Verteilte Systeme

V0.41

SS 2012

Prof. Dr. Christian Siemers
TU Clausthal, Institut für Informatik

Inhaltsverzeichnis

1 Einleitung

1.1 Zielgruppen

Diplomstudiengang Informatik:

Hauptstudium

Bereich praktische Informatik oder technische Informatik

Diplomstudiengang Wirtschaftsinformatik:

Hauptstudium

Bereich WiInf/praktische Informatik oder technische Informatik

Bachelor Informatik / Wirtschaftsinformatik

Pflicht

Weitere Studiengänge?

Übungen:

- Wesentlicher Teil der Programmieraufgaben: Arbeit mit Manuals und Tutorials.
- Gruppenabgaben von bis zu 3 Leuten sind erlaubt.
- Bei Gruppenabgaben muss jedes Gruppenmitglied mind. einmal im Semester die Lösungen vorstellen.
- Die Übungsaufgaben werden mind. eine Woche vor der Übung in der Vorlesung verteilt/online gestellt.

1.2 Literatur

[1] Coulouris, Dollimore & Kindberg: Distributed Systems – Concepts and Design. 4. Auflage, Addison Wesley.

[2] Tanenbaum & Van Steen:: Verteilte Systeme – Prinzipien und Paradigmen. 2. Auflage, Pearson Studium

2 Definition und Eigenschaften Verteilter Systeme

2.1 Definition des Begriffs Verteiltes System

Definition 2.1:

Ein Verteiltes System ist eine Menge vernetzter Computer, auf denen gemeinsam eine Anwendung läuft.

Naive Definition!

Definition 2.2 (Sloman, Kramer, 1989):

Ein verteiltes Dateisystem ist eines, in dem mehrere autonome Prozessoren und Datenspeicher […] so kooperierend zusammenarbeiten, dass ein gemeinsames Ziel erreicht wird. Die Prozesse koordinieren ihre Aktivitäten und tauschen Informationen über ein Kommunikationsnetzwerk aus.

Definition 2.3 (Tanenbaum, van Steen):

Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes köhärentes System erscheint

Definition 2.4 (Coulouris, Dollimore & Kindberg)

A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

Die Folgen:

- Nebenläufigkeit
- Keine globale Uhr
- Unabhängiges Versagen

Ziele verteiler Systeme:

a) historisch

- Reduzierte Kosten
- Modularität und einfachere Software
- Flexibilität und Erweiterbarkeit
- Verfügbarkeit
- Leistung
- Lokale Kontrolle

b) neu

- Neue Anwendungsmöglichkeiten, die nur als VS realisierbar sind
- File Sharing
- Chats
- Elektronische Meetings
- Mobile Anwendungen
- „Social Software“

Besondere Anforderungen:

- Heterogenität
    - Netzwerke
    - Betriebssysteme
    - Programmiersprachen
    - Hardware
    - Software-Implementierungen von verschiedenen Entwicklern
- Offenheit
    - Dynamische Erweiterung des Systems („Plug-In“) sollte möglich sein
    - Essentiell: Technische Schnittstellen müssen veröffentlicht sein
    - Prozess der Weiterentwicklung z.B. über RFC‘s
- Sicherheit
    - Vertraulichkeit
    - Datenintegrität
    - Verfügbarkeit
- Skalierung
    - System muss auch bei erheblicher Erhöhung der beteiligten Ressourcen funktional bleiben
    - Dimensionen: Anzahl der Rechner, User, Verbindungen, Dateien, …
    - Kostenkontrolle physikalischer Ressourcen
    - Kontrolle des Performance-Verlusts
    - Sicherung der Verfügbarkeit von Software-Ressourcen
    - Verhinderung von Leistungs-Bottlenecks
- Fehlerbehandlung
    - Erkennung von Fehlern
    - Maskierung von Fehlern (z.B. Doppelte Speicherung von Daten, Ignorieren und Neuanfordern bei fehlerhaften Nachrichten)
    - Fehlertoleranz
    - Erholung von Fehlern (Recovery)
- Nebenläufigkeit
    - Synchronisation verschiedener unabhängiger Prozesse, Aufgaben, Nutzer
    - Kontrolle des Zugriffs auf gemeinsam genutzte Ressourcen
- Transparenz (Unsichtbarkeit):
    - Zugangstransparenz: gleicher Zugriff auf lokale & entfernte Ressourcen
    - Ortstransparenz: Adresse von Ressourcen (physisch, Netzwerk)
    - Nebenläufigkeitstransparenz: keine sichtbare Störung durch andere Prozesse
    - Fehlertransparenz: Fehler werden verborgen, Benutzer und Anwendungen können Ziel trotzdem erreichen
    - Replikationstransparenz: Existenz mehrerer Datenkopien ist dem Benutzer/Anwendungsprogramm nicht bekannt

- Mobilitätstransparenz: Ressourcen (Daten, Programme, Rechner) können ohne Einfluss auf das System migriert werden
- Leistungstransparenz: Dynamische Rekonfiguration zur Leistungssteigerung möglich und nicht vom Benutzer sichtbar
- Skalierungstransparenz: Dynamische Größenänderung möglich und nicht vom Benutzer sichtbar

Zusammenfassend bedeutet dies für Verteilte (Rechner-)Systeme, dass alles für ein offenes System konzipiert wird, also oberste Priorität Flexibilität gepaart mit Zuverlässigkeit.

In Verteilten, eingebetteten Systemen liegt diese Priorität im Allgemeinen auf ganz anderem Gebiet: Echtzeitfähigkeit und Zuverlässigkeit des Systems (und damit des Netzwerks).

2.2 Architekturmodelle Verteilter Systeme

Definition:

Architektur eines Systems = Struktur, bestehend aus Komponenten (Funktionen) und deren Zusammenhang (Interaktionen, Abhängigkeiten)

Spezielle Fragen der Architektur eines Verteilten Systems:

- Wo sind Komponenten im Netzwerk?
- Welche Rollen und Kommunikationsmuster gibt es?
- Umgang mit Herausforderungen (aus Anwendungskontext), z.B.
  - *Ausfall eines Rechners in einem File Sharing System (Fehlerbehandlung)*
  - *Spitzen im Zugriff auf Webserver (Skalierung)*
  - *Unterstützung zukünftiger technischer Standards (Heterogenität & Offenheit)*

2.2.1 Grundlegendes Architekturmodell

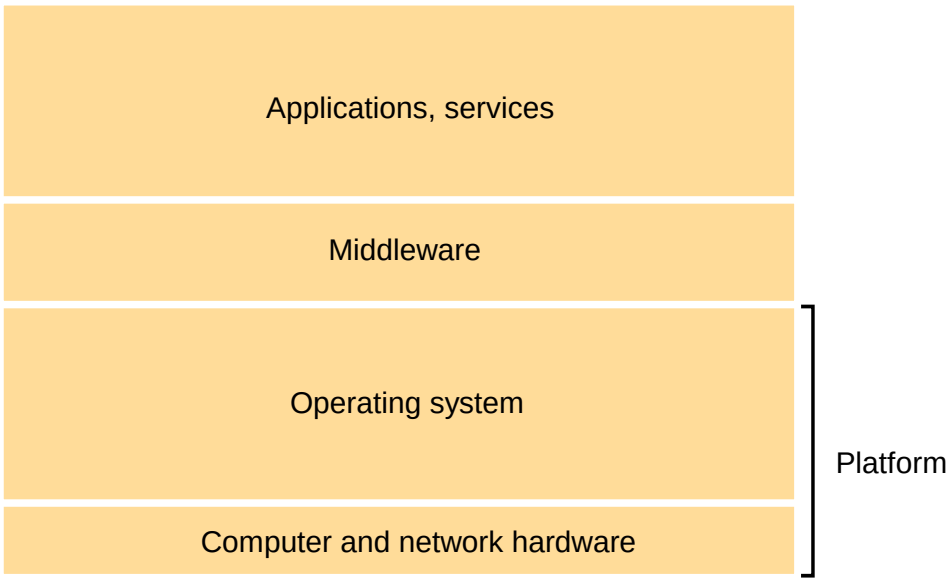


Bild 2.1 Architekturmodell Verteilter Systeme

- Middleware kann von Anwendungen verwendet werden, um Funktionen auf OS/Hardware-Ebene und Netzwerkebene zu realisieren (auch Kommunikation!)
- Middleware leistet Beitrag zur Transparenz des Systems

- Beispiele für Middleware:
    - CORBA (Common Object Request Broker Architecture)
    - RMI (Remote Method Invocation)
    - RPC (Remote Procedure Call)
    - Web Services
    - DCOM (Distributed Component Object Model)

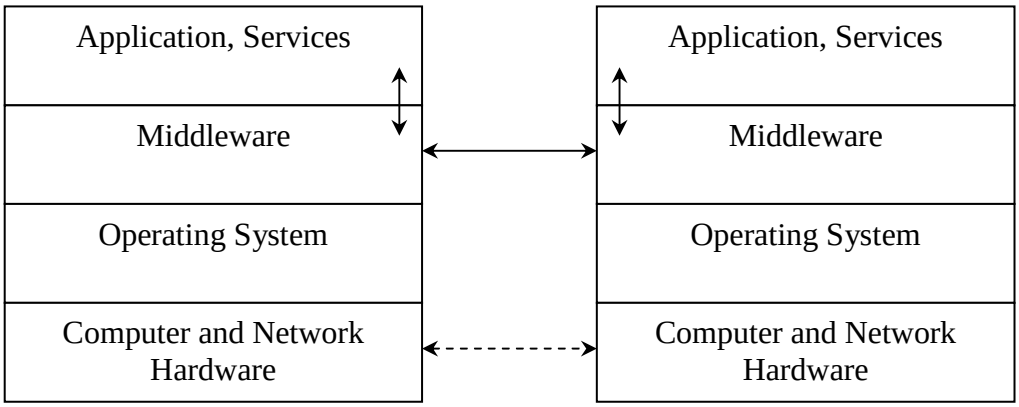


Bild 2.2 Kommunikationen in Verteilten Systemen

Typisch: Delegation der Kommunikation an die Middleware. Dies ist oft eine sinnvolle Strategie bei der Entwicklung von Verteilten Systemen

Aber: Falls Anwendungen spezifische Anforderungen haben (die nicht sinnvoll in allgemeiner Middleware realisierbar sind), findet Kommunikation auch auf Anwendungsebene statt.

Beispiele:

- Transfer sehr großer Daten über unsichere Netze
- Logisches FIFO in Verteilten Systemen

2.2.2 Verfeinerung des Architekturmodells

Zentrale Frage bei der Architektur ist: Welche Komponenten liegen wo?

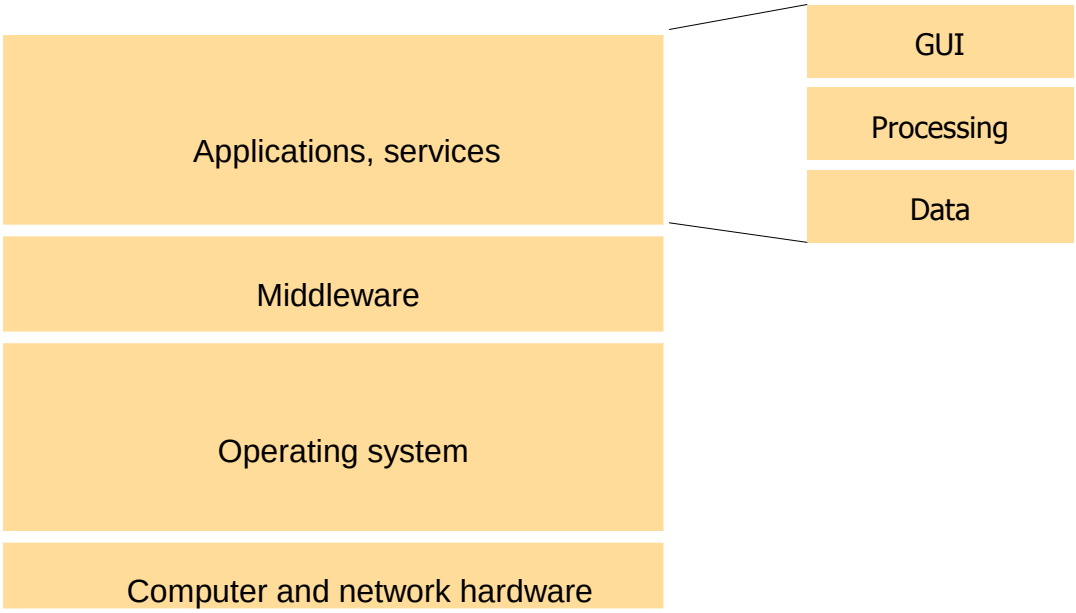


Bild 2.3 Verfeinerung Application Tier

Architekturmodell 1: Client/Server

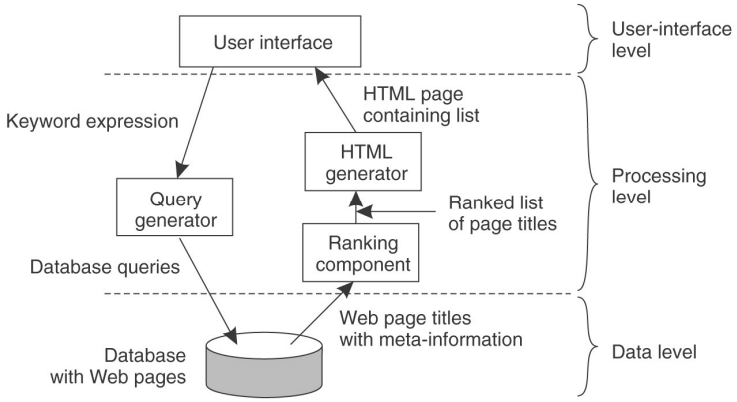


Bild 2.4 Architekturmodell Internet-Suchmaschinen

- „Klassisches“ Modell Verteilter Systeme: Client-Anwendungen stellen Anfragen an Server und erhalten Antwort
- Server fragen dazu ggf. bei anderen Servern nach.

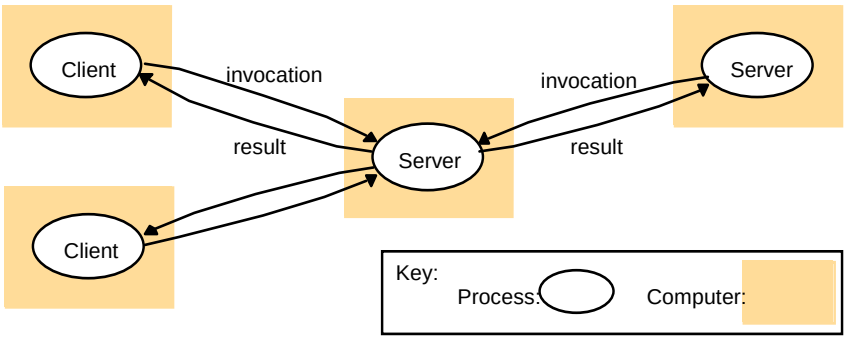


Bild 2.5 Client-Server- Modell

Beispiel: Virtual Network Computing

- 2-Tier Architektur (direkte Client-Server-Kommunikation)
- Client erhält Bildschirminhalt des Servers
- Aktionen des Clients (Mausklicks, Tastatureingaben etc) werden an Server weitergeleitet
- Ausführung der Aktionen auf Server: Änderung des Zustands der Anwendung
- „Thin Client“-Ansatz

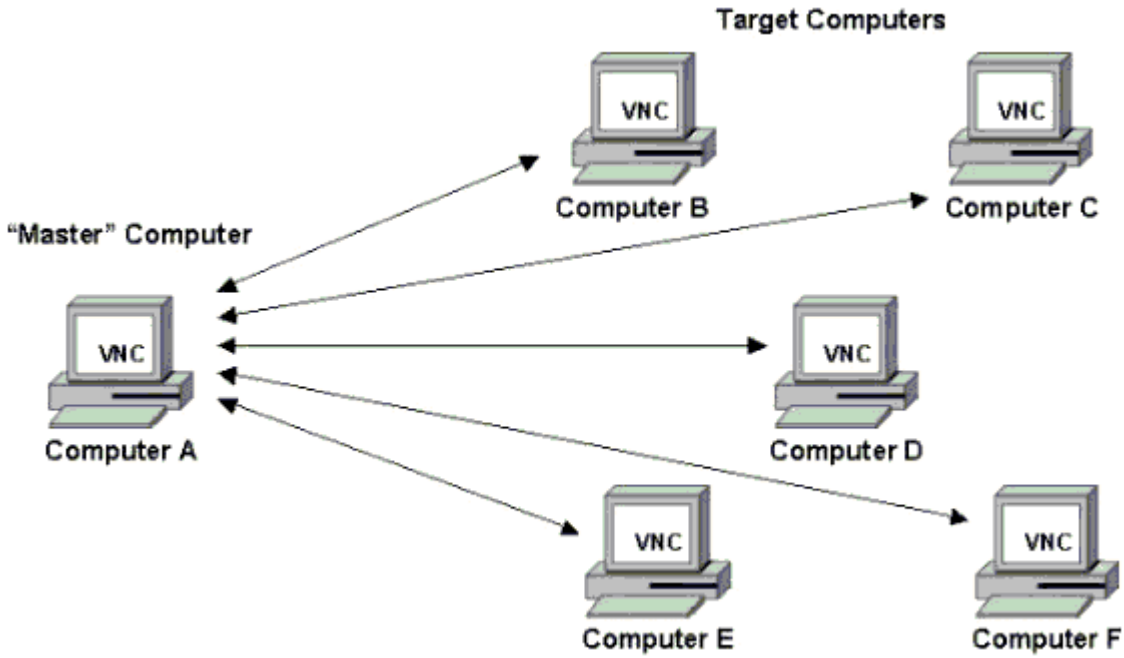
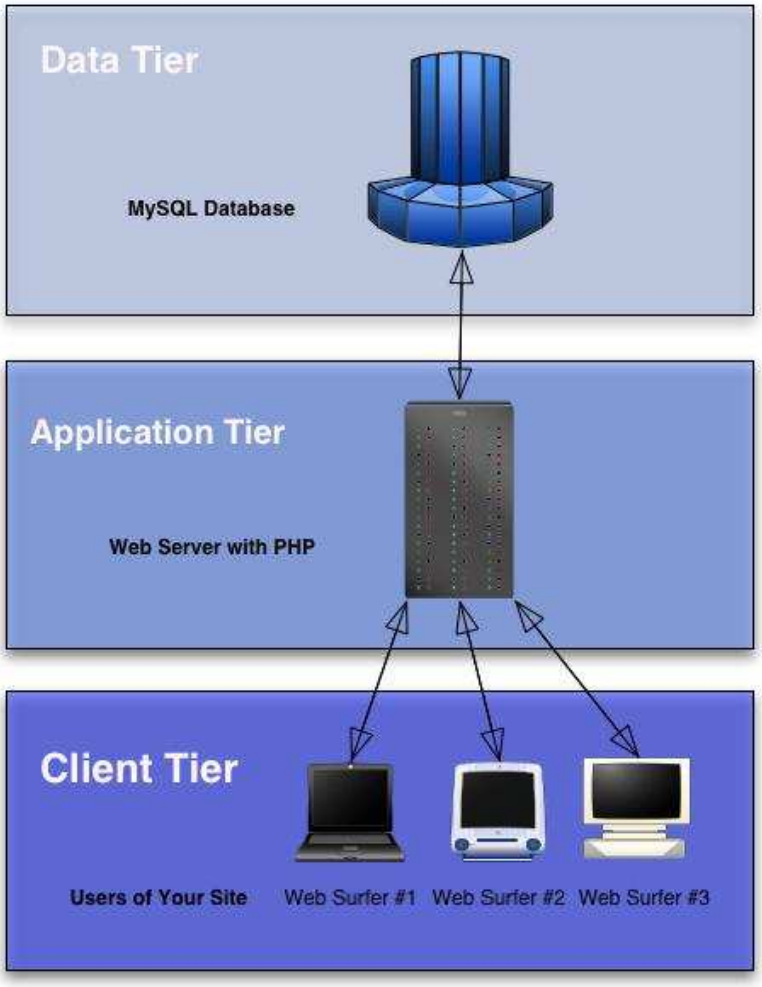


Bild 2.6 Virtual Network Computing

Beispiel 3-Tier-WEB-Architektur



Beispielszenario: Eine WWW-Suchmaschine reagiert auf Anfragen von Clients und gibt auf Suchworte eine Liste von URLs zurück. Gleichzeitig hat diese Suchmaschine mehrere Crawler, die das Web nach URL‘s durchsuchen und die Datenbank aktualisieren.

Welche Anforderungen bestehen hinsichtlich der Synchronisation der Prozesse?

- Immer die auf die Anfrage passende Antwort geben, auch bei vielen nebenläufigen Anfragen
- Ergebnisse eines Crawlers sollten die eines anderen nicht überschreiben
- Während der Datenbankeintrag zu einem Schlüsselwort geändert wird, müssen Antworten zu diesem Anfragewort warten (Aktualität der Suchergebnisse: Neue Seiten in Datenbank, Löschen von Einträgen aus Datenbank)

Variation der Client-Server-Architektur

Dienste mit mehreren Servern

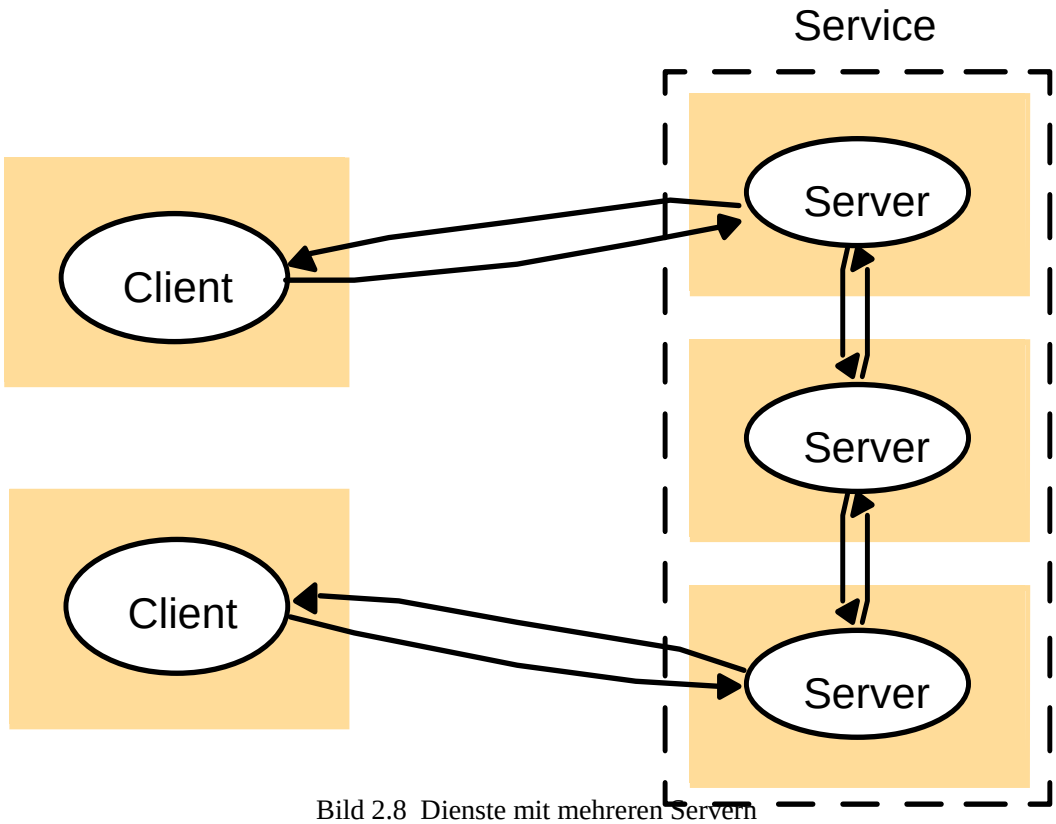


Bild 2.8 Dienste mit mehreren Servern

Proxy-Server (von *proximus*, lat., der Nächste):

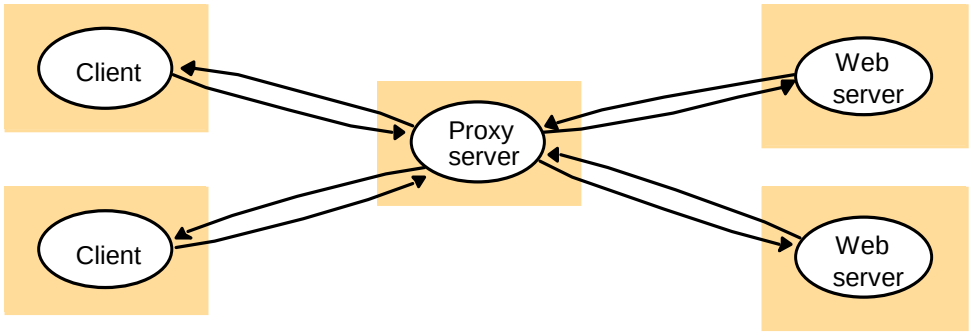


Bild 2.9 Proxy Server

- Caching von Daten (Zwischenspeicherung), z.B. zur Verbesserung der Antwortzeit
- Firewall zur Kontrolle des Zugriffs auf Ressourcen

Architekturmodell 2: Peer-to-Peer-Architektur

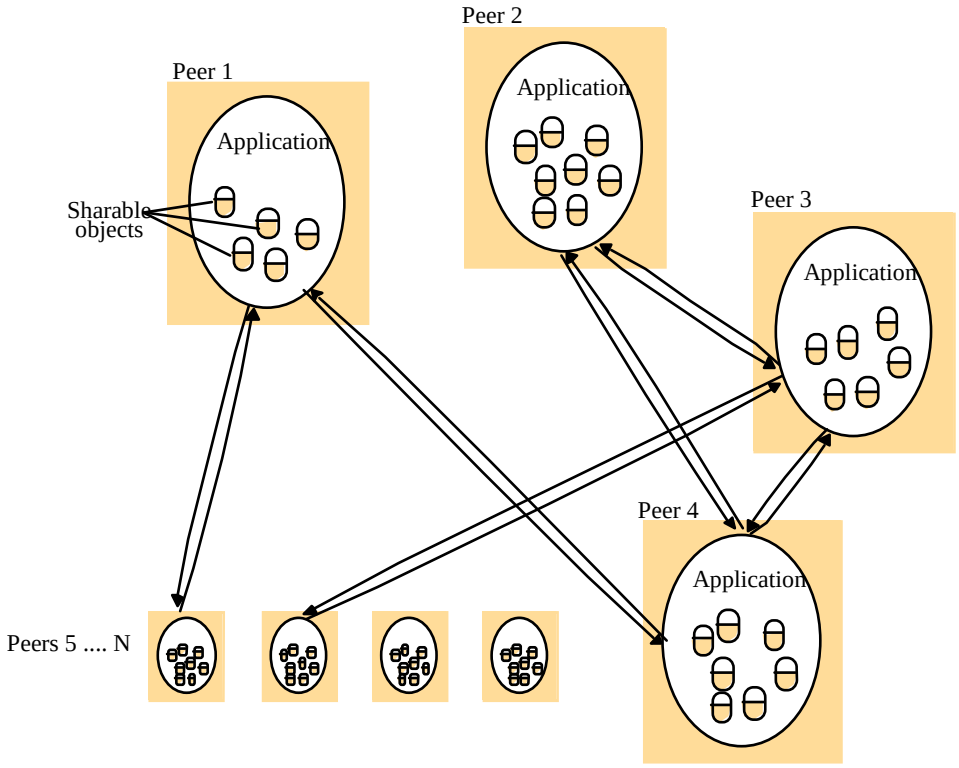


Bild 2.10 Peer-to-Peer-Architektur

Das Wort *Peer* bedeutet Gleichgestellter oder Ebenbürtiger. Jeder Peer ist gleichzeitig Server und Client!

Jeder Peer hat (evtl. verschiedene) Daten und (den gleichen) Programmcode, insbesondere zur Koordination.

Allgemeine Vor- und Nachteile von P2P-Architekturen:

+ Keine Bottlenecks
+ Robustheit
+ Flexible Verteilung der Belastung
– Schwierigkeit, globale Information zu erhalten
– Kontrolle & Administration

Vergleich zwischen C/S- und P2P-Architektur

Ein Programm zum Filetransfer soll entworfen werden. Als Architektur kommt entweder ein Client-Server-Modell oder P2P in Frage. Was sind die Vor- und Nachteile dieser beiden Wahlmöglichkeiten?

C/S Vorteile:

- Zentraler Server: es ist sehr einfach, Daten zu indexieren.
- Absolute Kontrolle über die Daten: Verfügbarkeit und Zugang durch denjenigen der den Server kontrolliert.

C/S Nachteile:

- Zentraler Server kann schnell ein Bottleneck bezüglich Bandbreite werden
- Dateien müssen zu jedem Client vollständig übertragen werden, das bedeutet hohe Kosten wegen des erzeugten Netzwerkverkehrs
- Zentraler Server kann ein „Single Point of Failure“ sein: wenn nur dieser ausfällt ist ein Datenaustausch unmöglich

P2P Vorteile:

- Kein zentraler Server: das ganze Netz ist schwerer angreif- und abschaltbar (z.B. um staatlicher Zensur zu entgehen). Viele Knoten können ausfallen und doch wird das Netzwerk noch funktionstüchtig bleiben
- „Verteilte“ Übertragung der Dateien: man ist nicht auf die maximale Übertragungsgeschwindigkeit eines Knotens angewiesen, sondern kann – bei entsprechender Verbreitung der Datei im Netzwerk – von beliebig vielen Knoten gleichzeitig Teile der Datei herunterladen, bis die eigene maximale Bandbreite ausgereizt ist.
- Wenn eine Datei einmal im Netzwerk ist, übernehmen die Nutzer die weitere Verbreitung. Der eigentliche Einsteller erreicht somit ein sehr weites Publikum unter Aufwendung sehr weniger Ressourcen. Beispiel: eine Firma die ein Update über Bittorrent Technologie verbreitet benötigt dazu wesentlich weniger Bandbreite als wenn sie jedem Client einzeln das komplette Update überträgt.

P2P Nachteile:

- Kein zentraler Server – Clients müssen sich irgendwie finden, es muss eine geeignete Lösung gefunden werden die anderen Clients und Dateien zu lokalisieren (Name-Service Problem)
- Man hat quasi keine Kontrolle über die Daten im Netz, Dateien wieder aus dem Netz zu entfernen ist schwierig. Auch können über meinen Rechner/Knoten Informationen, Dateien oder Teile davon, von anderen Nutzern übertragen werden welches ethische, moralische oder auch rechtliche Probleme darstellen kann.

3 Grundlagen der Kommunikation in Netzwerken

3.1 Grundlegende Eigenschaften von Netzwerken

Typen von Netzwerken:

Beispiele verschiedener Typen von Netzwerken:

- LAN (Local Area Network)
- WAN (Wide Area Network)
- MAN (Metropolitan Area Network)
- Internet (Netz der Netze)

Wichtige Charakteristika:

- Datenübertragung (Geschwindigkeit, Modus, Medien)
- Netzwerktopologie
- Protokolle
- Dienste

Übertragungszeit von Nachrichten:

- Latenz = Verzögerung zwischen Senden einer Nachricht und Beginn des Empfangs der Nachricht
- Datentransferrate = Geschwindigkeit (Bit/Sekunde), mit der Daten übertragen werden, nachdem die Verbindung erstellt wurde.
- Übertragungszeit = Latenz + Datenvolumen/Datentransferrate

- Bandbreite eines Netzwerks = Totaler Durchsatz des Netzwerks (Datenvolumen pro Zeit)
- Frage: Ist Bandbreite = Datentransferrate?

Übertragungsmodi:

Broadcasting: Daten werden an alle Knoten geschickt.

Circuit switching: Für jede Datenübertragung wird Kanal aufgebaut

Packet switching: Daten werden in Pakete unterteilt. Jedes Paket wird durch das Netzwerk übertragen, im Router wird nach dem „Store and Forward“-Prinzip gearbeitet.

Frame Relay: „Virtuelle Übertragungskanäle“ mit kontinuierlicher Übertragung. Hier kann eine Garantie für *Quality of Service* übernommen werden.

Übersicht zu Netzwerkcharakteristika:

	Example	*Range*	*Bandwidth (Mbps)*	*Latency (ms)*
Wired:				
LAN	Ethernet	1-2 km	10-1000	1-10
WAN	IP routing	worldwide	0.010-600	100-500
MAN	ATM	250 km	1-150	10
Internetwork	Internet	worldwide	0.5-600	100-500
Wireless:				
WPAN	Bluetooth (802.15.1)	10 - 30m	0.5-2	5-20
WLAN	WiFi (IEEE 802.11)	0.15-1.5 km	2-54	5-20
WMAN	WiMAX (802.16)	550 km	1.5-20	5-20
WWAN	GSM, 3G phone nets	worldwide	0.01-02	100-500

Tabelle 3.1 Zusammenfassung einiger Netzwerkcharakteristika

3.2 ISO/OSI-7-Schichtenmodell

ISO bedeutet "*International Organization for Standardization*", OSI steht für "*Open System Interconnetiont*".

Netzwerksoftware ist typischerweise in Schichten aufgeteilt (Idee: Von der Anwendung zur physikalischen Netzwerkebene)

Grundprinzip:

- Jede Schicht des Senders scheint mit der entsprechenden Schicht beim Empfänger zu kommunizieren (mittels speziellem Protokoll)
- Umsetzung: höhere Schicht (mit abstrakterer Aufgabe) beim Sender delegiert Aufgaben an niedrigere Schicht und fügt protokollspezifische Informationen zur Nachricht hinzu
- usw….
- Datenübertragung findet auf unterster Ebene statt.
- Mit Hilfe des „Auspackens" der Nachricht (Interpretation der protokollspezifischen Informationen) kann beim Empfänger die Nachricht zur höchsten Schicht geliefert werden.

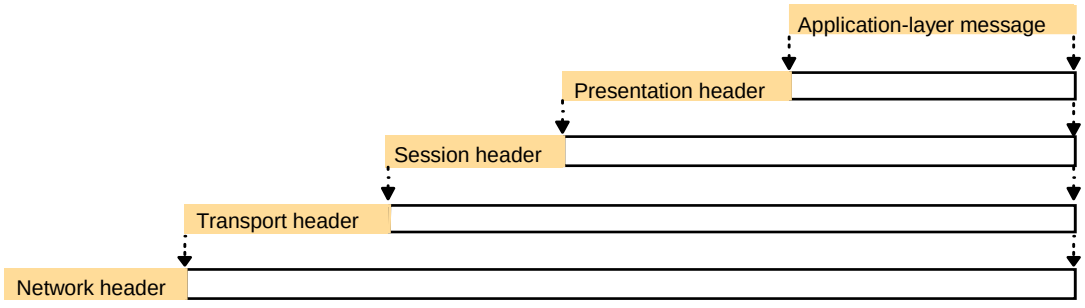


Bild 3.1 Hinzufügen von Header-Informationen bei der Übertragung

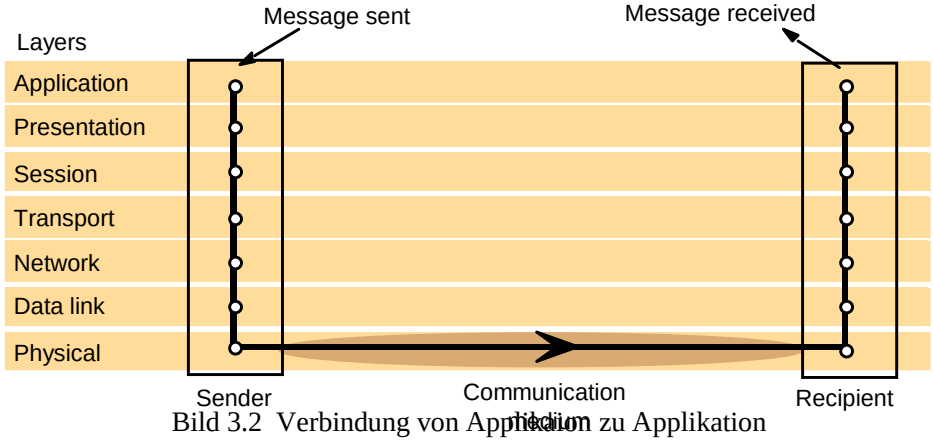


Bild 3.2 Verbindung von Application zu Applikation

Die Schichten im OSI-Modell haben folgende Aufgaben:

Layer	*Description*	*Examples*
Application	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP, FTP, SMTP, CORBA IIOP
Presentation	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required.	Secure Sockets (SSL),CORBA Data Rep.
Session	At this level reliability and adaptation are performed, such as detection of failures and automatic recovery.	TCP, UDP
Transport	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless.	
Network	Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required.	IP, ATM virtual circuits
Data link	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts.	Ethernet MAC, ATM cell transfer, PPP
Physical	The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).	Ethernet base- band signalling, ISDN

Tabelle 3.2 Darstellung der Layer 1-7 im OSI-Schichtenmodell

3.3 Routing

Für die Übermittlung von Daten über mehrere Netze/Verbindungen sind die Schichten 3 bis 5 zuständig. Eines der wichtigsten Probleme, die zu lösen sind, ist das Routing.

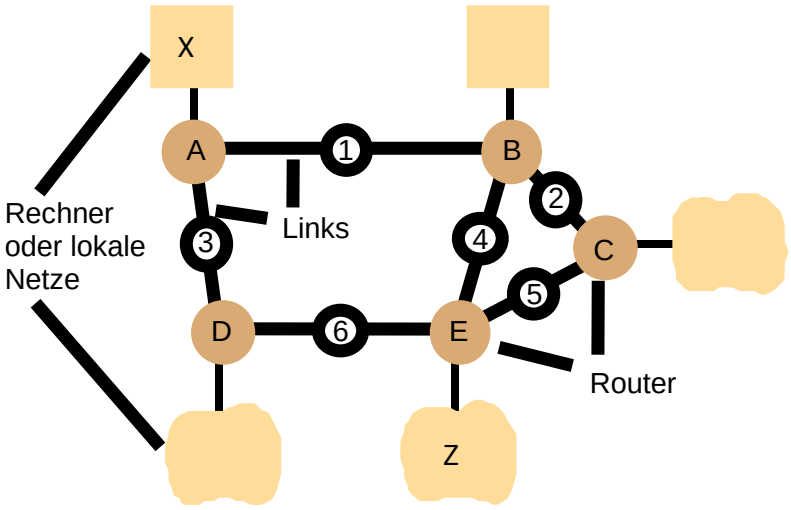


Bild 3.3 Hypothetisches Netz zur Erläuterung Routing

Aufgaben für Routing-Algorithmen:

1. Bestimmung von Routen, die Nachrichten nehmen sollen (abhängig vom Übertragungsmodus!)
2. Dynamische Aktualisierung der Information, auf Basis derer die Routenbestimmung erfolgt

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Routings from B

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

Routings from C

To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D

To	*Link*	*Cost*
A	3	1
B	3	2
C	6	2
D	local	0
E	6	1

Routings from E

To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

Bild 3.4 Routingtabellen der einzelnen Router

```
Send: Each t seconds or when Tl changes, send Tl on each non-faulty outgoing link.

Receive: Whenever a routing table Tr is received on link n:
for all rows Rr in Tr {
      if (Rr.link ≠ n) {
            Rr.cost = Rr.cost + 1;
            Rr.link = n;
            if (Rr.destination is not in Tl)
                  add Rr to Tl;
                  // add new destination to Tl
            else for all rows Rl in Tl {
                  if (Rr.destination = Rl.destination and
                        (Rr.cost < Rl.cost or Rl.link = n))
                        Rl = Rr;
                  // Rr.cost < Rl.cost : remote node has better route
                  // Rl.link = n : remote node is more authoritative
            }
      }
}
```

Bild 3.5 RIP (Routing-Information-Protocol-) Algorithmus

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	8	1
E	4	1
F	2	1

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Bild 3.6 Empfangene Routingtabelle via Link 1 (links) und resultierende Routingtabelle

Was passiert bei einem Crash der Netzwerkverbindung D-E? Dies ist in Bild 3.7 dargestellt. Der Link von D nach E wird durch ∞ als Distanz dargestellt, die Reaktion bleibt jedoch aus.

