten

Property Manager Schnittstelle

- Ermöglicht Konfiguration von Eigenschaften
  - für alle Objektgruppen
  - für alle replizierten Objekte eines bestimmten Typs
  - für ein bestimmtes repliziertes Objekt zum Zeitpunkt der Erzeugung
  - für ein bestimmtes repliziertes Objekt zu Laufzeit

Spezifischere Definitionen haben höhere Priorität

Zeitpunkt der Konfiguration

Eigenschaften	Standard	Typ	Erzeugung	Dynamisch
Replikation	X	X	X	
Mitgliedschaft	X	X	X	
Konsistenz	X	X		
Monitoring	X	X		
Granularität	X	X	X	X
Fabriken		X	X	X
Initiale Replikatanzahl	X	X	X	
Minimale Replikatanzahl	X	X	X	X

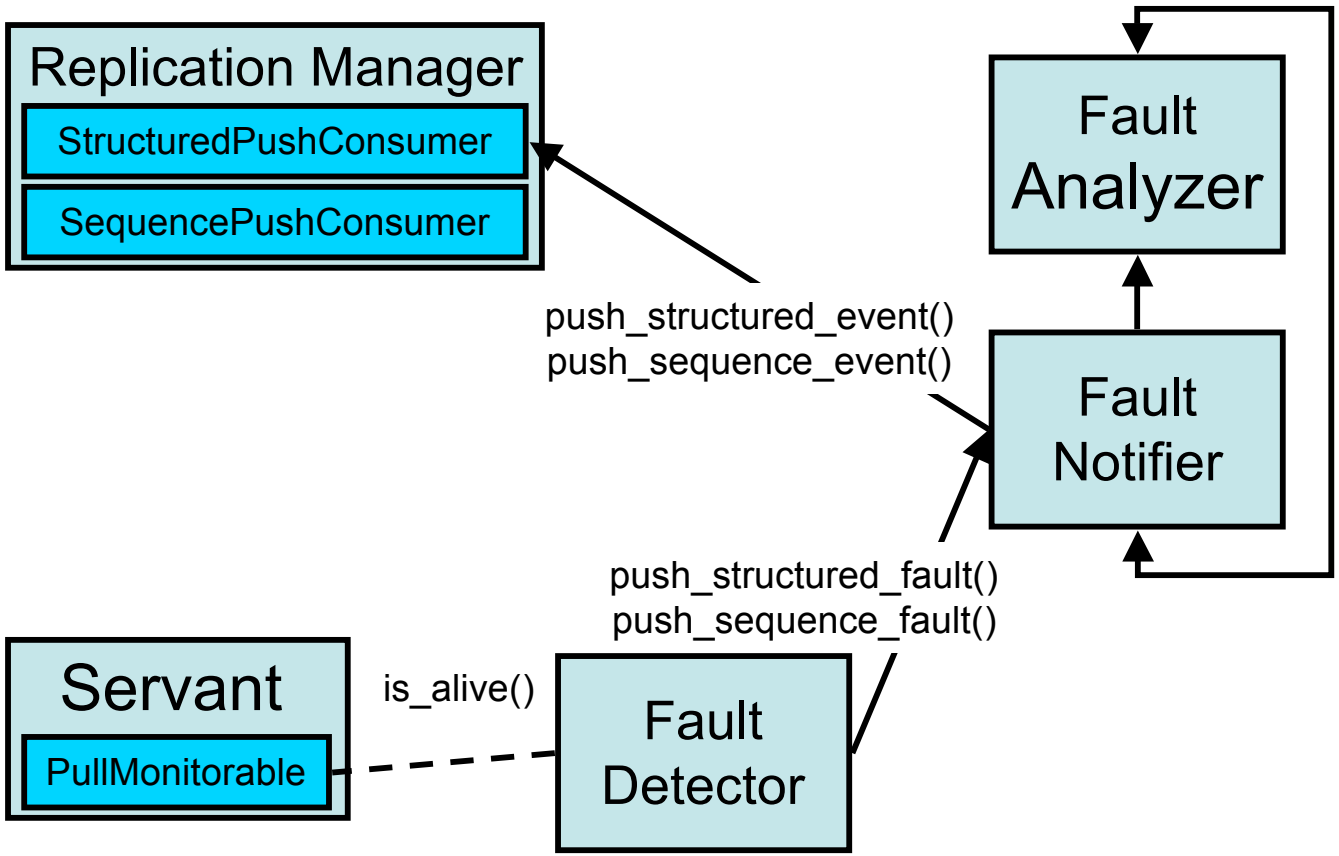
Object Group Manager

- Operationen der `Object Group Manager` Schnittstelle:
  - `create_member()`
  - `add_member()`
  - `set_primary_member()`
  - `remove_member()`
  - `locations_of_members()`
  - `get_object_group_ref()`
  - `get_object_group_id()`
  - `get_member_ref()`

Fehlermanagement

- Fault Detector
  - Erkennt Fehler und erzeugt Fehlerberichte
- Fault Notifier
  - Sammelt und verarbeitet Fehlerberichte von Fault Detectors und Fault Analyzers
  - Bildet eine Art Datenbank für alle Fehlerberichte
- Fault Analyzer
  - Spezifisch für jede Anwendung
  - Verarbeitet Fehlerberichte und fasst abhängige Fehler zusammen

Fehlermanagement



Fehlermanagement

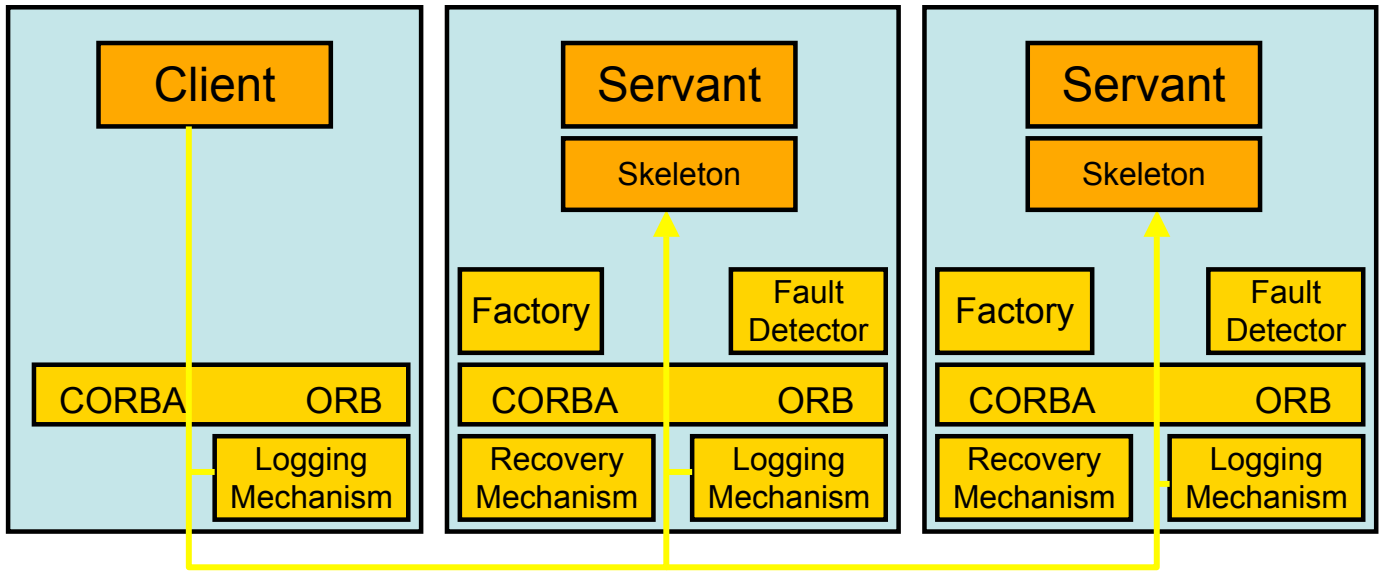
- Fehlerweitergabe durch Fault Detectors
  - einzelne Fehlerreporte (`CosNotification::StructuredEvent`)
  - Sammelreport (`CosNotification::EventBatch`)
  - Fehlertyp: (`ObjectCrashFault`)
    - `Domain_name - FT_CORBA`
    - `Type_name - ObjectCrashFault`
    - `Location - host/process`
    - `TypeId - IDL:Library:1.0`
    - `ObjectGroupId - 4711`
  - Sind alle Objekte eines Ortes ausgefallen wird `TypeId` und `ObjektGroupId` nicht im Fehlerreport vermerkt
  - Sind alle Objekte eines Types ausgefallen wird die `ObjektGroupId` weggelassen

Fehlermanagement

- Fehlerverarbeitung
  - Fehlerberichte werden durch Fault Detectors erzeugt und via *push* an den Fault Notifier weitergeleitet
  - Beliebige Konsumenten registrieren sich beim Fault Notifier und werden über Fehler benachrichtigt
    - Filter reduzieren das Nachrichtenaufkommen

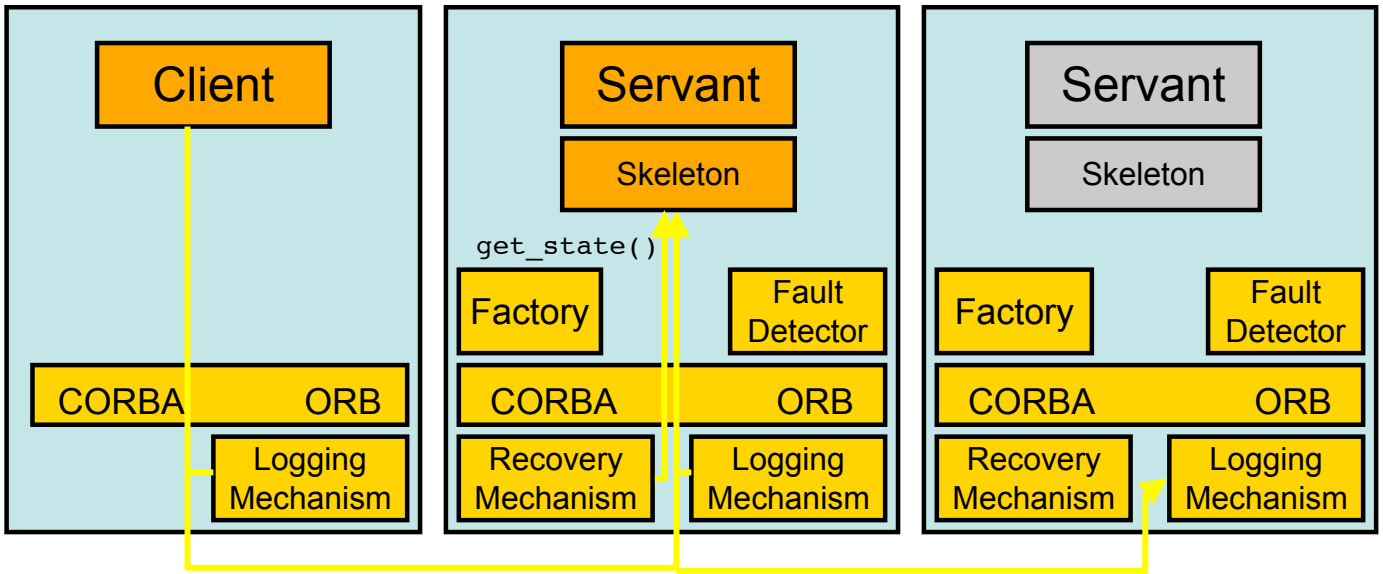
Logging und Recovery

Beispiel: Active Replication



Logging und Recovery

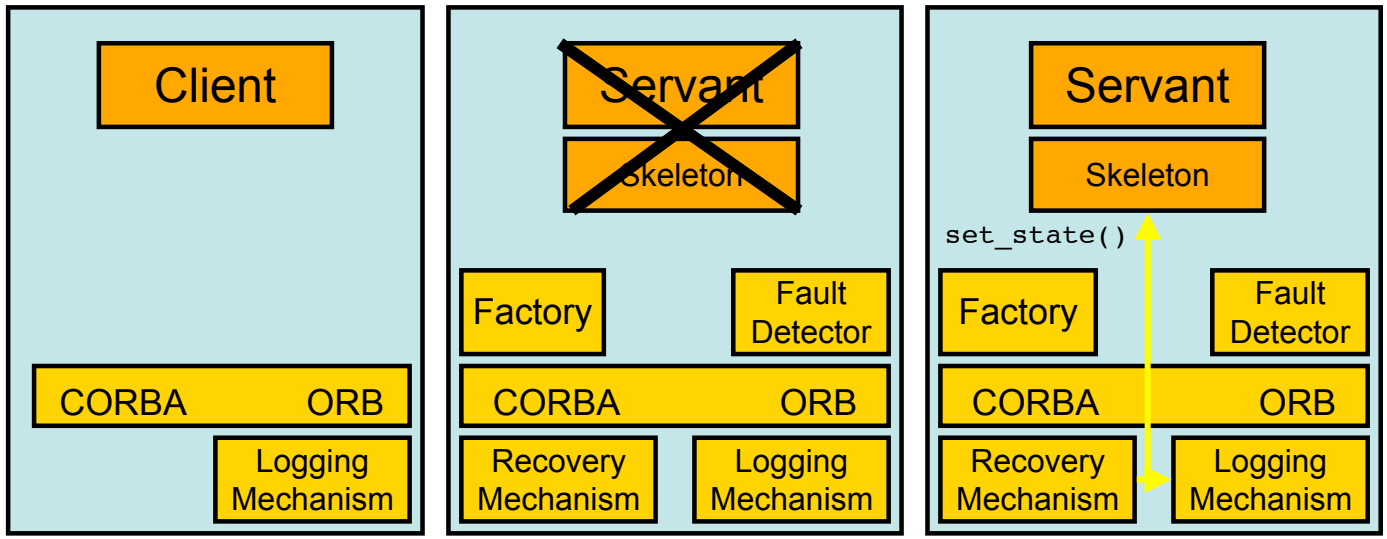
Beispiel: Cold Passive Replication — Normalbetrieb



- Anfragen werden protokolliert durch die Middleware
- Zyklische Erzeugung eines Sicherungspunktes (`get_state()`) der an die Standorte von passiven Replikaten vermittelt wird

Logging und Recovery

Beispiel: Cold Passive Replication — Wiederherstellung



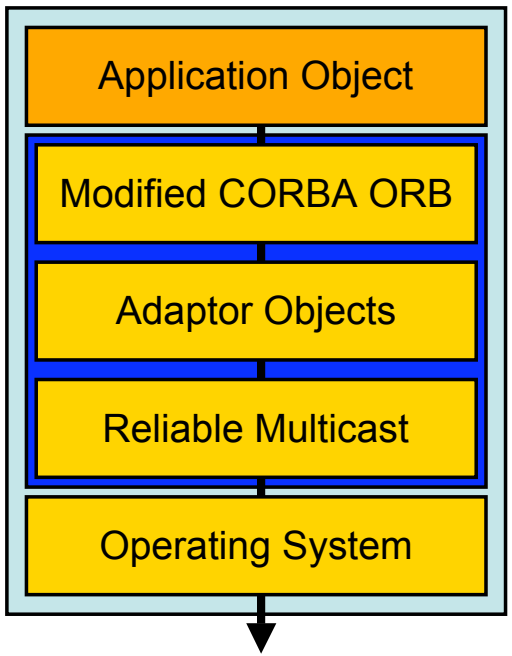
- Bei einem Ausfall wird zunächst der letzte Sicherungspunkt eingespielt und danach bereits bearbeitete aber im Sicherungspunkt noch nicht erfasste Anfragen nochmals ausgeführt
  - Nach Abschluss dieses Vorgangs können neuen Anfragen bearbeitet werden

Implementierungsvarianten

- Integrativer Ansatz
  - Fehlertoleranz wird direkt durch den ORB realisiert
- Dienst-Ansatz (entstanden vor FT-CORBA)
  - Unterstützung für replizierte Dienste wird als CORBA Service angeboten
- Interceptor-Ansatz
  - Aufrufe werden durch Interceptoren (unterhalb des ORBs) abgefangen und für die Bereitstellung von replizieren Diensten modifiziert

Integrativer Ansatz

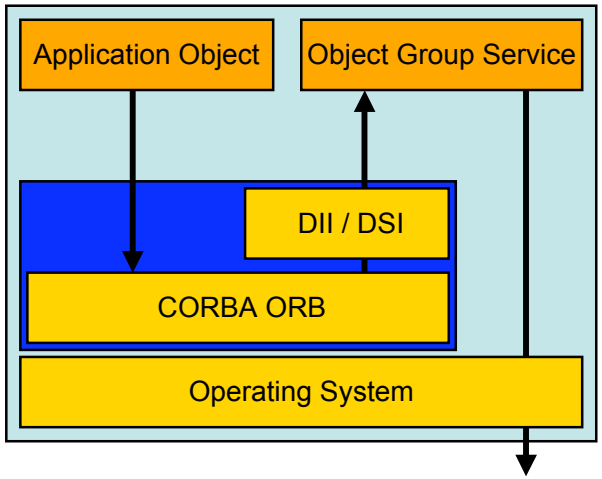
- ORB wird durch eine Gruppenkommunikation ergänzt
- IIOP wird durch ein proprietäres Protokoll ausgetauscht



