Verteilte Systeme

5. Semester/6. Semester

Patrick Jungk

DHBW - Stuttgart

3. Oktober 2024

About

Zum Kennenlernen:

- BA-Studium (Friedrichshafen) Kooperation: EADS-Dornier GmbH
- Diplom im Umfeld Flugdatenschreiber / Testsysteme
- Lotus Notes Entwicklung / Entwicklung Kollaborationssysteme bei EADS-Deutschland GmbH
- Dozent an der BA-FN für „Verteilte Systeme“
- Master Studium Universität Kontanz (Schwerpunkt: interactive data analysis)
- Deutscher Sparkassen Verlag (DSV): Bereich Kryptosysteme (verteilte)
- aktuell: Mercedes-Benz AG: Bereich Cybersecurity / PKI / Ausbildung

Organisatorisches

Die Vorlesung umfasst:

- Präsenszeit und Selbststudium
- bei Ausfall: möglichst Ersatztermin
- KI gestützte Praxis an konkreter Aufgabe
- Grundidee der Vorlesung: Theorie und praktische Erfahrung „erleben“: Schwerpunkt Systeme im Netz

Kontakt per email: patrick.jungk@boxwork.eu
Kursweb-Inhalte: Moodle (druckerfreundlichen) Skript und Übungen

KI gestützte Praxis

Grundidee:

1. KI Tools in Ausarbeitung erlaubt
2. Ausarbeitung mit Analyse
3. geteilte und eingeschränkte Ressourcen
4. ACHTUNG: kritische Reflexion als Schwerpunkt erwartet

Inhalt gemäß Vorlesungsplan I/II

- Einführung in die verteilten Systeme
- Anforderungen und Modelle, Hard- und Softwarekonzepte
- Multiprozessor, Multicomputer, Betriebssystemunterstützung
- Verteilte Dateisysteme
- Kommunikation in verteilten Systeme

Inhalt gemäß Vorlesungsplan II/II

- Sockets, Remote Procedure Calls (RPC), Remote Method Invocation (RMI)
- Middlewarearchitekturen
- Synchronisation, Zeit und Uhren (physikalisch, logisch)
- Replikation
- verteilte (Internet) Anwendungen
- Erstellen eine Anwendung

Inhalt geplant

siehe Inhalt gemäß Vorlesungsplan mit folgender Anpassung:

- Fehlertoleranz in verteilten Systemen
- Verteilte Dateisysteme =>verteilte Datenhaltung
- Erstellen eine Anwendung anhand der Übungen

Nicht betrachtet werden folgende Aspekte (aus Zeitgründen)

- Sicherheit in verteilten Systemen
- verteilte Datenbanken

Kapitelaufbau

bis auf die Einleitung:

1. Theorieunterkapitel mit Hinweis zu Selbststudium
2. Praxis: Diskussion, Aufgabe, Übung/Projekt

Prüfung

1. Testat/Seminarentwurf mit Konzept (ca. 15 Seiten)
2. Programmentwurf
3. Test und Reflexion (als Teil des Konzepts + Ausführung)

Buchempfehlungen

- **„Verteilte Systeme - Grundlagen und Paradigmen“ - Andrew Tanenbaum, Marten von Steen, Pearson Studium (April 2007), ISBN 978-3827372932** (Bibel)
- “Distributed Systems: Concepts and Design”- Coulouris, Dollimore and Kindberg (2001), 3rd Edition
- “UNIX Network Programming” - Stevens (1990)
- “Linux Socket Programming” - Walton (2001)
- “Computer Architecture: A Quantitative Approach” - Hennessy and Patterson (2003), 3rd Edition
- “Java Network Programming” - Harold (2004), 3rd Edition
- “Middleware in Java: Leitfaden zum Entwurf verteilter Anwendungen” - M. Mathes, Steffen Heinzl, Vieweg (2005)

Inhaltsverzeichnis

1 Einleitung

2 Kriterien; Konzepte

3 Kommunikation

4 Synchronisation, Koordination

5 Fehlertoleranz, Fehlerhandling

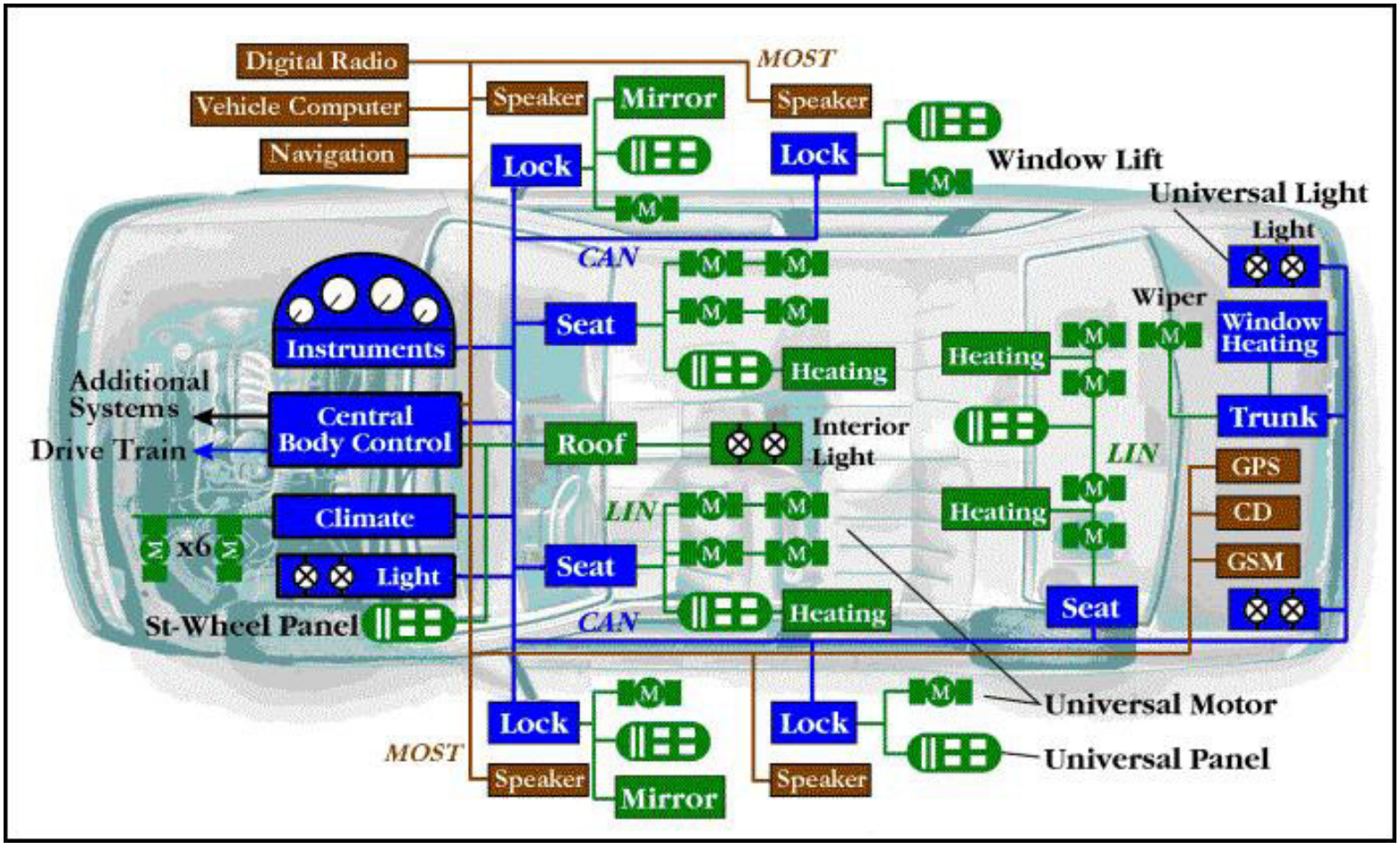
6 Verteilte Daten

Inhaltsverzeichnis Kapitel 1

1 Einleitung
- Motivation
- Ziele und Definition

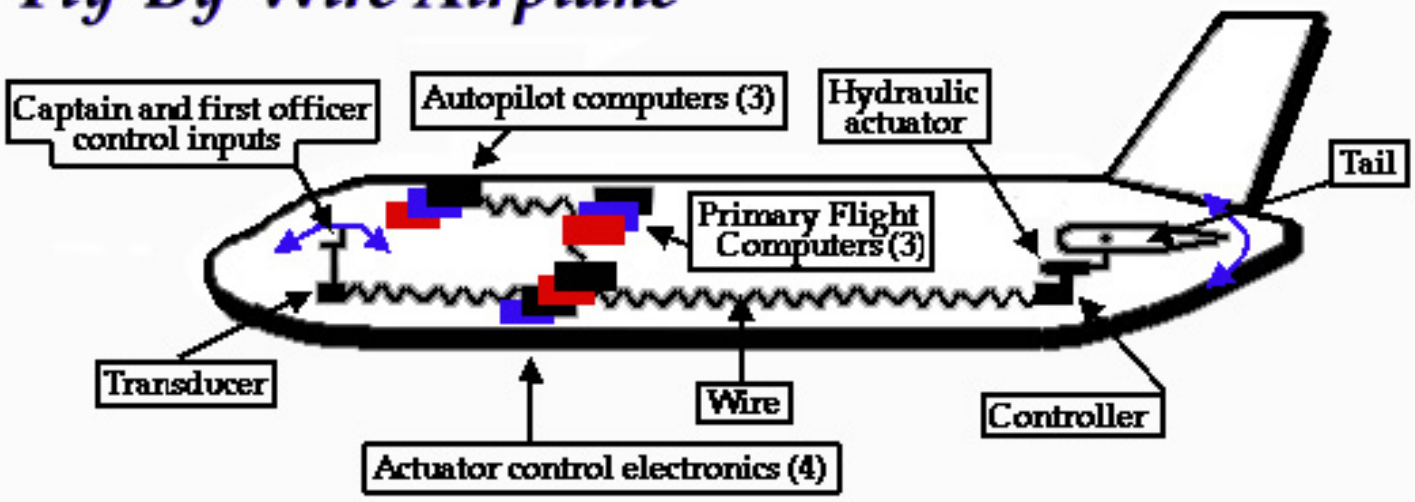
zunächst Allgemeine Motivation und Definitionen...

Wie viele Systeme sehen Sie?

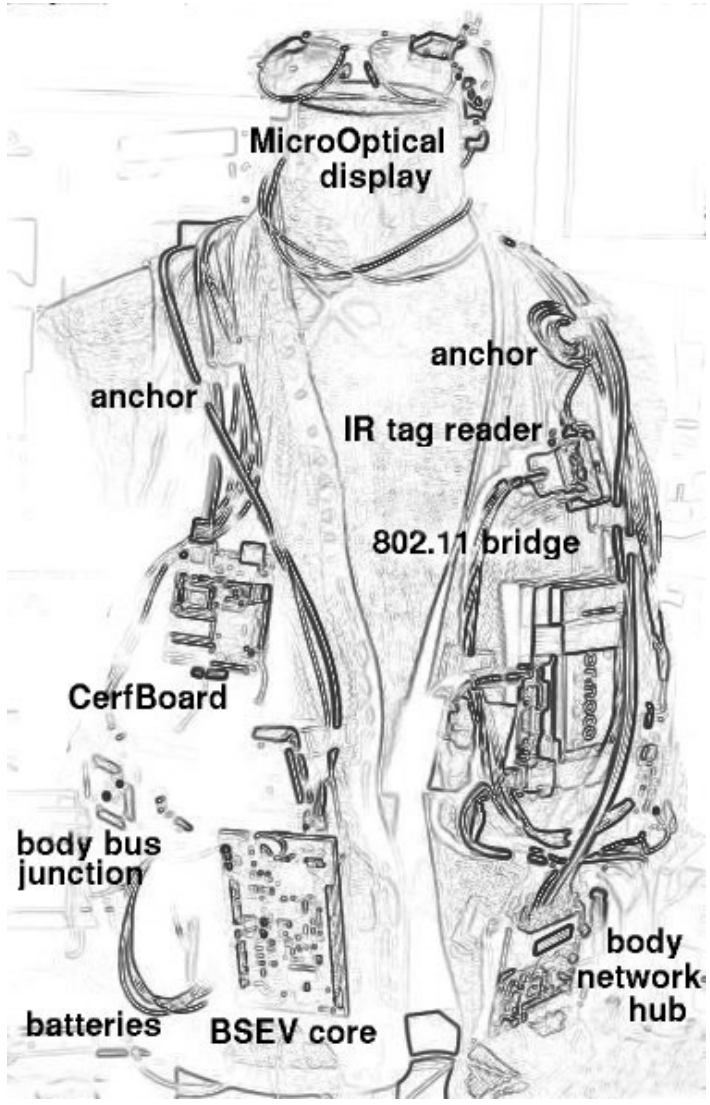

Digital Radio
Vehicle Computer
Navigation
MOST
Speaker
Mirror
Speaker
Lock
Lock
Window Lift
Universal Light
Light
CAN
Wiper
Instruments
Seat
Heating
Window Heating
Heating
Additional Systems
Drive Train
Central Body Control
Roof
Interior Light
LIN
Trunk
GPS
CD
GSM
Climate
x6
Light
LIN
Heating
Seat
St-Wheel Panel
CAN
Heating
Seat
Universal Motor
Universal Panel
Lock
Lock
MOST
Speaker
Speaker
Mirror

... und hier?

Fly-By-Wire Airplane



und hier?


MicroOptical
display
anchor
anchor
IR tag reader
802.11 bridge
CerfBoard
body bus
junction
batteries
BSEV core
body
network
hub

Ziele

Oberstes Ziel ist es den **Benutzer** mit den ihm zur Verfügung stehenden **Ressourcen zu verbinden**. (z.B.: Daten, HW-Ressourcen wie Drucker, (Festplatten-)Speicher wie NAS, Funktionen und Dienste)

Ein weiteres Ziel ist die **Zusammenarbeit über weite Distanzen** mit mehreren „Instanzen“ zu gewährleisten: (z.B.: Video-Konferenzen, Git, Webseminare, Verteiltes Rechnen wie SETI@Home oder Worldcomminity-Grid)

Definitionen

nach Andrew S. Tanenbaum:
„ein Zusammenschluss unabhängiger Computer, welcher sich für den Benutzer als ein einzelnes System präsentiert.“
nach Peter Löhr:
„eine Menge interagierender Prozesse (oder Prozessoren), die über keinen gemeinsamen Speicher verfügen und daher über Nachrichten miteinander kommunizieren“
weitere Definition:
“ein Zusammenschluss unabhängiger Ressourcen und Funktionen, die sich dem Benutzer als ein einzelnes System präsentieren.”

Kurz...

Vieles erscheint wie eins !

!!! DARE TO PREPARE !!!

Inhaltsverzeichnis

2 Kriterien; Konzepte
- Kriterien
- Hardware
- Software
- Middleware

Kriterien

Allgemein

Um ein verteiltes System zu beschreiben, müssen gewisse Kriterien erfüllt sein. Diese Kriterien bestimmen das WIE, WAS, WO, WANN.

- Transparenz
- Offenheit
- Skalierbarkeit
- Sicherheit

Transparenz

Definition

Die Tatsache zu verbergen, dass es sich bei einem verteilten System nicht um **ein** sondern um **mehrere** Rechnersysteme handelt, nennt man **Transparenz**.

Transparenz

Unterscheidung

- Zugriffstransparenz
- Positionstransparenz
- Migrationstransparenz
- Relokationstransparenz
- Nebenläufigkeitstransparenz
- Fehlertransparenz
- Persistenztransparenz

Diskussion: was bedeutet was?

Transparenz

Transparenzgrad

Aus folgenden Gründen ist es nicht immer sinnvoll alle Transparenzarten zu erfüllen.

- Leistung kann unzumutbar schlecht sein
- Konsistenz über große Entfernungen kann zu hohem Datenfluss kommen, bzw. mehrere Sekunden dauern

Offenheit

Definition und IDL

Ein **offenes verteiltes System** ist ein System, dass seine Dienst **Standardregeln entsprechend** anbietet (Syntax und Semantik).

- Solche Regeln werden in **Protokollen** formalisiert (Übertragungsprotokoll wie TCP/IP, UDP...)
- Dienste werden im allgemeinen durch **Schnittstellen** spezifiziert, sog. **IDL (Interface Definition Language)**

Die IDL definiert:

- exakter Namen einer Funktion
- Übergabeparameter
- Rückgabeparameter

Offenheit

Miniatur Clustering

Um ein offenes System flexibel zu halten ist es notwendig das System mit einer Menge relativ **kleiner** leicht **austauschbarer** Komponenten zu realisieren. Das impliziert Definitionen von Schnittstellen auf:

- Benutzerebene
- zu anderen Komponenten

Dies ermöglicht einen hohen Grad an Benutzerfreundlichkeit, kann jedoch zur „Überforderung“ des Benutzers aufgrund vieler zu konfigurierbarer Komponenten führen.

Skalierbarkeit

Definition

Die Skalierbarkeit eines Systems kann, nach Neumann (1994), nach (mindestens) 3 Dimensionen gemessen werden:

1. Größe (Benutzer, Ressourcen)
2. Geographisch
3. Administrativ

Ein System verliert in der Regel an Leistung, wenn es in eine der Dimensionen erweitert wird.

Skalierbarkeit

Problem (Größe)

Eine Zentrale Ressource kann allein wegen der Kommunikation die Skalierbarkeit in der Größe behindern.

- Zentrale Dienste - Ein einziger Server für alle Benutzer
- Zentrale Daten - Eine einzigen Telefonverzeichnis für alle Benutzer
- Zentrale Algorithmen - Zentrale Routing Tabelle für Routing

Diskussion: warum?

Skalierbarkeit

Lösung (Größe)

Dezentrale Ressource mit dezentralen Algorithmen mit folgenden Merkmalen

- keine Maschine besitzt vollständige Informationen über den Systemstatus
- jede Maschine trifft Entscheidungen nur auf die lokalen Daten
- der Ausfall einer Maschine beeinflusst das System nicht, bzw. nicht gravierend
- eine globale *Uhr* wird nicht vorausgesetzt.

Skalierbarkeit

Problem (Geographisch)

Synchrone Kommunikation in WANs:

- blockiert einen Dienst für andere Anforderer, bis aktuelle Anfrage abgearbeitet
- unzuverlässige Verbindungen (meinst PtP)
- lange Verzögerungen (Übertragungszeiten)
- Probleme der Größenskalierung greifen auch hier

- *Asynchrone Kommunikation / Bearbeitung*: Während der Prozess/Systemanteil auf eine Serverantwort wartet, kann dieser eine weitere Aufgabe des Benutzers erfüllen.
- *Minimierung der Kommunikation*: Die Kommunikation auf das Notwendige reduzieren und die aufwendigen Berechnungen auf die Benutzer-Maschinen vorlagern.
- *Verteilung*: Zerlegung einer Komponente in mehrere Teile und Verteilung auf das System.
- *Replikation*: Redundante Verteilung der Daten (Replikation) - dies führt zu ?

Diskussion: Welche Beispiele sind vorstellbar?

Kriterien

Sicherheit

Die Sicherheit eines Systems teilt sich in:

- Sicherheit gegen unbefugten Zugriff von **Innen**
- Sicherheit gegen unbefugten Zugriff von **Außen**
- Sicherheit gegen **Ausfall** durch Angriffe von **Innen** (erhöhter unnützer Download)
- Sicherheit gegen **Ausfall** durch Angriffe von **Außen** (DOS-Angriffe, DDOS-Angriffe)

Selbststudium

Anstöße für weiterführendes Selbststudium:

- Replikation in verteilten Systemen
- Sicherheitsaspekte in verteilten Systemen

Hardware

Anforderungen

- Skalierbarkeit
- Ausfallsicherheit
- schnelle Reaktionszeiten (Performance)
- Transparenz
- Wartbarkeit

Hardware

Aufteilung und Kommunikation*

Verteilte Systeme teilen sich grob in folgende Bereiche:

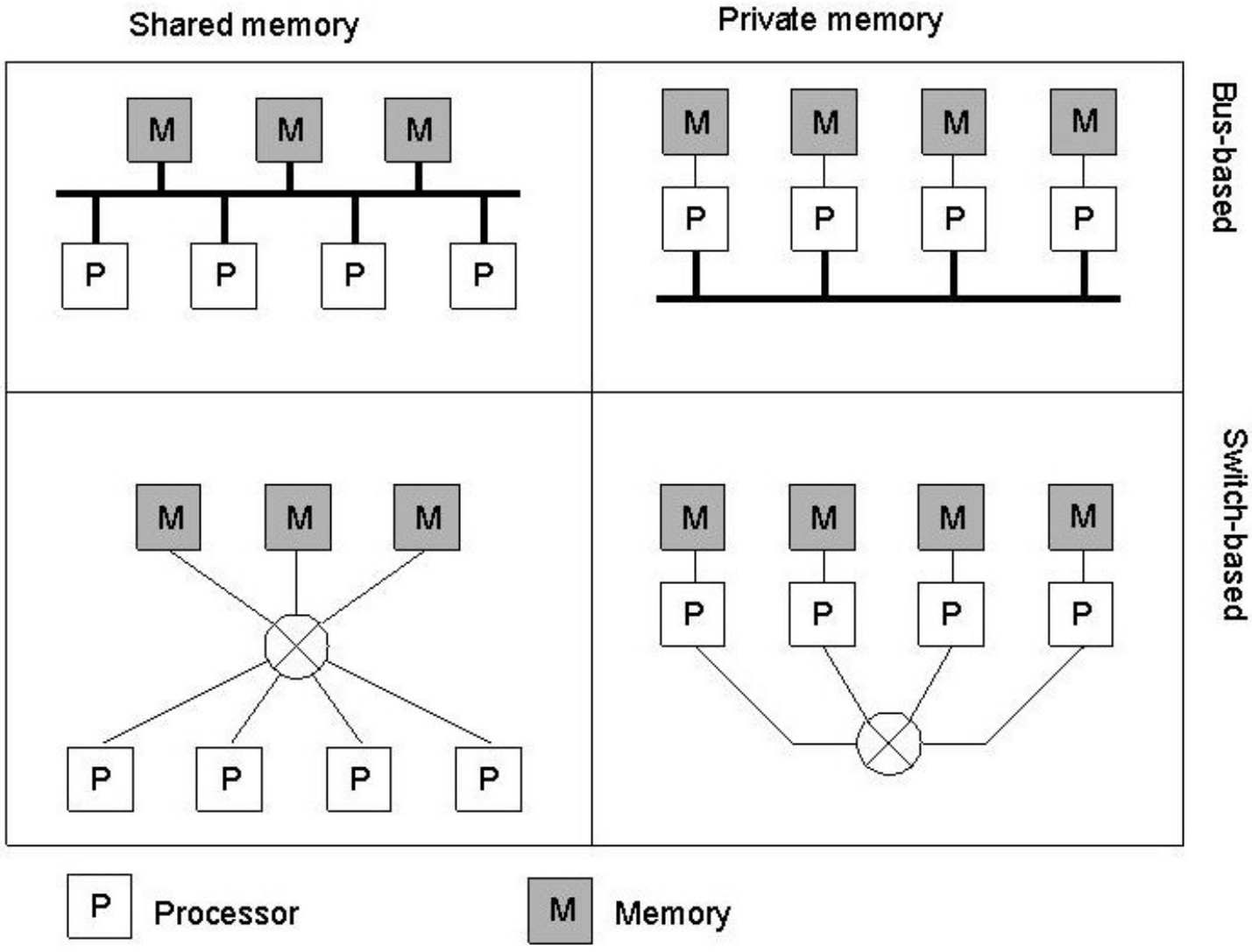
1. Multiprozess(or) (gemeinsamer Speicher)
2. Multicomputer (getrennter Speicher)
3. Multiprozessor/-computer (Mischung aus 1 und 2)

Verteilte Systeme kommunizieren über:

- Bussystem (innerhalb eines Rechners, „alte“ Coraxial-Netze, Broadcasts)
- Geschaltetes System (z.B. Switched-Networks, Sockets)

Hardware

Konzepte



Hardware

Multiprozess(or)

- alle Prozessoren/Prozesse haben (über ein Bussystem) Zugriff auf einen **gemeinsamen** Speicher bzw. IO
- Timeslots regeln die Kommunikationsreihenfolge
- Recheneinheiten werden ggf. geteilt (z.B. MIPs bei Host-Systemen)
- Caching pro Prozess(or) für höhere Geschwindigkeiten (Trefferraten von 90% und mehr können erreicht werden)

Herausforderungen:

- paralleler Zugriff auf selben Speicherbereich bei gleichzeitigem Speichern (Synchronisation)
- Skalierbarkeit auf einem System beschränkt durch Ressourcen (und BUS-Breite)

Multicomputer-Systeme unterteilen sich in:

- **Homogen**: ein einziges verbindendes Netzwerk, gleichgeartete Rechnersysteme meist zur parallelen Verarbeitung (gängig einzelnen produktiven Bereichen)
- **Heterogen**: unterschiedliche Netzwerke, verschiedene Rechnersysteme und -typen (gängig in Firmen)

Jede Komponente verfügt über einen eigenen Speicher.
Bussysteme oder Schaltnetzwerke (z.B. Fast Ethernet) verbinden die Komponenten zu einem SAN (System Area Network).
Die gebräuchlichsten Topologien für die Vernetzung sind Maschen oder Hypercubes.
Das Spektrum von geschalteten Multicomputersysteme reicht von:

- MPPs (Massively Parallel Processors): Supercomputer mit schnellen Verbindungsleitungen hohen Durchsatzes und mehreren hundert Prozessoren

bis:

- COWs (Clusters of Workstations): Standard - PCs verbunden mit normalen Kommunikationskomponenten

Heterogene Multicomputersysteme zeichnen sich aus durch:

- unterschiedliche Rechenleistung (CPU, Speicher)
- unterschiedlichste Vernetzung
- selten Systemgesamtsicht, d.h. eine Anwendung kann nicht von einer gewissen Leistung in den einzelnen Komponenten ausgehen.
- eine Softwareschicht wird für die Anwendung gebraucht - verteiltes System
- Systeme von Systemen gängig - „massiv“ verteiltes System

Bsp.: Campusübergreifende verteilte Systeme, aufbauend auf vorhandener Infrastruktur.

