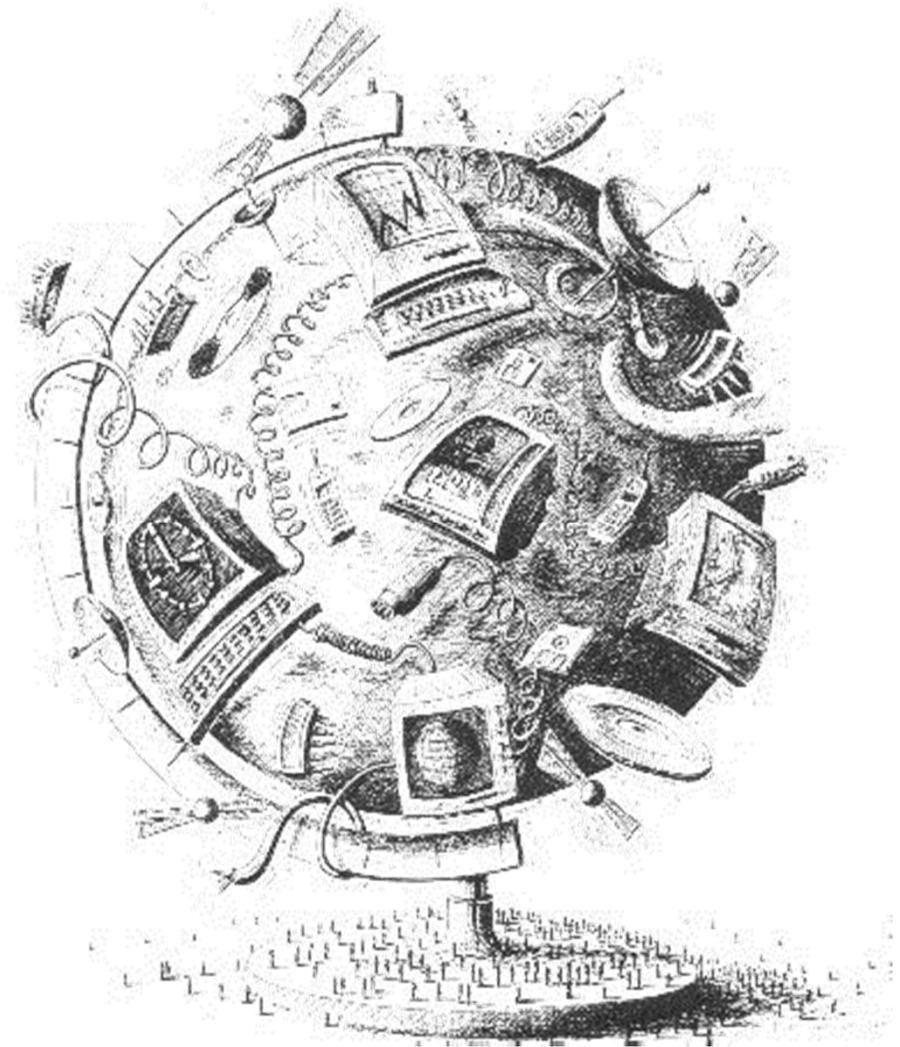


Verteilte Systeme

*Friedemann
Mattern*



Prüfungsrelevant ist der Inhalt der Vorlesung, nicht alleine der Text dieser Foliensammlung!

Wer bin ich? Wer sind wir?



Prof. **Friedemann Mattern**

+ mehrere Assistenten
und Assistentinnen



Fachgebiet „Verteilte Systeme“
im Departement Informatik,
Institut für Pervasive Computing



Wer bin ich? Wer sind wir?



Prof. **Friedemann Mattern**

+ mehrere Assistenten
und Assistentinnen

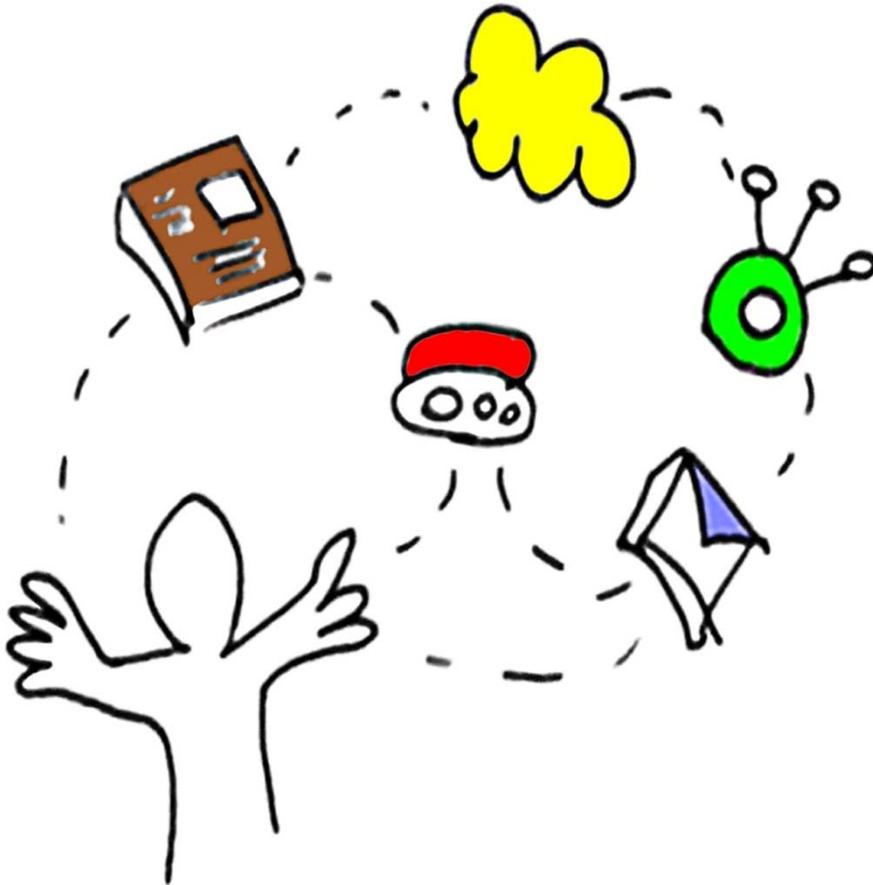
Matthias Kovatsch

Ansprechperson für or-
ganisatorische Aspekte
kovatsch@inf.ethz.ch



Fachgebiet „Verteilte Systeme“
im Departement Informatik,
Institut für Pervasive Computing

Mit was beschäftigen wir uns?



- Infrastruktur für verteilte Systeme
- Internet der Dinge
- Ubiquitous Computing
- Sensornetze
- Smart energy
- Verteilte Anwendungen und Algorithmen

Mehr zu uns:
www.vs.inf.ethz.ch

Organisatorisches zur Vorlesung

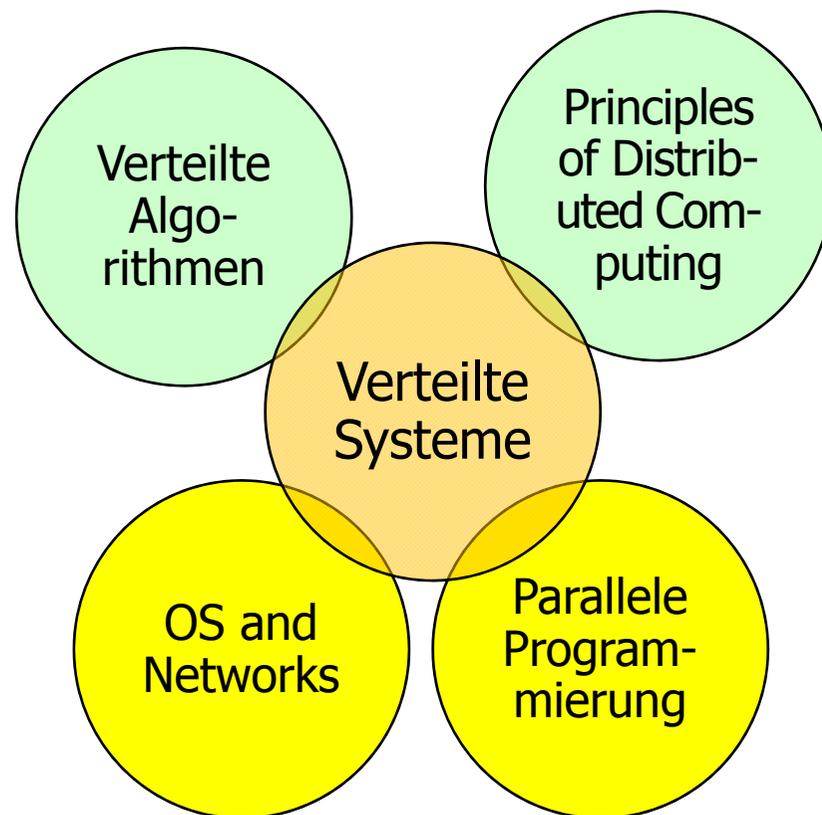
- Format: 6G+1A: **Vorlesung** und **Praktikum** integriert
 - Mo 9:15 - 12:00, Fr 9:15 - 12:00, jew. NO C 6
 - Praktikum ist inhaltlich *komplementär zur Vorlesung* (mobile Kommunikationsplattformen: Android, HTC Desire etc.)
 - Gelegentliche *Assistentenstunden* (zu den "Vorlesungsterminen") zur Besprechung der Praktikumsaufgaben und Vertiefung des Stoffes
 - Gelegentliche *Denkaufgaben* (ohne Lösung...) in der Vorlesung
- Sinnvolle **Vorkenntnisse** (Grundlagen)
 - 4 Semester der Bachelorstufe Informatik
 - Grundkenntnisse Computernetze und Betriebssysteme (z.B. Prozessbegriff, Synchronisation)
 - UNIX, Java ist hilfreich

Organisatorisches (2)



- **Folienkopien** jeweils einige Tage nach der Vorlesung
 - im pdf-Format bei www.vs.inf.ethz.ch/edu
- **Prüfung** schriftlich
 - bewertete Praktikumsaufgaben gehen in die Prüfungsnote ein
- Vorlesung ab November: Prof. **Roger Wattenhofer**

Einordnung der Vorlesung



- „Verteilte Systeme“ ist ein **Querschnittsthema**
- Gewisse Überschneidungen mit anderen Vorlesungen unvermeidlich

Thematisch verwandte Veranstaltungen (Masterstufe)

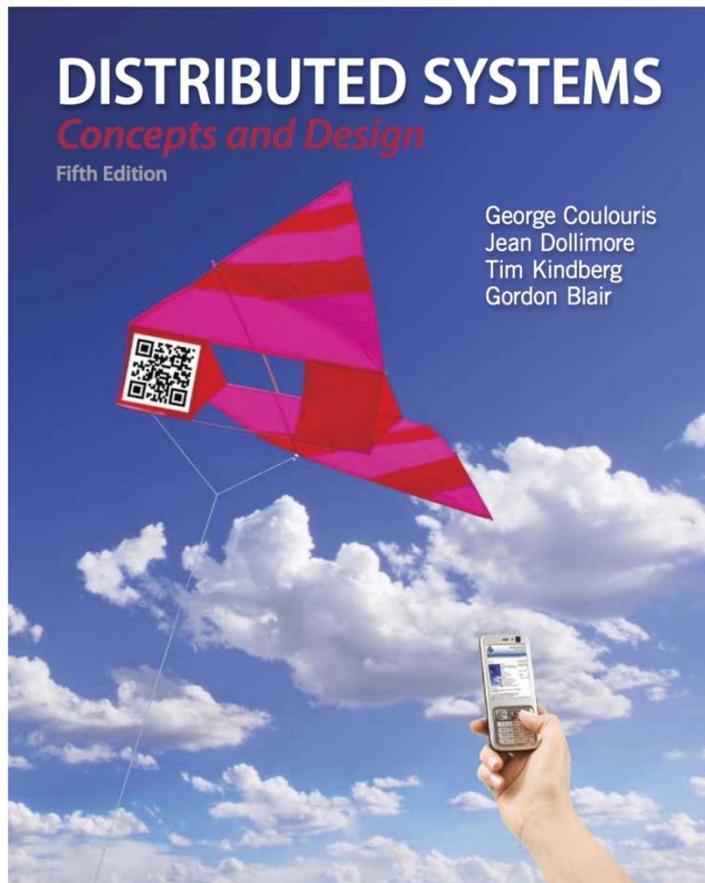


- Ubiquitous Computing
- Enterprise Application Integration - Middleware
- Web Services and Service Oriented Architectures
- Verteilte Algorithmen
- Principles of Distributed Computing
- Distributed Information Systems
- Advanced Operating Systems
- Fehlertoleranz in Verteilten Systemen

Es werden nicht
immer alle Vorle-
sungen angeboten!

-
- Einschlägige Seminare
 - Praktikum ("Lab")
 - Semester- und Masterarbeit

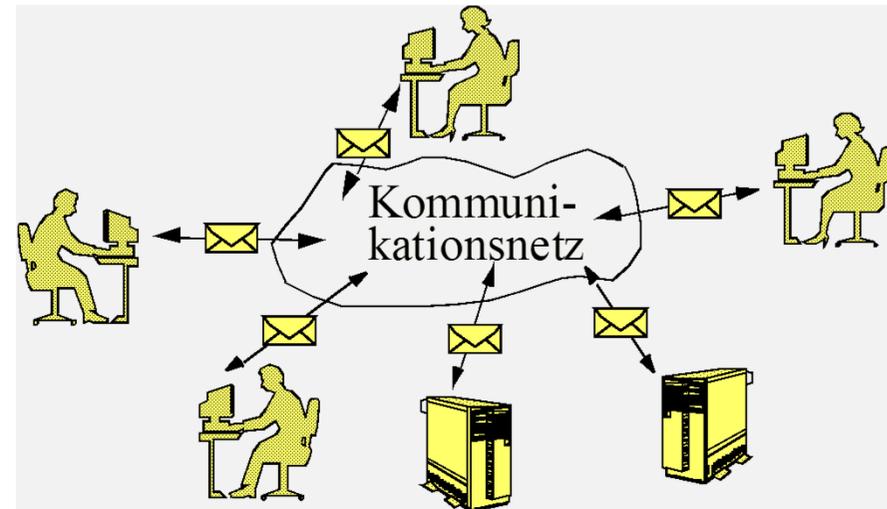
Literatur



- **G. Coulouris, J. Dollimore, T. Kindberg, G. Blair:** Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley, 2011
- **A. Tanenbaum, M. van Steen:** Distributed Systems: Principles and Paradigms (2nd ed.). Prentice-Hall, 2007
- **O. Haase:** Kommunikation in verteilten Anwendungen (2. Aufl.). R. Oldenbourg Verlag, 2008
- **A. Schill, T. Springer:** Verteilte Systeme (2. Aufl.). Springer Vieweg, 2011

„Verteiltes System“ – zwei Definitionen

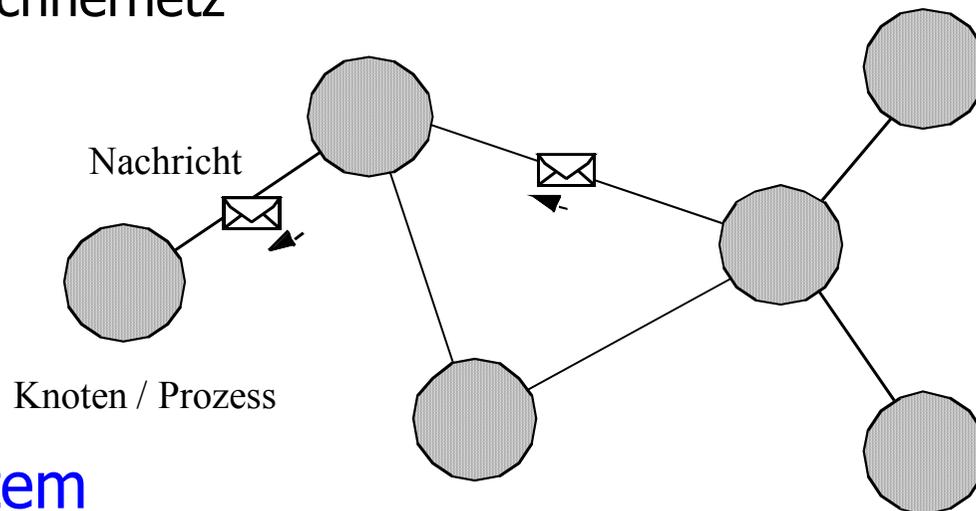
- A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communication network. — *H. Bal*



- A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. — *Leslie Lamport*
 - welche Problemaspekte stecken hinter Lamports Charakterisierung?

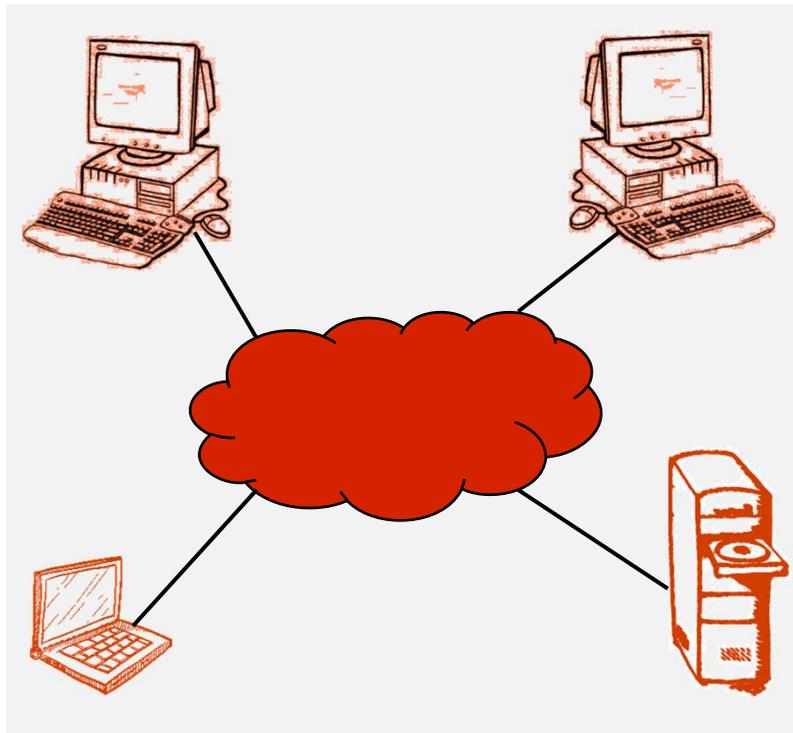
„Verteiltes System“

- Physisch verteiltes System
 - Compute-Cluster ... Rechnernetz



- Logisch verteiltes System
 - Prozesse (Objekte, Agenten)
 - Verteilung des Zustandes (keine globale Sicht)
 - Keine gemeinsame Zeit (globale, genaue Uhr)

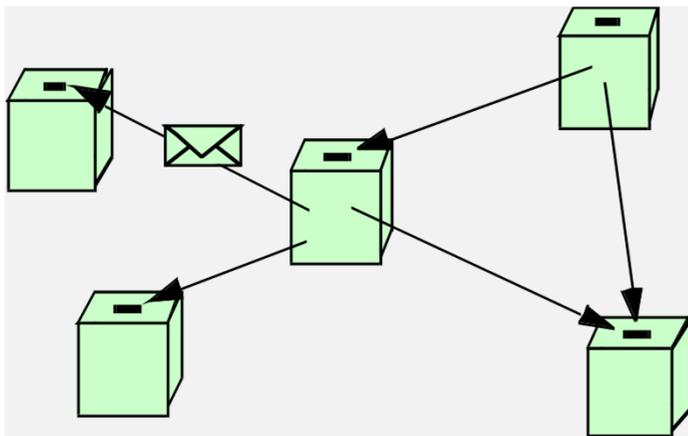
Sichten verteilter Systeme (1)



- Computernetz mit "Rechenknoten", z.B.
 - Compute-Cluster
 - Local Area Network
 - Internet
- Relevante Aspekte:
 - Routing, Adressierung,...

Zunehmende Abstraktion 

Sichten verteilter Systeme (2)

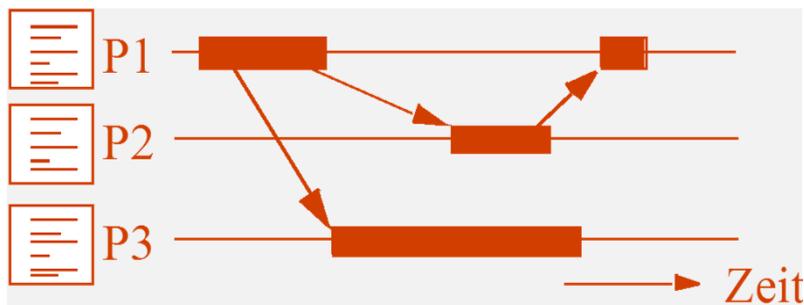


kommunizierende Prozesse,
kooperierende Objekte

- **Objekte** in Betriebssystemen, Middleware, Programmiersprachen
- "Programmierersicht"
 - z.B. Client mit API zu Server

Zunehmende Abstraktion 

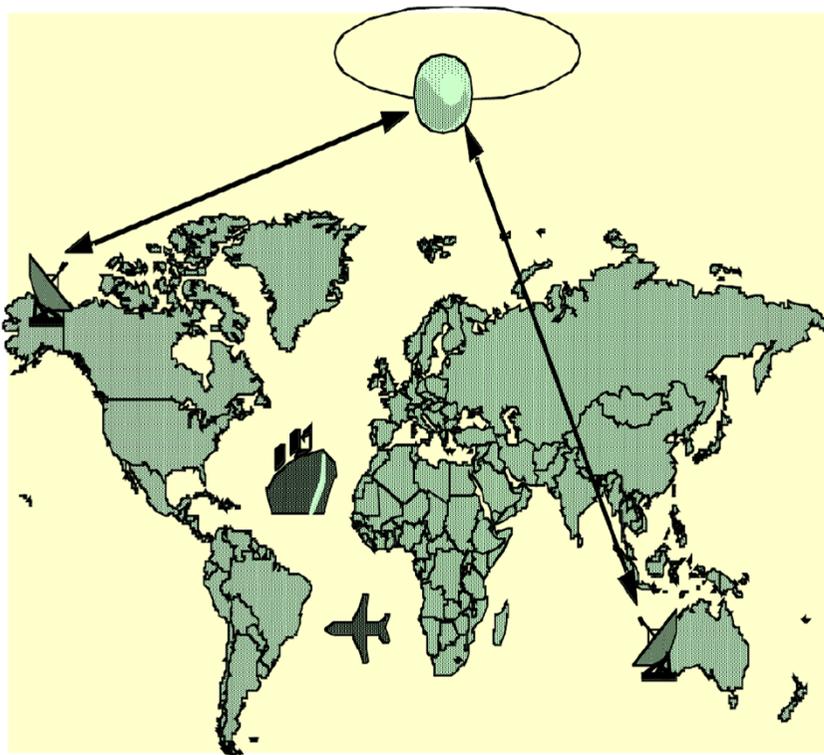
Sichten verteilter Systeme (3)



- Algorithmen- und Protokollebene
 - Aktionen, Ereignisfolgen
 - Konsistenz, Korrektheit

- Man kann verteilte Systeme auf **verschiedenen Abstraktionsstufen** betrachten
- Es sind dabei jeweils **unterschiedliche Aspekte** relevant und interessant

Die verteilte Welt



Auch die "reale Welt" ist ein **verteilt**es System:

- viele gleichzeitige ("parallele") Aktivitäten
- exakte globale **Zeit** nicht erfahrbar / vorhanden
- keine konsistente Sicht des **Gesamtzustandes**
- Kooperation durch explizite **Kommunikation**
- **Ursache** und **Wirkung** zeitlich (und räumlich) getrennt

Warum verteilte Systeme?

- **Es gibt inhärent geographisch verteilte physische Systeme**
 - z.B. Steuerung einer Fabrik, Zweigstellennetz einer Bank (Zusammenführen / Verteilen von Information)
- **Electronic commerce**
 - kooperative Informationsverarbeitung räumlich getrennter Institutionen (z.B. Reisebüros, Kreditkarten,...)
- **Mensch-Mensch-Telekommunikation**
 - E-Mail, Diskussionsforen, Blogs, digitale soz. Netze, IP-Telefonie,...
- **Globalisierung von Diensten**
 - Skaleneffekte, Outsourcing,...

Wirtschaftliche Aspekte

- Outsourcing von Diensten, Verlagerung in eine „Cloud“, kann günstiger sein als eine lokal-zentralisierte Lösung
- Compute-Cluster manchmal besseres Preis-Leistungs-verhältnis als Supercomputer

Verteilte Systeme als „Verbunde“



- Verteilte Systeme **verbinden** räumlich (oder logisch) getrennte Komponenten zu einem bestimmten **Zweck**
-

- **Systemverbund**

- gemeinsame Nutzung von Betriebsmitteln, Geräten,...
- einfache inkrementelle Erweiterbarkeit

- **Funktionsverbund**

- Kooperation bzgl. Nutzung jeweils spezifischer Eigenschaften

- **Lastverbund**

- Zusammenfassung der Kapazitäten

- **Datenverbund**

- allgemeine Bereitstellung von Daten

- **Überlebensverbund**

- Redundanz durch Replikation

Historische Entwicklung (1)



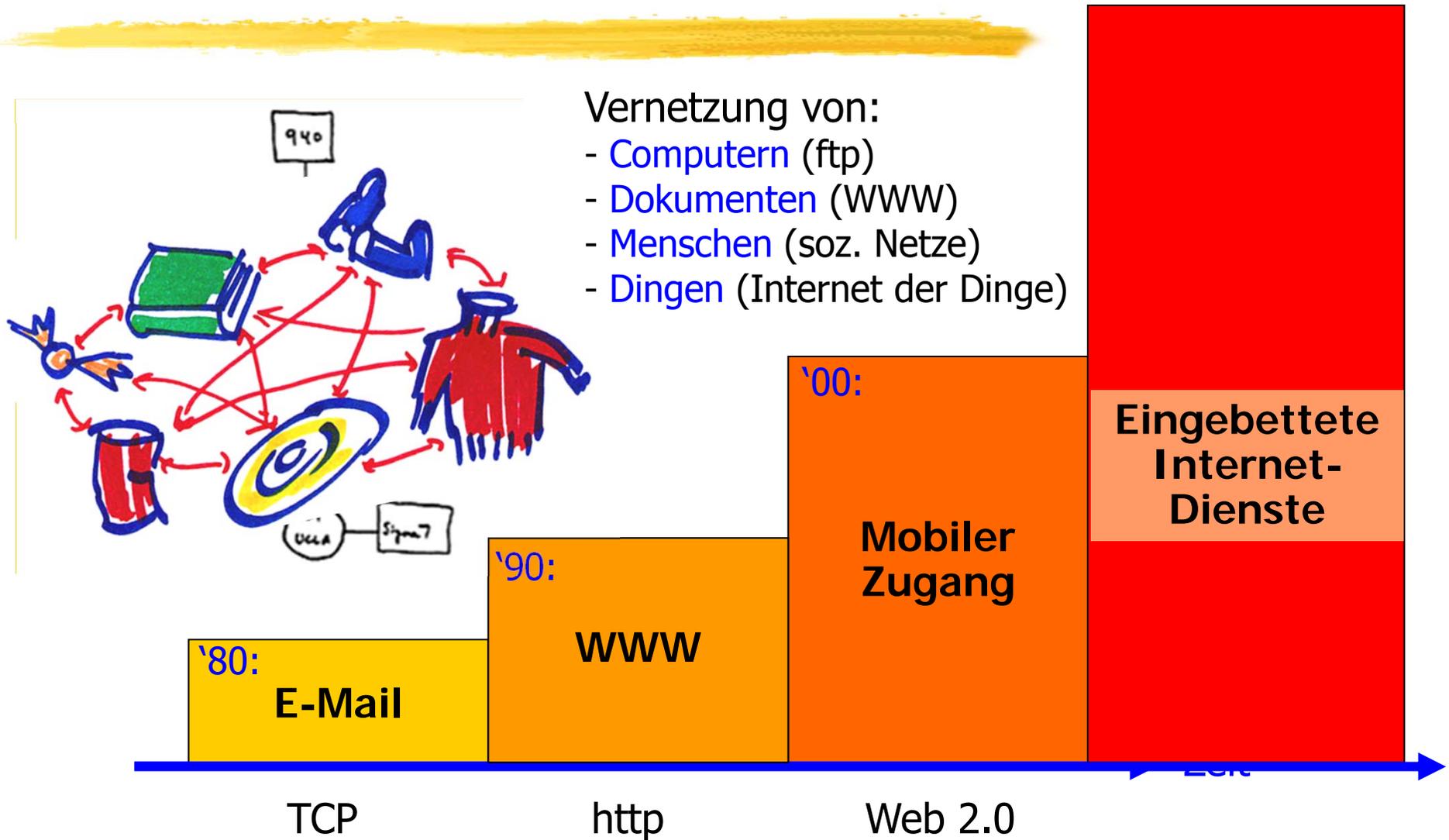
- **Rechner-zu-Rechner-Kommunikation**
 - Zugriff auf entfernte Daten (“Datenfernübertragung”, DFÜ)
 - dezentrale Informationsverarbeitung war zunächst ökonomisch nicht sinnvoll (zu teuer, Fachpersonal nötig)
→ Master-Slave-Beziehung (“Remote Job Entry”, Terminals)
- **ARPA-Netz (Prototyp des Internet)**
 - symmetrische Kooperationsbeziehung (“peer to peer”)
 - file transfer, remote login, E-Mail
 - Internet-Protokollfamilie (TCP/IP,...)