- Nachteil: Aufwendige Implementierung
- Vorteil: Effizient und transparent für die Applikation und kann FT-CORBA-kompatibel sein

Dienst-Ansatz

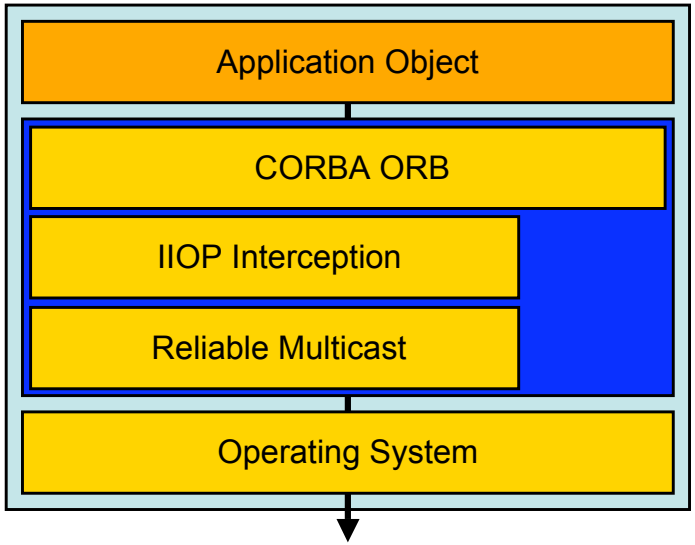
- Anwendung kommuniziert über lokale Dienste mit dem replizierten Objekt



- Nachteile:
  - Indirektion über den ORB
  - Nicht transparent für die Anwendung und nicht FT-CORBA-kompatibel
- Vorteil: Keine Veränderung des ORBs nötig und damit portabel

Interceptor-Ansatz

- Interceptoren fangen Aufrufe ab und vermitteln sie weiter an das replizierte Objekt



- Nachteil: Interceptoren sind oft spezifisch für ein Betriebssystem
- Vorteil: Transparent für die Anwendung

Zusatzanforderungen bei der Anwendungsentwicklung

- Zustandstransfer
  - Objekt muss die `Checkpointable`-Schnittstelle implementieren
  - Erfordert den gesamten Objektzustand als Bytearray bereitzustellen
    - Aufgabe des Entwicklers!
  - Sicherungspunkt kann erstellt werden wenn sich das Objekt in einem konsistenten Zustand befindet
    - alle laufenden Aufrufe werden beendet
    - neue Aufrufe werden blockiert
- Determinismus
  - Keine Aufrufe zu externen Informationsquellen
  - In der Regel Verlust von mehrfädiger Ausführung

Zusammenfassung

- FT-CORBA bildet einen Standard zur Entwicklung fehlertoleranter Infrastrukturen und Anwendungen
- Durch Middleware-Mechanismen und starke Konsistenz sind replizierte Objekte weitgehend transparent für Anwendungen
- Objektimplementierungen müssen angepasst werden
  - Determinismus
  - Zustandstransfer
- Kritikpunkte
  - Einheit der Replikation: Objekt vs. Prozess
  - Nichtdeterminismus ist Sache des Anwendungsentwicklers
  - Zustand: Objekt + Infrastruktur
    - Zustand der Infrastruktur ist schwer zu ermitteln
  - Konfiguration ist kompliziert
  - Stark konsistente Replikation erfordert aufwendige Gruppenkommunikation

Literatur

[OMG, 2004] Object Management Group
CORBA/IIOP Specification (Chapter 23, Fault Tolerant CORBA)
OMG Technical Committee Document formal/04-03-21, 2004.

[Felber, 2004] Pascal Felber, Priya Narasimhan
Experiences, Strategies, and Challenges in Building
Fault-Tolerant CORBA Systems
IEEE Transactions on Computers, vol. 53, no. 5, pp. 497-511, 2004

[Baldoni, 2002] R. Baldoni, C. Marchetti, R. Panella, L. Verde
Handling FT-CORBA Compliant Interoperable Object Group References
WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 2002

[Narasimhan, 2007] Priya Narasimhan
Fault-Tolerant CORBA: From Specification to Reality
Computer , vol. 40, no. 1, pp.110-112, Jan. 2007

Überblick

Zeit – Überblick

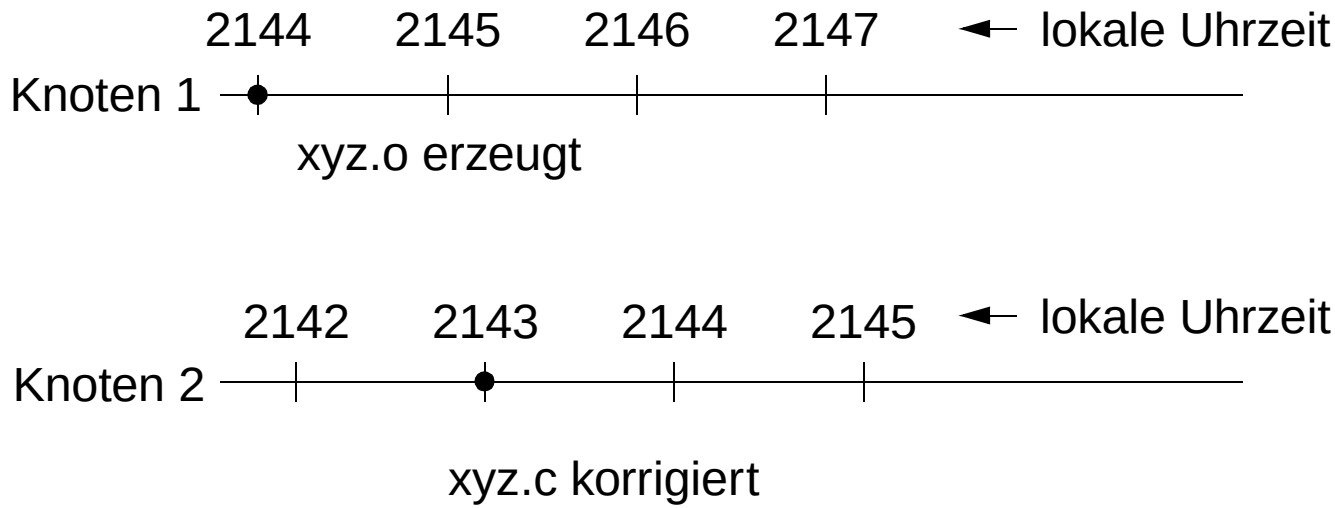
- Uhrensynchronisation
- Zeit im Verteilten System
- Logische Uhren
- Synchronisation von physikalischen Uhren
  - Konvergenz-Algorithmus - CNV
  - Network Time Protocol - NTP

Uhrensynchronisation

- Notwendigkeit von Uhrensynchronisation
  - Zeit als Mittel der Reihenfolgebestimmung
- Probleme der Uhrensynchronisation
- Logische Uhren
  - Lamport-Uhren
  - Vektoruhren
- Synchronisation von physikalischen Uhren
  - Grundlagen
  - Konvergenzalgorithmus CNV
  - Network Time Protocol NTP

Zeit als Mittel der Reihenfolgebestimmung

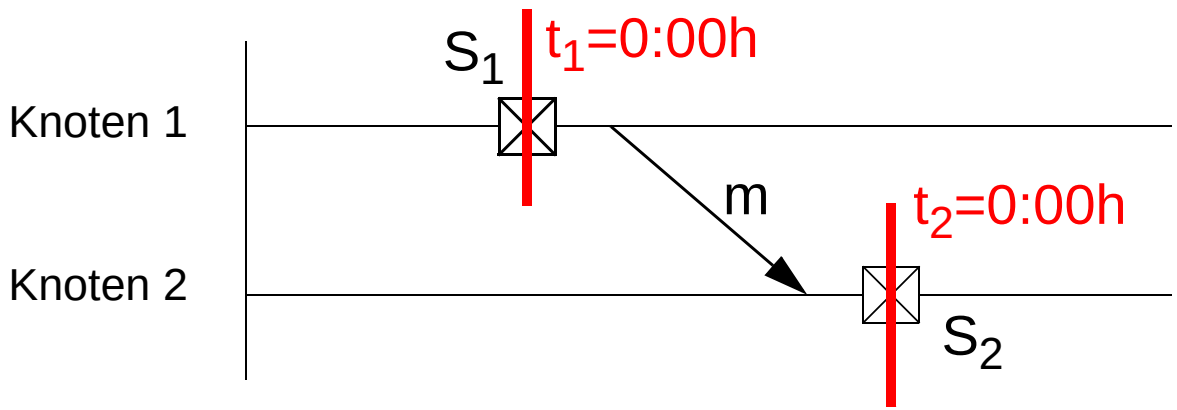
- Beispiel: `make`



⇒ Modifikation von `xyz.c` durch Knoten 2 wird nicht erkannt
⇒ Die Datei wird nicht neu übersetzt!

Zeit als Mittel der Reihenfolgebestimmung

- Beispiel: Verteilte Zustandssicherung
  - Gegeben: Verteiltes System aus interagierenden Knoten
  - Für eine konsistente Zustandsicherung soll bei allen Knoten zu festgelegter Uhrzeit der Zustand gesichert werden
  - Problem: Inkonsistente Sicherungspunkte möglich, falls die lokalen Uhren voneinander abweichen:



$\Rightarrow$ Auswirkung der Nachricht `m` in S_2 enthalten, aber nicht in S_1!

Probleme der Uhrensynchronisation

- Es gibt keine völlig identischen physikalische Uhren
  - Abweichende Initialisierung (konstantes Offset)
  - Abweichende Ganggeschwindigkeit (Frequenzfehler)
  - Umgebungseinflüsse (z.B. Bauteilalterung, **Temperaturabhängigkeit**)

  $\Rightarrow$ Ohne Synchronisierung kann Fehler immer größer werden!

- Gemeinsame Uhr für alle Knoten eines verteilten Systems (meist) nicht realisierbar

- „Zentrale Referenzuhren“ (Funk, z.B. Langwellensender DCF77 in Mainflingen, amerikanischer Kurzwellensender WWV in Fort Collins, GPS) nur mit technisch beschränkter Genauigkeit

Logische Uhren

- Grundidee (Logische Uhren nach Lamport):

  Aussagen zur Reihenfolge unterschiedlicher Ereignisse

  - Nur benötigt, wenn eine Interaktion zwischen einzelnen Komponenten des verteilten Systems erfolgt ist!
  - Eine Aktion b, die logisch durch die Interaktion bedingt „nach“ einer Aktion a stattfindet, soll stets den größeren Zeitstempel tragen

  $\Rightarrow$ Synchronisation bei Kommunikation

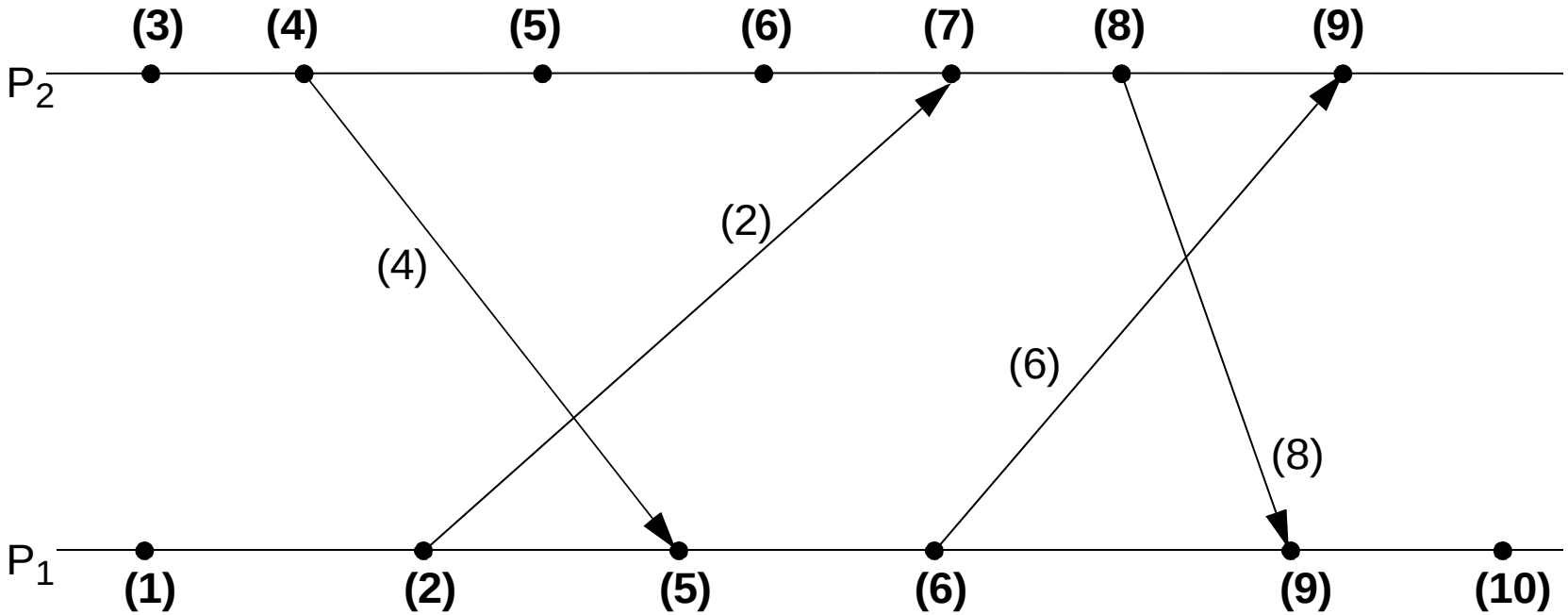
- Anmerkung: Die im Folgenden beschriebenen Verfahren setzen voraus, dass die Kommunikation nur durch Nachrichtenaustausch über ein Kommunikationsnetz erfolgt

Logische Uhren

- Beschreibung des Algorithmus: Jeder Prozess i verfügt über einen Zähler C_i *(„Uhr“)*, der auf folgende Weise zählt:
  - Bei Ausführung einer lokalen Aktion und bei Ausführung einer Sende-Aktion durch Prozess i wird seine Uhr C_i um 1 erhöht; die Aktion bekommt den Wert nach dem Erhöhen als Zeitstempel
  - Jede Nachricht m trägt als Zeitstempel t_m den Zeitstempel der Sendeaktion
  - Bei Empfang einer Nachricht durch Prozess i bei lokalem Uhrenstand C_i und empfangenem Zeitstempel t_m wird die lokale Uhr auf $max(C_i, t_m) + 1$ gestellt und dieser Wert als Zeitstempel des Empfangsereignis verwendet

Logische Uhren

- Beispiel für zwei Prozesse P_1 und P_2



Logische Uhren

- Eigenschaften
  - Potentielle kausale Abhängigkeit eines Ereignisse E_2 von einem Ereignis E_1 [Kurznotation: $E_1 \rightarrow E_2$]:

    $$E_1 \rightarrow E_2 \Rightarrow t(E_1) < t(E_2)$$

  - Die Umkehrung

    $$t(E_1) < t(E_2) \Rightarrow E_1 \rightarrow E2$$

    gilt nicht!