Systemskalierung betrifft Redundanz und Performance; techniken dazu:

- vertikale Skalierung (Multithreading/-processing)
- horizontale Skalierung (Multicomputer/-server)
- Mesh: Mischung aus horizontaler und vertikaler Skalierung
- Virtualisierung (OS Mehrfach)
- Containerisierung („Miniaturkernel“ + Anwendung)

Wichtigsten Anforderungen an die Software:

- s.o.: die Software muss die HW-Anforderungen genau so abdecken, bzw. unterstützen
- Einfachheit
- Interoperabilität
- Portierbarkeit

- Zugriff auf einen Speicher
- Mehrere Prozesse parallel
- Anzahl der Prozessoren für Programm transparent halten

Lösung der Probleme bei konkurrierenden Zugriffen über:

- Semaphoren
- Monitor
- Mutex

Software

Multicomputer Systeme*

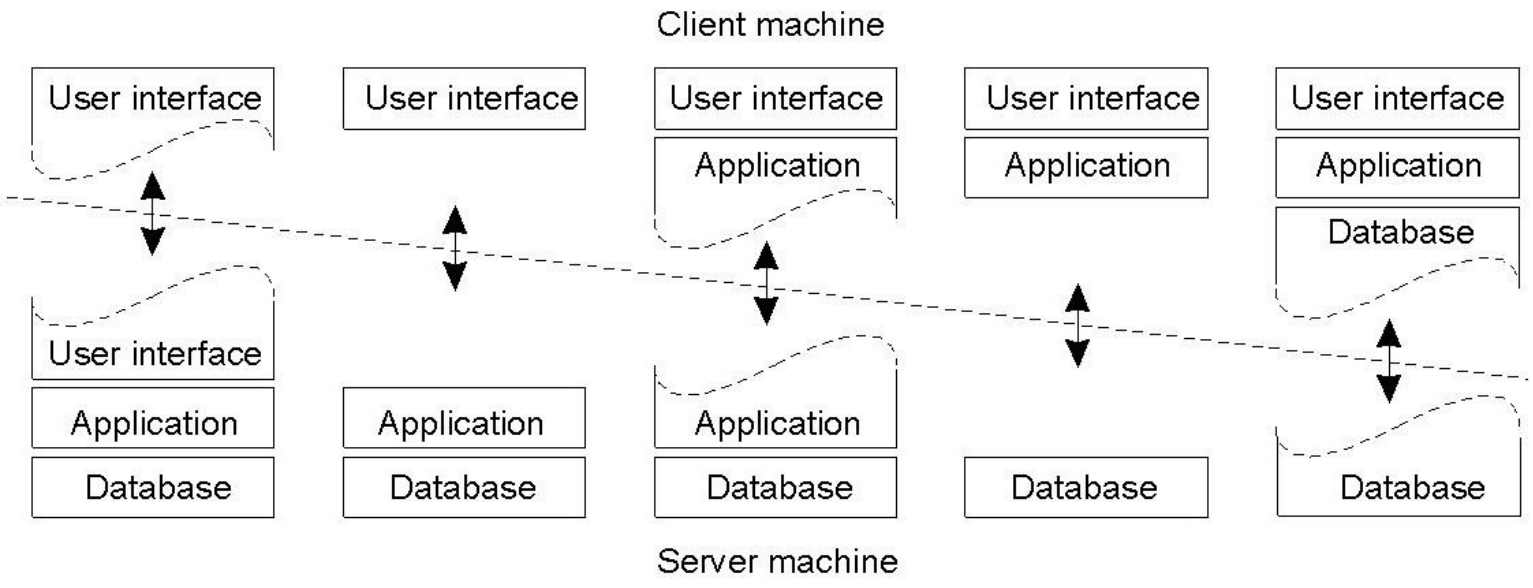
Jeder Knoten besitzt

- einen eigenen Kernel und eigene Hardware
- ein separates Modul für das Senden und Empfangen von Nachrichten an und von anderen Knoten
- eigenen Speicher, CPU, Festplatte
- logische Aufteilung der „Arbeit“ auf mehrere Maschinen

Multicomputer Betriebssysteme *können* eine *Software-Implementierung* (Konzept) eines gemeinsam genutzten Speichers bereitstellen. Ansonsten nur Funktionen zur Nachrichtenübergabe.

Software

Aufteilung der Architektur



Die Semantik der Nachrichtenübergabe bei Multicomputer-Systemen kann sich zwischen den verschiedenen Systemen wesentlich unterscheiden.
Unterschiede ergeben sich durch

- Pufferung, bei Sender, oder Empfänger, oder bei Beiden
- keine Pufferung
- Sender wird blockiert
- Empfänger wird blockiert

Funktionen und Aufgaben im wesentlichen, wie bei traditionellen Betriebssystemen (vergl. Betriebssysteme, Tanenbaum, 2001).

- streng gekoppelt
  - „Betriebssystem“ erscheint als ein System
  - verteiltes Betriebssystem, **DOS** (Distributed Operating System)
  - Verwaltung von Multiprozessorsystemen und homogenen Multicomputersystemen
- locker gekoppelt
  - „Betriebssystem“ erscheint als mehrere Betriebssystem, die miteinander arbeiten
  - Netzwerkbetriebssystem, **NOS** (Network Operating System)
  - Verwaltung von heterogen Multicomputersystemen
  - führt durch Erweiterungen zur **Middleware**

Software

Middleware

Die Middleware stellt eine **Erweiterung** eines (Netzwerk-)Betriebssystem als zusätzliche **Software-Schicht** über ein - meist heterogenes - Multicomputer-System dar Ziel: Verbergen der Heterogenität.
... mehr dazu später

Software

Zusammenfassung - Ziele

System	**Beschreibung**	**Wichtigstes Ziel**
DOS	Streng gekoppeltes Betriebssystem für Multiprozessorsystemen und Homogene Multicomputersystemen	Hardware-Ressourcen verbergen und verwalten
NOS	Locker gekoppeltes Betriebssystem für heterogen Multicomputersystemen	Anbieten lokaler Dienste für entfernte Clients
Middleware	Zusätzliche Schicht über dem NOS, die allgemeine Dienste implementiert	Verteilungstransparenz erzielen

Software

Zusammenfassung - Vergleich 1/2

Aspekt	**Verteiltes Betriebs-system**		**Netzwerk-betriebs-system**	**Middleware-basiertes-Betriebs-system**
	Multi- pro-zessoren	**Multi-computer**		
Transparenz	Sehr Hoch	Hoch	Gering	Hoch
1 OS auf allen Knoten	Ja	Ja	Nein	Nein
Anzahl Kopi-en OS	1	N	N	N
Kommunikation	1 Speicher	Nachrichten	Dienste	Modell

Software

Zusammenfassung - Vergleich 1/2

Aspekt	**Verteiltes Betriebs-system**		**Netzwerk-betriebs-system**	**Middleware-basiertes-Betriebs-system**
	Multi- pro-zessoren	**Multi-computer**		
Resourcen-verwaltung	Global, Zentral	Global, verteilt	pro Kno-ten	pro Knoten
Skalierbarkeit	Nein	Moderat	Ja	Variiert
Offenheit	Geschlossen	Geschlossen	Offen	Offen

Selbststudium

Anstöße für weiterführendes Selbststudium:

- Multiprozess(or) mit Kreuzpunktschalter
- Synchronisierungspunkte, zuverlässige Kommunikation und Pufferung
- Multicomputer Systeme: gemeinsamer Speicher und Speichermechanismen
- Netzwerkbetriebssysteme und bereitgestellte Dienste

Middleware

Allgemein

Die Middleware stellt eine **Erweiterung** eines Netzwerkbetriebssystem als zusätzliche **Software-Schicht** dar.
Aufgabe der Middleware ist das **Verbergen** der Heterogenität und die **Verteilungstransparenz** zu verbessern.
Da die Kommunikation bei Netzwerkbetriebssystemen häufig über Sockets oder das Filesystem ablaufen, stellt diese Art eine schlechte Verteilungstransparenz dar.

Middleware

Positionierung

Die Middleware wird zwischen der Anwendungsschicht und der Schicht des Netzwerkbetriebssystems platziert.
Das Netzwerkbetriebssystem stellt zu den einzelnen Ressourcen und den anderen Knoten im Verbund der Middleware einfache Zugriffspunkte bereit.
Die Middleware stellt standardisierte Schnittstellen und Dienste für die Anwendungen bereit. Es gibt zur Zeit mehrere Standards, die **nicht** miteinander kompatibel sind.

Middleware

Modelle

- **Anwendungsorientiert**: Enge Koppelung an Anwendung; erweiterte Laufzeitumgebung; Bereitstellung notweniger Dienste (Corba/JEE)
- **Kommunikationsorientiert**: Parameter werden an eine entfernte Prozedur gesendet und die Rückgabeparameter empfangen. (RPC, RMI)
- **Nachrichtenorientiert**: synchrone bzw. asynchrone Austausch von Nachrichten (z.B. JMS) „JAVA“ - Art eines „RPCs“
- **Netzwerkorientiert**: Socketverbindungen
- verteilte Objekte
- verteilte Dokumente
- Webservices/Mircoservices

Middleware

Dienste

- **Namensgebung**
- **Sitzungsverwaltung**
- Funktionen zur Speicherung **Persistenz**
- **verteilte Transaktionen**: Lese- und Schreibzugriffe werden atomar ausgeführt
- Sicherheitsfunktionen

Middleware

Offenheit

Eine komplette Offenheit in einer Middleware bedeutet eine komplette, d.h. **vollständige**, Spezifikation und Einhaltung der Selben. Zusätzliche Funktionalitäten, die mit der Zeit entstehen können nicht spezifiziert werden. Festlegung auf Protokolle und einheitliche Verweise (URLs) sind kaum möglich.

Selbststudium

Anstöße für weiterführendes Selbststudium:

- Service Oriented Architecture (SOA)
- Webservices / Microservices

!!! DARE TO PREPARE !!!

Inhaltsverzeichnis

3 Kommunikation
- Berkeley-Sockets
- MPI
- Nachrichtenorientierte persistente Kommunikation
- Streamorientierte Kommunikation
- RPC
- Entfernter Objektaufruf
- REST

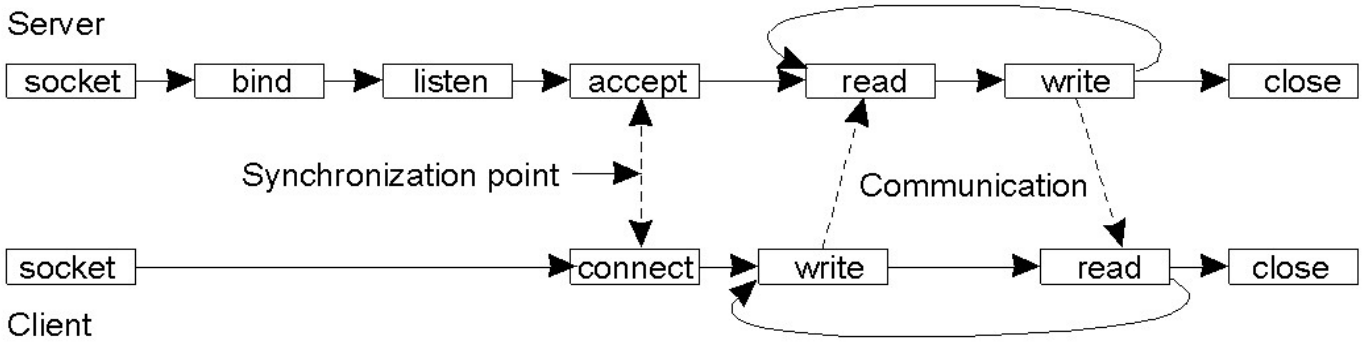
Sockets

Berkeley-Sockets*

Wichtigste und weit verbreitetste Verbindungsart ist der Berkeley-Socket oder auch **Socket** genannt. Ein Socket ist ein *eindeutig definierter Verbindungspunkt* an den eine Applikation Daten schreiben kann. Dieser kann verbindungsorientiert und verbindungslos sein. Beim Erstellen eines Sockets bedeutet dies intern, dass Ressourcen von einem OS reserviert werden, um dass Senden und Empfangen von Nachrichten für das angegebene Protokoll (z.B. TCP oder UDP) zu ermöglichen.

Sockets

Ablauf



Sockets

Elementare Funktionen der Serverseite

Funktion	**Bedeutung**
Socket	Erzeugt einen neuen Kommunikationspunkt
Bind	Ordnet einem Socket eine lokale Adresse zu
Listen	kündigt Bereitschaft an, dass Verbindungen akzeptiert werden
Accept	Blockiert den Aufrufer, bis eine Verbindungsanforderung ankommt

Sockets

Elementare Funktionen der Clientseite

Funktion	**Bedeutung**
Socket	Erzeugt einen neuen Kommunikationspunkt
Connect	Versucht aktiv eine Verbindung aufzubauen

Sockets

Elementare Funktionen beider Seiten

Funktion	**Bedeutung**
Send	sendet Daten über die Verbindung
Receive	empfängt Daten über die Verbindung
Close	Gibt Verbindung frei

Selbststudium

Anstöße für weiterführendes Selbststudium:

- verbindungslose und verbindungsorientierte Kommunikation im Vergleich
- IPC
- IPV4 vs. IPV6

MPI

Message-Parsing Interface

Sockets

- **generisch**: send, receive
- **Funktionsumfang**: gering, unflexibel
- **nicht standartisierte** Kommunikation

hocheffiziente MPI-Bibliotheken

- **Standartisierte** Nachrichtenorientierte Kommunikation
- **Funktionsumfang**: komplex, vielseitig
- **Zweck**: parallele gemeinsame Hochleistungsberechnungen

- **Kommunikation**: Prozessgruppen
- **Zuordnung Quelle-Ziel**: **gruppenID-, prozessID-** Paar
- **Synchronisation**: Sender und Empfänger werden in der Regel nicht synchronisiert
- **Laufzeitumgebung**: kümmert sich um die Kommunikation

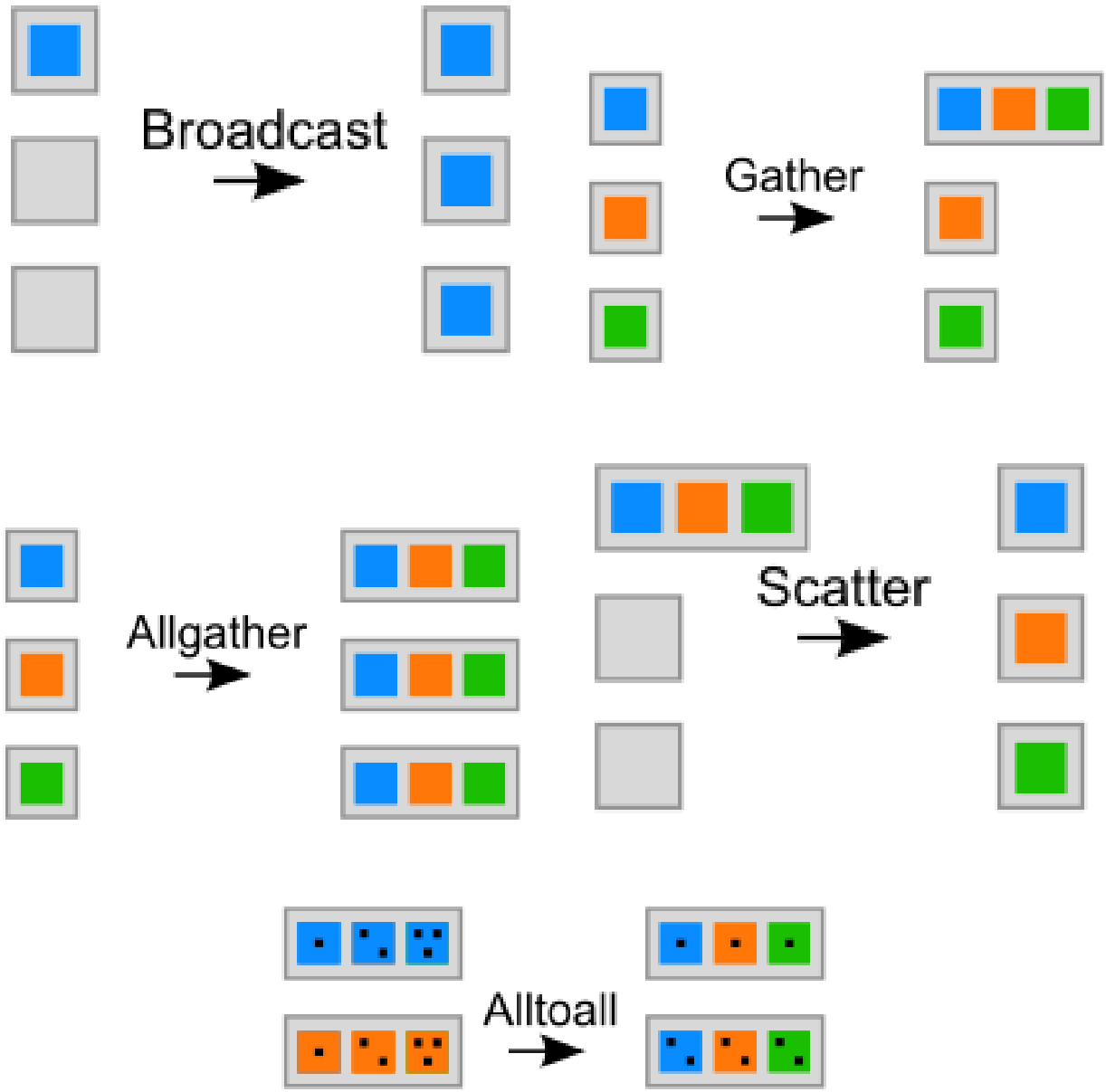
MPI

Kommunikationskonzepte*

- **Broadcast (ausstrahlen)**: MPI-Prozess schickt allen anderen Prozessen in seiner Gruppe die gleichen Daten.
- **Gather (sammeln)**: MPI-Prozess sammelt die Daten aller beteiligten Prozesse ein
- **Allgather**: jeder Prozess schickt an jeden anderen Prozess die gleichen Daten
- **Scatter (streuen)**: MPI Prozess schickt jedem beteiligten Prozess ein unterschiedliches, aber gleich großes Datenelement
- **All-to-all (Gesamtaustausch)**: analog Allgather;nur der i-te Teil des Sendebuffers wird an den i-ten Prozess geschickt

MPI

Kommunikationskonzepte



MPI

einige Elementare Funktionen 1/2

Funktion	**Bedeutung**
MPI_bsend	Ausgehende Nachricht an den lokalen Sendepuffer anfügen
MPI_send	Eine Nachricht senden und warten, bis diese an einen lokalen oder entfernten Puffer kopiert wurde
MPI_ssend	Eine Nachricht senden und warten, bis der Empfang beginnt
MPI_sendrecv	Eine Nachricht senden und auf eine Antwort warten

MPI

einige Elementare Funktionen 2/2

Funktion	**Bedeutung**
MPI_isend	Referenz auf ausgehende Nachricht übergeben und fortfahren
MPI_issend	Referenz auf ausgehende Nachricht übergeben und warten, bis der Empfang beginnt
MPI_recv	Eine Nachricht empfangen; blockieren, wenn es keine Nachricht gibt
MPI_irecv	Prüfen ob es eine eingehende Nachricht gibt, aber nicht blockieren

Selbststudium

Anstöße für weiterführendes Selbststudium:

- MPI V2
- MPI V3

Nachrichtenorientierte persistente Kommunikation

Charakteristika

Die Nachrichtenorientierte persistente Kommunikation ist charakterisiert durch:

- Nachrichtenwarteschlangensysteme (z.B. **MOM - Message-Oriented Middleware**).
- Zwischen - Speicherkapazität
- lange Nachrichtenübertragungszeiten (Minuten)
- Bsp.: Email
- Garantie für den Sender, dass Nachricht *irgendwann* ankommt.
  - keine Garantie wann
  - keine Garantie ob gelesen wird

Nachrichtenorientierte persistente Kommunikation

Koppelung

Die persistente Kommunikation ist eine *lokal gekoppelte Kommunikation*. D.h. der Empfänger muss nicht am Laufen sein, wenn der Sender sendet. Sender und Empfänger laufen unabhängig voneinander. Ein **Warteschlangenmanager** verwaltet die Warteschlangen.

Nachrichtenorientierte persistente Kommunikation

Daten

Eine Nachricht kann beliebige Daten enthalten. Wichtig ist, dass die Daten richtig adressiert sind. Hierbei an eine eindeutige Zielwarteschlange.