Historische Entwicklung (2)



- **Workstation- und PC-Netze (LAN)**
 - bahnbrechende, frühe Ideen bei XEROX-PARC (XEROX-Star als erste Workstation, Desktop-Benutzerinterface, Ethernet, RPC, verteilte Dateisysteme,...)
- **Kommerzielle Pionierprojekte als Treiber (WAN)**
 - z.B. Reservierungssysteme, Banken, Kreditkarten
- **Web / Internet als Plattform**
 - für electronic commerce etc.
 - web services
 - neue, darauf aufbauende Dienste
- **Mobile Geräte**
 - z.B. Smartphones
- **Internet der Dinge**

Änderung der Vernetzungsqualität aus historischer Sicht



Historie von Konzepten

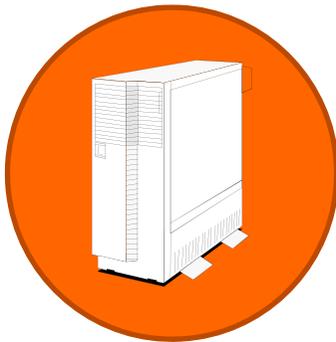
- **Concurrency, Synchronisation**
 - war bereits klassisches Thema bei Datenbanken und Betriebssystemen
- **Programmiersprachen**
 - „kommunizierende“ Objekte
- **Parallele und verteilte Algorithmen**
- **Semantik von Kooperation / Kommunikation**
 - mathematische Modelle für Verteiltheit (z.B. CCS, Petri-Netze)
- **Abstraktionsprinzipien**
 - Schichten, Dienstprimitive,...
- **Verständnis grundlegender Phänomene der Verteiltheit**
 - Konsistenz, Zeit, Zustand,...

Entwicklung „guter“ Konzepte, Modelle, Abstraktionen etc. zum Verständnis der Phänomene **dauert oft lange** (notwendige Ordnung und Sichtung des verfügbaren Gedankenguts)

Diese sind jedoch für die Lösung praktischer Probleme **hilfreich**, oft sogar **notwendig!**

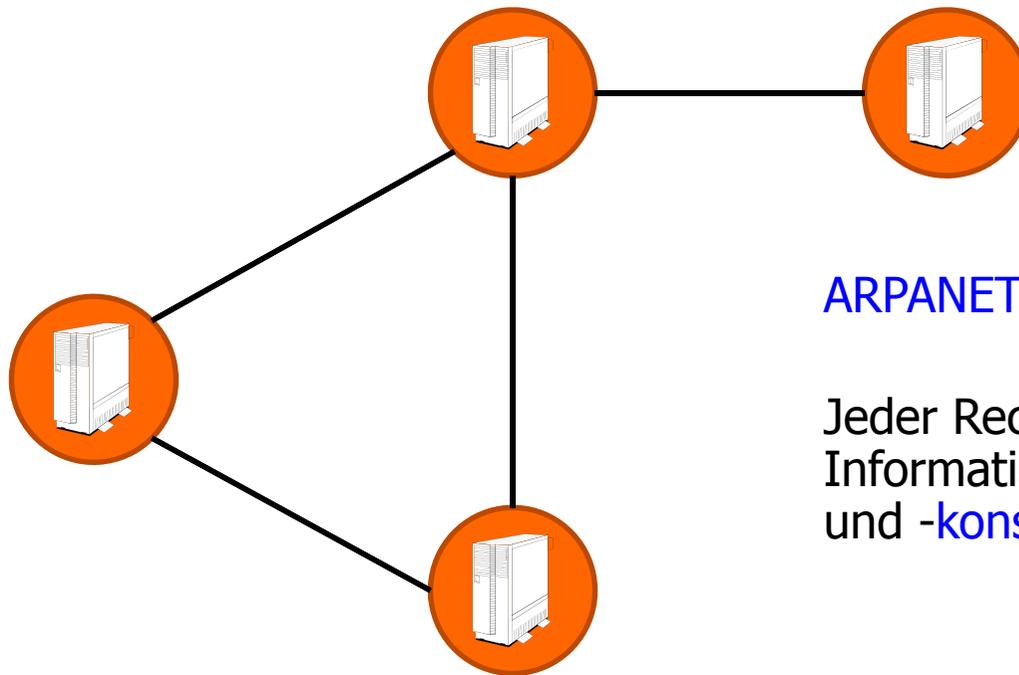
Architekturen verteilter Systeme

- Zu Anfang waren Systeme **monolithisch**



- Nicht verteilt / vernetzt
- **Mainframes**
- **Terminals** als angeschlossene „Datensichtgeräte“ („Datenendgerät“: Fernschreiber [tty], ASCII)

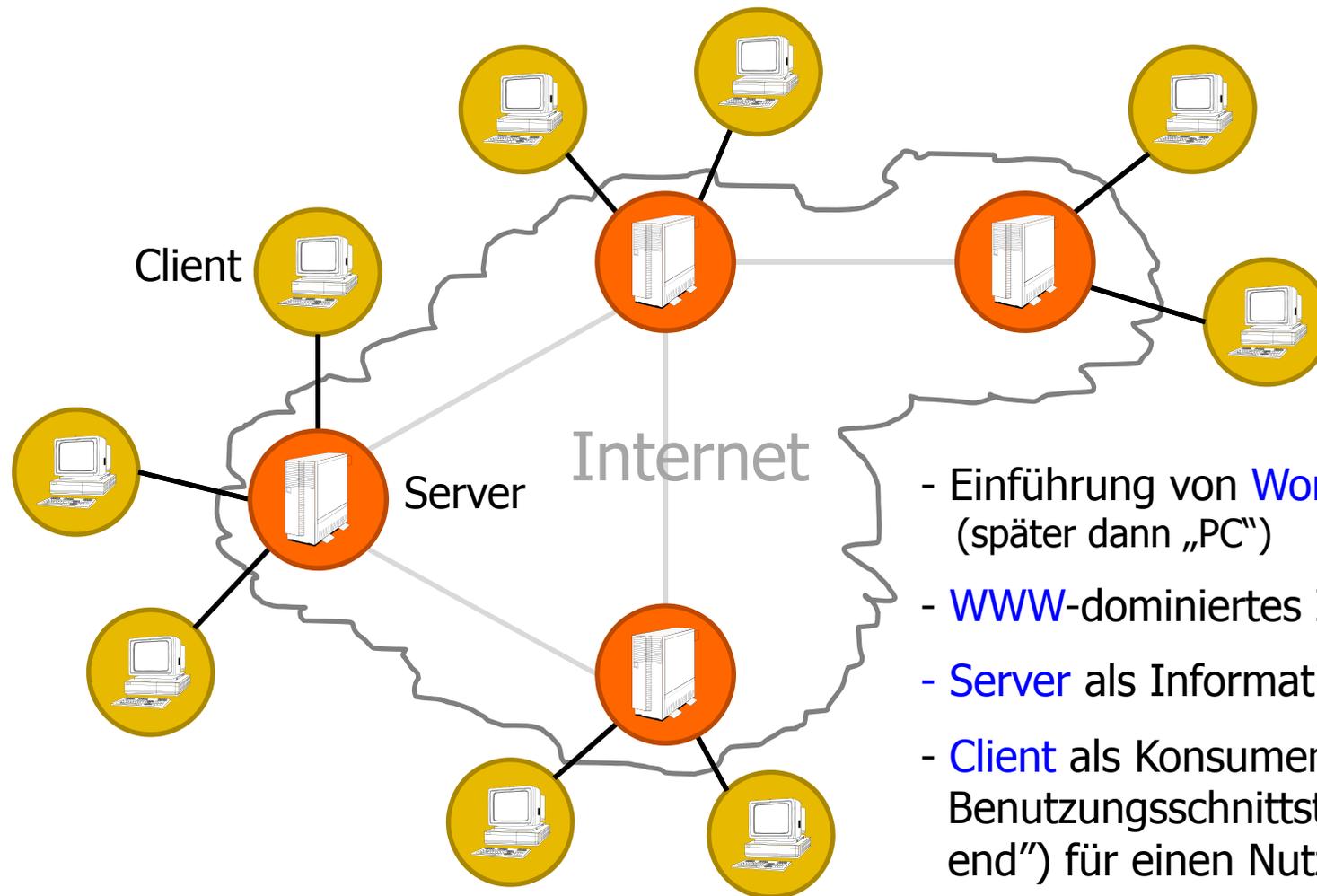
Architekturen verteilter Systeme: Peer-to-Peer



ARPANET 1969

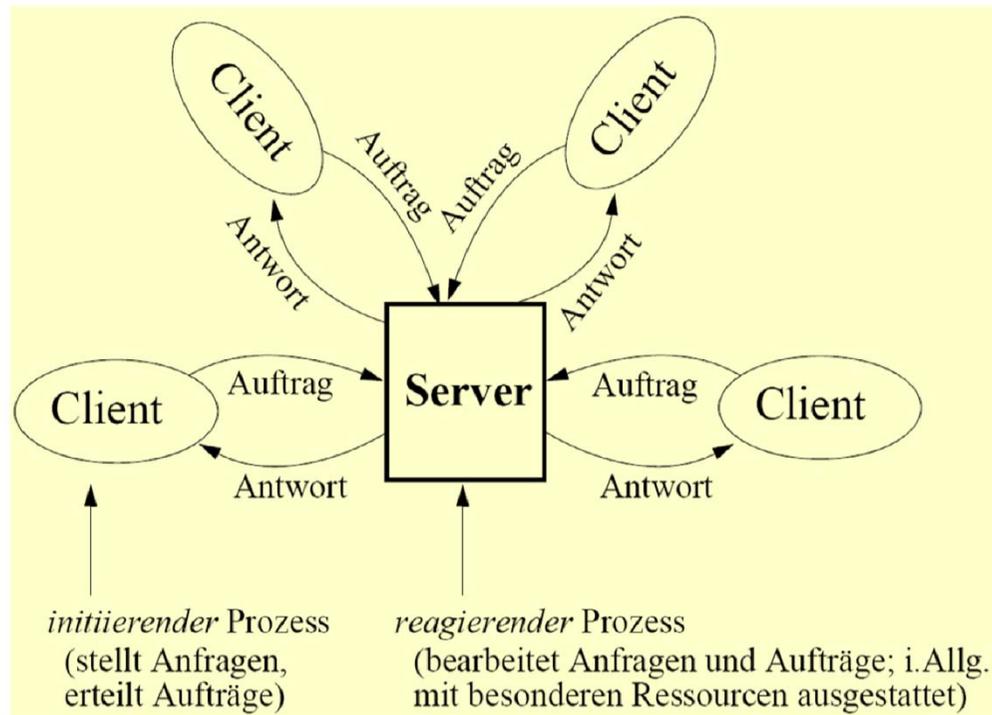
Jeder Rechner **gleichzeitig**
Informations**anbieter**
und **-konsument**

Architekturen verteilter Systeme: Client-Server



- Einführung von **Workstations** (später dann „PC“)
- **WWW**-dominiertes Internet
- **Server** als Informationsanbieter
- **Client** als Konsument; gleichzeitig Benutzungsschnittstelle („front end“) für einen Nutzer

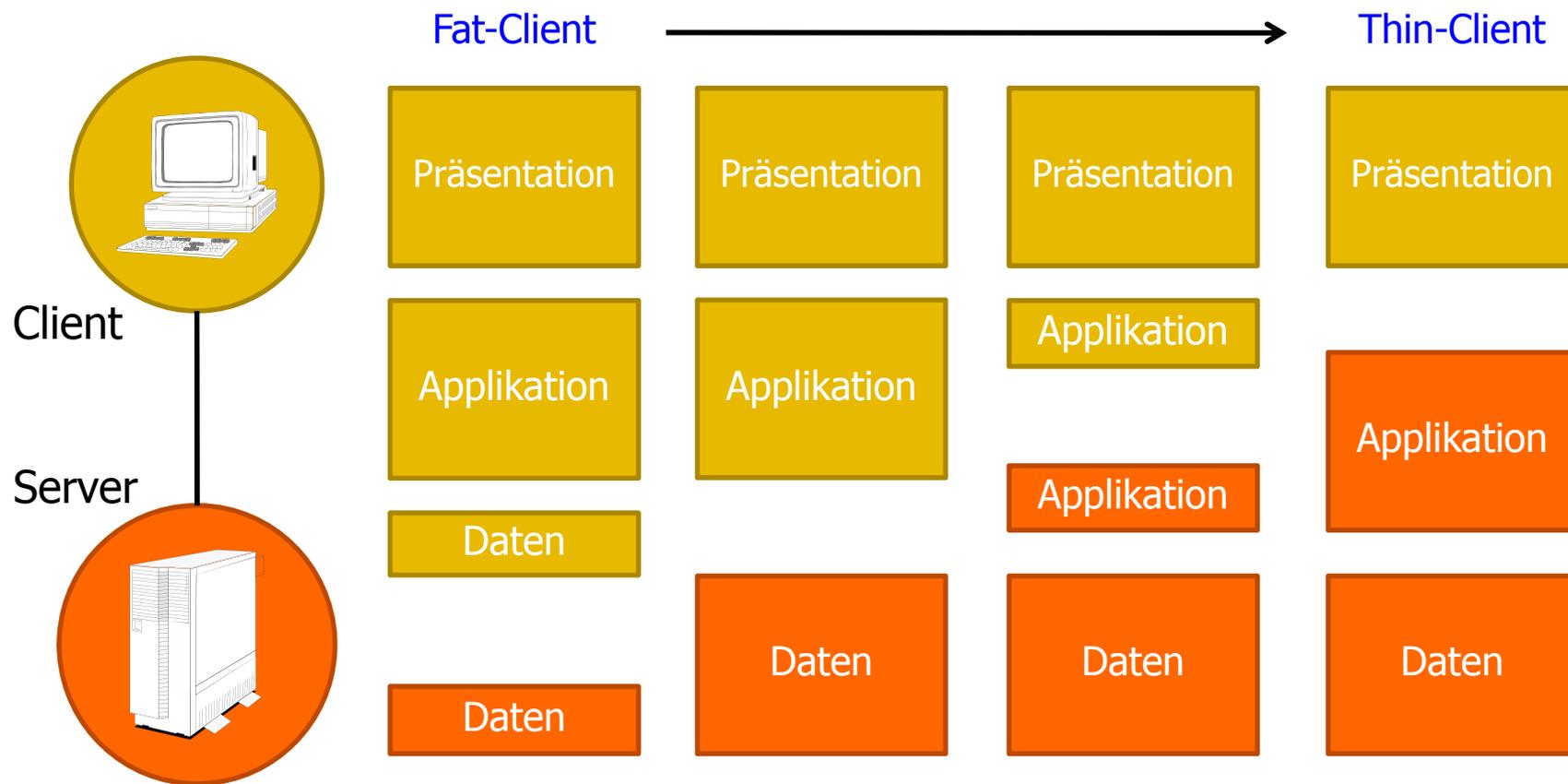
Client-Server



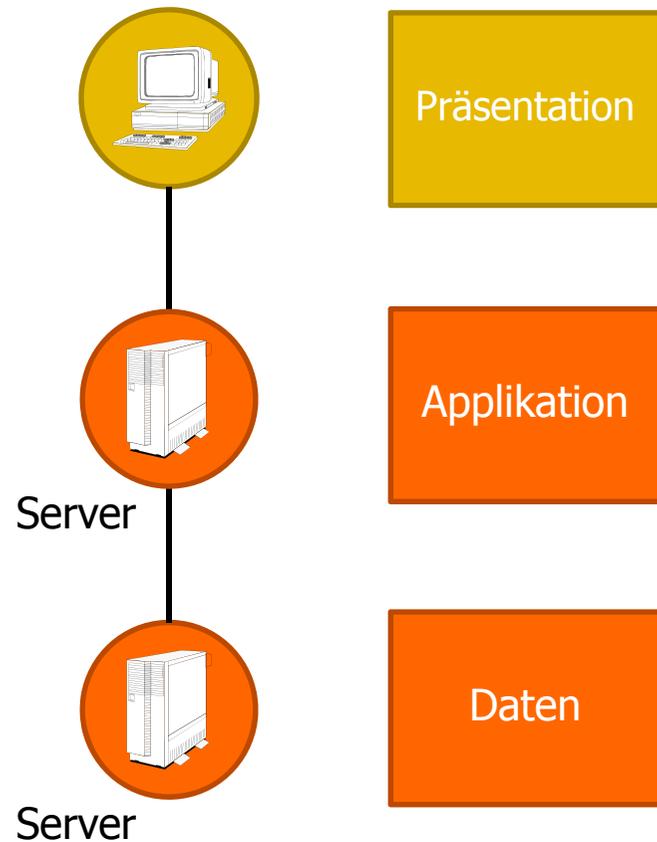
Populär auch wegen des eingängigen Modells

- entspricht Geschäftsvorgängen in unserer Dienstleistungsgesellschaft
- gewohntes Muster → intuitive Struktur, gute Überschaubarkeit

Architekturen verteilter Systeme: Fat- und Thin-Client



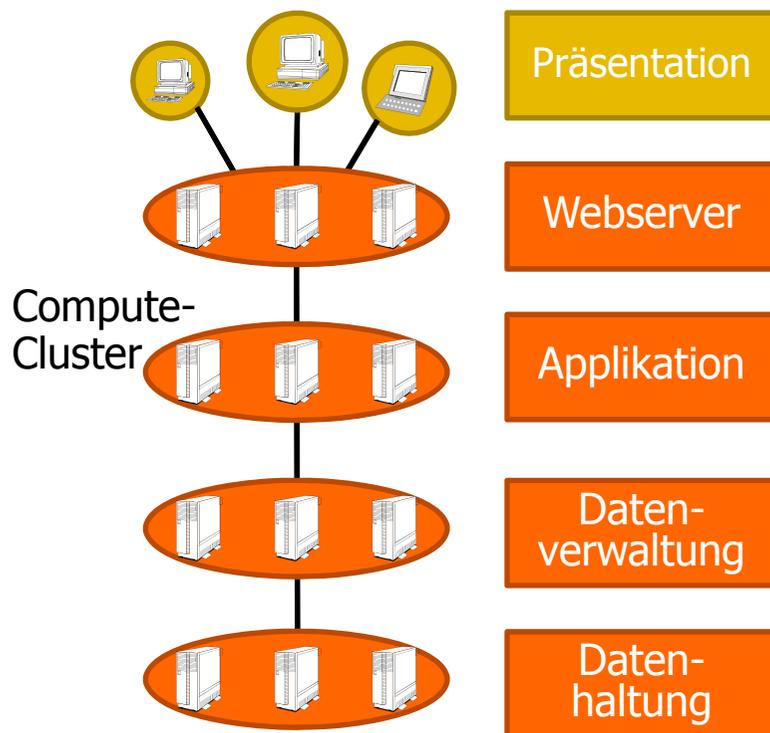
Architekturen verteilter Systeme: 3-Tier



Verarbeitung wird auf **mehrere physikalische Einheiten** verteilt

- Logische Schichten mit minimierten Abhängigkeiten
- Leichtere Wartung
- Einfaches Austauschen

Architekturen verteilter Systeme: Multi-Tier

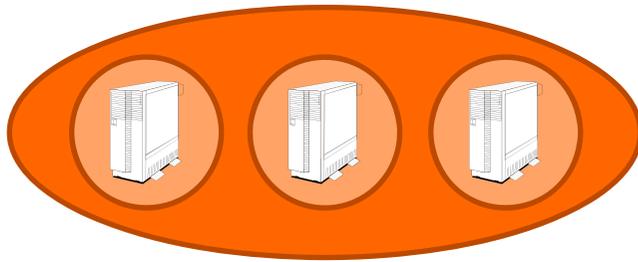


Weitere Schichten sowie mehrere physikalische Einheiten pro Schicht („Compute-Cluster“) erhöhen die **Skalierbarkeit** und **Flexibilität**

Mehrere Webserver ermöglichen z.B. **Lastverteilung**

Verteilte Datenbanken in der Datenhaltungsschicht bietet Sicherheit durch Replikation und hohen Durchsatz

Architekturen verteilter Systeme: Compute-Cluster



Vernetzung kompletter Einzelrechner

Räumlich konzentriert (wenige Meter)

Sehr schnelles Verbindungsnetz

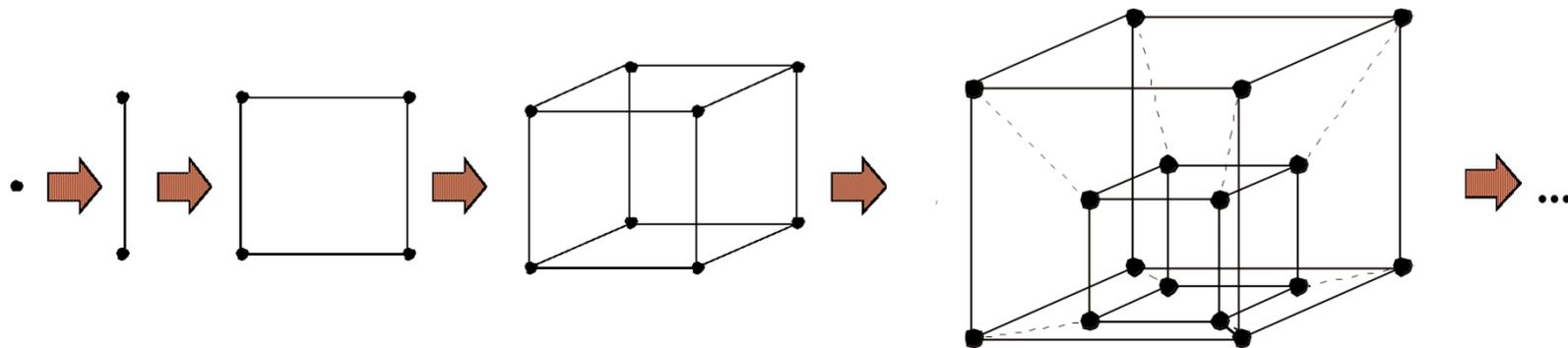
Es gibt diverse **Netztopologien**, um die Einzelrechner (*als Knoten in einem Graphen*) miteinander zu verbinden – diese sind unterschiedlich hinsichtlich

- Skalierbarkeit der Topologie
- Routingkomplexität
- Gesamtzahl der Einzelverbindungen
- maximale bzw. durchschnittliche Entfernung zweier Knoten
- Anzahl der Nachbarn eines Knotens
- ...

Bestimmt mit die Kosten und die Leistungsfähigkeit eines Systems

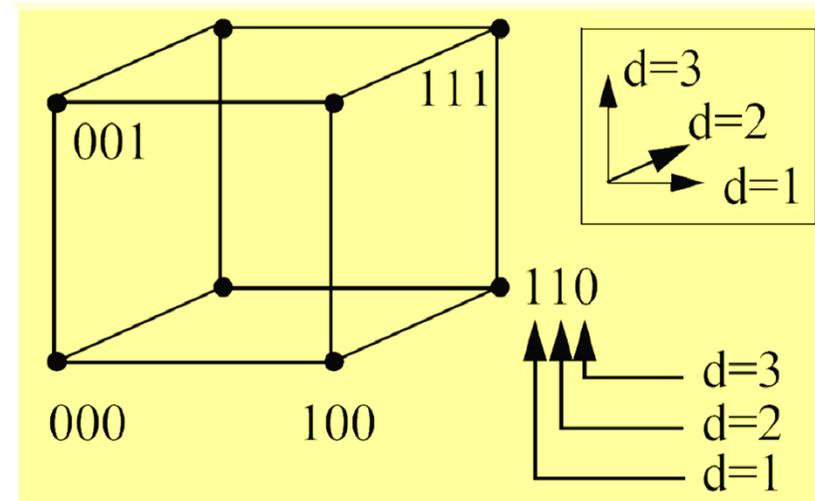
Beispiel: Hypercube-Verbindungstopologie

- **Würfel der Dimension d**
 - Vorteil: einfaches Routing, kurze Weglängen
 - Nachteil: Viele Einzelverbindungen ($O(n \log n)$ bei n Knoten)
- **Rekursives Konstruktionsprinzip**
 - Hypercube der Dimension 0: Einzelrechner
 - Hypercube der Dimension $d+1$: „Nimm zwei Würfel der Dimension d und verbinde korrespondierende Ecken“

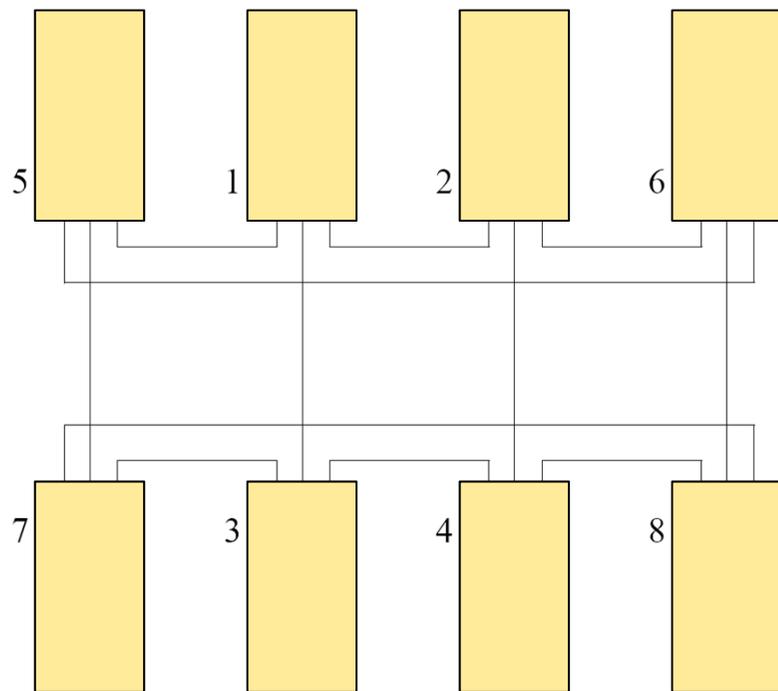


Routing beim Hypercube

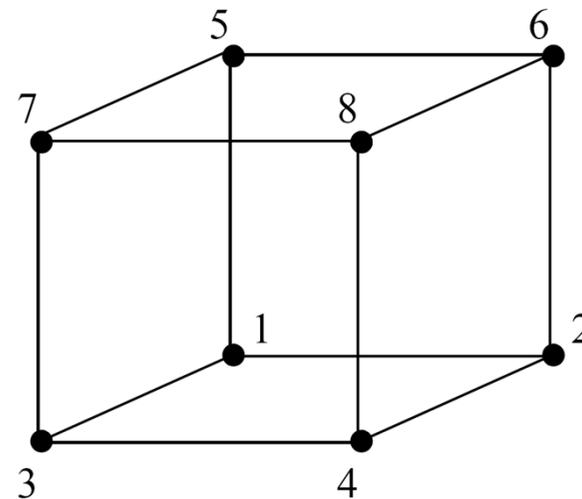
- Knoten **systematisch nummerieren** (entspr. rekursivem Aufbau)
- Zieladresse **bitweise xor** mit Absenderadresse
- Wo sich eine "1" findet, in diese Dimension muss gewechselt werden
- Maximale Weglänge: d ; durchschnittliche Weglänge = $d/2$ (Induktionsbeweis als einfache Übung)