  - Die Zeitstempel erzeugen eine **partielle** Ordnung auf der Menge aller Ereignisse. (Es gibt Ereignisse die „gleichzeitig“ auftreten und somit nicht geordnet werden können.)
- Ergänzung zu vollständiger Ordnung
  - Jeder Prozess erhält eindeutige Identifikation
  - Zeitstempel bestehend aus Prozess i und lokaler Zeit C_i.
  - Anordnung:

    $$(C_i, i) < (C_k, k) \Leftrightarrow C_i < C_k \text{ oder } (C_i = C_k \text{ und } i < k)$$

- Jeder Knoten besitzt nun eine lokale Uhr, die aus einem Vektor der Länge N (= Anzahl der vorhandenen Knoten) besteht

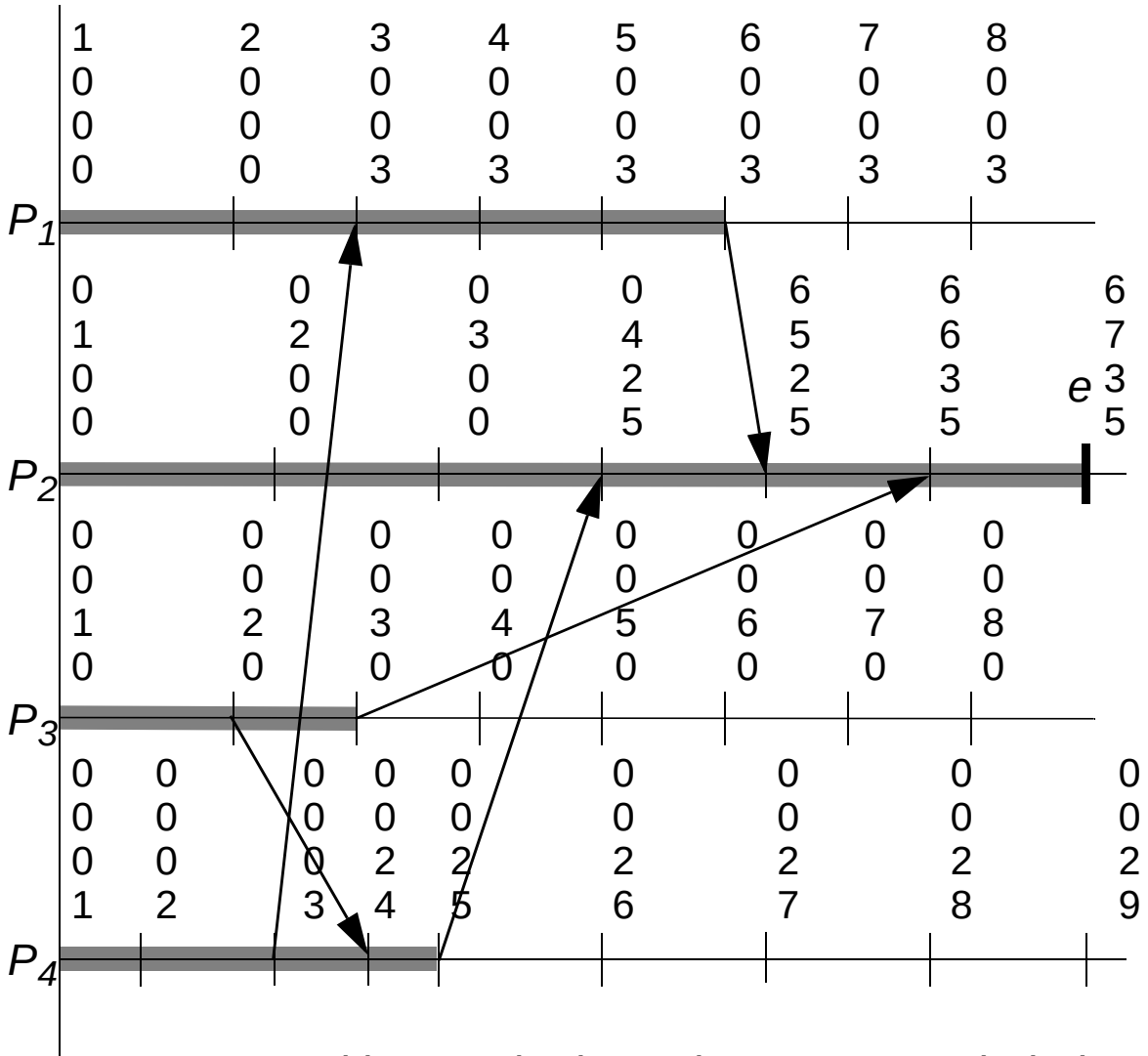
- Implementierung:

1. Initialisierung jedes Zeitvektors mit dem Nullvektor.
2. Bei jedem lokalen Ereignis eines Prozesses P_i wird dessen Komponente in seinem Zeitvektor inkrementiert: $C_i[i]++$
3. Ein Prozess P_i, der eine Nachricht empfängt, inkrementiert seine Komponente im Zeitvektor und kombiniert diesen danach komponentenweise mit dem empfangenen Zeitvektor t:

$$C_i[i]++$$

$$\text{für } k = 1 \ldots N: \quad C_i[k] := max(C_i[k]; t[k])$$

Logische Uhren



Logische Uhren

- Eigenschaften
  - Erzeugt eine kausale Ordnung: $t(E_1) < t(E_2) \Leftrightarrow E_1 \rightarrow E_2$

    Definition von $<$ mit $t(E_1) := (a_1, ..., a_n)$ und $t(E_2) := (b_1, ..., b_n)$:

    $$t(E_1) < t(E_2) \Leftrightarrow (\forall i : a_i \leq b_i) \wedge (\exists i : a_i < b_i)$$

- Vorteil
  - Exakte Aussage zum kausalen Zusammenhang von Ereignissen mit Hilfe des Zeitstempels möglich

- Nachteil
  - Hoher Kommunikationsaufwand (Vektor aus N Elementen in jeder Nachricht; skaliert schlecht für großes N!)

Synchronisation von physikalischen Uhren

- Physikalische Zeit basierend auf Atom-Sekunde: *TAI*
  - „Die Sekunde ist das 9.192.631.770-fache der Periodendauer der dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustandes von Atomen des Nuklids ^{133}Cs entsprechenden Strahlung“

    [Gültige Definition seit 1967, davor über Erdrotation (bis 1956) bzw. über Erdumlaufzeit um die Sonne festgelegt.]
  - Genauigkeit von Cs-Uhren: bis ca. 10^{-14} (entspricht ca. $0.8ns$/Tag!)

- Verbreitung der amtlichen Zeit: Normalfrequenz/Zeitzeichensender
  - Langwelle: z.B. DCF77 (Mainflingen, D); WWVB (Boulder, US); maximale Genauigkeit bis ca. $50\mu s$
  - Kurzwelle: z.B. WWV (Colorado), WWVH (Hawaii); maximale Genauigkeit bis ca. $1ms$
  - GPS-Satelliten: maximale Genauigkeit ca. $0.3\mu s$

Synchronisation von physikalischen Uhren

- Softwarebasierte Synchronisation
  - Master-/Slave mit Verteilung einer Referenzzeit (broadcast oder poll)
  - Einigungsverfahren
- Charakterisierungsmöglichkeiten von Algorithmen
  - Monotonie (Zeitsprünge bei Korrekturen?)
  - Synchronisation mit der amtlichen Zeit
  - Robustheit gegen Netzpartitionierung
  - Fehlertoleranz gegen falsch gehende Referenzuhren
  - Referenztreue
  - Erzeugte Netzlast
  - Fehleraussage

Konvergenz-Algorithmus CNV

- Aufgabe des Algorithmus
  - Vermeidung einer Akkumulation einer immer größer werdenden Abweichung zwischen mehreren Uhren
- Ablauf
  - Jeder Prozess liest die Uhr jedes anderen Prozesses
  - Er stellt seine eigene Uhr auf den Mittelwert, wobei für Uhren, die **mehr** als eine festgelegte Schranke δ von der eigenen abweichen, der Wert der **eigenen** Uhr genommen wird
  - Die gewünschten Eigenschaften (Konvergenz gegen eine gemeinsame Uhrzeit, Tolerierung von fehlerhaften Uhren) werden erfüllt, wenn weniger als ein Drittel der Uhren fehlerhaft ist

Konvergenz-Algorithmus CNV

- Annahmen
  - Benötigt $n > 3m$ Prozesse, um m fehlerhafte Prozesse zu tolerieren
  - Initial sind alle Prozesse auf in etwa die gleiche Uhrzeit synchronisiert (Fehler kleiner als δ)
  - Die Uhren aller korrekten Prozesse laufen mit in etwa der korrekten Rate
  - Ein korrekter Prozess kann die Uhr eines anderen korrekten Prozess mit (vernachlässigbar) kleinem maximalen Fehler ϵ lesen
- Ziel
  - Zu jeder Zeit sollen alle korrekten Prozesse in etwa die gleiche Uhrzeit haben (Fehler kleiner als δ)

Konvergenz-Algorithmus CNV

- Plausibilitätsbetrachtung
  - Definitionen
    - p, q seien fehlerfreie Prozesse, r irgendein Prozess
    - $C_{p,q}$ sei die Uhrzeit von q, so wie sie p bekannt ist
  - Es gilt:
    - r fehlerfrei $\Rightarrow C_{p,r} \approx C_{q,r}$
    - r fehlerhaft $\Rightarrow |C_{p,r} - C_{q,r}| < 3\delta$
  - Alle Prozesse p stellen ihre Uhr auf $\sum_i (C_{p,i})/n$

Konvergenz-Algorithmus CNV

- Plausibilitätsbetrachtung
  - Schlechtester Fall:

$$\text{(n-m)-mal } (C_{p,i} - C_{q,i}) \approx 0$$
$$\text{m-mal } (C_{p,i} - C_{q,i}) < 3\delta$$

  - Also: neue Differenz zwischen p und q ist $(3m/n) * \delta$
    Mit $n > 3m$ folgt: neue Differenz $< \delta$
  - Vernachlässigt sind dabei
    - die Zeit für die Ausführung des Algorithmus sowie
    - Fehler durch nicht gleichzeitiges Ablesen der Uhren

Das Network Time Protocol – NTP

- Ausführliche Informationen zu NTP
  - `http://www.ntp.org` (Offizielle NTP-Homepage)
  - `http://www.eecis.udel.edu/~mills` (Homepage David Mills)
- Geschichte
  - Entwickelt seit 1982 (NTPv1, RFC 1059) unter Leitung von D. Mills
  - Seit 1990 NTPv3 (teilweise immer noch in Verwendung)
  - Aktuelle Version NTPv4, seit 1994
- Eigenschaften von NTP
  - Zweck: Synchronisierung von Rechneruhren im bestehenden Internet
  - Derzeit weit über 100.000 NTP-Knoten weltweit
    - 151 aktive öffentliche Stratum 1 - Knoten (Stand Okt. 2005)
    - 208 aktive öffentliche Stratum 2 - Knoten
  - Erreichbare Genauigkeiten von ca. $10ms$ in WANs; $< 1ms$ in LANs
  - NTP-Dämon auf fast allen Rechnerplattformen verfügbar, von PCs bis Crays; Unix, Windows, OS X, eingebettete Systeme
  - Fehlertolerant

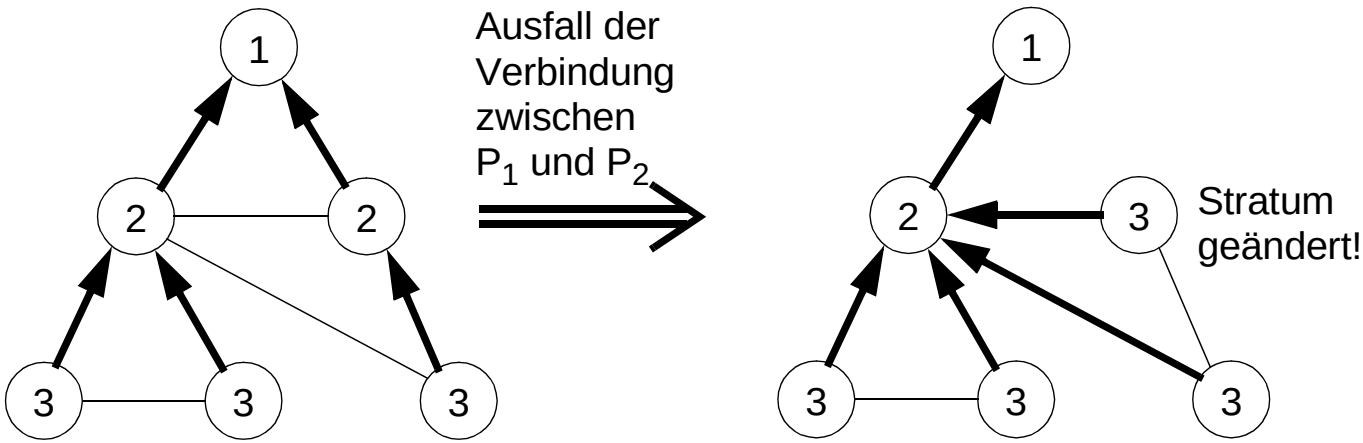
Das Network Time Protocol – NTP

- Grundlegender Überblick
  - Primäre Server (*Stratum 1*), über Funk oder Standleitungen an amtliche Zeitstandards angebunden
  - Synchronisation weiterer Knoten nach primären Servern über ein selbstorganisierendes, hierarchisches Netz
  - Verschiedene Betriebsarten (Master/Slave, symmetrische Synchronisation, Multicast, jeweils mit/ohne Authentisierung)
  - Zuverlässigkeit durch redundante Server und Netzpfade
  - Optimierte Algorithmen, um Fehler durch Jitter, wechselnde Referenzuhren und fehlerhafte Server zu reduzieren
  - Lokale Uhren werden in Zeit und Frequenz durch einen adaptiven Algorithmus geregelt

Das Network Time Protocol – NTP

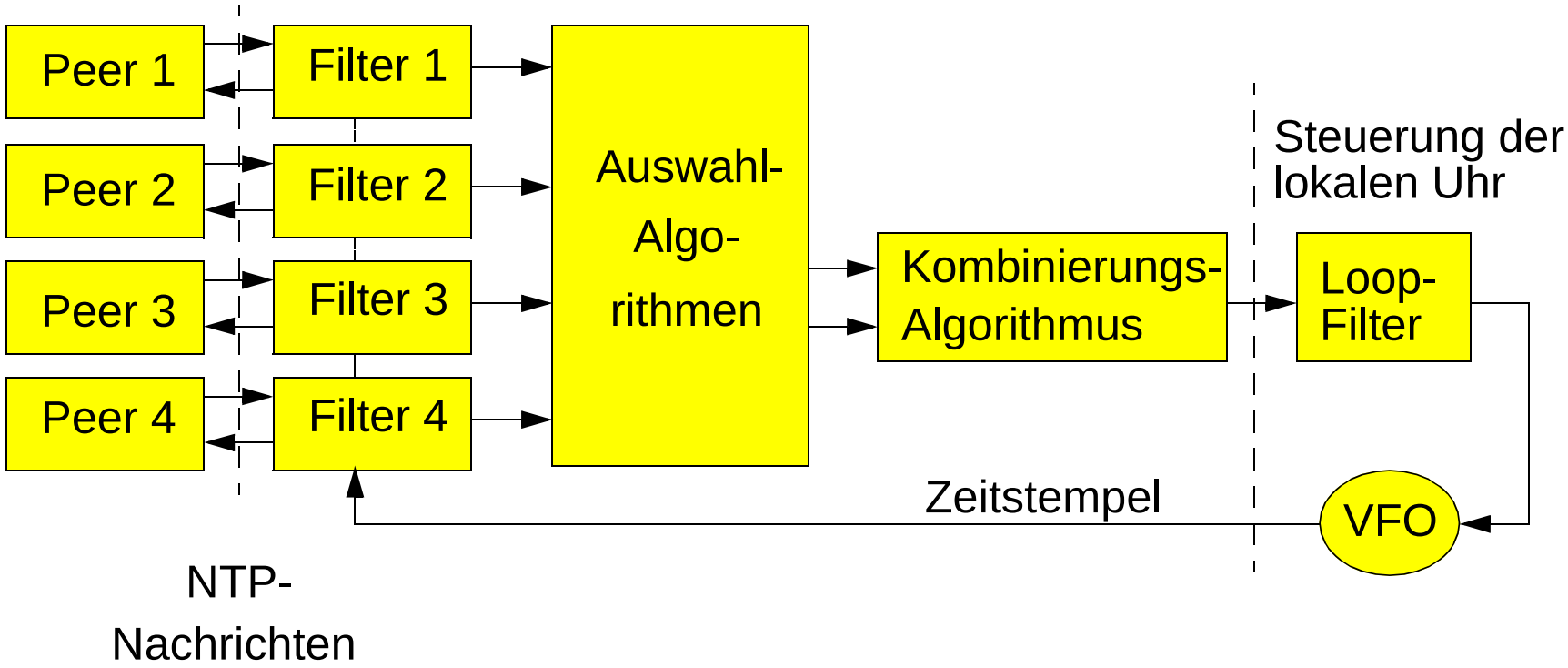
- Stratum:
  - 1: primärer Zeitgeber
  - $i > 1$: synchronisiert mit Zeitgeber des Stratums $i - 1$

- Stratum kann dynamisch wechseln:



Das Network Time Protocol – NTP

- Architektur-Überblick

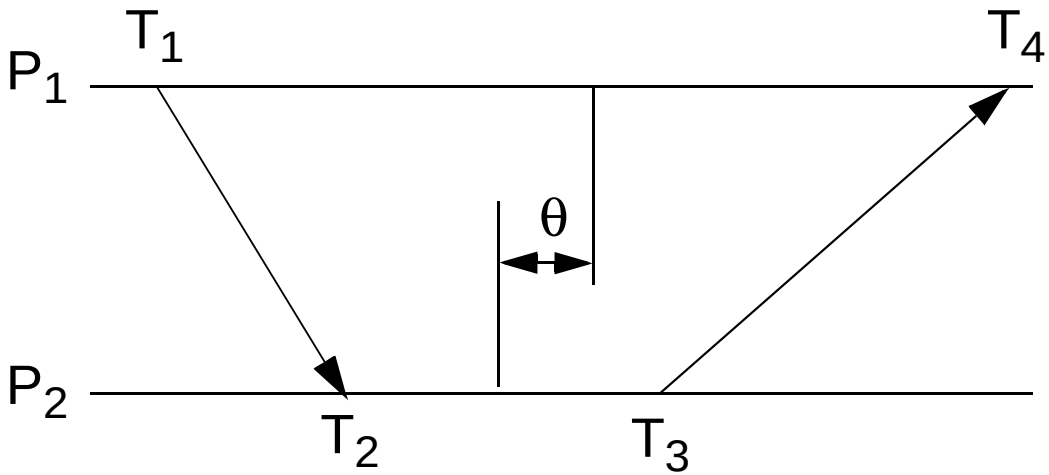


Das Network Time Protocol – NTP

- **Mehrere Referenzserver** („Peer") für Redundanz und Fehlerstreuung
- **Peer-Filter** wählen pro Referenzserver den jeweils besten Wert aus den **acht** letzten Offset-Messwerten aus
- Die **Auswahlalgorithmen** versuchen zunächst richtig gehende Uhren („*truechimers*") zu erkennen und falsche Uhren („*falsetickers*") herauszufiltern, und wählen dann möglichst genaue Referenzuhren aus
- Der **Kombinationsalgorithmus** berechnet einen gewichteten Mittelwert der Offset-Werte (frühere NTP-Versionen wählen einfach den am vertrauenswürdigsten erscheinenden Referenzknoten aus)
- **Loop Filter** und **VFO** (Variable Frequency Oscillator) bilden zusammen die geregelte lokale Uhr. Die Regelung ist so entworfen, dass Jitter und Drift bei der Regelung minimiert werden