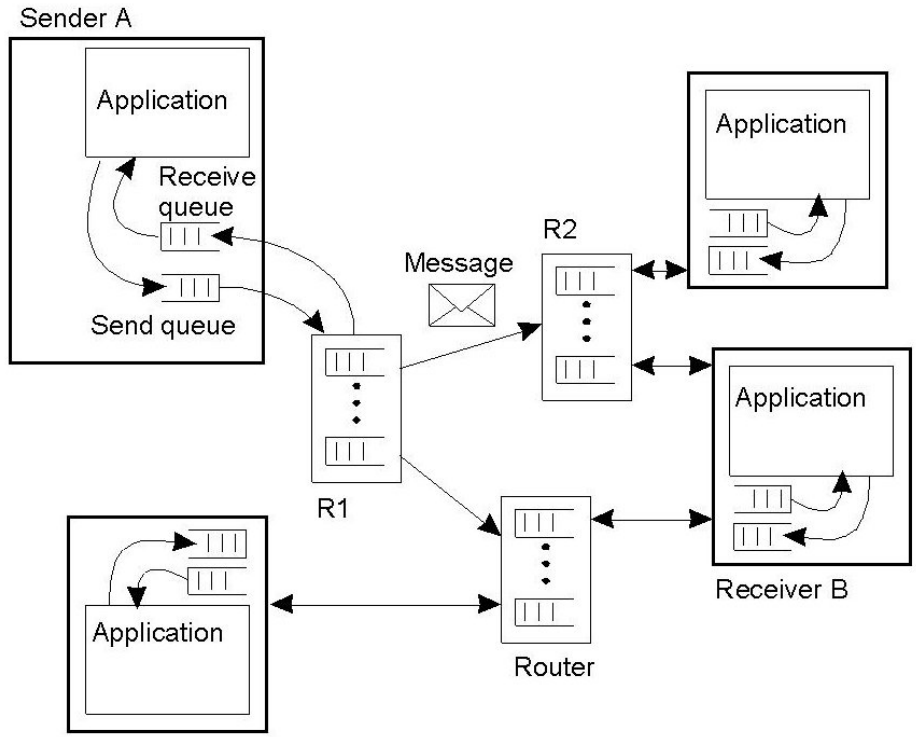
Nachrichtenorientierte persistente Kommunikation

Elementare Funktionen*

Funktion	**Bedeutung**
Put	Fügt eine Nachricht an eine Warteschlange an
Get	Blockiert, bis eine Nachricht aus der Warteschlange gelesen werden kann und entfernt diese dann
Poll	Warteschlange wird überprüft und die nächste Nachricht entfernt. KEIN Blockieren
Notify	Eine Verarbeitungsroutine inst., die aufgerufen wird, wenn eine Nachricht in die angegebene Warteschlange gestellt wird (Call-Back-Mechanismus)

Nachrichtenorientierte persistente Kommunikation

Darstellung



Streamorientierte Kommunikation

Allgemein

Problem bisher: Zeitnahe/Zeitkritische/Zeitabhängige Übertragung von Daten im verteilten System. Wie z.B. Audio-/Videostreams.
Auch bezeichnet als: **Unterstützung fortlaufender Medien**

Datenstreams können auf fortlaufende und diskrete Medien angewandt werden. Hierbei unterscheidet man:

- asynchroner Übertragungsmodus: Datenelemente hintereinander, **keine** Timingbedingung
- synchroner Übertragungsmodus: definierte **maximale** Ende-zu-Ende-Verzögerung
- isochroner Übertragungsmodus: definierte **minimale und maximale** Ende-zu-Ende-Verzögerung
- einfacher Stream: Datenelemente eines Typs hintereinander
- komplexer Stream: Datenelemente mehrerer Typen ineinandergeschachtelt.

Streamorientierte Kommunikation

QoS (**Quality of Service**)

Ein QoS definiert bei Streams die Zeitabhängigen Informationen und definiert:

- Rechtzeitigkeit
- Umfang
- Zuverlässigkeit

Anforderungen werden durch präzise **Flussspezifikationen** (Bandbreite, Übertragungsrate, Verzögerungen, usw.) definiert.

Streamorientierte Kommunikation

Synchronisation*

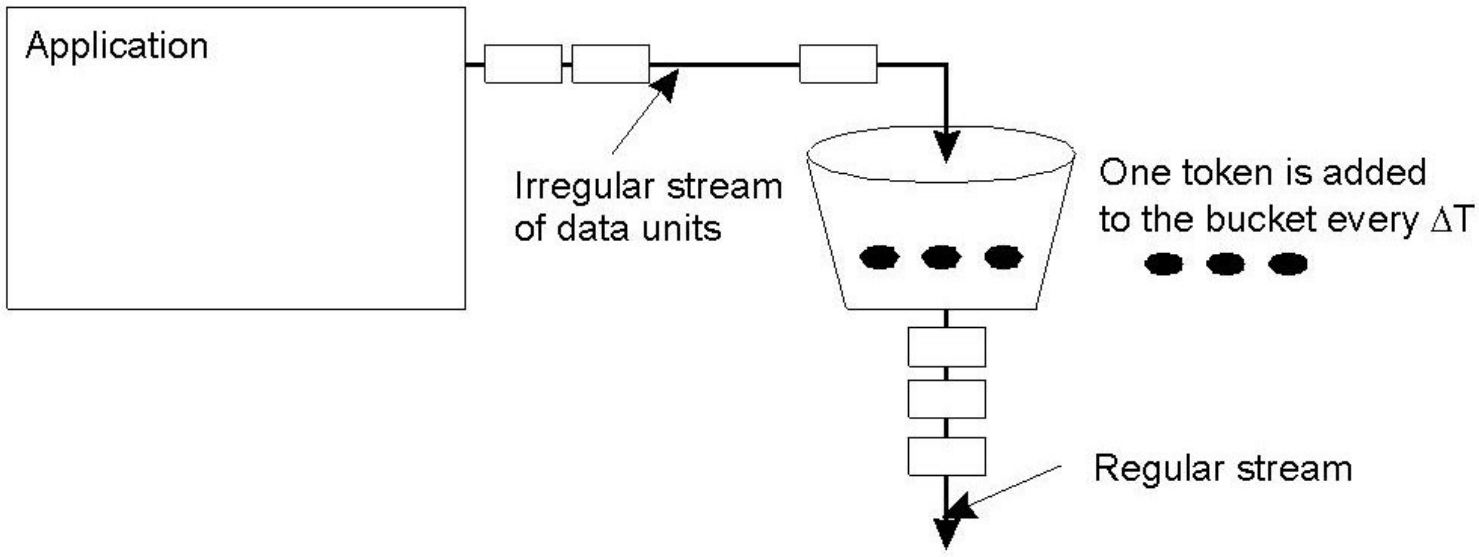
Die Synchronisation von mehreren/einem Stream wird durch eine Middleware - Schicht realisiert:

- pro Stream eine Routine, die zusammen die Streams auf Synchronität überprüfen und die Geschwindigkeit gegebenenfalls anpassen.
- Streams benötigen Zeitstempel

Streams können auf Sender oder Empfänger-Seite synchronisiert werden.

Streamorientierte Kommunikation

Synchronisation - Token Bucket



RPC - Remote Procedure Call

Konventioneller Prozeduraufruf*

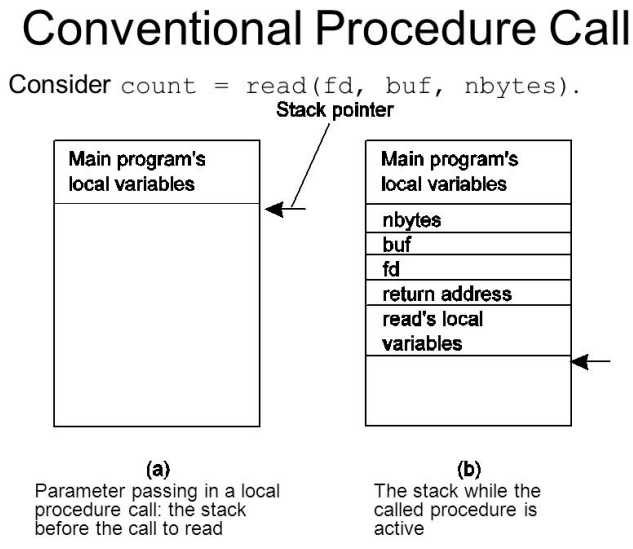
count = read(fd, buf, nbytes)

1. Aufrufer legt Variablen in Stack ab
2. Prozedur wird ausgeführt
3. Rückgabewert wird in Register abgespeichert
4. Rückkehradresse wird vom Stack entfernt
5. Aufrufer entfernt Parameter von Stack und stellt Originalzustand her

RPC - Remote Procedure Call

Konventioneller Prozeduraufruf

count = read(fd, buf, nbytes)



(a)Parameterübergabe in einem lokalen Prozeduraufruf: der Stack vor dem Aufruf von “read”
(b)Der Stack während dem Aufruf

RPC - Remote Procedure Call

Parameterübergabe

Parameter können (in C) mittels

- **call-by-value**
- **call-by-reference**

übergeben werden. Nicht in C unterstützt gibt es noch:

- **call-by-copy/restore** (ähnlicher Effekt wie **call-by-reference**)

RPC - Remote Procedure Call

Konzept

Der entfernte Prozedurenaufruf soll wie ein lokaler Aufruf aussehen.

RPC - Remote Procedure Call

Stubs

Stubs sind Prozedur-Versionen der Original-Versionen, die anstelle der eigentlichen Funktion, eine Nachricht an die entfernte Stelle mit den Parametern der Funktion übersendet und auf die Antwort wartet. Es gibt 2 Arten von Stubs:

- **Client-Stub**: Sendet die Nachricht an Server
- **Server-Stub**: Empfängt die Nachricht und wandelt diese die passende Prozedur um.

RPC - Remote Procedure Call

Ablauf*

1. Clientprozedur ruft Stub auf
2. Stub erzeugt Nachricht und ruf OS auf
3. OS sendet Nachricht an entferntes OS
4. Entferntes OS gibt Nachricht an Server-Stub weiter
5. Server-Stub packt Parameter aus und ruft Server auf
6. Server erledigt Aufgabe und gibt Ergebnis über den selben Weg zurück.

RPC - Remote Procedure Call

Parameterübergabe RPC

Durch Verpacken der Parameter in einen Stream (sog. **Marshalling**) werden die Parameter an den entfernten Punkt übergeben. Dieser Entpackt die Parameter (sog. **Unmarshaling**) dann wieder und kann damit die Prozedur ausführen.
Probleme kann es durch die verschiedenen Zeichensätze und die Byte-Order geben. Beide Seiten müssen die selben Protokolle und Darstellungsformen unterstützen.

RPC - Remote Procedure Call

Parameterübergabe RPC - Referenzen

Referenzen sind **nur** lokal gültig !
Lösung : **call-by-reference** wird durch **call-by-copy/restore** ersetzt.
Anwendbar auf Strukturen und Arrays.

RPC - Remote Procedure Call

IDL

Nach Definition der Übertragungsart müssen die Schnittstellen der Stubs definiert werden. Diese werden meist mit Hilfe einer **IDL (Interface Definition Language)** spezifiziert. Daraus wird dann der Stub kompiliert (zusammen mit entsprechenden Laufzeit- und Compilezeit-Schnittstellen).

RPC - Remote Procedure Call

erweiterte RPC Modelle

zum Nachschlagen:

- **Doors**
- asynchrone RPCs
- verzögerter synchroner RPC
- Einwege-RPC
- **DCE (Distributed Computing Environment)** RPC

Entfernter Objektaufruf

Eigenschaften Objekte

Die wichtigsten Eigenschaften von Objekten sind:

- Verbergen der Interna vor der Außenwelt
- Zugriff auf Daten nur über definierte Schnittstellen (Getter, Setter, etc.)
- Objekt - ID

Entfernter Objektaufruf

Zugriff auf entferntes Objekt nur über Schnittstelle bei sog. **Proxy** (ähnlich Client-Stub). Methodenaufrufe werden in Nachrichten verpackt und an den Server-Stub (häufig als **Skeleton** bezeichnet) gesendet. Das Ergebnis wird über Nachrichten zurückgesendet. Besonderes Merkmal:

- Status des Objektes nicht verteilt, sondern auf einer Maschine: Daher **entferntes Objekt**

Ein Objekt kann auch einen verteilten Status haben, dann spricht man von einem **allgemein verteilten Objekt**.

Entfernter Objektaufruf

Man unterscheidet im allgemeinen zwischen:

- **persistenten Objekten**: werden nach Beendigung in einen sekundären Speicher geladen und können wieder reinitialisiert werden.
- **transienten Objekten**: werden nach Beendigung vernichtet.

Entfernter Objektaufruf

Man kann sich zu einem Objekt mittels einer Referenz binden. Dies setzt folgende Informationen bei der Referenz voraus:

- Server-Adresse
- Server-Port
- Objekt - Referenz (Kennung, Speicherbereich, ID)
- Art des Objektes (meist von Server bereitgestellt)

Problem: Serverprozess stürzt ab und Referenzen werden ungültig.

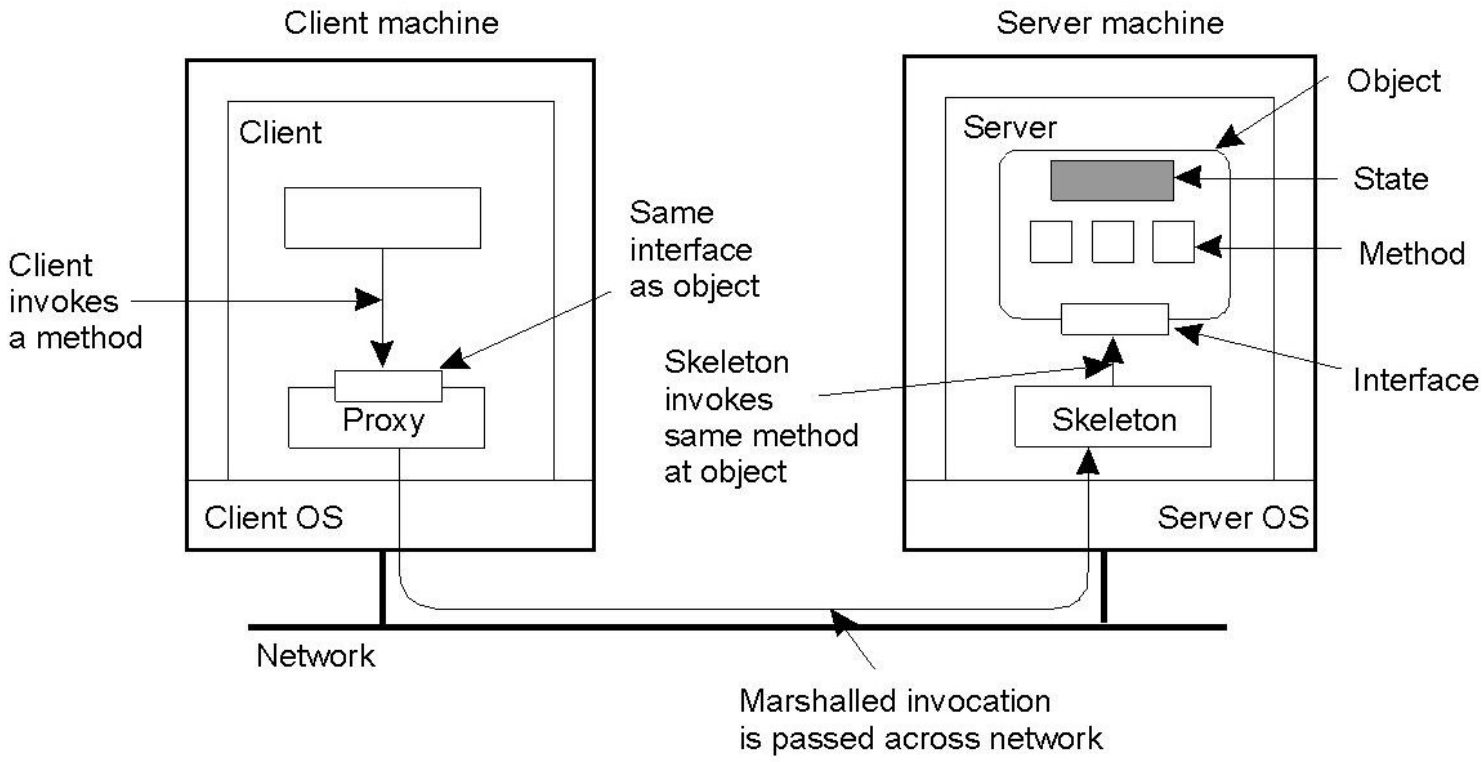
Entfernter Objektaufruf

Verteilte Objekte - Ablauf*

1. Skeleton / Objekt registieren
2. Skeleton aufrufen
3. Marshalling
4. Unmarshalling
5. Invoke
6. Return (Marshalling, Unmarshalling, etc.)

Entfernter Objektaufruf

Verteilte Objekte - Ablauf



Entfernter Objektaufruf

Methodenaufruf - RMI

Die Methoden eines Objektes werden mittels **RMI (Remote Method Invocation)** aufgerufen (ähnlich RPC). Man unterscheidet:

- statische RMI
- dynamische RMI

Entfernter Objektaufruf

statisches RMI

Die Schnittstellen müssen bei der Entwicklung der Client - Applikation bekannt sein. Die Client-Applikation muss in der Regel neu kompiliert werden, wenn sich die Schnittstelle ändert.

Entfernter Objektaufruf

dynamisches RMI

Die Applikation wählt während der Laufzeit aus, welche Methode sie aufrufen will. Ein Methodenaufruf sieht meist folgendermaßen aus:
invoke(Objekt, Methode, Eingabeparameter, Ausgabeparameter)

Entfernter Objektaufruf

Probleme

Beim verteilten Zugriff auf Objekte können verschiedene Probleme auftreten:

- gleichzeitige Zugriff
- Objekte werden während der Laufzeit zerstört
- Objektreferenzen ändern sich

Entfernter Objektaufruf

Bsp.: Java RMI

Wichtige Eigenschaften bei der Java RMI

- Proxys enthalten dieselben Schnittstellen wie lokales Objekt
- Das Klonen des entfernten Objektes erfolgt von Server aus. Der dazugehörige Proxy wird nicht geklont
- Objekt kann als Monitor konstruiert werden, um Methoden zu **synchronisieren**. Dies ist nur auf dem Client zu implementieren.
- Objekt kann serialisierbar sein, d.h. Objekttyp oder Funktionstyp kann als Parameter gesendet werden

REST

Representational State Transfer

Maschine zu Maschine Kommunikation

1. mittels HTTP(s) Requests (Fragen zu HTTP???)
2. standardisierte Resourcerequests
3. unterstützt durch breite Infrastruktur (Web- und Application-Server, HTTP-fähige Clients)
4. Basis für Webservices
5. Statusübergang durch Calls („statuslos“ im Backend)

REST

Basismethoden

1. **GET**: Abruf einer Ressource
2. **POST**: Einfügen neuer (Sub-)Ressourcen (als URI)
3. **PUT**: Anlegen einer Ressource/Ändern einer Ressource
4. **DELETE**: Löschen einer Ressource

REST

erweiterte Methoden

1. **HEAD**: Fordert Metadaten zu einer Ressource an.
2. **OPTIONS**: Prüft, welche Methoden (POST, DELETE, etc.) auf einer Ressource zur Verfügung stehen.
3. **CONNECT**: Anfrage wird durch einen TCP-Tunnel geleitet
4. **TRACE**: Gibt die Anfrage zurück, wie sie der Zielserver erhält; Debugging/Proxy-Change

REST

Parametrisierung

1. **URL**: Teile der URL sind die Request-Parameter (begrenzt)
2. **BODY**: JSON/XML werden im Request Body mitgeschickt und können verarbeitet werden
3. **REQUEST?**: analog PHP werden in der URL Request-Parameter mittels ? übergeben

1. **BODY**: JSON/XML werden im Response Body mitgeschickt und können verarbeitet werden
2. **URL**: Redirects zu neuer Ressource

!!! DARE TO PREPARE !!!

Inhaltsverzeichnis

4 Synchronisation, Koordination
- Synchronisation
- Zeit
- globaler Status
- globaler Koordinator
- wechselseitiger Ausschluss

Synchronisation

Herausforderungen*

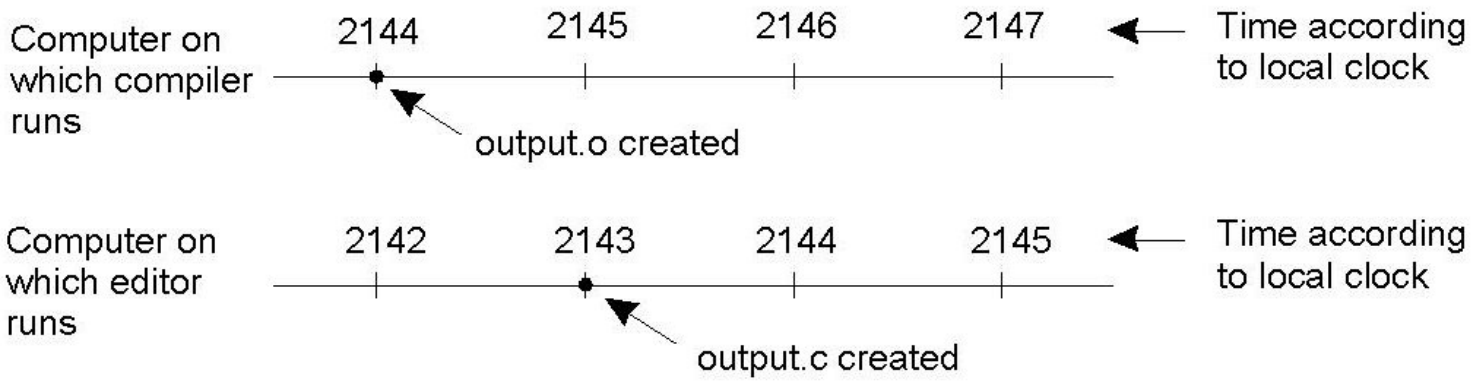
Frage der zeitlichen Reihenfolge:

- Verteiltes Kompilieren
- Logging und Fehlersuche
- Transaktionssicherheit mit Zeitstempel
- Zeitstempel und Reihenfolge (Zeitslots)

Synchronisation

Problemstellung

Welche Probleme können in einem verteilten System entstehen, wenn die Zeit über die Elemente des Systems differieren?
Bsp.: verteiltes Kompilieren



Physische Uhren sind mehr Timer als Uhren im eigentlichen Sinne, die bestehen physikalisch aus einem präzisen Quarz, der N-mal in der Sekunde schwingt. Logisch besteht er aus:

- Zählerregister
- Halteregister

Das Zählerregister wird pro Schwingung um 1 reduziert. Ist der Zähler auf 0 angekommen wird ein Interrupt geschaltet und aus dem Halteregister das Zählerregister neu geladen. Die Uhr kann somit so programmiert werden, dass diese 60 mal in der Minute Interrupts erzeugt (oder eine andere Frequenz). Ein Interrupt wird als **Uhr-Tick** bezeichnet.

physikalische Zeit

Abweichung (drift)

Es gilt für die Abweichung:

$$1 - q \leq \frac{dC}{dt} \leq 1 + q$$

mit
q: Konstante Abweichung (Ungenauigkeit) vom Hersteller vorgegeben
C: Wert der internen Software - Uhr
t: Wert der UTC-Zeit
bei der **perfekten Uhr** ist $\frac{dC}{dt} = 1$