Routings from D

To	*Link*	*Cost*
A	3	1
B	3	2
C	6	2
D	local	0
E	6	∞

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Bild 3.7 Reaktion in der Routingtabelle für A (rechts) nach Übermittlung einer sehr großen Übertragungsdauer

Dies ist die größte Schwachstelle in dem Algorithmus: Nur „positive“ Meldungen werden sofort umgesetzt, bei negativen Meldungen, also z.B. dem Ausfall eines Routers oder einer Verbindungen, versucht der Algorithmus, Umwegleitungen zu finden – auch auf die Gefahr, dass diese gar nicht mehr existieren.

Der (einfache) RIP-Algorithmus, im Internet auch als Distance Vector Routing Algorithmus bezeichnet, hat eine Evolution erfahren und wurde etwa 1980 durch den Link State Routing Algorithmus abgelöst. Dieser Algorithmus läuft etwa wie folgt ab:

1. Jeder Router ermittelt seine Nachbarn (die Router, die direkt an einem Link angeschlossen sind) und bestimmt die Übertragungskosten (z.B. Laufzeit) dorthin. Dies wird in einer Tabelle zusammengefasst.
2. Die Tabelle wird an *alle* Router im Netz per Flooding übertragen.

3. Jeder Router im Netz bestimmt nun anhand der Tabellen, die ihm zur Verfügung stehen, die Router, die erreichbar sind, sowie den für ihn besten Weg dorthin (per Dijkstra-Algorithmus)

Daneben müssen noch Besonderheiten berücksichtigt werden, z.B., dass

- Router A routet alle Pakete mit unbekanntem Ziel über Link 8
- Router B routet alle Pakete mit Adresse 138.37.88.* über Link 4
- Eine Ausgewogenheit in der Nutzung der Links kann durch Wahrscheinlichkeiten hergestellt werden.

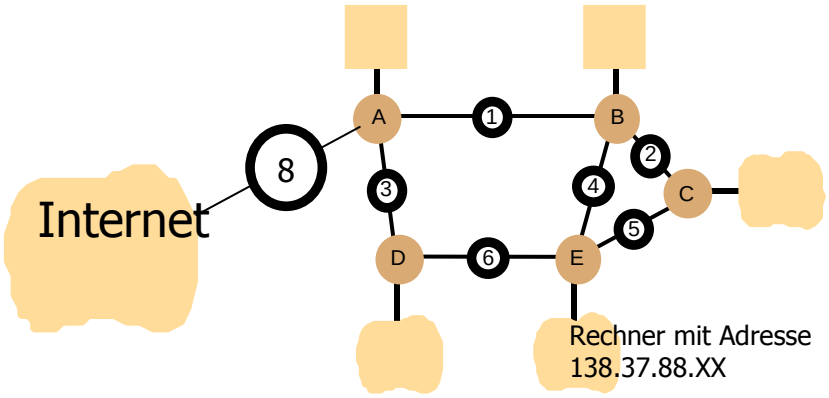


Bild 3.8 Beispielhafte Darstellung eines Teilnetzes

3.4 Adressierung im Internet Protocol

Ziele:

- Adressierung aller Rechner im Internet – Daten sollen von jedem Rechner zu jedem anderen geschickt werden können
- Effizienz in der Nutzung der Adressen

Ansatz:

- Adressen aus 4 Byte (IPv4) $\rightarrow 2^{32}$ mögliche Adressen
- Zusammenfassung von Klassen zusammengehöriger Adressen

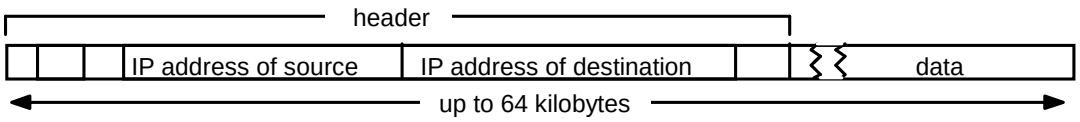


Bild 3.9 IP-Header (IP V4)

Die Verteilung der Adressen basierte auf Adressierungsklassen; diese wurden 1982 definiert.

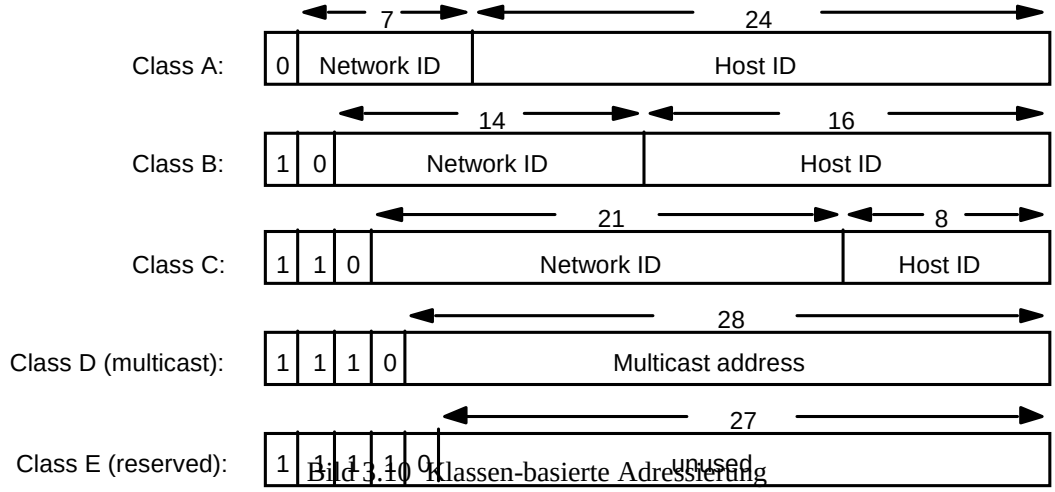


Bild 3.10 Klassen-basierte Adressierung

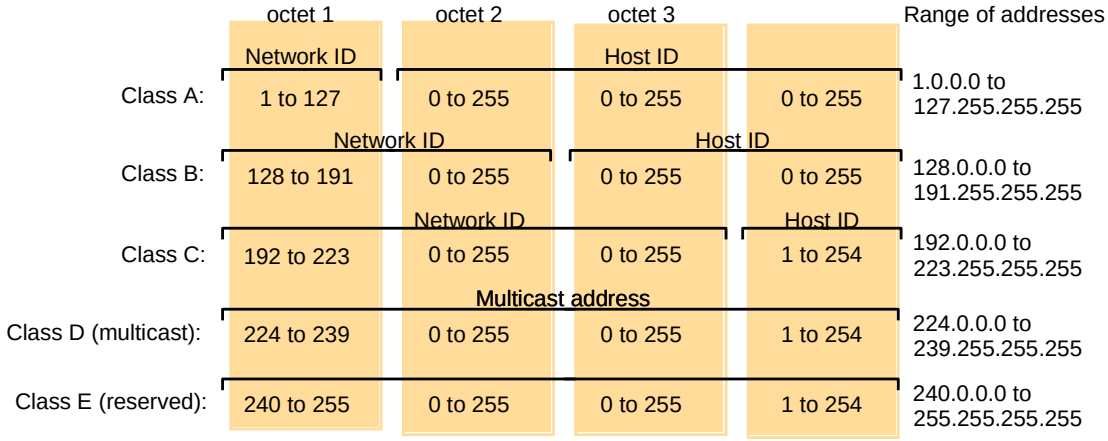


Bild 3.11 Zuteilung Adressräume zu Adressklassen

Probleme:

- Class-C Netze oft zu klein, zu wenige Class-B Netze (nur ca. 16000)
- Intelligentes Routing auf Basis der IP-Adresse schwierig
- Generell zu wenig Adressen verfügbar (Ende bereits für 1996 vorhergesagt)

Folgende Gegenmaßnahmen wurden entwickelt:

CIDR (Classless Inter-Domain Routing) seit 1996:

- Für „neue“ Netze wurde auf starre Klasseneinteilung verzichtet
- Zusammenfassung von freien (z.B. Class-C) Netzen, die in einem zusammenhängenden Adressraum liegen, zu einem größeren Netz
- Ebenfalls möglich: Aufteilung von existierenden zu großen Adressräumen
  - Beispiel: Adressraum eines Class-C Netzes wird für 3 Gruppen aus 8 Rechnern genutzt (de facto nur 6 Rechner, da eine Adresse für Netzwerk und eine für Broadcast)
- Ansatz zur Lösung: Bitmaske
  - Die ersten x Bits der 32 IP-Adressbits spezifizieren das Netzwerk (Maske)
  - Die weiteren Bits spezifizieren den Rechner
  - Beispiel für eine Gruppe: 138.37.95.232/29

- CIDR bietet auch Regionsinformation (z.B.: 194.0.0.0 bis 195.255.255.255 sind Adressen in Europa)

DHCP (Dynamic Host Configuration Protocol) und NAT (Network Address Translation):

- Nicht alle ans Internet angeschlossene Rechner benötigen eine feste Adresse.
- DHCP: *Zeitweilige* Vergabe von IP-Adressen an Rechner
- NAT: keine Vergabe einer „echten“ IP-Adresse mehr. Stattdessen wird (über DHCP) eine lokal gültige Adresse, z.B. aus dem als „für private Nutzung“ vorgesehenen Class-C Netz 192.168.0.X, vergeben.
  - Will ein Rechner nach außen kommunizieren, so ersetzt der NAT-fähige Router die IP-Senderadresse im Paket durch seine eigene und den Quellport durch einen virtuellen Port (Verweis auf Router-interne DB)
  - Empfängt der NAT-Router ein Paket von außen, so ersetzt er seine Adresse und den virtuellen Port mittels DB-Anfrage durch die lokale Adresse und den Port, und leitet das Paket ins lokale Netz weiter
- Problem: Anbieten von Serverdiensten in NAT-Netzwerken

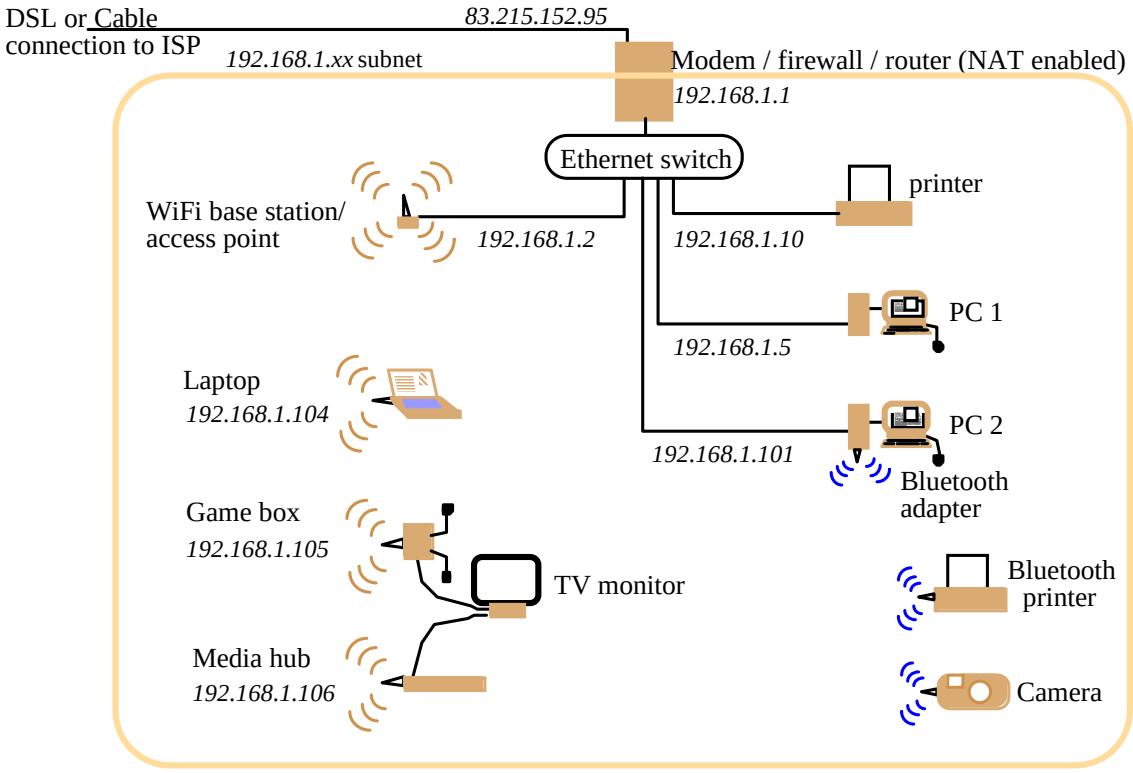


Bild 3.12 Beispiel NAT-Netzwerk

IP V6

- Adressen mit 128 Bit
- Adressraum ausreichend – bei verschwenderischer Nutzung 1000 IP-Adressen pro Quadratmeter
- Adressraum von IPv4 eingebettet (dadurch flüssige Migration möglich)
- Neue Möglichkeiten, z.B. anycast, Sicherheit auf IP-Ebene

3.5 Protokolle auf Transportschicht im Internet: TCP und UDP

- IP ist ein Protokoll auf der OSI-Schicht 3 (Network).
- IP leistet, dass Nachrichtenpakete vom Sender zu ihrem Empfänger gelangen (routing).

- IP leistet nicht: Kommunikation zwischen verschiedenen Prozessen auf Sender / Empfangsrechner!
- Dies ist Aufgabe der OSI-Schichten 4 und 5 (Transport / Session)
- Bekannteste Protokolle hier:
- UDP (User Datagram Protocol)
- TCP (Transmission Control Protocol)

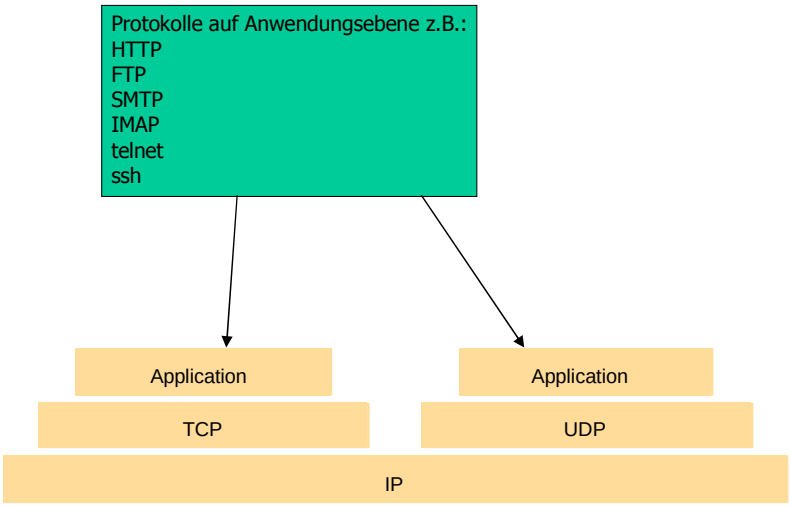


Bild 3.13 TCP/UDP aus Sicht der Programmierer

- Zentrales Konzept sowohl bei UDP als auch TCP: Ports
- Jeder Rechner hat 65536 logische Ports, Prozesse können Ports benutzen um mit Prozessen auf anderen Rechnern zu kommunizieren.
- Schreibweise IP:Port (also z.B. 138.37.94.248:80)
- Well known ports (0-1023)
  - Bekannte Standard-Services (z.B. Web, Mail, FTP, …)
  - Vergeben durch IANA
  - (siehe http://www.iana.org/assignments/port-numbers)
- Registered Ports (1024-49151)
  - Registriert durch IANA
- Freie / Dynamische Ports (49152-65536)
  - Frei für private Nutzung

Prozesse und Sockets

- Prozesse verwenden Sockets, um über Ports zu senden / empfangen
- Ein Prozess kann mehrere Sockets verwenden
- Über den gleichen Socket kann gesendet und empfangen werden
- Nur ein Socket pro Protokoll, Port und Maschine!

User Datagram Protocol:

- Verbindungslose Kommunikation zwischen Prozessen über Austausch einzelner Pakete
- Keine Anlieferungsgarantie, geringe Datengröße!
- Schnell, wenig Overhead

- Header in UDP-Paketen: Quellport, Zielport, Länge (,Checksumme)
- Wenn Checksumme ungleich 0: Empfänger vergleicht selbst berechnete Checksumme aus Daten mit empfangener. Falls diese nicht übereinstimmen, wird Paket ignoriert.

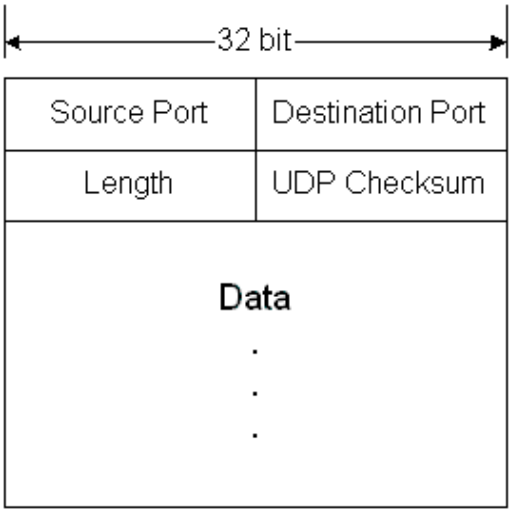


Bild 3.14 UDP Header

Transmission Control Protocol

- Verbindungsorientiert, Prozesse haben Datenstrom (Stream) zur Kommunikation
- Verlässliche Übertragung auch größerer Datenmengen, die auf mehrere IP-Pakete aufgeteilt sind.
- Mechanismen:
  - Sequenzierung von Datenpaketen: Empfänger kann Datenpakete in Buffer sortieren, bevor sie in den InputStream gehen
  - Bestätigung von Nachrichten (entweder eingebettet in Antwortnachrichten oder in separaten Bestätigungspaketen)
  - Datenflusskontrolle: Empfänger schickt zusammen mit Bestätigung eine „Window Size“ – diese gibt an, wie viele Daten der Sender bis zur nächsten Bestätigung schicken darf.
  - Wiederholte Übertragung: Wenn ein Paket nicht bestätigt wird, wird es erneut gesendet
  - Checksummen: Erkennung beschädigter Pakete, wie bei UDP
- Mehr Overhead (Verbindungsaufbau, Kontrolle, …)

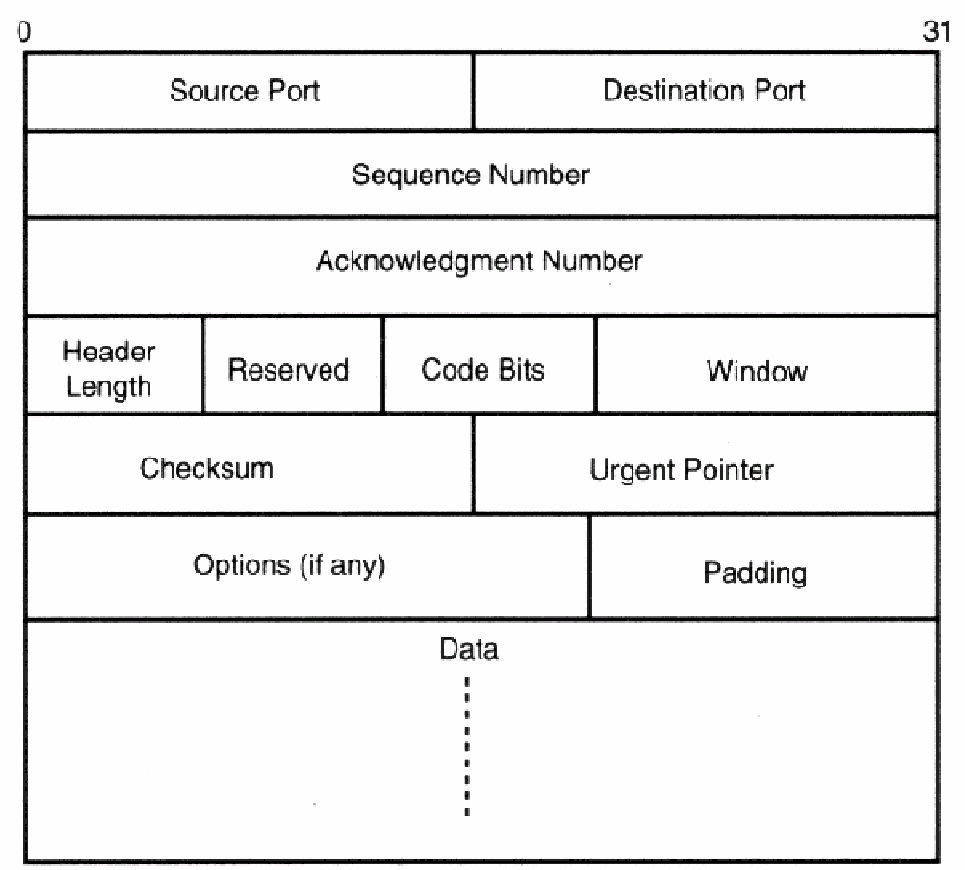


Bild 3.15 TCP-Header

3.6 Zusammenfassung: Nachrichten in TCP/UDP-IP-Netzen

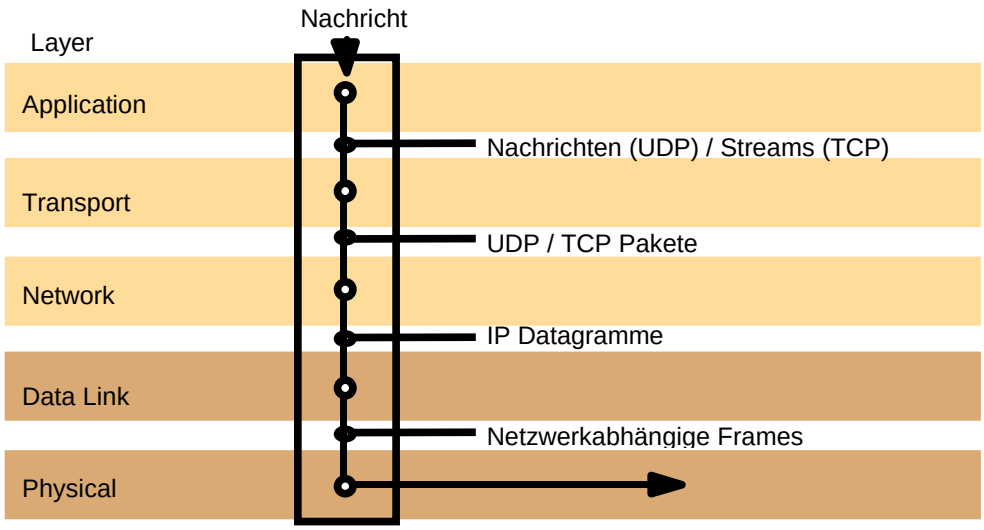


Bild 3.16 Übertragung einer Nachricht per TCP/UDP-IP

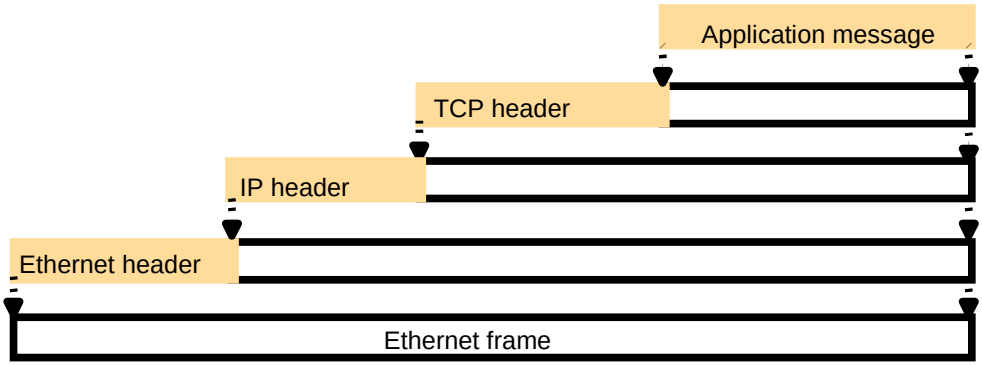


Bild 3.17 Hinzufügung von Header-Informationen bei der Übertragung

- TCP notwendig für alle Anwendungen, in denen Sicherheit, Korrektheit bzw. vollständig übertragene Daten relevant sind.
- Anwendungsgebiete von UDP: Schnelle, nicht notwendigerweise verlässliche, Kommunikation. Beispiele:
  - Voice over IP
  - Multimedia-Streaming
  - Domain Name Service (DNS)
  - RIP
  - Remote File Server (NFS)
  - Einige Anwendungen setzen eine eigene „lightweight“-Übertragungskontrolle über UDP ein

Kommunikationsmodelle in lokalen Netzen

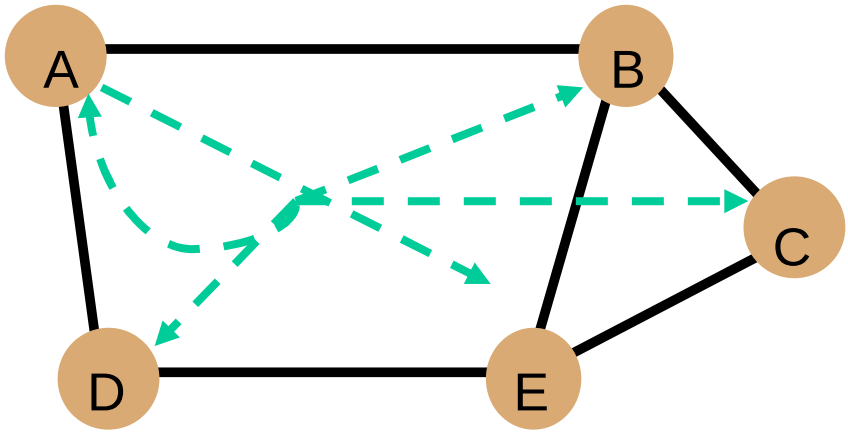


Bild 3.18 Kommunikationsmodelle in lokalen Netzen

Folgende Kommunikationsmodelle existieren:

- Unicast: Der Sender sendet an **einen** Empfängerrechner (TCP und UDP)
- Broadcast: Der Sender sendet an **alle** Rechner im lokalen Netz (über spezielle IP-Adresse für Broadcast) (nur UDP)
- Multicast: Der Sender sendet an **einige** Rechner (diese haben sich vorher zu einer Multicast-Gruppe angemeldet) (nur UDP)

3.7 Prinzipien der UDP-Kommunikation (in Java)

- Prinzip: „Zusammenbauen“ des UDP-Pakets
  - Zielport, Länge, Daten
  - Quellport wird automatisch eingefügt
- Senden des Pakets ist keine blockierende Anweisung
- Empfangen eines Pakets blockiert den Thread:
  - Ggf. Multithreading verwenden
  - Timeouts möglich

UDP-Client in Java

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
    // args give message contents and server hostname
    DatagramSocket aSocket = null;
    try {
        aSocket = new DatagramSocket();
        byte [] m = args[0].getBytes();
        InetAddress aHost = InetAddress.getByName(args[1]);
        int serverPort = 6789;
        DatagramPacket request = new DatagramPacket(m,  args[0].length(),
        aHost, serverPort);
        aSocket.send(request);
        byte[] buffer = new byte[1000];
        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(reply);
        System.out.println("Reply: " + new String(reply.getData()));
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e){System.out.println("IO: " + e.getMessage());}
    }finally{
        if( aSocket != null)
            aSocket.close();
    }
    }
}
```

UDP-Server in Java

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer,
                    buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
```

```
                request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {
            if(aSocket != null)
                aSocket.close();
        }
    }
}
```

UDP-Multicast in Java

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
    // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut = new DatagramPacket(m, m.length, group,
                6789);
            s.send(messageOut);
            // get messages from others in group
            byte[] buffer = new byte[1000];
            for(int i=0; i< 3; i++) {
                DatagramPacket messageIn =
                    new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData()));
            }
            s.leaveGroup(group);
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
        }finally {
            if(s != null) s.close();}
    }
}
```

Prinzipien der TCP-Kommunikation:

- Client erzeugt Socket und versucht den Verbindungsaufbau
- Server erzeugt ServerSocket und akzeptiert die Verbindung
- Server und Client haben danach jeweils einen DataInputStream und einen DataOutputStream, durch die die Daten fließen.
- Server kann auf einem Socket auch mehrere Verbindungen akzeptieren

TCP-Client in Java

```
import java.net.*;
import java.io.*;
```

```
public class TCPClient {
    public static void main (String args[]) {
    // arguments supply message and hostname of destination
    Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out = new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);            // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received: "+ data);
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {
            if(s!=null)
                try {s.close();
                } catch (IOException e){
                    System.out.println("close:"+e.getMessage());
                }
        }
    }
}
```

TCP-Server in Java

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
    try{
        int serverPort = 7896;
        ServerSocket listenSocket = new ServerSocket(serverPort);
        while(true) {
            Socket clientSocket = listenSocket.accept();
            Connection c = new Connection(clientSocket);
        }
    } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
        clientSocket = aClientSocket;
        in = new DataInputStream( clientSocket.getInputStream());
        out =new DataOutputStream( clientSocket.getOutputStream());
        this.start();
        } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
```

```
        try {                    // an echo server
        String data = in.readUTF();
        out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally{
            try {clientSocket.close();
            }catch (IOException e){/*close failed*/}}
    }
}
```

4 Interprozesskommunikation und verteilte Objekte

4.1 Elementare Kommunikationsprotokolle

Aufbau eigener Protokolle ist mit direkten Socketverbindungen möglich. UDP ist dabei manchmal sinnvoller als über TCP:

- Expliziter Verbindungsaufbau oft nicht notwendig,
- die TCP-Bestätigungen sind durch Antworten ggf. unnötig,
- Datenflusskontrolle ist oft überflüssig

Die Kehrseite sind die Fehlermöglichkeiten bei der Verwendung von UDP:

- Verlust einer Nachricht (Request, Reply, Acknowledge)
- Vertauschung der Reihenfolge von Anfragen
- Crash einer Maschine (Client, Server)

Daher: Mechanismen zur Fehlererkennung:

- Clients versehen Aufrufe mit *Timeouts*
- Was tun, wenn nach Timeout noch keine Antwort da ist?
  - Request nicht angekommen? Servercrash? Reply nicht angekommen?
  - Ansatz: *Wiederholtes Senden des Requests*, bis entweder „sicher“ ist dass Server nicht erreichbar ist, oder Reply erhalten wurde

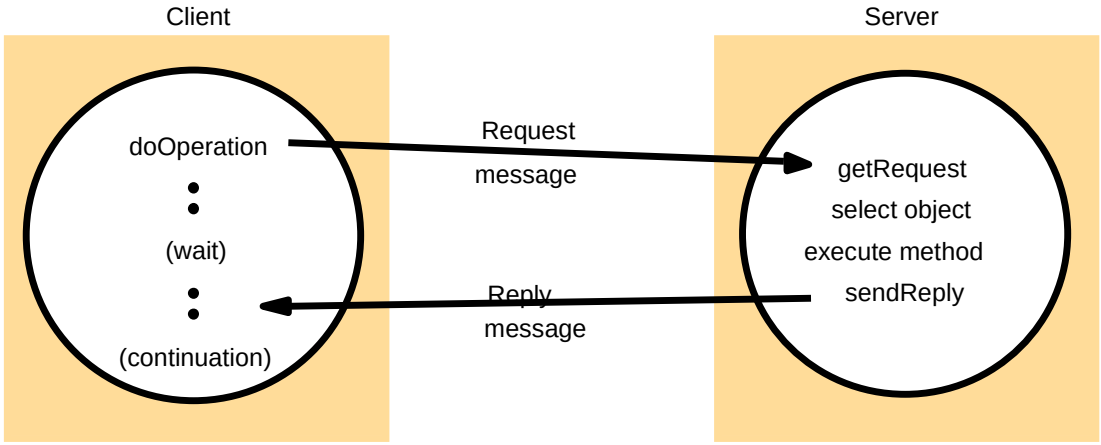


Bild 4.1 Request-Reply-Kommunikation zwischen Client und Server

- Was ist, wenn das Problem nicht am Request lag?
  - Falls ein Request mehrfach ankommt, müssen ggf. *Duplikate ignoriert* werden (außer bei idempotenten Operationen)
  - Wurde die Reply-Nachricht bereits geschickt, kann der Server mittels einer *Historie* nachvollziehen, ob der Request erfolgt war, und den Reply nochmals versenden.

4.2 Mechanismen zur Interprozesskommunikation: RPC

Grundfrage: wie können mehrere verteilte Programme miteinander kommunizieren? Hierzu sind verschiedene Möglichkeiten gegeben und in Bild 4.2 in aufsteigender Abstraktion dargestellt.

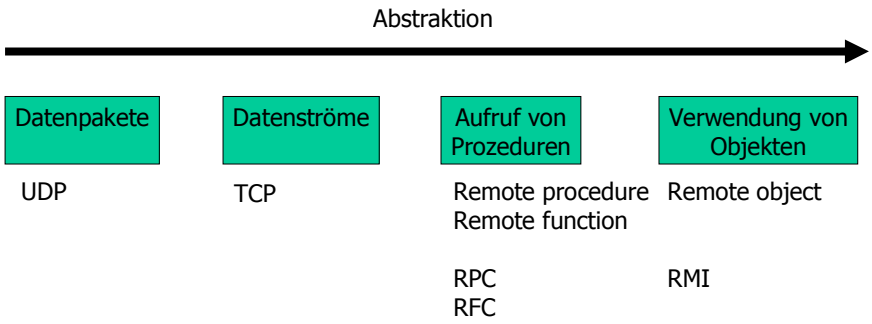


Bild 4.2 Interprozesskommunikationsformen

Oberhalb der nativen UDP/TCP-Methode existieren also Remote Procedure Call (RPC) und Remote Method Invocation (RMI):

Remote Procedure Call:

- Ziel: Ausführungstransparenz (Aufrufe entfernter Prozeduren sollten sich nicht von Aufrufen lokaler Prozeduren unterscheiden)
- 2 Grundmodelle von RPCs existieren: synchron und asynchron

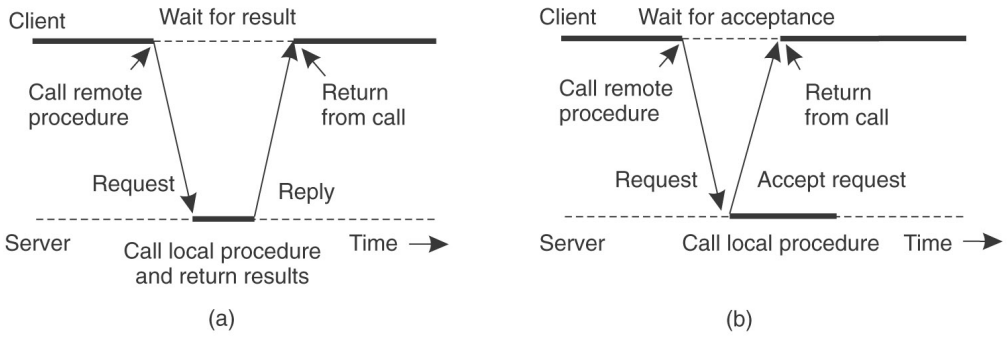


Bild 4.3 Remote Procedure Call
a) synchron (blockierend) b) asynchrony (nicht-blockierend)

Neben diesen beiden Grundmodellen kommt auch ein weiteres Modell zur Anwendung, u.a. als *verzögerter synchroner RPC* bezeichnet:

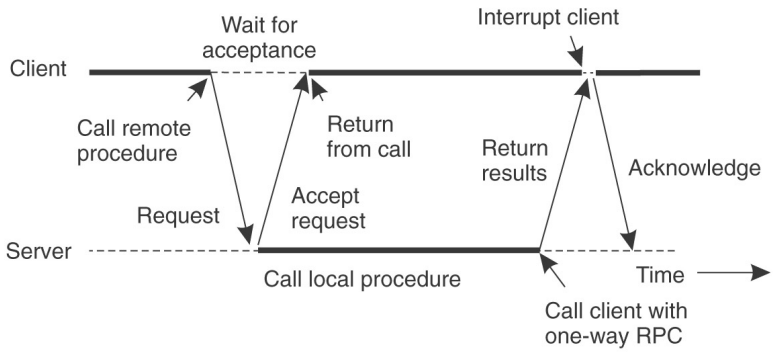


Bild 4.4 Remote Procedure Call mit Rückmeldung (Einweg-RPC), auch als verzögerter synchroner RPC bezeichnet

Aufrufsemantiken von RPC:

- Lokale Prozeduren werden nach Aufruf genau einmal ausgeführt (*exactly-once Semantik*)
- Für entfernte Prozeduren ist dies nicht erreichbar (im Sinn von garantierbar, weil die Verbindung nicht sicher ist)
- Alternativen:
  - *Maybe Semantik* (ok für Anwendungen, in denen der Verlust von Aufrufen unproblematisch ist)
  - *At-least-once Semantik* (gut für idempotente Operationen)
  - *At-most-once Semantik* (gut für die meisten Anwendungen)

Mechanismus			*RPC-Semantik*
Wiederholtes Senden	*Filtern von Duplikaten*	*Wiederausführung oder Historie*	
Nein	N/A	N/A	*Maybe*
Ja	Nein	Wiederausführung	*At-least-once*
Ja	Ja	Historie	*At-most-once*

Tabelle 4.1 Zusammenfassung der verschiedenen Semantiken für RPCs

Anmerkung: Wenn eine Anfrage ohne Schaden oder Veränderung mehrfach gestellt werden kann, wird sie als *idempotent* bezeichnet. Beispiel: Abfrage eines Kontostandes (idempotent) versus Überweisung (nicht idempotent).

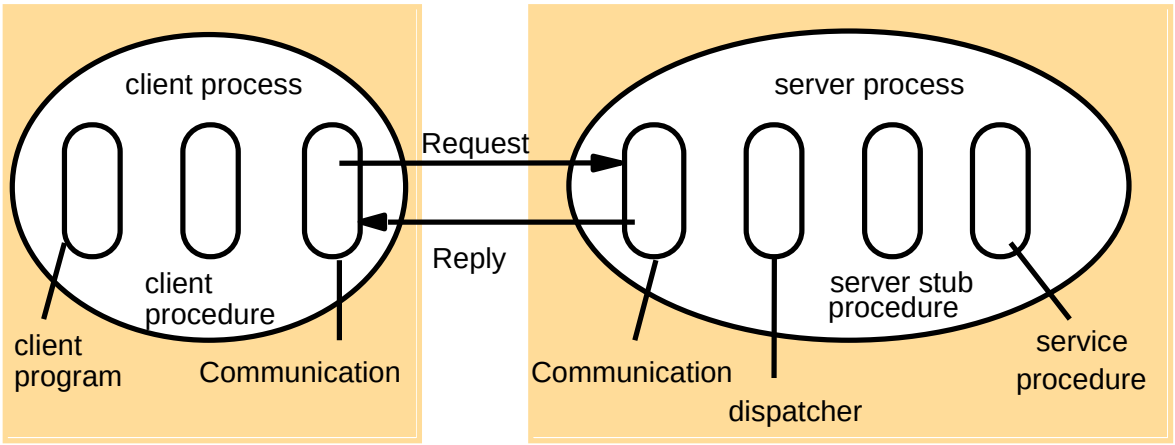


Bild 4.5 Schema eines Remote Procedure Calls

„Stubs“ (Stümpfe): Diese dienen der Vermittlung von RPCs zwischen Programm und Kommunikationssystem. Die einzelnen Stubs haben folgende Aufgaben:

Client Stub:

- Wandelt Aufrufdaten in definiertes Format um (Token Stream)
- Schreibt diese nach out

Dies wird als *Marshalling* bezeichnet.