Hypercube

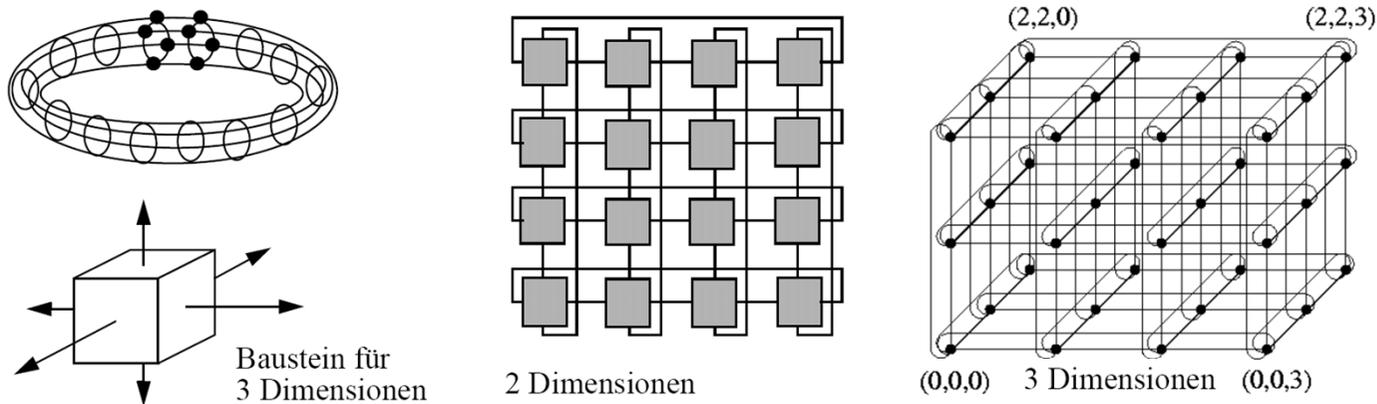


- Dies stellt auch einen **Hypercube** dar
 - ist nur nicht als Würfel gezeichnet



Eine andere Verbindungstopologie: Der d-dimensionale Torus

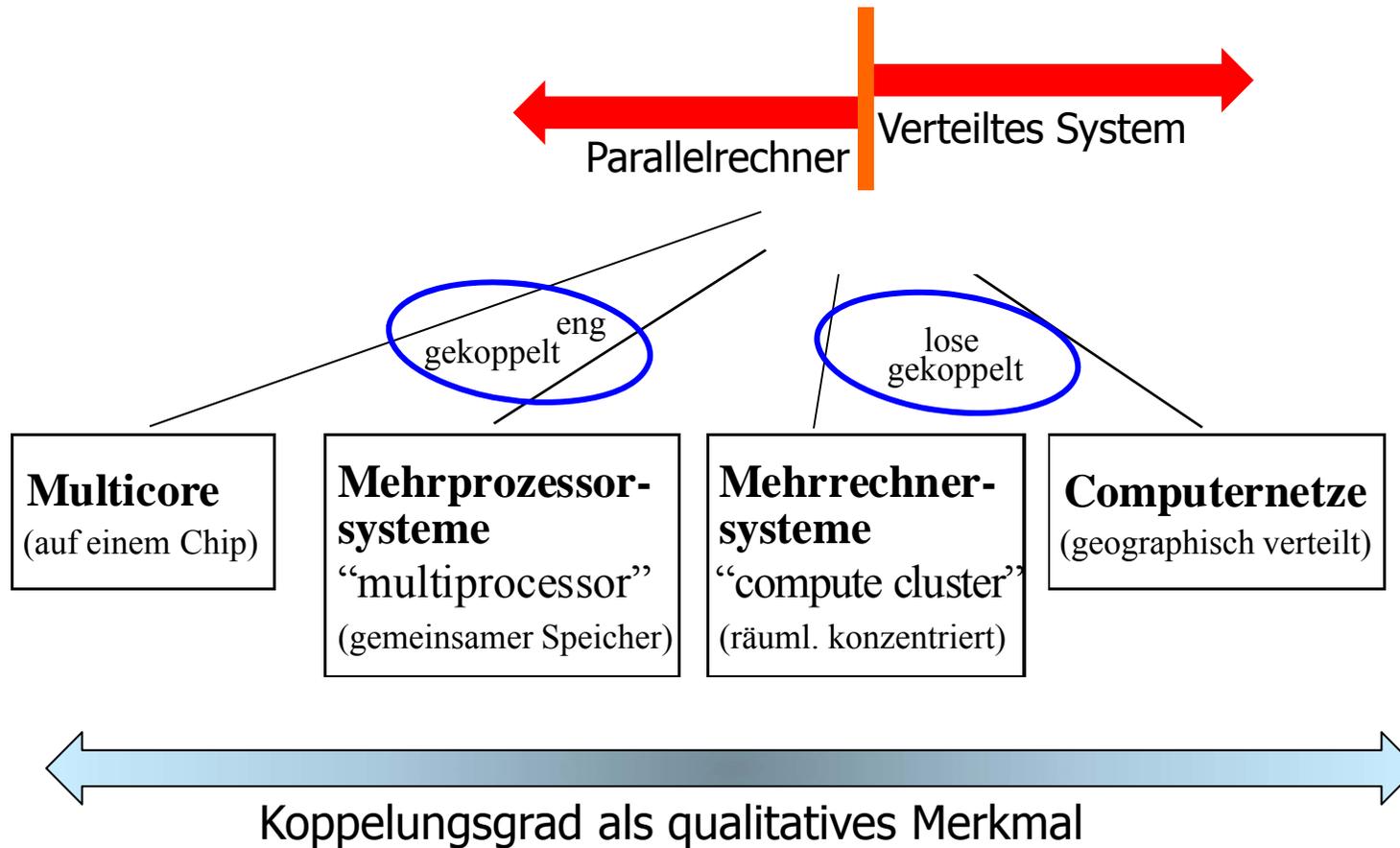
- Gitter in d Dimensionen mit "wrap-around"



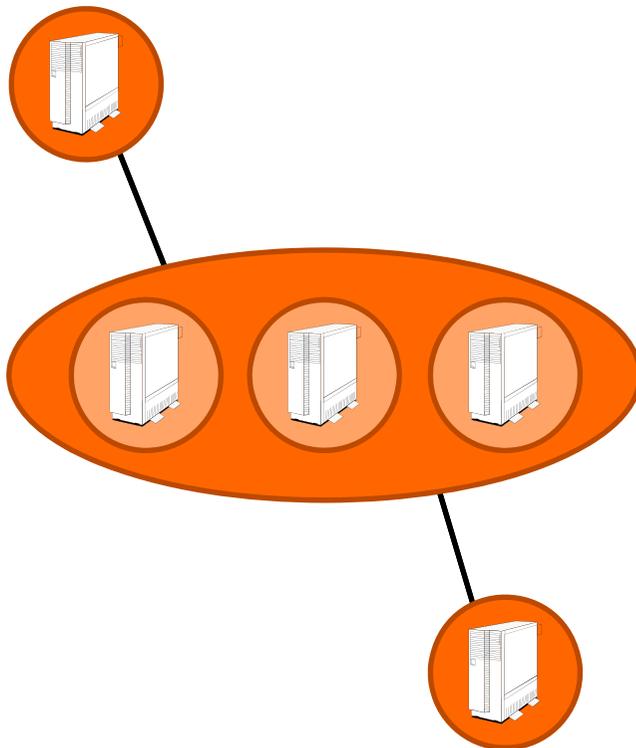
- Rekursives Konstruktionsprinzip:** Nimm w_d gleiche Exemplare der Dimension $d-1$ und verbinde korrespondierende Elemente zu einem Ring
 - Sonderfall **einfacher Ring:** $d = 1$
 - Sonderfall **Hypercube:** d-dimensionaler Torus mit $w_i = 2$ für alle Dimensionen i

Es gibt noch einige andere sinnvolle Verbindungstopologien (auf die wir nicht eingehen)

Parallelrechner ↔ verteiltes System



Architekturen verteilter Systeme: Service-Oriented Architecture (SOA)



Eine **Unterteilung** der Applikation in einzelne, unabhängige Abläufe innerhalb eines **Geschäftsprozesses** erhöht die Flexibilität weiter

Lose Kopplung zwischen Services über Nachrichten und events (statt RPC = Remote Procedure Call)

Services können bei Änderungen der Prozesse **einfach neu zusammengestellt** werden („development by composition“)

Services können auch von **externen Anbietern** bezogen werden

Oft in Zusammenhang mit **Web-Services**

Architekturen verteilter Systeme: Cloud-Computing

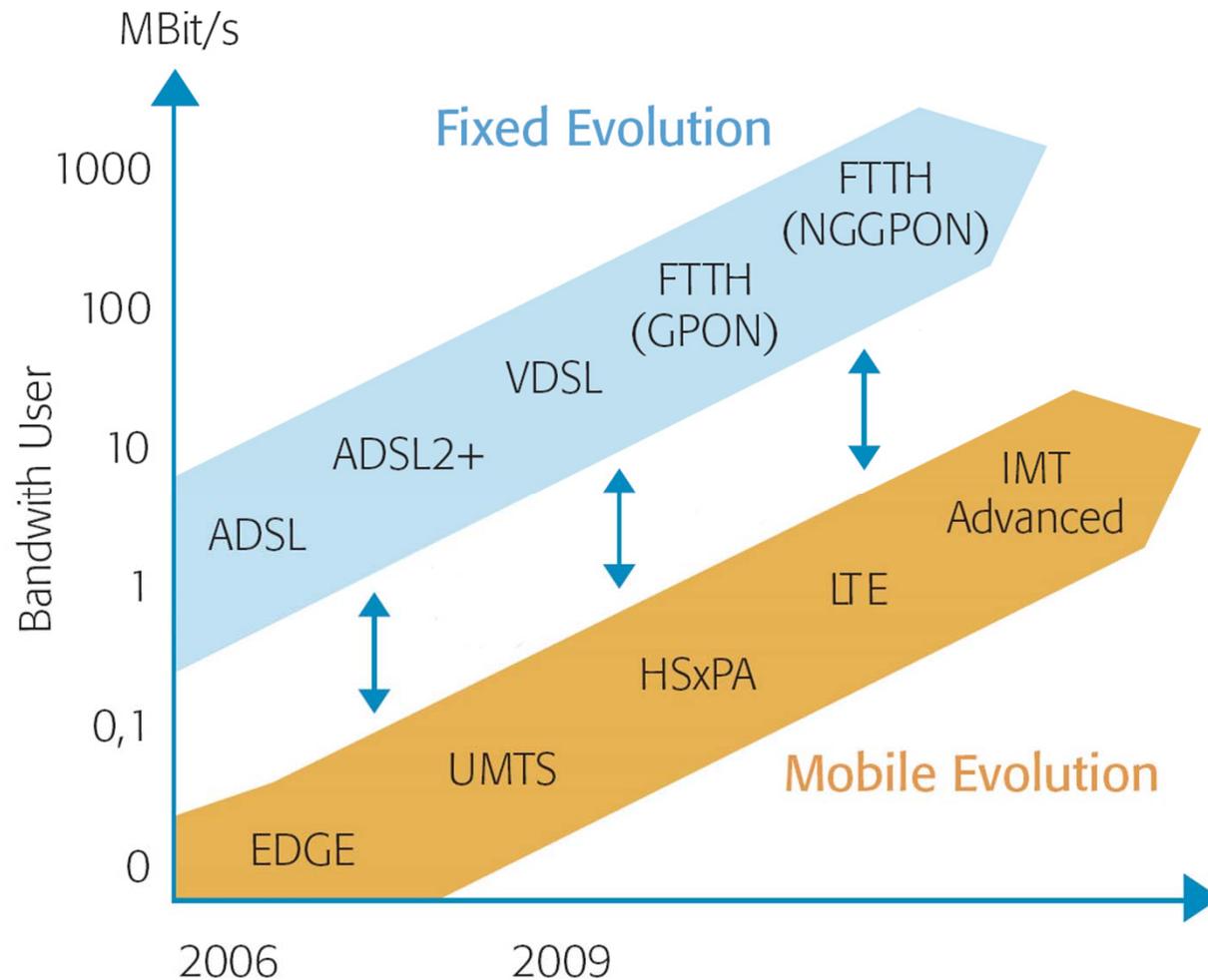


Massive Bündelung der Rechenleistung an zentraler Stelle

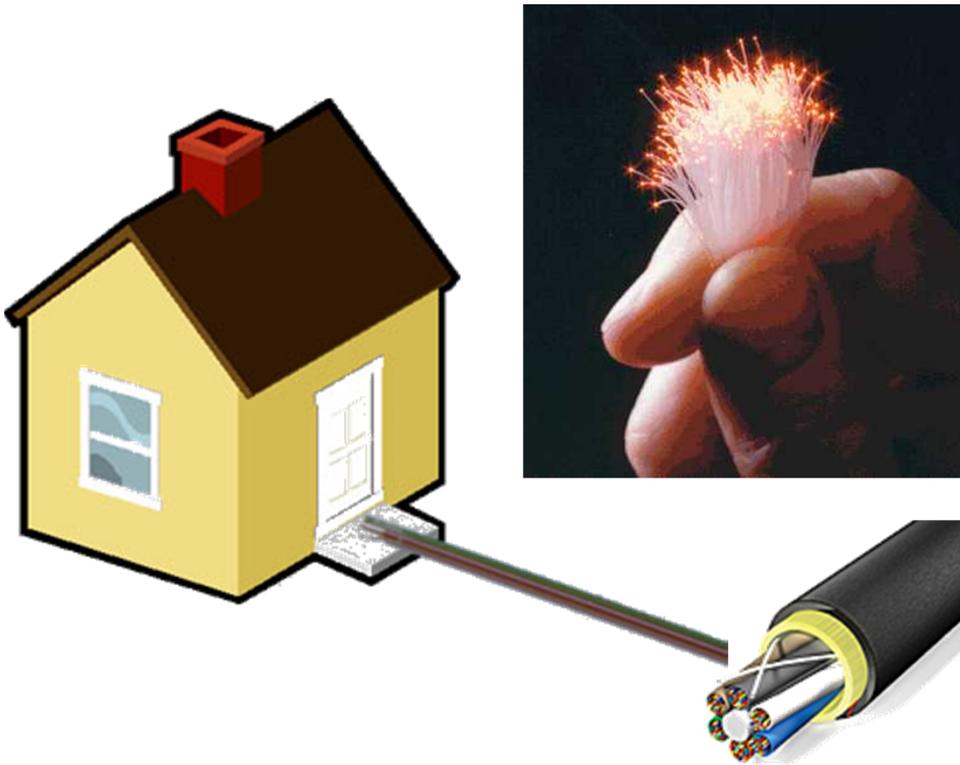
Outsourcen von Applikationen in die Cloud

Internet im Wesentlichen nur noch als Vermittlungsinstanz

Motivierender Trend: Stetige Erhöhung der Bandbreite für Endnutzer



Hochgeschwindigkeit ins Haus



- Konvergenz TV, Telekommunikation und Internet
- Technologiewechsel → erhebliche Investitionen
 - wirtschaftliche Faktoren und Bedingungen

- Telefondraht → Internet → TV
 - TV-Kabel → Telefon → Internet
- } → Glasfaser

Cloud-Computing ...für Privatanutzer



E-Mail wird beim
Provider gespeichert



Cloud-Computing ...für Privatanutzer



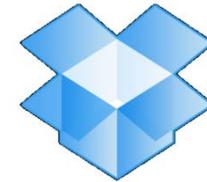
Fotos werden bei **flickr** gespeichert

Videos bei **You Tube**

Cloud-Computing ...für Privatanutzer



Private Dokumente werden bei einem Storage Provider abgelegt



Das Tagebuch wird öffentlich „im Netz“ als Blog geführt

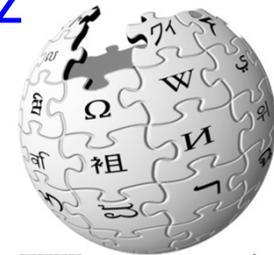


Cloud-Computing ...für Privatanutzer



Informieren tut man
sich im Netz

Google

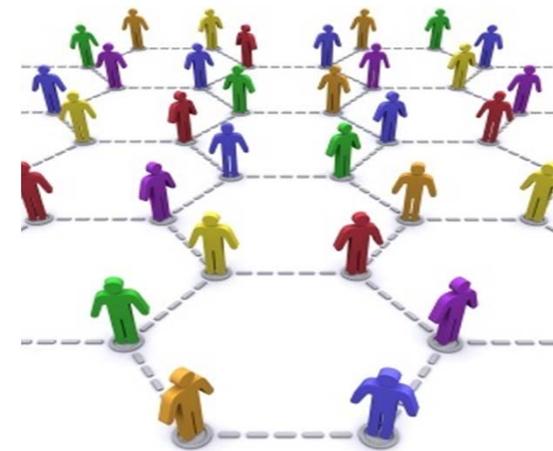


WIKIPEDIA
Die freie Enzyklopädie

Cloud-Computing ...für Privatanutzer



Vernetzen tut man sich bei „social networks“ oder „digital communities“ im Netz



Cloud-Computing ...für Privatanutzer



Plattformen im Netz
nutzt man zum

- Kaufen
- Spielen
- Kommunizieren

- ... **Google**[™]
Romance BETA

Cloud-Computing ...für Privatanutzer



Vorteile für Nutzer:

- von überall zugreifbar
- keine Datensicherung
- keine Softwarepflege

Kein PC, sondern billiges Web-Terminal, Smartphone etc.

Wie Wasserleitungen einst den eigenen Brunnen überflüssig machten...

Cloud-Computing



Voraussetzungen?

- Überall Breitband (fest & mobil)
- Netz-Verlässlichkeit (Versorgungssicherheit, Datenschutz,...)
- Wirtschaftlichkeit

Cloud-Computing



Plattformen:

- Wer betreibt sie? Wo?
- Wer verdient daran?
- Wer bestimmt?
- Wer kontrolliert?
- Welche Nationen profitieren davon?

Beispiel: Google-Datenzentren



- Jedes Datenzentrum hat **10 000 – 100 000 Computer**
- Kostet über **500 Mio \$** (Bau, Infrastruktur, Computer)
- Verbraucht **50 – 100 MW** Energie (Strom, Kühlung)
- Neben Google weitere (z.B. Amazon, Microsoft, Ebay,...)

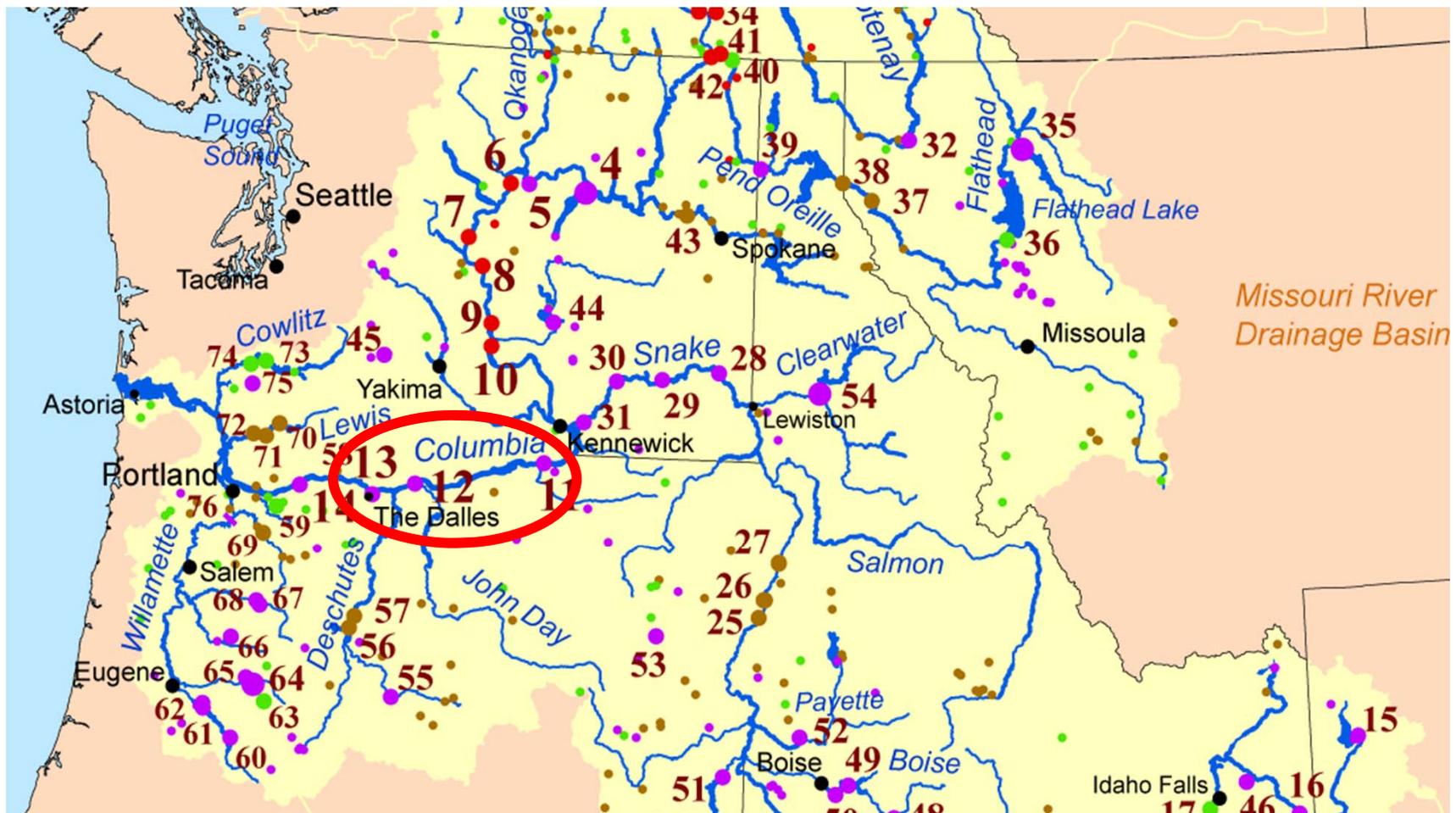
Google Data Center Groningen



Google Data Center Columbia River



The Dalles, OR, Columbia River



Google Data Center Columbia River



Energiezufuhr „Discovery Substation“ : 115 kV / 13.8 kV

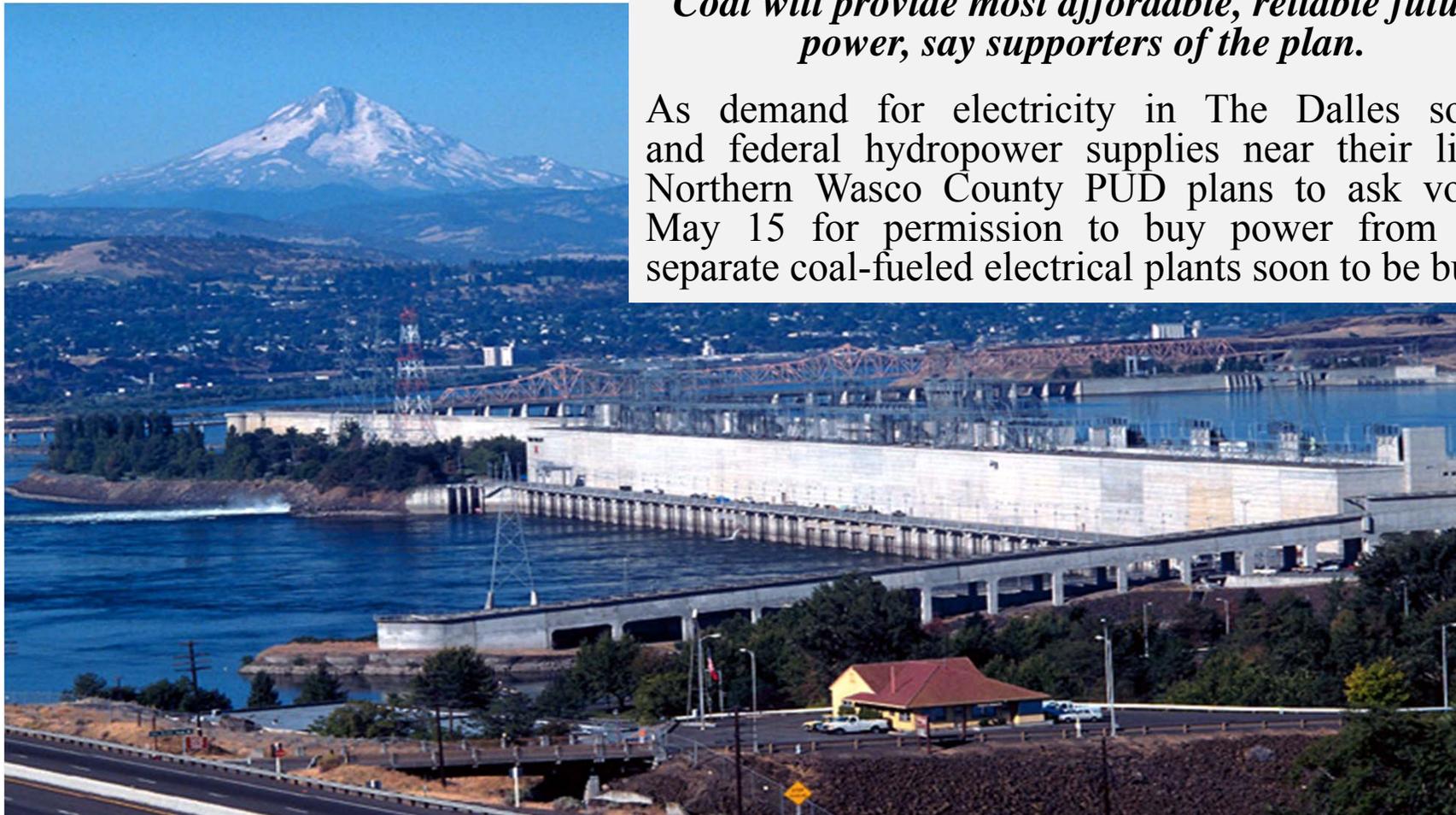


Nahes Kraftwerk: Dalles Dam Power Station, Columbia River

The Chronicle, March 1, 2007 –
PUD to seek vote on coal power

Coal will provide most affordable, reliable future power, say supporters of the plan.

As demand for electricity in The Dalles soars, and federal hydropower supplies near their limit, Northern Wasco County PUD plans to ask voters May 15 for permission to buy power from two separate coal-fueled electrical plants soon to be built.



Innenansicht eines Cloud-Zentrums

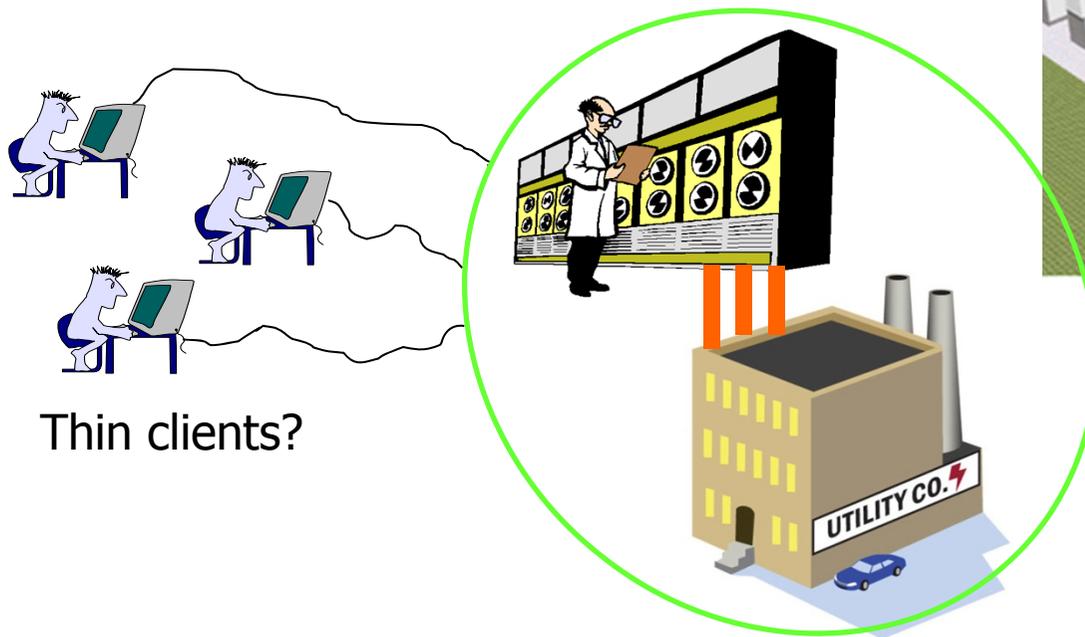


- Effizient wie **Fabriken**
 - Produkt: Internet-Dienste
- **Kostenvorteil** durch Skaleneffekt
 - Faktor 5 – 7 gegenüber traditionellen „kleinen“ Rechenzentren
- Angebot nicht benötigter Leistung auf einem **Spot-Markt**

Das entwickelt sich zu einem wesentlichen Geschäft!