Wenn zwei Uhren entgegengesetzt abweichen, d.h. die eine läuft langsamer und die zweite schneller, dann beträgt die Differenz untereinander bis zu

$$\Delta t$$

Wenn garantiert werden soll, dass zwei Uhren sich nie mehr als um δ unterscheiden, müssen die Uhren alle $\delta/2q$ Sekunden neu synchronisiert werden. Die Synchronisationsalgorithmen hierzu unterscheiden sich nur darin, wie diese Synchronisation erfolgt.

physikalische Zeit

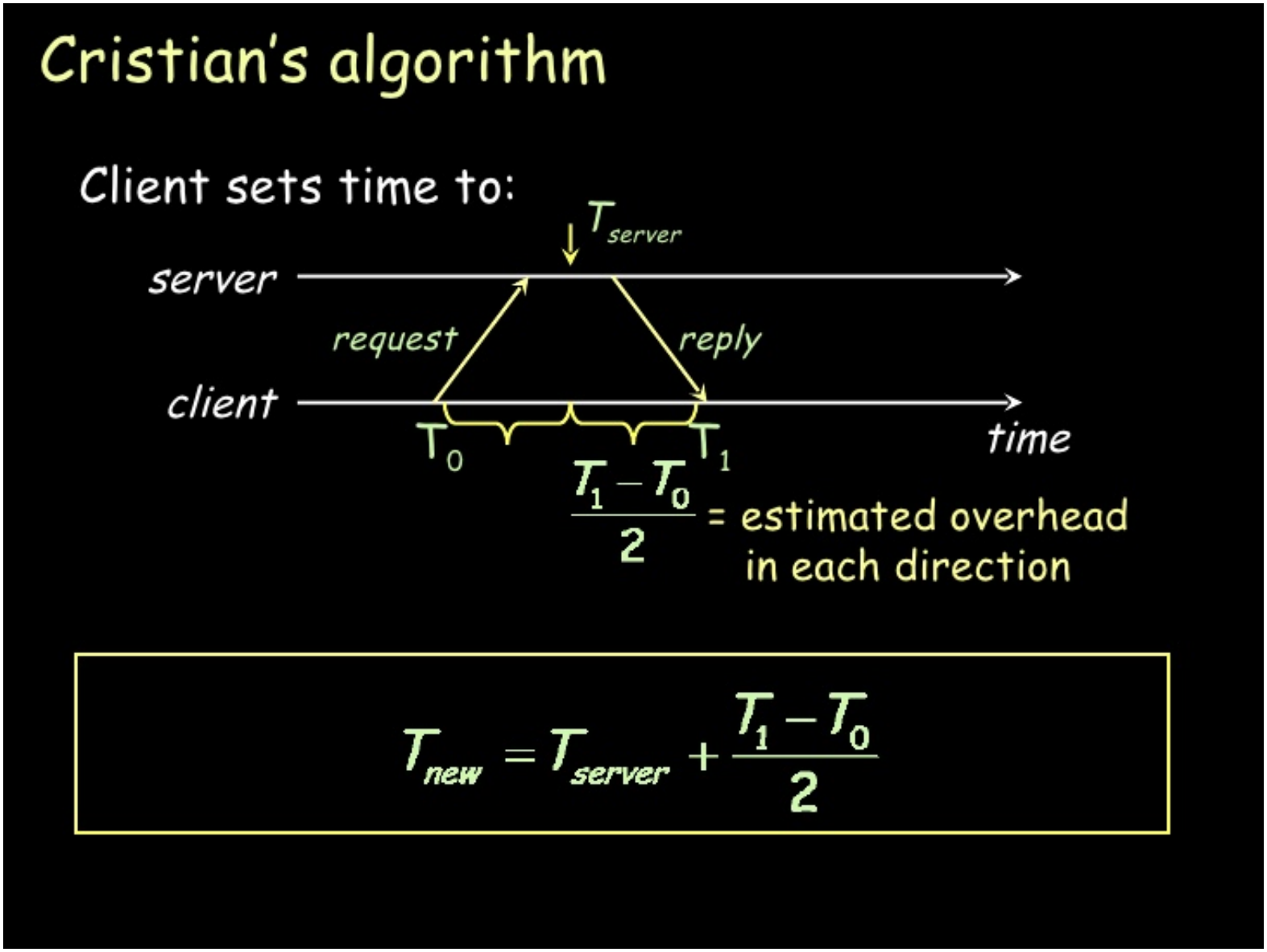
Algorithmus von Christian*

- alle $\delta/2q$ Sekunden wird die Zeit von einem Client bei einem **Zeitserver** angefragt
- falls lokale Uhr langsamer, wird die neue Zeit gesetzt
- falls lokale Uhr schneller, wird die lokale Uhr verlangsamt (z.B. addiert pro Tick nicht 10ms sondern 9 ms auf aktuellen Wert)
- Problem Übertragungszeit: Lösung hier, Zeit wird lokal gemessen

physikalische Zeit

Algorithmus von Christian

Der Algorithmus von Christian:



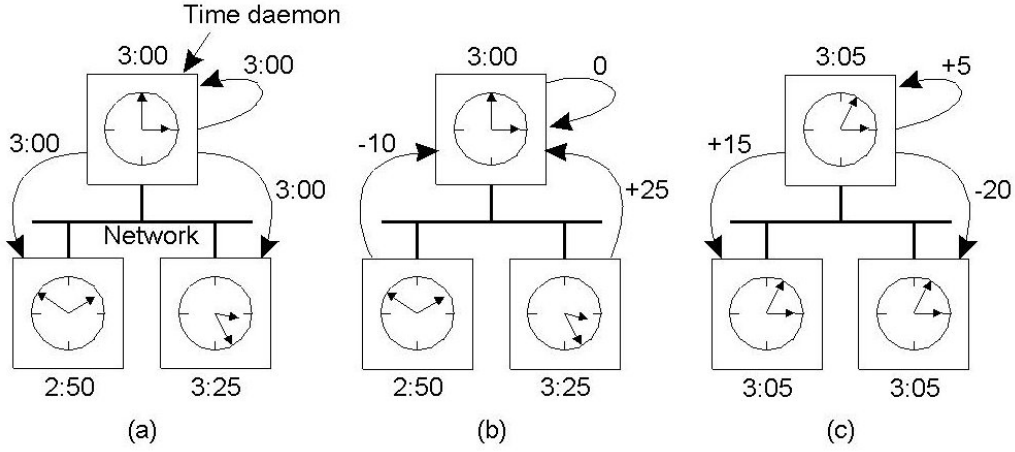
physikalische Zeit

Berkeley-Algorithmus*

- in regelmäßigen Abständen wird von Server aus die Zeit aller Maschinen abgefragt
- Server bildet die durchschnittliche Zeit
- Server sendet Anpassungsanforderungen an Clients

physikalische Zeit

Berkeley-Algorithmus



physikalische Zeit

Mittelwert-Algorithmus

- durch Broadcasts werden die Zeiten aller Maschinen ins Netz gesendet.
- jede Maschine erwartet die Antworten in einem bestimmten Intervall S und bildet den Mittelwert, auf den er sich dann anpasst.
- Extremwerte können hierbei u.U. verworfen werden.

logische Zeit

logische Uhr

In einem System kann es ausreichend sein, dass alle Maschinen sich über eine Zeit einig sind, diese aber NICHT mit der realen Zeit übereinstimmen muss. Für Algorithmen solcher Zeitanpassungen sprechen wir von **logischen Uhren**.

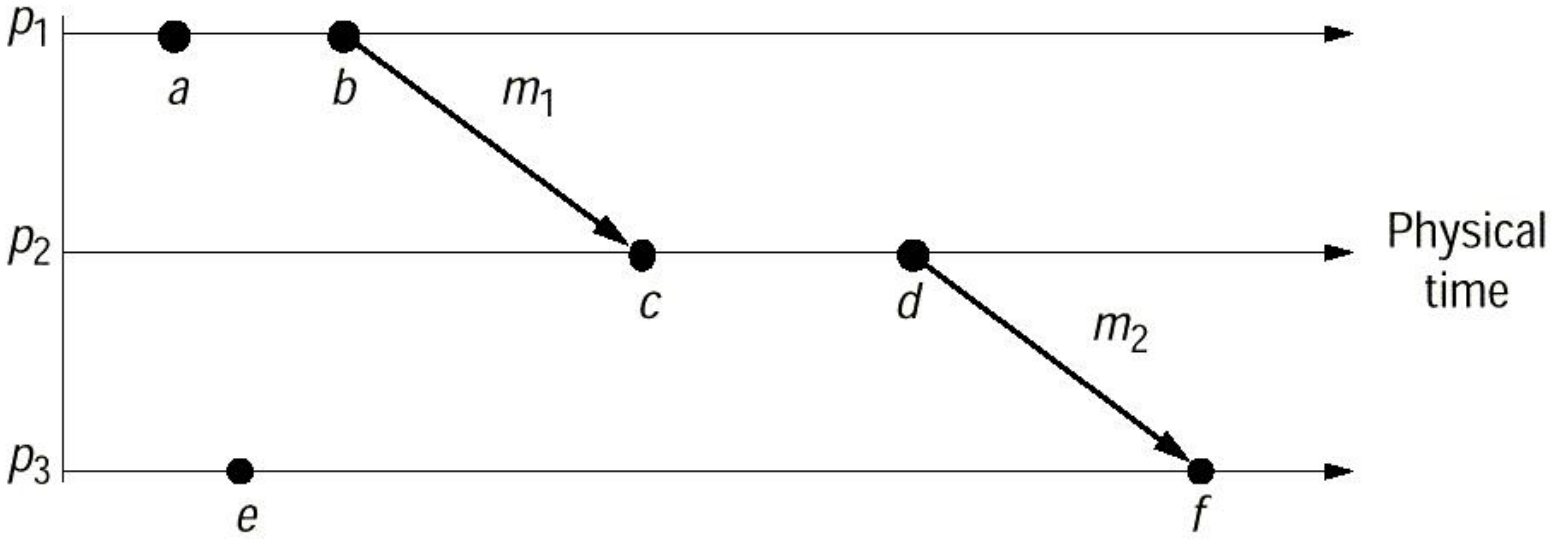
logische Zeit

Algorithmus von Lamport*

- Voraussetzung: EIN EREIGNIS PASSIERT VOR DEM ANDEREN
- Jede Nachricht wird mit dem Zeitstempel des Senders versehen
- ist die Zeit des Empfängers hinter der Zeit des Zeitstempels Z, passt der Empfänger seine Zeit auf Z + 1 an
- Oft wünschenswert, dass es keine 2 Ereignisse gibt, die innerhalb des selben Ticks passieren. Prozess - Nummer kann dezimal angefügt werden (z.B. 40.1, 40.2)

logische Zeit

Algorithmus von Lamport



Vektor - Zeitstempel können das Problem der kausalen Zeitfolge lösen.

- Voraussetzung: EIN EREIGNIS PASSIERT VOR DEM ANDEREN
- Jedes Ereignis a besitzt einen Zeitvektor $VT(a)$
- Ereignis a ist kausal vor Ereignis b wenn: $VT(a) < VT(b)$

globaler Status

Zusammensetzung

Der **globale Status** eines Systems besteht aus:

- lokaler Status der einzelnen Prozesse
- übertragene Nachrichten (gesendet, aber noch nicht ausgeliefert)

globaler Status

Snapshot*

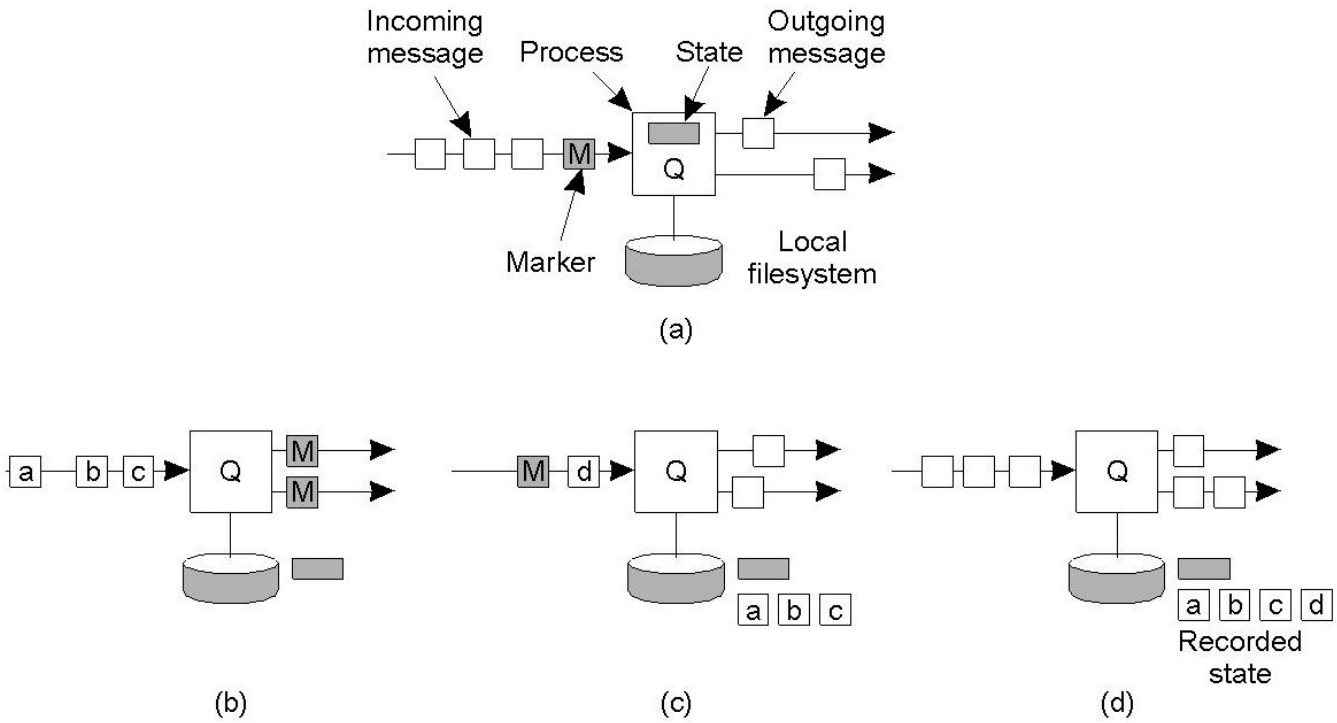
Eine unkomplizierte Methode für einen **globalen Status** ist die **verteilte Momentaufnahme**, auch **Snapshot** genannt.

- ein konsistenter Status wird reflektiert
- zu jeder gesendeten Nachricht muss im Status auch der Sender erfasst werden, um ein komplettes Bild zu geben

Man kann das Konzept eines solchen Status graphisch in einem **Schnitt** (hier konsistenter Schnitt) repräsentieren.

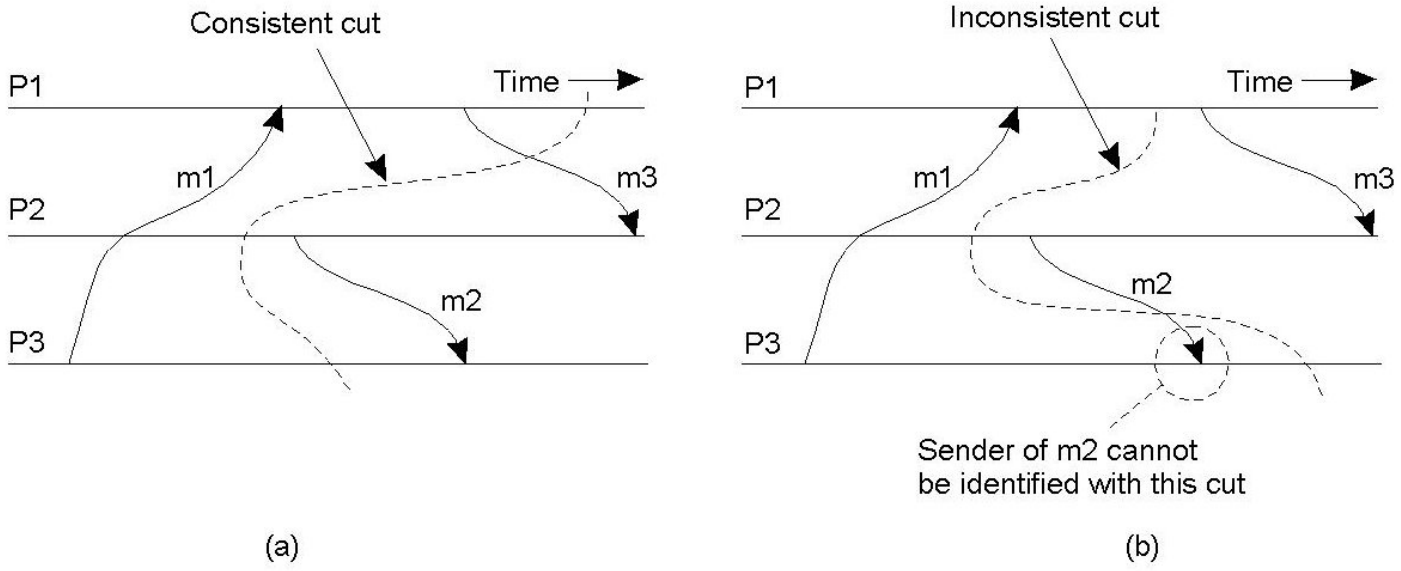
globaler Status

Snapshot



globaler Status

konsistenter Schnitt



- (a) konsistenter Schnitt
- (b) inkonsistenter Schnitt

globaler Koordinator

Allgemein

In einem verteilten System kann es sinnvoll sein einen Koordinator - Prozess zu bestimmen, der die anderen Prozesse in Zeit und Raum koordiniert. Ein solcher Prozess muss zu einem Koordinator *erwählt* werden.

globaler Koordinator

Wahl - Algorithmen

Die Algorithmen zur Bestimmung des Koordinators suchen nach dem Prozess mit der höchsten Prozessnummer. Sie unterscheiden sich in der Art der Suche.

globaler Koordinator

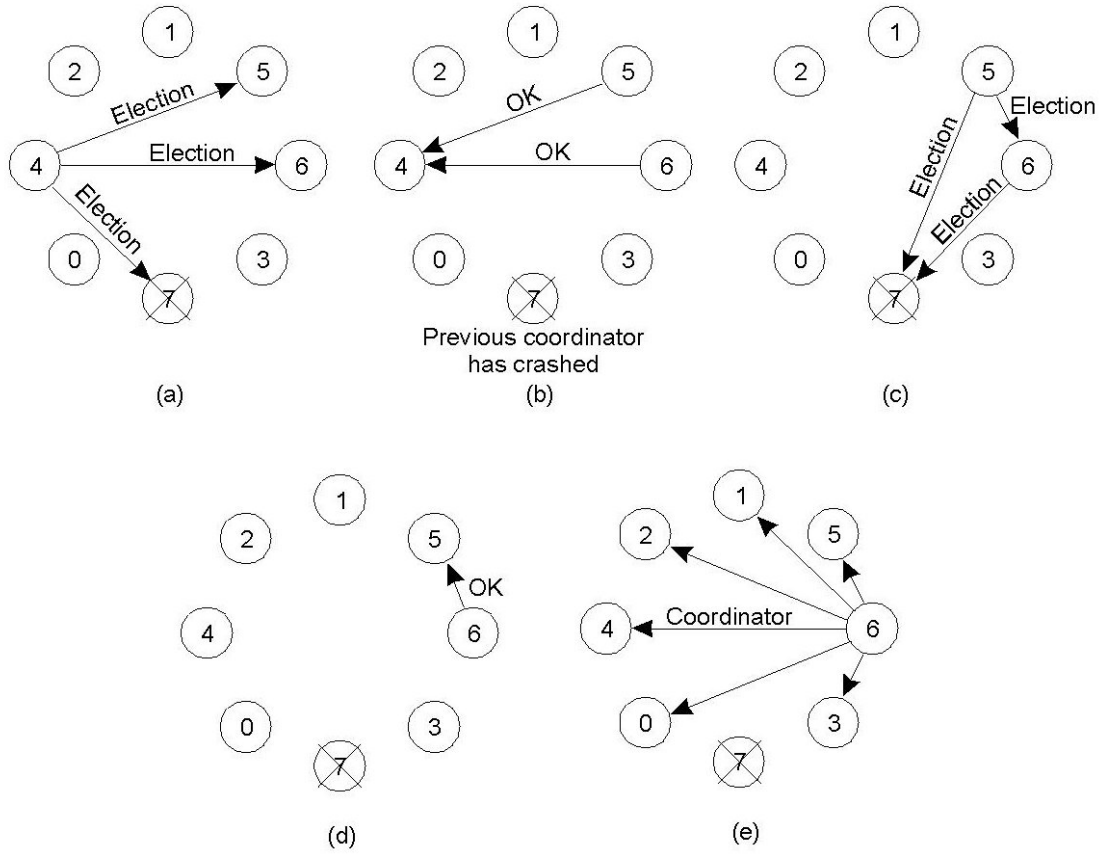
Der Bully-Algorithmus*

Der Bully-Algorithmus(Garcia-Molina, 1982):

1. Prozess P sendet eine ELECTION-Nachricht, nachdem er gemerkt hat, dass der alte Koordinator nicht mehr reagiert, an alle Prozesse mit höherer Nummer
2. Wenn kein Prozess reagiert, gewinnt P und wird Koordinator
3. Wenn ein höher gelegener Prozess antwortet, übernimmt dieser die Aufgabe des Suchens indem er ein OK an P sendet um die Übernahme zu signalisieren, P ist damit fertig
4. Erwacht ein Prozess, hält dieser eine Wahl ab
5. ein Koordinator feststeht, sendet er eine Nachricht an alle anderen Prozesse, damit dies bekannt wird

globaler Koordinator

Der Bully-Algorithmus



globaler Koordinator

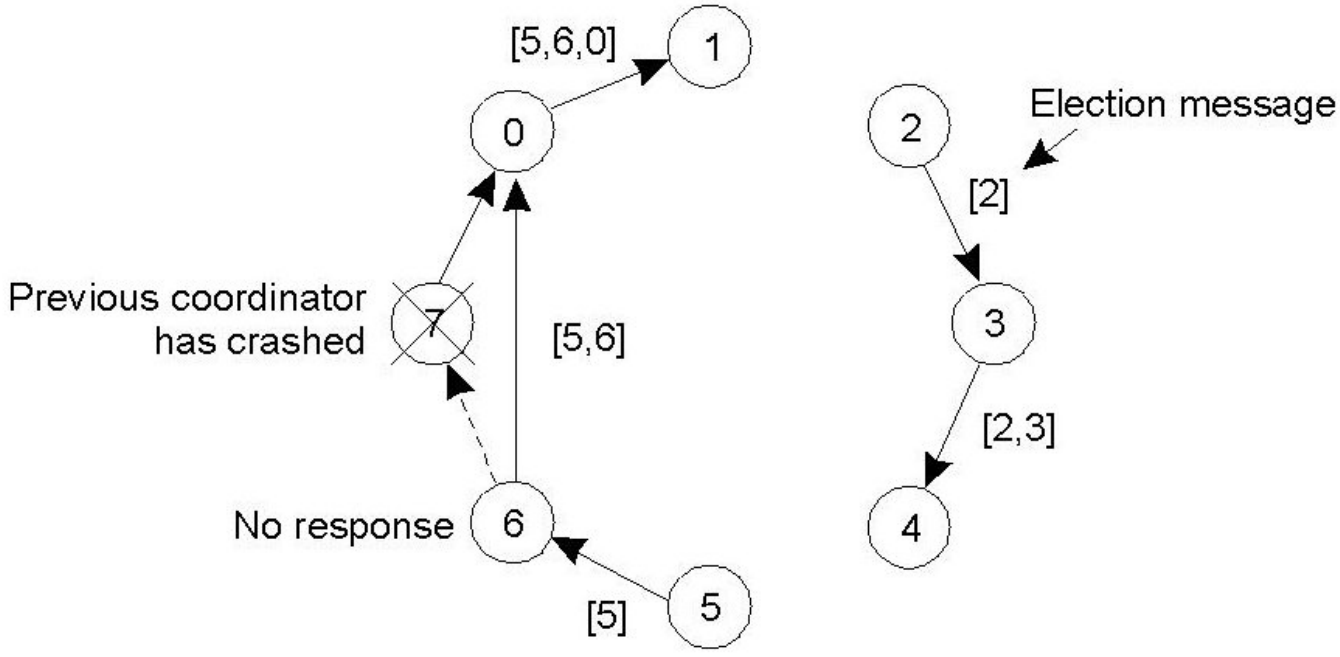
Der Ring-Algorithmus*

Der Ring-Algorithmus, Voraussetzung die Prozesse sind physisch und logisch geordnet und jeder Prozess kennt seinen Nachfolger.