Server Stub:

- Liest Daten von in, wandelt Token Stream in Parameter der Prozedur um

- Leitet Aufruf an Prozedur weiter
- Gibt Ergebnis an Client Stub zurück (wiederum mittels Token Stream)

Dies wird als *Unmarshalling* bezeichnet

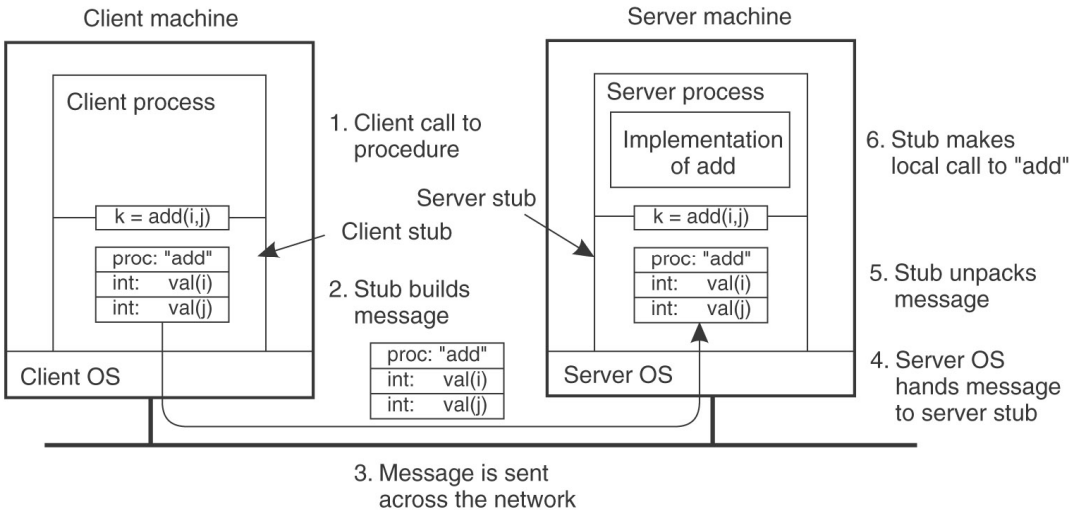


Bild 4.6 Prinzip des Marshalling/Unmarshalling bei RPCs

Client:

```
x=4; y=3;
f(x,y);
print(x);
```

Server:

```
procedure f(a,b):
  a = a-b;
```

Allgemeine Prinzipien der Parameterübergabe:

- *Call-by-value* (Parameter werden kopiert, hierdurch entstehen keine Seiteneffekte)
- *Call-by-reference* (Zeiger auf gemeinsame Parameter)

Call-by-value hat das Problem, dass hierdurch keine Daten dauerhaft geändert warden, falls dies notwendig ist. Das wesentliche Problem für call-by-reference hingegen ist, dass dies einen gemeinsamen Adressraum voraussetzt, was bei einem verteilten System nicht gegeben ist. Als Kompromiss bietet sich das häufig verwendetete Prinzip *call-by-value-result* an.

- Lokale Variable (a) übernimmt bei Prozeduraufruf den Wert der übergebenen Variablen (x)
- Änderungen an der lokalen Variable a bewirken keine Änderung des Wertes von x
- Bei Verlassen der Prozedur erhält x den letzten Wert der lokalen Variable a.
- Verfahren ähnlich zu call-by-reference, wobei das Problem des getrennten Adressraums überwunden wird.

4.3 Remote Method Invocation (RMI)

RMI verwendet folgende Basistechniken der Objektorientierten Programmierung:

- Kapselung
- Vererbung (Klassen / Schnittstellen)
- Nachrichtenaustausch

Die ersten beiden Punkte stellen das Remote Interface dar, Punkt 3 wird wieder auf Request/Reply abgebildet.

- Hauptproblem beim entfernten Aufruf von Objektmethoden:
  „Objektparameter“ sind Referenzen
  „call-by-reference“ Problem
  Problem der Serialisierung von Objekten
- Ausführungstransparenz bei RPCs:
  Aufrufe entfernter Prozeduren sollen sich nicht von Aufrufen lokaler Prozeduren unterscheiden.
- Ausführungstransparenz für RMI in OO-Sprachen:
  Der Zugang zu entfernten Objekten (z.B. die Verwendung von Methoden) sollte sich nicht vom Zugang zu lokalen Objekten unterscheiden.

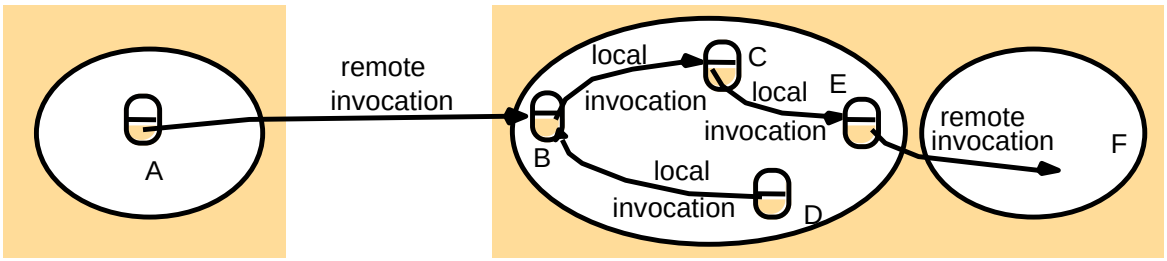


Bild 4.7 Remote Method Invocation

Zentrale Fragen:

- Wie stellen Objekte ihre Schnittstellen für entfernten Zugriff bereit?
- Wie können Objekte auf entfernte Objekte zugreifen?

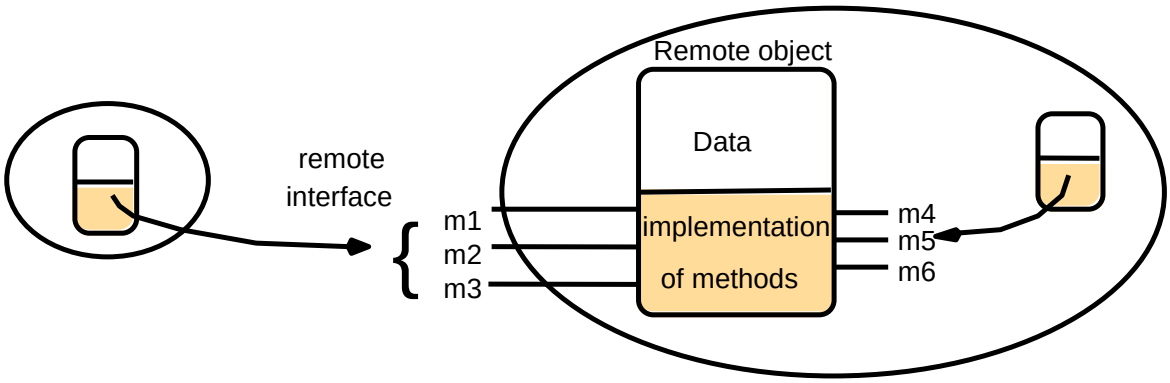


Bild 4.8 Remote/local Interface

Methoden m1,m2,m3 sind für entfernten Zugriff, Methoden m4,m5,m6 sind für lokalen Zugriff.

Typisches Muster: Methoden für entfernten Zugriff sind auch lokal nutzbar

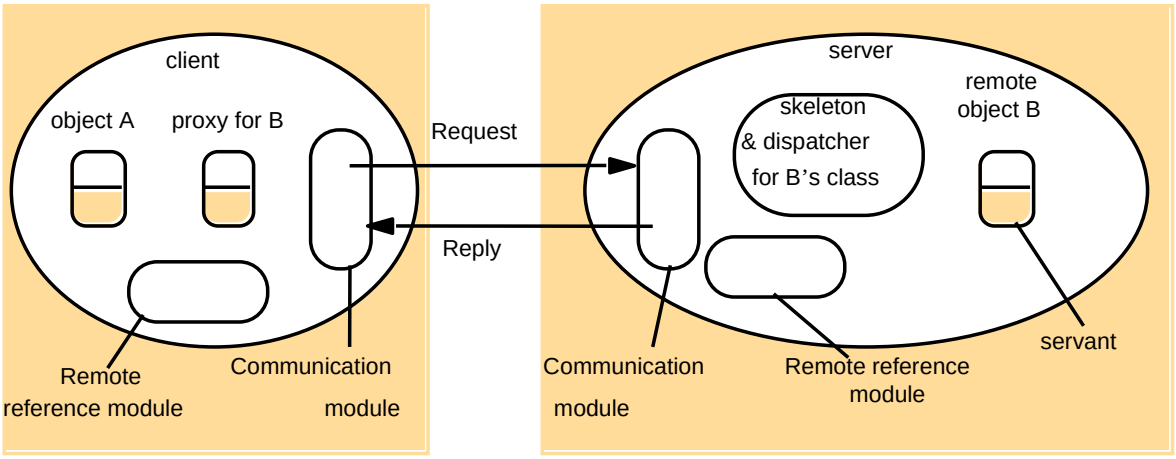


Bild 4.9 RMI-Basismodell für entfernten Zugriff auf Objekte

Communication Module:

- verantwortlich für Request/Reply Nachrichten
- legt Aufrufsemantik fest (*at-most-once* für viele RMI-Implementierungen)

Remote Reference Module:

- Tabelle mit Referenzen auf 1) lokale entfernt nutzbare Objekte und 2) entfernte Objekte, die lokal genutzt werden
- erstellt Proxies

Clientseitiger Proxy / Stub:

- Lokaler Ersatz für serverseitiges Objekt (Ausführungstransparenz!)
- Stellt Programmierschnittstelle (Remote Interface) bereit
- Aufruf einer Methode = Marshalling und Weiterleitung eines Requests (requestId, objectId, methodId, arguments)

Serverseitiger Dispatcher (einer pro Remote-Klasse):

- Verteilt die Requests (requestId, objectId, methodId, arguments) an Skeletons

Serverseitiges Skeleton (eines pro Remote-Klasse):

- Unmarshalling der Methodenparameter und Weiterleitung an Servant

Servant:

- Stellt eigentliche Funktionalität bereit (Resultat dann über Skeleton an Client)

4.4 Java RMI

Beispielanwendung: „Shared Whiteboard“. Server hält Liste mit (geometrischen) Formen (z.B. Linien, Kreise, etc), Clients können Liste aktualisieren und abrufen.

Remote Interfaces:

```
import java.rmi.*;
import java.util.Vector;
public interface ShapeList extends Remote {
       Shape newShape(GraphicalObject g) throws RemoteException;
       Vector allShapes() throws RemoteException;
       int getVersion() throws RemoteException;
}
```

```
public interface Shape extends Remote {
       int getVersion() throws RemoteException;
       GraphicalObject  getAllState() throws RemoteException;
}
```

Prinzipien der Objektserialisierung: Werden Objekte als Parameter verwendet (in Methodenaufrufen), so gelten folgende Regeln:

- Remote Objects (d.h., Instanzen von Klassen, die Remote-Interfaces deklarieren) werden als Referenz auf entfernte Objekte serialisiert, also mittels Proxies und Remote Object Module.
- Non-Remote Objects, die das Interface java.io.Serializable implementieren, werden kopiert und mittels call-by-value verwendet.
- Non-Remote Objects, die nicht das Interface java.io.Serializable implementieren, können nicht genutzt werden (→ Exception)

Welche Konsequenzen ergeben sich daraus?

Beispiel:

```
public class Point implements Serializable {
  private String id;
  private int x;
  private int y;
// Konstruktoren und Methoden
}
```

- Objekte werden serialisiert, indem die Werte aller Instanzvariablen (d.h. der Zustand des Objekts) übertragen werden
- Hat ein Objekt Instanzvariablen vom Objekttyp, so wird dies rekursiv durchgeführt.
- Ausnahme: Deklaration von Variablen als transient.
- Point p = new Point("id1",100,200);

				Erklärung
Point	8-byte version number			*Klassenname, Version*
3	int x	int y	java.lang.String id	*Anzahl und Typ der Instanzvariablen*
	100	200	id1	*Werte der Instanzvars*

Tabelle 4.2 Prinzip der Serialisierung von Java-Objekten

- Marshalling / Unmarshalling über Java Reflection

Muss ein Client die Klassen kennen, die er vom Server verwendet?

- Im Prinzip ja – Clients können auf serialisierten Objektkopien beliebig arbeiten, insbesondere auch Methoden ausführen
- Anforderung, dass Server und Clients die gleichen Klassen besitzen, ist aber eine starke Einschränkung (z.B. könnte kein Client dann neue GraphicalObjects definieren und an den Server schicken)
- Lösung: dynamisches Übertragung von Klassen (in RMI möglich)
- Wird eine Klasse verwendet, deren Code lokal nicht vorliegt, so wird diese übertragen
  - Remote Objects: Proxy-Code
  - Non-Remote Objects: Code der Klasse

- Serverseitiger Prozess, bei dem Remote Objects registriert werden (dient als Verzeichnis für clientseitige Remote Object Modules)
- Prinzip:
  - Serverprozess registriert Objekt
  - Clientprozess sucht Objekt (lookup)
  - Clientprozess erhält Objektreferenz
  - Client kann entfernte Aufrufe machen
  - Serverprozess löscht Objektregistrierung

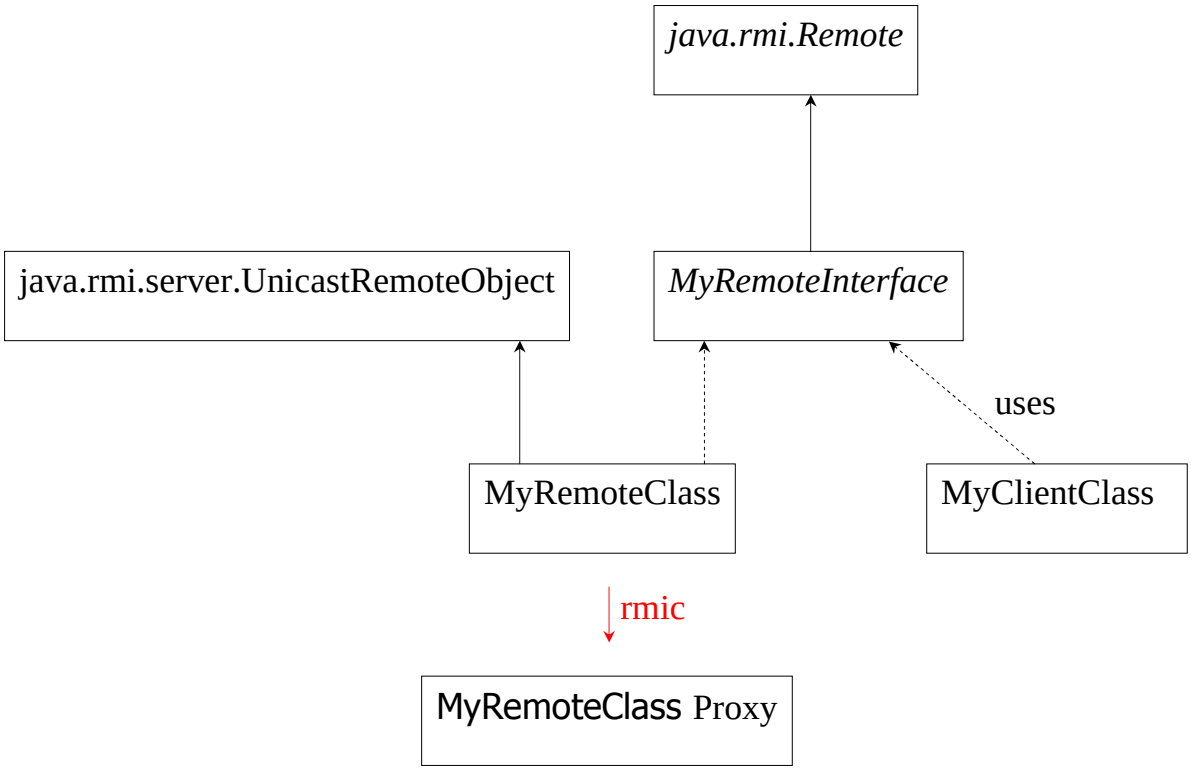


Bild 4.10 Zusammenhang der wichtigsten RMI-Klassen

Beispielcode für Server (Naming)

```
import java.rmi.*;
public class ShapeListServer{
       public static void main(String args[]){
              System.setSecurityManager(new RMISecurityManager());
               try{
                     ShapeList aShapeList = new ShapeListServant();
                     Naming.rebind("Shape List", aShapeList );
                     System.out.println("ShapeList server ready");
                     }catch(Exception e) {
                     System.out.println("ShapeList server main " + e.getMessage());}
       }
}
```

Beispielcode Server :

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
```

```
        private Vector theList; // contains the list of Shapes
        private int version;
        public ShapeListServant() throws RemoteException{...}
        public Shape newShape(GraphicalObject g) throws RemoteException {
               version++;
               Shape s = new ShapeServant(g, version);
               theList.addElement(s);
               return s;
        }
        public Vector allShapes() throws RemoteException{...}
        public int getVersion() throws RemoteException { ... }
}
```

Beispielcode Client:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
       System.setSecurityManager(new RMISecurityManager());
       ShapeList aShapeList = null;
       try{
              aShapeList  = (ShapeList) Naming.lookup("//servername.Shape List");

              Vector sList = aShapeList.allShapes();
       } catch(RemoteException e) {System.out.println(e.getMessage());
       } catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

In der gezeigten Lösung muss ein Client wiederholt beim Server nachfragen, um den aktuellen Stand der Anwendung abzufragen und seinen eigenen lokalen Datenstand anzupassen. Dies hat Nachteile (Effizienz, Konsistenz).

Welche Alternativen gibt es?

- Callbacks: der Server informiert alle registrierten Clients
  - Der Client erzeugt ein Remote Object, welches eine für den Server aufrufbare Methode (z.B. update) enthält.
  - Der Server bietet eine Methode an, mit der Clients Referenzen zu ihren Remote Objects in eine serverseitige Liste eintragen können.
  - Bei Aktualisierung der Daten ruft der Server die update-Methode auf allen registrierten Remote Objects auf.
- Nachteil des Verfahrens: Clients haben Serverfunktionalität, Server muss Serie von RMI-Aufrufen bei Clients machen (problematisch, wenn Clients ihr Verlassen nicht melden)

4.5 Plattform-unabhängige RMI-Ansätze

OMG (Object Management Group) existiert seit 1989:

„OMG™ is an international, open membership, not-for-profit computer industry consortium. OMG Task Forces develop enterprise integration standards for a wide range of technologies, and an even wider range of industries. OMG's modeling standards enable powerful visual design, execution and maintenance of software and other processes. OMG's middleware standards and profiles are based on the Common Object Request Broker Architecture (CORBA®) and support a wide variety of industries." (von www.omg.org)

CORBA Features sind:

- Ortstransparenz für verteilte Objekte
- Unabhängigkeit von Programmiersprache und Betriebssystem
- Interoperabilität von Komponenten
- Portierbarkeit von Code (Unabhängigkeit von konkreten CORBA-Implementierungen)
- CORBA Services (z.B. Events, Security, …)

CORBA Kernkomponenten:

- Objektmodell
- Sprache zur Beschreibung von Remote Interfaces (IDL)
- Systemarchitektur
- Kommunikationsprotokolle (GIOP, IIOP)

CORBA-Objektmodell

- Modell für „Remote Objects“ ähnlich wie bei Java RMI
- CORBA setzt keine OO-Sprache beim Client voraus! Jedes Programm, das Requests an Remote Objects schicken kann (und Replies empfangen kann), kann CORBA Client sein.
- CORBA-Objekt = Remote Object
- Keine Klassen!
- Auch auf dem Server keine OO-Sprache Voraussetzung: CORBA-Objekte können von allen Programmiersprachen umgesetzt werden
- Definition von CORBA-Objekten (= Remote Interfaces) in spezieller Sprache IDL (umfasst Methoden und Datenstrukturen)

IDL: Interface Definition Language:

IDL-Schnittstellen beschreiben CORBA-Objekte sowie zugehörige Datenstrukturen und Methoden.

```
struct Rectangle{
       long width;
       long height;
       long x;
       long y;
} ;

struct GraphicalObject {
       string type;
       Rectangle enclosing;
       boolean isFilled;
};

interface Shape {
       long getVersion() ;
       GraphicalObject getAllState() ; // returns state of the GraphicalObject
};

typedef sequence <Shape, 100> All;
interface ShapeList {
       exception FullException{ };
       Shape newShape(in GraphicalObject g) raises (FullException);
       All allShapes();  // returns sequence of remote object references
       long getVersion() ;
};
```

- Parameterübergabe:
  - CORBA-Objekte (=Parameter, die durch IDL-Interface deklariert werden) werden als Remote reference übergeben.
  - Primitive Typen und Strukturen werden serialisiert und als Wert übergeben
- Aufrufsemantik:
  - At-most once
  - Schlüsselwort oneway bei Methoden → maybe-Semantik
- Interfaces erlauben Mehrfachvererbung
- Attribute möglich (Konvertierung zu get/set Methoden)
- IDL-Interfaces werden durch spezielle Compiler in Programmiersprachen übersetzt

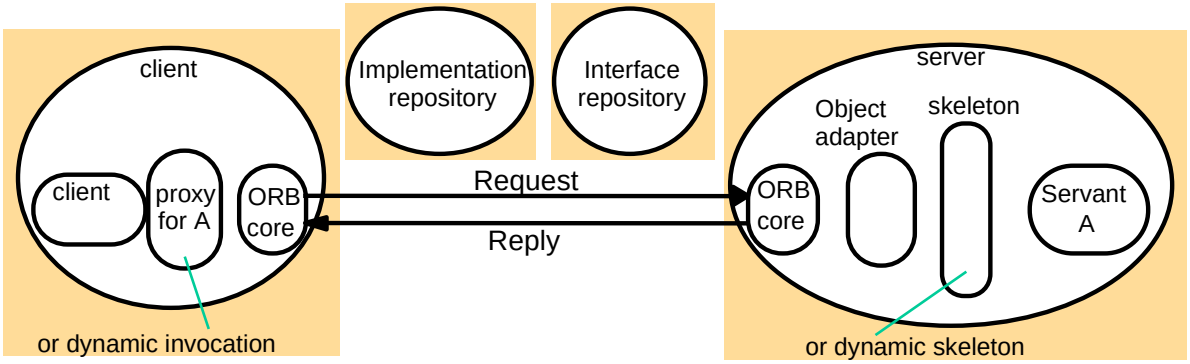


Bild 4.11 CORBA-Architektur

ORB (Object Request Broker) core:

- Kommunikationsmodul, verwendet *GIOP* (*General Inter-ORB Protocol*), ein abstraktes Kommunikationsprotokoll
- Konvertierung zwischen Remote References und Strings

Object Adapter:

- Erzeugung von Remote Object References
- Verteilung von RMI-Aufrufen an Skeletons
- Aktivierung und Deaktivierung von Servants

POA (Portable Object Adapter): neuer Standard seit CORBA 2.2

- Unabhängigkeit von verschiedenen ORB‘s
- Trennung von CORBA-Objekten und Servants
- Spezifikationen (z.B. neuer Thread pro Aufruf?)

Skeletons und Proxys:

- Rolle wie in Java RMI (Marshalling / Unmarshalling, Weiterleitung)
- Skeleton in Server-Sprache, Proxy in Client-Sprache (Generiert aus IDL)!

Implementation Repository:

- Verzeichnis aller CORBA-Server mit deren Diensten (= CORBA-Objekten)
- Dynamische Aktivierung von Servern
- Nicht jeder Dienst *muss* eingetragen werden!

Interface Repository:

- Verzeichnis aller IDLs
- Möglichkeit, Dienste dynamisch zu verwenden (CORBA Reflection): Clients fragen im IR nach angebotenen Diensten und nutzen diese Information für Requests (kein Code-Downloading)
- IR nicht notwendig für statischen Aufruf, nicht von jedem ORB vorgesehen

Dynamische Skeletons:

- Ermöglicht Hinzufügen von CORBA-Objekten zu Servern zur Laufzeit
- Analog zu Java RMI: Request wird über Reflection untersucht und an Servants verteilt

Mit dem IDL-Interface erstellte Java Interfaces aus CORBA Interface ShapeList:

```
public interface ShapeListOperations {
       Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
       Shape[] allShapes();
       int getVersion();
}

public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity { }
```

Zusätzlich: Klassen für structs (GraphicalObject, Rectangle), Server Skeleton (ShapeListPOA), Client Stubs (_ShapeListStub) und einige Hilfsklassen (Konvertierung CORBA-Objekt ↔ Java Object)

```
ShapeListServant
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
       private POA theRootpoa;
       private Shape theList[];
       private int version;
       private static int n=0;
       public ShapeListServant(POA rootpoa){
              theRootpoa =rootpoa;
              // initialize the other instance variables
       }

public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {
       version++;
       Shape s = null;
       ShapeServant shapeRef = new ShapeServant(g, version);
       try {
              org.omg.CORBA.Object ref = theRoopoa.servant_to_reference(shapeRef);
              s = ShapeHelper.narrow(ref);
       } catch (Exception e) {}
       if( n >=100 ) throw new ShapeListPackage.FullException();
       theList[n++] = s;
       return s;
}

public  Shape[] allShapes() { ... }
public int getVersion() { ... }

Server (main):
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
```

```
import org.omg.PortableServer.*;
public class ShapeListServer {
       public static void main(String args[]) {
           try{
             ORB orb = ORB.init(args, null);
             POA rootpoa = POAHelper.narrow(
                         orb.resolve_initial_references("RootPOA"));
             rootpoa.the_POAManager().activate();
             ShapeListServant SLSRef = new ShapeListServant(rootpoa);
             org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef);
             ShapeList SLRef = ShapeListHelper.narrow(ref);
             org.omg.CORBA.Object objRef = orb.resolve_initial_references(
                                    "NameService");
             NamingContext ncRef = NamingContextHelper.narrow(objRef);
             NameComponent nc = new NameComponent("ShapeList", "");
             NameComponent path[] = {nc};
             ncRef.rebind(path, SLRef);
             orb.run();
           } catch (Exception e) { ... }
       }
}