Zukünftige Container-Datenzentren

- Hunderte von **Containern** aus je einigen tausend Compute-Servern
 - mit Anschlüssen für Strom und Kühlung
- Nahe an **Kraftwerken**
 - Transport von Daten billiger als Strom



Thin clients?

Cloud-Computing für die Industrie und Wirtschaft

- **Spontanes Outsourcen** von IT inklusive Geschäftsprozesse
 - Datenverarbeitung als Commodity
 - Software und Datenspeicher als Service
- Keine Bindung von Eigenkapital
 - **Kosten nach „Verbrauch“**
- **Elastizität**: Sofortiges Hinzufügen weiterer Ressourcen bei Bedarf
 - virtualisierte Hardware



Architekturen verteilter Systeme: **Zusammenfassung**



- Peer-to-Peer
- Client-Server (Fat-Client vs. Thin Client)
- 3-Tier
- Multi-Tier
- Service-Oriented Architecture (SOA)
- Compute-Cluster
- Cloud-Computing

Charakteristika und Problem- aspekte verteilter Systeme



- Räumliche Separation und Autonomie der Komponenten führen (relativ zu zentralen Systemen) zu **neuen Problemen**:
 - **partielles Fehlverhalten** möglich (statt totaler "Absturz")
 - fehlender globaler **Zustand** / exakt synchronisierte **Zeit**
 - mögliche **Inkonsistenzen** (z.B. zwischen Datei und Verzeichnis / Index)
- Typischerweise **Heterogenität** in Hard- und Software
- Hohe **Komplexität**
- **Sicherheit** (Vertraulichkeit, Authentizität, Integrität, Verfügbarkeit,...)
 - **notwendiger** als in isolierten Einzelsystemen
 - aber **schwieriger** zu gewährleisten (mehr Angriffspunkte)

Gegenmittel?



- Gute **Werkzeuge** (“Tools”) und **Methoden**
 - z.B. Frameworks und Middleware als Software-Infrastruktur
 - **Abstraktion** als Mittel zur Beherrschung von Komplexität
 - z.B. Schichten (Kapselung, virtuelle Maschinen) oder
 - Modularisierung (z.B. Services)
 - Adäquate **Modelle, Algorithmen, Konzepte**
 - zur Beherrschung der Phänomene rund um die Verteiltheit
-
- **Ziel der Vorlesung**
 - Verständnis der **grundlegenden Phänomene**
 - Kenntnis von geeigneten Konzepten und Methoden

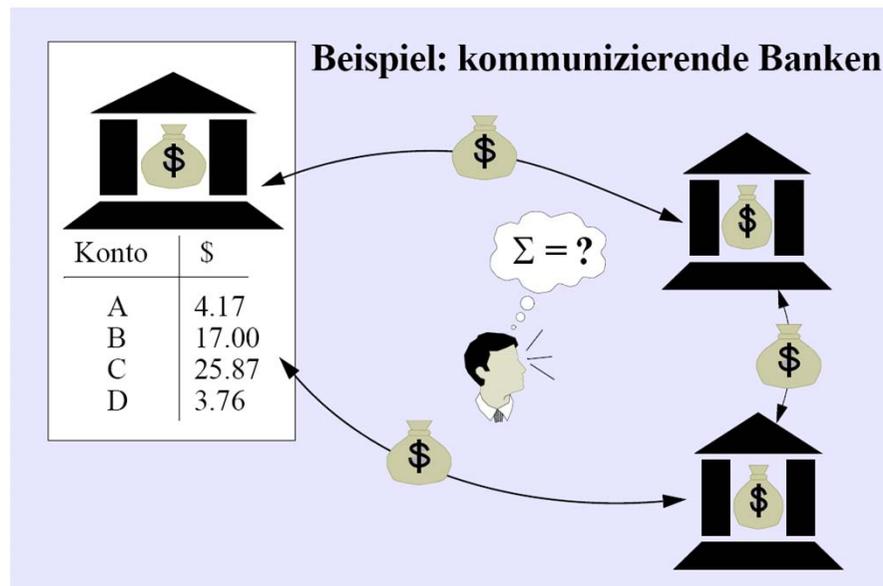
Einige konzeptionelle Probleme und Phänomene verteilter Systeme



- 1) Schnappschussproblem
- 2) Phantom-Deadlocks
- 3) Uhrensynchronisation
- 4) Kausaltreue Beobachtungen
- 5) Geheimnisvereinbarung über unsichere Kanäle

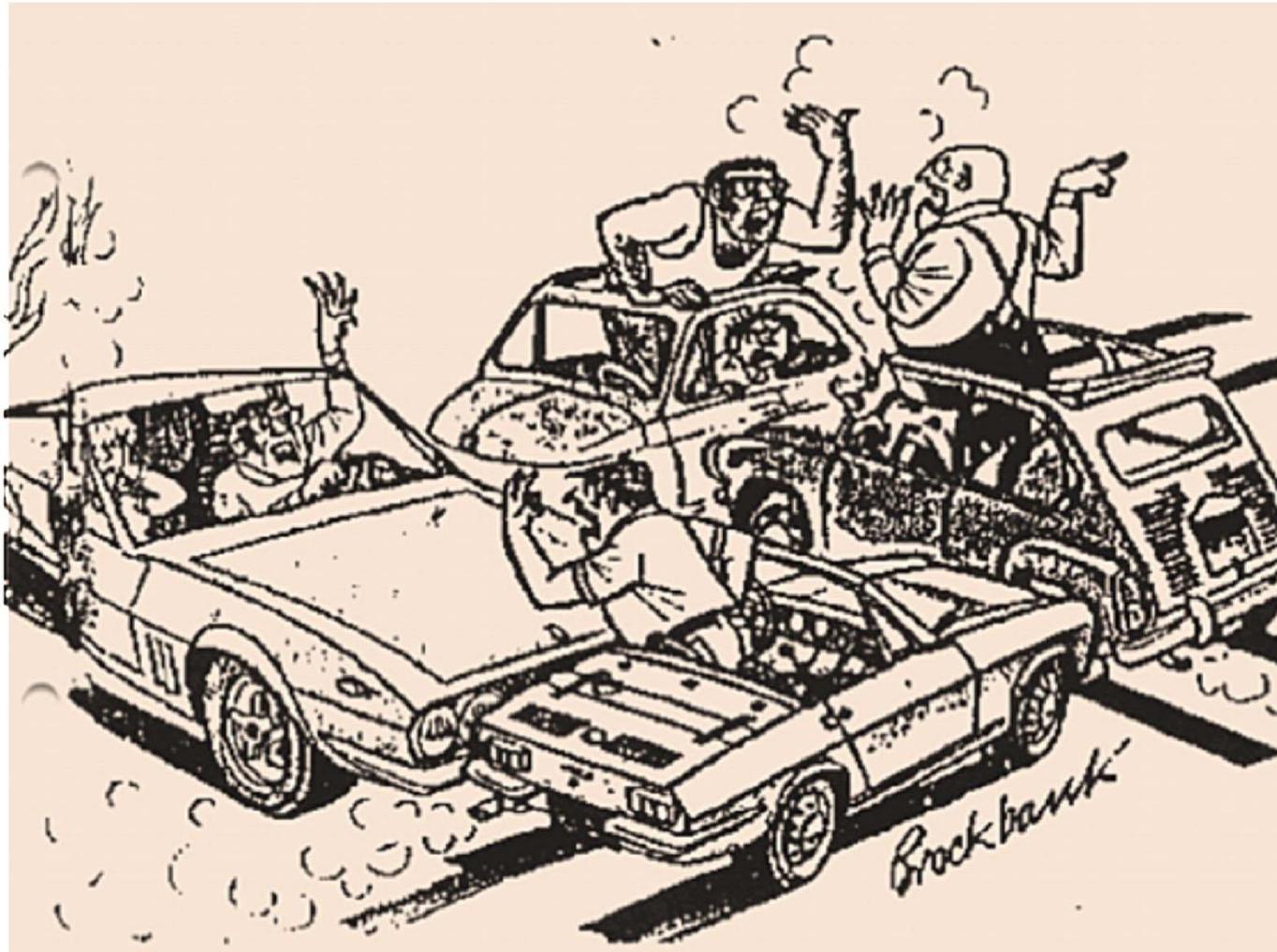
-
- Dies sind einige einfach zu erläuternde Probleme und Phänomene – natürlich gibt es noch viel mehr und viel komplexere Probleme konzeptioneller wie praktischer Art
 - Achtung: Manches davon wird nicht hier, sondern in anderen Vorlesungen (z.B. "Verteilte Algorithmen") eingehender behandelt!

Ein erstes Beispiel: Wieviel Geld ist in Umlauf?



- Hier: konstante Geldmenge
- **Ständige Transfers** zwischen den Banken
- Niemand hat eine **globale Sicht**
- Es gibt keine **gemeinsame Zeit** ("Stichtag")
- Anwendung: z.B. verteilte Datenbank-Sicherungspunkte

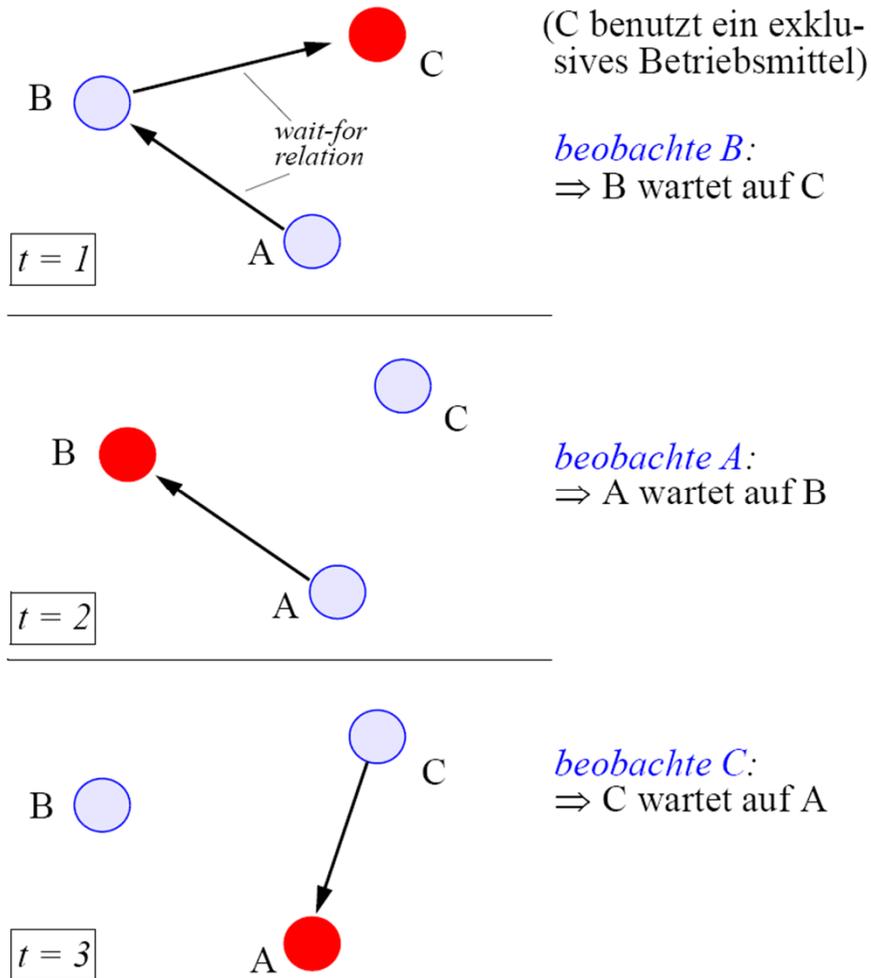
Ein zweites Beispiel: Das Deadlock-Problem



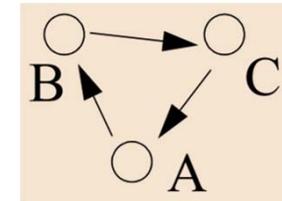
Ein zweites Beispiel: Das Deadlock-Problem



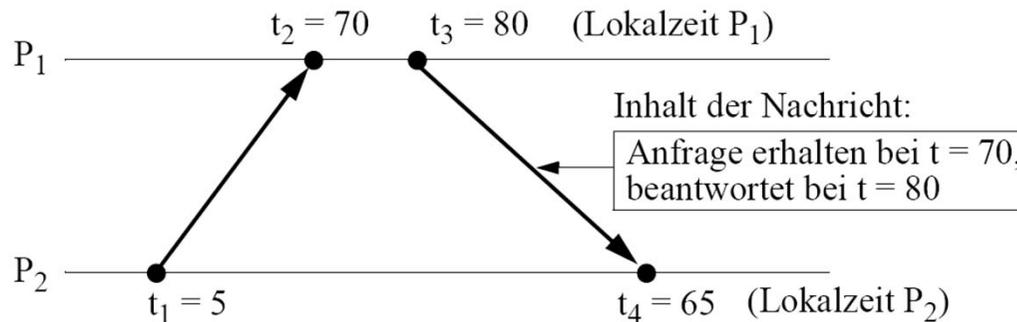
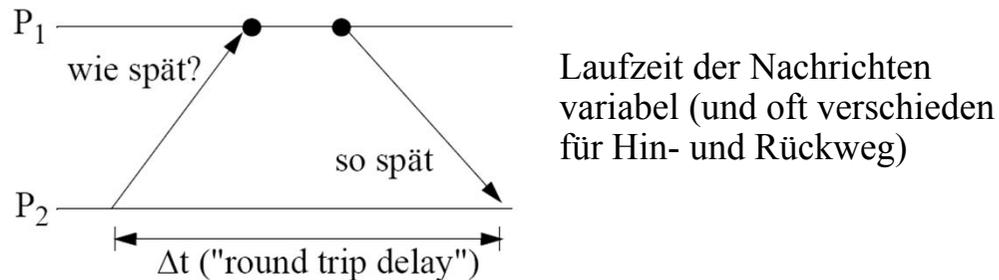
Phantom-Deadlocks



- Aus den Einzelbeobachtungen darf man **nicht** schliessen:
- A wartet auf B und B wartet auf C und C wartet auf A
- Diese **zyklische Wartebedingung** wäre tatsächlich ein Deadlock
- Die Einzelbeobachtungen fanden hier aber zu **unterschiedlichen Zeiten** statt
- **Lösung** (nur echte Deadlocks erkennen) ohne Uhren, globale Zeit, Zeitstempel etc.?



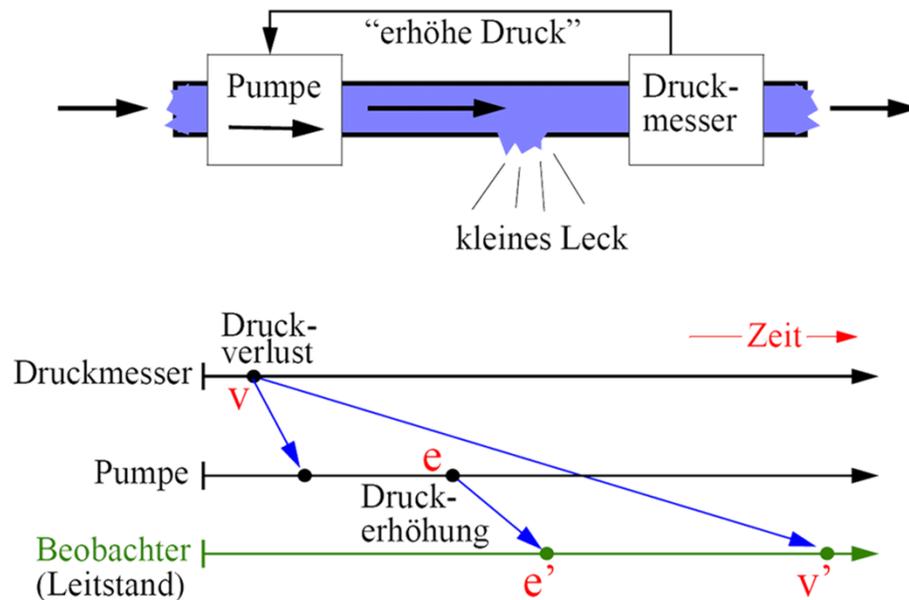
Ein drittes Problem: Uhrensynchronisation



- Uhren gehen nicht unbedingt **gleich schnell!**
 - Gilt wenigstens "Beschleunigung ≈ 0 ", d.h. ist konstanter Drift gerechtfertigt?
- Wie kann man den **Offset** der Uhren ermitteln oder zumindest approximieren?

Ein viertes Problem: (nicht) kausaltreue Beobachtungen

- Gewünscht: Eine **Ursache** stets vor ihrer (u.U. indirekten) **Wirkung** beobachten

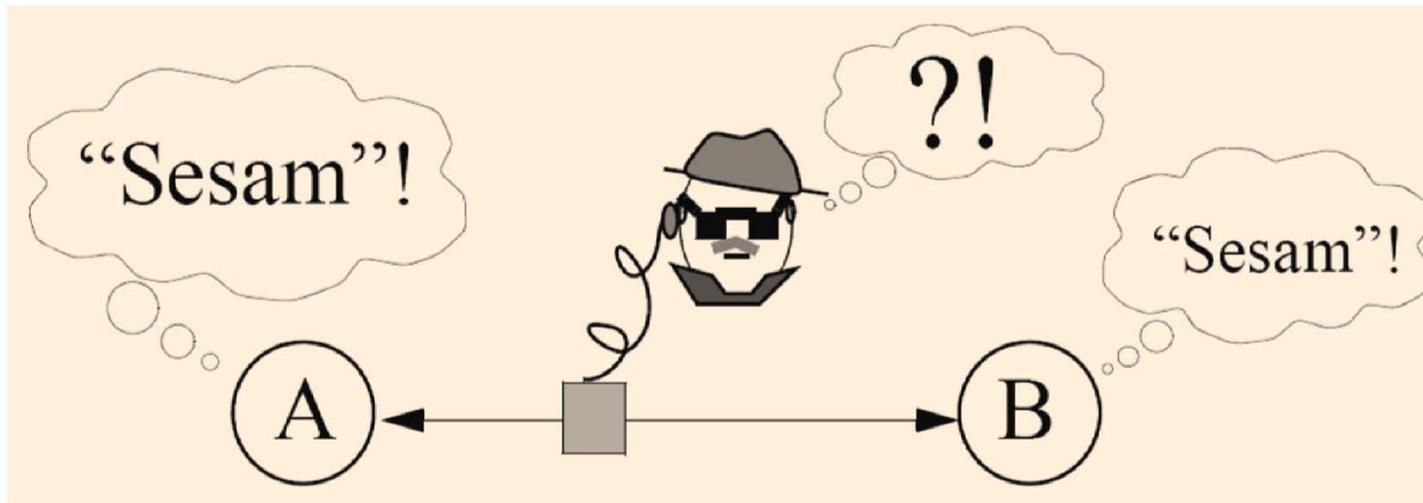


Falsche Schlussfolgerung des Beobachters:

Es erhöhte sich der Druck (aufgrund einer unbegründeten Aktivität der Pumpe), es kam zu einem Leck, was durch den abfallenden Druck angezeigt wird.

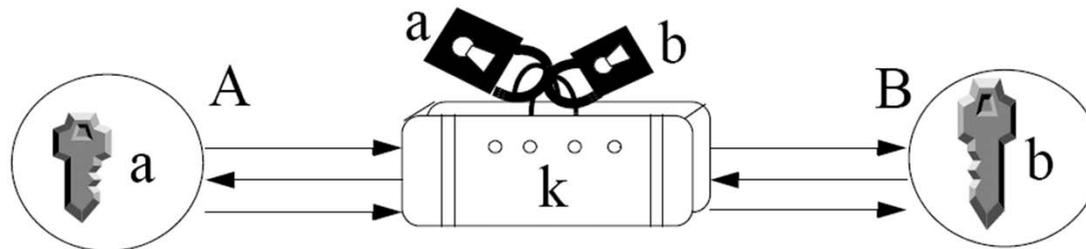
Und noch ein letztes Problem: Verteilte Geheimnisvereinbarung

- Problem: A und B wollen sich über einen unsicheren Kanal auf ein gemeinsames geheimes Passwort einigen



Verteilte Geheimnisvereinbarung (2)

- Idee: **Vorhängeschlösser** um eine sichere **Truhe**:



1. A denkt sich Passwort k aus und tut es in die Truhe.
2. A verschliesst die Truhe mit einem Schloss a .
3. A sendet die so verschlossene Truhe an B.
4. B umschliesst das ganze mit seinem Schloss b .
5. B sendet alles doppelt verschlossen an A zurück.
6. A entfernt Schloss a .
7. A sendet die mit b verschlossene Truhe wieder an B.
8. B entfernt sein Schloss b .

- Problem: Lässt sich das so **softwaretechnisch** realisieren?

Kommunikation

Kooperation durch Informationsaustausch



- Prozesse sollen **kooperieren**, daher untereinander **Information austauschen** können
 - falls vorhanden, evtl. über einen gemeinsamen **globalen Speicher** (dieser kann physisch oder evtl. nur logisch existieren als „virtual shared memory“)
 - oder mittels **Nachrichten**:
Daten an eine entfernte Stelle kopieren

Kommunikation



Notwendig, damit Kommunikation klappt, ist jedenfalls:

1. ein dazwischenliegendes **physikalisches Medium**
 - z.B. elektrische Signale in Kupferkabeln (oder der „Äther“?)
2. einheitliche **Verhaltensregeln**
 - Kommunikationsprotokolle
3. gemeinsame **Sprache** und gemeinsame **Semantik**
 - gleiches Verständnis der Bedeutung von Kommunikationskonstrukten und -regeln

Also trotz Verteiltheit gewisse **gemeinsame Aspekte!**

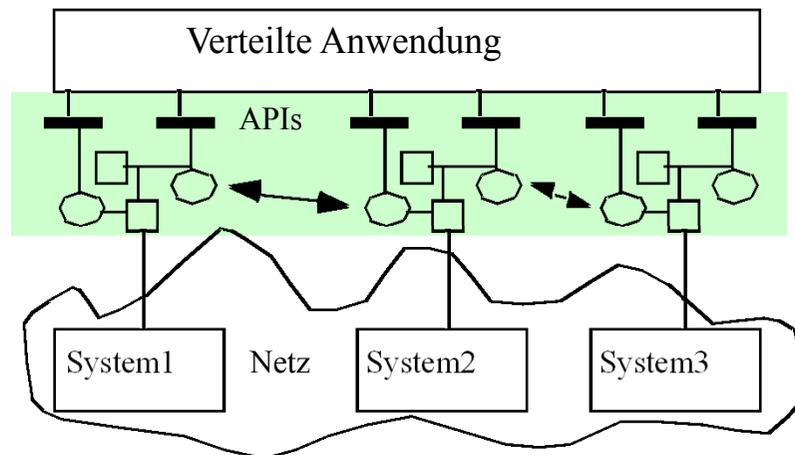
Nachrichtenbasierte Kommunikation



- **send → receive**
- Implizite **Synchronisation**: Senden vor Empfangen
 - Empfänger erfährt, wie weit der Sender mindestens ist
- Nachrichten sind **dynamische Betriebsmittel**
 - verursachen Aufwand und müssen verwaltet werden

Message Passing System (1)

- Organisiert den Nachrichtentransport
- Bietet **Kommunikationsprimitive** (als **APIs**) an
 - z.B. send (...) bzw. receive (...)
 - evtl. auch ganze **Bibliothek** verschiedener Kommunikationsdienste
 - verwendbar mit gängigen Programmiersprachen (C, Java,...)



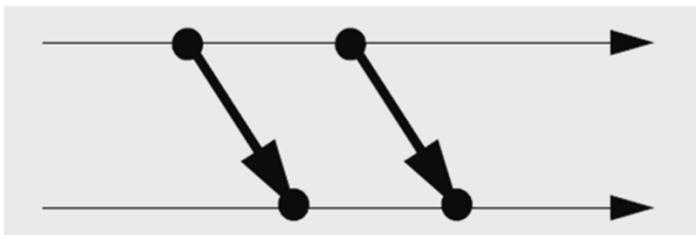
- Besteht aus Hilfsprozessen, Pufferobjekten, ...
- **Verbirgt Details** des zugrundeliegenden Netzes bzw. Kommunikationssubsystems

Message Passing System (2)

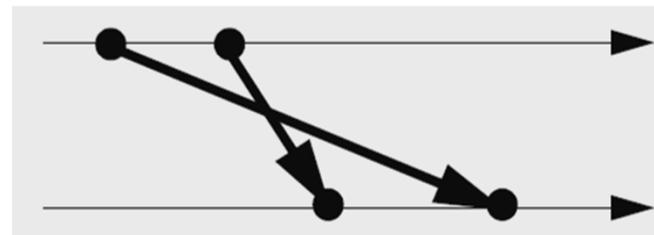
- Verwendet vorhandene Netzprotokolle und implementiert damit eigene, „höhere“ Protokolle
- **Garantiert** (je nach „Semantik“) **gewisse Eigenschaften**
 - z.B. Reihenfolgeerhalt oder Prioritäten von Nachrichten
- **Abstrahiert von Implementierungsaspekten**
 - z.B. Geräteadressen oder Längenrestriktionen von Nachrichten etc.
- **Maskiert gewisse Fehler**
 - mit typischen Techniken zur Erhöhung des Zuverlässigkeitsgrades: Timeouts, Quittungen, Sequenznummern, Wiederholungen, Prüfsummen, fehlerkorrigierende Codes,...
- **Verbirgt Heterogenität** unterschiedlicher Systemplattformen
 - erleichtert damit **Portabilität** von Anwendungen

Ordnungserhalt von Nachrichten: FIFO

- Manchmal werden vom Kommunikationssystem Garantien bzgl. **Nachrichtenreihenfolgen** gegeben
- Eine solche Garantie stellt z.B. **FIFO** (First-In-First-Out) dar: Nachrichten zwischen zwei Prozessen (also auf dem Kommunikationskanal zwischen Sender und Empfänger) überholen sich nicht: **Empfangsreihenfolge = Sendereihenfolge**



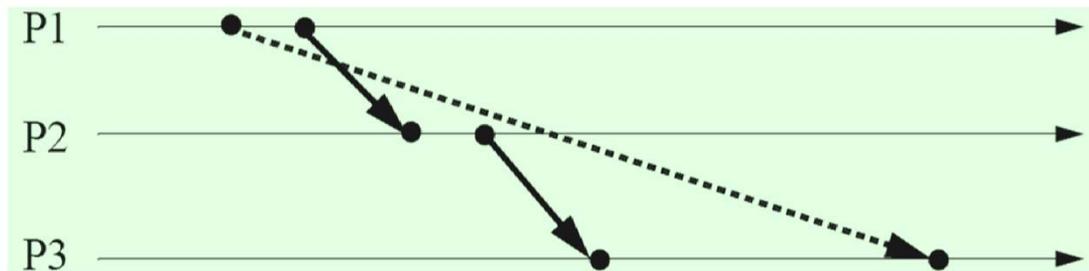
FIFO



kein FIFO

Ordnungserhalt von Nachrichten: kausale Ordnung

- FIFO verbietet allerdings nicht, dass Nachrichten evtl. (über eine Kette anderer Nachrichten) **indirekt überholt** werden



Zwar FIFO, aber nicht kausal geordnet

- Möchte man auch dies haben, so muss die Kommunikation **kausal geordnet** sein (Anwendungszweck?)
 - keine Information erreicht Empfänger **auf Umwegen schneller** als auf direktem Wege („Dreiecksungleichung“)
 - entspricht einer „**Globalisierung**“ von FIFO auf mehrere Prozesse
 - **Denkübung**: Wie garantiert (d.h. implementiert) man kausale Ordnung auf einem System ohne Ordnungsgarantie?

Prioritäten von Nachrichten? (1)



- Achtung: **Semantik** ist a priori nicht ganz klar:
 - Soll (kann?) das Transportsystem Nachrichten höherer Priorität bevorzugt (=?) befördern?
 - Können (z.B. bei fehlender Pufferkapazität) Nachrichten niedrigerer Priorität überschrieben werden?
 - Wie viele Prioritätsstufen gibt es?
 - Sollen aus einer Mailbox (= Nachrichtenspeicher) immer zuerst Nachrichten mit höherer Priorität geholt werden?