1. Wenn ein Prozess erkennt, dass der Koordinator nicht mehr funktioniert, dann sendet er eine ELECTION-Nachricht an seinen Nachfolger mit seiner Prozess - ID in einer Liste
2. der nächste Prozess schreibt seine ID in die Liste, usw.
3. Sobald die Nachricht wieder am Ursprung ist, sendet der Erste die Liste in Form einer COORDINATOR-Nachricht, mit der er den Koordinator und die Gruppenmitglieder bekannt gibt

globaler Koordinator

Der Ring-Algorithmus



wechselseitiger Ausschluss

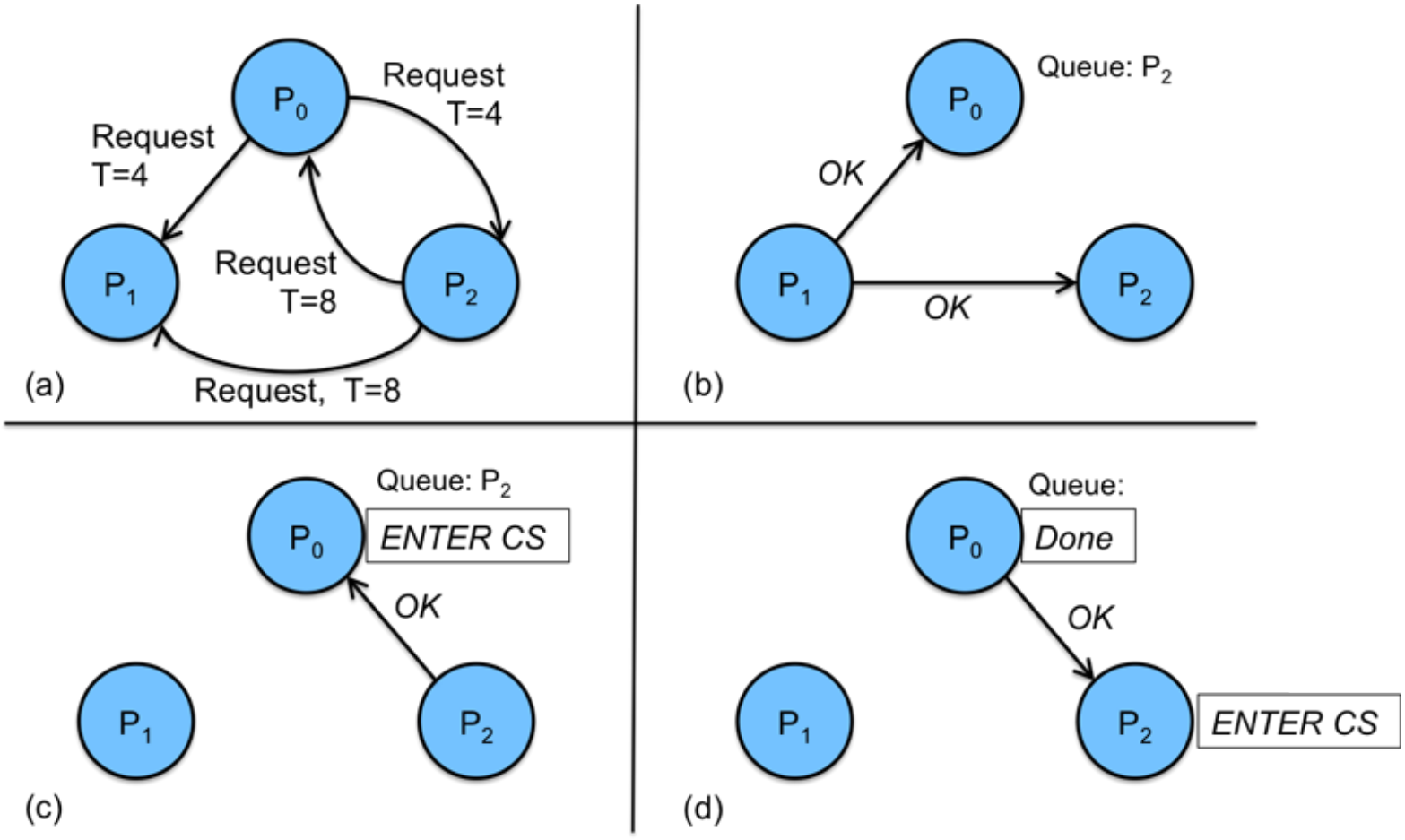
Algorithmen∗

Um auf Daten synchronisiert zuzugreifen, muss ein verteilter wechselseitiger Zugriff realisiert sein. Dabei kann es sich um folgende Typen handeln:

- **zentraler Algorithmus**: Ein Koordinator verwaltet die gemeinsamen Ressourcen und berechtigt die einzelnen Prozesse zum Zugriff
- **verteilter Algorithmus**: Alle Prozesse bestimmen, ob der anfordernde Prozess Zugriff bekommt (der Prozess mit dem geringsten Zeitstempel gewinnt)
- **Token Ring Algorithmus**: Ein Token pro Ressource (oder für alle), der umgeht, bestimmt wann ein Prozess zugreifen darf.

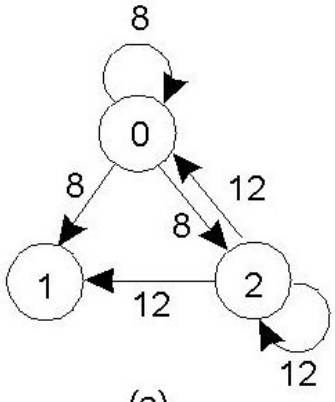
wechselseitiger Ausschluss

zentraler Algorithmus

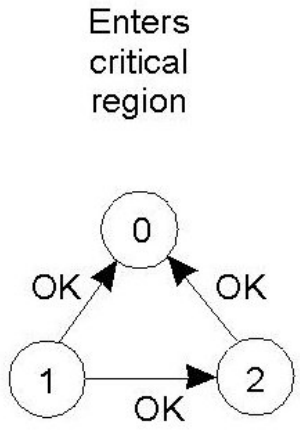


wechselseitiger Ausschluss

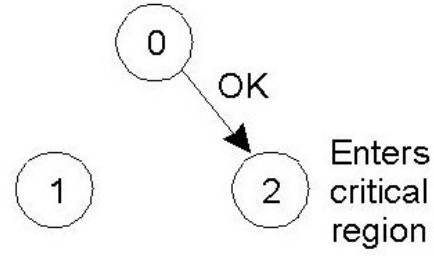
verteilter Algorithmus



(a)



(b)

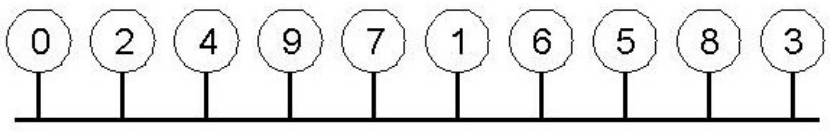


(c)

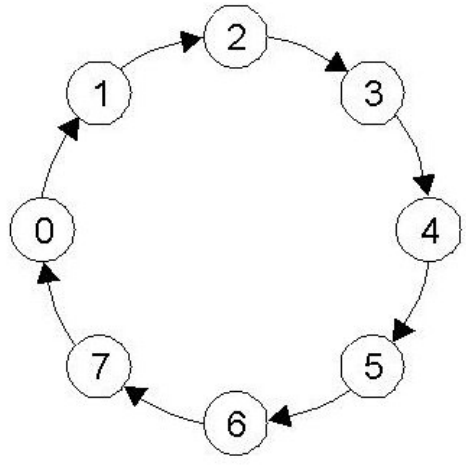
wechselseitiger Ausschluss

Token Ring Algorithmus

Token Ring Algorithmus:



(a)



(b)

wechselseitiger Ausschluss

Vergleich der Algorithmen

Algorithmus	**Nachrichten pro Austritt / Eintritt**	**Verzögerung vor dem Eintritt**	**Probleme**
Zentralisiert	3	2	Koordinator - Absturz
Verteilt	2(n-1)	2(n-1)	Absturz eines beliebigen Prozesses
Token Ring	1 bis ∞	0 bis n-1	Verlorenes Token, Prozessabsturz

!!! DARE TO PREPARE !!!

Inhaltsverzeichnis

5 Fehlertoleranz, Fehlerhandling
- Grundlegende Konzepte
- Prozesselastizität
- Zuverlässige Client-Server-Kommunikation
- Zuverlässige Gruppen-Kommunikation
- Verteiltes Commit
- Wiederherstellung

Die Fehlertoleranz ist eng an **verlässliche Systeme** gekoppelt. Die Verlässlichkeit beinhaltet unter Anderem:

- Verfügbarkeit
- Zuverlässigkeit
- Sicherheit
- Wartbarkeit

Fehlertoleranz

Fehlerklassifizierung

Fehler können klassifiziert werden in:

- **vorübergehende Fehler**: Ein Vogel stört eine Leitung und verursacht Bit-Fehler
- **periodische Fehler**: Wackelkontakt
- **permanente Fehler**: defekter Chip

Fehlertoleranz

Fehlermodelle - verschiedene Fehlertypen I/II

Fehlertyp	**Beschreibung**
Absturzfehler	Ein Server wurde unterbrochen, hat aber bis zu diesem Zeitpunkt korrekt gearbeitet
Auslassungsfehler	Ein Server reagiert nicht auf eingehende Anforderungen
Empfangsauslassung	Ein Server erhält keine eingehenden Anforderungen
Sendeauslassung	Ein Server sendet keine Nachrichten
Timingfehler	Die Antwortzeit eines Servers liegt außerhalb eines festgelegten Zeitintervalls

Fehlertoleranz

Fehlermodelle - verschiedene Fehlertypen II/II

Fehlertyp	**Beschreibung**
Antwortfehler	Die Antwortzeit eines Servers ist falsch
Wertfehler	Der Wert der Antwort ist falsch
Statusübergabefehler	Der Server weicht vom korrekten Steuerfluss ab
Zufällige Fehler	Ein Server erzeugt zu zufälligen Zeiten zufällige Antworten

Fehlertoleranz

Fehlermaskierung durch Redundanz

Fehlertoleranz kann in verteilten Systemen durch Verbergen von Fehlern vor den Prozessen erlangt werden.
$\Rightarrow$ **Fehlermaskierung**
Redundanz ist dabei ein wichtigstes Kriterium. Man unterscheidet:

- **Informationsredundanz**: z.B. Checksummen
- **zeitliche Redundanz**: z.B. Wiederholung von Transaktionen
- **physische Redundanz**: z.B. zusätzliche Hardware

Fehlertoleranz

Prozesselastizität

Bei dem Schutz vor Fehlern bei Prozessen spielen Prozessgruppen eine signifikante Rolle, welche Prozesse in Gruppen replizieren.

Innerhalb einer Prozessgruppe werden Prozesse redundant gehalten. Alle redundanten Prozesse erhalten eine Anforderung. Aus diese Weise wird ein Prozess die Arbeit übernehmen. Intern unterscheidet man:

- flache Gruppen
- hierarchische Gruppen

Die Gruppenmitgliedschaft kann dynamisch sein. Eine Gruppenverwaltung kann *zentral* (Gruppen-Server) oder *dezentral* (verteilt auf alle Mitglieder) sein. Elementare Funktionen zum Betreten und Verlassen von Gruppen sowie für die Kommunikation müssen gegeben sein.

Fehlertoleranz

Replikationsanzahl innerhalb einer Gruppe

Problem bei der Replikation innerhalb einer Gruppe ist die Frage, wieviele Kopien vorhanden sind.

- Wenn man davon ausgeht, dass k-Prozesse ausfallen können, braucht man k+1-Prozesse
- Wenn man davon ausgeht, dass k-Prozesse fehlerhafte Nachrichten senden, braucht man 2k+1-Prozesse

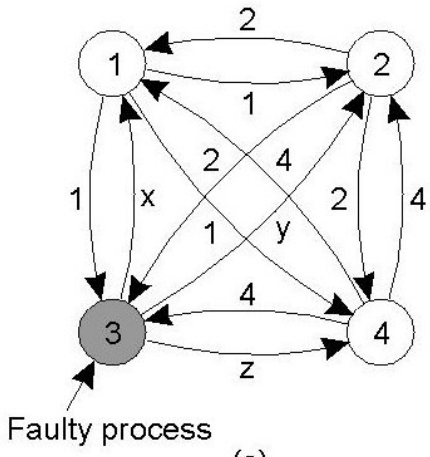
Man spricht von einer k-fachen Fehlertoleranz, wenn ein System k-Fehler kompensieren kann. Der empfangende Prozess kann der einfachen Mehrheit oder dem einzigen Antwortenden glauben.

Wenn Werte von verschiedenen Replikationionsinstanzen versendet werden, ist die Frage, welche Werte vertrauenswürdig sind. Das Problem hierbei nennt man:

das Problem der byzantinischen Generäle

Fehlertoleranz

Byzantinischen Generäle



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Fehlertoleranz

Zuverlässige Client-Server-Kommunikation: Punkt-zu-Punkt Kommunikation

In vielen verteilten Systemen wird eine zuverlässige Punkt-zu-Punkt Kommunikation verwendet. Diese basiert auf einem zuverlässigem Transportprotokoll, wie TCP. Dabei gehen Nachrichten nicht verloren.
Absturzfehler werden dabei jedoch häufig **NICHT** behandelt (maskiert).

Fehlertoleranz

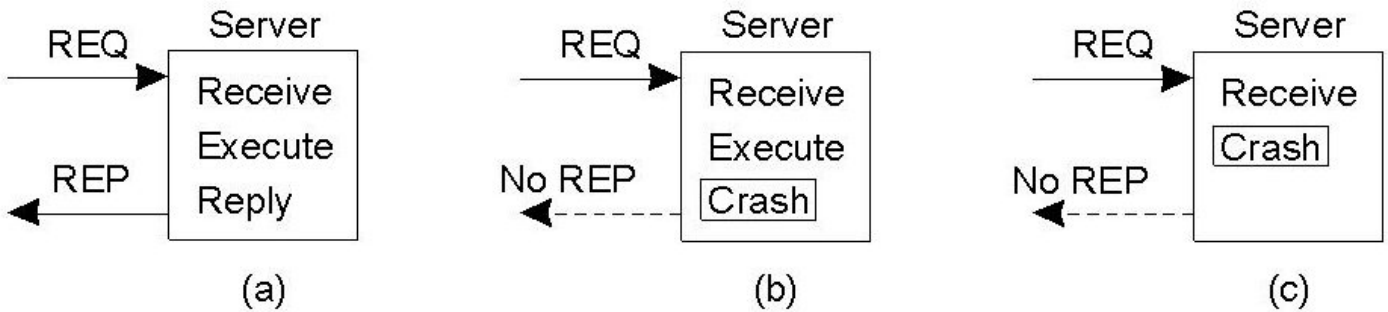
Zuverlässige Client-Server-Kommunikation: RPC/RMI und Fehler

Ziel RPC/RMI: Verbergen der Kommunikation.
Fehlerklassen die bei RPC/RMI-Kommunikation auftreten können:

1. Der Client findet den Server nicht
2. Die Anforderung vom Client an den Server geht verloren
3. Der Server stürzt ab, nachdem er eine Anforderung empfangen hat
4. Die Antwortnachricht vom Server an den Client geht verloren
5. Der Client stürzt ab, nachdem er eine Anforderung gesendet hat

Fehlertoleranz

verlorene Nachrichten



1. Der Client findet den Server nicht

- Server läuft nicht
- Adressierung stimmt nicht
- Firewall blockt
- Stub stimmt nicht mehr überein

Mögliche Lösungen:

- Exception
- Signal SIGNOSERVER zurückgeben

Problem: Transparenz geht verloren.

2. Die Anforderung vom Client an den Server geht verloren
Mögliche Lösung:

- es wird ein Timer gesetzt
- nach Ablaufen des Timers wird u.U. X-mal versucht die Nachricht zu senden, bevor der Client aufgibt.

Problem: Länge des Timers sollte nicht zu lang und nicht zu kurz sein.

3. Der Server stürzt ab, nachdem er eine Anforderung empfangen hat

Der Server stürzt ab und kann die angeforderte Funktion nicht mehr ausführen.

Dabei ist die Frage wann er eine Antwort sendet, wenn nur eine Bestätigung gefordert ist (z.B. Drucken):

- **VOR** dem Ausführen
- **NACH** dem Ausführen

3. Der Server stürzt ab, nachdem er eine Anforderung empfangen hat

Sendet er keine Antwort, so ist eine mögliche Lösung

- es wird ein Timer gesetzt
- nach Ablauf des Timers:
  - **Mindestens-einmal-Semantik**: Anforderung wird mindestens einmal bis X-mal gesendet, bevor ein Fehler erzeugt wird
  - **Höchstens-einmal-Semantik**: Anforderung wird einmal gesendet und dann sofort ein Fehler erzeugt
- Server kündigt nach Absturz und neu booten, den Absturz an alle Clients. Wie reagieren Clients???

3. Der Server stürzt ab, nachdem er eine Anforderung empfangen hat Reaktion des Clients auf die Absturzmeldung des Servers

- keine erneute Anfrage
- erneute Anfrage
- erneute Anfrage bei fehlender Antwort (Bestätigung)
- erneute Anfrage bei vorliegender Antwort (Bestätigung)

4. Die Antwortnachricht vom Server an den Client geht verloren FAZIT:
Da der Client nie wissen kann, ob der Server seine Aufgabe erfüllt hat, gibt es keine vollständig zufriedenstellende Lösungsmöglichkeit.

4. Die Antwortnachricht vom Server an den Client geht verloren Client sendet erneut die Nachricht, Server sendet die Antwort erneut.
Client sendet keine Antwort für das Empfangen (oder keine Antwort). Mögliche Lösung:

- es wird ein Timer gesetzt
- Server senden Antwort X-mal (die Verbindung kann dann u.U. als zu fehlerhaft markiert und abgebrochen werden kann)
  - Nachrichten, deren Anforderungen X-mal gesendet werden können, ohne Auswirkungen (z.B. Dateiinformationen anfordern) nennt man **idempotent**

Problem: Nicht-idempotente Anfragen (z.B. Überweisung)
Lösung: Seq.Nr

5. Der Client stürzt ab, nachdem er eine Anforderung gesendet hat Antworten (auf der Serverseite), die keiner Empfängt nennt man **Waisen (Orphan)**. Waisen können mehrere Probleme verursachen:

- Vergeuden von CPU - Zeit bei erneuten Anforderungen
- Verwirren beim Empfang des Waisen nach erneuter Anfrage
- Erzeugung von untergeordneten Waisen
- Sperrung von Ressourcen

Was ist mir Waisen zu tun ?

5. Der Client stürzt ab, nachdem er eine Anforderung gesendet hat Mögliche Lösungen nach Nelson (1981):

- Exterminierung
- Reinkarnation
- gnädige Reinkarnation
- Ablauf

Exterminierung

Ansatz

- Client Stub erzeugt Log-Eintrag
- durch Log können Waisen erkannt werden
- nach dem Booten wird der Waise explizit gelöscht

Problem

- Aufwand Datensatz in das Log zu schreiben
- Waisen können Waisen erzeugen
- Löschen könnte aufgrund partitionierter Netze nicht möglich sein

Reinkarnation

Ansatz

- Zeit wird in sequentielle Abschnitte unterteilt
- Nach Neu-Booten wird ein neuer Zeitabschnitt durch einen Broadcast vom Client angekündigt
- Dabei werden alle Waisen dieses Clients gelöscht
- Trifft ein Waise dennoch ein, hat er einen falschen Zeitabschnitt und kann gelöscht/ignoriert werden

Problem

- Broadcast können über Routergrenzen geblockt werden

gnädige Reinkarnation

Ansatz

- Wenn Zeitabschnittsbroadcast eintrifft, sucht jede Maschine den Eigentümer der entfernten Berechnung.
- Wenn der Eigentümer nicht gefunden werden kann, wird die Berechnung gelöscht

Problem

- Broadcast können über Routergrenzen geblockt werden

Ablauf

Ansatz

- jeder RPC erhält eine Standardzeit T
- Kann der RPC diese Zeit nicht einhalten, muss eine Verlängerung beantragt werden
- erhält dieser diese nicht wird der RPC angebrochen

Problem

- T

Fehlertoleranz

Zuverlässige Gruppen-Kommunikation: Grundlegendes Schema für zuverlässiges Multicasting

Ansatz

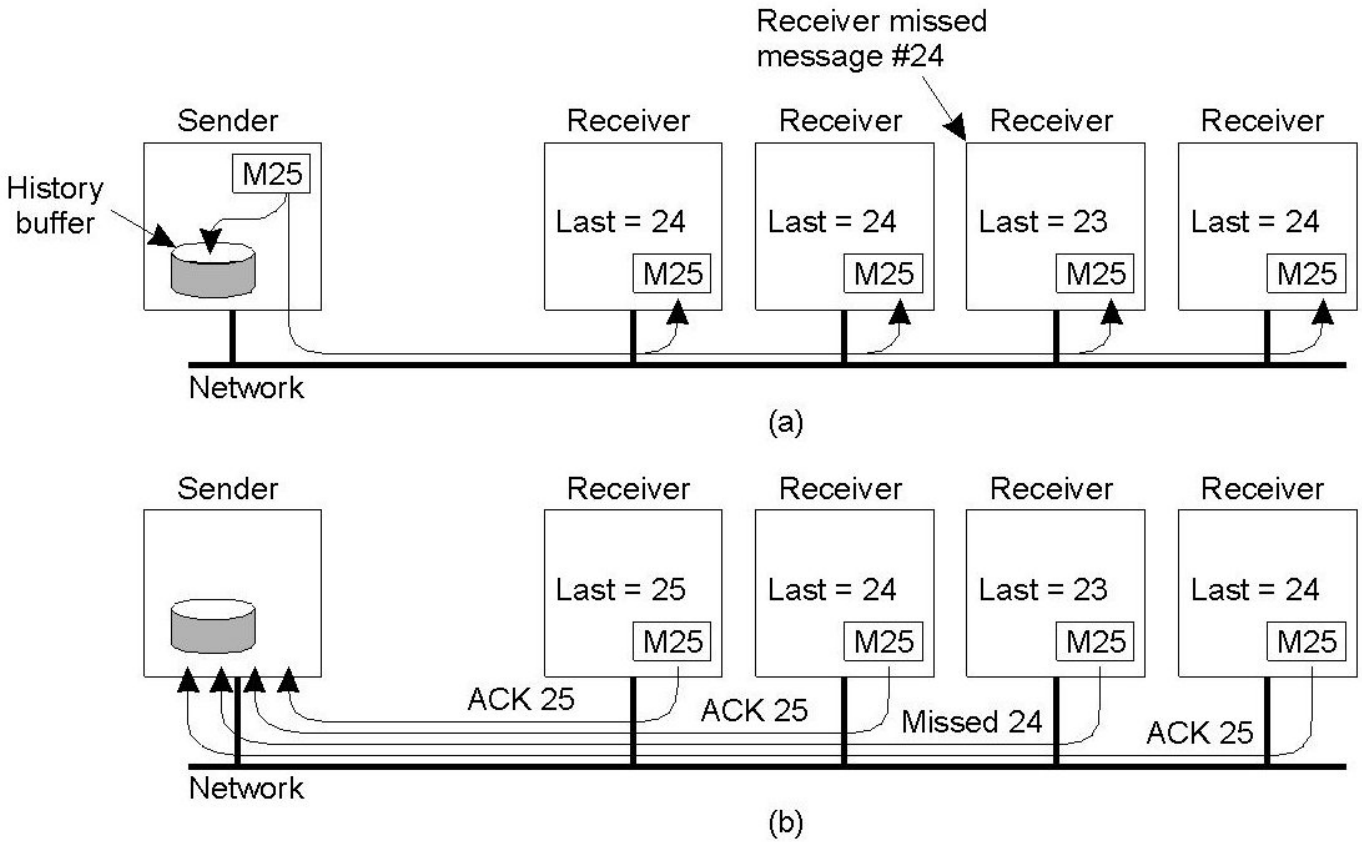
- alle Teilnehmer empfangen die sendeten Nachrichten (in sortierter Reihenfolge)
- Die Teilnehmer senden ein „ACK Seq.Nr“ oder ein „NACK Seq.Nr“
- Verlorene Nachricht wird erneut an den Teilnehmer gesendet

Problem

- Sender muss auf N Antworten warten
- Weitere Teilnehmer treten der Gruppe bei/verlassen die Gruppe

Fehlertoleranz

Zuverlässige Gruppen-Kommunikation: Grundlegendes Schema für zuverlässiges Multicasting



Fehlertoleranz

Zuverlässige Gruppen-Kommunikation: Skalierbarkeit zuverlässigem Multicasting

Ansatz

- Empfänger senden nur Feedback-Information, wenn diese einen Nachricht nicht empfangen haben

Problem

- Nachrichten müssen theoretisch ewig im Verlaufspuffer des Senders gehalten werden
- Feedback - Explosion wird nicht zwingend unterdrückt

Nicht hierarchische Feedbacksteuerung
Reduktion der Feedbacks durch **Feedback-Unterdrückung** nach Schema des **SRM (Scaleable Reliable Multicasting)-Protokolls**.