Client:
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
       public static void main(String args[]) {
           try{
             ORB orb = ORB.init(args, null);
             org.omg.CORBA.Object objRef = orb.resolve_initial_references(
                                    "NameService");
             NamingContext ncRef = NamingContextHelper.narrow(objRef);
             NameComponent nc = new NameComponent("ShapeList", "");
             NameComponent path [] = { nc };
             ShapeList shapeListRef = ShapeListHelper.narrow(ncRef.resolve(path));
             Shape[] sList = shapeListRef.allShapes();
             GraphicalObject g = sList[0].getAllState();
           } catch(org.omg.CORBA.SystemException e) {...}
       }
}
```

- Naming ist zentraler Service in CORBA
- Weitere Dienste (optional, je nach ORB-Anbieter) sind vorhanden (siehe Tabelle 4.3)
- Zentrales Argument für CORBA: Fülle von relevanten Diensten + Unabhängigkeit von Programmiersprache

Service	Description
Collection	Facilities for grouping objects into lists, queue, sets, etc.
Query	Facilities for querying collections of objects in a declarative manner
Concurrency	Facilities to allow concurrent access to shared objects
Transaction	Flat and nested transactions on method calls over multiple objects
Event	Facilities for asynchronous communication through events
Notification	Advanced facilities for event-based asynchronous communication
Externalization	Facilities for marshaling and unmarshaling of objects
Life cycle	Facilities for creation, deletion, copying, and moving of objects
Licensing	Facilities for attaching a license to an object
Naming	Facilities for systemwide naming of objects
Property	Facilities for associating (attribute, value) pairs with objects
Trading	Facilities to publish and find the services an object has to offer
Persistence	Facilities for persistently storing objects
Relationship	Facilities for expressing relationships between objects
Security	Mechanisms for secure channels, authorization, and auditing
Time	Provides the current time within specified error margins

5 Peer-To-Peer-Systeme

Im Kapitel 4 (Interprozesskommunikation) war die zentrale Frage, wie verteilte Prozesse miteinander kommunizieren. Die Antworten lauteten (in aufsteigender Abstraktion):

- Pakete
- Streams
- Prozeduren/Funktionen
- Objekte/Methoden

Der Aufbau größerer Systeme und Anwendungen auf dieser Basis kann nun auf mehreren Paradigmen beruhen:

- Client/Server-Architektur (evtl. Multi-Tier, Mehrebenen-System): Hier bestehen die wesentlichen Herausforderungen in der *Skalierung* und der *Ausfallsicherheit*.
- Peer-to-Peer-Architektur: Herausforderungen hier sind das *Finden von Ressourcen* und die immanente *dynamische Rekonfiguration*
- Mischformen

Dieses Kapitel befasst sich mit den Basisalgorithmen von Peer-To-Peer-(P2P-)Systemen.

5.1 Definition von Peer-To-Peer-Systemen

Einige „Definitionen“:

- Ein P2P-System ist ein dezentrales System mit gleichartigen Anwendungen (= Peers).
- „Anwendungen, die Ressourcen an den ‚Enden des Internets‘ ausschöpfen – Speicher, Prozessorleistung, Inhalte, menschliche Präsenz“ (Shirky 2000)

Charakteristika von P2P-Anwendungen:

- Sicherstellung, dass alle Benutzer zum System beitragen
- Alle Teilnehmer in einem P2P-System haben die gleiche Rolle (auch wenn sie ggf. unterschiedliche Ressourcen beitragen)
- Die korrekte Funktion hängt nicht von der Existenz eines zentral administrierten Systems ab
- Anbieter und Nutzer des Systems können einen gewissen Grad an Anonymität erreichen (was allerdings nicht in jedem P2P-System gewollt ist)
- Kernfrage: Wahl der Algorithmen zur Speicherung und zum Auffinden von Daten. Die Ziele hierbei sind: Verfügbarkeit, Skalierbarkeit, Effizienz

1999: „Original Napster“:

Erstes P2P-Filesharingsystem, das Bedarf und Machbarkeit dezentraler Ansätze für (Musik-)Downloads zeigte. Es erzeugte allerdings eine weltweite Debatte über Urheberrecht in P2P-Systemen:

- Inhalte konnten illegal weitergegeben werden
- Waren die Betreiber von Napster dafür verantwortlich?
- Die juristische Antwort war „Ja“, wegen des zentral verwalteten Index-Dienstes (US-Urteil, 2001)

Als Nachfolge wurde Napster als kostenpflichtiger Dienst (mit komplett anderer Systemarchitektur) eingeführt.

Andere P2P-Filesharing-Ansätze sind Freenet, Kazaa, Gnutella, BitTorrent (mehr Anonymität, bessere Skalierung)

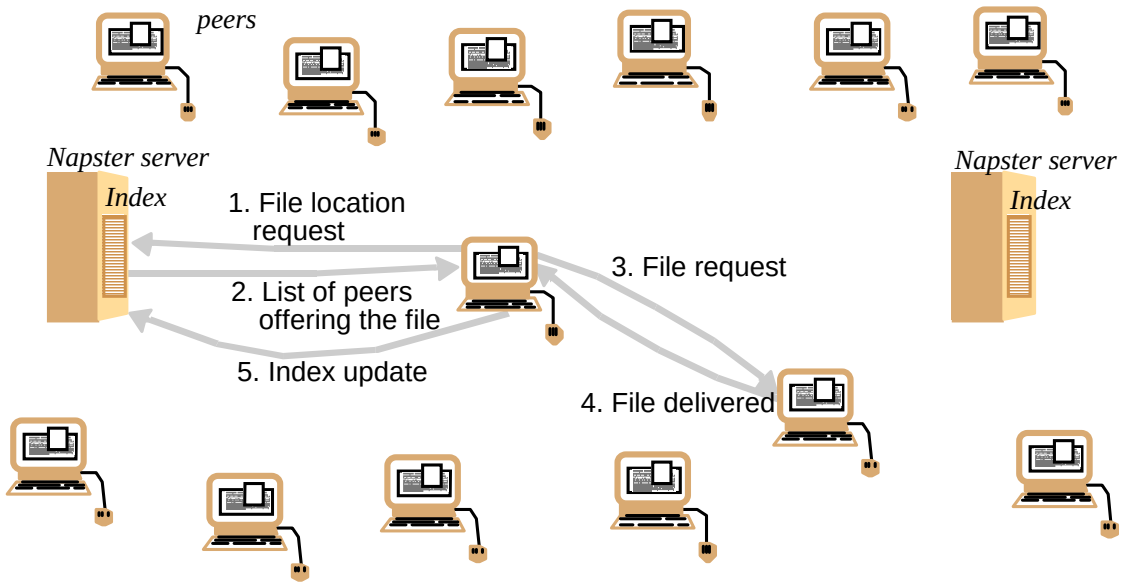


Bild 5.1 Napster-Architektur (1999)

Seti@home

SETI (Search for Extraterrestrial Intelligence) ist ein wissenschaftliches Gebiet, dessen Ziel in der Erkennung intelligenten Lebens außerhalb der Erde besteht. Ein Ansatz – mit Radio-SETI bezeichnet – nutzt die Aufnahmen von Radio-Teleskopen und versucht, darin schmalbandige Radiosignale aus dem All stammend zu identifizieren. Da diese Radiosignale aufgrund ihrer Schmalbandigkeit nicht aus natürlichen Ursachen stammen können, würde eine Erkennung einen starken Hinweis auf extraterristrische Technologie – mithin Intelligenz genannt – liefern.

Bild 5.2 Ausschnitt seti@home-Homepage (www.gridrepublic.org)

Hier wurde das Prinzip der Nutzung „freier CPU-Zyklen“ zum Wohl der Allgemeinheit genutzt.

5.2 Human Computation

Ziele von “Human Computation”, hier für ESP (Extra Sensory Perception):

“When people play the game they help determine the contents of images by providing meaningful labels for them. If the game is played as much as popular online games, we estimate that most images on the Web can be labeled in a few months.

Having proper labels associated with each image on the Web would allow for more accurate image search, improve the accessibility of sites (by providing descriptions of images to visually impaired individuals), and help users block inappropriate images.”

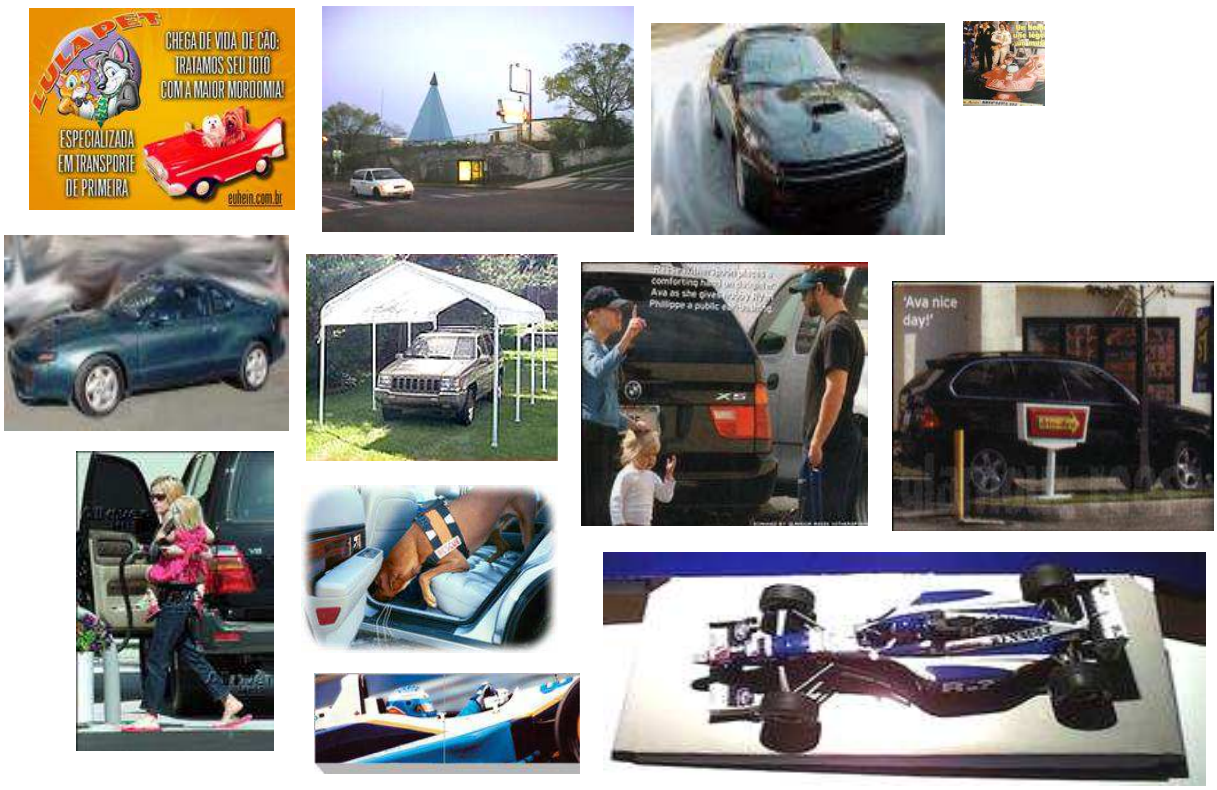


Bild 5.3 Beispielbilder mit gleichem Label

5.3 Die “nächste” Generation von P2P-Systemen

2P2 Middleware: Allgemeine Rahmensysteme zum Management verteilter Ressourcen in P2P-Systemen (Platzierung, Suche). Zentrale Anforderungen sind:

- Globale Skalierbarkeit
- Load Balancing
- Optimierung lokaler Interaktion
- Anpassung an dynamische Verfügbarkeit von (physikalischen) Ressourcen
- Datensicherheit in Umgebungen mit heterogener Vertrauenswürdigkeit
- Anonymität und Resistenz gegen Zensur

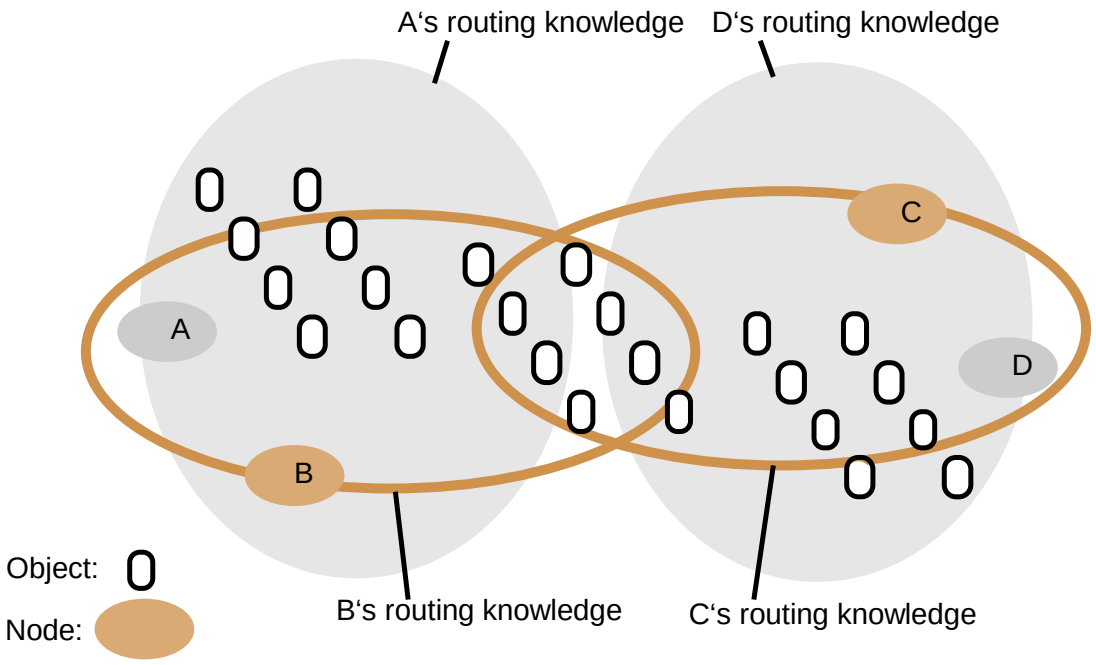


Bild 5.4 Zum Problem der Verteilung von Informationen im P2P-Netz

5.3.1 Routing in P2P-Netzen

Das Routing-Prinzip in P2P-Systemen besteht in der global eindeutigen Benennung von Objekten durch GUIDs (Gloabl User-IDs). Typischerweise gilt:

- Nodes (Rechner) und Objects haben 128 Bit GUIDs
- Berechnung der GUIDs durch sichere Hashfunktion (im Sinne einer Gleichverteilung, also kein Hinweis auf Ursprung)

Die Ziele dieses Overlay Routing sind:

- **Finden von Objekten per GUID, Ausführung von Operationen auf diesen Objekten (ggf. Änderung des Objekts), dies insbesondere unter dem Aspekt des Load Balancing**
- Publizieren von Objekten mit GUIDs
- Entfernen von Objekten unter Angabe der GUID
- Anpassung an neu hinzukommende und nicht mehr verfügbare Nodes

	IP	*Application-level routing overlay*
Scale	IPv4 is limited to 2^{32} addressable nodes. The IPv6 name space is much more generous (2^{128}), but addresses in both versions are hierarchically structured and much of the space is pre-allocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID name space is very large and flat (>2^{128}), allowing it to be much more fully occupied.
Load balancing	Loads on routers are determined by network topology and associated traffic patterns.	Object locations can be randomized and hence traffic patterns are divorced from the network topology.
Network dynamics (addition/deletion of objects/nodes)	IP routing tables are updated asynchronously on a best-efforts basis with time constants on the order of 1 hour.	Routing tables can be updated synchronously or asynchronously with fractions of a second delays.
Fault tolerance	Redundancy is designed into the IP network by its managers, ensuring tolerance of a single router or network connectivity failure. *n*-fold replication is costly.	Routes and object references can be replicated *n*-fold, ensuring tolerance of *n* failures of nodes or connections.
Target identification	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
Security and anonymity	Addressing is only secure when all nodes are trusted. Anonymity for the owners of addresses is not achievable.	Security can be achieved even in environments with limited trust. A limited degree of anonymity can be provided.

Bild 5.5 Vergleich der Adressierungen IP/P2P

5.3.2 Speicherung von Objekten

Ein Objekt mit einer GUID X wird auf dem Node gespeichert, dessen GUID am nächsten an X ist, sowie auf den *r* weiteren nächsten Nodes (Replikation).

- Finden von Objekten = Finden von Nodes = Finden von GUID
- Bezeichnung: Verteilte Hashtabellen

Als Alternative bietet sich an, einen zusätzlichen Location-Service für Objekte (welche Nodes haben Objekt mit gegebener GUID?) einzubauen.

5.3.3 Ein einfacher Routing-Algorithmus

Ein erster Ansatz für einen Routing-Algorithmus besteht aus folgenden Teilen:

- Jeder Node speichert ein „Leaf-Set“ *L*, bestehend aus den 2*l* GUIDs und IP-Adressen, die numerisch am nächsten an seiner eigenen GUID liegen (*l* kleinere, *l* größere)
- Zirkulärer Ansatz: der kleinere Nachbar von 0…0 ist 1….1.
- Ein Node A, der eine Nachricht mit Ziel GUID D erhält, leitet diese an den Knoten aus *L* weiter, dessen GUID am nächsten an D liegt.
- Wenn A selbst die nächste GUID hat, so ist die Nachricht angekommen.
- Achtung: Nähe = Nähe im GUID-Raum (≠ IP-Hops!)

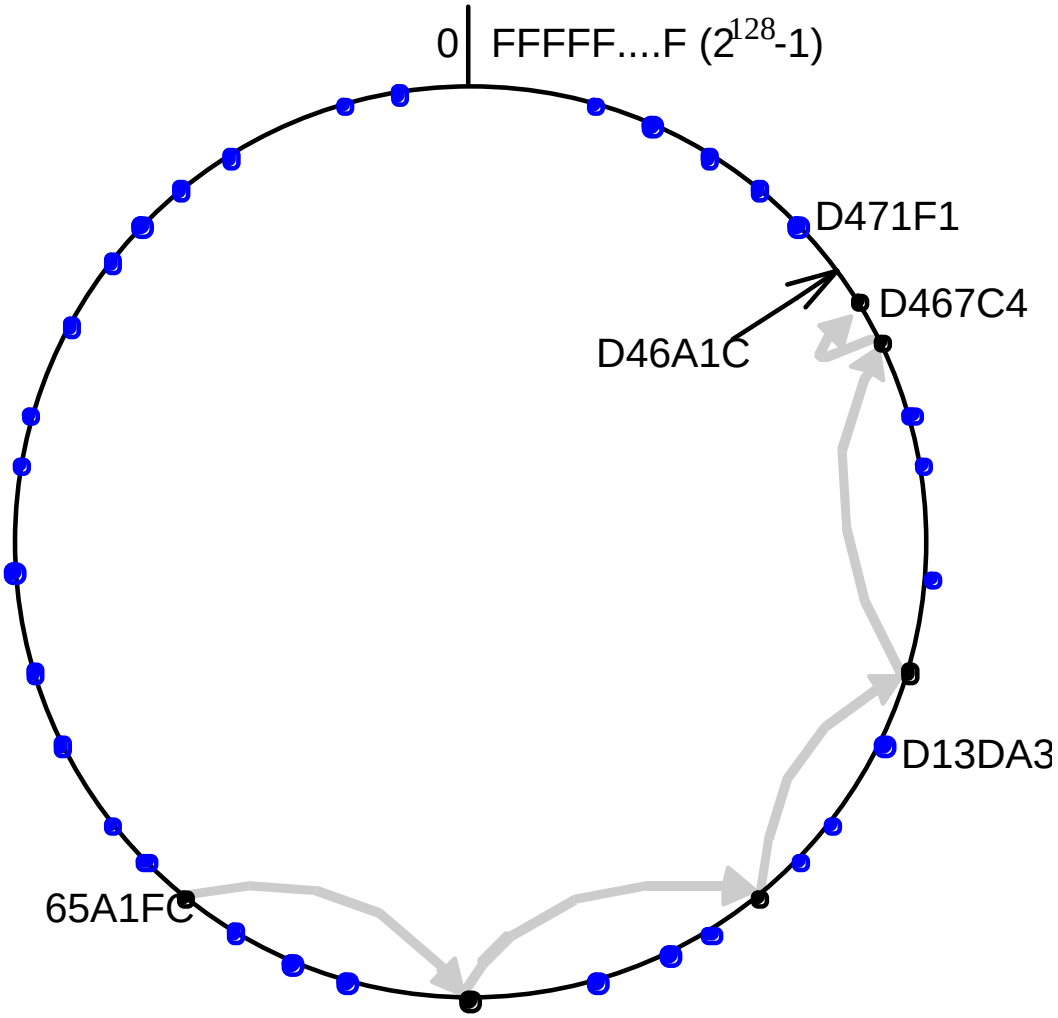


Bild 5.6 Routing einer Nachricht von 65A1FC nach D46A1C mit Leaf Sets der Größe 8 (l = 4)

Dieser Algorithmus hat im Allgemeinen eine Ordnung O(n) und gilt daher als eher untauglich!

5.3.4 Ein verbesserter Routing-Algorithmus

2001 wurde von Rowstron und Druschel ein verbesserter Routing-Algorithmus vorgeschlagen, der Bestandteil eines gesamten API (Application Programmer's Interface) war (Pastry-API). Hauptbestandteil dieses Routing-Algorithmus ist die Routingtabelle in jedem Knoten (Node):

- Jeder Node hat baumartige Routingtabelle, die GUID und IP einer Teilmenge aller Nodes im System repräsentiert.
- Hieraus resultiert eine größere Dichte für nähere GUIDs, der Algorithmus hat die Ordnung $O(\log_{16} N)$

Die weiteren Bestandteil des Pastry API:

- put(GUID, data)
- remove(GUID)
- value = get(GUID)

p =	GUID prefixes and corresponding nodehandles n														
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
	n	n	n	n	n	n		n	n	n	n	n	n	n	n
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6F	6E
	n	n	n	n	n		n	n	n	n	n	n	n	n	n
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E
	n	n	n	n	n	n	n	n	n	n		n	n	n	n
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE
	n		n	n	n	n	n	n	n	n	n	n	n	n	n

Bild 5.7 Tabelle für die Node 65A1xx

In der vorstehenden Tabelle (die eigentlich aus 4 Tabellen besteht) wird in einer Node gespeichert, welche Node n (als Paar (GUID, IP-Adresse)) sie für eine Verbindung wählen muss. Hier ist eine Node mit der GUID 65A1xx gewählt, und eine Kommunikation zu 65A1yy wäre in den ersten 4 Stellen bereits in der Tabelle enthalten. Wird zu einer Node mit der GUID D46A1C kommuniziert, dann hilft die Tabelle in 65A1xx natürlich nur einen Schritt weiter.

Die in den Nodes enthaltenen Tabellen haben folgende Eigenschaften:

- Die Stellen sind hexadezimal codiert.
- Jedes n steht für ein Paar (GUID, IP)
- Die Zeile p in einer Tabelle zeigt die Nodes, deren Ziel-GUID mit der Node-GUID in den höchsten (p+1) Stellen übereinstimmt.

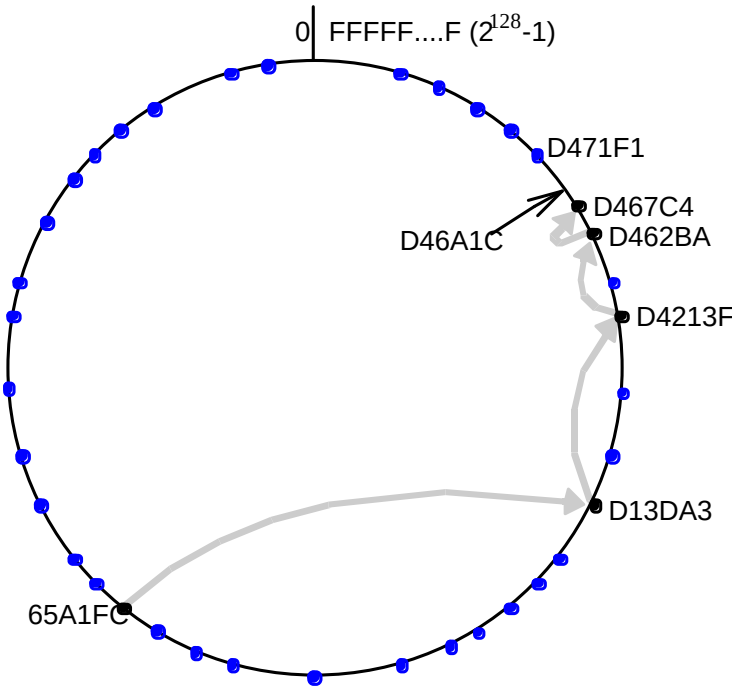


Bild 5.8 Zirkulares, verbessertes Routing

Der Algorithmus wird im Originaltext so angegeben:

To handle a message *M* addressed to a node *D* (where *R*[*p,i*] is the element at column *i*, row p of the routing table):

1. If ($L_{-l} < D < L_l$) { // the destination is within the leaf set or is the current node.
2. Forward *M* to the element L_i of the leaf set with GUID closest to *D* or the current node *A*.
3. } else { // use the routing table to dispatch M to a node with a closer GUID
4. find p, the length of the longest common prefix of D and A. and i, the $(p+1)^{th}$ hexadecimal digit of D.
5. If ($R[p,i] \neq null$) forward M to $R[p,i]$ // route M to a node with a longer common prefix.
6. else { // there is no entry in the routing table
7. Forward M to any node in L or R with a common prefix of length i, but a GUID that is numerically closer.
}
}

Bild 5.9 Routing-Algorithmus nach Rowstron und Druschel

Das damit aber verbundene Problem besteht in dem Aufbau der Routingtabelle und des Leafsets. Hier wird folgendes Verfahren angegeben:

- Berechnung einer GUID (X)
- Kontaktierung eines nahen Nodes (A) mit spezieller join-Nachricht. Nah hier in der Bedeutung „Nähe IP-Hops“.
- A leitet diese an den Node *Z* weiter, dessen GUID am nächsten an *X* liegt. Seien *B, C,…* die Nodes, durch die das Routing weitergeleitet wird. *A, B, C, …, Z* schicken dabei den relevanten Teil ihrer Routingtabellen an *X*.
- *X* baut daraus seine Tabelle auf (Heuristik: erste Zeile von *A*, zweite Zeile von *B*, …)
- Leaf-Set von *X* kann durch Leaf-Set von *Z* approximiert werden.
- Hat *X* seine Tabelle und Leaf-Set, so schickt *X* seine Tabelle an alle dort verzeichneten Nodes, die daraufhin ihre Tabellen aktualisieren.

5.3.5 Verlassen des Systems und Integration neuer Nodes

Kein explizites Protokoll für Verlassen des Systems:

- Verlassen = Fehler
- Fehler := Nächster Nachbar kann keine Kommunikation herstellen

Die Reparatur hat folgende Gestalt:

- Reparatur der Leaf-Sets: Node L, der Fehler diagnostiziert, sucht nach Node L‘, der nahe beim fehlerhaften Node liegt. L‘ schickt sein Leaf-Set an L. Diese überlappen und erlauben es L daher, sein Leaf-Set zu korrigieren. Danach informiert L seine Nachbarn; diese führen dann den gleichen Prozess aus
  Folglich können *l*-1 nebeneinander liegende Nodes gleichzeitig ausfallen
- Reparatur der Routingtabellen „per discovery“: Nachricht wird an Node aus der gleichen Zeile der Tabelle geroutet.

5.3.6 Lokalität

- Rechner mit ähnlichen GUIDs sind normalerweise nicht nah in Bezug auf IP-Hops.

- Grund: Ausfall von Subnetzen soll nicht zu Ausfall von nebeneinander liegenden GUIDs führen.
- Aber: In den Routingtabellen soll jeweils der nächste (=am schnellsten erreichbare) Node stehen.
- Dieser wird (bei Vorliegen von Alternativen) dynamisch bestimmt
- Führt nicht zu optimalem Routing, aber nur zu 30-50% längeren Wegen

5.3.7 Fehlertoleranz

- Routing-Algorithmus geht von korrekten Tabellen und Leaf-Sets aus.
- Nodes senden „heartbeat“-Nachrichten, Fehler werden erkannt und Tabellen werden aktualisiert. Aber: dies kann ggf. länger dauern und daher zu Routing-Fehlern führen
- Daher: Wichtige Nachrichten werden mit „at-least-once“-Semantik geschickt (mehrfache Wiederholung auch ohne Antwort).
- Ein geringer Prozentsatz der Nachrichten wird absichtlich auf „Umwege“ geschickt → Umgang mit fehlerhaften Nodes.

5.3.8 Evaluation im Web Caching

Quelle: http://research.microsoft.com/en-us/um/people/antr/past/squirrel.pdf

- Web Caching: Zwischenspeichern von Webseiten für lokalen wiederholten Zugriff
- Üblicherweise in zentralem Proxy-Server, der damit stark belastet ist.
- Test bei Microsoft (Squirrel): 105 Clients (Cambridge), 36000 Clients (Redmond). Jeder Node steuerte 100 MB Speicher bei.
- Cache Hits:
  Zentraler Proxy 29% (R) und 38% (C)
  Squirrel 28% (R) und 37% (C)
- Latenz für Benutzer:
  4,11 hops (R) und 1,8 hops (C)
  1 Hop für zentralen Proxy
  Aber: Unterschied gering (lokale Hops vs Internet-Zugriff), daher kein Unterschied in der Verwendung des Systems
- Belastung der Nodes:
  0,31 Requests pro Minute (R)

In der Zusammenfassung wird darauf hingewiesen, dass ggf. sekundäre Effekte wesentlich mehr Einfluss haben werden. Hierzu zählen z.B. der hohe Anteil an Harddisk-Zugriffen, der bei zentralen Servern limitierend wirkt.

6 Koordination Verteilter Prozesse

Die bislang betrachtete Prozesskommunikation entspricht der Realisierung der Basisfunktionalität „Prozess A sendet Nachricht an Prozess B“, z.B. über Sockets oder RMI. Dies kann in mehreren Abstraktionsstufen realisiert werden, und die Abstraktion hat den Sinn, die Verteilung im Netz quasi unsichtbar zu machen.

Die nächste, höhere Stufe der Prozesskopplung besteht in der *Prozesskoordination*. Koordination bedeutet u.a. Abstimmung, Synchronisation (zeitlich wie algorithmisch). Hierzu sind einige Punkte zu behandeln:

- Zeit (relative wie absolut), insbesondere die Gleichzeitigkeit
- Wechselseitiger Ausschluss des Zugriffs auf gemeinsame Ressourcen
- Wahlalgorithmen
- Gruppenorientierte Kommunikationsprotokolle
- Transaktionen und Nebenläufigkeitskontrolle

6.1 Zeit in Verteilten Systemen

Zur Erinnerung: „A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages”

Aber: Computer haben Hardwareuhr(en) und Softwareuhren, jeweils mit einer nicht verhinderbaren Abweichung (Skew) und Geschwindigkeitsunterschied (Drift) zur „realen“ Zeit. Die zentrale Frage ist also: *Wie können lokale Uhren im Verteilten System genutzt werden?*

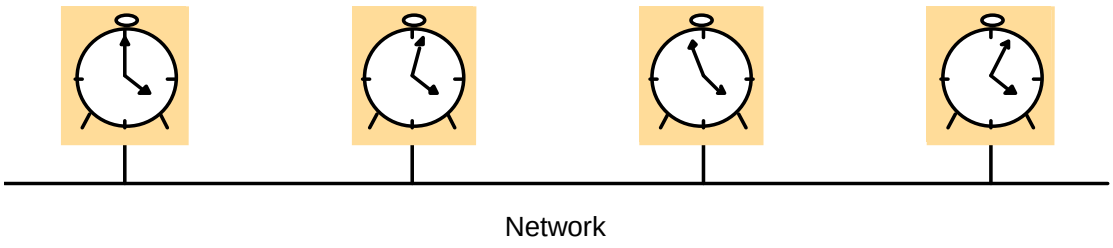


Bild 6.1 Zeit in Verteilten Systemen

6.1.1 Astronomische Zeitrechnung

Die ursprünglich Basis für die Zeitrechnung basiert auf astronomischen Beobachtungen, und zwar bezogen auf die Erde:

1 Solartag = Zeitspanne zwischen zwei „Sun Transits“ (eine Drehung der Erde um eigene Achse)

1 Solarsekunde = ((1/24)/60)/60 eines Solartags

Um 1940 ergaben Forschungen, dass die Erde an Rotationsgeschwindigkeit verliert, d.h. die Tage werden länger. Die Größenordnung dieses Verlusts beträgt ca. 0,5 ns/Jahr. Vor 300 Millionen Jahren hatte ein Jahr daher ca. 400 Tage!

Außerdem existieren kurzfristige Schwankungen in der Tageslänge: So ist die Rotationsgeschwindigkeit im Nordhalbkugel-Sommer (größere Landmasse, Pflanzen treiben Blätter) geringer als im Südhalbkugel-Sommer (Verlängerung des Tags ebenfalls in Größenordnung 1 ns)

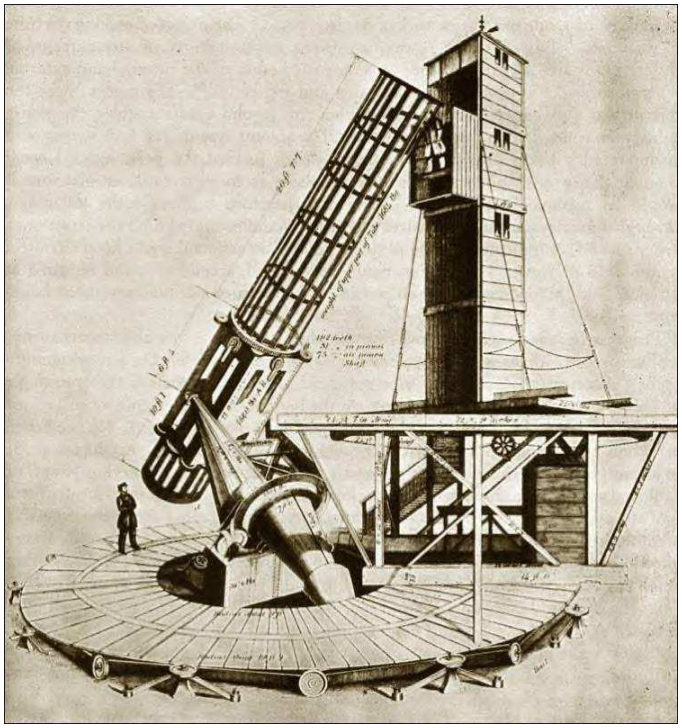


Bild 6.2 Messung der Länge eines Sonnentags

6.1.2 Atomuhren und UTC:

Die neue Definition einer Sekunde lehnt sich an der Strahlungsfrequenz von Atomen an:

- Cäsium 133 schwingt mit 9.192.631.770 Hz (im Grundzustand $^2S_{1/2}$ (F=4, m_F=0) $\rightarrow$ $^2S_{1/2}$ (F=3, m_F=0), eine Hyperfeinstruktur). Zur Erklärung: Diese Schwingung im atomaren Bereich ist außerordentlich „langsam“.
- In 3 Millionen Jahren eine Gangunsicherheit von einer Sekunde, in neueren Versionen sogar in 30 Millionen Jahren (relative Gangunsicherheit 10^{-15})
- Atomzeit (TAI, temps atomique international) als Referenz

Mit diesem Zeitmaß wird nun die UTC (Coordinated Uniform Time) durch Abgleich der Atomuhr mit der Astronomischen Uhr gebildet. Dies führte bislang zu etwa 30 Schaltsekunden.

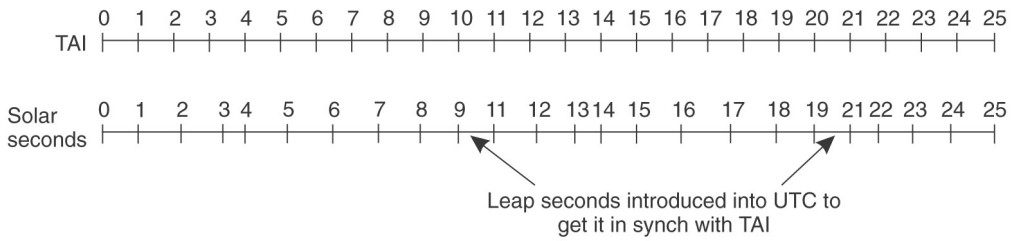


Bild 6.3 Korrektur der UTC mittels Einfügung von Schaltsekunden

6.1.3 Synchronisation physikalischer Uhren

Um physikalischen Uhren, deren Fortschreiten also etwas mit dem Sekundenbegriff zu tun hat, zu koordinieren bedarf es einiger Vorüberlegungen: Welcher Ansatz soll versucht werden, und worin ist ein (oder sind mehrere) Qualitätskriterien zu sehen?

- Ansatz: Es wird versucht, die lokalen (physikalischen) Uhren der beteiligten Rechner so genau wie möglich zu koordinieren.
- Wichtig ist hier (und zugleich ein Qualitätskriterium): Angabe von Fehlerintervallen
- Prinzipien der Synchronisation von Uhren C1,…,Cn:
  - *Externe Synchronisation*: Wenn für eine Grenze D > 0 und eine UTC-Zeitquelle S gilt, dass |S(t) – C*i*(t)| < D für i = 1,…*n* und alle realen Zeiten t ist, so sind die Uhren C*i akkurat* mit Toleranz D.
  - *Interne Synchronisation*: Wenn für eine Grenze D > 0 gilt, dass |C*i*(t) – C*j*(t)| < D für *i*,*j* = 1,…*n* und alle realen Zeiten t ist, so sind die Uhren C*i übereinstimmend* mit Toleranz D.
- Folgende Beziehungen gelten zwischen den Begriffen:
  - Übereinstimmend $\not\Rightarrow$ Akkurat
  - Akkurat mit Toleranz D $\Rightarrow$ Übereinstimmend mit Toleranz 2*D

Naiver Ansatz

Als naiver Ansatz könnten jetzt also zwei Prozesse p1 und p2 ihre Zeit wie folgt synchronisieren:

- Prozess p1 schickt seine aktuelle Zeit C1(t) an Prozess p2.
- p2 setzt beim Empfang der Nachricht zum Zeitpunkt t' seine Uhr auf C2(t') := C1(t) +T_{trans}, wobei letzteres die Übertragungszeit der Nachricht ist (== t'-t).
- Ergebnis: Uhren sind übereinstimmend.
- Aber: T_{trans} ist leider nicht bekannt und kann auch nicht berechnet werden – die Übertragungszeit von Nachrichten variiert in allen Verteilten Systemen (Paketverluste, Netzauslastung, Routing, …) → Algorithmus so nicht umsetzbar

Ansatz für synchrone Verteilte Systeme

- In einem *synchronen* Verteilten System gibt es für die Übertragungszeit einer Nachricht eine maximale Obergrenze (die meisten Verteilten Systeme sind *nicht* synchron!). Dahinter steckt auch der Begriff des synchronen Systems, das so definiert ist, dass der Nachrichtenaustausch in konstanter Zeit erledigt ist (siehe hierzu auch Embedded Systems Handbuch, Kapitel 2.2). Das System heißt perfekt synchron, wenn die konstante Zeit = 0 ist.
- Synchronisationsalgorithmus für synchrone Systeme
  - Prozess p1 schickt seine aktuelle Zeit C1(t) an Prozess p2.
  - p2 setzt beim Empfang der Nachricht zum Zeitpunkt t' seine Uhr auf C2(t') := C1(t) +(max+min)/2, wobei min und max die minimale und maximale Übertragungszeit sind.
  - Ergebnis: Uhren sind übereinstimmend mit Toleranz u/2, wobei u als (max-min) die Spanne zwischen minimaler und maximaler Übertragungszeit ist.
- Allgemein: *n* Uhren in einem synchronen verteilten System können bestenfalls mit der Toleranz u*(1-1/n) übereinstimmen.
- Algorithmus nicht verwendbar für asynchrone Verteilte Systeme

Asynchrone Verteilte Systeme I: Algorithmus von Cristian:

- Ziel: Externe Synchronisation in asynchronen Verteilten Systemen
- Ansatz: Messung der Roundtrip-Zeit für Nachrichten
  - Prozess p fragt Zeitserver S in Nachricht m_r nach Zeit
  - S antwortet mit S(t) in Nachricht m_t.
  - T_{round} = Zeit zwischen Senden von m_r und Empfang von m_t.

- p setzt seine Zeit auf S(t) + T_{round} / 2
- Bei bekannter minimaler (einfacher) Übertragungszeit min gilt, dass der Time-Server zum (Time-Client-) Zeitpunkt t+min frühestens und zum Zeitpunkt (t+ T_{round} -min) spätestens gesendet hat. Daraus folgt:
- Uhr von p ist akkurat mit Toleranz T_{round} / 2 – min

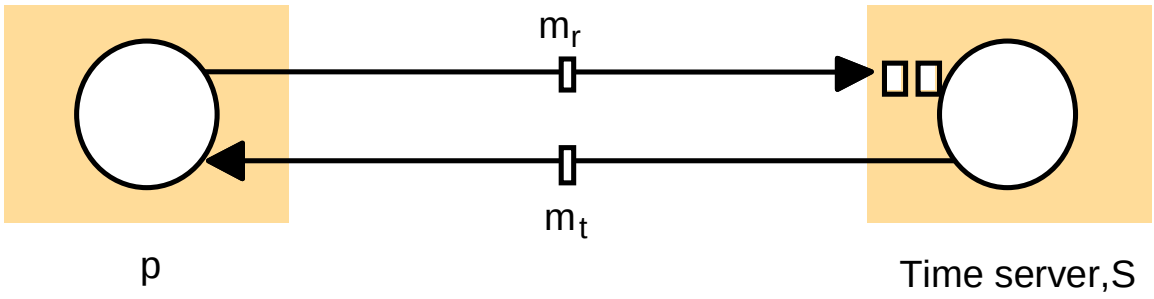


Bild 6.4 Zeitsynchronisation mittels zentralem Zeitserver

Probleme des Algorithmus von Cristian:

- Ausfall des Zeitservers
- Fehlfunktion des Zeitservers (defekt oder mutwillig) führt zu Fehlfunktion aller Uhren

Asynchrone Verteilte Systeme II: Berkeley-Algorithmus

- Ein Teilnehmer ist Master, alle anderen Slaves. Bei Ausfall des Masters kann ein neuer gewählt werden (siehe Wahlalgorithmen).
- Master fragt Slaves regelmäßig nach deren Zeit
- Dann bildet Master den Durchschnitt dieser Zeiten (und seiner eigenen), wobei „Ausreißer“ ignoriert werden.
- Master sendet den einzelnen Clients dann die Zeit, um die deren Uhren jeweils justiert werden müssen, also die Zeitdifferenzen.

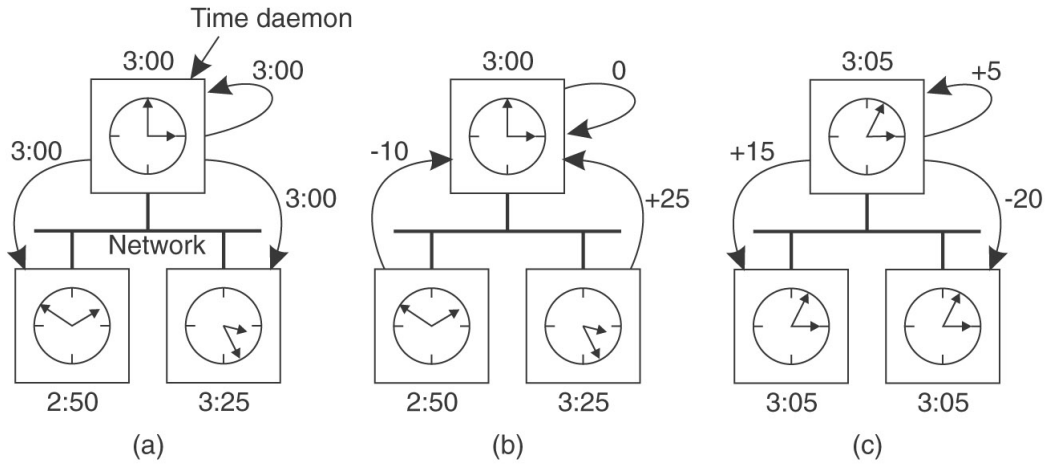


Bild 6.5 Zeitsynchronisation durch Mittelwertbildung
a) Aktueller Master fragt Slaves nach Zeitdifferenz b) Slaves antworten c) Master sendet Zeitkorrektur

6.1.4 Network Time Protocol (NTP)

Das Ziel dieses Protokolls ist die Synchronisation von Uhren im Internet mit UTC. Hierfür sind verschiedene Modi (alle unter Nutzung von UDP) möglich:

- **Multicast**:
  Server schickt Zeit periodisch an Multicast-Gruppe; diese setzt ihre Zeit dementsprechend (kleine Verzögerung einkalkuliert). *Nur für Verwendung im LAN geeignet.*
- **Procedure-call**:
  Ähnlich zu Cristian-Methode (Server erwartet Anfragen, sendet seine Zeit zurück). *Verwendung, wenn höhere Genauigkeit gefordert, oder wenn Multicast nicht zur Verfügung steht*
- **Symmetric**:
  Server tauschen permanent Nachrichtenpaare mit Zeitstempeln aus. *Verwendung für höchste Genauigkeit → Serverkommunikation*

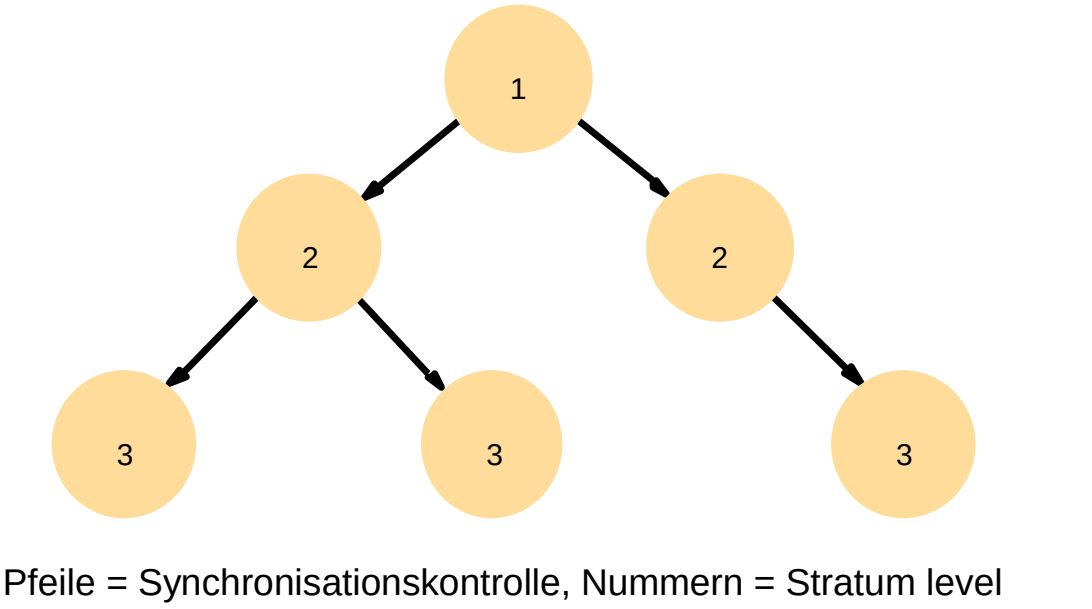


Bild 6.6 Prinzip der Zeitübermittlung: Server 1 hält UTC und übermittelt diese über viele Ebenen an alle anderen

Das Wort Stratum (Level) bedeutet hierbei die Qualität(sstufe) in der Synchronisation.

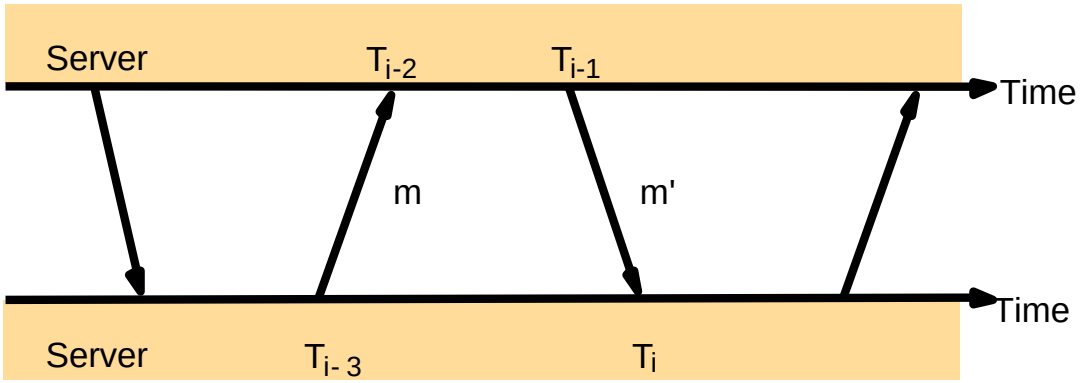


Bild 6.7 Prinzip der symmetrischen, ständigen Zeitsynchronisation

- Jede Nachricht hat drei Zeitstempel (hier: m' hat Stempel T_{i-3}, T_{i-2} und T_{i-1})
- Nachrichten können verloren gehen. Wenn sie ankommen, sind aber die Zeitstempel gültig.
- Zwischen T_{i-2} und T_{i-1} kann ggf. große Zeitspanne bestehen.

- Für jedes Nachrichtenpaar berechnet der Algorithmus ein Offset o_i (Unterschied der Uhren) und ein Delay d_i (Übertragungszeit für die beiden Nachrichten)
- Ist o der echte Unterschied der Uhr von B im Vergleich zu A, und sind t und t' die Übertragungszeiten für m und m', so gilt:
  - $T_{i-2} = T_{i-3} + t + o$ und $T_i = T_{i-1} + t' - o$
  - $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
  - $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i) / 2$ und $o = o_i + (t - t') /2$
- Es kann gezeigt werden: $o_i - d_i/2 <= o <= o_i + d_i/2$.
- Daher garantiert die Wahl von o_i eine Übereinstimmung mit Toleranz $d_i/2$
  - TP-Server halten Historie der Paare (o_i,d_i) und berechnen statistische Streuwerte (filter dispersion). Hohe Dispersion deutet auf weniger verlässliche Datenquelle hin.
  - Zur Berechnung von o wird Paar mit geringstem d_i verwendet.
  - Kommunikation mit mehreren anderen Servern: Nur der beste peer (delay, stratum level) wird zur Aktualisierung der Uhr verwendet.

6.1.5 Logische Uhren

Das generelle Problem aller Synchronisationsalgorithmen für physikalische Uhren besteht darin, dass nur Approximation möglich ist. Im Gegensatz dazu ist die logische Reihenfolge (Ereignis x geschieht vor Ereignis y) ist für einen lokalen Prozess p_i stets exakt entscheidbar. Als Notation wird hierfür: $x \rightarrow_i y$ gewählt.

Die Idee ist nun, eine solche logische Reihenfolge „$\rightarrow$" von Ereignissen („*geschieht-vor*") auch für verteilte Prozesse zu konstruieren.

- Geschehen zwei Ereignisse in einem Prozess p_i, so ist die Reihenfolge bestimmt.
- Das Empfangen einer Nachricht (in einem Prozess p_y) passiert nach dem Senden der Nachricht (in einem Prozess p_x).

Diese Relation „happens before" (HB) zeigt folgende, formale Eigenschaften:

Für alle Ereignisse e, e' und e'' gilt:

- HB1: Wenn es einen Prozess p_i gibt mit $e \rightarrow_i e'$, dann gilt allgemein $e \rightarrow e'$
- HB2: Für jede Nachricht m gilt: $send(m) \rightarrow receive(m)$
- HB3: Aus $e \rightarrow e'$ und $e' \rightarrow e''$ folgt $e \rightarrow e''$
- Allerdings ist $\rightarrow$ keine totale Relation von Ereignissen. Gilt weder $a \rightarrow b$ noch $b \rightarrow a$, so heißen die Ereignisse nebenläufig: $a \parallel b$.
- Weiterhin ist zu beachten, dass $\rightarrow$ nicht bedeutet, dass eine kausale Beziehung zwischen den Ereignissen besteht! Es ist lediglich eine zeitliche Beziehung.

Logische Uhren (lokale Zähler für Ereignisse) erlauben es nun, die „happened-before" Relation numerisch zu fassen. Hierzu werden (als ersten Ansatz) Lamport-Zeitstempel (Lamport Counter, LC) eingeführt und wie folgt berechnet:

- Jeder Prozess p_i hat Zähler L_i. Dieser ist am Anfang (beim Start des Prozesses) 0.
- LC1: L_i wird um eins erhöht, bevor ein Ereignis durch p_i verarbeitet wird: $L_i := L_i + 1$
- LC2: Wenn ein Prozess p_i eine Nachricht sendet, schickt sie dieser den Zeitstempel $t = L_i$ mit
- LC3: Wenn ein Prozess p_j eine Nachricht (m, t) empfängt, berechnet er $L_j := \max(L_j, t)$ und wendet LC1 an, bevor er das Ereignis receive(m) mit einem Zeitstempel versieht.

Es gilt stets: $e \rightarrow e' \Rightarrow L(e) < L(e')$. Die Umkehrung allerdings gilt i.Allg. nicht: $L(e) < L(e') \not\Rightarrow e \rightarrow e'$

Vektorzeitstempel

Das Manko, dass die vorstehende Umkehrung nicht gilt, lässt sich durch Vektorzeitstempel beheben, Hierzu muss die Anzahl der Prozesse in dem betrachteten System bekannt sein. In einem solchen System mit *n* Prozessen hat jeder Prozess p_i einen Zeitstempelvektor V mit n Komponenten. Am Anfang ist V_i= {0,…,0} für alle *i*. Auf dieses System werden folgende Regeln angewendet:

VC1: $V_i[i]$ wird um eins erhöht, bevor ein Ereignis durch p_i verarbeitet wird: $V_i[i] := V_i[i] + 1$

VC2: Wenn ein Prozess p_i eine Nachricht sendet, schickt sie dieser den Zeitstempelvektor $t = V_i$ mit

VC3: Wenn ein Prozess p_i eine Nachricht (m, t) empfängt, berechnet er $V_j[k] := \max(V_j[k], t[k])$ und wendet VC1 an, bevor er das Ereignis receive(m) mit einem Zeitstempel versieht.

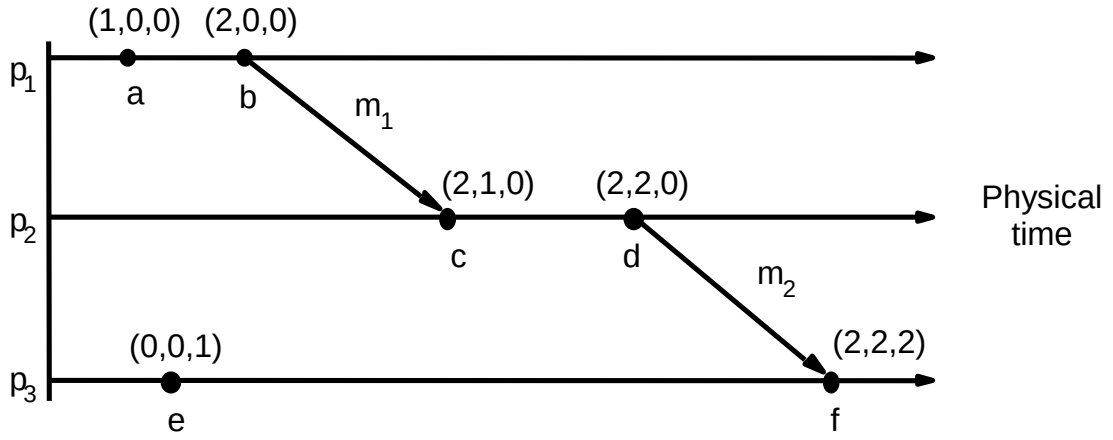


Bild 6.8 Logische Uhren mit Vektorzeitstempeln

Für den Vergleich von Vektorzeitstempeln gelten folgende Regeln:

- V = W :⇔ V[i] = W[i] für alle *i*
- V <= W :⇔ V[i]<= W[i] für alle *i*
- V < W :⇔ V <= W und V ≠ W

Es gilt: e → e' ⇔ V(e) < V(e')

Beweisidee ⇒

- Zerlege e → e‘ in Folge e = e0 → e1 → e2 → e3 → e4 … → en = e‘
  so, dass jeweils $e_i \rightarrow e_i + 1$ im selben Prozess ablaufen oder ein Paar aus send/receive stammt.

Beweisidee ⇐

Annahme: Es sei !(e → e‘).

Sei e in Prozess i und e‘ in Prozess j

Betrachte k := $V_i[i]$ (e)

Wegen Annahme existiert keine Folge von Ereignissen, die p_i‘s k-tes Ereignis (oder ein späteres!) enthält und bei p_j bei (oder vor!) e‘ endet.

6.2 Gegenseitiger Ausschluss

Das MUTEX (MUTual EXclusion)-Problem ist wie folgt charakterisiert: Stelle sicher, dass nur ein Prozess gleichzeitig auf eine Ressource zugreift (= sich in einer „critical section“) befindet.

Die lokale Lösung für das MUTEX-Problem ist trivial: Zu diesem Zweck wird an einer Speicherstelle hinterlegt, ob die Ressource gerade genutzt wird oder nicht. Die dafür genutzten Speicherstellen bzw. Variablen werden häufig als *Semaphoren* bezeichnet.

In verteilten Systemen, bei denen die Prozesse verteilt sind und somit keinen gemeinsamen Speicher haben, muss ein Verfahren für Distributed MUTEX entwickelt werden, denn die Standardlösung ist nicht nutzbar. Einfache Beispiele sind die Koordination des Zugriffs auf Netzwerk, insbesondere bei wireless LAN (WLAN, Bild 6.8) und das von Dijkstra dargestellte Philosophenproblem, bei dem 5 am Tisch sitzende Philosophen nur zwei Aktivitäten kennen: Essen und Denken. Zum Zweck des Essens sind Spaghetti, Teller und 5 Gabeln vorhanden, es kann aber zu Blockaden kommen, wenn man zum Essen von Spaghetti 2 Gabeln benötigt und jeder zunächst eine Gabel greift, um dann auf das Freiwerden der anderen zu warten.

Bild 6.9 Zum MUTEX-Problem

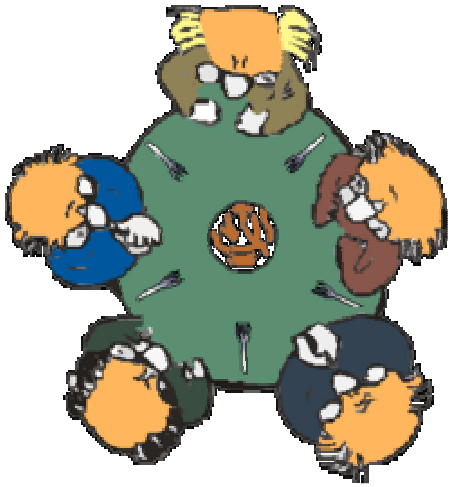


Bild 6.10: Philosophen-Problem (Dijkstra)

Wenn man nun also Algorithmen für das Distributed-MUTEX-Problem entwickeln bzw. diskutieren will, muss man sich über die Kriterien zur Beurteilung der Güte Gedanken machen. Folgende Güte-Kriterien seien hier vorgeschlagen:

1. Zu jedem beliebigen Zeitpunkt darf sich höchstens einer der beteiligten Prozesse im kritischen Abschnitt befinden (*safety*)
2. Anfragen, den kritischen Abschnitt betreten oder verlassen zu dürfen, sind irgendwann erfolgreich (*lifeness*)
3. Wenn eine Anfrage, den kritischen Abschnitt zu betreten, vor einer anderen erfolgt, so wird der Zutritt in dieser Reihenfolge gewährt (*ordering*)

Zur Implementierung von Algorithmen werden einige Operationen durch die Prozesse benötigt. Hier eine minimale Auswahl:

- enter() // Wunsch, kritischen Abschnitt zu betreten (wird ggf. blockiert, bis es möglich ist)
- resourceAccess() // Zugriff auf gemeinsame Ressource
- exit() // Verlassen des kritischen Abschnitts (andere Prozesse können ihn jetzt betreten)

Weiterhin werden über die Prozesse einige Annahmen gemacht:

- Prozesse sind fair (und verlassen den kritischen Abschnitt wieder)

- Der Crash eines Prozesses kommt nicht vor
- Man hat eine verlässliche Kommunikation

Neben der Beurteilung der Güte können einige Parameter der MUTEX-Algorithmen gemessen werden, so dass auch quantitative Aussagen vorliegen::

- *Bandbreite*
  (gemessen in Anzahl der Nachrichten in jeder enter()- und exit()-Operation)
- *Clientverzögerung*
  (Wie lange muss Clientprozess warten?)
- Einfluss auf Durchsatz: *Synchronisationsverzögerung*
  (wie viele Nachrichten sind zwischen exit() bei einem Prozess und dem Betreten des kritischen Abschnitts beim nächsten nötig?)
- Sicherheit/*Safety*: Es können keine fehlerhaften Zustände eintreten (statische Eigenschaft)
- Lebendigkeit/*Liveness*: Alle Zustände werden in endlicher Zeit erreicht.

6.2.1 Einfacher MUTEX-Algorithmus 1: Server-basiert

Ein einfacher, Server-basierter Algorithmus zum gegenseitigen Ausschluss beruht darauf, dass die einzelnen Prozesse (oder Clients) Anfragen (*Request for Token*) stellen und diese durch den Server als zentrale Stelle verwaltet werden. Um dies zu gewährleisten, sind 3 verschiedene Nachrichten notwendig: *Request Token*, *Release Token* und *Grant Token*.

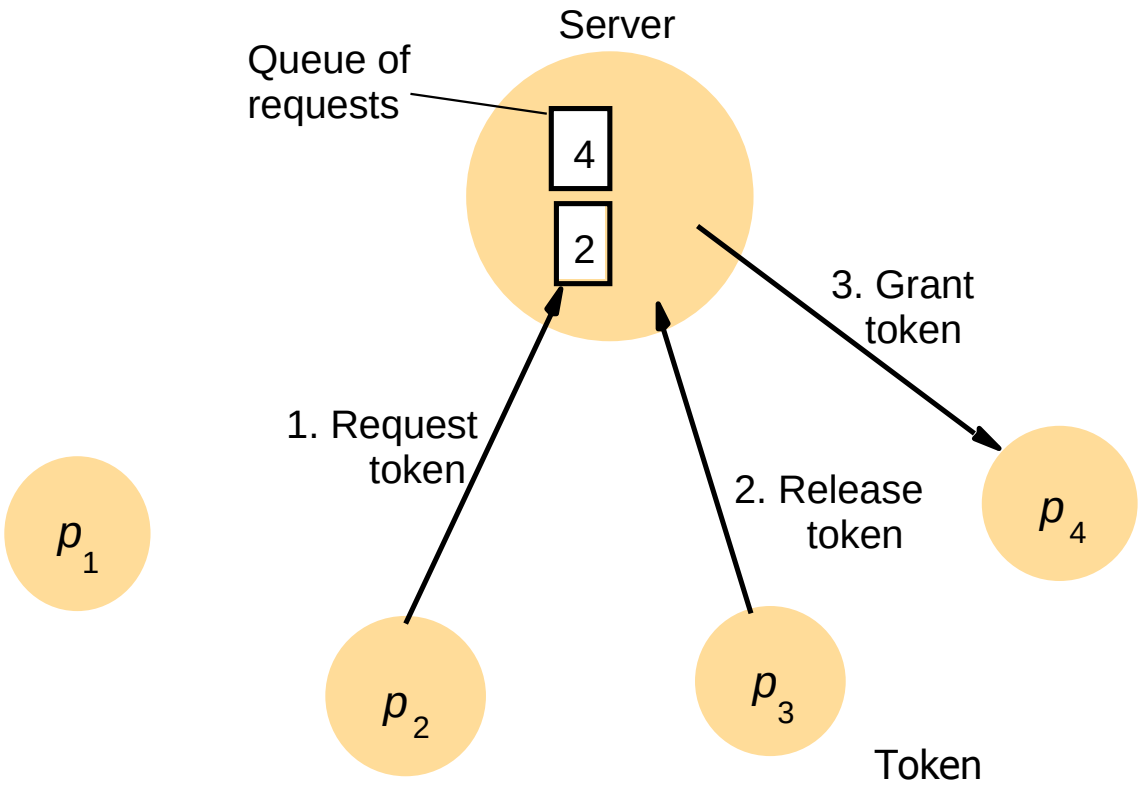


Bild 6.11 Server-basierter DMUTEX-Algorithmus

In dem in Bild 6.11 dargestellten Server-basierten Algorithmus (für DMUTEX) verwaltet der Server die Warteschlange der Anfragen (Queue of Requests). Die eingetragenen Nummern entsprechen den Prozess-Nummern, nicht den Ressourcen. Im Beispiel hier hat P3 offenbar den Token, P4 wartet bereits.

Wenn nun P2 einen Request for Token stellt, passiert nichts – außer dass der Request in der Warteschlange gespeichert wird. Wenn nun P3 den Token zurückgibt, dann wird dieser neu verteilt, und zwar an P4 (Grant Token).

Die drei wichtigen Eigenschaften

- Safety (+)
- Liveness (+)
- Ordering (-)

sind zum Teil erfüllt. Eine Reihenfolge allerdings kann so nicht eingehalten werden, jedenfalls nicht perfekt, denn der Server speichert die Anfragen in der eingehenden Reihenfolge – nicht in der zeitlichen. Das ist aber in den meisten Fällen akzeptabel.

Für die weiteren Parameter gelten folgende Schätzungen:

- Die Bandbreite (Anzahl der Nachrichten pro enter() und exit()-Aktionen) beträgt 3 Nachrichten pro Anfrage (Request, Grant, Release)
- Die Clientverzögerung (Wartezeit) beträgt mindestens die Zeit für einen Roundtrip (Request/Grant + Bearbeitungszeit).
- Die Synchronisationsverzögerung beträgt zwei Nachrichten (Request/Grant)

Als wesentlicher Nachteil muss angesehen werden, dass der Server den Flaschenhals (Bottleneck) darstellt! Allein aus diesem Grund versucht man gerne, die Abstimmung zum Ressourcenzugriff im verteilten System selbst, also ohne Server zu machen.

6.2.2 Ein einfacher Algorithmus, Ansatz 2: Token Ring

Der zweite Ansatz besteht im Wesentlichen darin, dass auf den zentralen Server verzichtet wird und anstelle dessen ein (logischer) Ring gleichberechtigter Clients aufgebaut wird. Jede Ressource, die im gegenseitigen Ausschluss verwaltet werden muss, bekommt einen Token (= Zugriffsberechtigung), die im Ring weitergereicht wird, wenn sie nicht (mehr) benötigt wird. Diese Ressource kann beispielsweise einem Client zur Verwaltung zugeordnet werden, dieser Client muss den Token dann erzeugen.

Auch hier wird die zeitliche Ordnung nicht eingehalten, und zwar komplett nicht, alle anderen Kriterien sind jedoch erfüllt:

- Safety (+)
- Liveness (+)
- Ordering (–)

Für die anderen, Performance-orientierten Eigenschaften gelten folgende Schätzungen:

- Bandbreite: quasi-kontinuierlich
- Clientverzögerung: Zeit für 0 bis *N* Nachrichten
- Synchronisationsverzögerung: zwischen 1 und *N* Nachrichten

Das wesentliche Problem dieser Architektur liegt darin, dass bei Ausfall des Clients, der den Token besitzt, der Ring verloren geht. Es wurde zwar zu Beginn angenommen, dass kein Prozess (und auch kein Client) crasht, aber diese Annahme ist natürlich wirklichkeitsfremd.

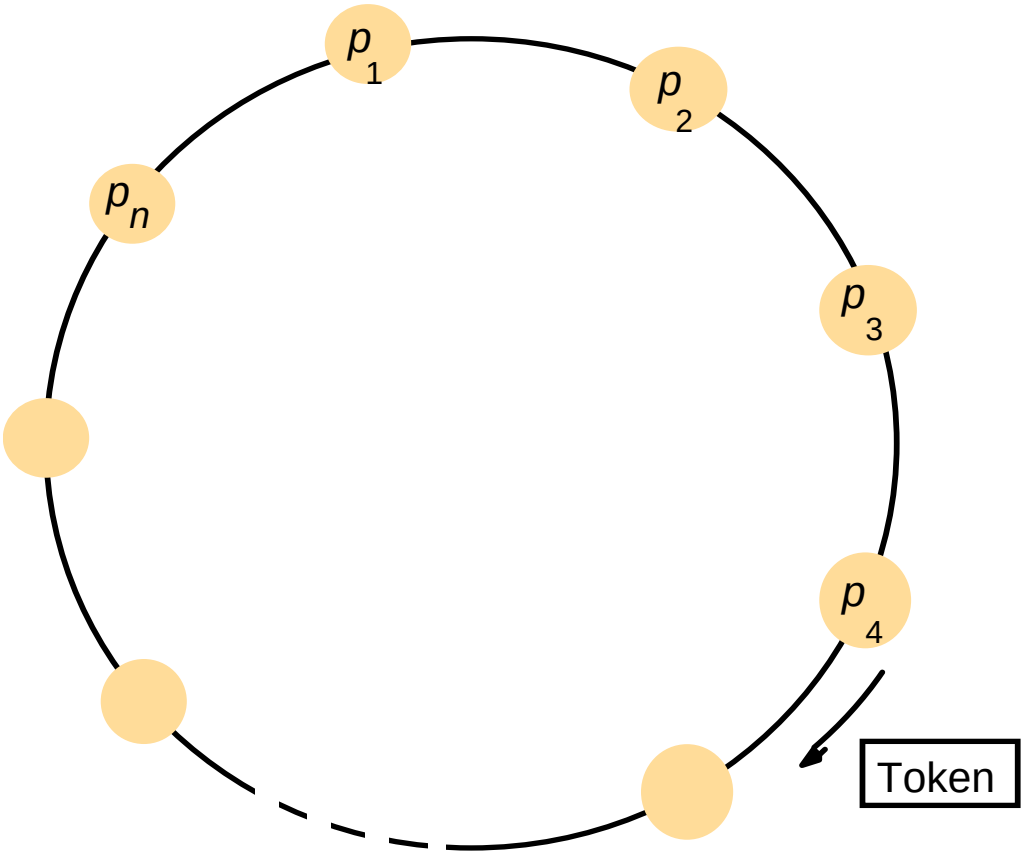


Bild 6.12 Aufbau logischer Token-Ring für MUTEX-Algorithmus

6.2.3 DMUTEX mit logischen Uhren: Ricart & Agrawala (1981)

Die Grundlage für den nächsten verteilten Algorithmus zur Verwaltung von MUTEX besteht darin, dass jeder Prozess (Client) seine eigenen Ressourcen verwaltet und auch den Status dafür speichert. Zudem werden Zeitstempel für das Eintreffen von Anforderungen (Requests) verwendet. Der nächste Algorithmus zum Distributed MUTEX hat folgende Gestalt:

Für alle Teilnehmer gilt:

```
On initialisation
      state := RELEASED;
```

Für einen anfordernden Teilnehmer gilt folgender Algorithmus:

```
To enter the section
      state := WANTED;
      T := request's timestamp;
      Multicast request to all processes;
      Wait until (number of replies received == (N – 1));
      state := HELD;
```

Für einen verwaltenden Teilnehmer gilt folgender Algorithmus:

```
On receipt of a request <Ti, pi> at pj (i ≠ j)
      if( state == HELD or (state == WANTED and (T, pj) < (Ti, pi))) then
            queue request from pi without replying;
      else
            reply immediately to pi;
      end if
```

```
To exit the critical section
      state := RELEASED;
      reply to any queued requests;
```

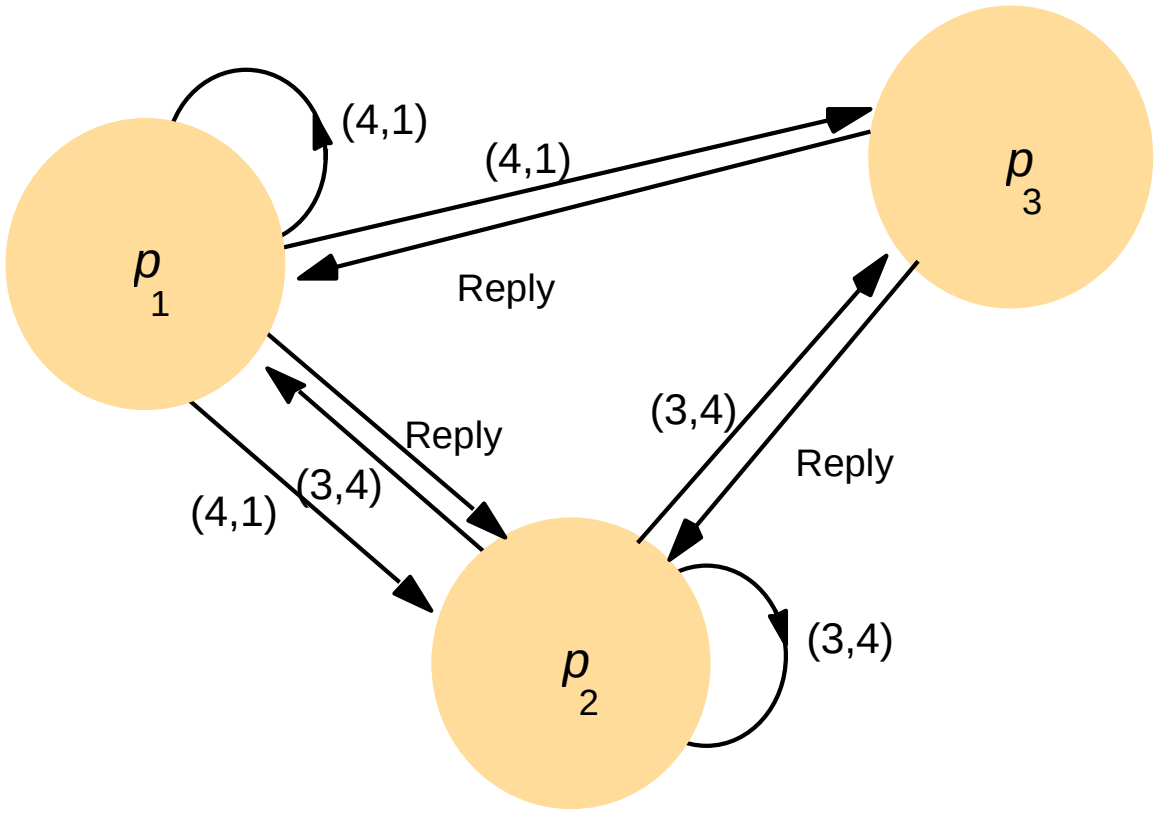


Bild 6.13 DMUTEX-Algorithmus mit logischen Uhren

Der MUTEX-Algorithmus nach Ricart und Agrawala basiert also auf der Nutzung von Zeitstempeln. Der Nichtempfang einer Statusnachricht auf ein Request wird so interpretiert, dass diese Ressource belegt ist oder von einem noch früheren Request angefordert wurde und somit zurzeit nicht verfügbar ist. So sind natürlich Ausfälle nicht erkennbar.

Dennoch: Mithilfe dieses Algorithmus' können die drei Forderungen

- Safety (+)
- Liveness (+)
- Ordering (+)

erfüllt werden, die Ordnung natürlich nur insoweit, wie es die Zeitstempel zulassen. Allerdings ist die Bandbreite (also die Nutzung des Netzwerks) recht hoch:

- Bandbreite: 2*(N-1)
- Clientverzögerung: Zeit für 1 Roundtrip
- Synchronisationsverzögerung: 1 Nachricht

6.2.4 DMUTEX mit Voting-Sets: Maekawa (1985)

Voting Sets sind Teilmengen aller Prozesse, die gebildet werden, um bestimmte Koordinierungsaufgaben in verteilten Systemen durchzuführen. Eine der Koordinationsaufgaben kann dabei z.B. darin bestehen, Ressourcen zu verwalten. Hierzu hat jeder Prozess pi (i=1,…,N) ein Voting Set, das aus einer Teilmenge der Prozesse besteht. Anstelle alle anderen Prozesse zu fragen, wird nur die Zustimmung des Voting Sets für den Zugang zum kritischen Abschnitt benötigt. Bedingungen für Voting Sets:

- pi ∈ Vi

- Vi ∩ Vj ≠∅ für alle *i* und *j* (je zwei Sets haben zumindest ein gemeinsames Element)
- | Vi | = K für alle *i* (alle Sets sind gleich groß)
- Jeder Prozess ist in M Voting Sets enthalten

Die optimale Lösung (Minimalität von K) ist etwa: K ≈ √N

Der Algorithmus hat etwa folgende, zu dem zuvor vorgestellten Gestalt:

Für alle Teilnehmer gilt:

```
On initialization
       state := RELEASED;
       voted := FALSE;
```

Für einen anfordernden Teilnehmer gilt folgender Algorithmus:

```
For pi to enter the critical section
       state := WANTED;
       Multicast request to all processes in Vi;
       Wait until (number of replies received = K);
       state := HELD;
```

Für einen verwaltenden Teilnehmer gilt folgender Algorithmus:

```
On receipt of a request from pi at pj
       if (state = HELD or voted = TRUE) then
              queue request from pi without replying;
       else
              send reply to pi;
              voted := TRUE;
       end if
For pi to exit the critical section
       state := RELEASED;
       Multicast release to all processes in Vi;
On receipt of a release from pi at pj
       if (queue of requests is non-empty) then
              remove head of queue – from pk, say;
              send reply to pk;
              voted := TRUE;
       else
              voted := FALSE;
       end if
```

Die Bewertung für diesen Algorithmus:

- Safety (+)
- Liveness (–)
- Ordering (–), kann durch Einführung von „geschieht-vor“-Speicherung der Requests eingeführt werden (so wie es im Algorithmus von Ricart und Agrawala erfolgt ist)
- Bandbreite: 2K+K ≈ 3√N
- Clientverzögerung: Zeit für 1 Roundtrip
- Synchronisationsverzögerung: 1 Roundtrip

6.3 Wahlalgorithmen

Das Ziel dieser Klasse von Algorithmen ist die Wahl eines *Koordinators* unter einer Menge von Prozessen. Beispielanwendungen hierfür sind:

- Wechselseitiger Ausschluss mit dynamischem Server
- Berkeley-Algorithmus (Uhrensynchronisation mithilfe Uhrenserver)

Die Grundannahmen für die Wahlalgorithmen:

- Prozesse haben eindeutige IDs
- Prozesse kennen die IDs der anderen Prozesse (zumindest eine Teilmenge davon)
- Prozesse können evtl. ausfallen (ggf. wird auch angenommen, dass kein Prozess ausfällt)
- Wahlen können von mehreren Prozessen nebenläufig initiiert werden

Jeder Prozess p(i) hat eine lokale Variable elected(i), welche nach Ablauf des Verfahrens die ID des gewählten Prozesses enthalten soll. Bei Start des Verfahrens wird elected(i) auf *null* gesetzt.

Die Kriterien zur Bewertung der Wahlalgorithmen sind:

- E1: Im Prozess p(i) gilt zu jedem Zeitpunkt elected(i) = *null* oder elected(i) = P, wobei P der am Ende der Wahl lebende Prozess mit der größten ID ist. Hierdurch existiert immer ein eindeutiger Zustand (safety).
- E2: Alle teilnehmenden Prozesse setzen irgendwann ihre Wahlvariable auf P – oder sie fallen aus (liveness)

Die Messung der Effizienz kann wie folgt erfolgen:

- Anzahl der Nachrichten pro Wahl
- Zeit pro Wahl

6.3.1 Wahlalgorithmus I: Ring (Chang & Roberts 1979)

Der nunmehr dargestellte Wahlalgorithmus von Chang und Roberts arbeitet in einer ringförmigen Topologie und kann angewendet werden, wenn Prozessausfälle ausgeschlossen sind. Das Vorgehen ist dann so organisiert:

- N Prozesse sind in einem Kommunikationsring organisiert, jeder Prozess p(i) kennt seinen Nachfolger p(i+1) mod N
- Prozesse können entweder Status „Teilnehmer“ oder „Nicht-Teilnehmer“ haben. „Teilnehmer“ entspricht auch Kandidat, Nicht-Teilnehmer ist der Standardwert.

Im Algorithmus verwendete Nachrichten:

- elect(k): Vorschlag zur Wahl des Prozesses mit ID *k*
- elected(k): Mitteilung, dass Prozess mit ID *k* gewählt wurde

Ausgangssituation des Algorithmus ist, dass alle Prozesse „Nichtteilnehmer“ sind. Es soll der Prozess mit der größten ID gewählt werden. Der Initiator der Wahl (p(i)) markiert sich als „Teilnehmer“ und schickt Nachricht elect(ID(p(i))) an seinen Nachfolger.

Wenn ein Prozess p(j) eine elect()-Nachricht erhält, so vergleicht er die enthaltene ID mit seiner eigenen ID. Ist die übermittelte ID…

- größer, so leitet er die ursprüngliche Nachricht an seinen Nachfolger weiter und markiert sich als Teilnehmer.
- kleiner und p(j) ist Nichtteilnehmer, so schickt er seine eigene ID in einer elect-Nachricht an seinen Nachfolger, und markiert sich als Teilnehmer
- kleiner und p(j) ist Teilnehmer, so führt p(j) keine Aktion aus
- gleich, so markiert sich p(j) als Nichtteilnehmer und Koordinator und schickt eine Nachricht elected(ID(p(j))) an seinen Nachfolger.

Wenn ein Teilnehmer-Prozess p(k) eine elected()-Nachricht erhält, so leitet er diese weiter, wird zum Nichtteilnehmer und setzt seine Variable elected(k) auf die erhaltene ID.

Der Algorithmus wählt also denjenigen Prozess mit der höchsten ID aus. Im nachfolgenden Bild 6.14 wird dieses Verfahren exemplarisch gezeigt. Die Wahl wurde durch Prozess mit ID 17 gestartet, das bisherige Maximum ist 24. Die Prozesse mit dem Status „Teilnehmer“ sind dunkler dargestellt.

- E1: Im Prozess p(i) gilt zu jedem Zeitpunkt elected(i) = *null* oder elected(i) = *P*, wobei *P* der am Ende der Wahl lebende Prozess mit der größten ID ist. (safety)
- E2: Alle teilnehmenden Prozesse setzen irgendwann ihre Wahlvariable auf *P* (liveness)

Worst-Case Analyse für die Performance:

- Der Prozess direkt hinter dem Prozess mit der höchsten ID startet die Wahl
- Die Folge: (N-1)-elect()-Nachrichten, bis der Prozess mit der höchsten ID gefunden ist, und N elect()-Nachrichten, bis dies bestätigt ist (den Prozess mit höchster ID wieder erreicht).
- N elected()-Nachrichten
- Insgesamt also 3N-1 Nachrichten
- Zeitbedarf: 3N-1 Nachrichtenübertragungen (maximal)
- Im bestmöglichen Fall initiiert der Prozess mit der höchsten ID die Wahl, dadurch werden nur 2*N-Nachrichten benötigt.

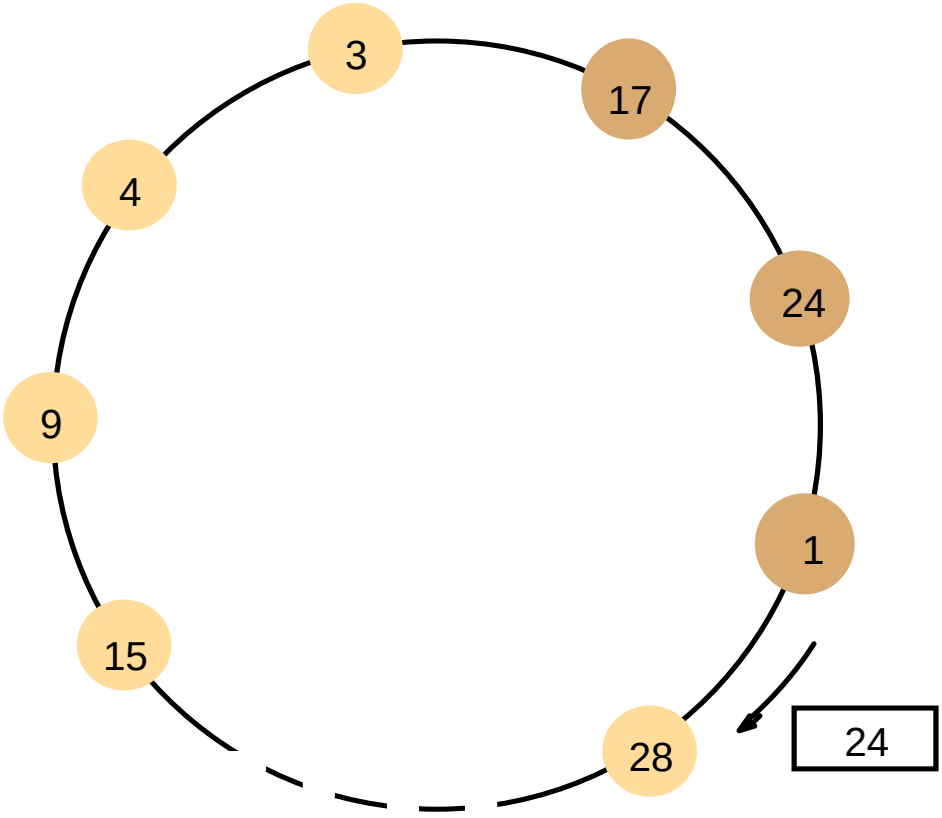


Bild 6.14 Wahlalgorithmus 1 (es wird die höchste ID gesucht)

6.3.2 Wahlalgorithmus 2: Bully (Garcia-Molina 1982)

Dieser Algorithmus ermöglicht die Wahl auch unter der Annahme, dass Prozesse ausfallen. Die Voraussetzungen hierfür sind:

- Es liegt ein synchrones System vor, d.h. ein Prozessausfall wird durch einen Timeout sicher erkannt
- Jeder Prozess kennt alle anderen Prozesse

Die im Algorithmus verwendeten Nachrichten:

- election: Wahlvorschlag an die anderen Teilnehmer
- answer: Antwort auf election-Nachricht an den Vorschlagenden
- coordinator(*k*): Mitteilung, dass Prozess mit ID *k* gewählt wurde

Der Prozess, der den Ausfall des bisherigen Koordinators feststellt, initiiert die Wahl. Dies können mehrere gleichzeitig sein. Der Ablauf des Wahlalgorithmus ist dann:

1) Initiator pi schickt election-Nachricht an alle Prozesse mit größerer ID als pi. Dies ist möglich, da alle Prozesse bekannt sind (gleichwohl ist natürlich nicht bekannt, welche davon noch aktiv sind)
2) Falls pi bis zum Timeout keine Antwort erhält, so sendet er eine Nachricht coordinator(ID(pi)) an alle Prozesse mit kleinerer ID als pi. Erhält pi eine Antwort von einem Prozess pk mit größerer ID, so führt pk Schritt 1) des Algorithmus aus.

Ein korrekter Prozess antwortet auf jede election-Nachricht mit einer answer-Nachricht.

Wenn ein Prozess pk eine coordinator-Nachricht erhält, so setzt er seine Variable elected(k) auf die in der Nachricht enthaltene ID.

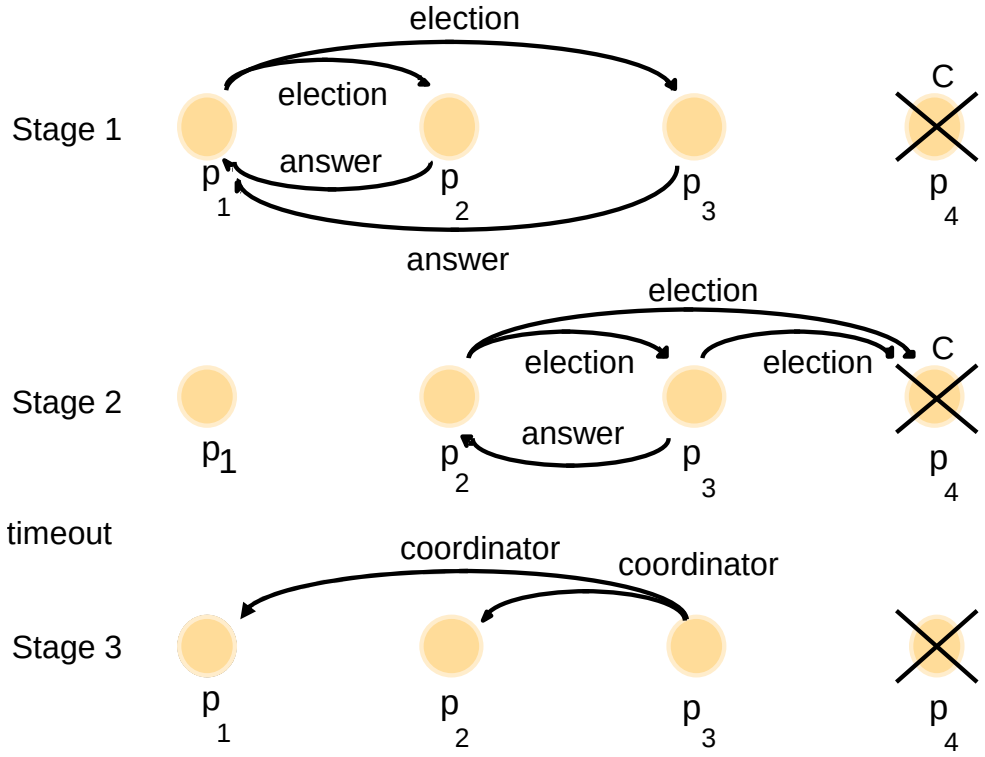


Bild 6.15 Bully-Wahlalgorithmus

Im Beispiel in Bild 6.15 initiiert p1 die Wahl, nachdem p4 ausgefallen ist und p1 dies zweifelsfrei festgestellt hat. In diesem Bild sei die Prozessnummer zugleich die ID. In der 1. Phase erreicht p1 die Prozesse p2 und p3 und – so die Annahme – erhält eine Answer-Nachricht von ihnen. P1 ist damit ausgeschieden.

P2 und p3 arbeiten nach dem gleichen Algorithmus weiter; während p2 von p3 eine Nachricht erhält und somit ebenfalls ausscheidet, erkennt p3 nach dem timeout, dass er zum Koordinator geworden ist. P3 sendet also die coordinator-Nachricht an alle.

Die Bewertung dieses Algorithmus:

- E1: Im Prozess pi gilt zu jedem Zeitpunkt elected(i) = *null* oder elected(i) = P, wobei P der am Ende der Wahl lebende Prozess mit der größten ID ist. (safety)
- E2: Alle teilnehmenden Prozesse setzen irgendwann ihre Wahlvariable auf P – oder sie fallen aus (liveness)

Probleme:

- E1 ist nur erfüllt, wenn ausgefallene Prozesse nicht mit Prozessen gleicher ID ersetzt werden!
- Timeouts erfordern synchrones System

Performance-Analyse:

- Worst-Case: Prozess mit kleinster ID startet Wahl. Dann $O(N^2)$ Nachrichten, da jeder andere Prozess Wahl startet. Zeitbedarf: 3+2 Nachrichtenübertragungen (*election*, answer, nochmals election, gefolgt von Timeout bei Warten auf *answer*, *coordinator*). Diese Schätzung gilt natürlich nur, wenn alle Nachrichtenübertragungen parallel ausgeführt werden.
- Best-Case: Neuer Koordinator bemerkt Crash des bisherigen Koordinators. Dann nur N-2 coordinator-Nachrichten notwendig. Zeitbedarf: 1+2 Nachrichtenübertragungen

6.4 Multicast-Kommunikation

Viele Algorithmen in verteilten Systeme erfordern Kommunikation mit Gruppen anderer Prozesse. Beispiele hierfür stammen z.B. aus dem Bereich der Uhren- bzw. Zeitsynchronisation und der Wahlalgorithmen:

- Berkeley-Algorithmus
- Ricart/Agrawala Algorithmus
- Bully-Algorithmus

Multicast ist dabei eine komplexe Operation, und oft wünscht man sich gewisse Zusagen an Multicast-Kommunikation (z.B. Verlässlichkeit, Reihenfolge der Auslieferung). Da aber standardmäßig nur IP-Multicast, der UDP-basiert arbeitet, zur Verfügung steht, ist Nachrichtenzustellung nicht garantiert: UDP/IP verwenden nicht bestätigte, verbindungslose Kommunikation.

6.4.1 Systemmodell für Multicast

Um eine Bewertung für bzw. Modellierung von Multicast-Algorithmen durchführen zu können werden Basisfunktionen definiert, die dann in den Modellen entsprechend genutzt werden. Folgende Basisfunktionen gelten hier als definiert:

- Die Operation `multicast(g, m)` sendet die Nachricht *m* zu allen Mitgliedern der Gruppe *g*.
- Die Operation `deliver(m)` liefert eine Multicast-Nachricht beim Empfängerprozess aus
- Jede Nachricht m enthält Informationen über den Ursprung `sender(m)` und das Ziel `group(m)`.

6.4.2 Einfacher Multicast

Im ersten Schritt wird ein Multicast entwickelt bzw. beschrieben, bei dem zugesichert ist, dass ein korrekter Prozess die Nachricht irgendwann erhält, sofern der Sender (sendende Prozess) nicht ausfällt. Dieser Algorithmus wird als Basic-Multicast (B-Multicast) bezeichnet.

Im Kern besteht der Algorithmus ausschließlich darin, dass verlässlich Unicast-Send-Operationen verwendet werden:

- B-Multicast einer Nachricht m: Für jedes $p \in$ group(m): `send(p, m)`
- Wenn eine Nachricht m bei p ankommt: `B-deliver(m)` bei p

Der größte (und wohl einzige) Vorteil besteht in der Einfachheit dieses Algorithmus. Es existieren aber auch Nachteile:

- Eine Acknowledgement-Implosion ist notwendig, um die Bestätigung der Unicast-Operationen in eine für die Multicast-Operation zu überführen.
- Die Nutzung der vorhandenen Bandbreite (des Netzes) ist schlecht, da die Nachricht ständig wiederholt wird. Die Zahl der Kommunikationen wächst mit O(n).

- Insbesondere bei großen Gruppen kommt es zu einer schlechten Skalierung

Eine alternative Implementierung über IP-Multicast ist ebenfalls möglich. Hier können mithilfe von Sequenznummern fehlende Nachrichten erkannt und angefordert werden, und es entsteht keine ACK-Implosion.

6.4.3 Verlässlicher Multicast

Einer der Nachteile beim B-Multicast ist es, dass der Sender zwischenzeitlich ausfallen kann und dann eine Teilmenge der Gruppe die Nachricht besitzt, eine andere aber nicht. Es war die Grundannahme des B-Multicast, dass der Ausfall nicht passieren würde, aber die Realität spricht dafür, diese Annahme zu verwerfen und einen verlässlichen Multicast ohne diese Annahme zu konzipieren.

Die Kriterien für einen verlässlichen Multicast sind:

- Jeder korrekte Prozess p liefert eine Nachricht m höchstens einmal aus (at-most-once-delivery). In diesem Fall ist p ∈ group(m), und m wurde von sender(m) durch Multicast verschickt. (Integrity)
- Wenn ein korrekter Prozess eine Nachricht m per Multicast versendet oder empfängt, so liefert er sie auch irgendwann aus. (Validity)

Diese Kriterien stammen aus der verlässlichen 1-zu-1-Kommunikation. Neu ist:

- Wenn ein korrekter Prozess eine Nachricht m ausliefert (nachdem sie zugestellt wurde), so wird diese auch allen anderen korrekten Prozesse in group(m) zugestellt. (Agreement)