Das Network Time Protocol – NTP

- Offset-Messung
  - Es werden die letzten 8 Messungen gespeichert; Index $i = 0$: neueste Messung



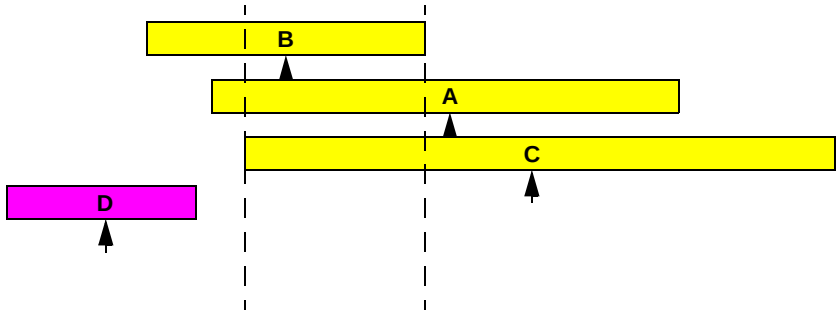
  - Reine Nachrichtenlaufzeit: $\delta = (T4 - T1) - (T3 - T2)$
  - Geschätztes Offset: $\theta = (T_2 + T_3)/2 - (T_1 + T_4)/2$
    - Exakter Wert, falls Laufzeiten in beide Richtungen gleich sind
    - Maximaler Fehler bei unsymmetrischen Laufzeiten: $\delta/2$

Das Network Time Protocol – NTP

- Peer-Filter-Algorithmus
  - Getrennte Ausführung pro Peer
  - Bei jeder Messung ermittelte Werte:
    - Offset θ, Laufzeit δ
    - Abschätzung des Messfehlers:
      $\epsilon = \mathit{Lesegenauigkeit} + \mathit{MAXDRIFT} * (T4 - T1)$
  - Eingetragene Messwerte werden in einen Puffer eingetragen
    - Speicherung der letzten 8 Messwerte
    - Aktualisierung der Fehlerabschätzung ϵ bei jeder neuen Messung um den möglichen Fehler durch Alterung (Uhrendrift)

      $i = 7...1 : (\delta_i, \theta_i, \epsilon_i) = (\delta_{i-1}, \theta_{i-1}, \epsilon_{i-1} + \mathit{MAXDRIFT} * \mathit{verstrichene\ Zeit})$
      $i = 0 : (\delta_0, \theta_0, \epsilon_0) = \mathit{neuer\ Messwert}$
  - Weitere Verarbeitung der Messwerte und Ermittlung eines *Korrektheitsintervalls* für den Peer

Das Network Time Protocol – NTP



- Auswahl-Algorithmus
  - Trennung von „truechimers” und „falsetickers”
  - DTS (Digital Time Service, Vorgänger-Algorithmus, einfacher): Ermittle Durchschnitt mit den meisten überlappenden Korrektheitsintervallen. Mittelpunkt des Intervalls wird als Offset zur Uhrenkorrektur verwendet.
  - Ziel bei NTP: Auswahl des Intervalls so, dass die Mittelpunkte der Intervalle der als korrekt betrachteten Knoten im Intervall liegen

Das Network Time Protocol – NTP

- Kombinierungsalgorithmus
  - Weiteres Aussortieren vergleichsweise schlechter Referenzen, bevorzugte Selektion von Referenzen mit kleinem Stratum
    $\rightarrow$ Zusammenstellung einer nach Qualität sortierten Liste von Referenzen (beste Referenz steht vorne)
  - Die übrig bleibenden Knoten werden zur Synchronisation verwendet
    - Einfache Variante: Falls der bisherige Referenzknoten sich in der Liste befindet, wird dieser weiterhin zur Synchronisation verwendet; ansonsten wird der Knoten am Anfang der Liste zum neuen Synchronisationsknoten
    - Komplexe Variante (NTPv4): Berechnung eines gewichteten Mittelwerts der Offsets aus allen Knoten
  - Das so ermittelte Offset wird an die Uhrenregelung (*Clock disciplin algorithm*) weitergegeben

Zusammenfassung

- Zeit in verteilten Systemen ist ein zentrale Herausforderung
- Wenn Aussagen zur kausalen Ordnung von Ereignissen benötigt werden, bietet sich der Einsatz von logischen Uhren an
  - Logische Uhren nach Lamport, erweiterbar zu einer totalen Ordnung auf alle Ereignisse, die kausale Beziehungen respektiert
  - Vektoruhren zur exakten Erfassung von kausalen Beziehungen
- Das Network Time Protocol (NTP) bietet die Möglichkeit, lokale Uhren mit für viele Zwecke ausreichender Genauigkeit an die offizielle Zeit zu synchronisieren
  - Hierarchisches Synchronisationsnetz
  - Selbstorganisierend und fehlertolerant

Überblick

Wahlalgorithmen: Überblick

Problemstellung

- In verteilten Systemen kann es viele Situationen geben, die einen ausgewählten Knoten erfordern („Anführer”)
  - Koordinator von verteilten Aktionen
  - Erzeugung von eindeutigen Token im System
  - ...
- Man möchte diesen Knoten automatisch bestimmen lassen
  - Initial bei Start des Systems
  - Zur Neukonfiguration des Systems nach Fehlern

Wahlalgorithmen: Überblick

Formale Anforderungen an einen Wahlalgorithmus

WA1 Eindeutigkeit: Es darf keine zwei Knoten P_1, P_2 im System geben, die sich beide als Anführer betrachten

WA2 Terminierung: In endlicher Zeit gibt es einen Knoten P_i, der sich als Anführer betrachtet

Wahlalgorithmen: Überblick

Varianten

- Alle Knoten erfahren vom Ende oder vom Ergebnis der Wahl
- Verschiedene Topologien: Ringe, Bäume, beliebige Netze
- Variable/unbekannte oder feste/bekannte Knotenanzahl?
- Nicht unterscheidbare Knoten oder Knoten mit eindeutigen IDs?

Erforderliche Randbedingung (für deterministische Wahl)

- Es muss möglich sein, die einzelnen Knoten voneinander zu unterscheiden (z.B. eindeutige Netzadresse)

Literatur zu Wahlalgorithmen

Wahl auf Ringen

- N. Lynch: Distributed Algorithms. Morgan Kauffmann Pub., 1996
- G. L. Lann: Distributed Systems - Towards a Formal Approach. In Information Processing 77, pp 155-160, IFIP, North-Holland, 1977
- E. Chang, R. Roberts: An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes, Comm. ACM, 22(5): 281-283, Mai 1979
- G. L. Peterson: An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. ACM Transactions on Programming Languages and Systems, pp 758-762, Oct. 1982

Wahl auf Bäumen und auf beliebigen Graphen

- F. Mattern: Verteilte Basisalgorithmen. Springer-Verlag, 1989

Wahl auf Ringen: Chang-Roberts-Algorithmus

Voraussetzung:

- Jeder Knoten besitzt eine eindeutige *ID*, auf welcher eine totale Ordnung definiert ist.

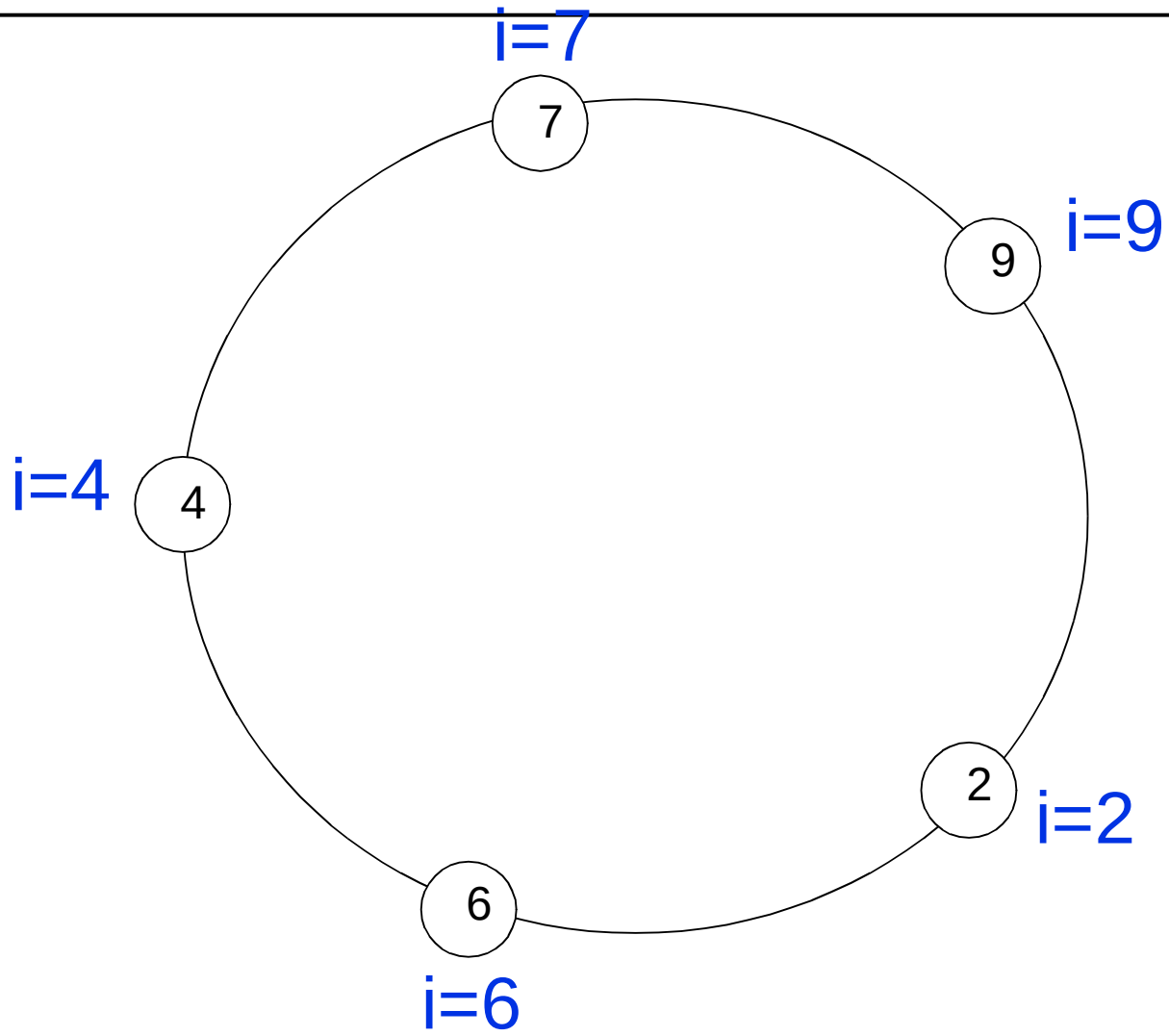
Ziel:

- Finde Knoten mit maximaler *ID*

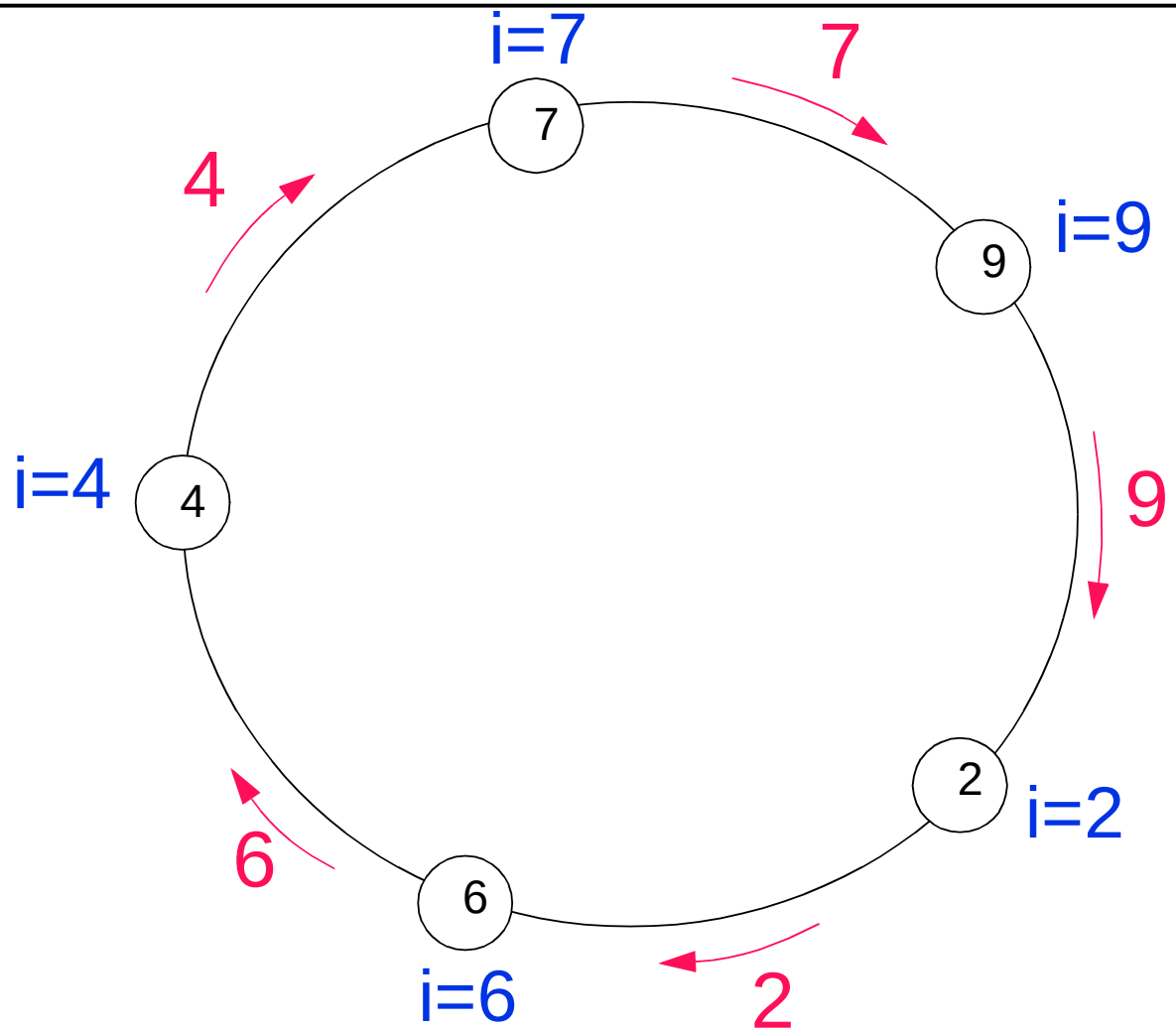
Ablauf

- Beim Start des Wahlalgorithmus (in allen Knoten): Sende eigene Knoten-ID im Ring weiter, Knoten wird wählbar
- Bei Empfang einer *ID* i_{rx}:
  - Falls $i_{rx} >$ eigene *ID*: i_{rx} weitersenden; Knoten nicht mehr wählbar
  - Falls $i_{rx} =$ eigene *ID*: Falls wählbar: als Anführer gewählt (sonst: Nachricht ignorieren)
  - Falls $i_{rx} <$ eigene *ID*: Nachricht ignorieren