Ansatz

- Empfänger senden nur Feedback-Information, wenn dieser eine Nachricht nicht empfangen hat per Multicasting
- Wenn ein zweiter Empfänger diese Nachricht auch nicht empfangen hat, sendet er kein Feedback
- Antwort wird als Multicasting in das Netz übertragen
- Feedback-Nachricht wird mit *zufälliger* Verzögerung gesendet

Nicht hierarchische Feedbacksteuerung

Problem

- Feedback-Nachrichten können mehrfach gesendet werden
- Feedback-Nachrichten blockieren die Empfänger, die die Nachricht beim ersten Mal empfangen haben unnötig (Lösung: Multicasting-Gruppen für Feedbacks)

Hierarchische Feedbacksteuerung (bei großen Gruppen)

Ansatz

- Gruppe der Empfänger ist in mehrere Untergruppen unterteilt und wie ein Baum angeordnet
- die Untergruppe des Senders bildet die Wurzel des Baums
- Innerhalb einer Untergruppe kann ein beliebiges Multicasting-Schema verwendet werden
- jede Untergruppe bestimmt einen lokalen Koordinator (Verarbeitung Feedback-Anfragen)

Hierarchische Feedbacksteuerung (bei großen Gruppen)

Problem

- Erzeugung des Baums, u.U. dynamisch

Ein Atomarer Multicast ist z.B. bei der Aktualisierung verteilter Replikationen erforderlich. Eine Garantie, ob alle, oder kein Prozess die Nachricht erhalten hat, ist ausschlaggebend.

Ansatz

- eine Nachricht m kann eindeutig einer Liste von Prozessen zugeordnet werden
- die Liste von Prozessen ist abhängig von der *Gruppensicht*
- Eine *Änderung der Sicht* wird sofort publiziert
- Auslieferung darf nur dann fehlschlagen, wenn der Sender von m abstürzt und dadurch eine Änderung der Sicht stattfindet

Man spricht von einer **virtuellen Synchronität**

Bei der virtuellen Synchronität unterscheidet man zusätzlich:

- nicht sortierte Multicasts
- FIFO-sortierte Multicasts
- kausal sortierte Multicasts
- vollständig sortierte Multicasts

Die Sortierung bestimmt die Reihenfolge und den Auslieferungszeitpunkt.

Beim Festschreiben von Daten in eine Datenbank wird bei einem verteilten System das

2-Phasen - Commit
oder
3-Phasen - Commit

verwendet (siehe DB-Vorlesung).

Grundlegend für die Fehlertoleranz ist die Wiederherstellung nach einem Fehler. Konzept der Wiederherstellung ist es, im Fehlerfall einen *fehlerhaften Status* durch einen *fehlerfreien Status* zu ersetzen. Im wesentlichen gibt es zwei Formen:

- **Rückwärts-Wiederherstellung (Backward Recovery)**
- **Vorwärts-Wiederherstellung (Forward Recovery)**

Fehlertoleranz

Wiederherstellung: Stabiler Speicher

Bei der **Rückwärts-Wiederherstellung (Backward Recovery)** ist ein stabiler Speicher unabdingbar, welcher die dazu notwendigen Daten gespeichert hält. Diese Stabilität erreicht man z.B. durch RAID-Systeme.

Damit nicht die Initialdaten bei einer **Rückwärts-Wiederherstellung (Backward Recovery)** eingespielt werden, müssen in regelmäßigen Abständen den globalen (**verteilte Momentaufnahme**) und lokalen Status abzusichern, um zu diesem Punkt wiederherstellen zu können.
Wenn die Kombination der lokalen Wiederherstellung von *unabhängigen Prüfpunkten* (Prüfpunkte, die jeder Prozess für sich bestimmend legt) keine verteilte Momentaufnahme bildet, muss der davorliegende Status wiederhergestellt werden, usw. Dies kann dann zu einem so genannten **Domino-Effekt** führen, der spätestens am Systemstart endet.
Gegen den Domino-Effekt können koordinierte Prüfpunkte helfen.

Koordinierte Prüfpunkte werden benutzt um die Prozesse synchron zu veranlassen, ihren Status zu speichern.

Lösung

- 2-Phasen-Protokoll
  - Koordinator senden eine CHECKPOINT_REQUEST-Nachricht an alle Prozesse
  - Prozess speichert seinen Status und sendet ein CHECKPOINT_DONE

Verbesserung

- Prüfpunktanforderung per Multicast an Prozesse senden, die von der Wiederherstellung des Koordinator abhängig sind

Fehlertoleranz

Wiederherstellung: Koordinierte Prüfpunkte

vom Koordinator anhängig

- Nachricht vom Koordinator, die direkt oder indirekt mit einer Nachricht zusammenhängt, die der Koordinator seit der letzten Aufzeichnung eines Prüfpunktes gesendet hat.
- Konzept einer **inkrementellen Momentaufnahme**

Die inkrementellen Momentaufnahme besteht aus folgenden Schritten:

- der Koordinator sendet nur an die Prozesse eine Prüfpunktanforderung (hier noch **kein** Schreiben), denen er seit der letzten Prüfpunktanforderung eine Nachricht gesendet hat
- der Prozess P empfängt die Anforderung und gibt diese auf gleiche Weise weiter
- usw.
- wenn alle Prozesse benachrichtigt sind, erfolgt ein Multicast, der das **Schreiben** veranlasst.

Da die Aufzeichnung von Prüfpunkten eine kostspielige Operation ist, wurden Techniken entwickelt, diese zu minimieren. Ein Ansatz ist:

Nachrichtenprotokollierung

Grundlegende Idee bei der Nachrichtenprotokollierung

- Wenn Nachrichten *wiederholt* werden können, so kann damit ein globaler konsistenten Status erreicht werden.

Fehlertoleranz

Wiederherstellung: Nachrichtenprotokollierung - Ablauf

- Ein ausgezeichneter Prüfpunkt wird als Ausgangsstatus definiert
- alle Nachrichten seit diesem Zeitpunkt werden erneut übertragen und bearbeitet

Voraussetzung: abschnittsweise deterministisches Modell, die Ausführung aller Prozesse erfolgt in einer Folge von Intervallen, in denen das Ereignis stattfindet. Diese Ereignisse sind dieselben wie im Kontext der „Passiert-vor-Relation“.

Fehlertoleranz

Wiederherstellung: Nachrichtenprotokollierung - Charakteristika

Charakteristika nach *Alvisi und Marzullo* (1998), jede Nachricht besitzt einen Header mit

- Senderinformationen
- Empfängerinformationen
- Sequenznummer (um Duplikate zu vermeiden)
- Auslieferungsnummer (um den Zeitpunkt des Auslieferns zu entscheiden)

Stabile Nachrichten (diejenigen, die in einen festen Speicher geschrieben wurden) können für die Wiederherstellung durch Wiederholung verwendet werden.

Fehlertoleranz

Wiederherstellung: Nachrichtenprotokollierung - Kopien und Abhängigkeiten

Für jede gesendete Nachricht m gilt:

- Es gibt eine Menge DEP(m), welche all die Prozesse beinhaltet, welche von der Auslieferung der Nachricht anhängig sind.
- Wenn eine Nachricht m' von m abhängig ist, und ein Prozess Q von m' abhängig ist, dann ist Q Teil von DEP(m)
- Es gibt eine Menge COPY(m), welche all die Prozesse beinhaltet, welche eine Kopie von m haben, diese aber noch nicht festgeschrieben haben.

Nur wenn alle Prozesse in COPY(m) ausfallen, kann m nicht mehr Wiederholt werden, ohne auf eine eventuell festgeschriebene Kopie zurückzugreifen.

Fehlertoleranz

Wiederherstellung: Nachrichtenprotokollierung - Protokollierung

Bei der Protokollierung unterscheidet man:

pessimistische Protokollierungsprotokolle

Es darf höchstens ein Prozess geben, der von einer nicht stabilen Nachricht m abhängig ist. D.h. höchstens eine nicht stabile Nachricht m darf an einen Prozess ausgeliefert werden. Nach Empfang muss diese erst festgeschrieben werden, bevor der Prozess senden darf.

optimistische Protokollierungsprotokolle

Alle abgestürzten Prozesse werden in einen Status überführt, in dem diese nicht mehr von der Nachricht m abhängig sind.

!!! DARE TO PREPARE !!!

Inhaltsverzeichnis

6 Verteilte Daten
- verteilte Transaktionen
- Verteilte Datenhaltung
- Replikation
- Datenzentrierte Konsistenzmodelle
- Clientzentrierte Konsistenzmodelle
- Verteilungsprotokolle
- Konsistenzprotokolle
- Raft

Transaktionen müssen immer dem **ACID** - Paradigma genügen. Hierzu müssen elementare Funktionen gegeben sein, die den Ablauf der Transaktion beschreiben. Bsp:

1. BEGIN_TRANSACTION: Start wird markiert
2. END_TRANSACTION: abschließen und Daten fest schreiben
3. ABORT_TRANSACTION: abbrechen und alten Wert wiederherstellen
4. READ: Daten lesen
5. WRITE: Daten schreiben

Flache Transaktionen genügen dem ACID - Paradigma. Problem hierbei ist, dass es nicht möglich ist, Teilergebnisse festzuschreiben (atomar). Bei großen Transaktionen kann es lange dauern, bis ein Ergebnis zurückgeliefert wird. Dabei kann das System stark verlangsamt werden, was bei einem Absturz zwischenzeitlich nicht akzeptabel ist.

Verschachtelte Transaktionen unterteilen eine große Transaktion in mehrere logische Teile und können diese gleichzeitig (auf mehreren Maschinen) bearbeiten.
Wird eine Transaktion abgebrochen müssen alle untergeordneten Transaktionen ebenfalls abgebrochen werden, bzw. alle Transaktionen in einem Transaktionsverbund. Bei einer Anfrage bekommt eine Transaktion, oder untergeordnete Transaktion alle benötigten Daten als Kopie. Erfolgt ein Abbruch werden diese eigenen Daten einfach gelöscht.

verteilte Transaktionen

Problemstellungen

Verteilte Transaktionen unterscheidet sich von einer verschachtelten Transaktion dadurch, dass die eine verteilte Transaktion logisch eine flache, unteilbare Transaktion ist, die auf verteilten Daten arbeiten. Probleme hierbei sind:

- Verteilungsalgorithmus der Daten
- verteilte Datensperrung
- verteilte Daten festschreiben

Bei der Implementierung werden im Allgemeinen 2 Methoden verwendet:

- Privater Arbeitsbereich: Daten werden zuerst komplett im privaten Arbeitsbereich bearbeitet und dann festgeschrieben
- Writeahead-Protokoll: Daten werden nach Protokollierung sofort festgeschrieben

verteilte Transaktionen

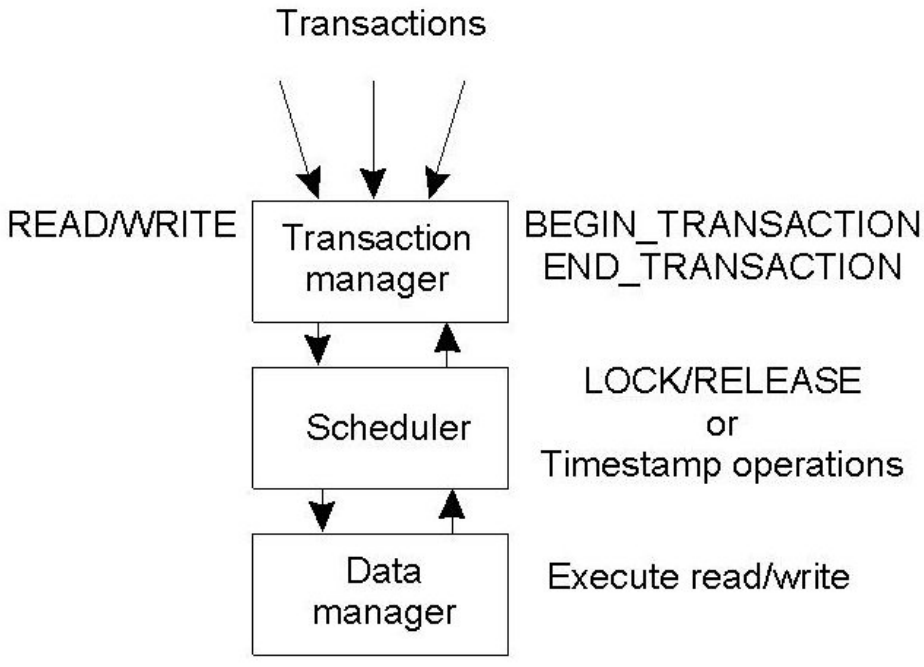
Nebenläufigkeit*

Ziel der Nebenläufigkeit ist es, dass mehrere Transaktionen gleichzeitig auf die Daten zugreifen können, ohne die Konsistenz zu gefährden. Dabei wird ein 3-schichtiges Modell verwendet:

1. Transaktionsmanager: kümmert sich um die Sicherstellung der Atomarität der Transaktionen
2. Scheduler: Kontrolliert die Abfolge der Nebenläufigkeit
3. Datenmanager: Führt Lese-und Schreibzugriffe aus

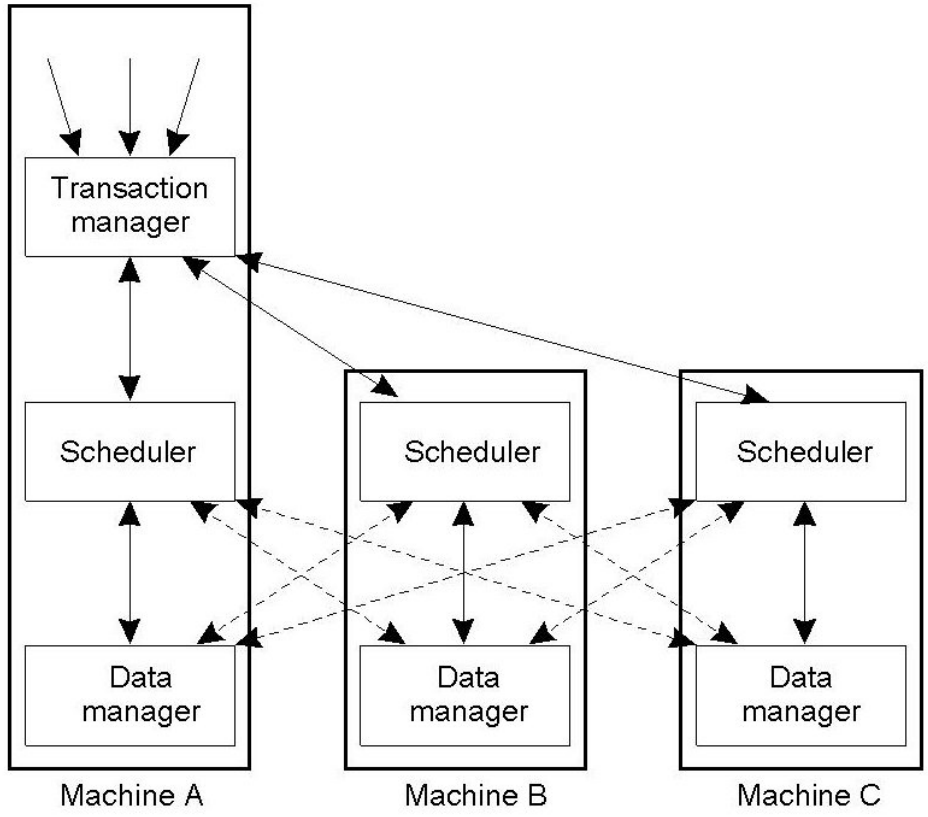
verteilte Transaktionen

Nebenläufigkeit: Modell)



verteilte Transaktionen

Nebenläufigkeit: verteiltes Modell)



Konflikte beim nebenläufigen Zugriff entstehen dann, wenn mindestens einer der beteiligten Prozesse eine Schreiboperation auf die Daten ausführt. Beim Lösen kommt man zu 2 generellen Ansätzen:

- pessimistische Ansatz: vor dem Zugriff wird synchronisiert
- optimistische Ansatz: nach dem Zugriff wird synchronisiert (mehrere Transaktionen rückgängig gemacht), *falls* ein Problem auftrat

Die gängigste Methode des Sperrens ist das **2-Phasen-Sperren**:

1. Alle benötigten Sperren werden zuerst komplett und nacheinander angefordert
2. Alle benötigten Sperren werden zuerst komplett und nacheinander freigegeben

Eine Verschärfung davon ist das **strenge 2-Phasen-Sperren**. Bei beiden Varianten kann ein **Deadlock** entstehen.

Bei der **pessimistischen Zeitstempel-Reihenfolge** wird jede Operation mit einem Zeitstempel versehen. Der Datenmanager verarbeitet die Operationen in Zeitstempel-Reihenfolge (kleinster Zeitstempel zuerst), wenn ein Konflikt vorliegt.

Verteilte Datenhaltung

Allgemein

Nicht immer werden Dateien über ein verteiltes Dateisystem im System transparent gehalten. Eine verteilte Datenhaltung hat eine geringe Verteilungstransparenz. Die Daten werden oft redundant (Kopien) auf Clients und Server gehalten.

Verteilte Datenhaltung

Anforderungen

- Redundante Haltung von Daten
- Nutzung muss folgenden Anforderungen genügen:
  - Schnelles Lesen und Speichern (vs. lokal)
  - Sicherer Zugriff (Authentifizierung, Autorisierung)
  - „Geregelter“ Zugriff (Synchronisationsmechanismen)
  - Gewährleistung der Konsistenz (Atomicity)
  - hohe Verfügbarkeit

Also: annähernd analog verteilter Dateisysteme

Verteilte Datenhaltung

Herausforderungen

- viele Nutzer, viele Dateien
- nicht vorhersehbare Nutzung
- Konflikte
- langsame Kommunikationswege
- Ressourcenverschiebungen (fehlende Relokationstransparenz)

Verteilte Datenhaltung

Einteilung

Eine Verteilte Datenhaltung lässt sich grob durch folgende Kriterien einteilen.

- Transparenz: gering bis nicht vorhanden
- meist mit Caching: lokale Kopien, die mit Serverkopien „verlinkt“ sind
- Replikation: mit/ohne automatischen Abgleich

Verteilte Datenhaltung

Konzepte

Zwei zentrale Konzepte werden aktuell verfolgt.

- Master-Slave: zentraler Knoten hält Datenhoheit; Änderungen gehen über Master
- Request-Broadcast: kein Master; Änderungen werden im Cluster angefragt und nach „OK“ verteilt.

Synchronisierung

Änderungen erfolgen zunächst lokal. Keine Synchronisation notwendig. Änderungen über mehrere Instanzen werden meist zentral zusammengeführt. Hier entsteht das Konfliktpotential, wenn:

- manuelle Aktualisierung: lange Zeit kein lokales Update
- vielen Änderungen parallel an einer Datei erfolgen (müssen)
- „blinde Änderungen“ gespeichert werden (Änderungen, die inhaltlich nichts ändern, aber zur Aktualisierung des Zeitstempels führen)

Beispiele

Git

Verteiltes Versionsverwaltungssystem mit lokaler Kopie des eingecheckten Standes.
Eigenschaften sind:

- parallele - nicht lineare - Entwicklung möglich
- lokaler Cache: vollwertiges Abbild von Server
- keine automatische Synchronisation; manueller Start nötig
- Pull/Commit und Push-Commit-Verfahren
- Merge über zentralen Server

Beispiele

Apache Zookeeper

Koordination eines Clusters (Gruppe von Knoten); gemeinsam genutzten Daten zu synchronisieren.