```
Init: Received := {};   // Leere Menge
R-multicast(g, m)in process p:
     B-multicast(g, m)
     On B-deliver(m)in process q with g = group(m):
          If (m not in Received) {
               Append m to Received;
               If (q is not p)
                    { B-multicast(g, m); }
               R-deliver(m);
          }
```

Bild 6.16 Algorithmus für verlässlichen Multicast (bei Benutzung eines abstrakten B-Multicast)

Der in Bild 6.16 dargestellte Algorithmus besteht im Kern daraus, dass jeder teilnehmende Prozess eine Liste über die gelieferten Nachrichten pflegt. Wird eine Nachricht m bei einem Prozess empfangen, wird sie irgendwann ausgeliefert (so die Annahme), wenn sie noch nicht in der Received-Liste steht. Ansonsten wird die Nachricht verworfen.

Wird m nun ausgeliefert, wird sie zugleich allen anderen Prozessen per Multicast geliefert. Dies ist eine hohe Redundanz (die Anzahl der Sendungen ist $O(N^2)$), die aber auch eine hohe Ausfalltoleranz bietet. Für den in Bild 6.16 gezeigten Algorithmus für einen verlässlichen Multicast gelten folgende Eigenschaften:

- Integrity
  Der if-Block kann pro Nachricht höchstens einmal durchlaufen werden, und wird nur bei den Empfängerprozessen durchgeführt
- Validity
  Wegen p = sender(m) ∈ g, liefert p die Nachricht m aus.
- Agreement
  Annahme: Korrekter Prozess q führt R-deliver(m) nicht aus
  ⇒ B-deliver(m) wurde bei q nicht bzw. unvollständig (B-multicast(m)) ausgeführt

⇒ kein anderer korrekter Empfängerprozess hat B-deliver(m) ausgeführt (denn dieses enthält B-multicast(m))
⇒ kein anderer korrekter Empfängerprozess führt R-deliver(m) aus

6.4.4 Verlässlicher Multicast über IP

Der verlässliche Multicast mithilfe eines abstrakten B-Multicast zeigt einen sehr hohen Kommunikationsaufwand, da jede Nachricht |g|-mal gesendet wird. Ein anderer Ansatz besteht darin, über den IP-Multicast zu kommunizieren. Die darin enthaltenen Grundideen sind:

- Die meisten Nachrichten werden korrekt zugestellt, den Rest übernimmt ein zusätzlich eingeführtes Acknowledgement (ACK). Dies führt zu einer starken Reduktion der Redundanz, wobei die Verlässlichkeit trotzdem erhalten bleibt.
- Acknowledgements werden – pro Sender – in den nächsten Nachrichten an die Gruppe integriert.
- Non-Acknowledgements (NAK) werden als explizite Nachrichten versendet

Bei einer geschlossenen Gruppe stellt sich das so dar, dass jeder Prozess seine eigenen Sendungen in Form einer Nummer protokolliert und die im Paket steckenden Sendenummern – für den empfangenden Prozess natürlich eine Empfangsnummer – ebenfalls speichert. Im Einzelnen:

- Jeder Prozess *p* verwaltet seine Sequenznummern in der Form S(p, g) für die von ihm gesendeten und R(q, g) für die zuletzt vom Prozess q stammende Nachricht, die zugestellt wurde.
- Wenn p eine Nachricht m R-multicastet, so schickt p den Tupel {m, S(p, g) und alle Acknowledgements <q, R(q, g)>} an die Multicastgruppe. Diese beinhalten die Nummer der letzten von anderen Prozessen erhaltenen (und zugestellten) Nachrichten.
- Nach Multicast wird der Zähler S(p, g) um 1 erhöht.

Beim Empfang (R-delivery) einer Nachricht von q mit Sequenznummer S gilt folgendes Vorgehen:

- Wenn S == R(q, g) +1 gilt, wird R(q, g) um 1 erhöht.
- Ist S <= R(q, g), so hat *p* die Nachricht schon ausgeliefert, diese wird daher ignoriert (Duplikat).
- Ist S > R(q, g) + 1, so fehlen p noch frühere Nachrichten von q. Diese werden bei q angefragt (z.B. durch explizites NAK), und die Nachricht mit Nummer S kommt in einen Puffer (Verzögerung der Auslieferung, siehe auch FIFO-Multicast).
- Gilt R > R(r, g) für irgendeines der von q mitgeschickten Acknowledgements <r, R>, so fordert p die fehlenden Nachrichten von r an.

Integrity: Folgt aus Entdeckung von Duplikaten und der Checksumme von IP-Multicasts über UDP

Validity: Folgt aus Eigenschaft von UDP-Multicasts

Agreement:

- Anforderung 1: Prozesse können fehlende Nachrichten entdecken → Bei Annahme eines endlosen Nachrichtenaustauschs möglich
- Anforderung 2: Es existiert stets eine Kopie einer verlorenen Nachricht → Alle Prozesse müssen Liste der ausgelieferten Nachrichten halten

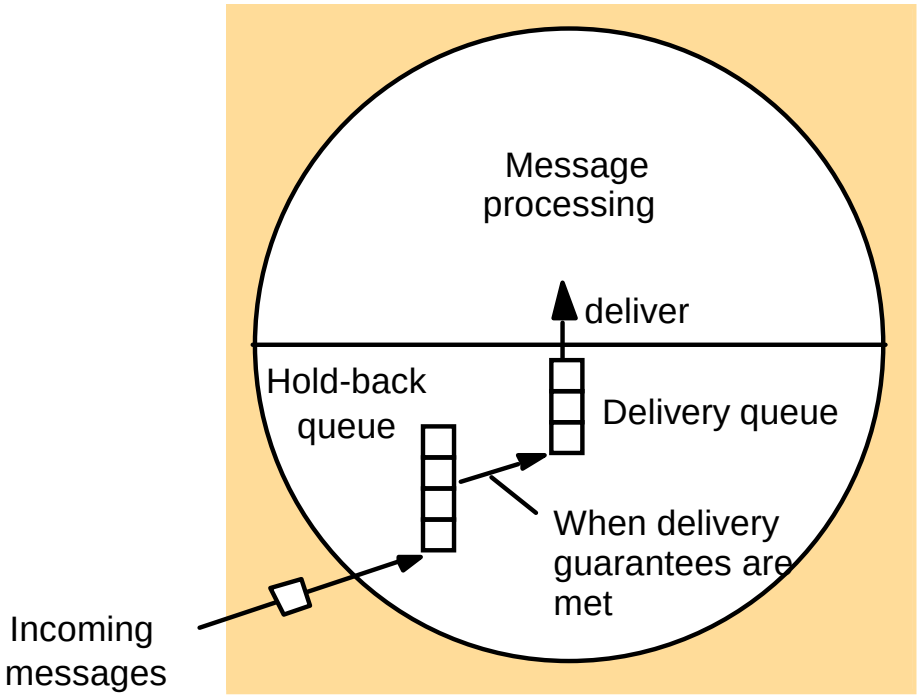


Bild 6.17 Delivery (mit Holdback) im Empfängerrechner

6.4.5 Reihenfolge von Nachrichten im Multicast

Ein gänzlich anderes Problem stellt sich, wenn die Reihenfolge der eingehenden (oder ausgehenden) Nachrichten wichtig ist. Sowohl im B-Multicast als auch im R-Multicast gilt, dass die Reihenfolge der Zustellung nicht gewährleistet werden kann. Häufig braucht man dies aber in garantierter Form.

- Häufig gewünschte Reihenfolgen sind:
  - FIFO: Wenn ein korrekter Prozess zunächst multicast(g, m) und dann multicast(g, m‘) ausführt, dann liefert jeder korrekte Prozess aus g, der m‘ ausliefert, m vor m‘ aus.
  - Logische Sortierung (Kausalordnung): Wenn multicast(g, m) → multicast(g, m‘) mit → als „happens-before“-Operator als logische Reihenfolge gilt, dann liefert jeder korrekte Prozess aus g, der m‘ ausliefert, m vor m‘ aus.
  - Totale Ordnung (Vollordnung): Wenn ein korrekter Prozess eine Nachricht m vor einer Nachricht m‘ ausliefert, dann liefert jeder korrekte Prozess aus g, der m‘ ausliefert, m vor m‘ aus.

Zunächst sieht es so aus, dass diese Ordnungen in etwa das gleiche bedeuten; das ist aber nicht der Fall, wie in den folgenden Abschnitten noch dargelegt werden wird. In jedem Fall gilt:

- Die logische Sortierung impliziert FIFO
- Folgende Kombinationen sind denkbar: total + FIFO, total + logisch
- Terminologie: „atomic multicast“ = verlässlich und totale Ordnung

6.4.6 FIFO-Multicast

Im FIFO-Multicast wird lediglich die lokale Sendereihenfolge garantiert. Dies kann wie folgt implementiert werden:

- Prozess *p* hat Sequenznummer S(p, g) und (für jeden anderen Prozess) eine Empfangs-Sequenznummer R(q, g).
- Beim FIFO-Multicast einer Nachricht m wird über B-Multicast die Nachricht m sowie die Sequenznummer S(p, g) verschickt. Dann wird S(p, g) um 1 erhöht.
- Empfängt *p* eine Nachricht von *q* mit Nummer S > R(q, g) +1, so legt er diese in eine Warteliste. Nur Nachrichten mit S = R(q, g) +1 werden FIFO-ausgeliefert, dann wird R(q, g) um 1 erhöht.

Variante: Dies kann statt über B-Multicast auch über R-Multicast realisiert werden.

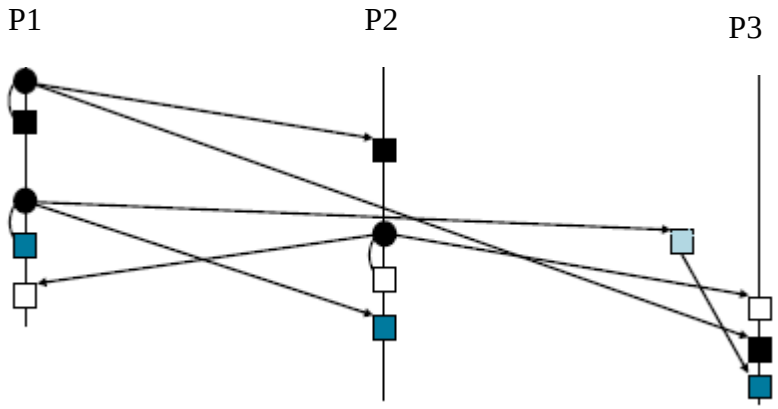


Bild 6.18 FIFO-Multicasts

6.4.7 Kausalordnung für Multicast

Die Kausalordnung garantiert die Zustellung nach der Reihenfolge, die durch die Relation → festgelegt wird. Die Implementierung über Basic Multicast ist wie folgt:

- Vereinfachung: geschlossene, nicht-überlappende Gruppen
- Basiert auf Zeitstempel-Vektor V
- Vp[q] ist die Anzahl der Nachrichten von Prozess *q*, von denen die nächste Nachricht von Prozess *p* kausal abhängt. Wenn über Reliable Multicast anstelle des Basic Multicast implementiert: **Reliable Multicast mit Kausalordnung**!

```
Initialisierung: Vp[i] := 0 für alle i
CO-multicast(g, m):
    Vp[p] := Vp[p] + 1;
    B-multicast(g, <Vp, m>);
On B-deliver(<Vq, m>): (Nachricht m von Prozess q empfangen)
    Store message in Queue;
    Wait until Vq[q] = Vp[q] + 1 and Vq[k] ≤ Vp[k] for all k ≠ q;
    Remove message from Queue;
    CO-deliver m;
    Vp[q] := Vp[q] + 1;
```

Bild 6.19 Pseudocode für Kausal-geordnetes Multicasting

Das Prinzip ist also sehr einfach: Man bestimme die Anzahl der Nachrichten, die vor einer Nachricht kommen müssen, und warte auf diese. Die Kausalordnung ist damit eine Halbordnung.

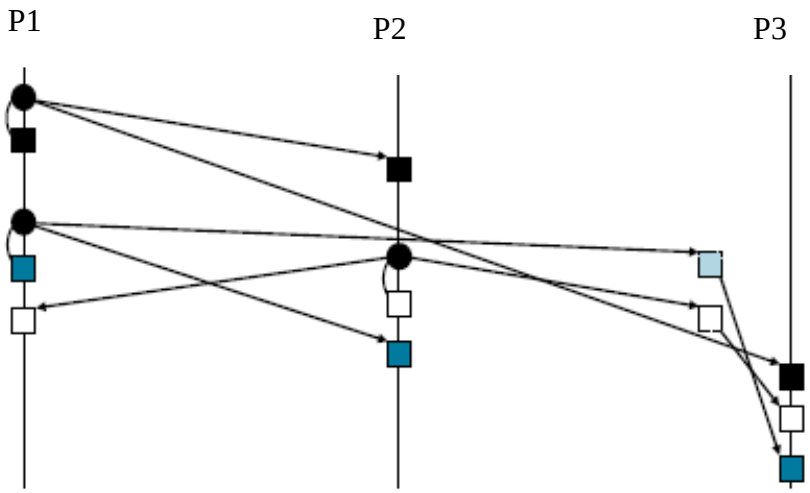


Bild 6.20 Kausalgeordneter-Multicasts

6.4.8 Totale Ordnung für Multicast

Die totale oder Vollordnung garantiert die Empfangsreihenfolge über alle Prozesse der Gruppe. Dies heißt, dass wenn beispielsweise zwei Prozesse an zwei andere Prozesse etwas senden, die Nachrichten bei beiden in gleicher Reihenfolge zugestellt werden. Die Reihenfolge allerdings ist willkürlich. Bild 6.21 zeigt ein Beispiel.

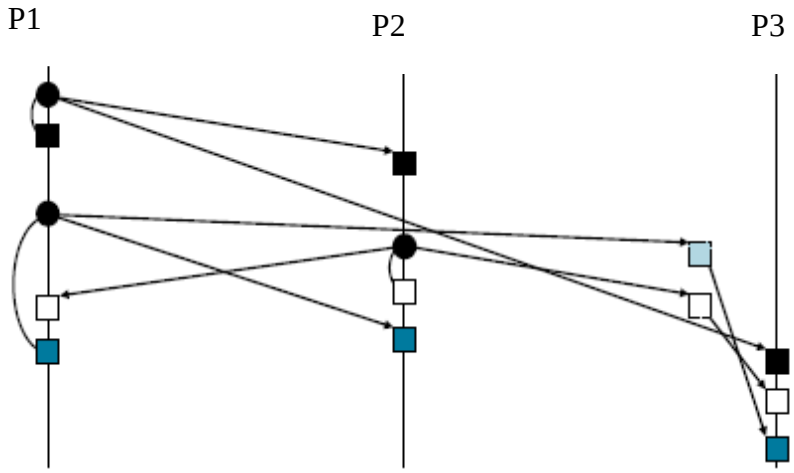


Bild 6.21 Vollordnung

Man beachte dabei, dass die Vollordnung weder FIFO noch die Kausalordnung beinhaltet.

Bei einer Implementierung über den Basic Multicast wird folgendes Verfahren gewählt:

- Zuweisung von (voll-)geordneten Sequenznummern
- Empfangs-Algorithmus ist dann ähnlich dem der FIFO-Ordnung
- Allokation der Sequenznummern:
  - Variante 1: zentral (Sequencer)
  - Variante 2: dezentral

Allokation von Sequenznummern, Variante 1: Sequencer

- Zustellungsreihenfolge der Nachrichten bei einem einzelnen Prozess, dem *Sequencer*, bestimmt Reihenfolge der gesamten Gruppe.
- Dies lässt sich durch folgenden Algorithmus implementieren:

```
Initialisierung:
      r_g := 0
TO-multicast(g, m):
      B-multicast(g ∪ sequencer(g), <m, id(m)>);
On B-deliver(<m, id>):
      store <m, id> in queue;
On B-deliver(<„order“, id, S>):
      Wait until <m, id> in queue and S = r_g;
      Remove entry <m, id> from queue;
      TO-deliver m;
      r_g := S + 1
```

Bild 6.22 Algorithmus für die Mitglieder der Gruppe g

```
Initialisierung:
      s_g := 0
On B-deliver(<m, i>):
      B-multicast(g, <„order“, i, s_g>);
      s_g := s_g + 1;
```

Bild 6.23 Algorithmus für den Sequencer

Allokation von Sequenznummern, Variante 2: dezentral

Das Prinzip dieses Algorithmus (ISIS) besteht darin, bei allen anderen Mitgliedern der Gruppe eine Sequenznummer abzufragen (Vorschläge) und daraus die größte auszuwählen:

- Jeder Prozess *q* hat zwei Zähler:
  - A_q ist die größte Sequenznummer, die er bisher beobachtet hat
  - P_q, die größte von *q* vorgeschlagene Nummer
- Algorithmus:
  - Prozess *p* B-Multicastet <m, id(m)>, wobei id(m) eine ID für m ist.
  - Jeder Prozess *q* antwortet mit einem Vorschlag für eine Sequenznummer ($P_q := Max(A_q, P_q)+1$) und speichert die Nachricht m mit Vorschlagsnummer P_q in einer lokalen Warteliste, die nach kleineren Vorschlagsnummern sortiert ist.
  - *p* sammelt alle Vorschläge ein und nimmt deren Maximum a als Sequenznummer. *p* B-multicastet <i, a>. Jeder Prozess *q* in g setzt
    $A_q := Max(A_q, a)$ und versieht m in seiner Liste mit Sequenznummer a. Wenn eine Nachricht, die in der lokalen Warteliste vorne steht, eine Sequenznummer hat, so wird diese TO-ausgeliefert.

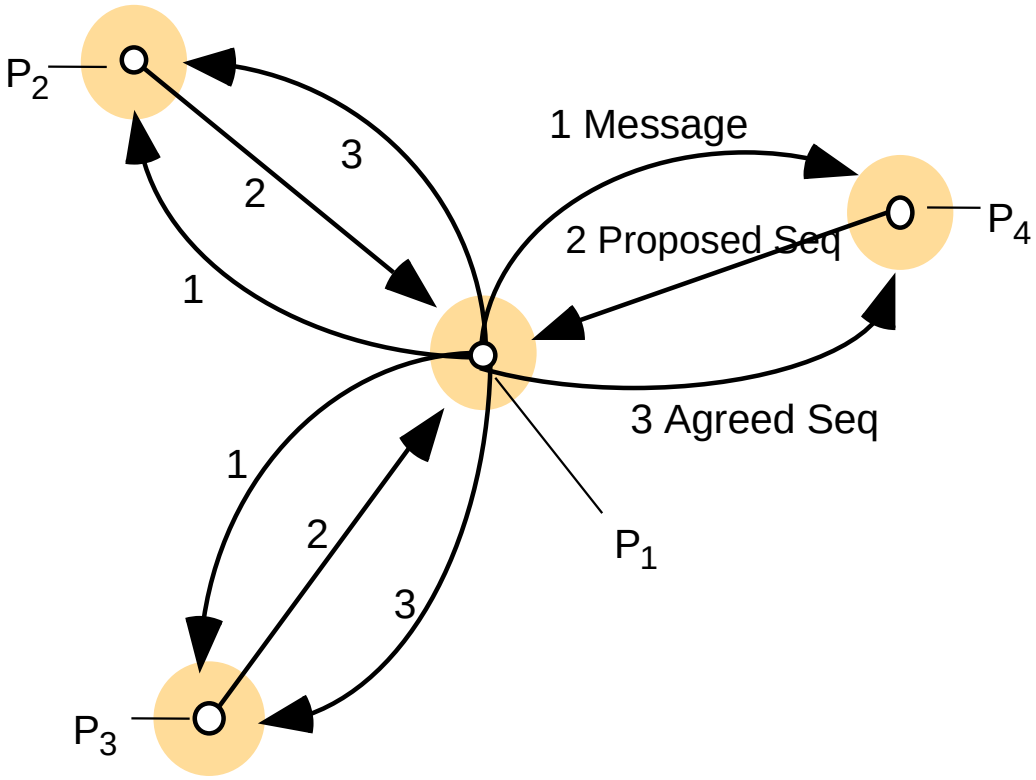


Bild 6.24 Dezentraler Algorithmus für totale Ordnung

Nachzuweisenist hierbei die Eigenschaft der totalen Sortierung – warum kann kein Prozess eine Nachricht „zu früh“ ausliefern?

Sei m1 eine Nachricht, die in der lokalen Warteliste vorne steht und eine Sequenznummer bekommt. Jede Nachricht, die nach diesem Zeitpunkt in die Warteliste kommt, hat eine höhere Vorschlagsnummer (und damit eine höhere Sequenznummer) als m1.

Sei also m2 eine Nachricht, die schon in der Liste steht. Dann gilt:

- Sequenznummer(m2) >= Vorschlagsnummer(m2)
- Vorschlagsnummer(m2) > Sequenznummer(m1)

  ⇒ Sequenznummer(m2) > Sequenznummer(m1)

Also muss m2 nach m1 ausgeliefert werden.

Vor- und Nachteile der einzelnen Lösungen:

Sequenzer-basierte Methode

+ Einfach zu implementieren

– Flaschenhals

– Single Point of Failure

ISIS

– Skaliert besser für große Gruppen

– Höhere Latenz

Beobachtungen

- Vollordnung über B-multicast mit FIFO-Ordnung ⇒Kausalordnung
- Ein zuverlässiger Multicast mit Vollordnung ist in einem asynchronen System unmöglich!

Hieraus kann man entsprechend kombinieren. Hierzu einige Anmerkungen:

1. Der Algorithmus kann auch über R-Multicast realisiert werden

   →Verlässlicher Multicast mit logischer Ordnung

2. Kombination mit Sequencer ist möglich

   → Multicast mit totaler und logischer Ordnung

3. Verlässlicher Multicast mit totaler Ordnung ist in asynchronen verteilten Systemen unmöglich

6.5 Nebenläufigkeitskontrolle

6.5.1 Problemstellung

Problemsituation: Parallele Prozesse, die auf gemeinsam genutzten Daten operieren, können sich wechselseitig beeinflussen. Dies kann am Beispiel einer Bank-Anwendung gezeigt werden:

Kontostand: 100€

Kein Dispo-Kredit

Client 1: if balance > 80 then withdraw(80)

Client 2: if balance > 60 then withdraw(60)

Was kann passieren, wenn Clientprozesse 1 und 2 parallel laufen?

Zur Untersuchung der Effekte nebenläufiger Prozessausführung bietet es sich an, folgende Basisoperationen zu verwenden:

- Lesen des Zustands eines Objekts (z.B. Inhalt einer Variable)
  - read(object)
- Schreiben des Zustands eines Objekts
  - write(object, value)
- Operationen, die zu verschiedenen Tasks (=Prozessen) gehören, werden durch Indizes unterschieden
  - read1(object) // Task 1 liest object
- Wichtig: Leseoperationen ≠ Schreiboperationen. Diese Operationen sind oft im Code vermischt (so genannte read-modify-write-Operationen).
- Beispiel:
- a += x
- Basisoperationen:
  - read(a)
  - read(x)
  - write(a,_)
- Arithmetische Operationen können ignoriert werden

Im obigen Bankbeispiel ergibt dies:

```
Task 1:                          Task 2:
a := read1(account)              a := read2(account)

if( a > 80 )                     if( a > 60 )
      write1( account, a-80 )          write2( account, a-60 )
```

Bild 6.25 Codebeispiel für nebenläufige Prozesse mit Wechselwirkung

Folgende Varianten können bei zufälliger, nebenläufiger Ausführungen auftreten:

1. Nur Task 1 wird durchgeführt
2. Nur Task 2 wird durchgeführt
3. Beide Tasks werden durchgeführt, Kontostand danach 40€
4. Beide Tasks werden durchgeführt, Kontostand danach 20€

Das Ergebnis ist also:

- Das endgültige Ergebnis ist nicht exakt vorhersehbar
  „Race condition“ – schwierig zu debuggen!
- Name für dieses Problem: „**Lost Update**“
  (Mehrere Tasks lesen ein Objekt bevor sie es schreiben)

Nächstes Problem: **Inconsistent Retrieval**. Hierunter wird eine inkonsistente Datenabfrage und –veränderung verstanden. Folgendes Beispiel sei hierzu gegeben:

Eine finanzielle Transaktion hebt von Konto a 100 Geldeinheiten ab und zahlt auf Konto b diese 100 Einheiten wieder ein. Dazwischen wird durch Transaktion W der Gesamtkontozustand abgefragt. Das Ergebnis wird, wenn die Reihenfolge aus Bild 6.26 eingehalten wird, um 100 Einheiten zu niedrig sein

Transaction *V*:	**Transaction *W*:**
a.withdraw(100) *b.deposit(100)*	*aBranch.branchTotal()*
a.withdraw(100);	
	total = a.getBalance()
	total = total+b.getBalance()
	total = total+c.getBalance()
b.deposit(100)	⋮

Bild 6.26 Aktionsreihenfolge zweier nebenläufig ablaufender Transaktionen

Grund für das Problem ist hier, dass Daten, die von nicht „vollendeten“ Tasks geschrieben wurden, gelesen werden.

Nächstes Beispiel: Online-Ticket-Verkauf.

- Funktionsprinzip der Applikation:
  - Serverfunktion is_ticket_available() wird von Client aufgerufen
  - Entscheidung des Users (bei Client-Applikation) wird erwartet
  - Serverfunktion buy_ticket() wird von Client aufgerufen

- Basisoperationen beim Client:
  - read(ticket)
  - input()
  - read(ticket)
  - write(ticket)
- Situation: Zwei Clients parallel, nur noch ein Ticket verfügbar

```
Task 1:                    Task 2:
a := read1(ticket)         a := read2(ticket)
input()                    input()
a := read1(ticket)         a := read2(ticket)
write1 (ticket)            write2(ticket)
```

Bild 6.27 Nebenläufiger Ablauf bei Online-Ticketverkauf

In diesem Fall versuchen zwei Benutzer unabhängig voneinander, ein Ticket zu bekommen. Dies kann zu folgenden Ergebnissen führen:

1. Nur Task 1 sieht ein verfügbares Ticket und bekommt es
2. Nur Task 2 sieht ein verfügbares Ticket und bekommt es
3. Beide Tasks sehen ein verfügbares Ticket, aber nur einer bekommt es.

- Das endgültige Ergebnis ist nicht exakt vorhersehbar, da die Tasks nicht isoliert sind
- Name für dieses Problem: „Unrepeatable reads“ (Mehrfaches Lesen eines Objekts führt zu verschiedenen Ergebnissen)

6.5.2 Verfahren zur Nebenläufigkeitskontrolle

Es wird also nach einem Verfahren zur Nebenläufigkeitskontrolle gesucht, das dieses Problem löst. (Randbemerkung: Auch in den Hardwarebeschreibungssprachen muss man dieses Problem lösen, wobei die Lösung dort anders aussieht!). Bei den Verteilten Systemen gilt:

- Allgemeines Prinzip: möglichst viel Nebenläufigkeit erlauben (nicht: Sperren des gesamten Buchungssystems, weil ein User sich verfügbare Tickets für einen Flug ausgeben lässt)
- Grundmechanismus zur Lösung dieses Problems: Zusammenfassung von Aktionen zu Transaktionen. Prinzipien dabei:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Das Ganze führt zur Frage der *seriellen Äquivalenz*:

- Frage: Wann ist eine Reihenfolge von Operationen akzeptabel?
- Antwort: Wenn das Ergebnis der Ausführung dieser Operationen dem entspricht, was bei Nacheinanderausführung der Transaktionen *in irgendeiner Reihenfolge* herauskommt (=serielle Äquivalenz)
- *Lost Updates*, *Inconsistent Retrievals* und *Unrepeatable Reads* können bei Nacheinanderausführung nicht auftreten. Daher auch kein Problem für seriell äquivalente Überlappungen.

- Nicht passend zur Definition von Serialisierbarkeit aus DB-Vorlesung
- Grund: Unterschiedliche Definition (View-Serialisierbar vs. Konflikt-Serialisierbar)

Definition: Zwei Schedules S1 und S2 werden *View-äquivalent* genannt, wenn folgende Bedingungen erfüllt sind:

1. Wenn die Transaktion *Ti* in S1 einen initialen Wert für das Objekt X liest, dann macht das die Transaktion *Ti* in S2 ebenfalls.
2. Wenn die Transaktion *Ti* in S1 einen von *Tj* in S1 für Objekt X geschriebenen Wert liest, dann macht das auch Transaktion *Ti* in S2.
3. Wenn Transaktion *Ti* in S1 die finale Transaktion zum Schreiben eines Werts für ein Objekt X ist, dann ist es die Transaktion *Ti* in S2 ebenfalls.

Definition: Ein Schedule wird *View-serialisierbar* genannt, wenn es View-äquivalent für mindestens einen seriellen Schedule ist.

Definition: *Konflikt-Aquivalenz.* Die Schedules S1 und S2 sind Konflikt-äquivalent, wenn die folgenden Bedingungen erfüllt sind:

1. Beide Schedules S1 und S2 besitzen den gleichen Satz von Transaktionen (inklusive der Ordnung der Aktionen innerhalb jeder Transaktion).
2. Die Ordnung aller Paare von Konflikt-Aktionen in S1 und S2 sind die gleichen.

Definition: Ein Schedule S wird Konflikt-serialisierbar genannt, wenn dieser Konflikt-äquivalent zu einem oder mehreren seriellen Schedules ist.

- Es gilt: Konflikt-Serialisierbar ⇒ View-Serialisierbar.
- Es gibt Abfolgen (mit blind writes), welche View-Serialisierbar aber nicht Konflikt-Serialisierbar sind
- Der Nachweis der View-Serialisierbarkeit ist NP-Vollständig – daher wenig praktische Relevanz.

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ & R(A) \\ W(B) & \\ Com. & \\ & W(A) \\ & Com. \end{bmatrix} \qquad H = \begin{bmatrix} T1 & T2 & T3 \\ R(A) & & \\ & W(A) & \\ & Com. & \\ W(A) & & \\ Com. & & \\ & & W(A) \\ & & Com. \end{bmatrix}$$

Bild 6.28 a) Konflikt-serialisierbarer Schedule (Lösung: <T1, T2>, nicht aber <T2, T1>
b) Nicht-konflikt-serialisierbarer Schedule (T2 führt „blind write“ aus)
(com.: committing)

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(balance*1.1)*	*b.setBalance(balance*1.1)*
a.withdraw(balance/10)	*c.withdraw(balance/10)*
balance = b.getBalance()	
*b.setBalance(balance*1.1)*	
	balance = b.getBalance()
	*b.setBalance(balance*1.1)*
a.withdraw(balance/10)	
	c.withdraw(balance/10)

Bild 6.29 Beispiel für Konflikt-serialisierbare Schedules

6.5.3 Serielle Äquivalenz

Operationskonflikte:

Operations of transaction		*Conflic*	*Reaso*
read	*read*	No	Because the effect of a pair of *reads* does not depend on the order in which they execute
read	*write*	Yes	Because the effect of a *read* and a *write* depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *writes* depends on the order of their execution

Bild 6.30 Operationskonflikte

Definition: *Serielle Äquivalenz*. Für zwei Transaktionen ist serielle Äquivalenz genau dann gegeben, wenn alle Paare von Operationen, die in Konflikt miteinander stehen, stets in der gleichen Reihenfolge ausgeführt werden (für alle Objekte, die von beiden Transaktionen verwendet werden).

Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Bild 6.31 Beispiel für Konflikt verursachende Operationen

In dem vorliegenden Beispiel gibt es zwei Konflikt verursachende Paare: write(i, 10) in T mit read(i) in U sowie write(j, 30) in U mit write(j, 20) in T. Hier stellt sich direkt die Frage, wie die serielle Äquivalenz von Transaktionen implementiert werden kann.

Hierzu folgende Lösungsansätze:

1. Sperren von Objekten (Prinzip: Nur ein Prozess erhält Zugang zu Objekt, „Distributed Mutual Exclusion“)
2. Optimistische Verfahren (Prinzip: Führe alle Änderungen durch, entdecke und löse Konflikte wenn notwendig)
3. Zeitstempelverfahren (Prinzip: Finde durch Zeitstempel heraus, ob Ausführung von Operationen serielle Äquivalenz erhält)

Sperren von Objekten:

In folgendem Bankbeispiel werden teilweise Objekte gesperrt, d.h. nur die sperrende Task erhält Zugang zu dem Objekt:

```
lock(account)
a := read(account)
if(a > X)
       write(account, a – X)
unlock(account)
```

Bild 6.32 Code-Beispiel für lock/unlock-Operation

Damit wird die Liste der Basisoperationen um zwei erweitert:

- lock (wartet bis Objekt verfügbar, setzt Sperre)
- unlock (gibt Sperre frei)

```
lock1( account )
a := read1( account )
if(a > 80)
       write1( account, a-80 )
unlock1( account )
```