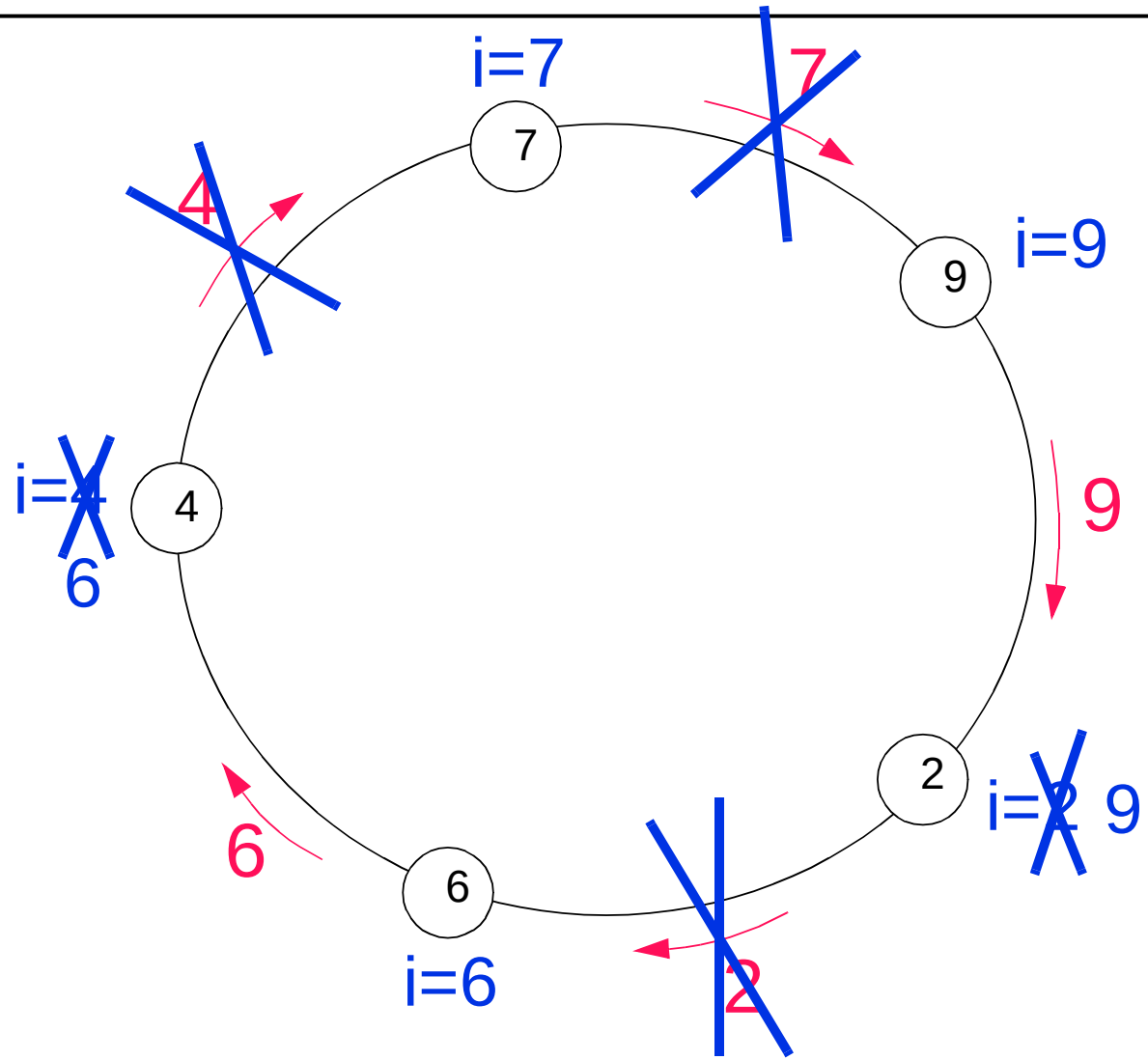
Chang-Roberts-Algorithmus



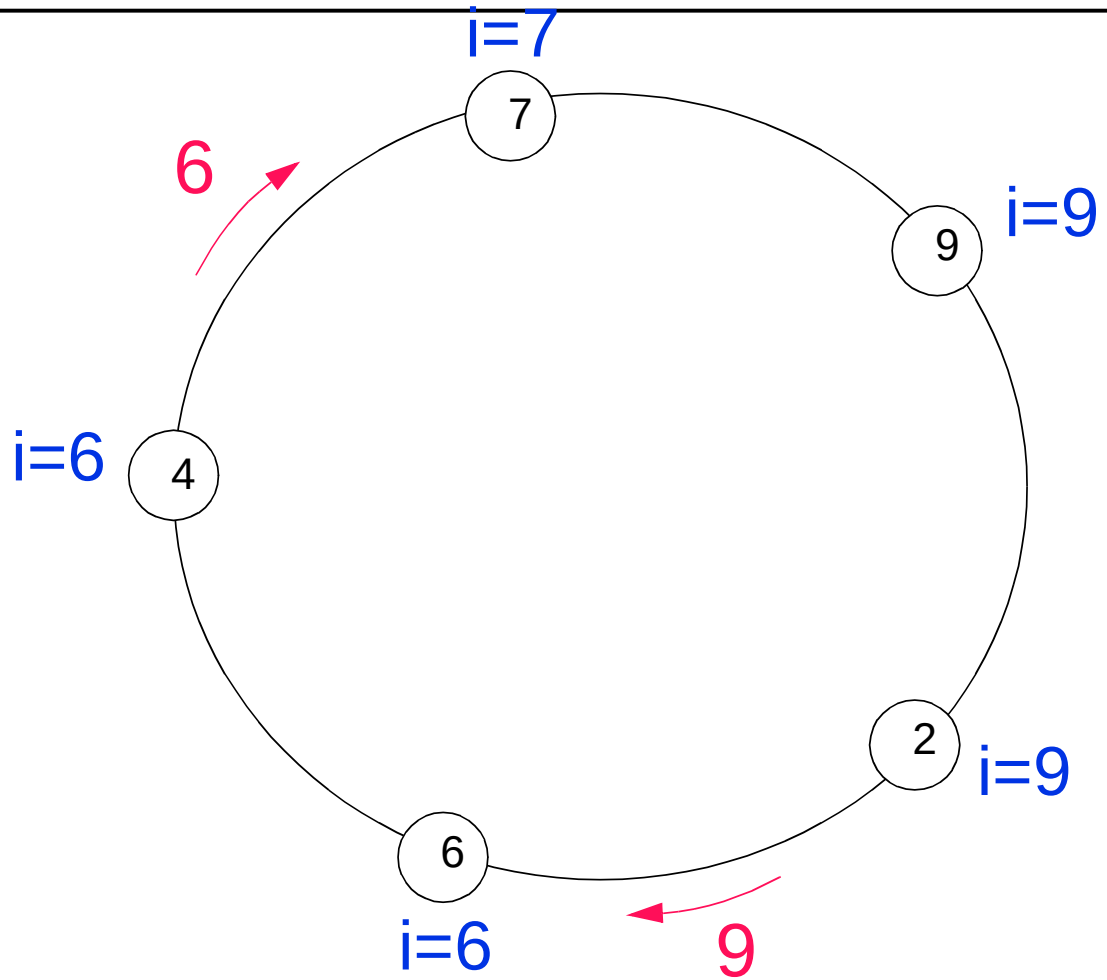
Chang-Roberts-Algorithmus



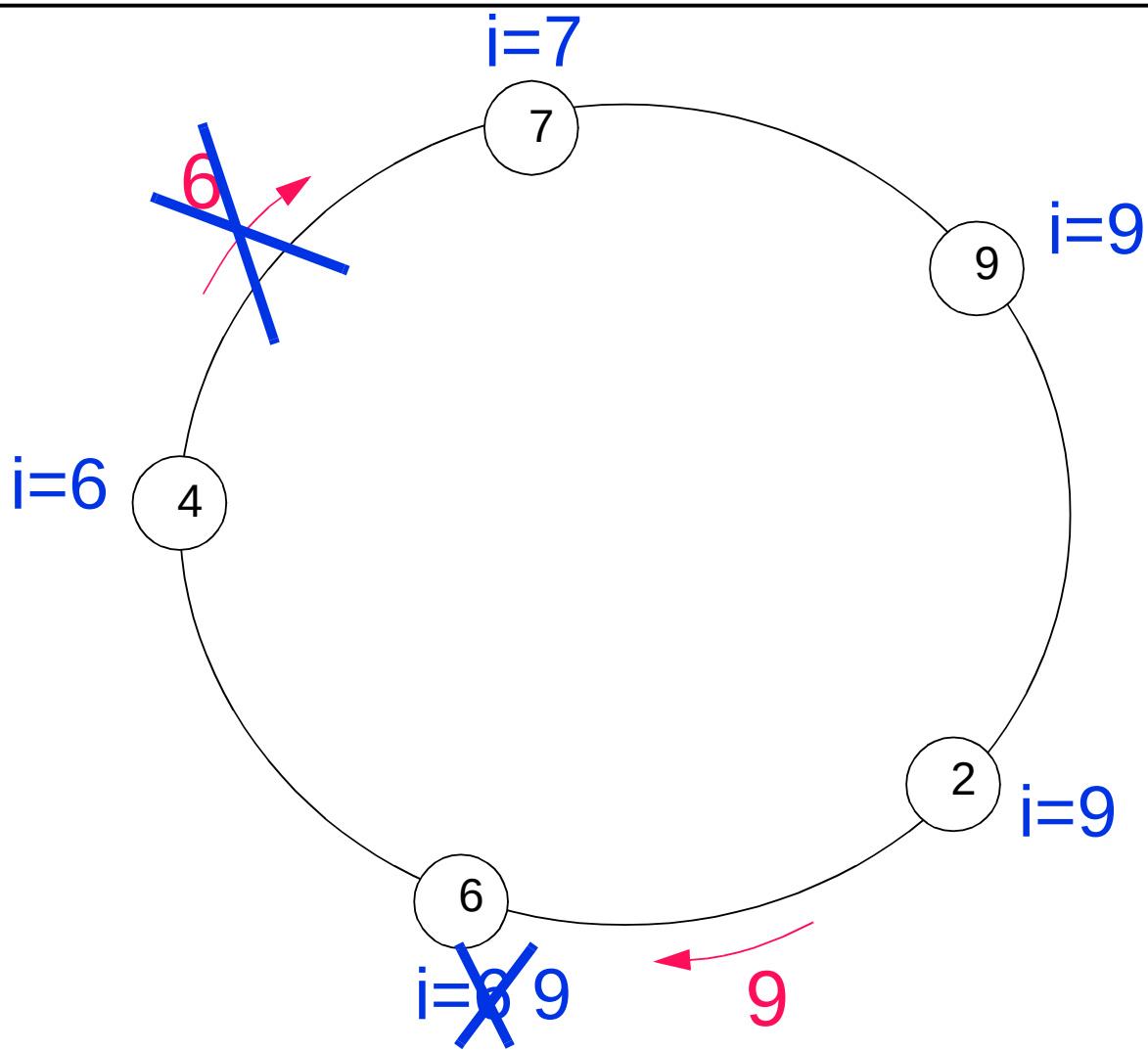
Chang-Roberts-Algorithmus



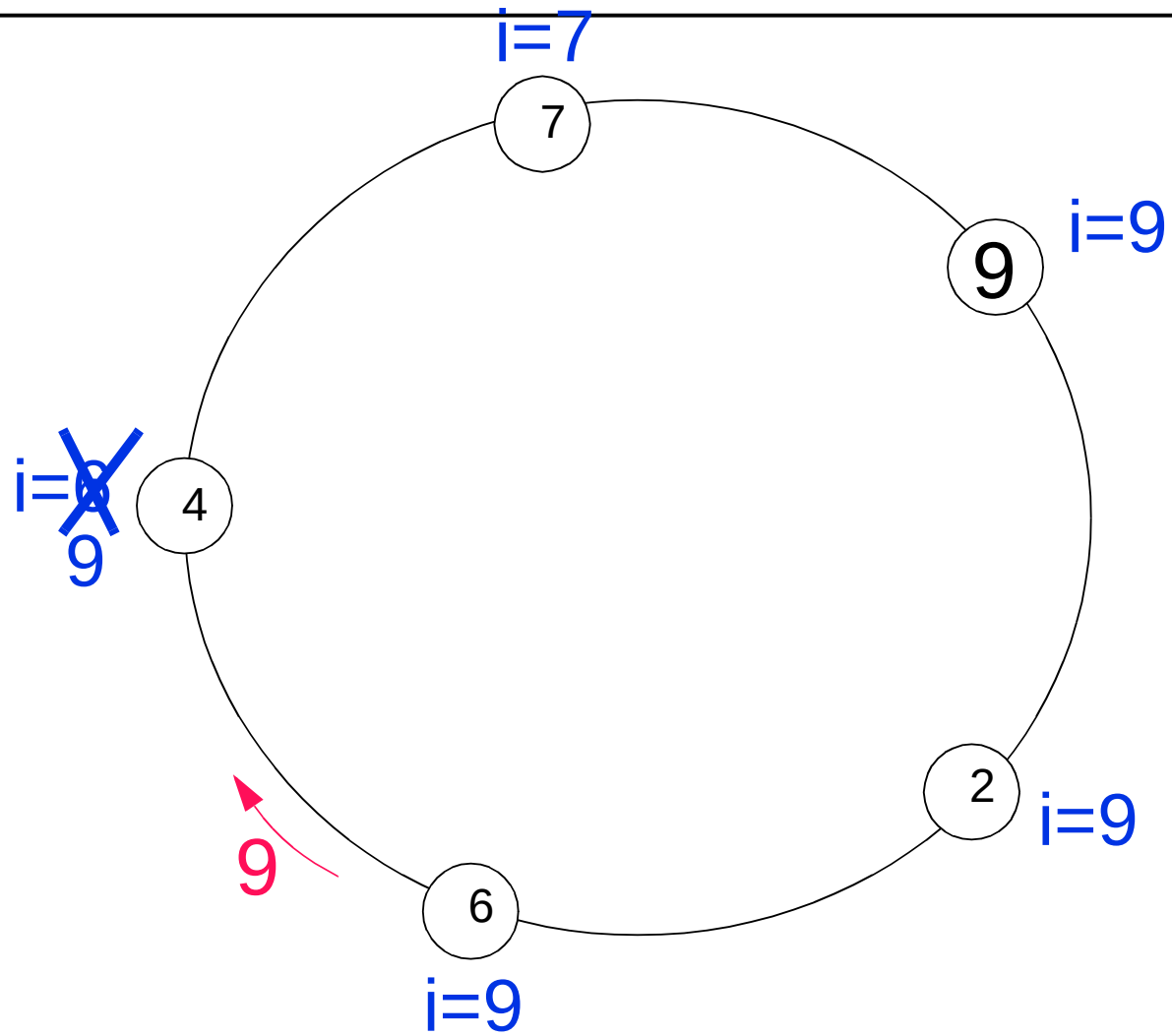
Chang-Roberts-Algorithmus



Chang-Roberts-Algorithmus



Chang-Roberts-Algorithmus



Chang-Roberts-Algorithmus

Eigenschaften

- Nach spätestens N Schritten gibt es einen Knoten, der sich für den Anführer hält
- Kein anderer Knoten hält sich je für den Anführer

Erweiterung

- Sobald ein Knoten als Anführer bestimmt ist, kann er dies durch spezielle Mitteilungs-Nachrichten allen anderen Knoten im Ring mitteilen
- Erst mit dieser Erweiterung wird eine Terminierung des Algorithmus erreicht (alle anderen Knoten warten sonst weiterhin auf Nachrichten)

Chang-Roberts-Algorithmus

Begründung der Korrektheit – Eindeutigkeit

- Initial wird eine Nachricht mit einer bestimmten *ID* nur vom Knoten mit dieser *ID* erzeugt
- Diese *ID* wird von weiteren Knoten nur dann weitergereicht, wenn deren eigene *ID* kleiner ist. Insbesondere: Der Knoten mit der größten *ID* reicht keine anderen *IDs* weiter
- Ein Knoten kann nur dann gewählt werden, wenn dessen *ID* alle Knoten durchlaufen hat
- Dies ist nach dem zweiten Punkt nur für die größte vorhandene *ID* möglich, es kann also maximal einen Anführer geben

Chang-Roberts-Algorithmus

Begründung der Korrektheit – Terminierung

- Eine Nachricht mit der größten *ID* wird gemäß Spezifikation von jedem Knoten weitergereicht.
- Nach *N* Nachrichten ist die größte *ID* wieder beim Ursprungsknoten angekommen; unter der Annahme, dass jede Nachricht in endlicher Zeit beim Empfänger ankommt, ist der Knoten mit größter *ID* in endlicher Zeit ermittelt

Chang-Roberts-Algorithmus

Komplexitätsbetrachtung

- Annahmen zur Analyse der Zeitkomplexität
  - Lokale Aktionen benötigen keine Zeit, gesendete Nachrichten kommen jeweils innerhalb einer maximalen Nachrichtenlaufzeit an
- Worst-Case-Betrachtung:
  - Zeit: $O(N)$ Nachrichtenlaufzeiten
  - Nachrichten: $O(N^2)$
    (Man ordnet die *IDs* absteigend an, dann ergibt sich $\sum_{i=1}^{N} i = n(n+1)/2$)
- Betrachtung des mittleren Werts:
  - Annahme: Alle möglichen Verteilungen der IDs sind gleichwahrscheinlich, IDs mit Werten von $1..N$
  - Zeit: unverändert $O(N)$
  - Es kann gezeigt werden: $O(N \log N)$ (Siehe Literatur)

Chang-Roberts-Algorithmus

Weitere Optimierungen möglich?

- Theoretische Erkenntnis (siehe Literatur): $O(N \log N)$ ist untere Schranke für die Nachrichtenkomplexität!
- Geht es besser als $O(N^2)$ im worst-case?