Prioritäten von Nachrichten? (2)

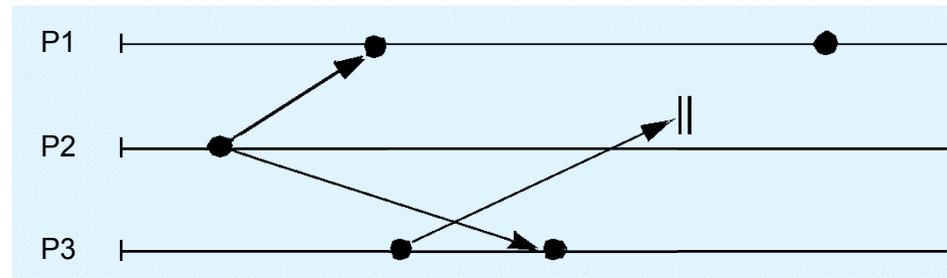
- Mögliche **Anwendungen**:
 - Unterbrechen / abbrechen laufender Aktionen (→ Interrupt)
 - Aufbrechen von Blockaden
 - Out-of-Band-Signalisierung } Durchbrechung der FIFO-Reihenfolge!

Vgl. auch Service-Klassen in **Computernetzen**: bei Rückstaus bei den Routern soll z.B. interaktiver Verkehr bevorzugt werden vor FTP etc.

- **Vorsicht** bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!
 - Inwiefern ist denn eine (faule) Implementierung, bei der „eilige“ Nachrichten (insgeheim) wie normale Nachrichten realisiert werden, tatsächlich „Betrug“ am Anwender?

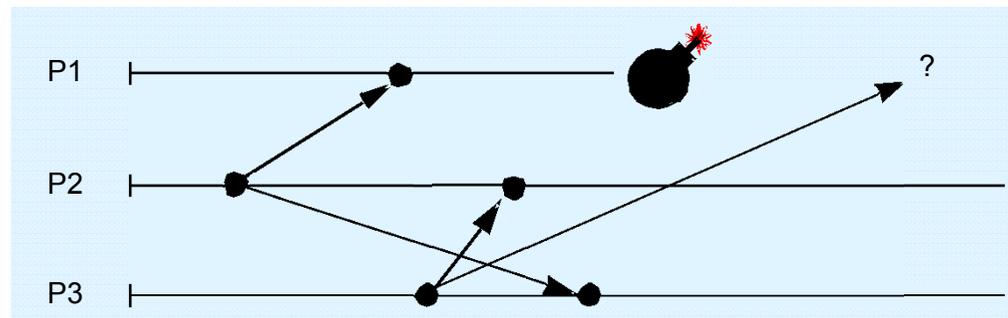
Fehlermodelle (1)

- Zweck: Klassifikation von Fehlermöglichkeiten;
Abstraktion von den tieferliegenden spezifischen Ursachen
- **Nachrichtenfehler** beim Senden / Übertragen / Empfangen:



→ verlorene
Nachricht

- **Crash / Fail-Stop**: Ausfall eines Prozessors:



→ Nicht mehr
erreichbarer /
mitspielender
Prozess

Fehlermodelle (2)

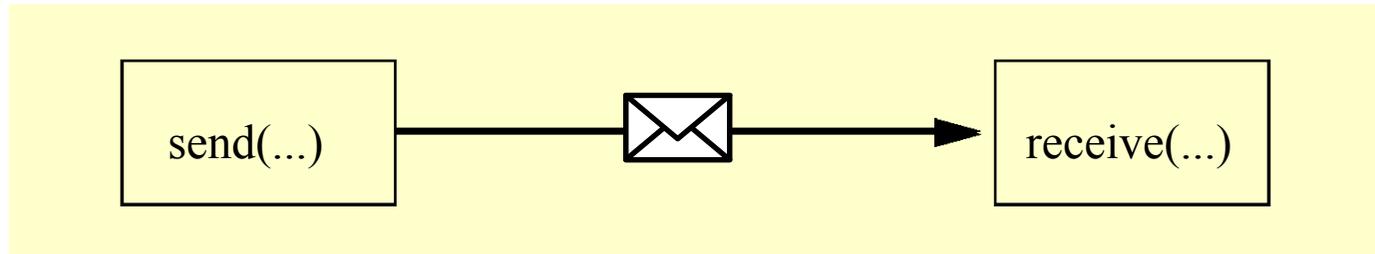


- **Zeitfehler**: Ereignis geschieht zu spät (oder zu früh)
- **„Byzantinische“ Fehler**: Beliebiges Fehlverhalten, z.B.:
 - verfälschte Nachrichteninhalte
 - Prozess, der unsinnige Nachrichten sendet

(solche Fehler lassen sich nur teilweise, z.B. durch **Redundanz**, erkennen)

-
- **Fehlertolerante** Algorithmen bzw. Systeme müssen das „richtige“ Fehlermodell berücksichtigen!
 - adäquate Modellierung der realen Situation / des Einsatzgebietes
 - Algorithmus verhält sich **korrekt nur relativ zum Fehlermodell**

Mitteilungsorientierte Kommunikation

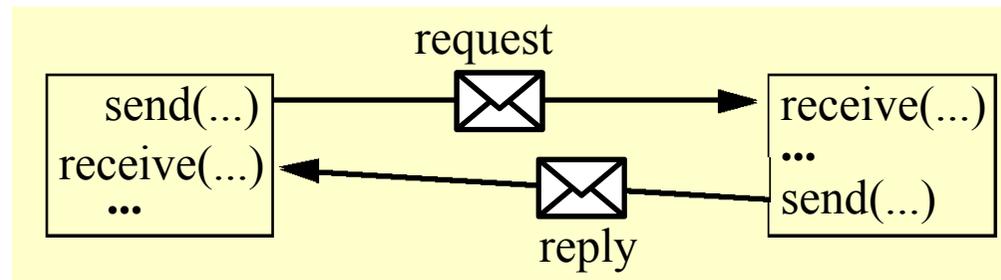


- Einfachste Form der Kommunikation
 - **Unidirektional**
 - Übermittelte Werte werden der Nachricht typw. als „Ausgabeparameter“ beim send (-API) übergeben
-
- Die **Nachricht** ist typischerweise eine Zeit lang **unterwegs**, bevor sie beim Empfängerprozess ankommt
 - **Sendprozess** kann nach dem Absenden aber i. Allg. direkt **weiterarbeiten**

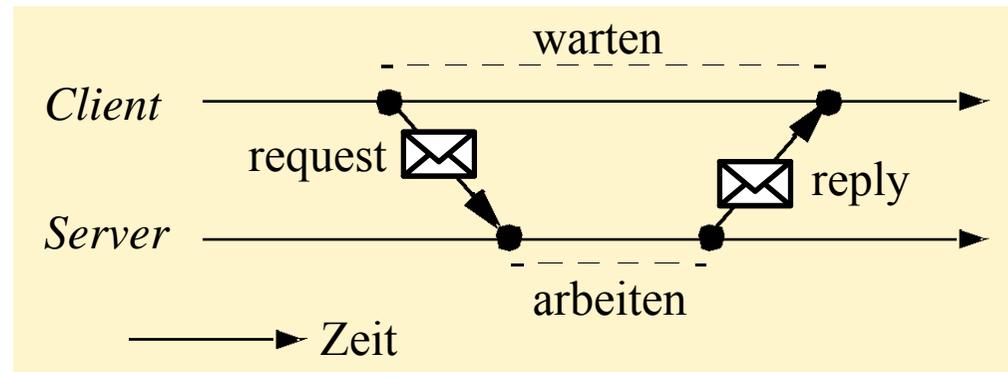
Asynchrone Kommunikation

Auftragsorientierte Kommunikation

Send und receive evtl. zu einem *einzig*en API zusammenfassen



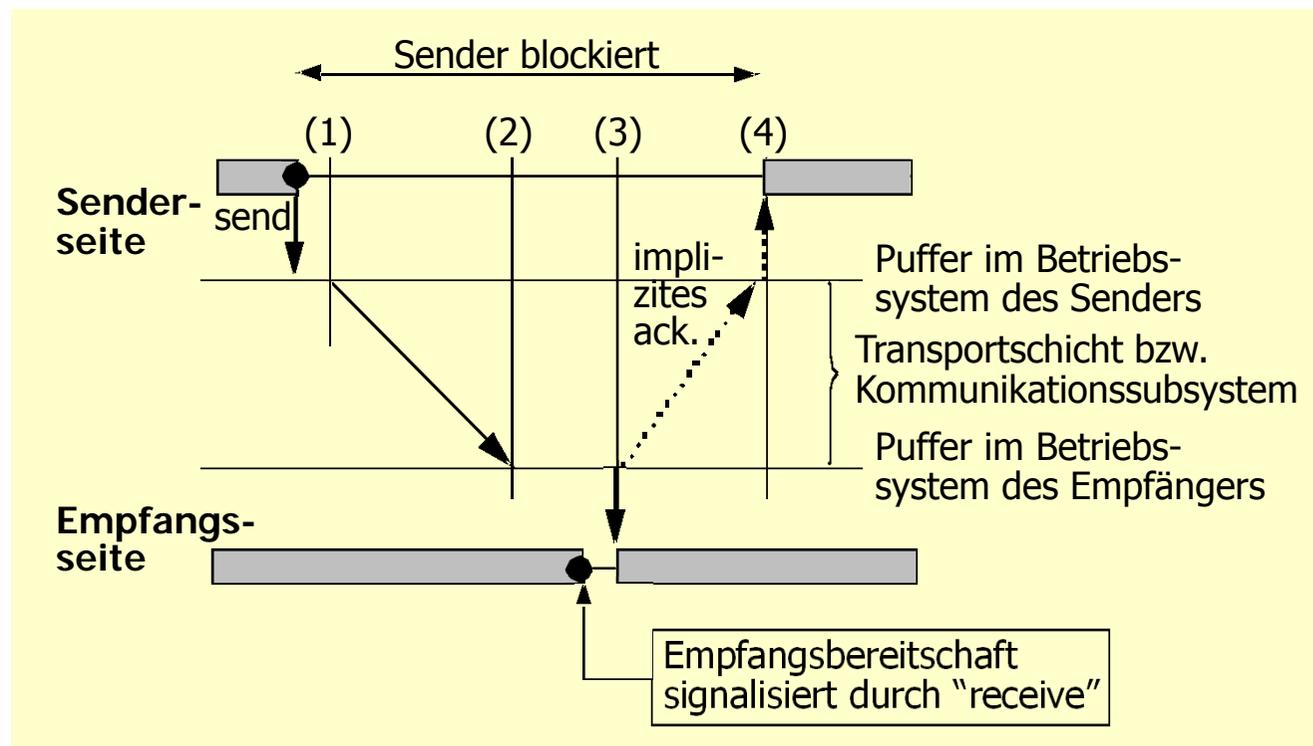
- Bidirektional
- Ergebnis des Auftrags wird als „Antwortnachricht“ zurückgeschickt
- Auftraggeber („Client“) wartet, bis Antwort eintrifft



Blockierendes Senden

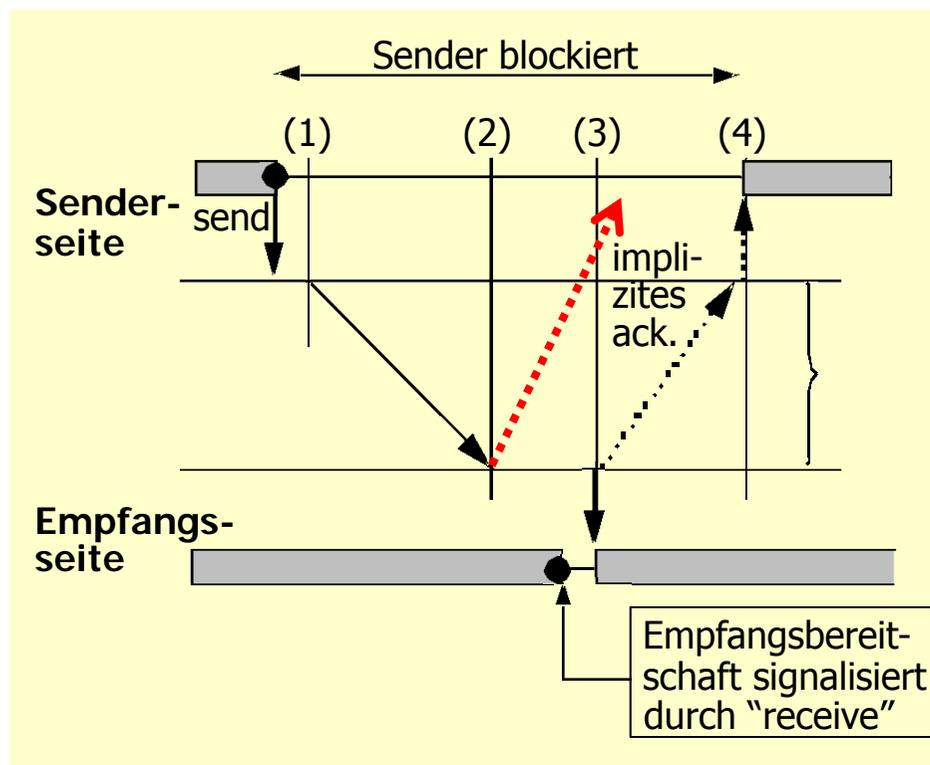
- **Blocking send**: Sender ist bis zum Abschluss der Nachrichtentransaktion blockiert
- Sender hat eine **Garantie** (Nachricht wurde zugestellt / empfangen)

was genau ist das?



Blockierendes Senden (2)

- Verschiedene Ansichten einer adäquaten Definition von „Abschluss der Transaktion“ aus Sendersicht:



Zeitpunkt 4 (automatische Bestätigung bei (3), dass der Empfänger receive ausgeführt hat) ist die höhere, anwendungsorientierte Sicht.

Falls eine Bestätigung bereits zum **Zeitpunkt 2** geschickt wird, weiss der Sender nur, dass die Nachricht am Zielort zur Verfügung steht und der Sendepuffer wieder frei ist.

(Vorher sollte der Sendepuffer nicht überschrieben werden, weil die Nachricht bei fehlerhafter Übertragung evtl. wiederholt werden muss.)

„gleich“ „zeitig“

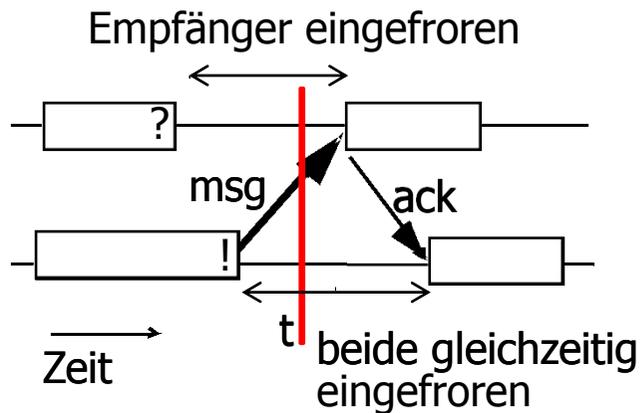
Synchrone Kommunikation



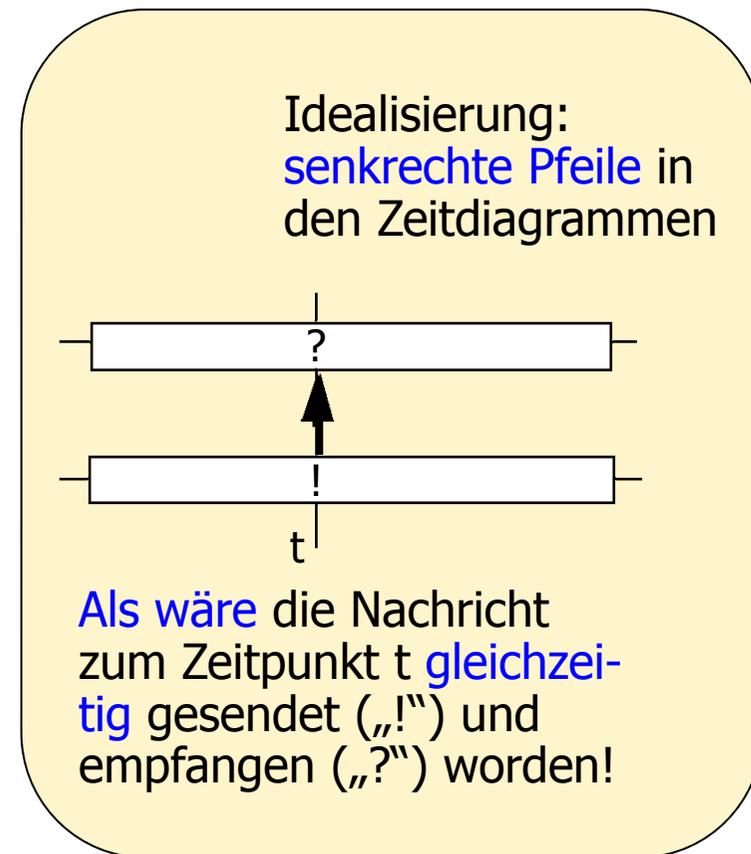
- **Idealisierung:**
Send und receive geschehen („im Prinzip“) **gleichzeitig**
- Wodurch ist diese Idealisierung gerechtfertigt?
(kann man auch mit einer Marssonde synchron kommunizieren?)

Synchrone Kommunikation mit „blocking send“ implementiert

a) „Receiver first“-Fall:

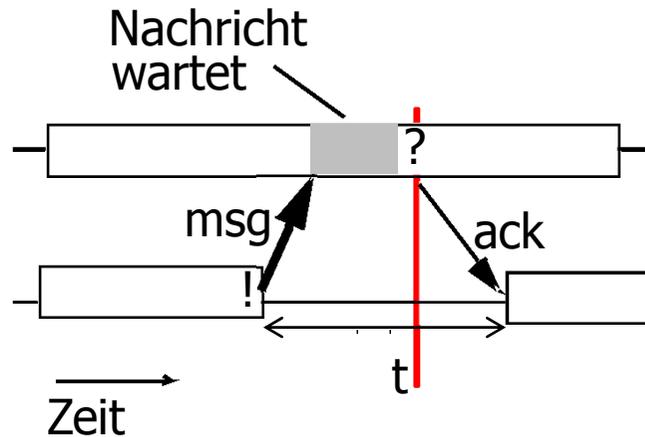


Bemerkung: „receive“ ist i.Allg. immer **blockierend** (d.h. Empfänger wartet so lange, bis eine Nachricht ankommt)

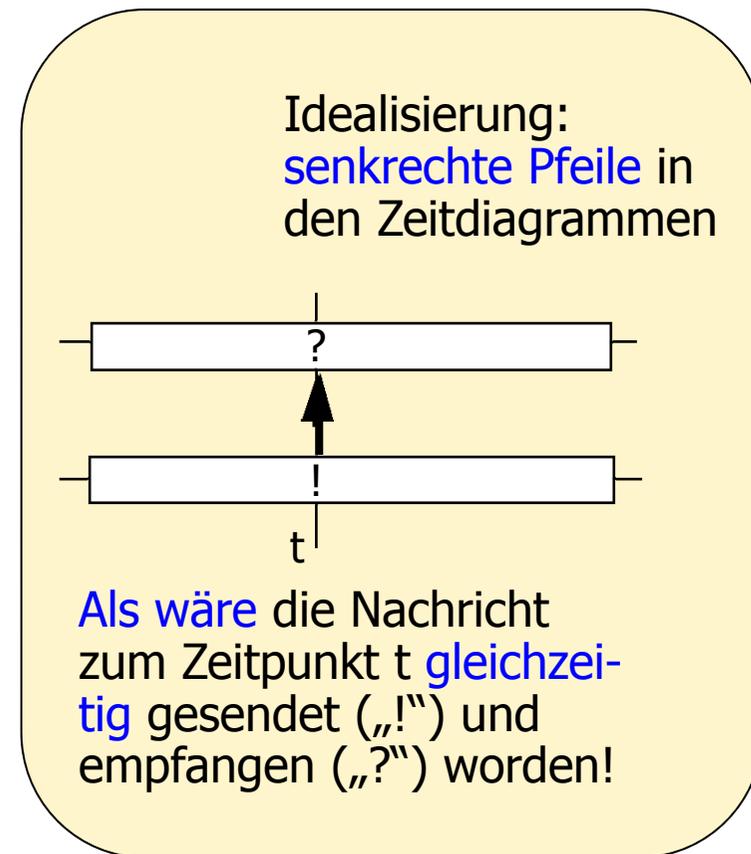


Synchrone Kommunikation mit „blocking send“ implementiert (2)

b) „Sender first“-Fall:

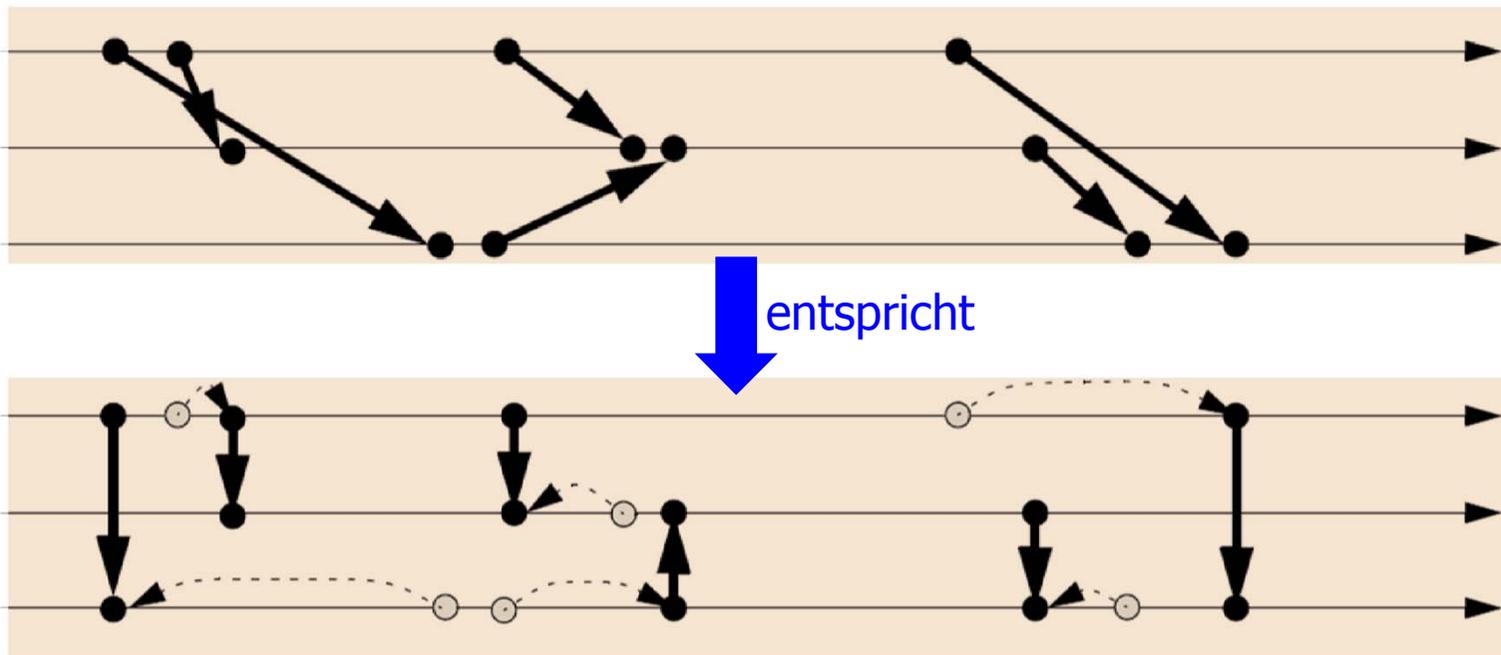


Der Sender ist eingefroren, seine Zeit steht quasi still → es gibt einen **gemeinsamen Zeitpunkt t**, wo die beiden Kommunikationspartner sich treffen → „Rendezvous“



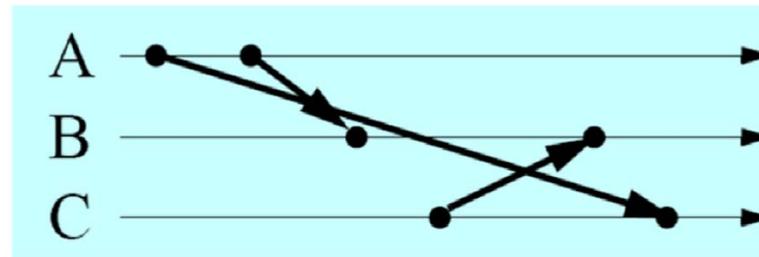
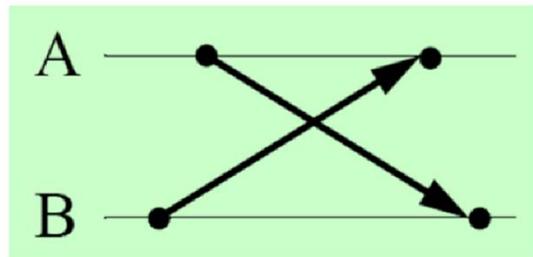
Virtuelle Gleichzeitigkeit

- Ein Ablauf, der synchrone Kommunikation benutzt, ist (bei Abstraktion von der Realzeit) durch ein **äquivalentes Zeitdiagramm** darstellbar, bei dem alle **Nachrichtenpfeile senkrecht** verlaufen
 - nur stetige Deformation erlaubt („**Gummiband-Transformation**“)



Virtuelle Gleichzeitigkeit?

- Folgendes geht **nicht virtuell gleichzeitig** (wieso?)



- Aber was geschieht eigentlich, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies **provoziert** wird?

Mehr dazu (nur) für besonders Interessierte: B. Charron-Bost, F. Mattern, G. Tel: *Synchronous, Asynchronous and Causally Ordered Communication*. Distributed Computing 9(4), pp. 173-191, www.vs.inf.ethz.ch/publ/

Deadlocks bei synchroner Kommunikation

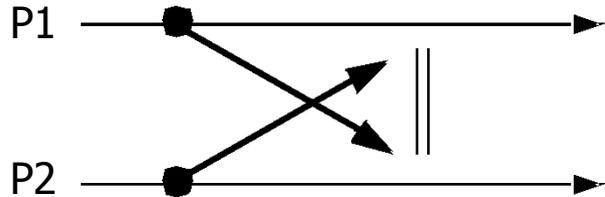
P1:

send (...) to P2;
receive...
...

P2:

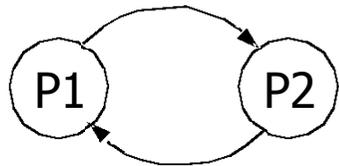
send (...) to P1;
receive...
...

In beiden Prozessen muss zunächst das **send** ganz ausgeführt werden, bevor es zu einem **receive** kommt



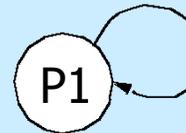
⇒ **Kommunikationsdeadlock!**

Zyklische Abhängigkeit der Prozesse voneinander: P1 wartet auf P2, und P2 wartet auf P1



„Wait-for-Graph“

Genauso tödlich:



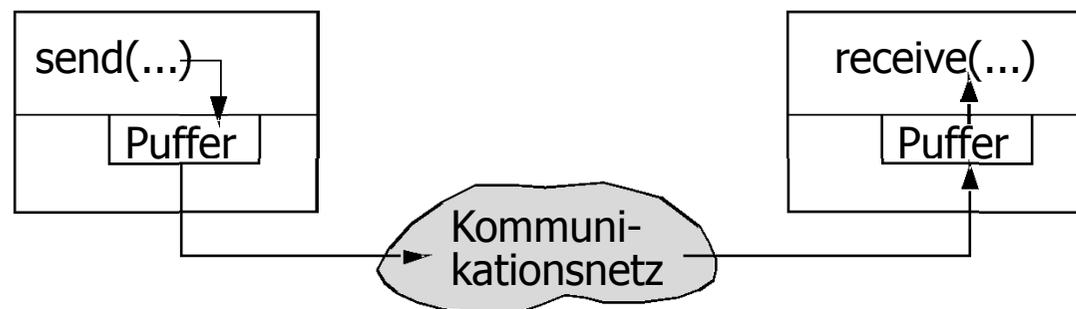
P1:

send (...) to P1;
receive...
...