```
lock2( account )
a := read2( account )
if(a > 80)
       write2( account, a-80 )
unlock2( account )
```

Bild 6.33 Nebenläufige Transaktionen mit lock/unlock

Transaction *T*		**Transaction *U***	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A, B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B, C*

Bild 6.34 Nebenläufige Transaktionen mit lock/unlock-Operation

6.5.4 Sperrverfahren für Transaktionen

Das Sperrverfahren ist bereits ein gutes Instrument, um die Ausführung von nebenläufigen Prozessen „sicher" zu machen. Die Frage ist nun allerdings, ob es zur Erlangung von serieller Äquivalenz ausreicht, jedes Objekt so lange durch eine Transaktion zu sperren, wie es durch diese Transaktion benötigt wird, und es dann direkt freizugeben.

Damit wäre folgende beispielhafte Reihenfolge möglich:

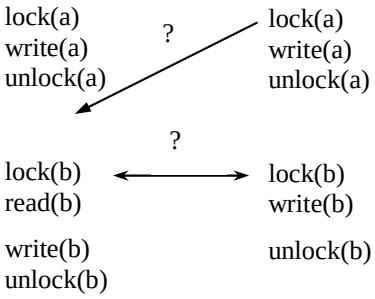


Bild 6.35 Sperrverfahren für Transaktionen mit beliebigem Unlock

Serielle Äquivalenz erfordert nun, dass eine Transaktion nach dem ersten Freigeben einer Sperre keine neue setzen darf. Bild 6.35 zeigt eine Verletzung dieser Forderung. Weitere Probleme können durch so genannte Dirty Reads (= frühzeitiges Auslesen, wobei die Wert-setzende Transaktion noch nicht vollständig beendet ist und abgebrochen werden kann) entstehen.

Als Lösung bietet sich ein *2-Phasen-Locking* an: In der *Growing Phase* sind Lock-Basisoperationen erlaubt, in der *Shrinking Phase* dann nur noch Unlock-Operationen (und natürlich andere).

Das einfache 2-Phasen-Locking kann also beispielhaft folgende Gestalt annehmen:

lock(a)
write(a)

lock(b)
read(b)
lock(c)

unlock(a)
write(c)
unlock(c)
write(b)
unlock(b)

In dem weitergehenden strikten 2-Phasen-Locking wird gefordert, dass die Freigaben erst am Ende aller anderen Operationen erfolgen. Im Beispiel also wie folgt:

lock(a)
write(a)

lock(b)
read(b)

lock(c)
write(c)

write(b)

unlock(a)
unlock(c)
unlock(b)

Die Vorteile dieses Verfahrens sind:

- Kein Abbruch von Transaktionen wg. Lesens ungültiger Daten (*Dirty Read*) notwendig
- Automatische Locks/Unlocks möglich

6.5.5 Weiterentwicklung: Verschiedene Arten von Sperren

Bei der bisherigen Verwendung von Locks wird die Nebenläufigkeit von Transaktionen in gewissen Konstellationen unnötig eingeschränkt. Mehrere Reads stehen z.B. nicht in Konflikt zueinander.

Beispiel:

T	U
read(a)	read(d)
write(b)	read(a)
write(c)	write(d)

Bild 6.36 Beispiel für unnötige Sperren

Mögliches Lösungsverfahren: unterschiedliche Sperren für Lese- und Schreibzugriffe

- „Many reads / one write“: Lesesperren können von mehreren Transaktionen gemeinsam gehalten werden.
- „Lost Updates“, „Unrepeatable Reads“, „Inconsistent Retrievals“ werden weiterhin verhindert.

Dies ergibt also folgendes Schema für eine lock()-Operation:

Für ein Objekt:		*Angeforderter Lock*	
		lesen	*schreiben*
Lock bereits gesetzt	*keiner*	OK	OK
	lesen	OK	warten
	schreiben	warten	warten

Bild 6.37 Einführung von Lese- und Schreibsperren

Striktes 2-Phasen-Locking mit Lese- und Schreibsperren:

1. Wenn eine Operation in einer Transaktion auf ein Objekt zugreifen möchte:
   a) Wenn das Objekt nicht gesperrt ist, dann sperre es und fahre fort.
   b) Wenn das Objekt eine durch eine andere Transaktion gesetzte Sperre hat, die einen Konflikt verursacht, dann warte.
   c) Wenn das Objekt eine durch eine andere Transaktion gesetzte Sperre hat, die keinen Konflikt verursacht, dann teile die Sperre und fahre fort.
   d) Wenn das Objekt eine durch die gleiche Transaktion gesetzte Sperre hat, dann wird die Sperre ggf. erhöht und fortgefahren (gibt es andere Sperren, die eine Erhöhung verhindern, so gilt Regel b)
2. Wenn eine Transaktion beendet oder abgebrochen wird, entfernt der Server alle durch diese Transaktion gesetzten Sperren.

6.5.6 Probleme mit Sperrverfahren

Das wichtigste Problem mit Sperrverfahren sind *Deadlocks*.

Transaction *T*		**Transaction *U***	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200);	write lock
b.withdraw(100);			
•••	waits for U's lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
		•••	
•••		•••	
•••		•••	

Bild 6.38 Beispiel für Deadlock-Situation

In einem Deadlock sind zwei oder mehrere Transaktionen sind blockiert, indem sie auf Freigaben von Sperren gegenseitig warten. Zur Erkennung von Deadlocks können z.B. *Wait-for-Graphen* eingesetzt werden:

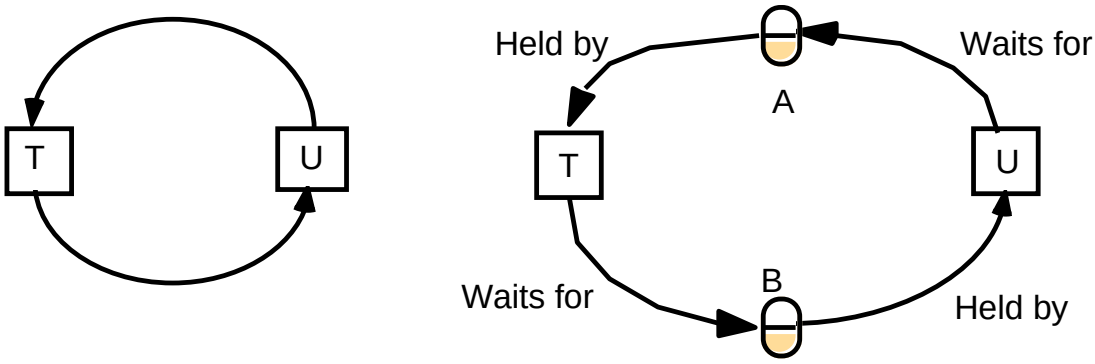


Bild 6.39 Wait-For-Graph

Eine Möglichkeit zu einem Deadlock liegt genau dann vor, wenn der Wait-for Graph einen Kreis hat.

Im folgenden Beispiel haben die Transaktionen *T*, *U* und *V* eine gemeinsame Lesesperre auf Objekt *C*. Transaktion *W* hat Schreibsperre auf Objekt *B*, *V* wartet auf Freigabe von *B*. *T* und *W* fordern nun eine Schreibsperre auf *C* an.

Dies ergibt folgendes Bild:

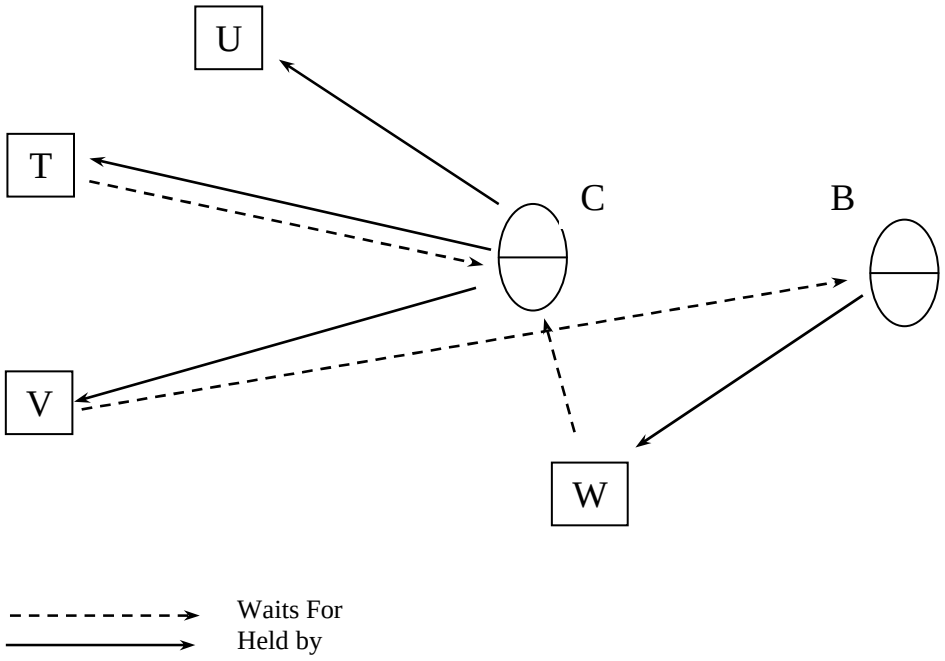


Bild 6.39 Wait-For-Graph

Resultat: Es existieren mehrere Kreise im Wait-for Graph. Jede Transaktion kann dabei (zu einem Zeitpunkt) nur auf ein Objekt warten, aber in mehreren Kreisen enthalten sein.

Der Umgang mit Deadlocks:

- Deadlock-Erkennung beim Server = finde Zyklen im Wait-for Graph (Probleme bei verteilten Transaktionen, bei denen mehrere Rechner zu sperrende Objekte halten!)
- Ansatz: suche diese Zyklen jedes Mal, wenn eine Kante hinzugefügt wird (alternativ hierzu in regelmäßigen Zeitabständen)
- Suche dann eine der Transaktionen aus dem Zyklus heraus, und breche diese ab. Faktoren bei der Auswahl: Anzahl der Zyklen, Alter der Transaktion, ggf. Priorität der Transaktion
- Alternative: Timeouts für Transaktionen (Sperren werden nach Timeout brechbar; wird eine Sperre gebrochen, so wird die entsprechende Transaktion abgebrochen)

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
	(timeout elapses)	•••	
T's lock on *A* becomes vulnerable		•••	
	unlock *A*, abort *T*	*a.withdraw(200);*	write locks *A* unlock *A*, *B*

Bild 6.40 Deadlock-Auflösung durch Timeout und ABort

6.5.7 Verteilte Transaktionen

Die bisherigen Transaktionen gingen von der Architektur mit verteilten Clients und einem zentraler Server aus. Für viele Szenarien ist es aber realistischer, dass die Objekte auf verteilten Servern liegen. Als Beispiel hierfür dient eine Überweisung von einer Bank zur anderen. Die dabei auftauchenden Probleme sind:

- Wann und wie kann Transaktion abgeschlossen werden?
- Wie wird mit Nebenläufigkeit umgegangen (serielle Äquivalenz?)
- Wie können ggf. Deadlocks erkannt werden?

Transaktionen Typ 1: Flache verteilte Transaktionen

In diesem Fall greift die Transaktion auf Objekte zu, die auf mehreren Servern liegen, und zwar direkt. Dies entspricht etwa der in Bild 6.41 gezeigten Struktur:

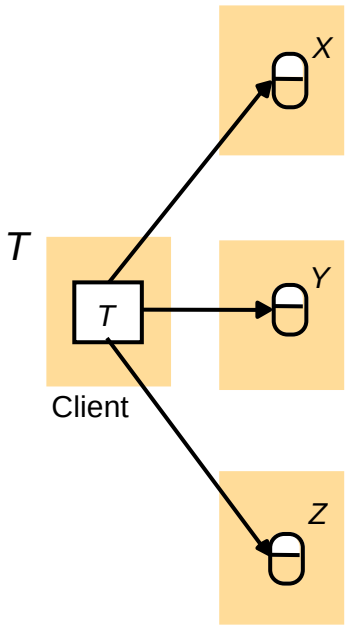


Bild 6.41 Flache verteilte Transaktion

Transaktionen Typ 2: Verschachtelte verteilte Transaktionen

Die Transaktion greift auf Objekte zu, die auf mehreren Servern liegen, *und hat ggf. weiter verschachtelte.und nebenläufige Subtransaktionen*

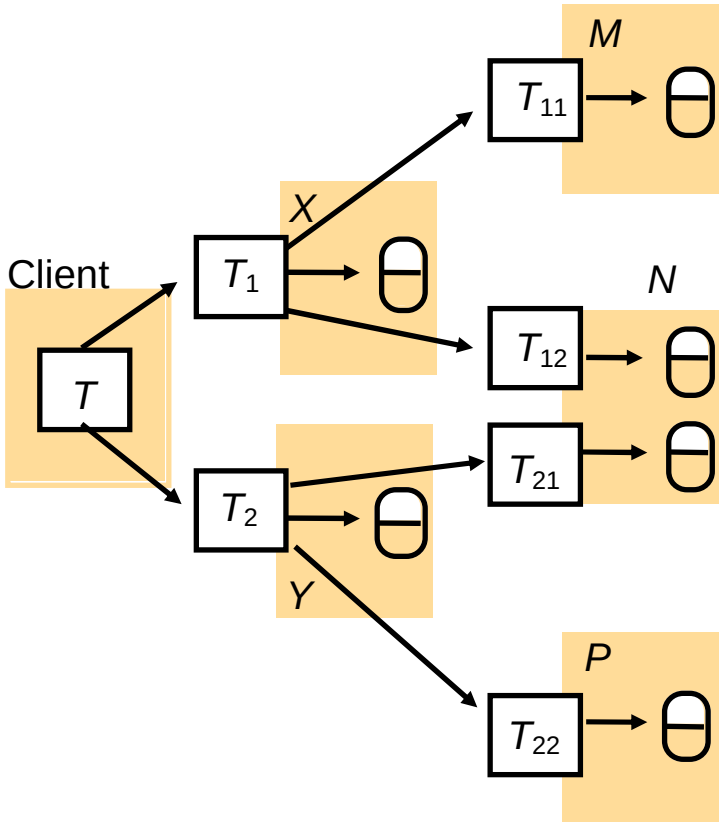


Bild 6.42 Verschachtelte verteilte Transaktion

Die Vorteile einer Verschachtelung liegen in folgenden Punkten:

- Grundprinzip: Subtransaktionen können unabhängig von Obertransaktion und anderen Subtransaktionen (auf dem gleichen Level) operieren.
- Brechen Subtransaktionen ab, können die Obertransaktionen trotzdem beendet werden (und ggf. darauf reagieren).
- Vorteile: erhöhte Nebenläufigkeit, verbesserte Robustheit
- Beispiele:
  - Senden einer Mail an Liste von Empfängern
  - countTotal() über alle Konten einer Firma

Commit/Abort-Regeln für verschachtelte Transaktionen

- Eine Transaktion darf erst dann abbrechen (abort) oder beenden (commit), wenn alle ihre Subtransaktionen ausgeführt wurden (completed).
- Wenn eine Subtransaktion ausgeführt wurde, so trifft sie eine unabhängige Entscheidung darüber, ob sie provisorisch beendet wird, oder abbricht (Abbrechen ist endgültig).
- Wenn eine Transaktion abbricht, so werden alle ihre Subtransaktionen abgebrochen (selbst wenn diese provisorisch beendet wurden).
- Wenn eine Subtransaktion abbricht, kann deren Eltern-Transaktion unabhängig darüber entscheiden, ob sie abbricht oder nicht.

- Wenn die oberste Transaktion endgültig beendet wird, so werden alle Subtransaktionen, die provisorisch beendet wurden und bei denen kein Vorfahr abgebrochen wurde, ebenfalls endgültig beendet.

Eine derartige verschachtelte Transaktion könnte also wie folgt aussehen:

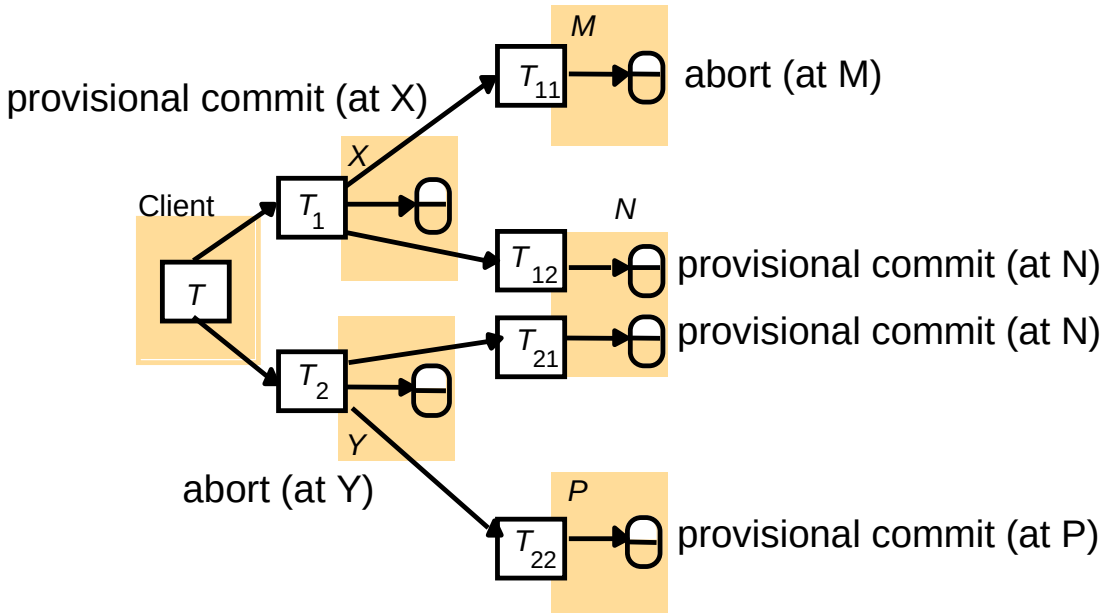


Bild 6.43 Beispiel für den Ablauf einer verschachtelten verteilten Transaktion

Als Beispiel für eine verschachtelte verteilte Bank-Transaktion sei folgende Darstellung gegeben:

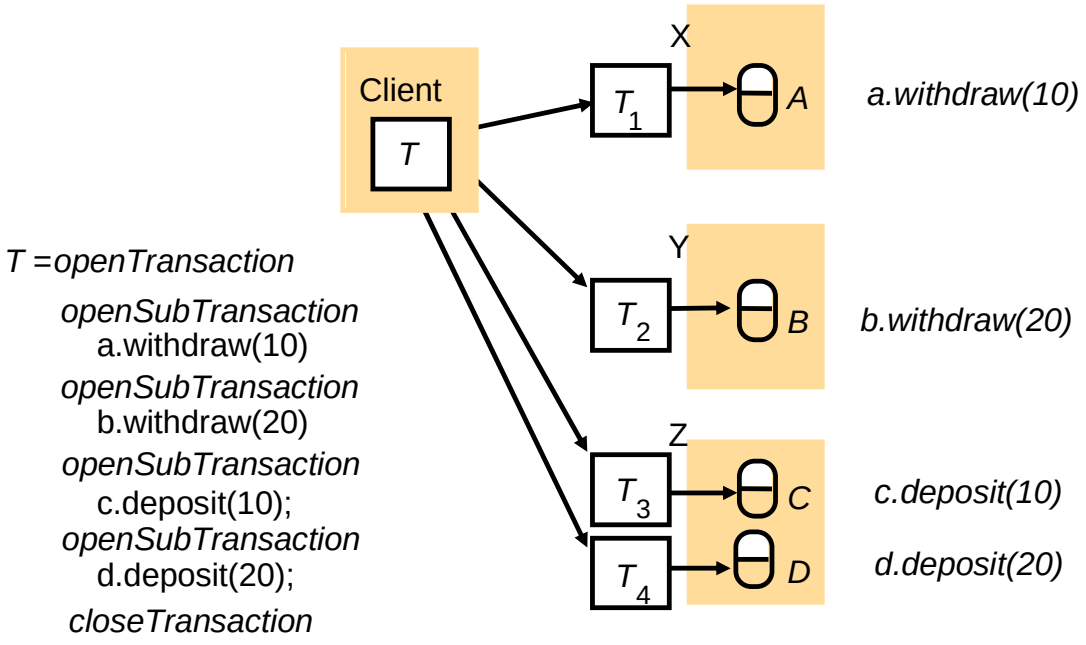
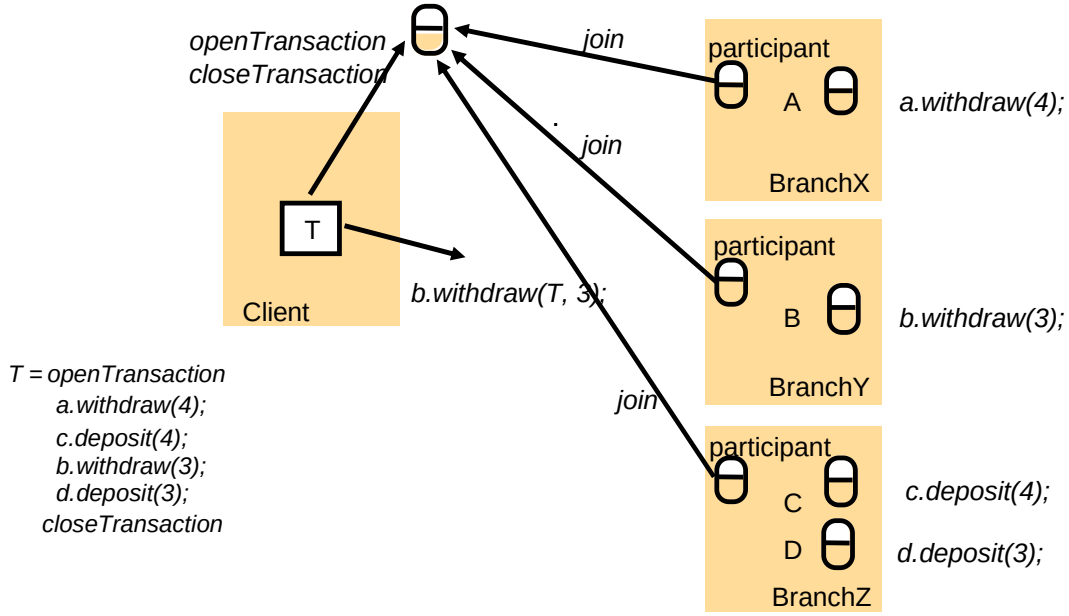


Bild 6.44 Beispiel einer Banking-Transaktion

Koordinatoren von verschachtelten verteilten Transaktionen:

- Bei einer verteilten Transaktion (flach oder verschachtelt) hat stets ein Server die Rolle des *Koordinators*
- Alle anderen Server, bei denen Objekte verwendet werden, sind Teilnehmer. Interface-Methode beim Koordinator: join(transaction, participant)

- Koordinator ist verantwortlich für Konsistenz der commit/abort Entscheidung (welche durch Client getroffen wird).
- Teilnehmer sind für ihre Objekte verantwortlich und können lokale aborts an Server mitteilen. Methode beim Koordinator: abortTransaction()



Der Koordinator ist typischerweise einer der Server, z.B. BranchX

Bild 6.44 Transaktions-Koordinatoren

6.5.8 Commit-Protokolle für verteilte Transaktionen

Sie einzelnen Schritte bzw. Transaktionen müssen bei einer verteilten Bearbeitung stets determiniert und nachvollziehbar sein. Diese Forderung wird durch das so genannte **ACID-Prinzip** ausgedrückt. Es ist ein Akronym aus den folgenden Teilforderungen nach

- der ***Atomarität*** *(atomicity)* als Forderung einer vollständigen Ausführung einer Transaktion ohne jede Seiten- oder Nebeneffekte;
- der ***Konsistenz*** *(consistency)* im Sinne der Konsistenzerhaltung der bearbeiteten Daten- oder Informationsmenge;
- der ***Isolation*** *(isolation)* hinsichtlich der Nichtverfügbarkeit bzw. Nichtsichtbarkeit der durch eine Transaktion erzeugten Zwischenergebnisse beliebiger Art;
- der ***Dauerhaftigkeit*** *(durability)* als Forderung, dass nach Beendigung einer Transaktion auch alle Änderungen erhalten bleiben.

Daraus ergibt sich:

- Entweder werden alle Operationen ausgeführt (commit) oder keine (abort).
- Eingeschränkt für verschachtelte Transaktionen: entweder alle Operationen von Transaktionen, *die provisorisch beendet wurden und bei denen kein Vorfahr abgebrochen wurde*, werden ausgeführt (commit) oder keine (abort).
- Problem bei verteilten Transaktionen: Objekte sind auf verschiedenen Servern!

Eine erste Lösung ist ein einfaches 1-Phase-Commit Protokoll:

- Koordinator sendet Commit/Abort Befehl an alle Teilnehmer.
- Requests werden wiederholt gesendet, bis alle geantwortet haben.

Dies ist meist unangemessen, denn ein Teilnehmer muss lokal den Abbruch veranlassen können.

2-Phasen-Commit-Protokoll für flache Transaktionen:

- Prinzip: Teilnehmer teilen ihre lokale Entscheidung dem Koordinator mit (Abstimmung). Der Koordinator trifft die Gesamtentscheidung und teilt sie allen Teilnehmern mit.
- Protokoll wird nicht verwendet, wenn während des Ablaufs der Transaktion ein Teilnehmer abortTransaction aufruft (dann direkter Abbruch), sondern wenn ein Client commit veranlassen will.
- Problem: Sicherstellung, dass Verfahren auch bei Ausfall eines Teilnehmers funktioniert.

Operationen im 2-Phasen-Commit Protokoll:

canCommit?(trans)-> Yes / No

- Aufruf seitens des Koordinators an die Teilnehmer der Transaktion, ob sie dieser Transaktion ein Commit geben können. Die Teilnehmer antworten jeweils.

doCommit(trans)

- Aufruf des Koordinators an die Teilnehmer, ihren Teil der Transaktion zu bestätigen.

doAbort(trans)

- Aufruf des Koordinators an die Teilnehmer, ihren Teil der Transaktion abzubrechen.

haveCommitted(trans, participant)

- Mitteilung des Teilnehmers an den Koordinator, dass sein Teil der Transaktion bestätigt wurde.

getDecision(trans) -> Yes / No

- Mitteilung des Teilnehmers an den Koordinator, dass er mit Yes votiert hat, aber weiterhin auf die Antwort (nach einer Wartezeit) wartet. Dies wird genutzt, um nach einem Server Crash in einen geordneten Zustand übergehen zu können oder verzögerte bzw. verloren gegangene Nachrichten auszugleichen.

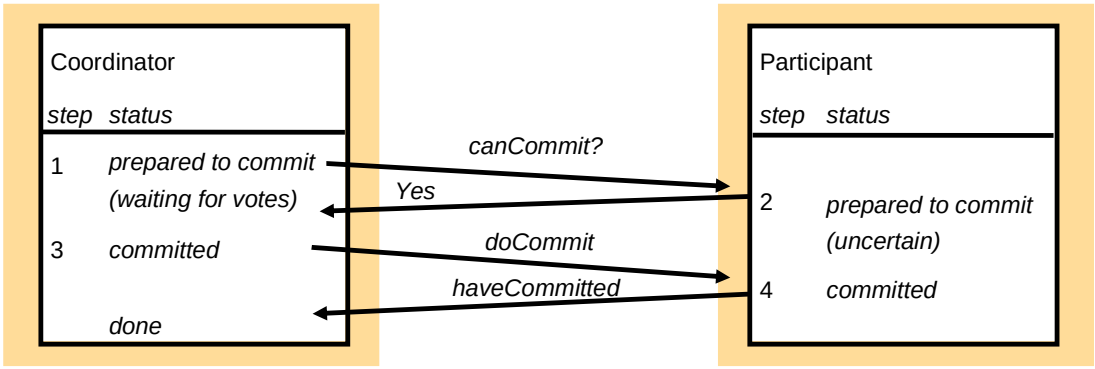


Bild 6.45 Ablauf des 2-Phasen-Protokolls für Commit

Der Ablauf diese 2-Phasen-Protokolls gestaltet sich wie folgt:

Phase 1 (voting phase):

1. Der Koordinator sendet einen *canCommit*?-Request an alle Teilnehmer dieser Transaktion.
2. Wenn ein Teilnehmer einen *canCommit*?-Request empfängt, antwortet er mit einem Wert (*Yes* oder *No*) an den Koordinator. Vor einem *Yes* werden alle Objekte gesichert, um den Commit vorzubereiten. Bei einem *No* hingegen bricht der Teilnehmer sofort ab.

Phase 2 (completion according to outcome of vote):

3. Der Koordinator sammelt die Antworten (einschließlich der eigenen).

   (a) Bei keinem aufgetretenen Fehler und wenn alle mit *Yes* geantwortet haben, entscheidet der Koordinator, dass die Transaktion bestätigt wird, und sendet einen *doCommit*-Request an alle.

(b) Im anderen Fall entscheidet der Koordinator den Abbruch und sendet einen *doAbort*-Requests an alle Teilnehmer, die mit *Yes* geantwortet haben.

4. Teilnehmer, die mit *Yes* geantwortet haben, warten auf ein *doCommit* oder *doAbort*-Request vom Koordinator. Wenn ein Teilnehmer eine dieser Nachrichten erhalten hat, wird die entsprechende Aktion sofort ausgeführt und im Fall des commit eine *haveCommitted*-Nachricht als Bestätigung an den Koordinator gesendet.

Eigenschaften des 2-Phasen-Protokolls:

Bei N Teilnehmern:

- Anzahl der Nachrichten proportional zu 4N
- Zeitaufwand: 4 Nachrichten (pro Teilnehmer)

Das Protokoll kann in synchronen verteilten Systemen mit beliebigen Ausfällen umgehen, allerdings sind lange Wartezeiten bei Koordinatorausfall in Kauf zu nehmen.

2-Phasen-Commit bei verschachtelten Transaktionen

- Der Koordinator der Top-Level Transaktion startet 2-Phasen-Commit
- Jede Transaktion hat Liste ihrer Subtransaktionen (über Join)
- Top-Level Transaktion kann daher (rekursiv) die Liste aller provisorisch beendeten Subtransaktionen erhalten.
- Subtransaktionen von abgebrochenen Transaktionen werden dabei aus der Liste entfernt (Servercrash = Abbruch)

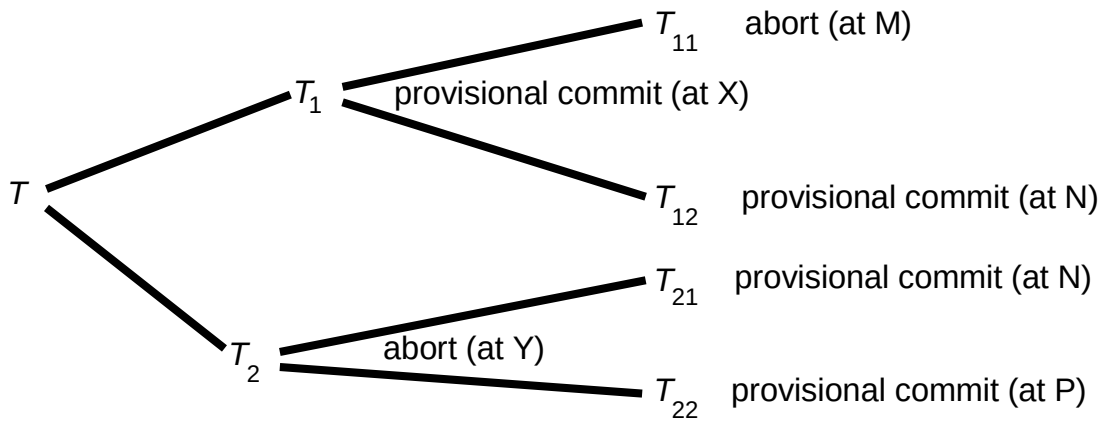


Bild 6.46 2-Phasen-Commit-Protokoll bei verschachtelten Transaktionen

Für dieses Protokoll existieren zwei Varianten:

- Phase 1 – hierarchische Variante: Der canCommit(trans, subTrans)-Request wird immer über eine Hierarchieebene durchgeführt.
- Phase 1 – nicht-hierarchische Variante: Der canCommit(trans, subTrans)-Request wird vom Koordinator direkt an alle Teilnehmer gesendet.
- Phase 2 wie bei flachen Transaktionen

6.5.9 Sperrverfahren für verteilte Transaktionen

- Sperren zur Nebenläufigkeitskontrolle auch bei verteilten Transaktionen verwendbar (genauso wie Zeitstempel und optimistische Verfahren)
- Jeder Server hat die Verantwortung für Nebenläufigkeitskontrolle auf seinen Objekten
- Gruppe von Servern ist verantwortlich für serielle Äquivalenz.

- Sperrverfahren:
  - lokale LockManager setzen Sperren
  - Freigabe von Sperren im Commit-Protokoll
  - Problem: ggf. können auf einem Server Wartebeziehungen nicht erkannt werden.

Beispiel für verteilte Transaktionen mit Sperrverfahren:

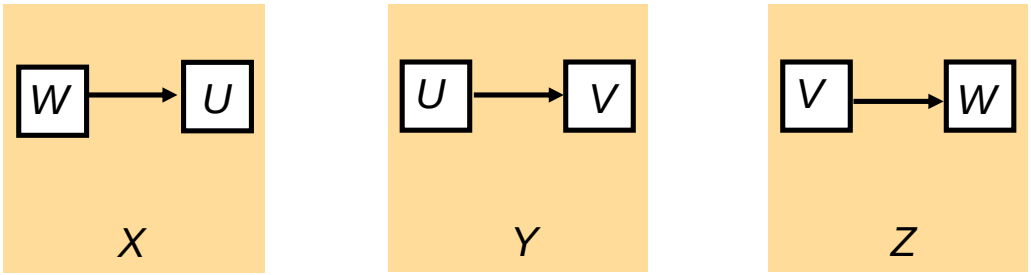
Situation: Server X hat Konto A, Server Y hat Konto B,
Server Z hat Konten C und D.

U	*V*	*W*
d.deposit(10) lock *D* at *Z*		
	b.deposit(10) lock *B* at *Y*	
a.deposit(20) lock *A* at *X*		
		c.deposit(30) lock *C* at *Z*
b.withdraw(30) wait at *Y*		
	c.withdraw(20) wait at *Z*	
		a.withdraw(20) wait at *X*

Deadlocks sind Kreise im globalen (!) Wait-for Graph. Dies muss sich nicht durch Kreise in lokalen Wait-for Graphen zeigen.

Bild 6.47 Beispielanwendung mit lock/unlock bei verteilten Anwendungen

Hieraus folgen folgende globalen und lokalen Wait-For-Graphen:



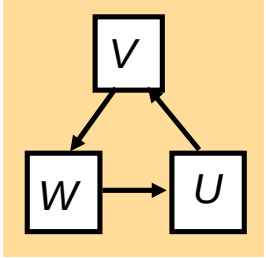


Bild 6.48 Lokale und globale Wait-For-Graphen

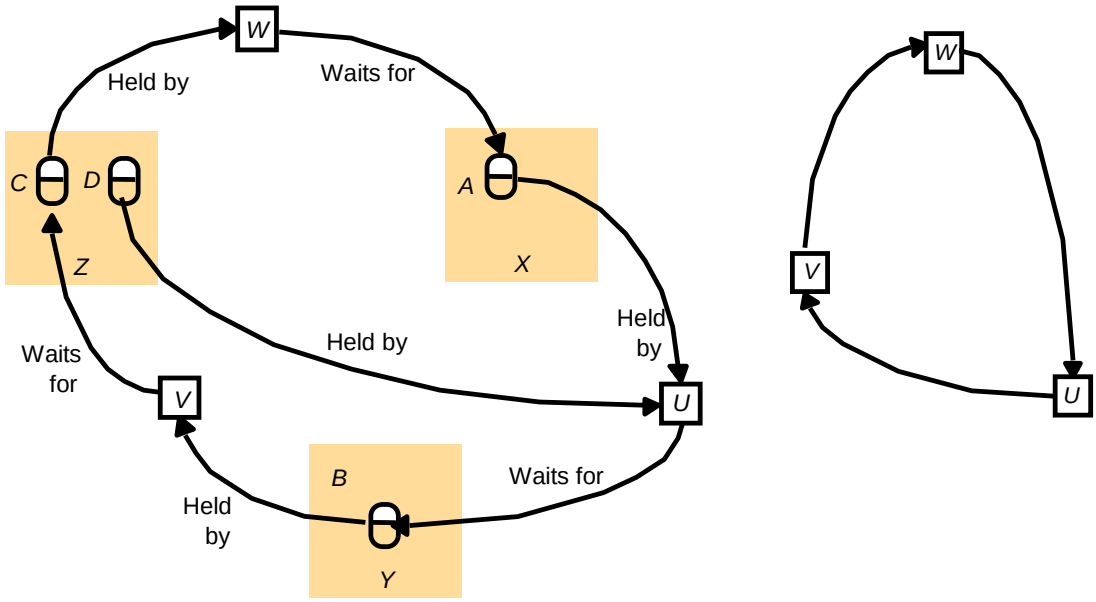


Bild 6.49 Konstruktion des globalen Wait-For-Graphen

Erkennung von verteilten Deadlocks:

1. Zentrale Konstruktion des globalen Wait-for Graphs (in gewissen Zeitintervallen)
   - Phantom-Deadlocks möglich, z.B. durch Kreis mit bereits abgebrochener Transaktion, oder durch Übertragungsreihenfolge
   - Kostenintensiv, fehleranfällig, schlechte Skalierung
2. Edge-Chasing: Verfolge verteilte Wartepfade. Jeder Server kann Nachfragen (Probes) starten, wenn er eine lokale „Wait-for“ Situation erkennt, bei der die Transaktion, welche die Sperre hält, selbst wartet.

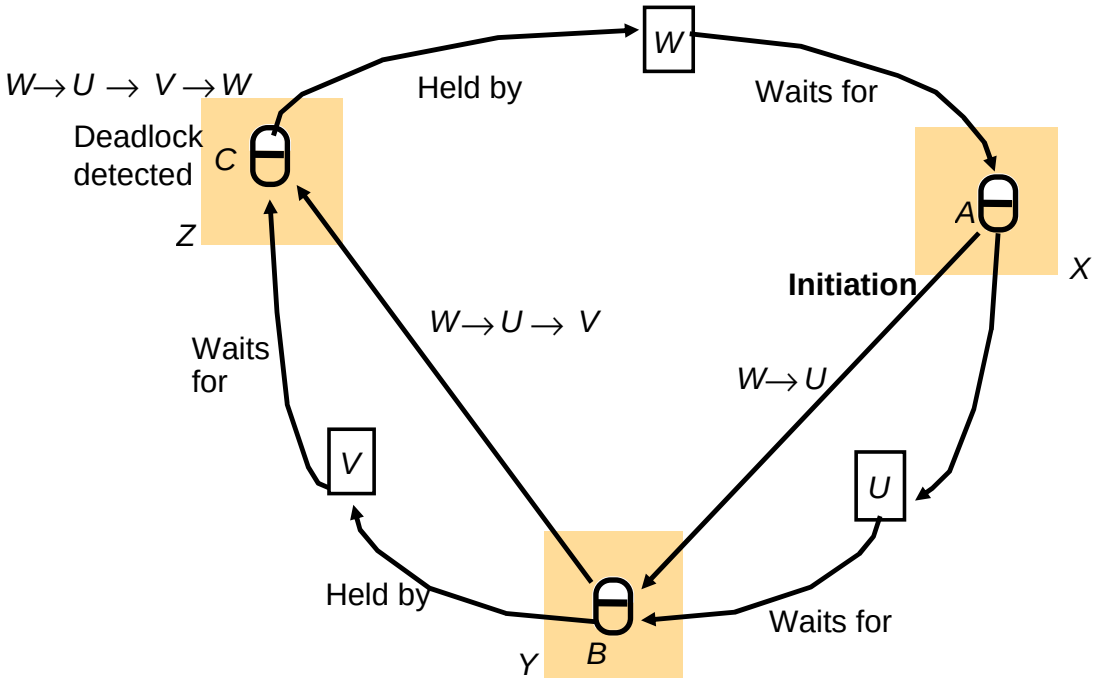


Bild 6.50 Edge-Chasing Verfahren zur Erkennung von Deadlocks

- Phase 1: Initialisierung. Wenn ein Server bemerkt, dass eine Transaktion T auf eine andere Transaktion U wartet, wobei U selbst auf ein Objekt auf einem anderen Server wartet, so initiiert er den Algorithmus

durch eine Nachricht T→U an den Server, durch den U blockiert wird. Wenn U durch eine gemeinsam genutzte Sperre blockiert wird, so gehen diese Nachrichten an alle betreffenden Server.

- Phase 2: Erkennung. Wenn ein Server eine Nachricht A1→…→An empfängt, so überprüft er ob An ebenfalls auf eine Transaktion B wartet. Ist dies der Fall, so wird die Nachricht A1→…→An→B an den Server weitergeschickt, auf den B wartet.
- Phase 3: Beseitigung. Wenn in Phase 2 ein Kreis festgestellt wird, so wird eine der Transaktionen des Kreises abgebrochen.

7 Architekturen verteilter Systeme

7.1 Allgemeines zur Systemarchitektur