Chang-Roberts-Algorithmus – bidirektional

Einfache randomisierte Variante

- Jeder Knoten würfelt initial einen Wert aus $0, 1$ und sendet seinen Wert entsprechend zum linken oder zum rechten Nachbarn weiter
- Normaler Chang-Roberts-Algorirthmus wird ausgeführt; alle Nachrichten werden dabei in dem Umlaufsinn weitergereicht, in dem sie bei einem Knoten ankommen

Eigenschaften

- Worst-case beträgt immer noch $O(N^2)$ Nachrichten, tritt aber mit geringerer Warscheinlichkeit auf
- Mittlere Nachrichtenzahl ist günstiger, aber noch aufwendiger zu berechnen und asymptotisch immer noch $O(N \log N)$

Anmerkungen zur Wahl auf Ringen

- Mögliche Probleme in der Praxis
  - Nachrichtenverluste
  - Rechnerausfälle
  - Mehrfache Ausführung
  - Doppelt vergebene IDs

- Konsequenz

Einfache algorithmische Idee in einem einfachen Systemmodell muss in der Realität ggf. in komplexere Algorithmen eingebettet werden, um den realen Randbedingungen gerecht zu werden

.

Wahl auf Bäumen

Grundidee:

Traversieren des kompletten Baumes, um den Knoten mit der größten *ID* zu ermitteln

- Sequentielles Traversieren möglich
- Paralleles Traviersieren bietet sich für verteilte Systeme an

Sequentielles Traversieren

Bekanntes *depth first*-Vorgehen wie bei lokalen Datenstrukturen:

```
class Node implements NodeRemote {

 ArrayList<Node> children = ...;
 int id = ...;

 int findMax() {
  int maxVal = id;
  Iterator<Node> it = children.iterator();

  while(it.hasNext()) {
   int val = it.next().findMax();
   if ( val > maxVal )
     maxVal = val;
  }
  return maxVal;
}
```

Paralleles Traversieren

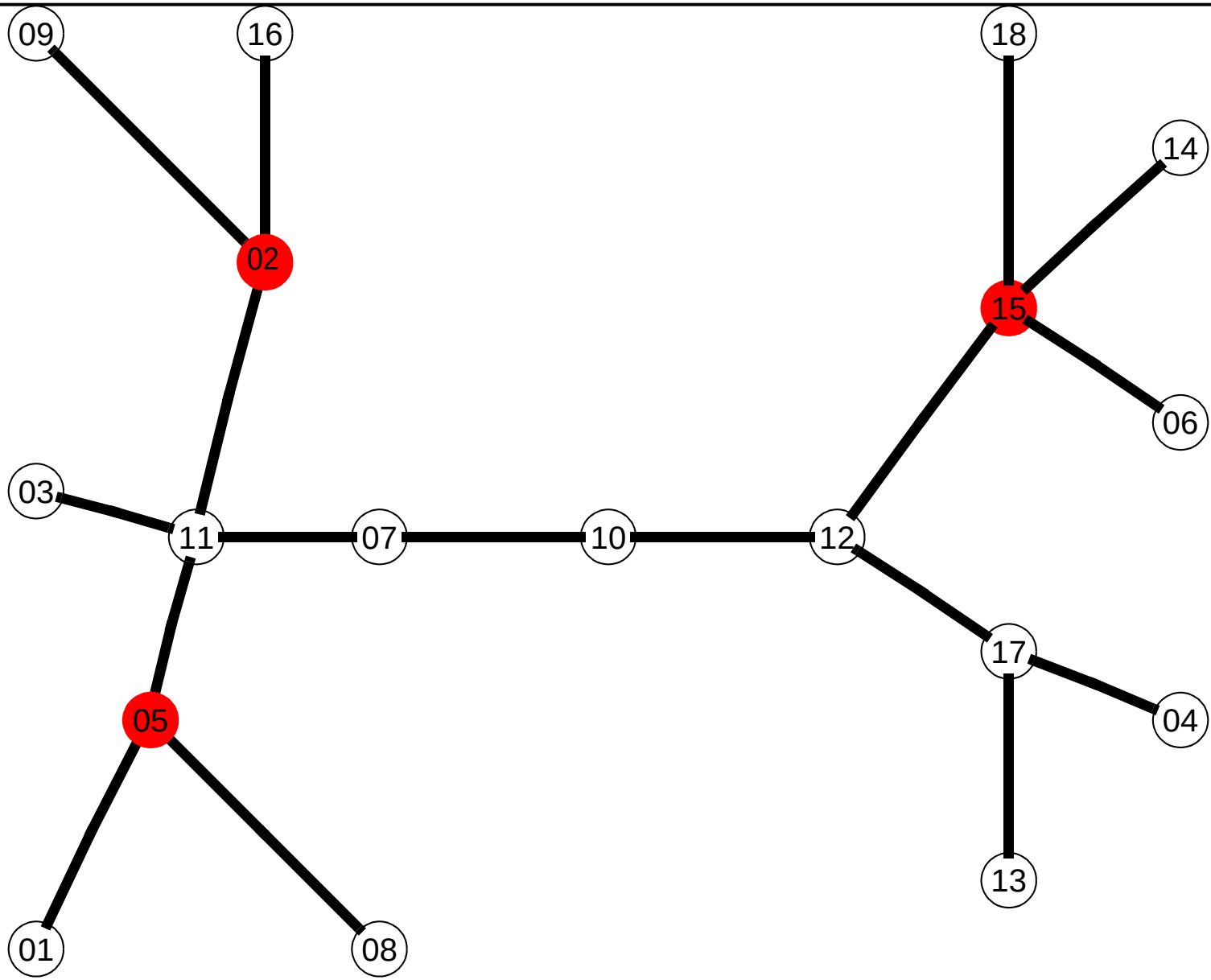
Zeitgewinn durch Parallelisierung

- Sequentiell: Bei N Knoten $2(N-1)$ Kommunikationsschritte!

Wellenverfahren

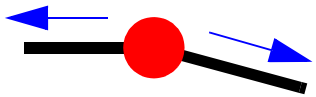
- *Explosionswelle* durchläuft den Baum bis zu den Blättern
- *Echowelle* läuft wieder zurück und bestimmt Anführer
  $\Rightarrow$ teilweise Überlappung mit Explosionswelle möglich
- *Informationswelle* kann alle Knoten über das Ergebnis der Wahl informieren

Wahl auf Bäumen

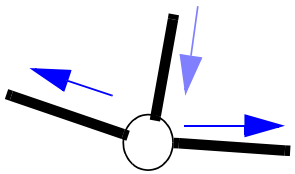


Wahlalgorithmus für Bäume

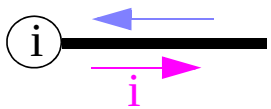
- Ein Starter sendet Explosionsnachrichten in alle Richtungen



- Ein innerer Knoten, der erstmals eine Explosionsnachricht empfängt, sendet wiederum Explosionsnachrichten in alle anderen Richtungen aus

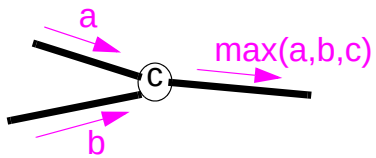


- Ein Blattknoten, der eine Explosionsnachricht erhält, sendet eine Echonachricht mit seiner ID zurück

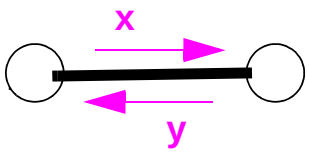


Wahlalgorithmus für Bäume

- Ein innerer Knoten mit k Kanten, der auf $k-1$ Kanten Echonachrichten erhalten hat, sendet auf der übrigen Kante eine Echonachricht mit dem Maximum aller *IDs*

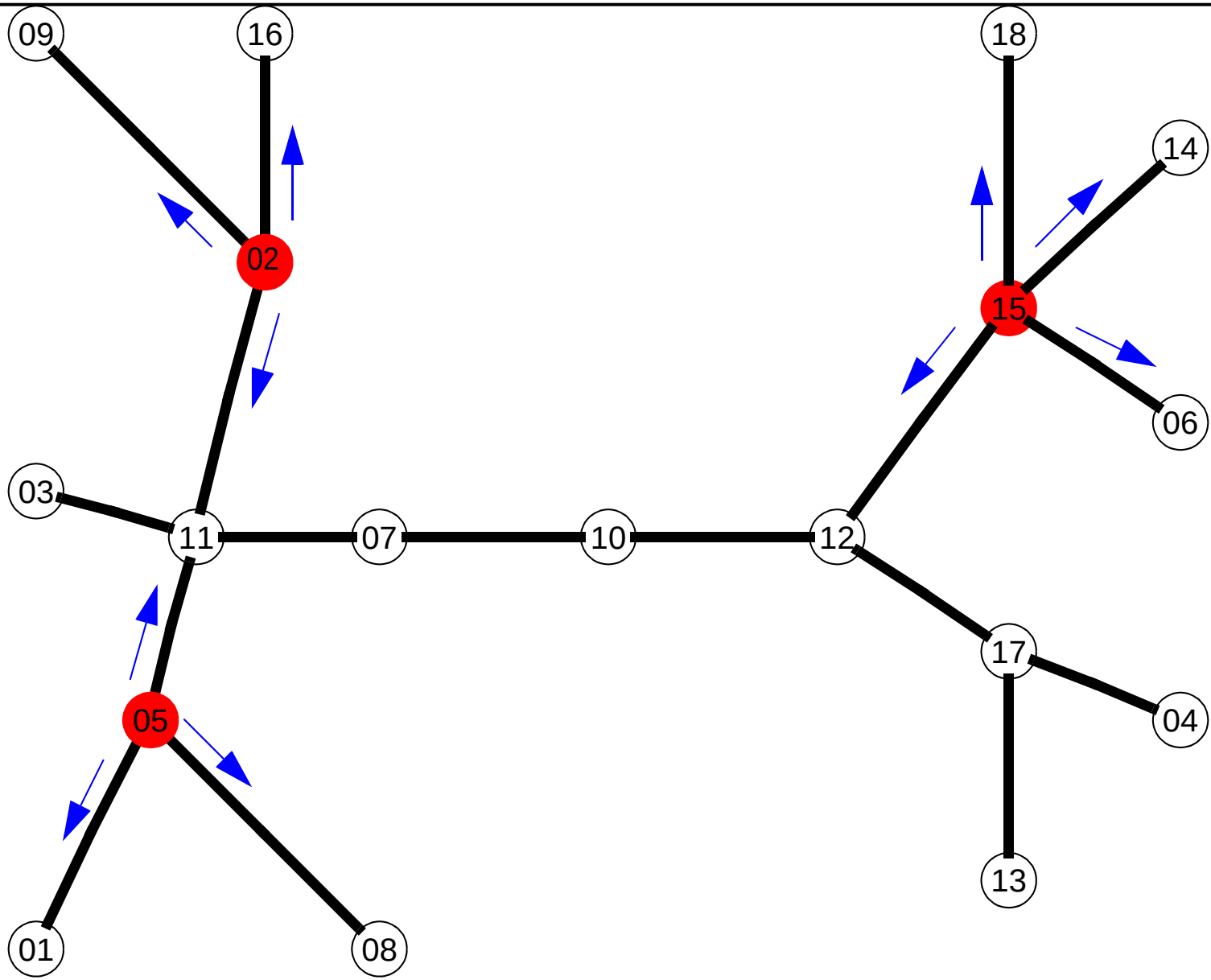


- Erhält ein Knoten nach Versenden seiner eigenen Echonachricht eine weitere Echonachricht, so ist das Maximum beider *IDs* die höchste ID-Nummer im Netz

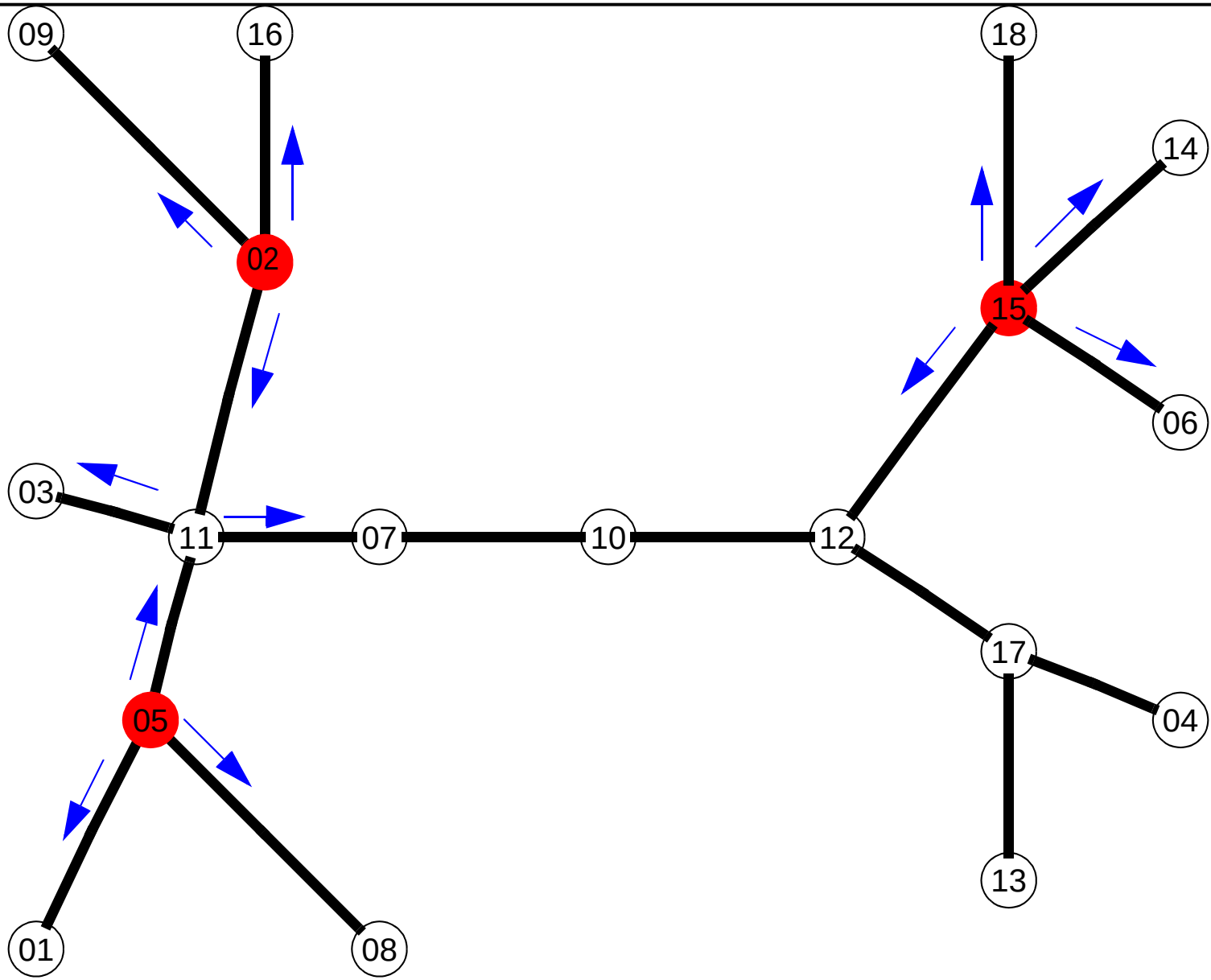


  - Muss lediglich ein beliebiger Anführer bestimmt werden, so kann man jetzt einfach einen der beiden Knoten auswählen
  - Ansonsten kann man in einer Informationswelle alle Knoten über die ermittelte höchste ID informieren