- robuste Datenverteilung
- Clustermanagement / Naming Service
- Leader - Wahl - einen Knoten =¿ ungerade Anzahl Knoten nötig
- zuverlässige Datenregister
- Atomicity

Beispiele

Verteilter Datastore (Facebook)

Verteilte Datenhaltung und Synchronisation (hauptsächlich auf Server) für Logging
Eigenschaften sind:

- Record-orientiert
- Besonderheit: Daten werden nur hinzufügt (Reduktion der Komplexität)
- Dateien können/müssen gekürzt werden
- Viele kleine Schreibzyklen (im System verteilt)
- lineares Schreiben
- Log-Sequencer + Log Nodes (Teilsichten)
- nutz Zookeeper

vgl.
https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs/

Beispiele

Verteilte Datenbank - Block-Chain

Nutzung verteilter Datenbank für Blockchain-Speicherung:

- Request-Broadcast - Prinzip
- Pear-to-Pear
- komplette konsistente Kopie der Chain auf „lokalen“ Server

Selbststudium

Anstöße für weiterführendes Selbststudium:

- Relokationstransparenz
- Ortsstransparenz
- WAN/LAN: Aufbau und Merkmale
- verteilte Datenbanken

Replikation

Definition: Replikation

„Replikation oder Replizierung bezeichnet die mehrfache Speicherung von Daten an typischerweise unterschiedlichen Standorten.“ (Quelle: WIKIPEDIA)

Replikation

Definition: Konsistenz

„Als Konsistenz bezeichnet man bei Datenbanken allgemein die Widerspruchsfreiheit von Daten.“ (Quelle: WIKIPEDIA)

„Jede Leseoperation für ein Datenelement x gibt einen Wert zurück, der dem Ergebnis der letzten Schreiboperation für x entspricht.“ (Quelle: Verteilte Systeme, A.Tanenbaum)

Replikation

Gründe für eine Replikation

Es gibt 2 Hauptgründe für eine Replikation:

- Zuverlässigkeit durch Datenkopien (im System verteilt)
- Leistung durch lokales Vorhandensein von Datenkopien (+Caching)

Die Hauptproblematik bei jeder Replikation:

KONSISTENZ

Sobald zwei oder mehr Instanzen (z.B. 2 Benutzer) im selben Zeitraum jeweils eine Replik (repliziertes Datenobjekt) ändern und diese wieder zusammengeführt werden müssen entsteht ein Konflikt.

Replikation

Konsistenz Fragestellung*

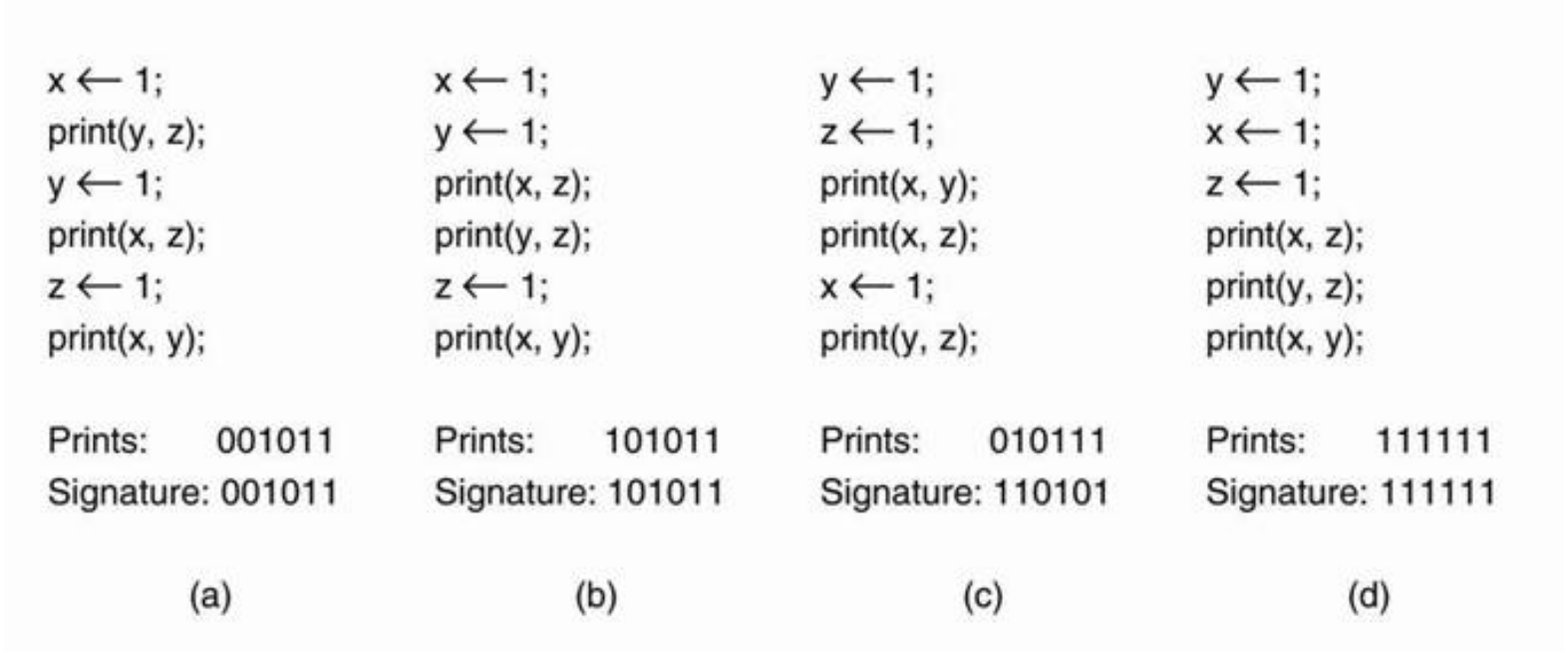
Wieviele mögliche Ausführungsreihenfolgen gibt es bei diesem lokalen Zugriff der 3 nebenläufigen Prozesse?
Wieviele mögliche Ausgaben gibt es bei diesem lokalen Zugriff?
Wieviele sind gültig?

Process P1	**Process P2**	**Process P3**
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

Replikation

Konsistenz Beispiel

4 Beispielhafte Abfolgen:



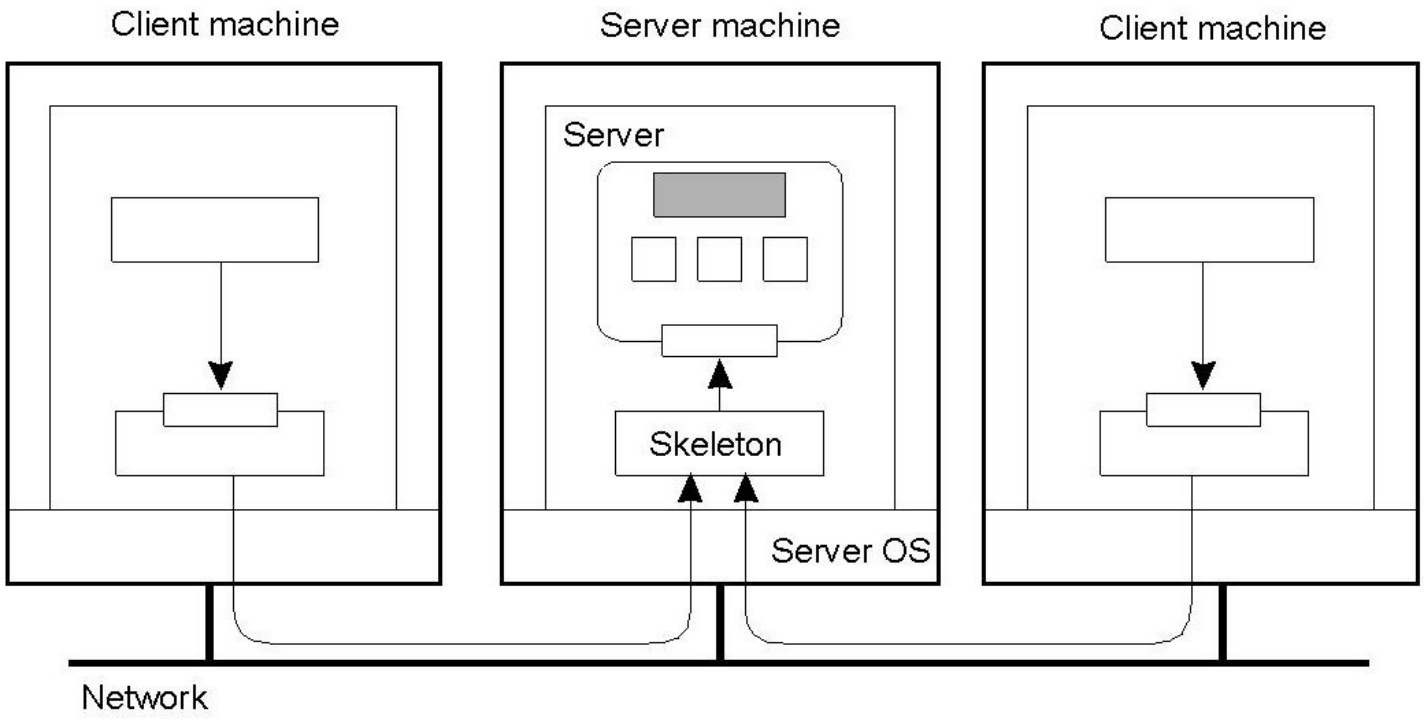
Replikation

verteilte Objekte*

Verteilte Objekte stellen eine Spezialform von verteilten Daten dar. Sie können selbst für ihre Konsistenz untereinander sorgen, oder dies dem System zu übergeben.
Um verteilte Objekte zu erzeugen, muss zunächst der Zugriff auf ein Objekt serverseitig synchronisiert werden. Hierzu gibt es zwei gängige Punkte für die Synchronisation.

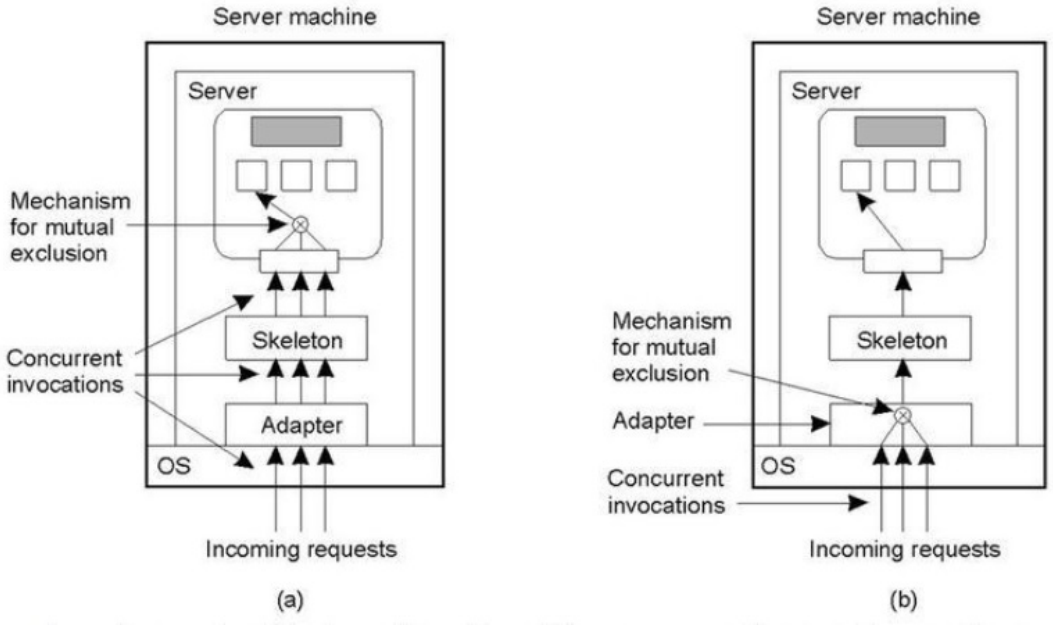
Replikation

verteilte Objekte - Synchroner Zugriff I/II



Replikation

verteilte Objekte - Synchroner Zugriff II/II



- (a) Synchronisation im Objekt
- (b) Synchronisation im Objektadapter

Replikation

Strenge Konsistenz

Jede Leseoperation für ein Datenelement x gibt einen Wert zurück, der dem Ergebnis der letzten Schreiboperation auf x entspricht.
Wichtig:

- „letzte Schreiboperation“?
- Übertragungszeiten
- Zeitabstand Lese- und Schreiboperation: Länge Zeitintervall

Das Ergebnis jeder Ausführung ist dasselbe, als wären die Operationen von allen Prozessen auf dem Datenspeicher in einer sequenziellen Reihenfolge ausgeführt worden, und die Operationen jedes einzelnen Prozesses erscheinen in dieser Abfolge in der von dem Programm vorgegebenen Reihenfolge.

Wichtig:

- kein „letzte Schreiboperation“
- jede gültige Reihenfolge ist erlaubt

Erweiterungen hierfür sind:

- **Linearisierte Konsistenz** mittels globaler Uhr

Schreiboperationen, die potenziell kausal verknüpft sind, müssen für alle Prozesse in derselben Reihenfolge sichtbar sein. Nebenläufige Schreiboperationen können auf unterschiedlichen Maschinen in einer beliebigen Reihenfolge sichtbar gemacht werden.

Wichtig:

- Abhängigkeiten müssen festgestellt werden
- Einsatz des Vektor-Zeitstempels

Replikation

FIFO Konsistenz

Schreiboperationen, die von einem einzigen Prozess ausgeführt werden, werden von allen anderen Prozessen in der Reihenfolge gesehen, in der sie ausgeführt wurden, während Schreiboperationen von unterschiedlichen Prozessen von anderen Prozessen in einer beliebigen Reihenfolge gesehen werden dürfen.

Wichtig:

- auch PRAM-Konsistenz genannt (Pipelined RAM)
- eine einzige Schreibquelle
- Schreiboperationen werden mit Prozess/-Sequenznummernpaar versehen

Replikation

Schwache Konsistenz

Die schwache Konsistenz ist geprägt durch Verwendung einer Synchronisierungsvariablen und wird im Allgemeinen durch 3 Eigenschaften gekennzeichnet:

- *Die Zugriffe auf einem Datenspeicher zugeordneten Synchronisierungsvariablen sind sequenziell konsistent.*
- *Für eine Synchronisierungsvariable ist keine Operation erlaubt, bis alle vorhergehenden Schreiboperationen überall abgeschlossen sind.*
- *Es dürfen keine Lese- und Schreiboperationen für Datenelemente ausgeführt werden, bis alle vorhergehenden Operationen für Synchronisierungsvariablen ausgeführt wurden.*

Nur das Endergebnis der Operationen für eine Synchronisierungsvariable wird nach beenden der Operationen „bekannt gegeben“.

Mögliche Erweiterungen umfassen den Aspekt des Bekanntgebens des Eintritts in einen geschützten Bereich:

- **Freigabe - Konsistenz**
- **Eintrittskonsistenz**

Replikation

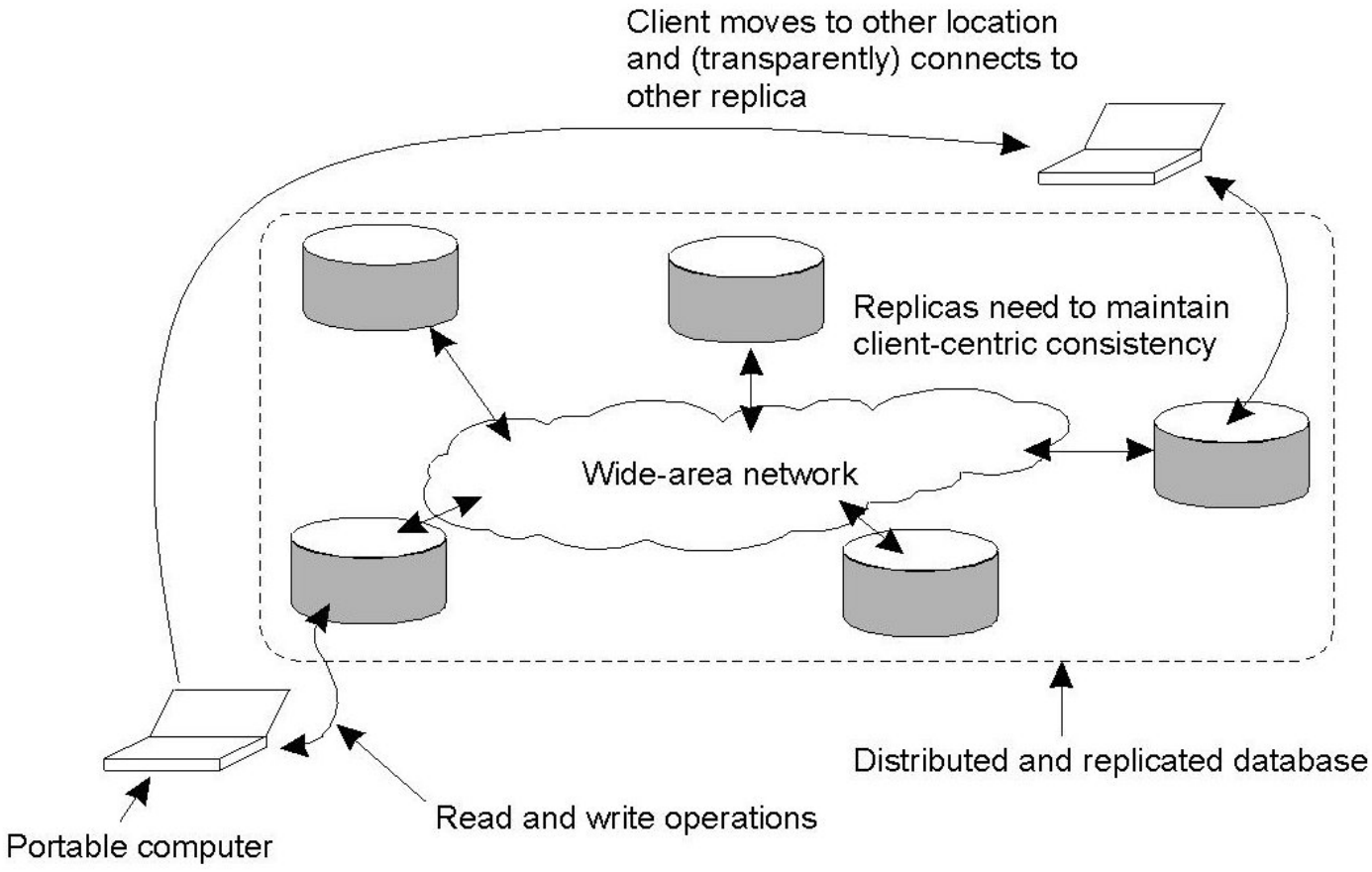
Eventuelle Konsistenz*

Daten werden über mehrere Clients „irgendwann“ abgeglichen.
D.h. die Konsistenz wird zu einem späteren Zeitpunkt hergestellt.
Die Inkonsistenz zum aktuelle Zeitpunkt wird in Kauf genommen.
Beispiel hierzu: WWW.
Problematisch:

- Zugriff eines Clients auf mehrere Repliken.

Replikation

Eventuelle Konsistenz: mobiler Zugriff



Die Grundlage der clientzentrierten Konsistenz umfasst folgende Randbedingungen:

- geringe Änderungsfluktualität
- Daten werden auf der client-seite geschrieben/gelesen (lokale Replik)
- es erfolgt ein Datenabgleich zwischen Client und Server/Clients
- Datenelemente haben einen „Eigentümer“
- Aktualität aller Daten im System zu einem Zeitpunkt X erst gegeben

Replikation

Monotones Lesen

Wenn ein Prozess den Wert eines Datenelements x liest, gibt jede nachfolgende Leseoperation für x durch diesen Prozess immer denselben Wert oder einen aktuelleren Wert zurück.
Wichtig:

- alte Werte werden NIE zurückgegeben
- Bsp.: Emails an mehreren Standorten lesen

Replikation

Monotones Schreiben

Eine Schreiboperation durch einen Prozess auf ein Datenelement x ist abgeschlossen, bevor eine nachfolgende Schreiboperation auf x durch denselben Prozess stattfindet.
Wichtig:

- vor jeder neuen Schreiboperation muss die vorliegende Kopie aktualisiert worden sein

Replikation

Read your Writes

Die Wirkung einer Schreiboperation durch einen Prozess auf einem Datenelement x wird immer von einer nachfolgenden Leseoperation auf x durch denselben Prozess gesehen.

Wichtig:

- jede Leseoperation sieht das Ergebnis der letzten Schreiboperation
- Lese- und Schreiboperation können hierbei ortsunabhängig sein

Replikation

Writes follows Read

Eine Schreiboperation durch einen Prozess auf einem Datenelement x, die einer vorhergehenden Leseoperation für x folgt, findet garantiert auf demselben oder einem neueren Wert von x statt, der gelesen wurde.

Wichtig:

- vor einem Schreiben wird der aktuelle Wert gelesen

Replikation

Implementierungen

Mögliche Implementierungen sind:

- Lese- und Schreib-IDs
- Zeitstempelvektor mit globaler Zeit

Replikation

Platzierung der Repliken

Mögliche Platzierung der Repliken:

- Permanente Repliken: Ausgangsmenge von Repliken
- Server-initiierte Repliken: Verteilung zur Leistungssteigerung
- Client-initiierte Repliken: lokale Kopie auf Client-Seite

Replikation

Aktualisierung der Repliken

Die Aktualisierung der Repliken umfasst drei Möglichkeiten:

- Weitergabe des Benachrichtigung einer Aktualisierung
- Übertragen von Daten von einer Kopie auf eine Andere
- Weitergabe von Aktualisierungsoperationen auf Kopien

Ansätze hierzu sind:

- Push-Protokoll
- Pull-Protokoll
- Push-Poll-Protokoll

Übertragung mittels:

- Unicast
- Multicast

Replikation

Löschen von Daten

Die Weitergabe des Löschen von Elementen ist problematisch, da:

- Löschen vernichtet Daten
- Referenz wird gelöscht
- mögliche ungewollte Wiederherstellung

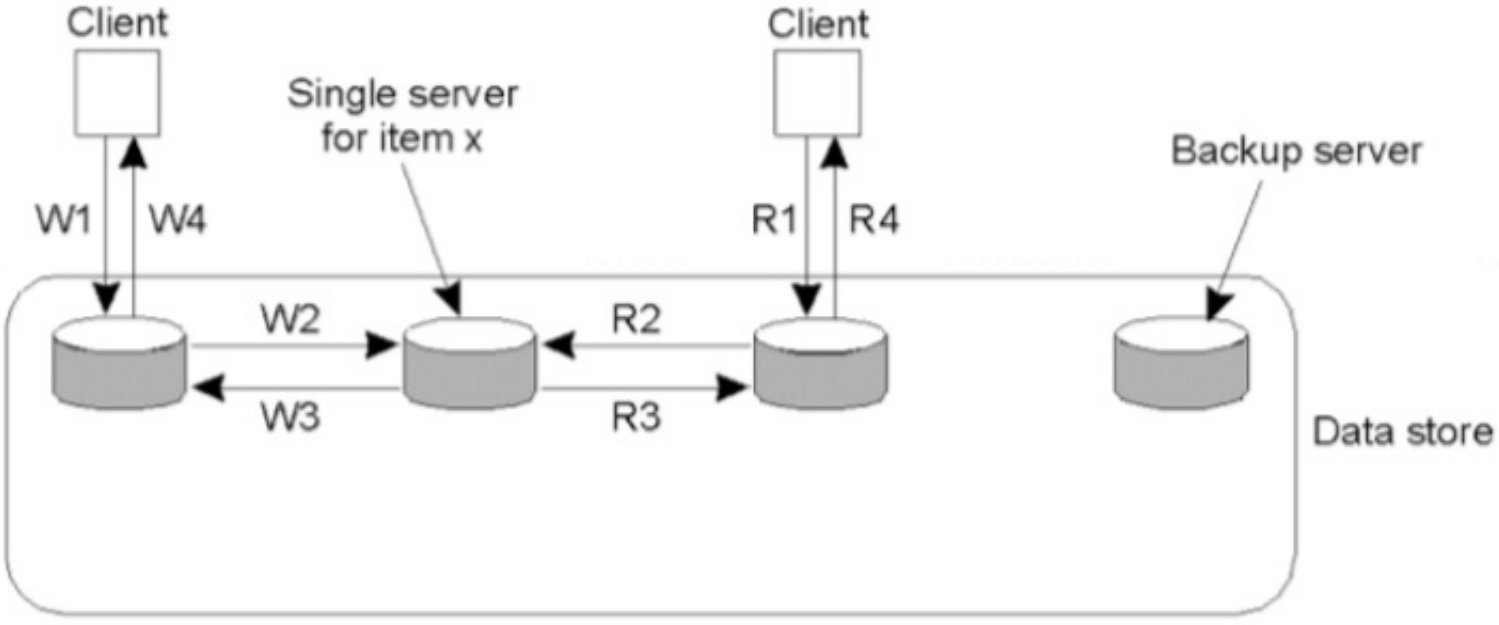
Hilfreich hierbei:

„STERBEURKUNDE“ oder DELETION-STUB

Ein Datenelement x im Datenspeicher wird einem verantwortlichen Server zugeordnet.
Problematisch hierbei sind zeitliche Verbindungsprobleme zwischen einzelnen Serverinstanzen.