Wenn man über Architekturen verteilte Systeme diskutieren will, muss man einige Dimensionen dabei betrachten. Folgende Beispiele seien hierzu benannt:

1. Zentral vs. Dezentral
   (Welche Rolle haben Prozesse? Welche Kommunikations/Koordinationswege sind vorhanden?)
2. Loose Coupling – Tight Coupling
   (Wie stark hängen Prozesse zusammen? Ist ein Austausch leicht möglich oder nicht? Hohe oder Niedrige Anforderungen an Schnittstellen?)
3. Grundlegende Metapher des Systems
   (Auf welchem Grundprinzip ist das System aufgebaut? Durch welchen Begriff lässt sich die Architektur am besten beschreiben?)
4. Verwendete Entwurfsmuster
   (Welche Prinzipen des SW-Engineerings Verteilter Systeme kommen zum Einsatz?)

Für zwei dieser Dimensionen sei eine Matrix angegeben:

Metapher →	Nachricht	Dienst	Komponente
Dezentral ↕ Zentral			

Bild 7.1 Matrix zur Einordnung in Architektur-Dimensionen

7.2 Zur Metapher des Systems

7.2.1 Nachrichten-basierte Systeme

- Prinzip: Prozesse kommunizieren durch den Austausch von Nachrichten (Unicast, Multicast, Broadcast)
- Typischerweise „Loose Coupling“
- Basistechnologien: UDP, TCP (,SSL, HTTP)
- Fortgeschrittenere Beispiel-Systeme:
  - JSDT (Java Shared Data Toolkit): Middleware für kooperative Anwendungen, bietet z.B. Gruppenkommunikation mit Kanälen, gemeinsame Datenobjekte mit Notification

- JMS (Java Messaging Service): Teil von Java EE 5, erleichtert asynchrone Kommunikation zwischen Java-Applikationen

7.2.2 Dienstbasierte Systeme

- „Service Oriented Architecture“ (SOA)
- Grid Computing
- Prinzip: Prozesse kommunizieren durch Aufruf von Diensten. Diese rufen ggf. wieder andere Dienste auf.
- Typischerweise „Loose Coupling“
- Basistechnologien: RPC, RMI (‚CORBA)
- Wichtigster heutiger Standard:
  - Web Services. XML-über-HTTP-basierter Dienstaufruf
  - „RMI unabhängig von Programmiersprache, Betriebssystem und Netzwerkinfrastruktur“

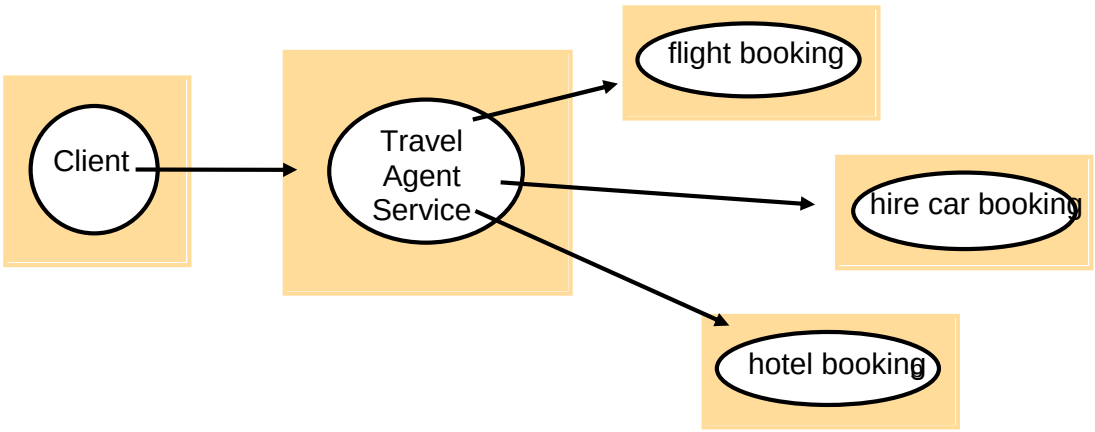


Bild 7.2 Beispiel eines dienstbasierten Systems

Dienste können synchron (Request/Reply) oder asynchron aufgerufen werden. Während im Allgmeinen bei Web Services ein synchroner Aufruf gewählt wird, unterstützt die Architektur auch asynchrone Aufrufe.

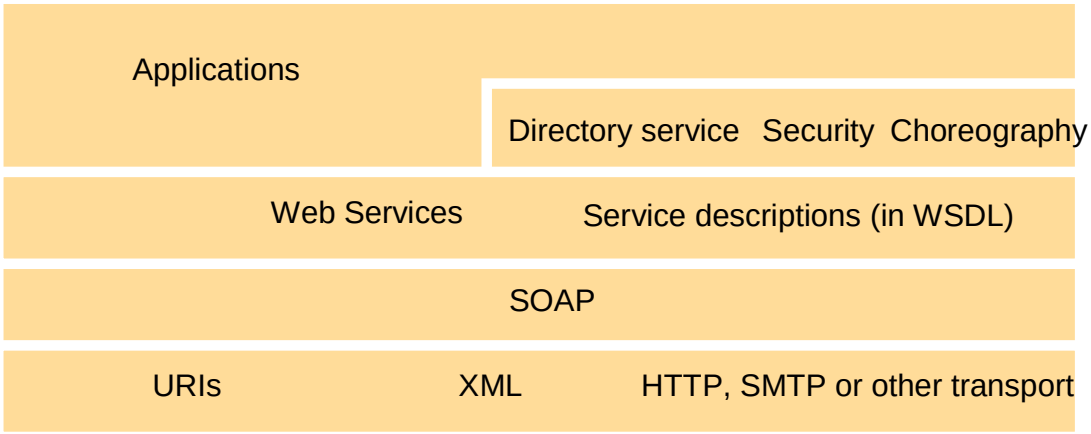


Bild 7.3 Web Service Infrastruktur

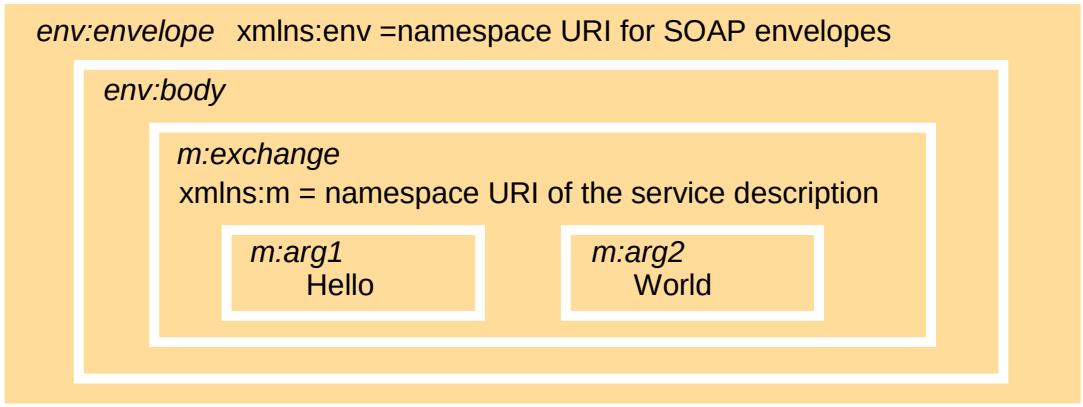


Bild 7.4 Beispiel eines SOAP-Nachricht (ohne Header)

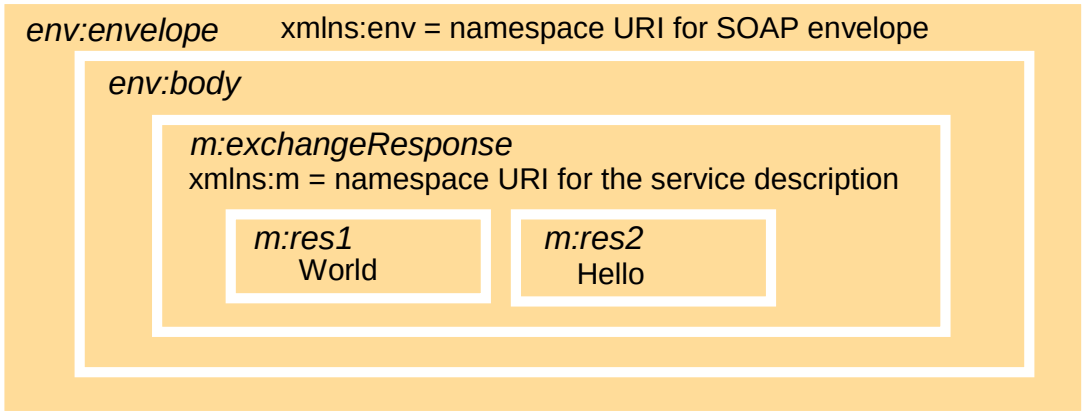


Bild 7.5 Beispiel einer SOAP-Antwort (ohne Header)

Die Beschreibung von Web Services kann in Web Service Description Language (WSDL), aufbauend auf XML (http://de.wikipedia.org/wiki/Web_Services_Description_Language) erfolgen. Hierin kann eine Spezifikation vollständig beschrieben werden:

- Schnittstellen (Operationen, Datentypen, …)
- Art der Kommunikation (z.B. SOAP über HTTP)
- URI (Uniform Resource Identifier) des Dienstes

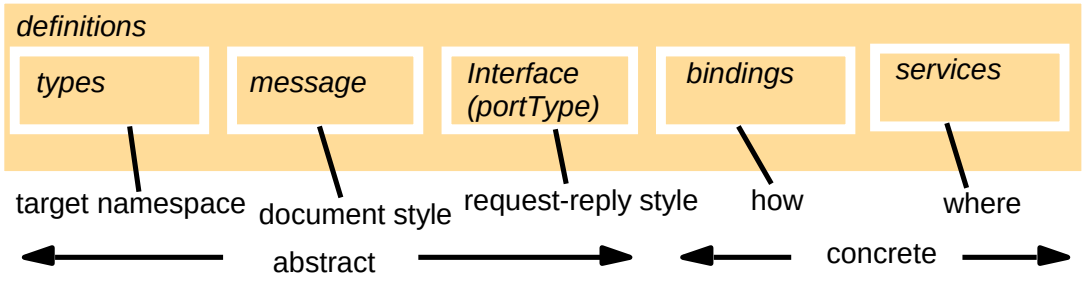


Bild 7.6 Komponenten einer Beschreibung in WSDL

7.2.3 Komponenten-basierte Systeme

Das Prinzip: Ein verteiltes System besteht aus Komponenten. Eine Komponente ist ein wiederverwendbares, abgeschlossenes Softwaremodul, das über eine wohldefinierte Schnittstelle verfügt. Die Entwicklung individueller Anwendungssysteme erfolgt durch Zusammensetzen und Verbinden

mehrerer Komponenten (Plug-und-Play Idee). Hierbei ist ein „tight coupling“ oder „loose coupling“ möglich.

Die Basistechnologien sind RMI und CORBA. Wichtige Standards:

- CORBA/CCM: Common Object Request Broker Architecture / CORBA Component Model
- COM/DCOM/COM+: Microsoft (Distributed) Component Object Model
- .NET: Auf offenen Standards basierende Plattform von Microsoft
- EJB: Enterprise Java Beans

7.2.4 Entwurfsmuster (Design Pattern)

- Entwurfsmuster = Lösungen für wiederholt auftretende Problemstellungen beim Softwareentwurf
- Getestete Standards
- Erleichtern Kommunikation unter Programmierern
- Beispiele für wiederholt auftretende Problemstellungstypen:
  - Grundansatz der Architektur (Welche Komponenten auf Client, welche auf Server? Wie erfolgt der Zugriff?)
  - Flexibilisierung und Vereinfachung des Zugangs zu Ressourcen
  - Nebenläufigkeitskontrolle

Literatur:

- Buschmann et al. (1998):
  Pattern-Orientierte Software-Architektur
  Addison Wesley
- Schmidt at al. (2002):
  Pattern-orientierte Software-Architektur. Muster für nebenläufige und vernetzte Objekte
  dpunkt-Verlag

Architekturmuster Layer:

- Problemstellung: Komplexe (verteile) Anwendungen mit einer Vielzahl von Funktionalitäten und einer Mischung von Aufgaben
- Anforderungen:
  - Änderungen am Code sollten nur lokale Auswirkungen im System haben. Die Stellen der möglichen Auswirkungen müssen leicht ermittelbar sein.
  - Teile des Codes müssen austauschbar sein.
  - Es kann sein, dass Teile des Systems später von anderer (externer) Seite weiterverwendet werden sollen.
- Lösung: Organisation des Systems in Schichten. Jede Schicht repräsentiert eine Abstraktionsschicht.
- Schichten haben Aufgaben, Namen und Schnittstellen

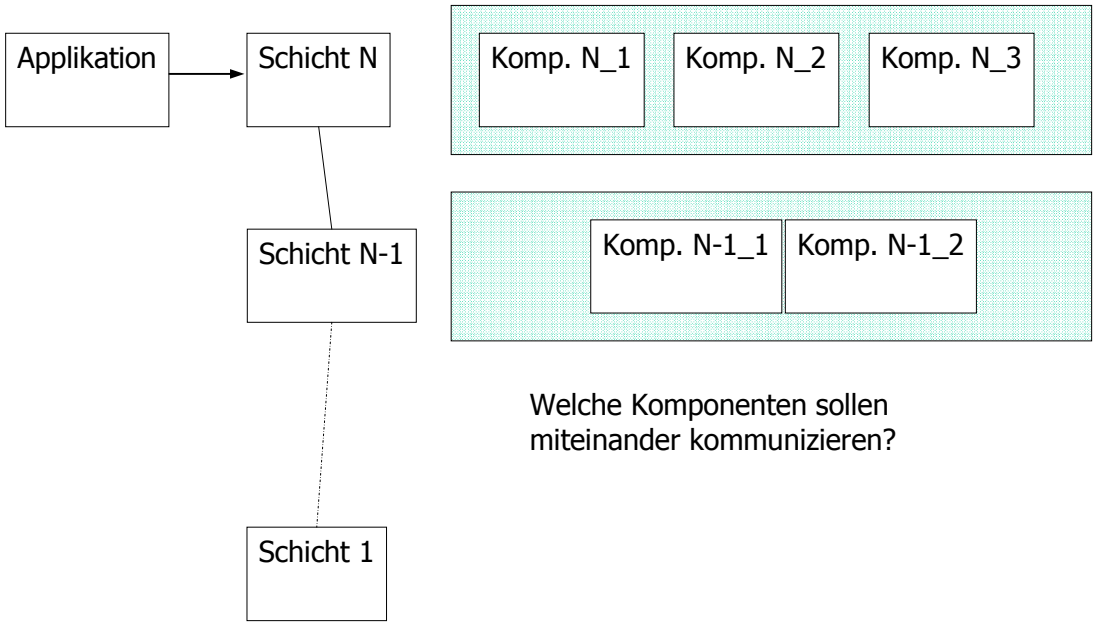


Bild 7.7 Kommunikation im Schichtenmodell

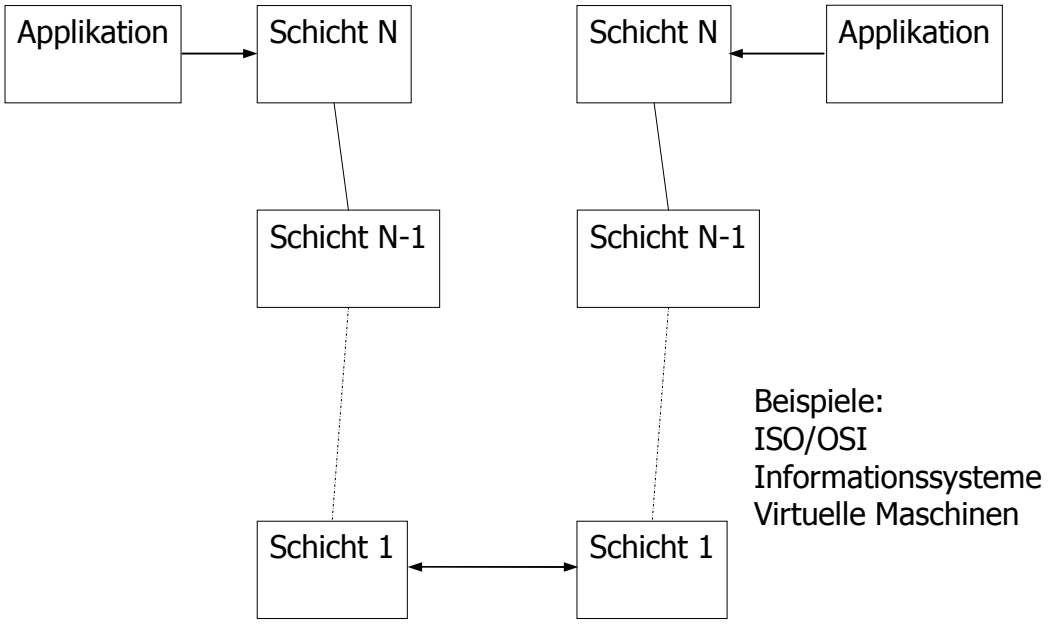


Bild 7.8 Reale Kommunikation im Schichtenmodell

Architekturmuster Model View Controller

- Problemstellung: Interaktive Anwendungen mit Datenhaltung und verschiedenen Benutzerschnittstellen.
- Anforderungen:
  - Daten können sich ändern und müssen in verschiedenen Sichten verfügbar sein.
  - Die Darstellung muss Datenänderungen sofort wiederspiegeln
  - Daten müssen ggf. leicht verteilbar sein
- Lösung: Unterteilung des Systems in Model (Verarbeitung), View (Ausgabe) und Controller (Eingabe)
- Views haben Controller (sofern sie ein UI haben)
- Modell kapselt Daten; ein Modell kann viele Views haben

- Modelle können zur Verteilung genutzt werden (entfernte Views/Controller)

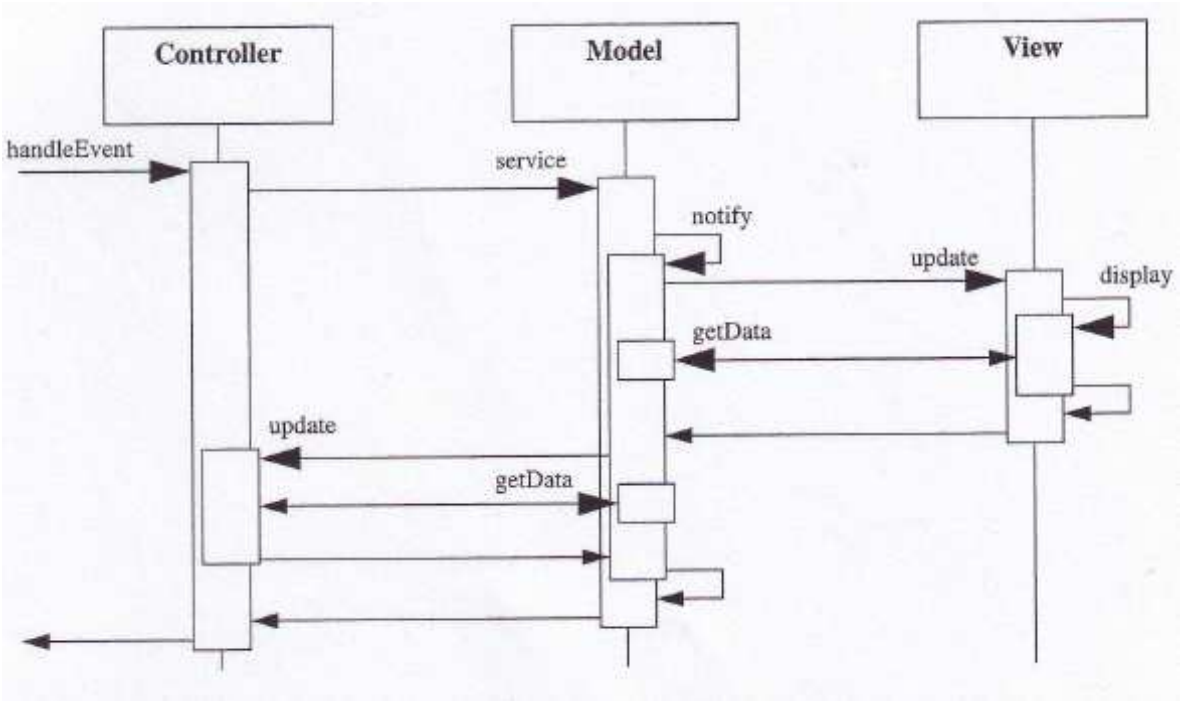


Bild 7.9 Model – View – Controller – Architektur

Architekturmuster Broker

- Problemstellung: Strukturierung von Softwaresystemen mit entkoppelten Komponenten.
- Anforderungen:
  - Interagierende Komponenten im „Loose Coupling“ Ansatz
  - Verteilung von Komponenten auf verschiedene Rechner
  - Dienste zum Hinzufügen und Entfernen von Komponenten
  - Transparenz der Verteilung
- Lösung: Einführung eines Vermittlers (Brokers) zur Entkopplung von Server und Client.
- Server melden sich beim Vermittler an und stellen ihre Dienste zur Verfügung.
- Clients greifen über den Broker auf die Dienste zu.
- Der Broker muss geeignete Server ausfindig machen und ggf. aktivieren sowie Fehlermeldungen an Clients senden / weiterleiten

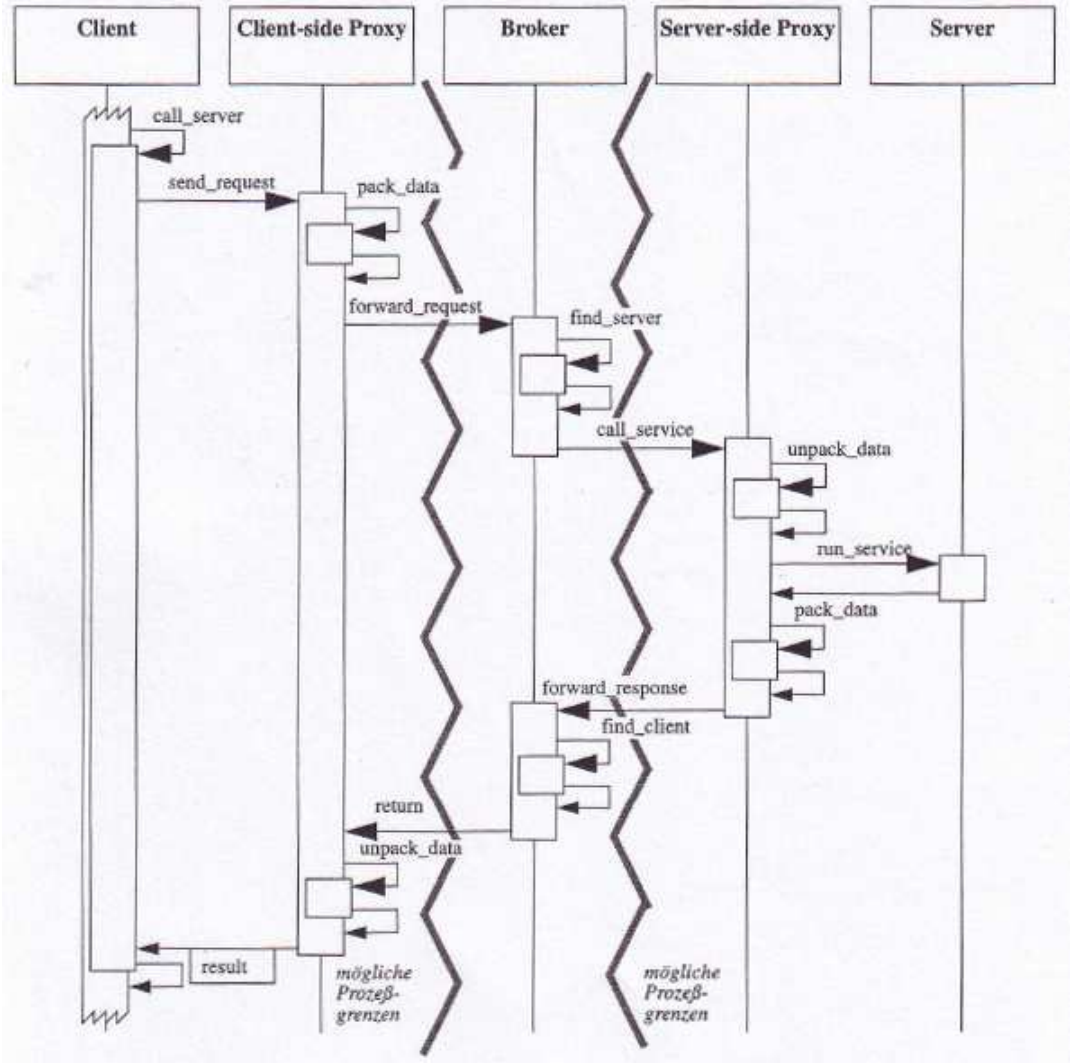


Bild 7.10 Broker-Architektur

8 Kooperative Systeme

Für die kooperativen Systeme sei ein wenig Geschichte aufgezeigt. So beginnt es in den 40er Jahre mit dem Essay des amerikanischen Ingenieurs Vannevar Bush mit dem Titel „As we may think“ (1945).

Hierin tritt ein „MEMEX“-Gerät (Memory Extender) auf: *„A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory.”*

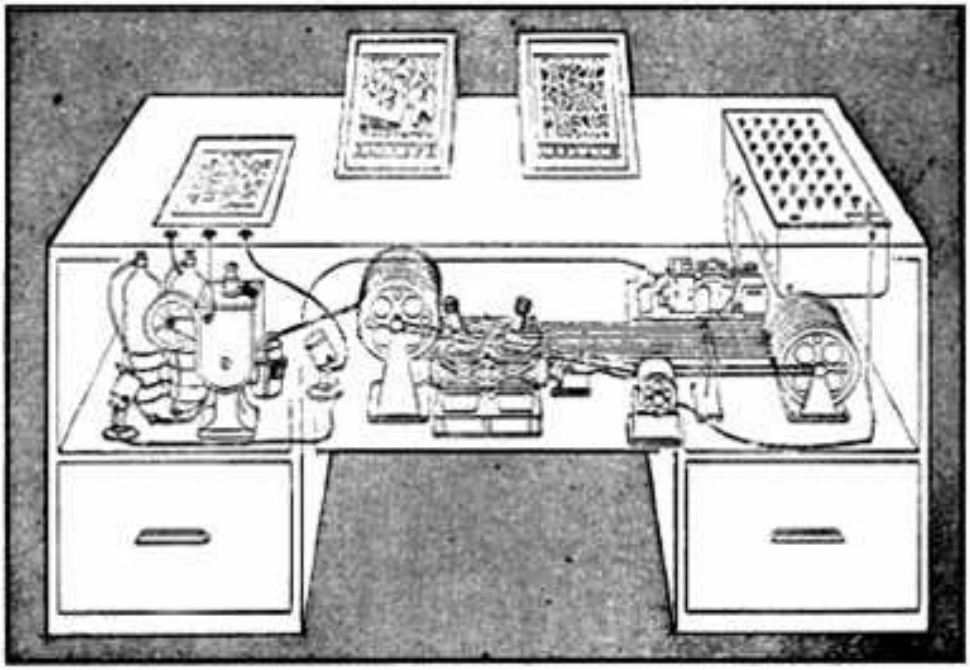


Bild 8.1 Ein MEMEX-Gerät

„And his trails do not fade. [….] A touch brings up the code book. Tapping a few keys projects the head of the trail. […] It is an interesting trail, pertinent to the discussion. So he sets a reproducer in action, photographs the whole trail out, and passes it to his friend for insertion in his own memex […].

Wholly new forms of encyclopedias will appear, ready-made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified.

The lawyer has at his touch the associated opinions and decisions of his whole experience, and of the experience of friends and authorities. The patent attorney has on call the millions of issued patents, with familiar trails to every point of his client's interest.

The physician, puzzled by its patient's reactions, strikes the trail established in studying an earlier similar case, and runs rapidly through analogous case histories, with side references to the classics for the pertinent anatomy and histology.

The chemist, struggling with the synthesis of an organic compound, has all the chemical literature before him in his laboratory, with trails following the analogies of compounds, and side trails to their physical and chemical behavior."

Der nächste Schritt erfolgt in den 60er Jahren: *Advanced Research Projects Agency* (ARPA) wird in den USA als Reaktion auf Sputnik-Schock gegründet.

- ARPANET (Vorgänger des Internets) entsteht
- Zielsetzung: Menschen zur Kooperation befähigen
  „Take any problem worthy of the name, and you find only a few people who can contribute effectively to its solution. Those people must be brought into close intellectual partnership so that their ideas can

come into contact with one another. But bring these people together physically in one place to form a team, and you have trouble, for the most creative people are often not the best team players, and there are not enough top positions in a single organization to keep them all happy. Let them go their separate ways, and each creates his own empire, large or small, and devotes more time to the role of emperor than to the role of problem solver. The principals still get together at meetings. They still visit one another. But the time scale of their communication stretches out, and the correlations among mental models degenerate between meetings so that it may take a year to do a week's communicating. There has to be some way of facilitating communication among people without bringing them together in one place. "
(Licklider, Chef von ARPA, 1968)

60er Jahre: „Office Augmentation" (Büro Verbesserung)

- u.a. ARPA-Projekt Forschung am SRI (System Research Incorporated, California)
- Hypermedia-System NLS (oNLine System): Erstes Hypertext-System, Erfindung der Maus („Pointing Device"), erste bildschirmbasierte Videokonferenz

Bild 8.2 Verbesserungen im elektronischen Büro (1960er Jahre)

- Erkenntnis: "integrate psychology and organizational development with all of these advances in computing technology."

Bild 8.3 Beispiel für Spezialgerät der 1970er Jahre

70er Jahre: „Office Automation"

- Textverarbeitung
- Unterstützung von Bürokommunikation im Electronic Information Exchange System (EIES)
  - Überwindung von Einschränkungen durch Zeit und Raum
  - Möglichkeit von Anonymität
  - Größere Gruppen können besser kommunizieren als in Telefonkonferenzen
  - Strukturierung der Kommunikation
  - Effektive Unterstützung der Gruppe bei Entscheidungsprozessen

1980er Jahre: Neue Entwicklungs- und Forschungsrichtung „Groupware“ auf Basis der Ergebnisse u.a. von EIES

Unterschiedliche Definitionsansätze:

- „Intentional group processes plus software to support them“ (Johnson-Lenz)
- „A co-evolving human-tool system.“ (Engelbart)
- „Computer-mediated collaboration that increases the productivity or functionality of person-to-person processes.“ (Coleman)
- „Computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.“ (Ellis)

1980er Jahre: Entstehung des Forschungsfeldes „CSCW“: Computer Supported Cooperative Work

- Interdisziplinäres Forschungsgebiet zwischen Informatik, Ökonomie, Psychologie und Soziologie.
- Beteiligt: Forscher aus dem Bereich HCI (Human Computer Interaction) und IS (Information Systems)
- Ziel: Entwicklung neuer Theorien und Technologien für die Koordination von Gruppen, die zusammen arbeiten
- Groupware=Technologien, CSCW = Forschungsgebiet zu diesen Technologien

1990er Jahre: Begriff „Groupware“ wird für Lotus Notes zu Marketingzwecken verwendet…

- Email client,
- Instant Messaging Client
- Browser,
- Notizfunktion,
- Kalender,
- Funktion zur Belegung von gemeinsamen Ressourcen,
- Diskussionslisten,
- Gemeinsame Kontaktdatenbank.

… und dann weiter von Microsoft für Outlook / Exchange Server

- Begriff „Social Software“ kommt auf (freiere Interaktionsformen)
- Rasante Entwicklung von Internet und WWW

Etwa um die Jahrtausendwende werden immer mehr Webseiten „interaktiv“:

- Benutzer können Inhalte auf Seiten beeinflussen und personalisieren
- Information → Interaktion, Kommunikation

2002: Social Software Summit, organisiert von Clay Shirky
„… something that gathered together all uses of software that supported interacting groups, even if the interaction was offline.”

Unterschied zu CSCW:
„that seems […] leaving out other kinds of group processes such as discussion, mutual advice or favors, and play"

Einige „Klassiker":

- Email
- Wikis
- Foren
- Chats
- Instant Messenger
- Web-Telefonie
- Videokonferenzen
- Blogs
  - Nachfolger von persönlichen Websites
  - Vernetzte „Online-Tagebücher" mit Kommentarmöglichkeiten
- Google Docs
  - Web-basiertes Office-Paket
  - Möglichkeit des gemeinsamen Editierens von Dokumenten (synchron und asynchron)
- Social Networking Services
  - Benutzer stellen Profil online und suchen für sie potenziell interessante andere User
  - System kann diese vorschlagen und bietet diverse Kommunikationskanäle an
  - Anwendungen: Dating, Unternehmen (z.B. Startups), Hobbybereich
- „Online-Fotoalbum"
- Bilder sind mit Schlagworten (Tags) versehen, welche zur Suche genutzt werden können

Wikipedia

- Online-Enzyklopädie – von jedem änderbar
  - In die Schlagzeilen gekommen wegen absichtlicher Fehlinformation in Artikeln
  - Wird als verlässliche Literaturquelle daher vermieden
  - Umfang beeindruckend:
    - hunderte Sprachen; >1.800.000 Einträge in Englisch
    - MS Encarta 2007: 64.000 Einträge
    - Encyclopaedia Britannica: 120.000 Einträge

Definitionen:

„Soziale Software": Software-Systeme, die

- menschliche Kommunikation, Interaktion und Zusammenarbeit ermöglichen bzw. unterstützen
- dazu dienen, Gemeinschaften aufzubauen und zu pflegen (typischerweise über das Internet: „Web Communities")
- sich teilweise selbstorganisiert entwickeln
- oftmals Technologien verteilter Systeme verwenden

Spezialfall Groupware / CSCW / Collaborative Software:

- Unterstützung von Arbeitsgruppen und -prozessen
- wichtig: gemeinsames Ziel bzw. gemeinsame Aufgabe
- Mögliche Arten der Klassifikation:
  - Wo und Wann arbeiten die Personen zusammen?
  - Welche Funktion hat das Groupware-Tool?
- Erste Art der Klassifikation:
  - Arbeiten Personen gleichzeitig oder nicht? (synchron / asynchron)
  - Sind die Personen am gleichen Ort oder nicht? (co-located / remote)

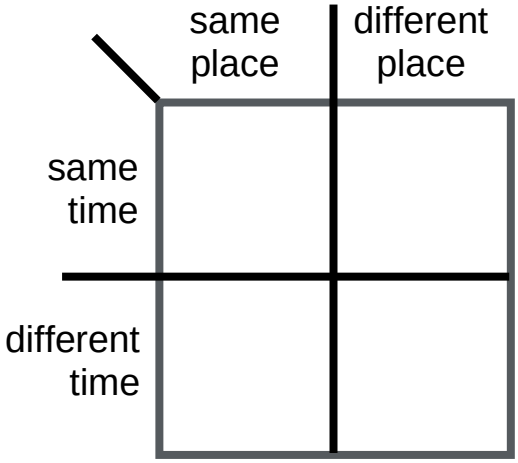


Bild 8.4 Klassifizierung Groupware-Software

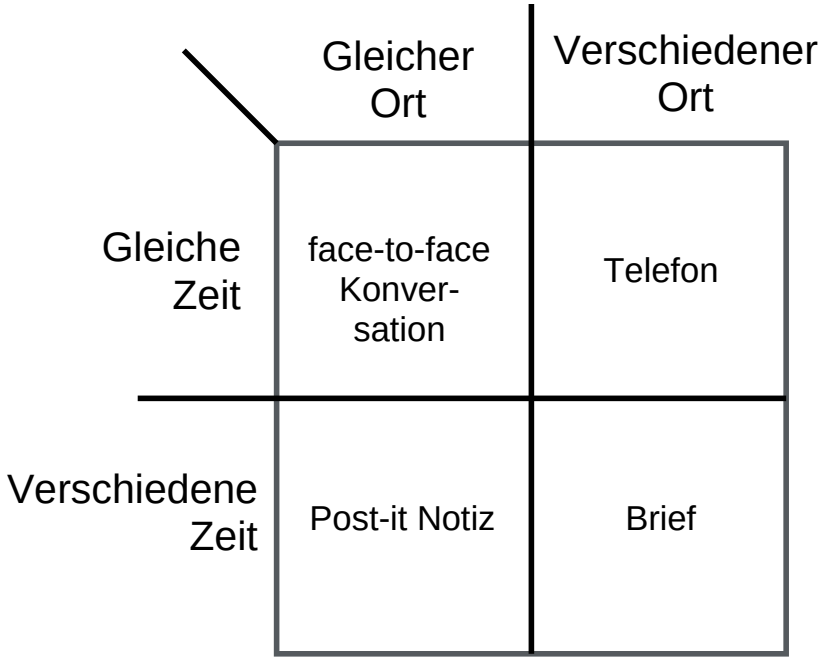


Bild 8.5 Matrix für menschliche Kommunikation

Die gemeinsame Arbeit beinhaltet sowohl die **Personen** (die arbeiten) als auch die **Artefakte** (mit denen gearbeitet wird).

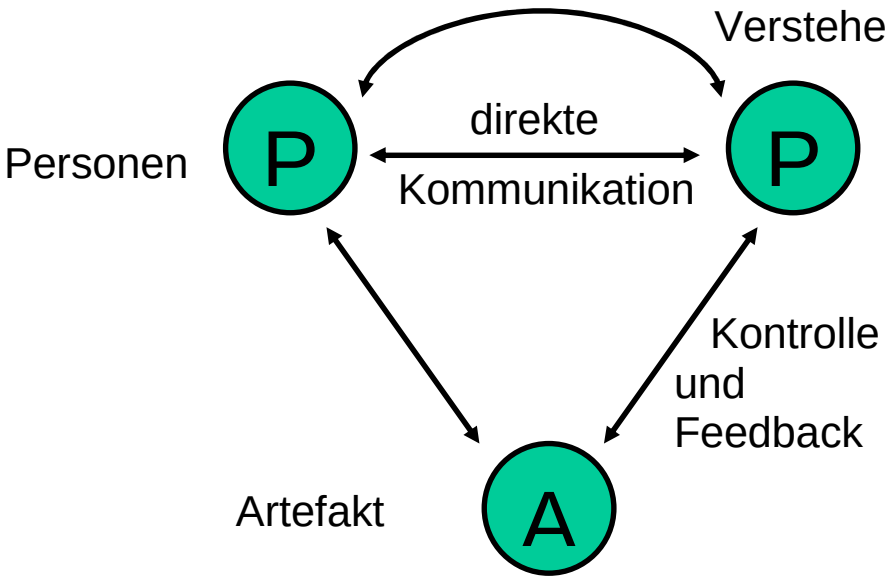


Bild 8.6 Kommunikationen zwischen Personen und Artefakten

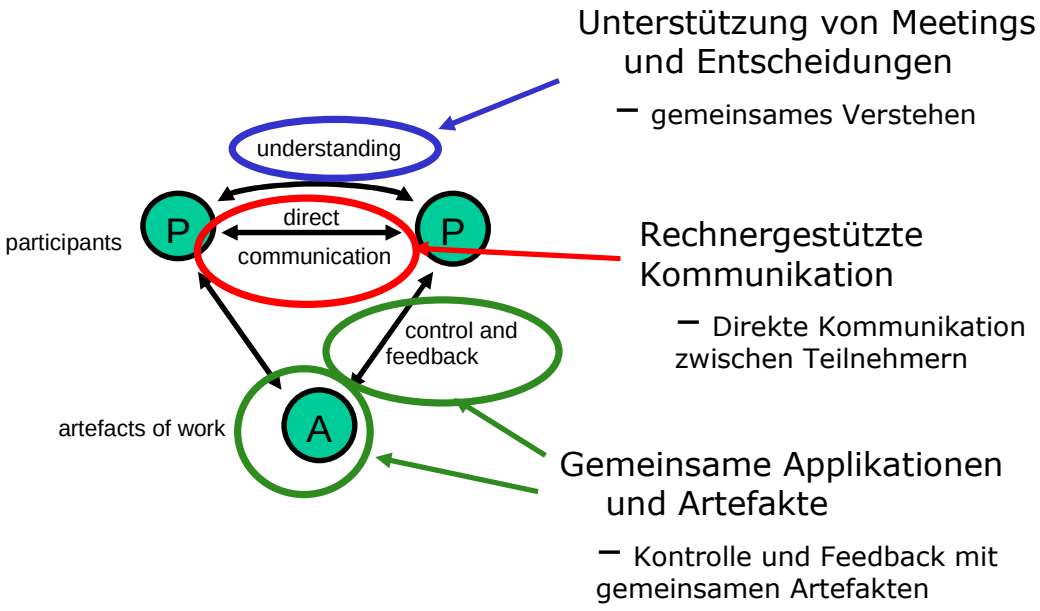


Bild 8.7 Unterstützung der Kommunikationen

Rechnergestützte Kommunikation:

- Email, Nachrichtenboards
- Strukturierte Nachrichtensysteme
- Textnachrichten
- Video und virtuelle Umgebungen

Email und Nachrichtenboards gelten allgemein als bekannt und als die erfolgreichste Groupware. Für die Email gilt:

- Asynchron + remote

- Email-Empfänger:
  direkt To:
  kopien Cc:

  Diese Formen ergeben identische Zulieferung, der Unterschied liegt in sozialer Absicht.

- Versendemodi
  - one-to-one – Email: direkte Kommunikation
  - one-to-many – Email: Verteilerlisten; Blackboards: Broadcast
- Kontrolle
  - Sender – Email, Private Verteiler
  - Administrator – Email, Verteilerlisten
  - Empfänger – Blackboards, Anmelden für Themen

Weiterentwicklung in Richtung strukturierter Nachrichtensysteme

- `Super'-Email
  – Kreuzung aus email und Datenbank
  – Sender füllt spezielle Felder aus
  – Empfänger kann eingehende Mails nach Feldinhalten filtern und sortieren

Übersicht zu rechnergestützten Kommunikationssystemen

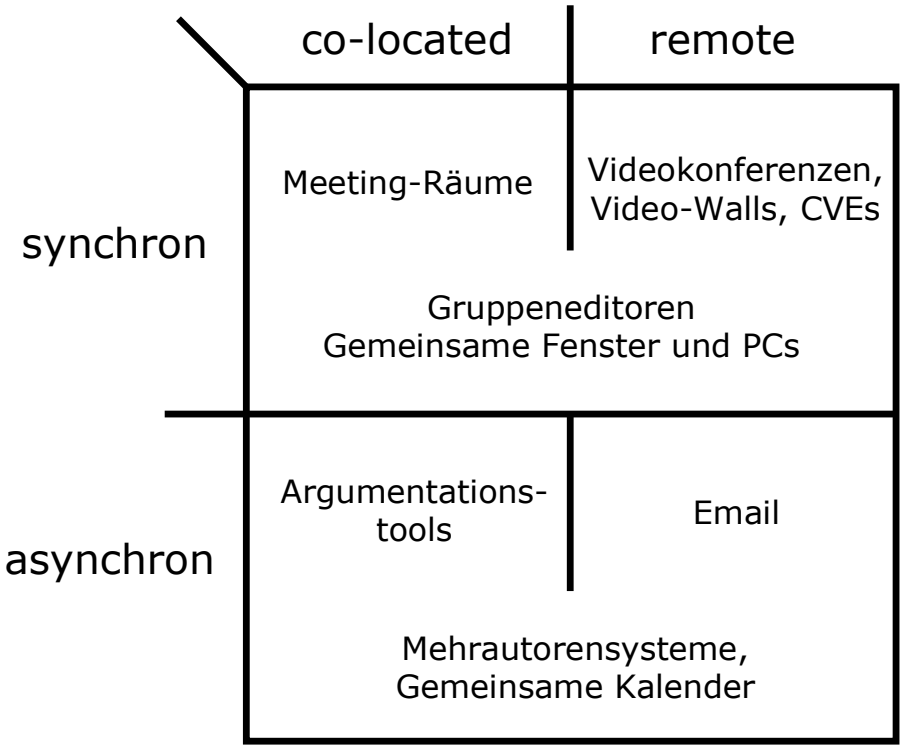


Bild 8.8 Raum/Zeit-Matrix zu rechnergestützten Kommunikationssystemen

9 Zusammenfassung

Welche Themen wurden behandelt?

- Grundlagen
  - Definition Verteiltes System
  - Ziele und Herausforderungen
- Kommunikation in Netzwerken
  - Typen und Charakteristika von Netzwerken
  - ISO/OSI Modell
  - Routing: RIP-Algorithmus, Routing im Internet
  - TCP und UDP, Ports und Sockets
  - Typen der Kommunikation (Unicast, Multicast, Broadcast)
- Interprozesskommunikation
  - Modelle der Interprozesskommunikation
  - RPC: Aufrufsemantik und Parameterübergabe
  - Objekte bei RMI (Remote Objects, Serialisierung)
  - Interprozesskommunikation in CORBA
- Peer-To-Peer Systeme
  - Definition und Entwicklungsgeschichte
  - Overlay Routing
  - Routing-Algorithmen am Beispiel von Pastry
- Zeit in Verteilten Systemen
  - Algorithmus von Cristian
  - Berkeley-Algorithmus
  - NTP
  - Logische Uhren: Lamport und Vektoruhren
- Wechselseitiger Ausschluss
  - Serverbasierter MUTEX
  - Ringbasierter MUTEX
  - Ricart & Agrawala
  - Maekawa
- Wahlalgorithmen
  - Chang & Roberts
  - Bully-Algorithmus
- Multicast-Kommunikation: Prinzipien und Algorithmen
  - B-Multicast

- R-Multicast
- Ordnung von Multicasts (FIFO, logisch, total)

- Nebenläufigkeitskontrolle
  - Ziele und grundsätzliche Verfahren
  - Eigenschaften von Transaktionen
  - Serielle Äquivalenz und Operationskonflikte
  - (Striktes) 2-Phasen-Locking
  - Erkennung und Beseitigung von Deadlocks
  - Commit-Protokolle bei verteilten Transaktionen
  - Sperrverfahren bei verteilten Transaktionen
- Architekturen Verteilter Systeme
  - Dimensionen
  - Nachrichtenbasierung: JMS
  - Dienstbasierung: Web Services
  - Komponentenbasierung: EJB
  - Muster
  - Architektur: Schichten
  - Architektur: Model View Controller
  - Architektur: Broker
  - Entwurfsmuster
- Kooperative Systeme
  - Entwicklungsgeschichte
  - Definition und Beispiele: CSCW / Groupware / Social Software
  - Klassifikation von Groupware: Raum/Zeit, Funktion
  - Rechnergestützte Kommunikation
  - Unterstützung von Meetings und Entscheidungen
  - Gemeinsame Applikationen und Artefakte