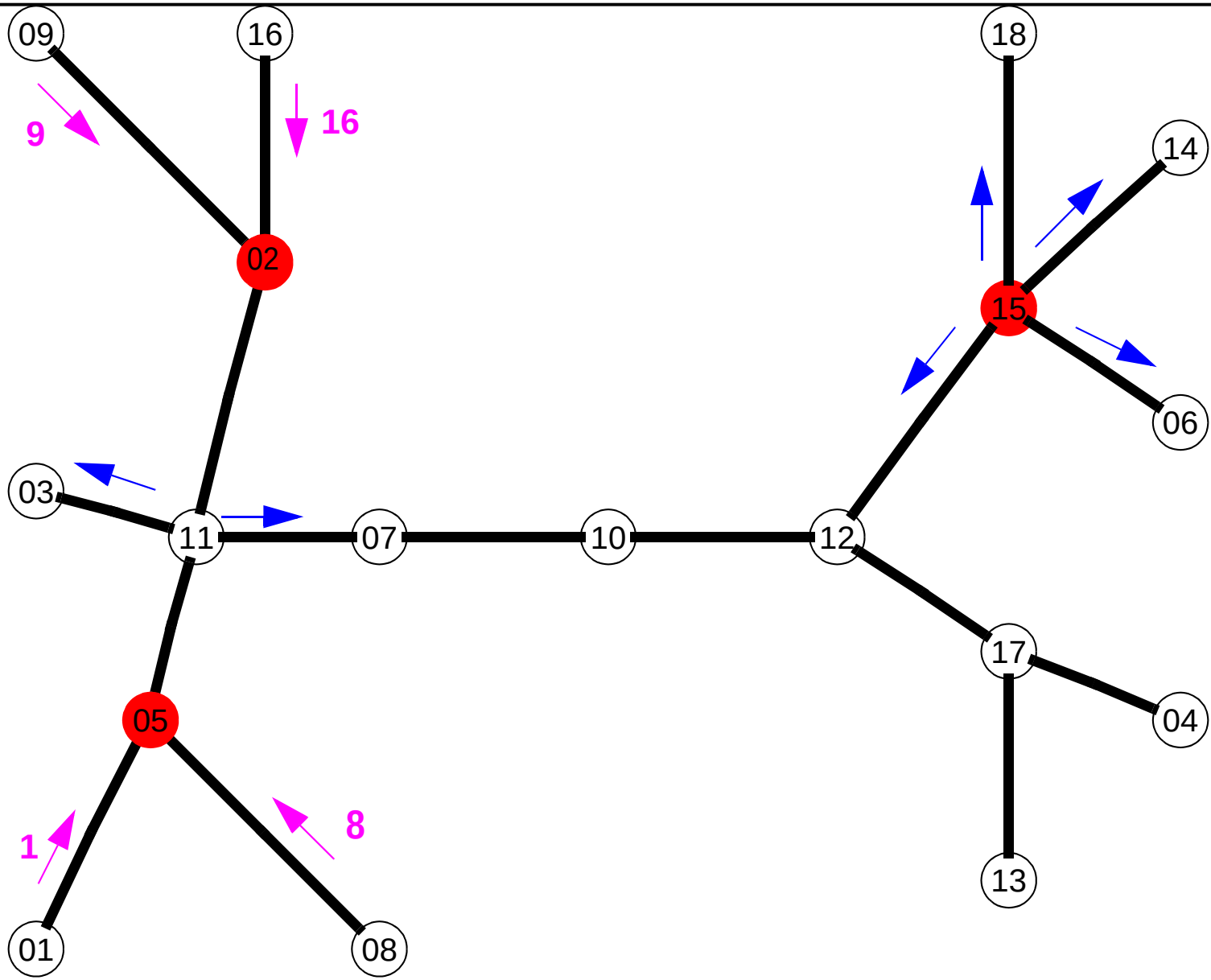
Wahl auf Bäumen



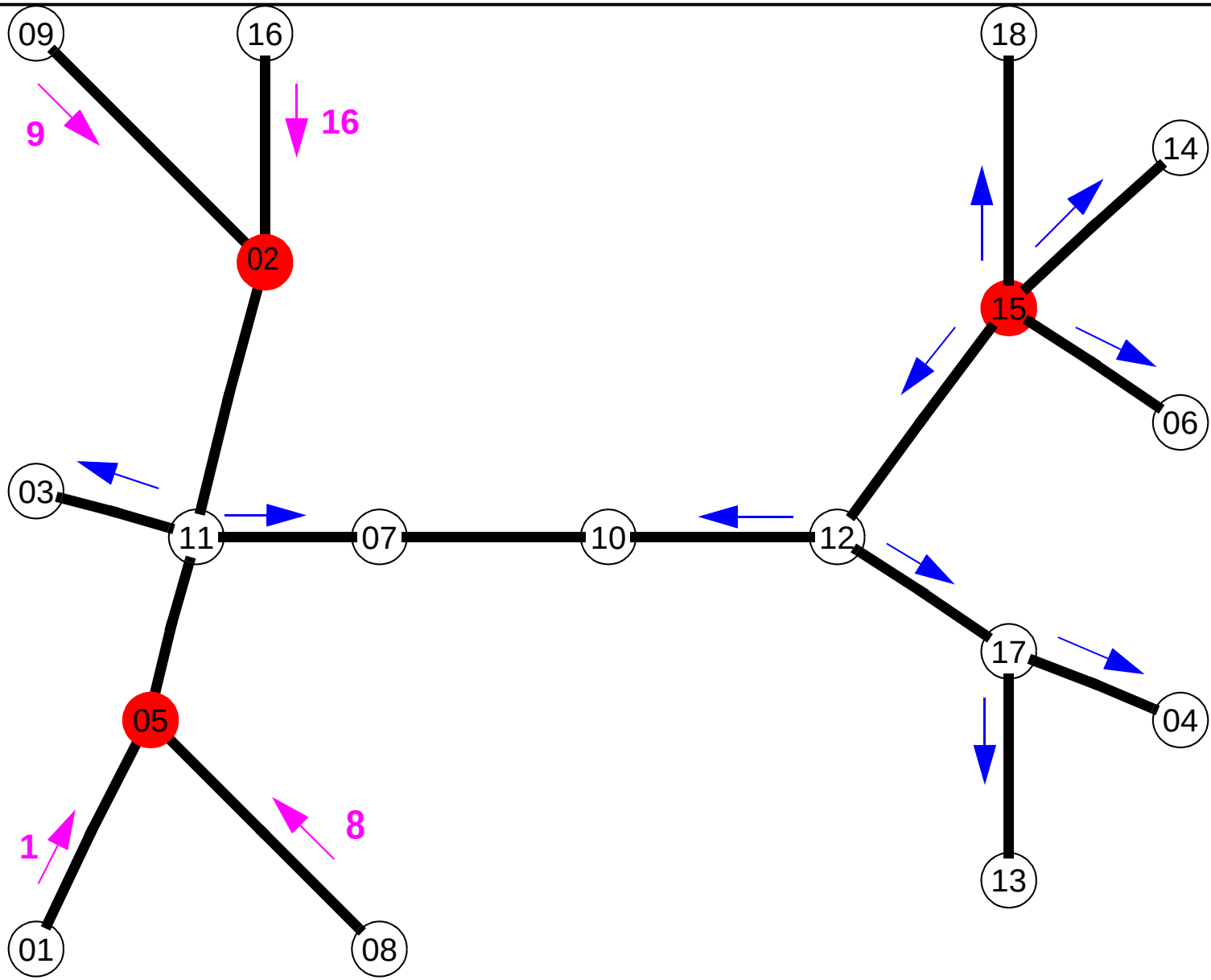
Wahl auf Bäumen



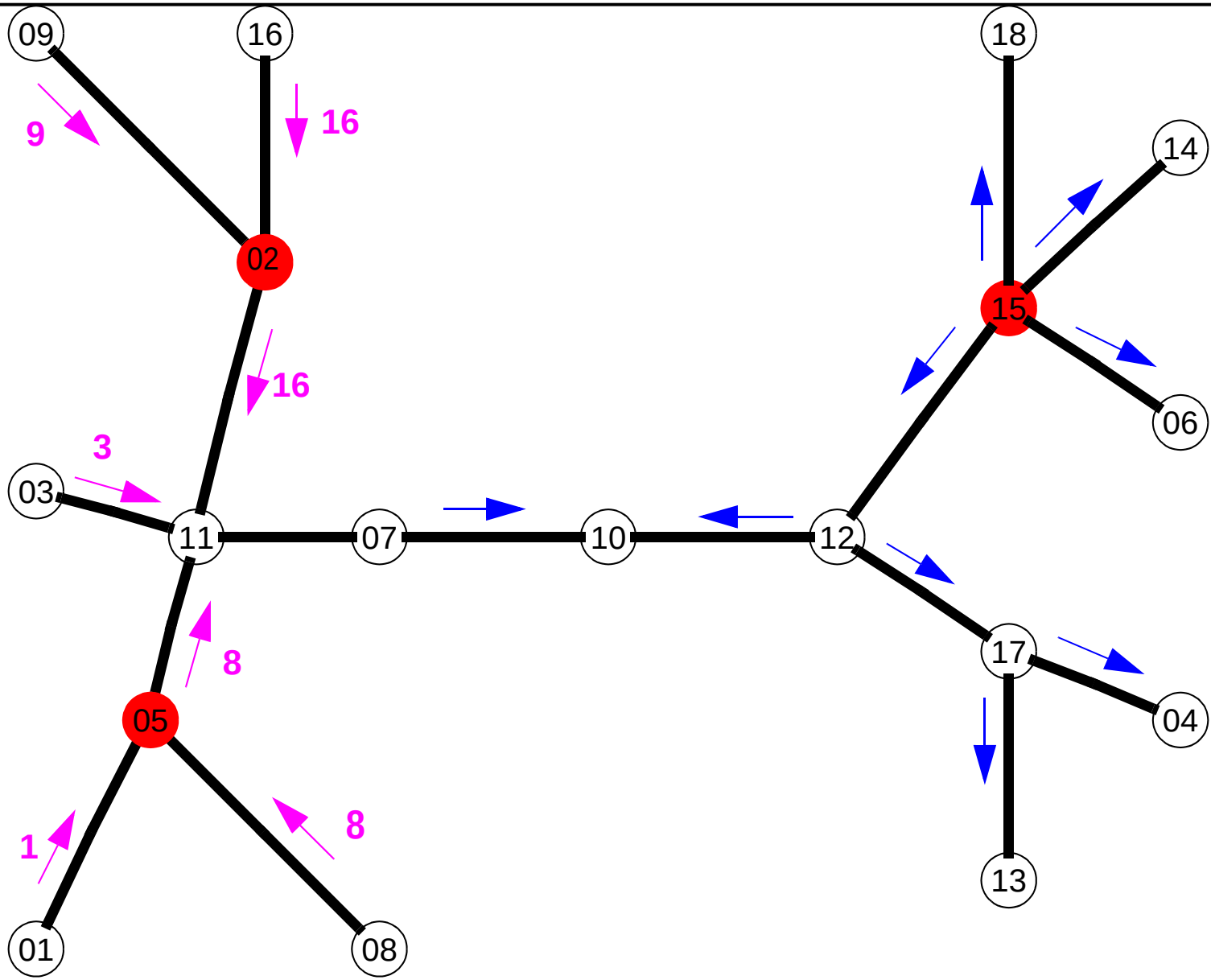
Wahl auf Bäumen



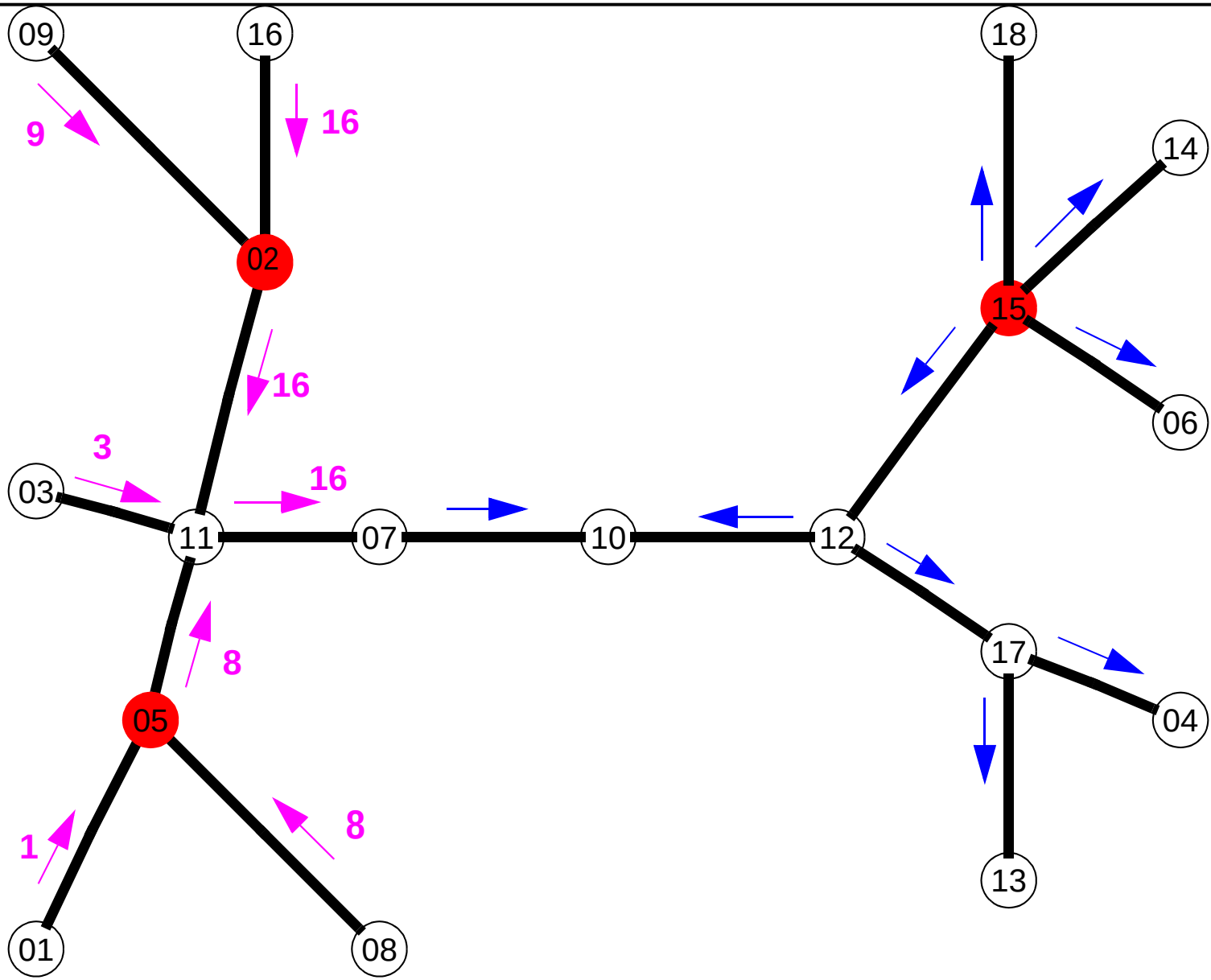
Wahl auf Bäumen



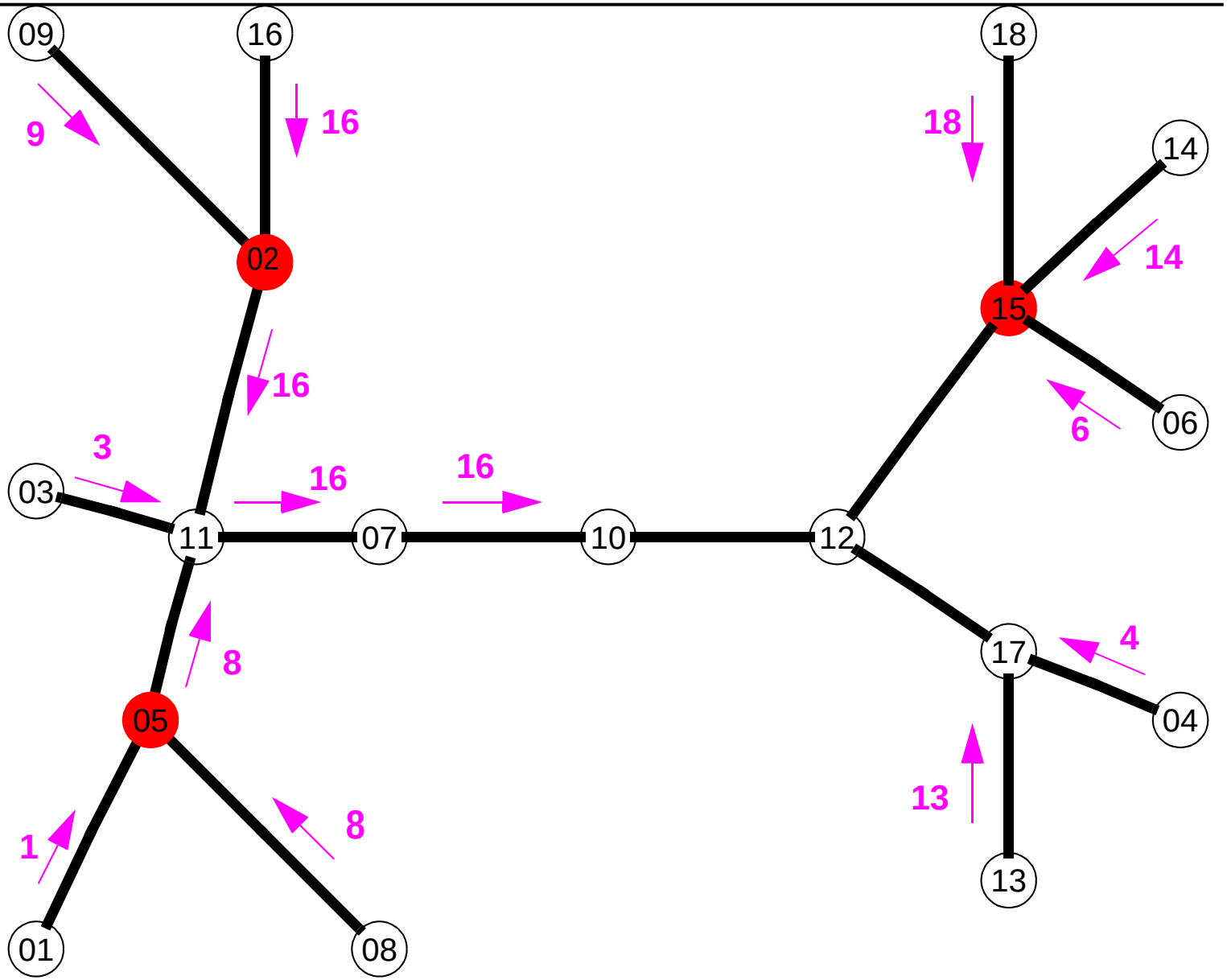
Wahl auf Bäumen



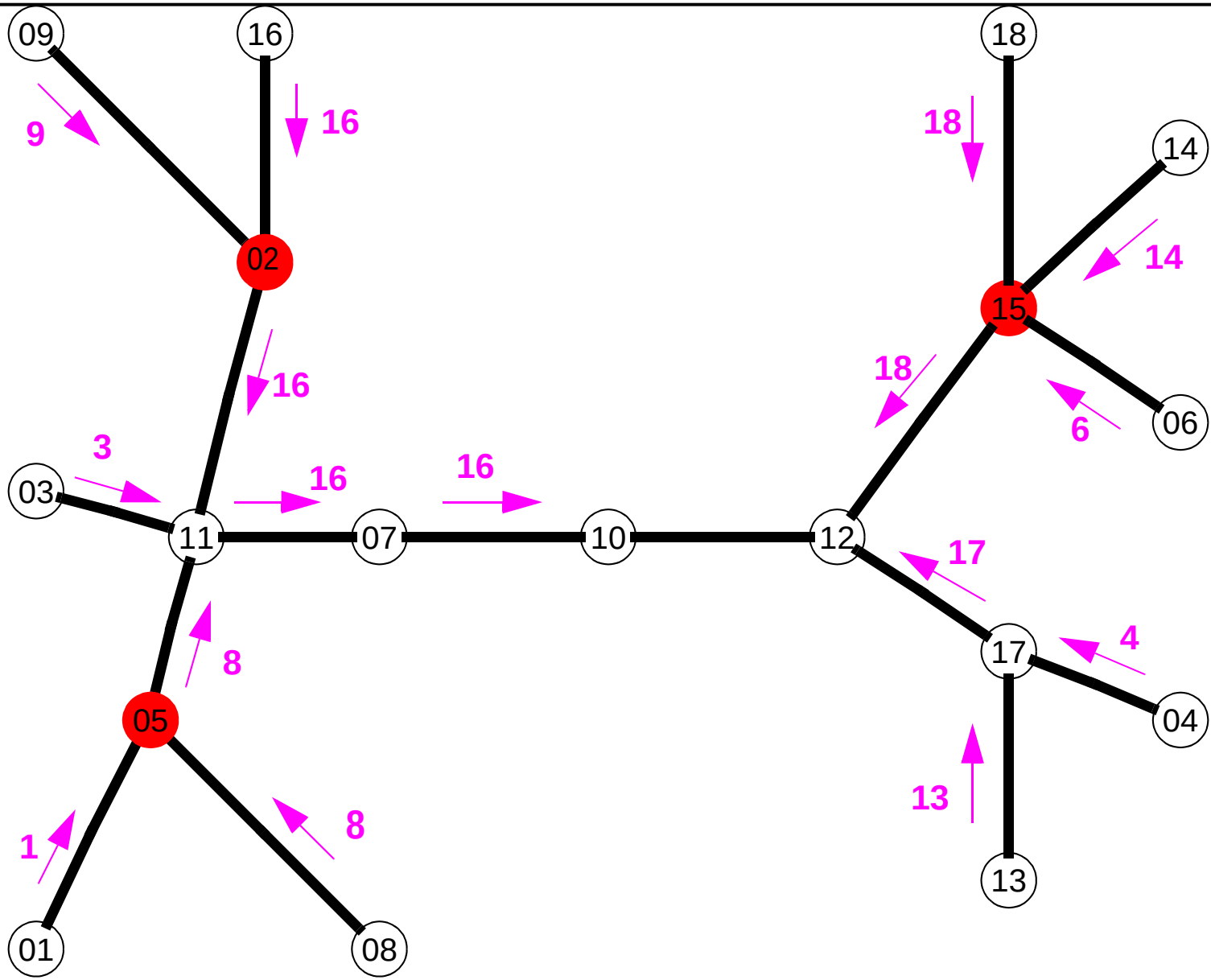
Wahl auf Bäumen



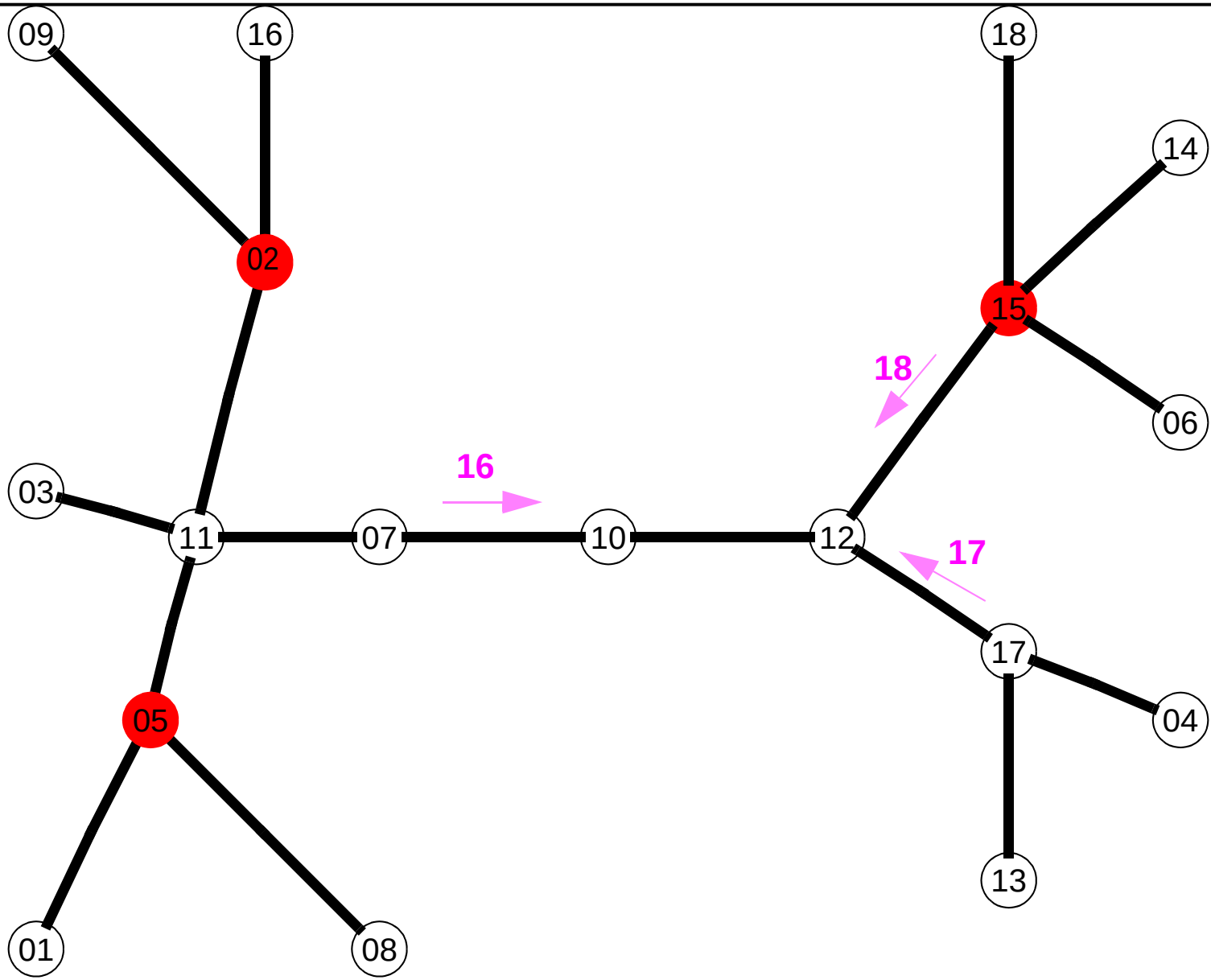
Wahl auf Bäumen



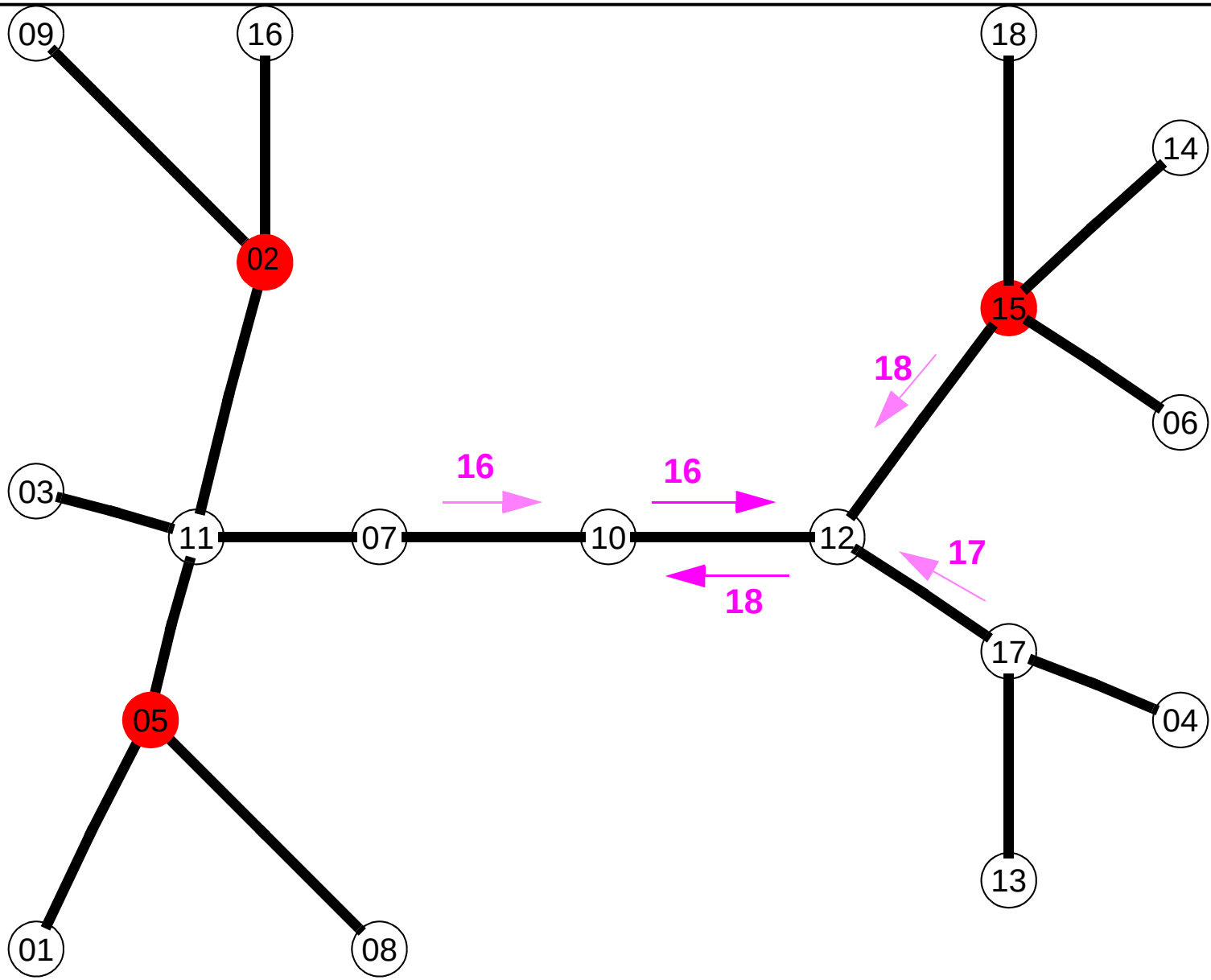
Wahl auf Bäumen



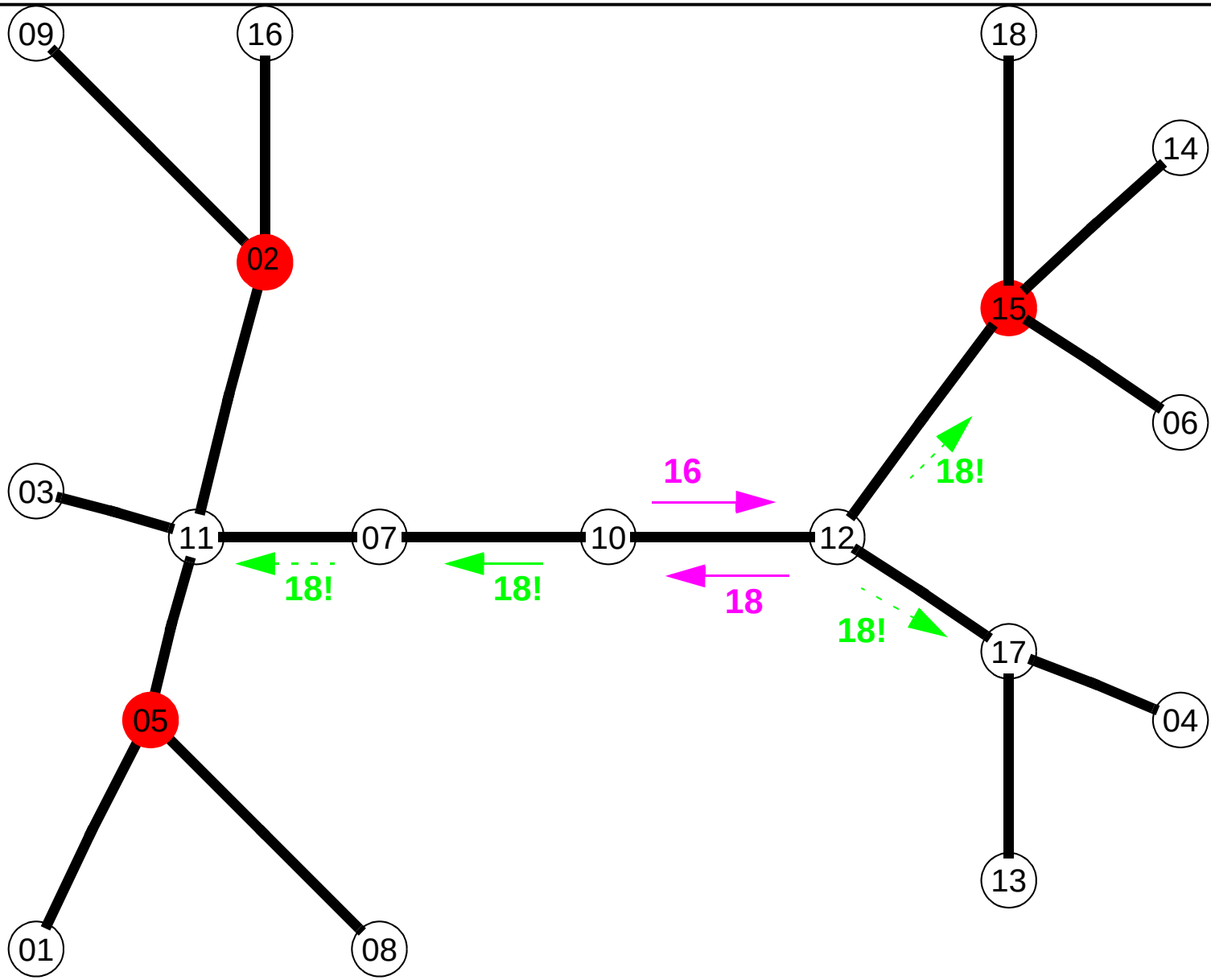
Wahl auf Bäumen



Wahl auf Bäumen



Wahl auf Bäumen



Wahl auf Bäumen: Zusammenfassung

- Initiierung des Algorithmus durch Start-Welle (Election-Welle)
- Bildung von gerichteten Baumkanten in einer Echo-Welle; dabei Informationssammlung (z.B. größte ID) möglich
- Es kann genau eine Kante geben, bei der Echo-Nachrichten in beiden Richtungen laufen
  - Gewinner kann durch gesammelte Information ermittelt werden
  - oder: Auswahl eines der beiden Knoten an dieser Kante als Gewinner

Wahl auf beliebigen Topologien

Sequentielle Traversierung des Netzes

- Pfadverfahren
- Kantenfärbungsverfahren

Parallele Traversierung des Netzes

- Echo/Election-Algorithmus
- ... und Varianten davon

Konstruktion eines virtuellen Baums auf dem Netz

Wahl auf beliebigen Topologien

- Wahlalgorithmen auf beliebigen Topologien: Algorithmen für Baumstrukturen + Zyklenerkennung
- Explorer- und Echowelle, ggf. Informationswelle
- Konstruktion eines virtuellen Baumes
- Verschiedene Varianten möglich:
  - Sequentielle Traversierung einfach, aber hoher zeitlicher Aufwand
  - Parallele Traversierung komplexer, aber wesentlich geringere Zeitkomplexität