Replikation

Entferntes Schreiben Protokoll - Weitergabe an festen Server

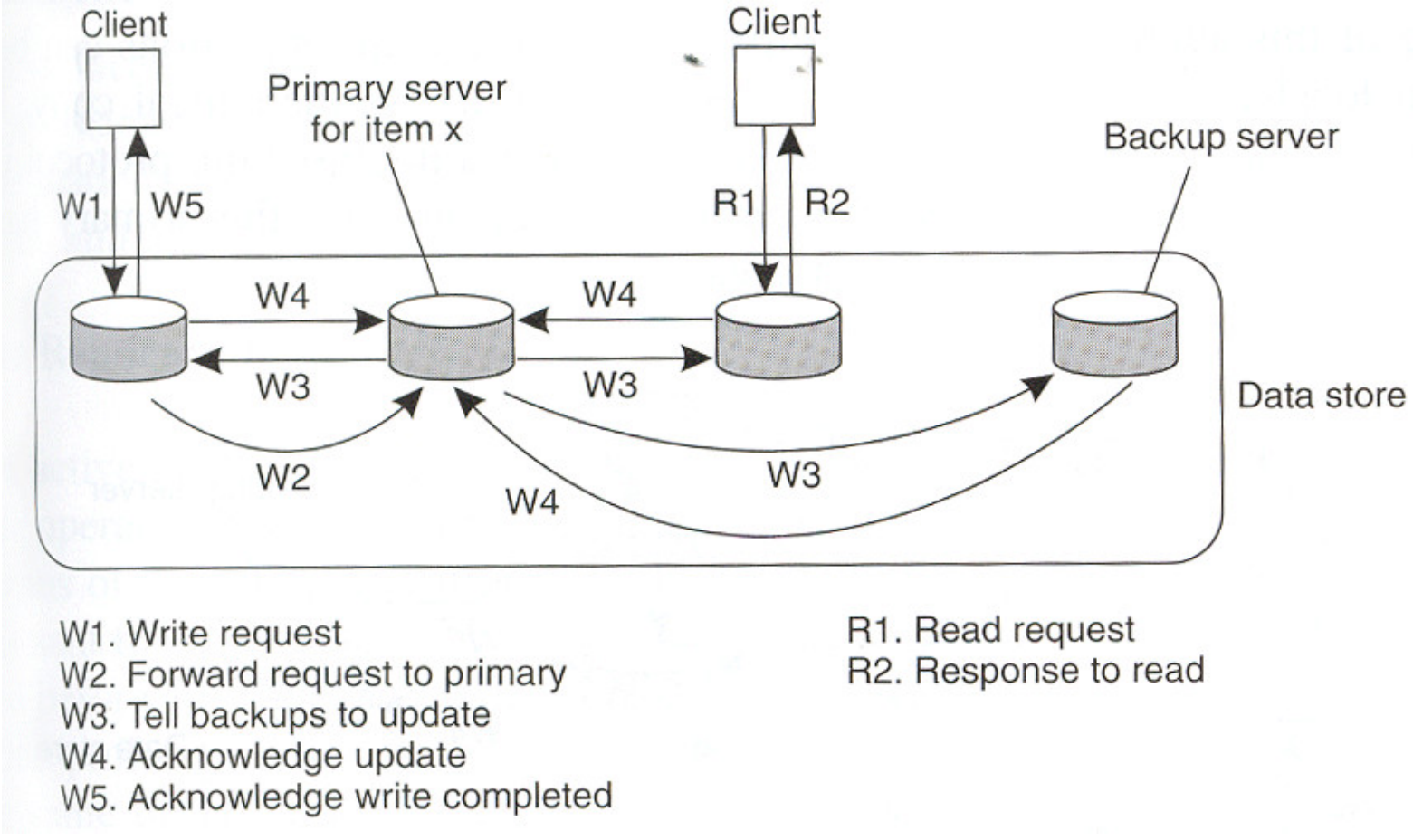


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

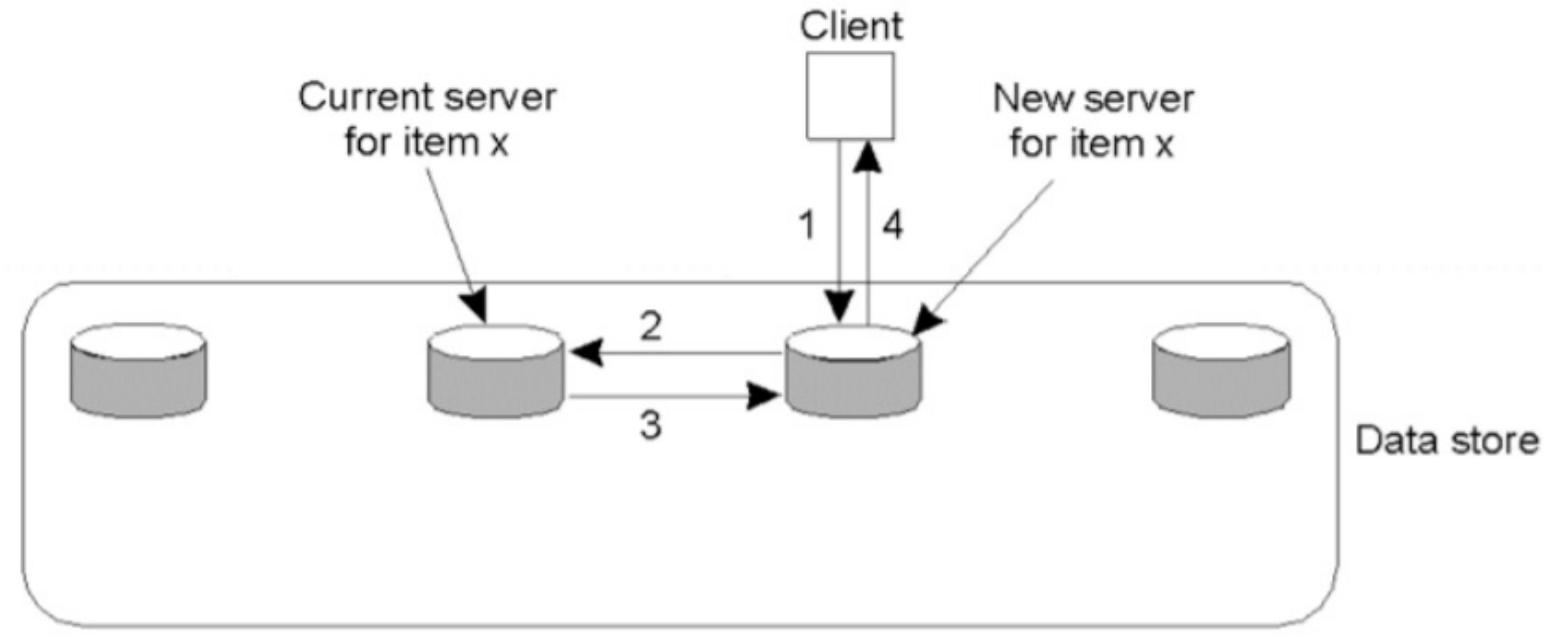
Replikation

Entferntes Schreiben Protokoll - Weitergabe an festen Server und Backup



Replikation

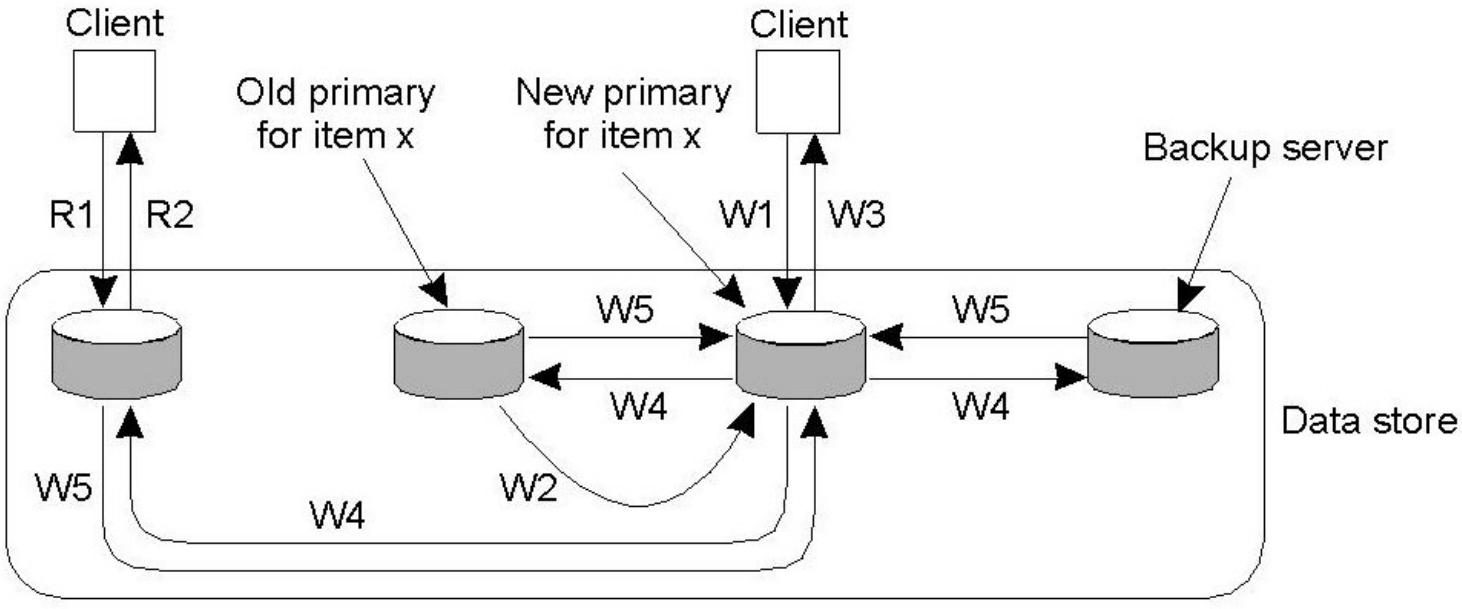
Lokales Schreiben Protokoll - eine Kopie wird migriert



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Replikation

Lokales Schreiben Protokoll - Primär Backup (mit Verschiebung)

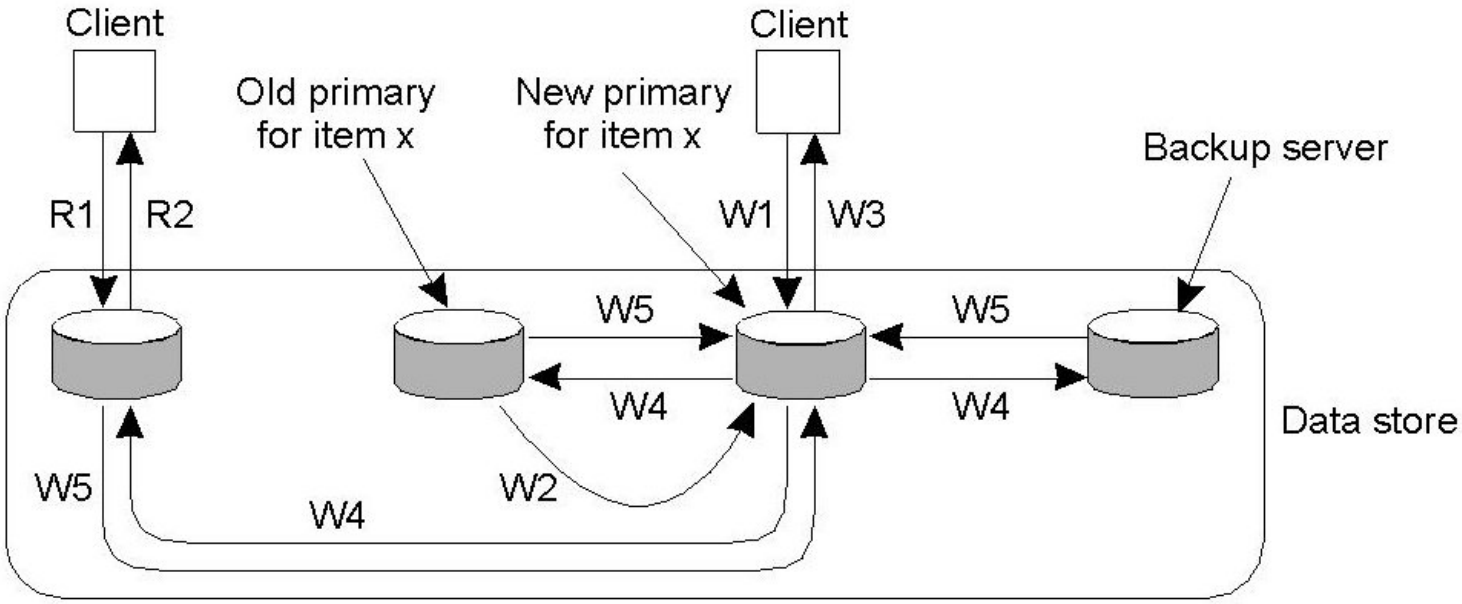


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Replikation

Lokales Schreiben Protokoll - Primär Backup

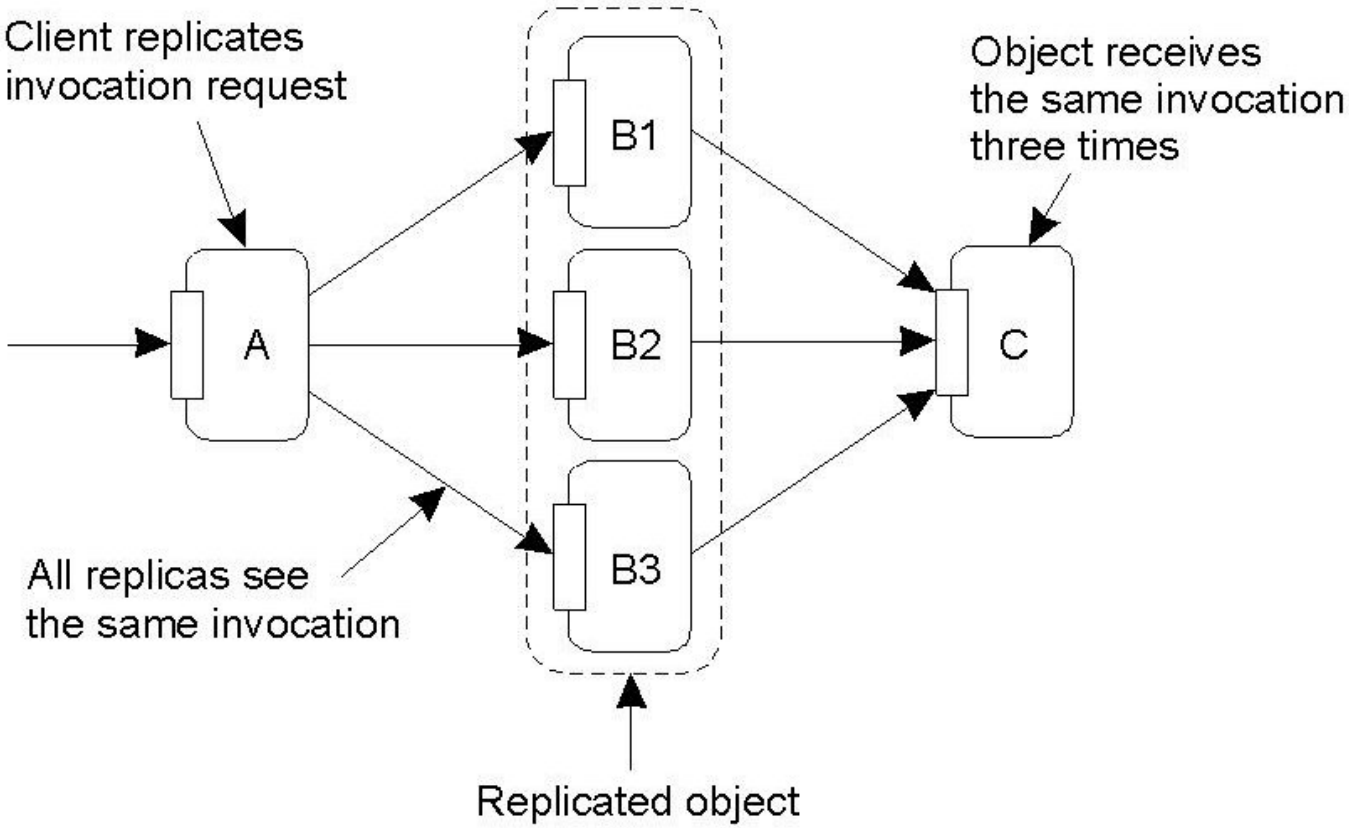


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

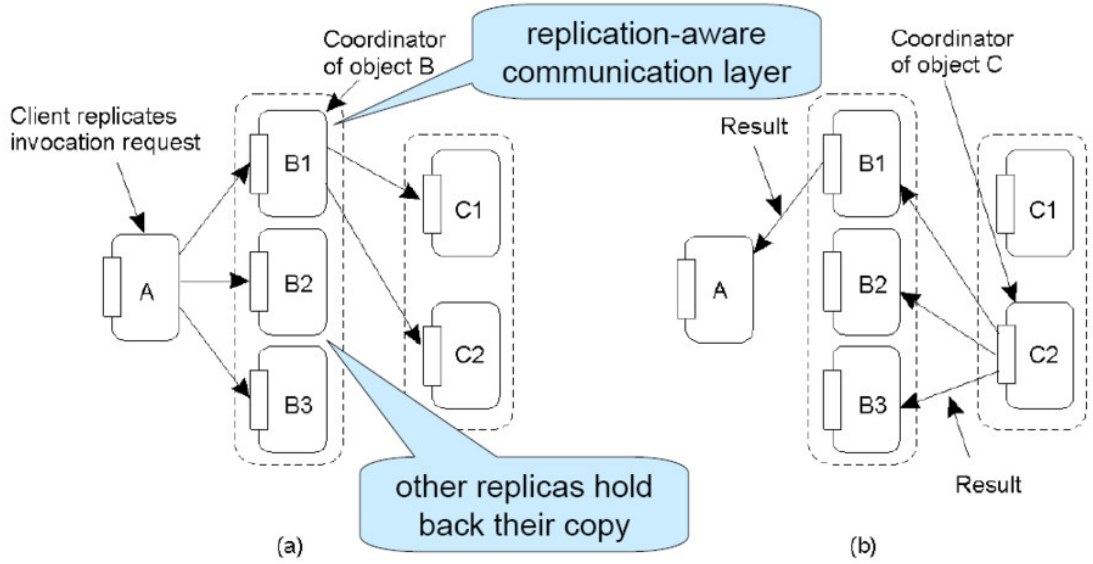
Replikation

Repliziertes - Schreiben - Protokolle



Replikation

Repliziertes - Schreiben - Protokolle - Koordinator

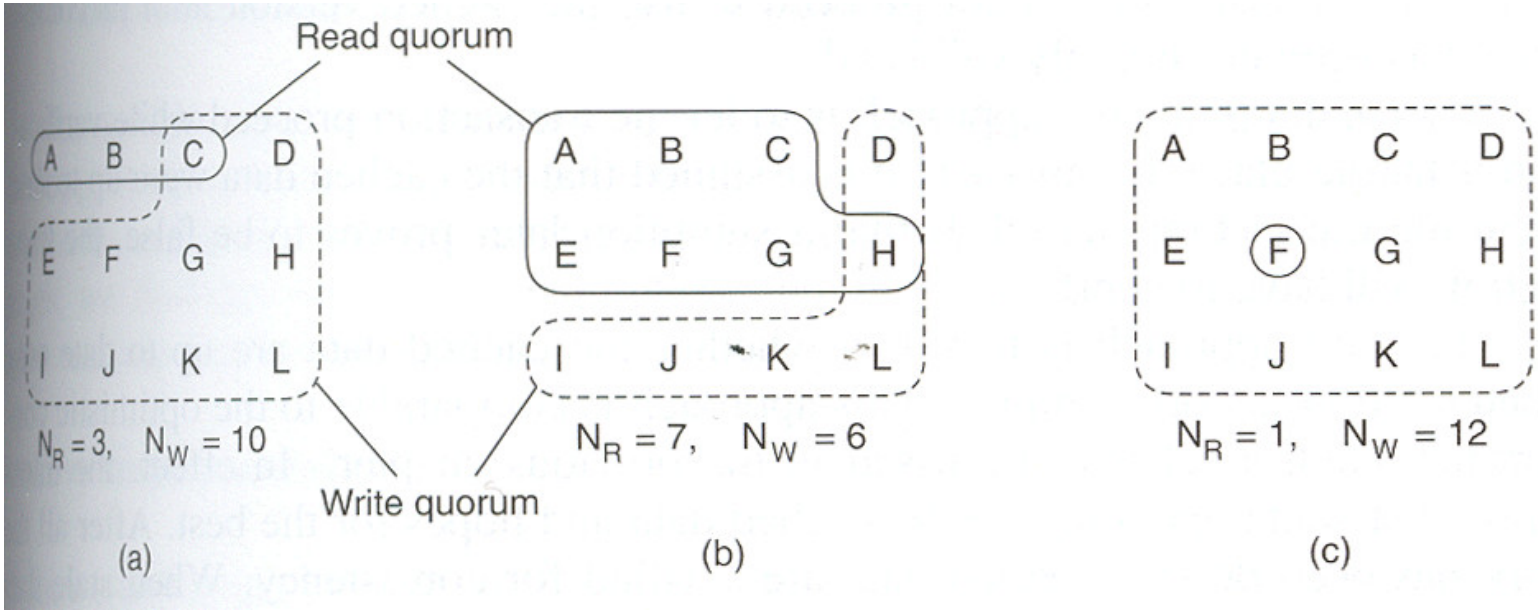


(a) Weitergabe einer Aufrufanforderung von einem replizierten Objekt an ein Anderes
(b) Rückgabe einer Antwort von einem replizierten Objekt an ein Anderes

Replikation

Quorum-basierte - Protokolle

Basis hierbei ist, dass bei N Repliken N/2+1 Zustimmungen einem Prozess für eine Operation vorliegen müssen.



(a) Korrekte Auswahl einer Lese-und Schreibmenge
(b) Eine Wahl, die zu Schreib/Schreib-Konflikten führen kann
(c) ROWA (read-one, Write-all)

Raft

ein neues „Wunder“

Algorithmus zu Datenverteilung und Ausfallsicherheit

1. benötigt ungerade Anzahl Knoten
2. konsensbasierend
3. Koordinator mit Wahlalgorithmus

Raft

Ablauf

https://raft.github.io/ http://thesecretlivesofdata.com/raft/

Raft

Einschätzung

. . . Teil der Diskussion . . .