Gleichnishaft entspricht der *synchronen* Kommunikation das Telefonieren, der *asynchronen* Kommunikation der Briefwechsel

Asynchrone Kommunikation

- **No-wait send**: Sender ist nur (kurz) bis zur lokalen Ab-
lieferung der Nachricht an das Transportsystem blockiert
 - diese kurzzeitige Blockade sollten für die Anwendung transparent sein
- Jedoch i.Allg. länger blockiert, falls das System mo-
mentan keinen **Pufferplatz** für die Nachricht frei hat
 - Alternative: Sendenden Prozess dann nicht länger blockieren,
sondern mittels „return value“ über Misserfolg des send informieren



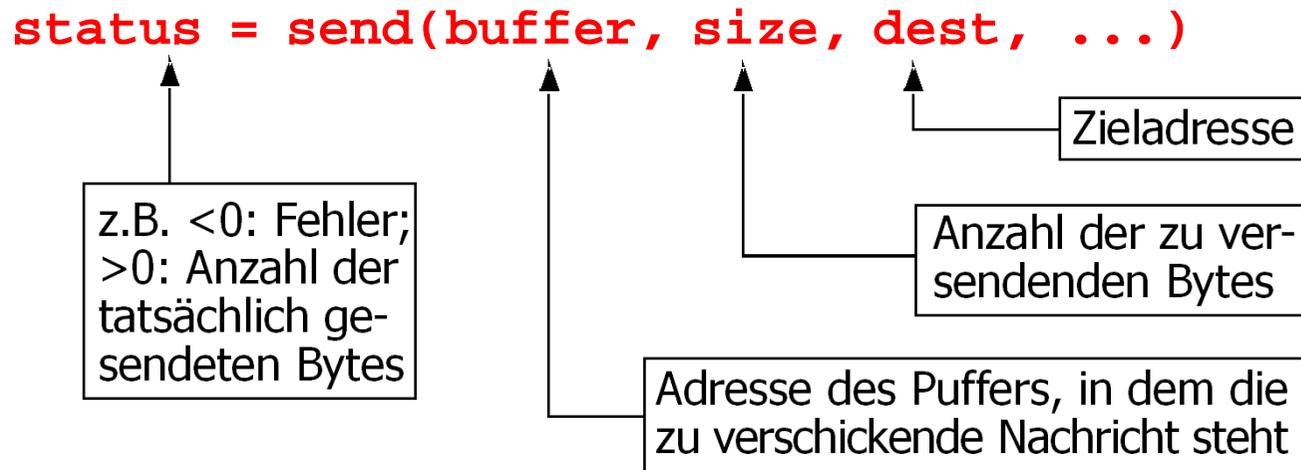
Asynchrone ↔ synchrone Kommunikation



- **Vorteile asynchroner Kommunikation** (im Vgl. zur syn. Komm.):
 - sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird
 - stärkere Entkoppelung von Sender / Empfänger
 - höherer Grad an Parallelität möglich
 - geringere Gefahr von Kommunikationsdeadlocks
- **Nachteile**
 - Sender weiss nicht, ob / wann Nachricht angekommen ist
 - Debugging der Anwendung oft schwierig (wieso?)
 - System muss Puffer verwalten

Sendeoperationen in der Praxis

- Es gibt Kommunikationsbibliotheken, deren Dienste von verschiedenen Programmiersprachen (z.B. C) aus aufgerufen werden können
 - z.B. **MPI (Message Passing Interface)** { Quasi-Standard; verfügbar auf diversen Plattformen
- Typischer Aufruf einer solchen Send-Operation:



Sendeoperationen in der Praxis (2)



- Derartige Systeme bieten im Allgemeinen mehrere **verschiedene Typen von Send-Operation** an
 - Zweck: Hohe **Effizienz** durch möglichst spezifische Operationen
 - **Achtung**: Spezifische Operation kann in anderen Situationen u.U. eine falsche oder unbeabsichtigte Wirkung haben (z.B. wenn vorausgesetzt wird, dass der Empfänger schon im receive wartet)
 - Vorsicht: Semantik und Kontext der Anwendbarkeit ist oft nur informell beschrieben

Synchron [?] = blockierend

- Kommunikationsbibliotheken machen oft einen Unterschied zwischen **synchronem** und **blockierendem** Senden
 - bzw. analog zwischen asynchron und nicht-blockierend
- **Blockierung** ist dann ein rein **senderseitiger** Aspekt
 - **blockierend**: Sender wartet, bis die Nachricht lokal vom Kommunikationssystem abgenommen wurde (und der Puffer wieder frei ist)
 - **nicht-blockierend**: Sender informiert Kommunikationssystem lediglich, wo bzw. dass es eine zu versendende Nachricht gibt (Gefahr des Überschreibens des Puffers bevor dieser frei ist!)
- **Synchron / asynchron** nimmt Bezug auf den **Empfänger**
 - **synchron**: Nach Ende der Send-Operation wurde die Nachricht dem Empfänger zugestellt (**asynchron**: dies ist nicht garantiert)

leider etwas verwirrend!

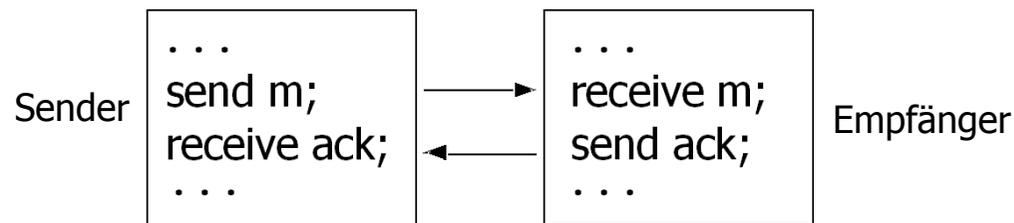
Nicht-blockierend



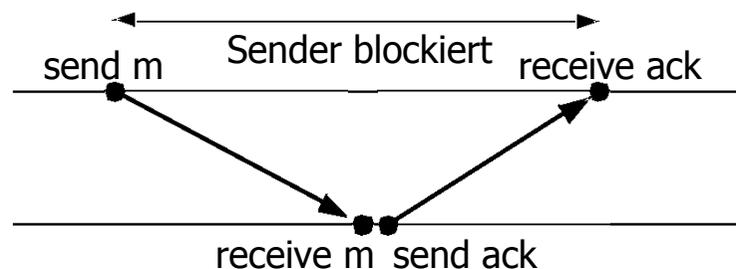
- Nicht-blockierende Operationen liefern oft einen „**handle**“
 - handle = send(...)**
 - dieser kann in Test- bzw. Warteoperationen verwendet werden
 - z.B. **Test**, ob Send-Operation beendet:
`if msgdone(handle)...`
 - oder z.B. **warten** auf Beendigung der Send-Operation:
`msgwait(handle)`
- Nicht-blockierend ist oft **effizienter**, aber evtl. **unsicherer** und **umständlicher** (evtl. `Test`; `warten`) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:



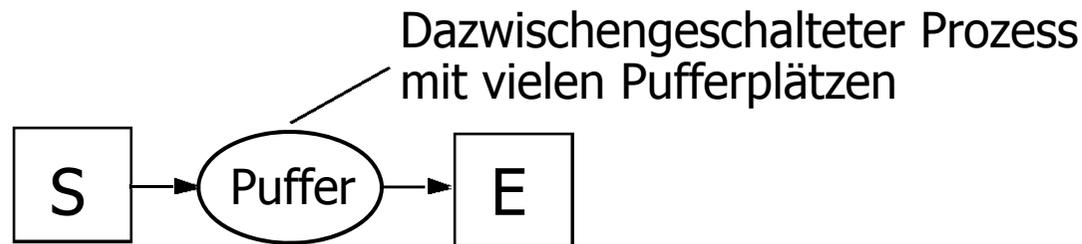
- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive



Dualität der Kommunikationsmodelle (2)

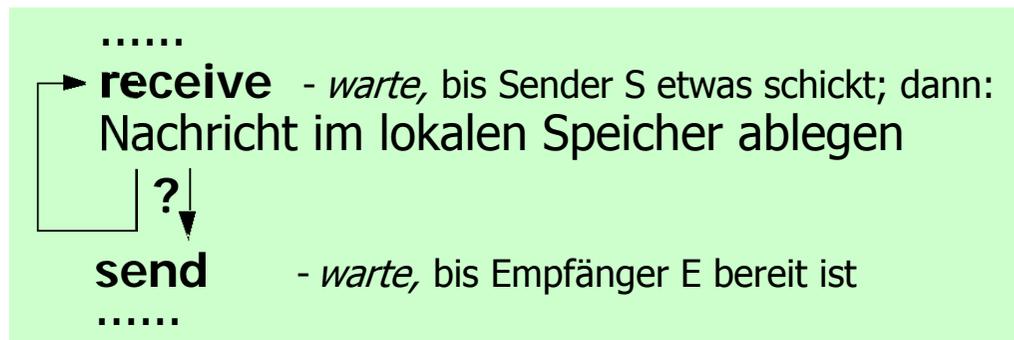
Asynchrone Kommunikation mittels synchroner:

Idee: Zusätzlichen Prozess vorsehen, der für die **Zwischenpufferung** aller Nachrichten sorgt



→ **Entkoppelung** von Sender und Empfänger

Wie realisiert man einen Pufferprozess bei syn. Kommunikation?

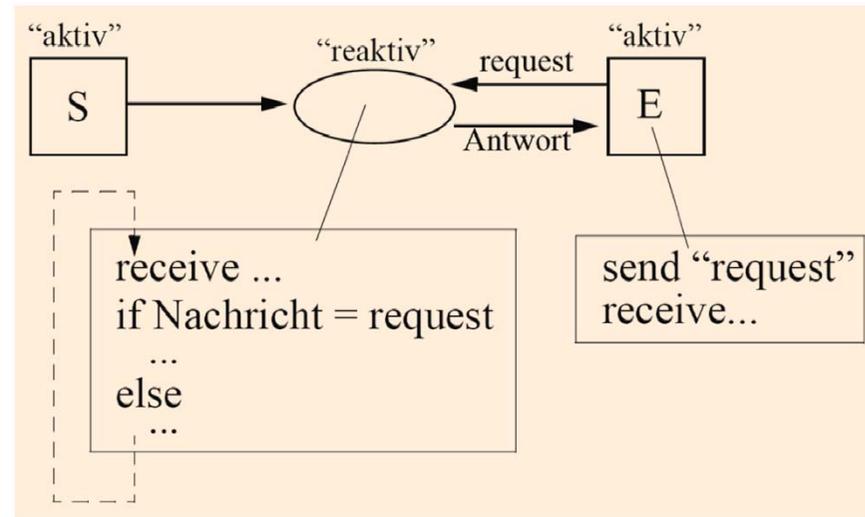


- **Dilemma:** Was tut der Pufferprozess nach dem Ablegen der Nachricht im lokalen Speicher?
 - wieder im receive auf den Sender warten, *oder*
 - in einem (blocking) send auf den Empfänger warten?
- Entweder Sender S oder Empfänger E könnte unnötigerweise **blockiert** sein!

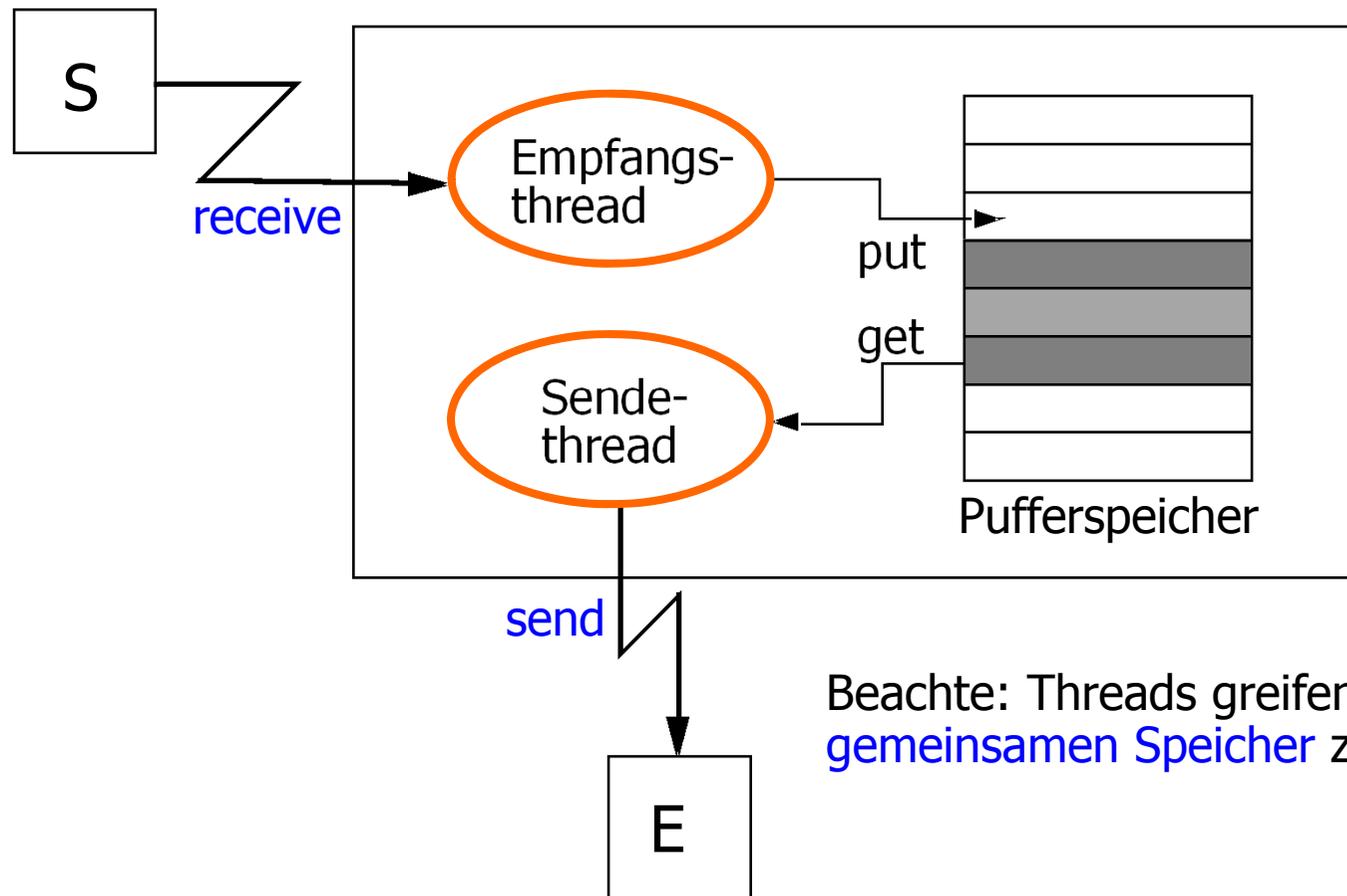
Bemerkung: **Puffer der Größe 1** lassen sich so realisieren → Kaskadierung im Prinzip möglich („Pufferpipeline“)

1. Lösung: Puffer als Server!

- E schickt „seinem“ Puffer einen „request“ (und muss dazu die Adresse des Puffers kennen)
- Puffer schickt E keine Antwort, wenn er leer ist
- Empfänger E wird nur dann verzögert, wenn Puffer leer
- Für Sender S ändert sich nichts
- Was tun bei vollem Puffer?
→ Dann sollte der Puffer keine Nachricht von S (wohl aber von E!) annehmen (Denkübung: wie programmiert man das?)



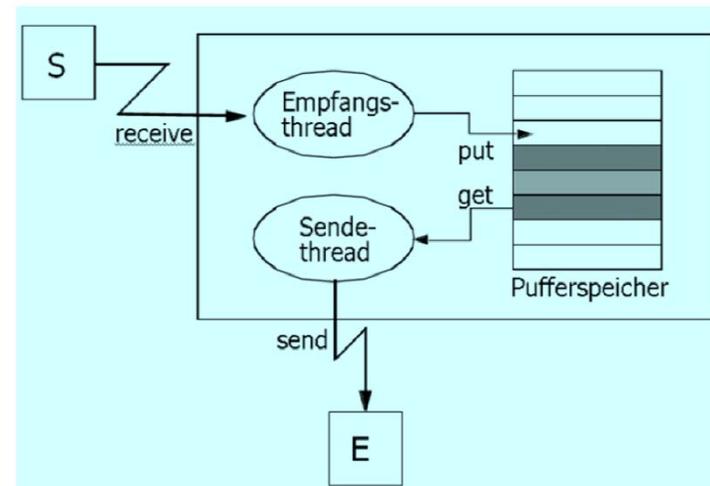
2. Lösung: Puffer als Multithread-Objekt



Beachte: Threads greifen auf
gemeinsamen Speicher zu

Puffer als Multithread-Objekt (2)

- **Empfangsthread** ist (fast) immer empfangsbereit
 - nur kurzzeitig anderweitig beschäftigt (put in lokalen Pufferspeicher)
 - evtl. nicht empfangsbereit, wenn lokaler Pufferspeicher voll
- **Sendethread** ist (fast) immer sendebereit
- Pufferspeicher ist i.Allg. zyklisch organisiert (→ FIFO)
- Pufferspeicher liegt im gemeinsamen Adressraum
- ⇒ **Synchronisation** der beiden Threads notwendig!
 - konkurrentes / paralleles Programmieren
 - klassische Themen der Betriebssystemtheorie



Hauptklassifikation von Kommunikationsmechanismen

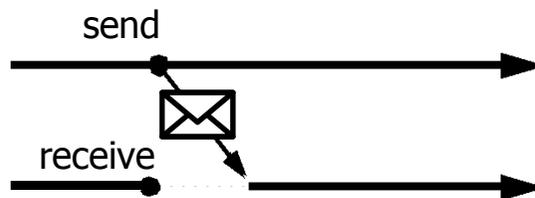
"orthogonal" → *Synchronisationsgrad*

		<i>Synchronisationsgrad</i>	
		asynchron	synchron
<i>Kommunikationsmuster</i>	Mitteilung	<i>no-wait send</i>	<i>Rendezvous</i>
	Auftrag	"asynchroner RPC"	<i>Remote Procedure Call (RPC)</i>

- Häufigste Kombination:
Mitteilung asynchron, Auftrag hingegen synchron
- Wir schauen uns nun der Reihe nach die 4 Kombinationen an

No-Wait Send

- **Asynchron-mitteilungsorientierte** Kommunikation



- **Vorteile**

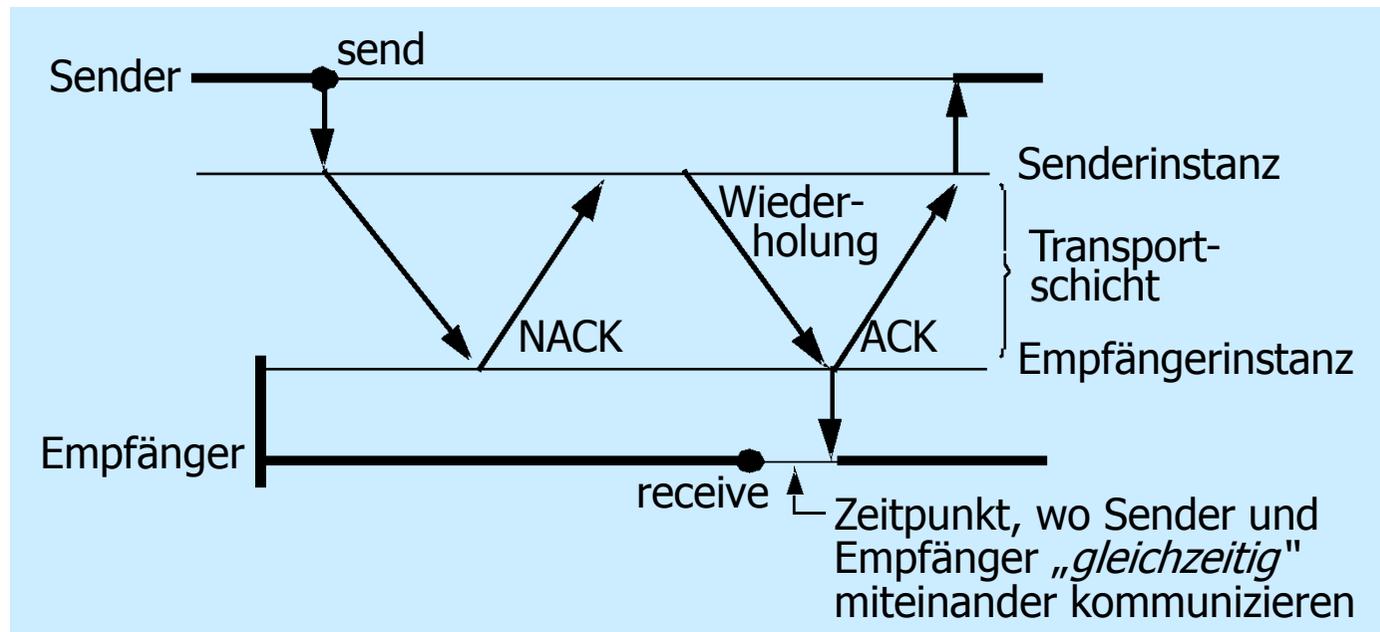
- weitgehende zeitliche Entkopplung von Sender und Empfänger
- einfache, effiziente Implementierung (bei kurzen Nachrichten)

- **Nachteile**

- keine Erfolgsgarantie für den Sender
- Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicher-
verwaltung,...) im Unterschied etwa zur synchronen Kommunikation
- Gefahr des „Überrennens“ des Empfängers bei zu häufigen
Nachrichten → Flusssteuerung („flow control“) notwendig

Rendezvous-Protokolle

- Synchron-mitteilungsorientierte Kommunikation



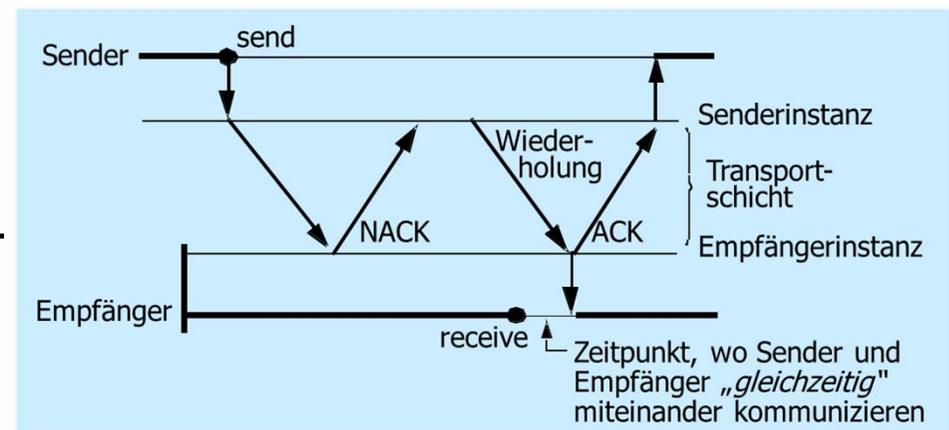
- Hier beispielhaft „**Sender-first**-Szenario“:
Sender wartet als Erster („Receiver-first-Szenario“ analog)

Rendezvous-Protokolle (2)

- **Rendezvous**: Der erste wartet auf den anderen („Synchronisationspunkt“)
- Mit **NACK / ACK** (vgl. Bild) sind wenig Puffer nötig → aber aufwändiges Protokoll („busy waiting“)
 - Alternative 1: Statt NACK („negative ACK“): Nachricht auf Empfängerseite puffern (Nachteil: Platzbedarf für Puffer)
 - Alternative 2: Statt laufendem Wiederholungsversuch: Empfängerinstanz meldet sich bei Senderinstanz, sobald Empfänger bereit



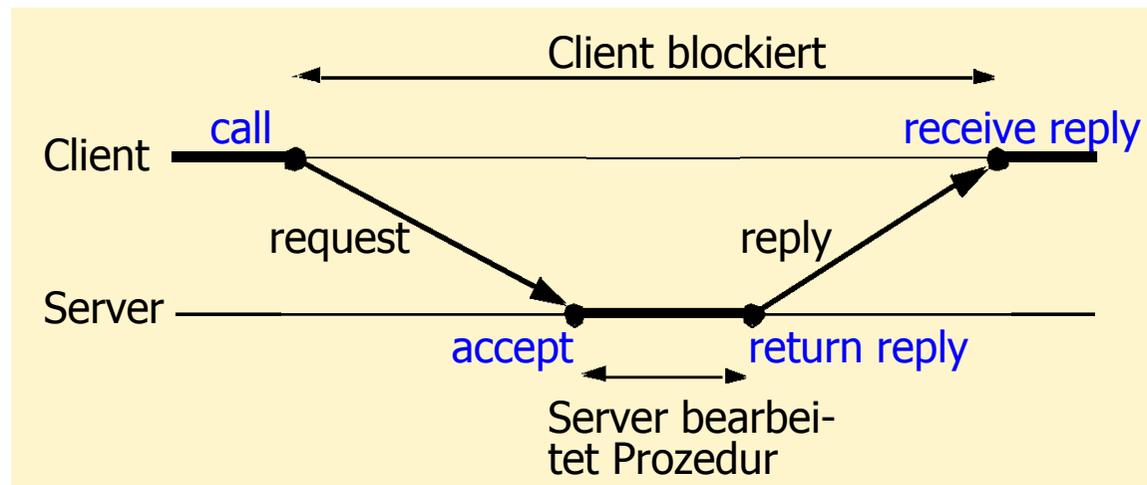
- Insbesondere bei langen Nachrichten sinnvoll: **Vorherige Anfrage**, ob bei der Empfängerinstanz genügend Pufferplatz vorhanden ist, bzw. ob Empfänger bereits Synchronisationspunkt erreicht hat



Remote Procedure Call (RPC)

- Aufruf einer „entfernten Prozedur“
- Synchron-auftragsorientiertes Prinzip

Keine Parallelität
zwischen Client
und Server



accept, return
reply, receive
reply etc. werden
„unsichtbar“
durch Compiler
bzw. Laufzeit-
system erledigt

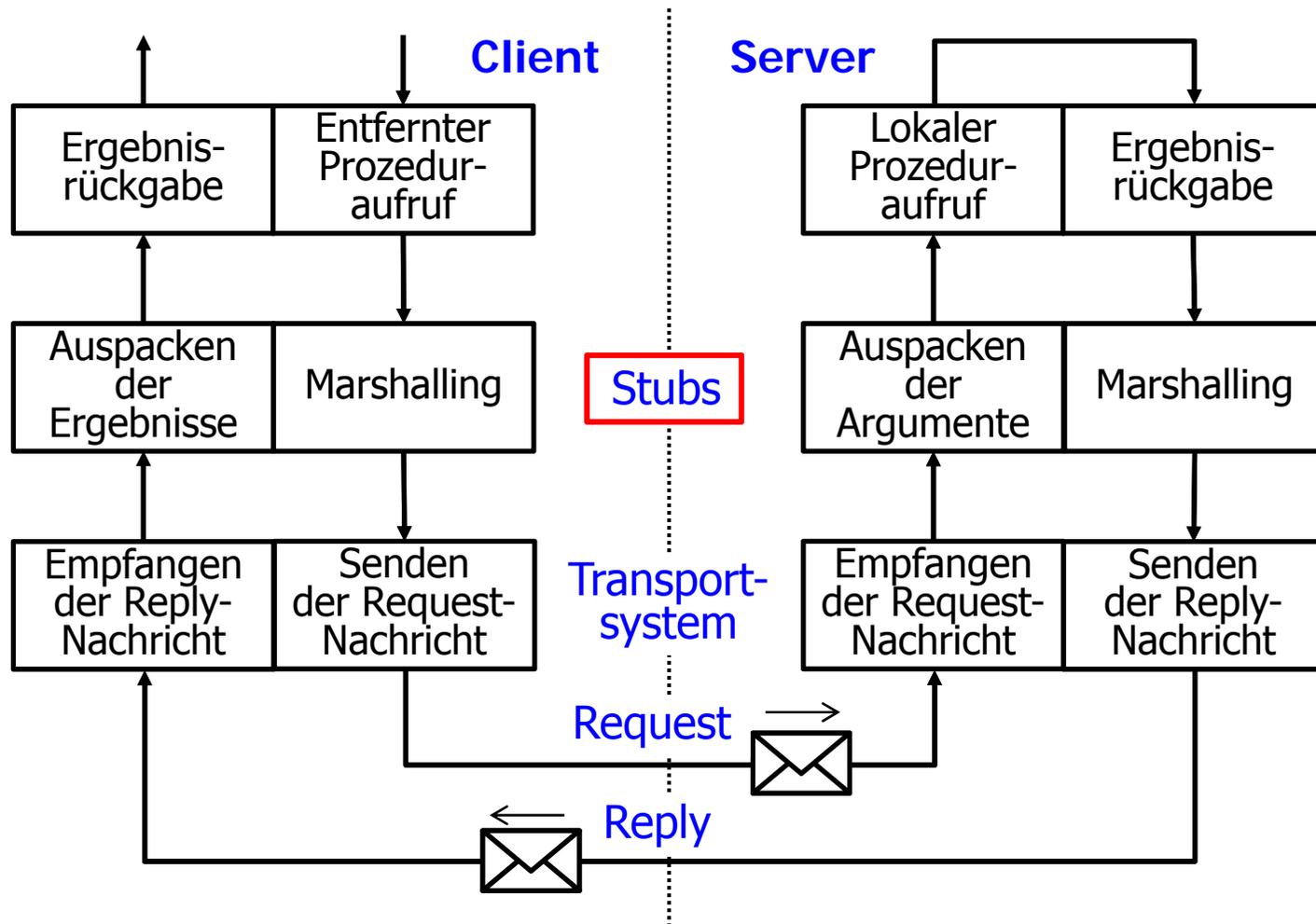
- Bei verteilten objektorientierten Systemen statt dessen:
„Remote Method Invocation“ (RMI), z.B. Java RMI

RPC: Pragmatik

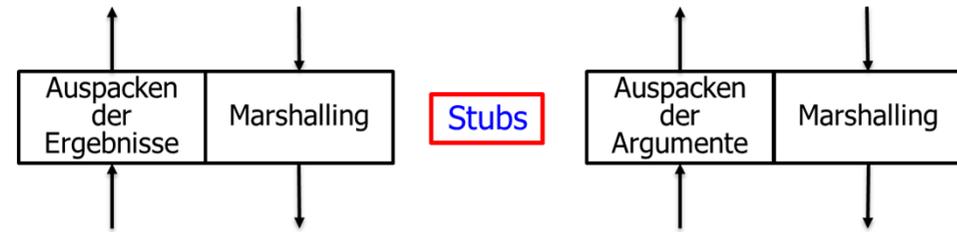


- Soll dem klassischen Prozeduraufruf möglichst gleichen
 - klare Semantik für den Anwender (Auftrag als „Unterprogramm“)
- Einfaches Programmieren
 - kein Erstellen von Nachrichten, kein Quittieren etc. auf Anwendungsebene
 - Syntax analog zu traditionellem lokalem Prozeduraufruf
 - Analoge Verwendung lokaler / entfernter Prozeduren
 - Typsicherheit (Datentypprüfung auf Client- und Serverseite möglich)
- Implementierungsproblem: „Verteilungstransparenz“
 - Verteiltheit so gut wie möglich verbergen

RPC: Implementierung



RPC: Stubs



- **Stub** = Stummel, Stumpf

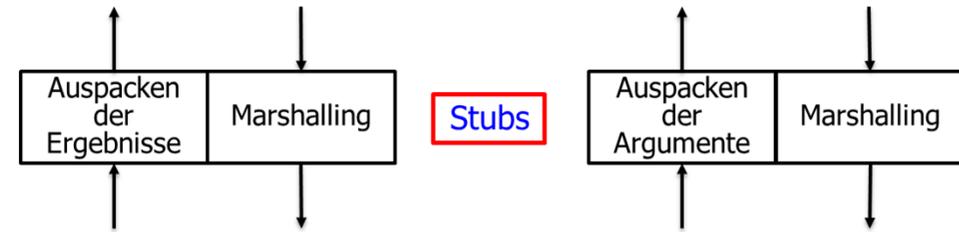
Client:

```
xxx ;  
call S.X(out: a ; in: b);  
xxx ;
```

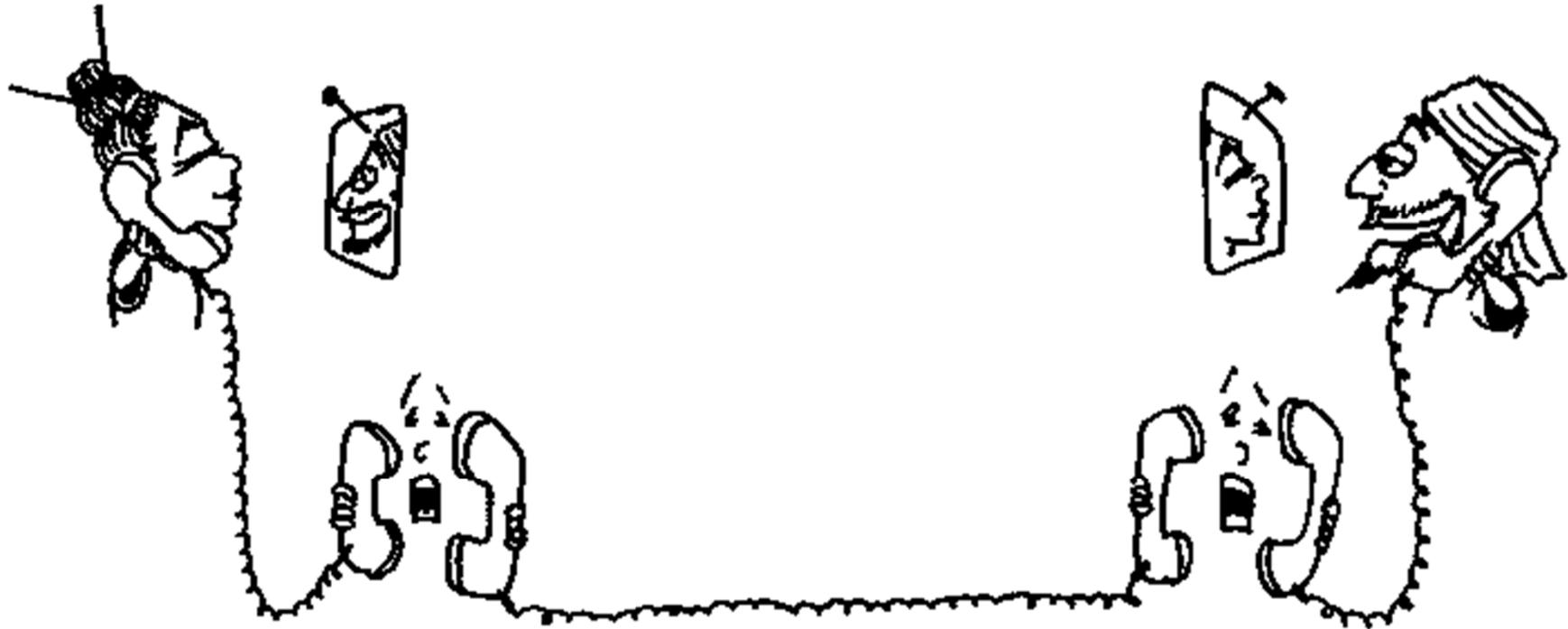
Ersetzt durch ein längeres Programmstück (*Client-Stub*), welches u.a.:

- Parameter a in eine Nachricht packt
- Nachricht an Server S versendet
- Timeout für die Antwort setzt
- Antwort entgegennimmt (oder evtl. exception bei timeout auslöst)
- Ergebnisparameter b mit den Werten der Antwortnachricht setzt

RPC: Stubs (2)



- Wirken als „proxy“
 - lokale Stellvertreter des entfernten Gegenübers
 - (Client-Stub bzw. Server-Stub)
- Simulieren einen lokalen Aufruf
- Sorgen für Zusammenbau und Entpacken von Nachrichten
- Konvertieren Datenrepräsentationen
 - bei heterogenen Umgebungen
- Können oft weitgehend automatisch generiert werden
 - z.B. aus dem Client- oder Server-Code sowie evtl. einer Schnittstellenbeschreibung
- Steuern das Übertragungsprotokoll
 - z.B. zur Behebung von Übertragungsfehlern



„Kommunikation mit Proxies“ (Bild aus dem Buch: „Java ist auch eine Insel“)

RPC: Kompatibilität von Datenformaten und -strukturen?

- Problem: Parameter **komplexer Datentypen** wie
 - Records, Strukturen
 - Objekte
 - Referenzen, Zeiger
 - Zeigergeflechte
 - sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein?
 - sollen Referenzen symbolisch, relativ,... interpretiert werden?
 - wie wird Typkompatibilität sichergestellt?
- Problem: RPCs werden oft in **heterogenen Umgebungen** eingesetzt mit unterschiedlicher Repräsentation, z.B. von
 - Strings (Längelfeld ↔ '\0' als Endekennung)
 - Character (ASCII ↔ Unicode)
 - Arrays (zeilen- ↔ spaltenweise)
 - Zahlen (niedrigstes Bit vorne oder hinten)

RPC: Marshalling



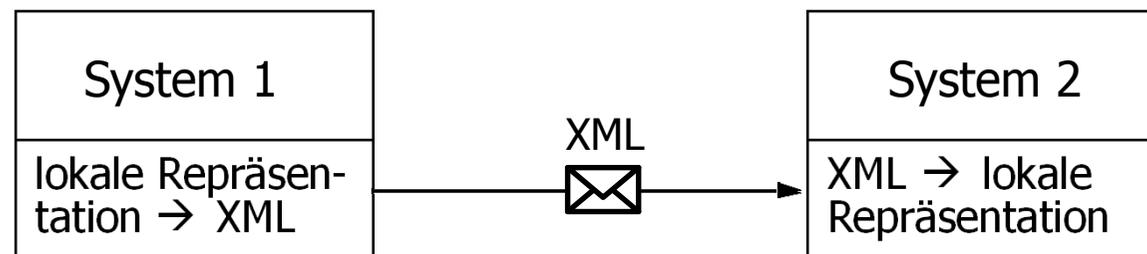
Marshalling: Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern

- evtl. dabei geeignete „standardisierte“ **Codierung** (komplexer) Datenstrukturen
- **Glätten** („flattening“) komplexer (evtl. verzeigeter) Datenstrukturen zu einer Sequenz von Basistypen (evtl. mit Strukturinformation)
- umgekehrte Transformation wird oft als „**unmarshalling**“ bezeichnet

RPC: Datenkonversion

1) Umwandlung in eine gemeinsame Standardrepräsentation

- z.B. **XML** bei „XML-RPCs“ (in Web-Applikationen)



- Beachte: Jeweils **zwei** Konvertierungen erforderlich; für jeden Datentyp jeweils Kodierungs- und Dekodieringsroutinen vorsehen

RPC: Datenkonversion (2)



2) Oder sendeseitig lokale Datenrepräsentation verwenden und dies in der Nachricht vermerken

- „receiver makes it right“
- Vorteil: bei gleichen Systemumgebungen/Computertypen ist keine (doppelte) Umwandlung nötig
- Empfänger muss aber die Senderrepräsentation kennen und mit ihr umgehen können

Generell: Datenkonversion ist überflüssig, wenn sich alle Kommunikationspartner an einen **gemeinsamen Standard** halten

RPC: Transparenzproblematik

- RPCs sollten so weit wie möglich **lokalen Prozeduraufrufen** (als bekanntes Programmierparadigma) gleichen, es gibt aber einige inhärente **Unterschiede**
 - z.B. bei Nichterreichbarkeit oder Absturz des Servers; RPCs dauern auch länger als lokale Prozeduraufrufe
- Beachte auch: Client-/Serverprozesse haben evtl. **unterschiedliche Lebenszyklen**
 - Server könnte z.B. noch nicht oder nicht mehr oder in einer „falschen“ (z.B. veralteten) Version existieren
 - Anwender oder Programmierer eines Clients hat typischerweise keine Kontrolle über den Server

RPC: Leistungstransparenz?

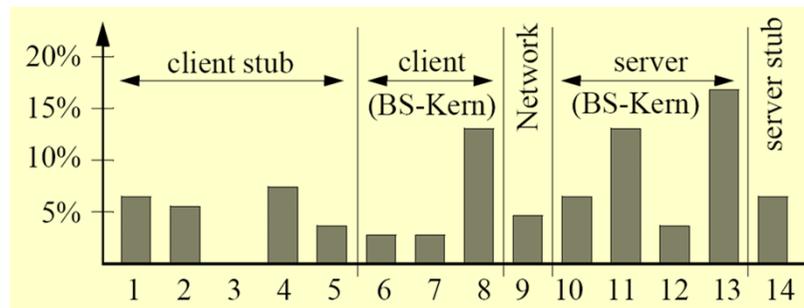


- RPC i.Allg. wesentlich **langsamer** als lokaler Prozeduraufruf
- **Kommunikationsbandbreite** ist bei umfangreichen Datenmengen relevant
- Oft ungewisse, variable **Verzögerungen**

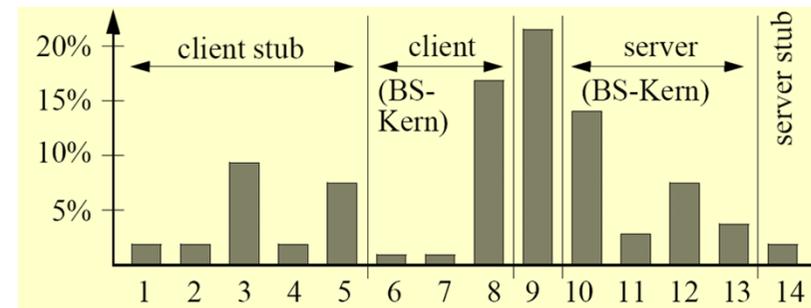
Effizienzanalyse eines RPC-Protokolls

(zitiert nach
A. Tanenbaum)

Null-RPC (Nutznachricht der Länge 0,
keine Auftragsbearbeitung)



1440 Byte-Nutznachricht
(ebenfalls keine Auftragsbearbeitung)



- | | | |
|-------------------------|----------------------------------------------|----------------------------------|
| 1. Call stub | 6. Trap to kernel | 10. Get packet from controller |
| 2. Get message buffer | 7. Queue packet for transmission | 11. Interrupt service routine |
| 3. Marshal parameters | 8. Move packet to controller
over the bus | 12. Compute UDP checksum |
| 4. Fill in headers | 9. Network transmission time | 13. Context switch to user space |
| 5. Compute UDP checksum | | 14. Server stub code |

- Eigentliche **Übertragung** (9) kostet relativ wenig
- **Rechenoverhead** (Prüfsummen, Header etc.) ist nicht vernachlässigbar
- Bei kurzen Nachrichten ist **Kontextwechsel** zwischen Anwendung und Betriebssystem relevant (6, 13)
- Mehrfaches **Kopieren** kostet viel

RPC: Ortstransparenz?



- Evtl. muss Server („Zielort“) bei **Adressierung** explizit genannt werden
- **Getrennte Adressräume** von Client und Server
 - keine Kommunikation über **globale Variablen** möglich
 - typischerweise keine **Pointer** / **Referenzparameter** als Parameter möglich

RPC: Fehlertransparenz?

- Es gibt **mehr Fehlerfälle**
 - beim klassischen Prozeduraufruf gilt: Client = Server → „alles oder nichts“
 - hier: partielle („einseitige“) Systemausfälle typisch (Server-Absturz, Client-Absturz)
- **Nachrichtenverlust**
 - ununterscheidbar von zu langsamer Nachricht!
- Client/Server haben zumindest zwischenzeitlich eine **unterschiedliche Sicht** des Zustands einer „RPC-Transaktion“
 - crash kann im „ungünstigsten Moment“ des RPC-Protokolls erfolgen

⇒ Fehlerproblematik ist also „kompliziert“!

Typische RPC-Fehlerursachen

Wir besprechen nachfolgend 4 **typische Fehlerursachen**:

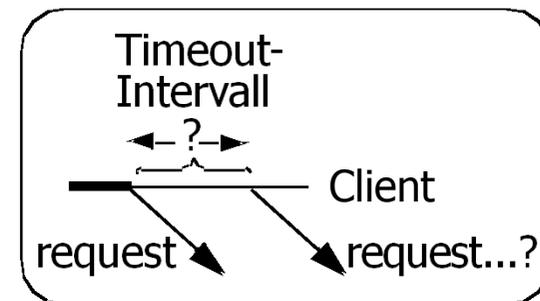
1. Verlorene Request-Nachricht
2. Verlorene Reply-Nachricht
3. Server-Crash
4. Client-Crash

Grundprobleme:

- Client / Server haben **temporär** eine **inkonsistente Sicht**
- **Timeout** beim Client kann **verschiedene Ursachen** haben (verlorener Request, verlorenes Reply, langsamer Request bzw. Reply, langsamer Server, abgestürzter Server,...) → Fehlermaskierung schwierig

Verlorene Request-Nachricht

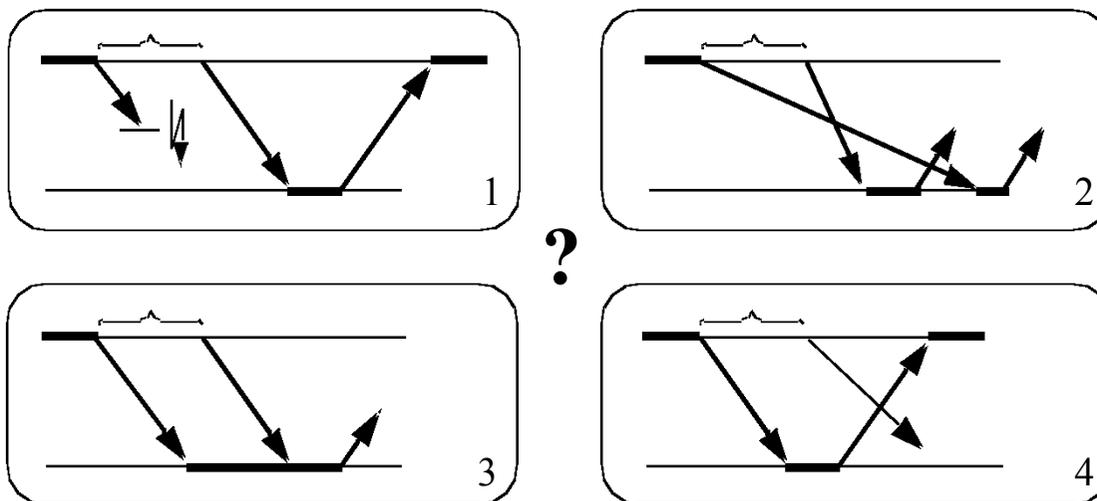
- **Gegenmassnahme:**
Nach Timeout (kein Reply) die Request-Nachricht erneut senden



- **Probleme:**
 - Wie viele Wiederholungsversuche maximal?
 - Wie gross soll der Timeout sein?
 - Was, wenn die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam?

Verlorene Request-Nachricht (2)

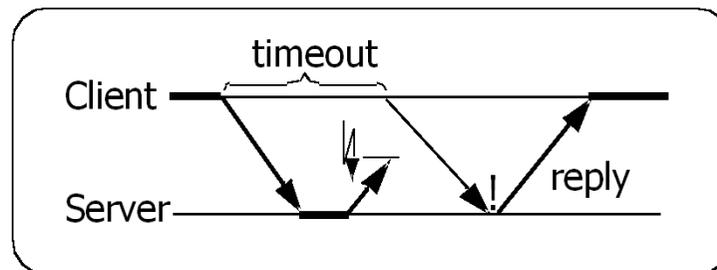
- **Probleme**, wenn Nachricht tatsächlich gar nicht verloren:
 - Doppelte Request-Nachricht!
(Gefährlich bei nicht-idempotenten serverseitigen Operationen!)
 - Server sollte solche Duplikate erkennen (Denkübung: Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet? Genügen Sequenznummern? Zeitmarken?)
 - Würde das Quittieren der Request-Nachricht etwas bringen?



Bei automatischer Wiederholung von Request-Nachrichten ist eine **Vielzahl möglicher Fälle** zu unterscheiden

Verlorene Reply-Nachricht

- **Gegenmassnahme 1:**
analog zu verlorener Request-Nachricht
 - also: Anfrage bei Timeout wiederholen

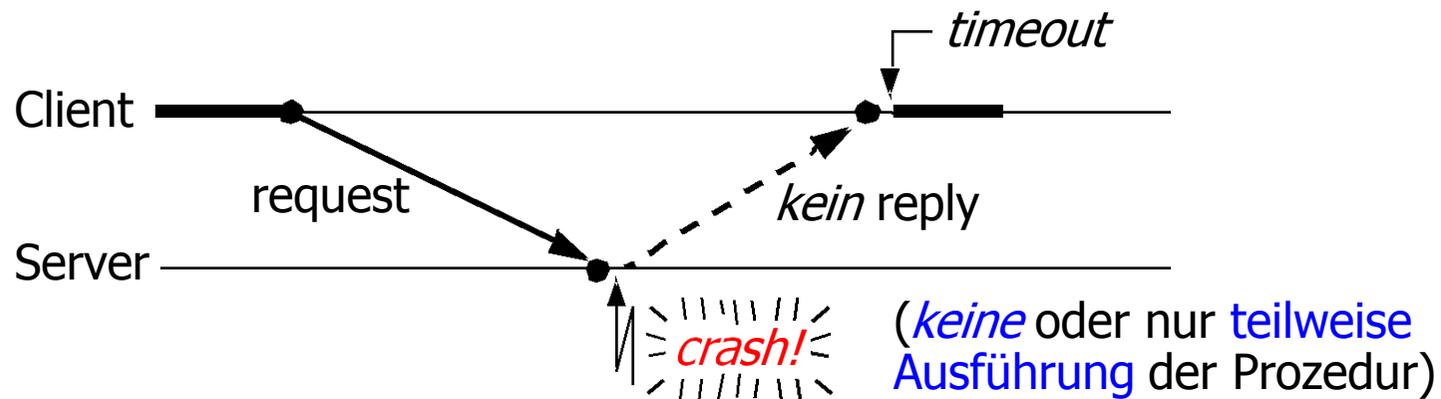
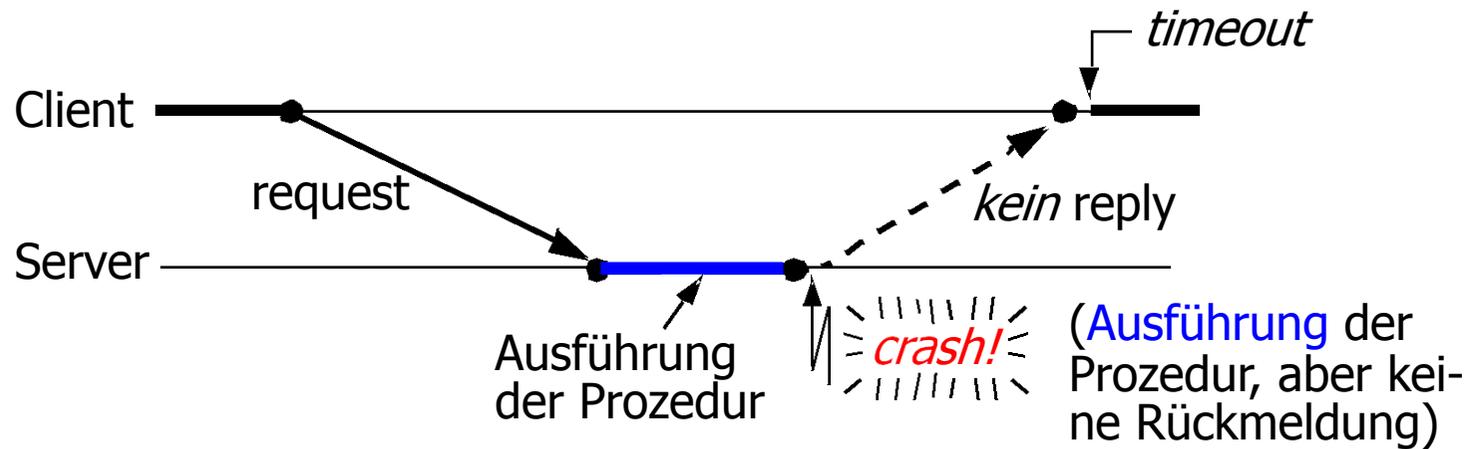


- **Probleme:**
 - vielleicht ging aber tatsächlich der **Request verloren?**
 - oder der **Server** war nur **langsam** und arbeitet noch?
 - ist aus Sicht des Clients nicht unterscheidbar!

Verlorene Reply-Nachricht (2)

- **Gegenmassnahme 2:**
Server hält eine „Historie“ versendeter Replies
 - falls Server Request-Duplikate erkennt und den Auftrag bereits ausgeführt hat: letztes Reply erneut senden, ohne die Prozedur nochmal auszuführen
 - pro Client muss nur das neueste Reply gespeichert werden
- Bei vielen Clients u.U. dennoch **Speicherplatzprobleme:**
→ Historie nach „einiger“ Zeit löschen bzw. kürzen
 - und wenn man ein gelöschttes Reply später dennoch braucht?
 - ist in diesem Zusammenhang ein ack eines Reply sinnvoll?

Server-Crash



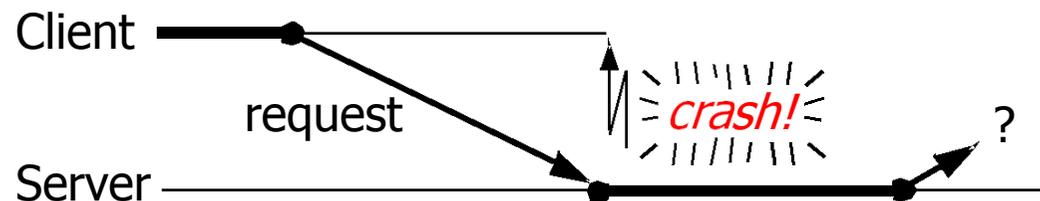
Server-Crash (2)



- **Probleme:** Wie soll der Client obige Fälle unterscheiden?
 - ebenso: Unterschied zu verlorenem request bzw. reply?
 - Sinn und Erfolg konkreter Gegenmassnahmen hängt u.U. davon ab!
 - Client meint evtl. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (→ falsche Sicht des Zustandes!)
- Evtl. Probleme nach einem **Server-Restart**
 - z.B. „Locks“, die noch bestehen (Gegenmassnahmen?) bzw. allgemein: „verschmutzter“ Zustand durch frühere Inkarnation
 - typischerweise ungenügend Information, um in alte Transaktionszustände problemlos wieder einzusteigen

Client-Crash

- Oder auch:
Client mittlerweile nicht mehr am reply interessiert



- Reply des Servers wird **nicht abgenommen**
 - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
 - blockiert i.Allg. Ressourcen beim Server!

Client-Crash (2)



- Problem: „Orphans“ (Waisenkinder) beim Server
 - Prozesse, deren Auftraggeber nicht mehr existiert
- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)
 - Orphans könnten aber bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben,...
- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)
- „Sehr“ alte Prozesse, die für einen Auftrag gegründet wurden, werden als Orphans angesehen und terminiert

RPC-Fehlersemantik-Klassen



- **Operationale Sichtweise:**
 - Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?
-

1. Maybe-Semantik:

- Keine Wiederholung von Requests
- **Einfach** und **effizient**
- Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
- Mögliche Anwendungsklasse: z.B. Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

RPC-Fehlersemantik-Klassen (2)

2. At-least-once-Semantik:

- Automatische Wiederholung (evtl. mehrfach) von Requests
- Keine Duplikatserkennung (zustandsloses Protokoll auf Serverseite)
- Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

1) und 2) werden etwas euphemistisch oft als „best effort“ bezeichnet

3. At-most-once-Semantik:

- Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
- Keine wiederholte Ausführung der Prozedur, sondern evtl. erneutes Senden des (gemerkten) Reply
- Geeignet auch für nicht-idempotente Operationen

RPC-Fehlersemantik-Klassen (3)

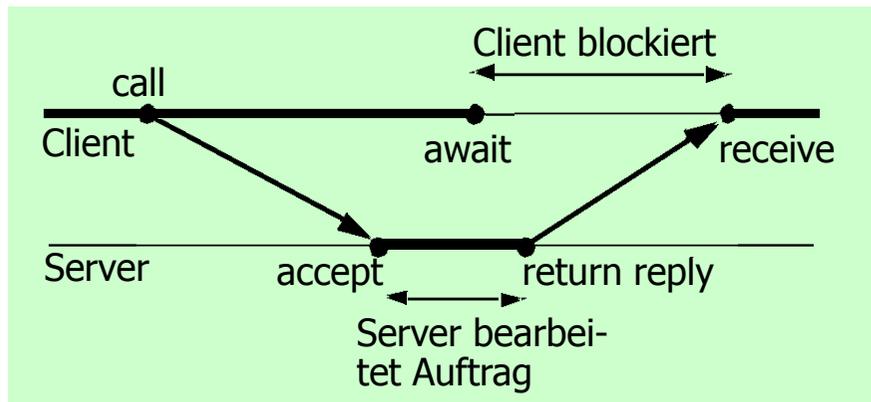
- **Maybe** → **at-least-once** → **at-most-once** → ...
ist zunehmend aufwändiger zu realisieren

Ist „**exactly-once**“ machbar?

- Man begnügt sich daher, falls es die Anwendung erlaubt, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: **so billig wie möglich, so „perfekt“ wie nötig**

Asynchroner RPC

- Andere Bezeichnung: „Remote Service Invocation“
- **Auftragsorientiert**, d.h. also: Antwortverpflichtung

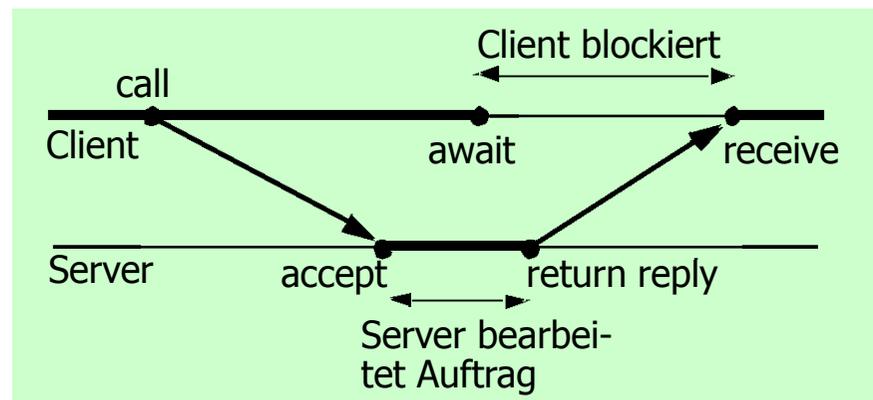


- **Parallelverarbeitung** von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Asynchroner RPC: Zuordnung Auftrag/Ergebnisempfang

- Unterschiedliche Ausprägung auf Sprachebene möglich

- „await“ könnte z.B. einen bei „call“ zurückgelieferten „**handle**“ als Parameter erhalten, also z.B.: `Y = call X(...); ... await(Y);`



- evtl. könnte die Antwort auch **asynchron** in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt- oder Exception-Routine)

Asynchroner RPC: Future-Variable

- Spracheinbettung evtl. auch durch „Future-Variablen“
- Future-Variable = „handle“ auf das in anderem Thread parallel berechnetes Funktionsergebnis
- Programm **blockiert nur dann**, wenn der Wert der Variable bei ihrer **Nutzung** noch nicht feststeht
- Beispiel (Scala):

```
...  
val x = future(callRPC())  
... // continue local computation  
println(x()) // await x iff necessary
```

Einige Varianten / Ergänzungen zu RPCs in der Praxis

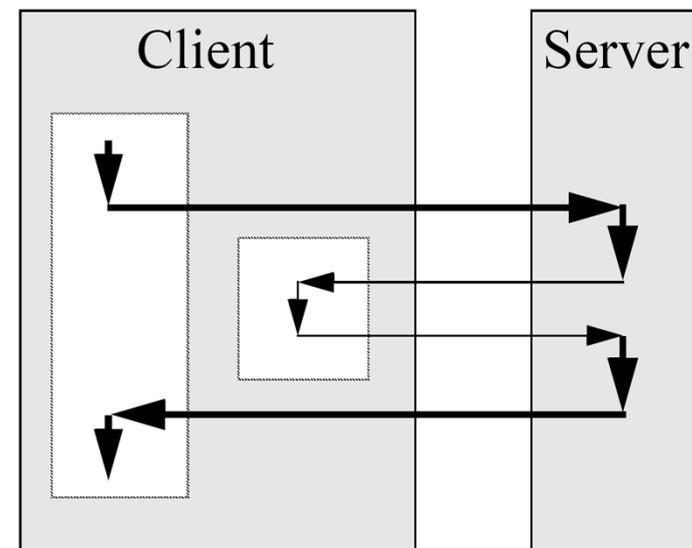


Wir besprechen nachfolgend 4 ergänzende Aspekte:

1. Rückrufe
2. Context-Handles
3. Broadcast bzw. Multicast
4. Sicherheit

Rückrufe („call back RPC“)

- **Temporärer Rollentausch** von Client und Server
 - um eventuell bei langen Aktionen **Zwischenresultate** zurückzumelden
 - um eventuell **weitere Daten** vom Client anzufordern
- Client muss Rückrufadresse beim call übergeben



Context-Handles



- Struktur mit **Kontextinformation** zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg
 - vgl. „cookies“: kleine Textdatei, die ein Web-Server einem Browser (= Client) schickt und die im Browser gespeichert wird
- Context-Handles werden **vom Server dynamisch erzeugt** und an den Client (bei „reply“) zurückgegeben
- Client kann diese beim **nächsten Aufruf** (unverändert) wieder **mitsenden**
 - Server „erinnert“ sich so an den Kontext
- Vorteil: Server selbst arbeitet „zustandslos“

Weitere Varianten / Ergänzungen



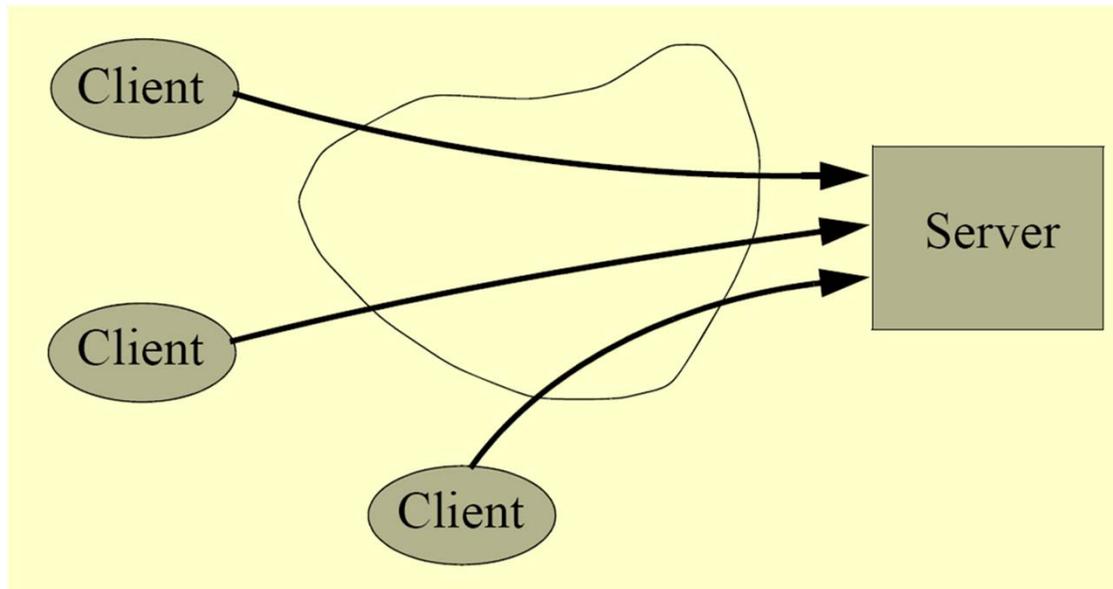
- **Broadcast** bzw. **Multicast**
 - Request wird „gleichzeitig“ an mehrere Server geschickt
 - RPC ist beendet mit der ersten empfangenen Antwort oder Client hat die Möglichkeit, nach einer Antwort auf eine weitere Antwort zu warten
- Oft wählbare **Sicherheitsstufen**, z.B.:
 - Authentifizierung bei Aufbau der Verbindung (“binding”)
 - Authentifizierung pro RPC-Aufruf oder pro Nachricht
 - Verschlüsselung der zugrundeliegenden Nachrichten
 - Schutz gegen Verfälschung
(digitale Signatur, verschlüsselte Prüfsumme, MAC)

Beispiel „Secure RPC“ (Teil des „SUN RPC“)

- Client und Server vereinbaren (geheim) einen **Session-Key K**
 - wird zum Verschlüsseln der Nachrichten genutzt
- Jeder **Request** enthält einen **mit K kodierten Zeitstempel**
- Erster Request enthält zusätzlich verschlüsselt eine **Zeitfenstergrösse W** als zeitliches Toleranzintervall sowie (ebenfalls verschlüsselt) **W-1**
 - „zufälliges“ **Generieren** einer ersten Nachricht ist so nahezu unmöglich
 - **replay** (bei kleinem W) ist ebenfalls erfolglos
 - W ist verschlüsselt, auch um Angreifern keine Information über die Fenstergrösse zu geben
- Server akzeptiert einen Request nur, wenn:
 - Zeitstempel **grösser als letzter Zeitstempel** und
 - Zeitstempel **innerhalb des Zeitfensters**
- Zur **Authentifizierung** enthält die Antwort des Servers (verschlüsselt) den letzten erhaltenen **Zeitstempel-1**

Gleichzeitige Server-Aufträge

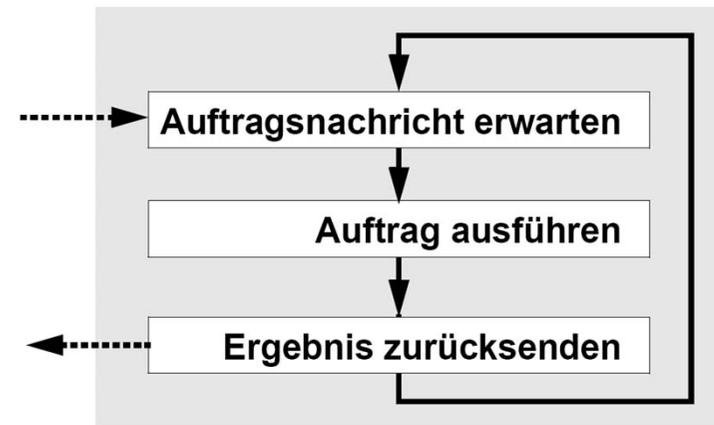
- Problem: Oft viele „gleichzeitige“ Aufträge



- Diverse Realisierungsmöglichkeiten →

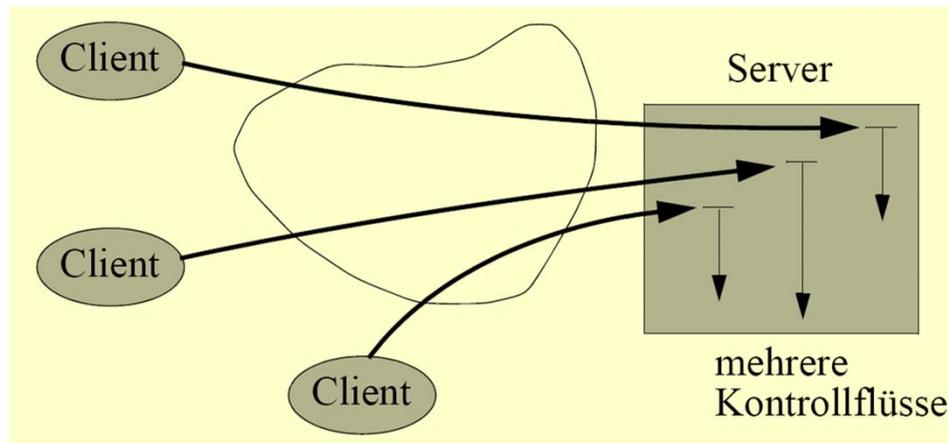
Iterativer Server („single threaded“)

- **Iterative Server** bearbeiten nur einen einzigen Auftrag pro Zeit
 - eintreffende Anfragen während der Auftragsbearbeitung: in Warteschlange puffern, abweisen oder schlichtweg ignorieren
 - einfach zu realisieren
 - bei trivialen Diensten mit kurzer Bearbeitungszeit oft sinnvoll, ansonsten konkurrenente Server



Konkurrente Server

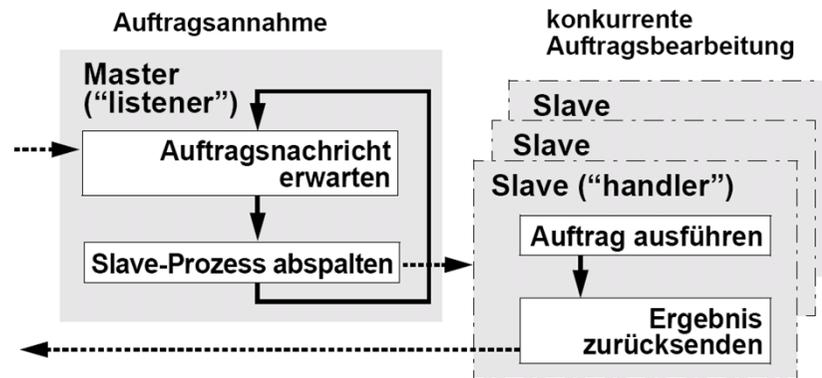
- (Quasi)gleichzeitige Bearbeitung mehrerer Aufträge
 - sinnvoll (d.h. effizienter für Clients) bei längeren Aufträgen



Verschiedene Realisierungen

- Verbund mehrerer Server-Maschinen (Cluster)
 - mehrere Prozessoren bzw. Multicore-Prozessoren
 - dynamische (oder vorgegründete Prozesse)
- Beachte: Auch bei Monoprocessor-Systemen ist **Timesharing** sinnvoll: Nutzung erzwungener Wartezeiten während einer Auftragsbearbeitung für Aufträge anderer Klienten; **kürzere mittlere Antwortzeiten** bei Jobmix aus langen und kurzen Aufträgen

Konkurrenente Server mit dynamischen Handler-Prozessen



Für jeden Auftrag gründet der **Master** einen neuen **Slave-Prozess** und wartet dann auf einen neuen Auftrag

Alternative: „**Process preallocation**“: Feste Anzahl statischer Slave-Prozesse (evtl. effizienter, da Wegfall der Erzeugungskosten)

- Neu gegründeter Slave („**handler**“) übernimmt den Auftrag
- Client kommuniziert dann direkt mit dem Slave
- Slaves sind typischerweise Leichtgewichtsprozesse („**threads**“)
- Slaves **terminieren** i.Allg. nach Beendigung des Auftrags
- Die **Anzahl gleichzeitiger Slaves** sollte begrenzt werden

Master/Slave



Subject: Identification of equipment sold to LA County

Date: Tue, 18 Nov 2003 14:21:16 -0800

From: "Los Angeles County"

The County of Los Angeles actively promotes and is committed to ensure a work environment that is free from any discriminatory influence be it actual or perceived. As such, it is the County's expectation that our manufacturers, suppliers and contractors make a concentrated effort to ensure that any equipment, supplies or services that are provided to County departments do not possess or portray an image that may be construed as offensive or defamatory in nature.

One such recent example included the manufacturer's labeling of equipment where the words "Master/Slave" appeared to identify the primary and secondary sources. Based on the cultural diversity and sensitivity of Los Angeles County, this is not an acceptable identification label.

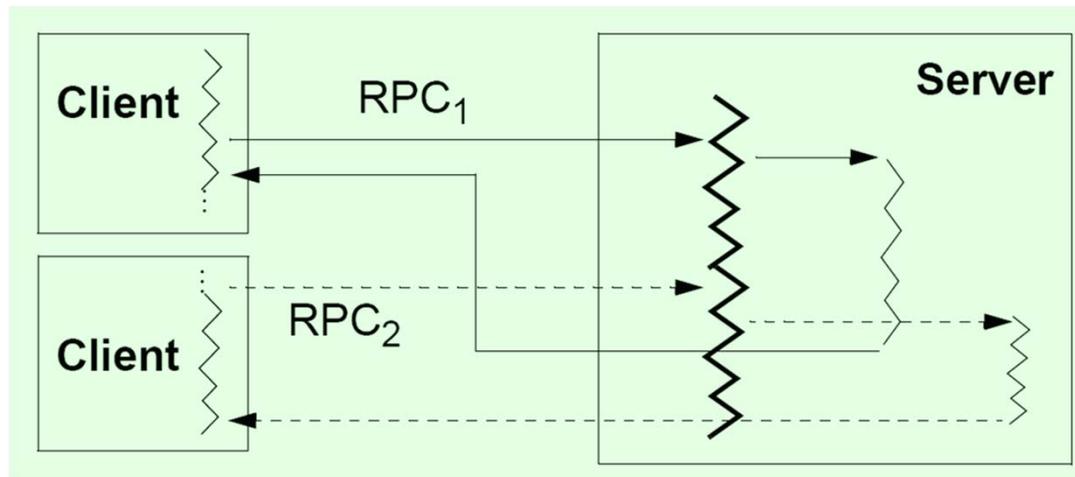
We would request that each manufacturer, supplier and contractor review, identify and remove/change any identification or labeling of equipment or components thereof that could be interpreted as discriminatory or offensive in nature before such equipment is sold or otherwise provided to any County department.

Thank you in advance for your cooperation and assistance.

Joe Sandoval, Division Manager
Purchasing and Contract Services
Internal Services Department
County of Los Angeles

Multithreading beim Client-Server-Konzept

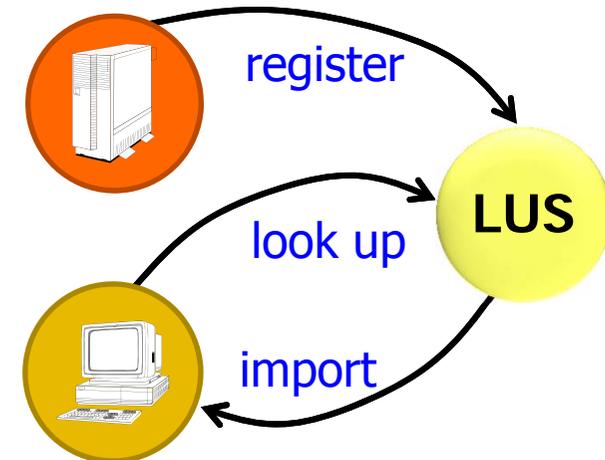
- **Server-Threads:** quasiparallele Bearbeitung von Aufträgen
 - Server bleibt ständig empfangsbereit



- **Client-Threads:** Möglichkeit zum „asynchronen RPC“
 - Hauptkontrollfluss delegiert RPCs an nebenläufige Threads
 - keine Blockade durch Aufrufe im Hauptfluss
 - echte Parallelität von Client (Hauptkontrollfluss) und Server

Lookup-Service

- Problem: **Wie finden sich Client und Server?**
 - haben i.Allg. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gemeinsame Umgebung)
 - → Lookup-Service (**LUS**) oder „Registry“ fungiert als „Service-Broker“
- **Server** gibt seinen Service (d.h. RPC-Routine) dem LUS bekannt
 - **register**: RPC-Schnittstelle „exportieren“ (Name, Parameter, Typen,...)
 - evtl. später auch wieder abmelden
- **Client** erfragt beim LUS die Adresse eines geeigneten Servers
 - Angabe des gewünschten Typs von Service beim **look up** oder **discovery**
 - „importieren“ der RPC-Schnittstelle



Lookup-Service (2)



- **Vorteile** („Mehrwert“): im Prinzip kann LUS
 - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
 - Autorisierung etc. überprüfen
 - durch Polling der Server die Verfügbarkeit eines Services testen
 - verschiedene Versionen eines Dienstes verwalten
- **Probleme:**
 - look up kostet Ausführungszeit (gegenüber statischem Binden)
 - zentraler LUS ist ein potentieller Engpass / „single point of failure“ (evtl. LUS geeignet replizieren / verteilen)
 - wie lernen Client oder Server die Adresse des „richtigen“ oder „zuständigen“ LUS kennen?
 - z.B. feste („well-know“) Adresse oder Broadcast in lokaler Umgebung

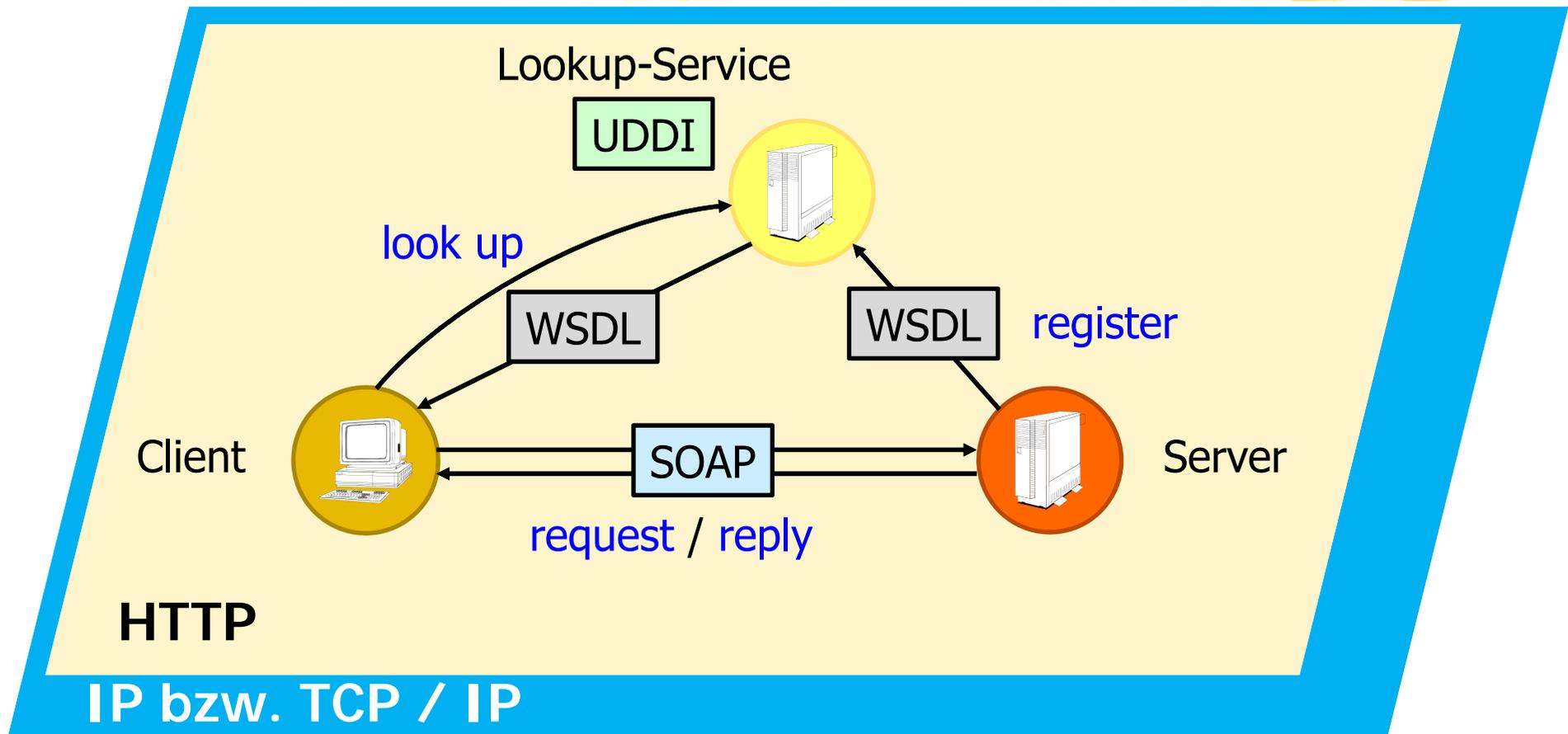
Web Services als Beispiel für das Client-Server-Modell

- **Problem:** Internet ist zu heterogen für eine einheitliche Programmiersprache oder RPC-Laufzeitumgebung
 - „Lösung“: **Web Services** als offener, plattform- bzw. sprachunabhängiger Standard, dessen Schnittstellen von diversen Plattformen implementiert werden können

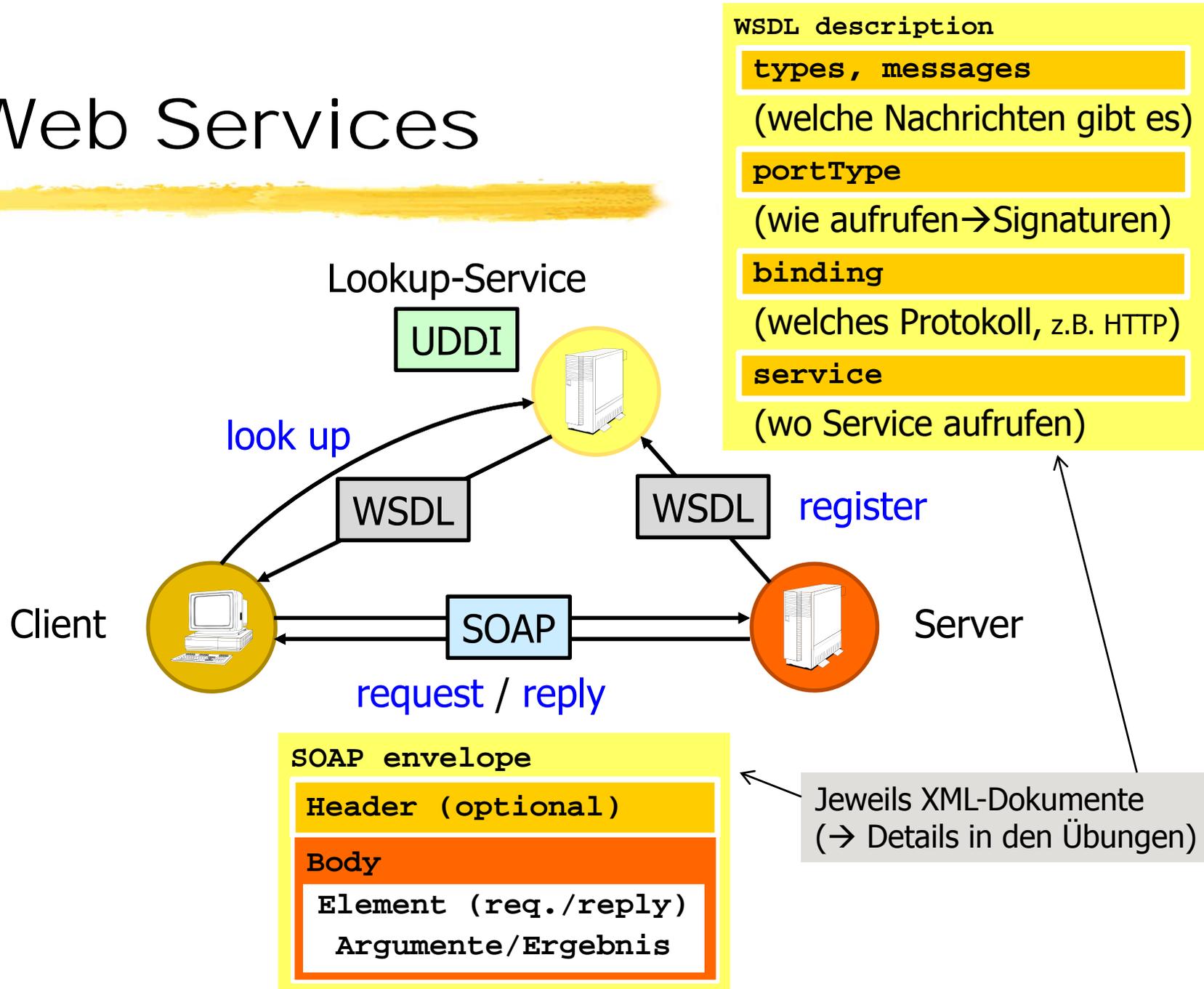
-
- **HTTP** fungiert als Transportschicht ←
 - **SOAP** als plattformunabhängige Protokollspezifikation (ursprünglich: „Simple Object Access Protocol“)
 - **UDDI** als Lookup-Service („Universal Description, Discovery and Integration“)
 - **WSDL** als standardisierte Service-Beschreibung („Web Services Description Language“)

Alternativ zu HTTP:
UDP oder **SMTP**, z.B.
für Mitteilungen ohne
Antwort oder wenn
Resultatberechnung
länger als typischer
HTTP-Timeout dauert

Web Services



Web Services



Web Services

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

XML-Dokument, das (bzgl. UDDI bzw. für einen Client) den Service beschreibt

XML (Extensible Markup Language):

- Auszeichnungssprache zur Darstellung **hierarchisch strukturierter** Daten in Form von Textdaten (z.B. ASCII)
- Unterstrukturen mit Start- (`<Tag-Name>`) und End-Tag (`</Tag-Name>`) oder einem Empty-Element-Tag (`<Tag-Name />`)
- **Attribute** bei einem Tag (Attribut-Name="Attribut-Wert") für Zusatzinformationen

```
<books>
  <book>
    <author>Karl May</author>
    <title>Winnetou</title>
    <ISBN>3-7802-0170-4</ISBN>
    <price format="EUR"/>
  </book>
  <pubinfo>
    <publisher>
      KM-Verlag
    </publisher>
    <town>Bamberg</town>
  </pubinfo>
</books>
```

WSDL: types

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

types

XML-Schema zur Typenbeschreibung

- Definition von Typen (bzw. deren Namen) als XML-„**element**“
- Meistens als Verweis auf einen „**complexType**“, der aus „elements“ der Basistypen (int, float,...) besteht
- Schema definiert auch einen **Namensraum**, der als **URI** angegeben wird
- Schema oft in externe **.xsd**-Datei ausgelagert

→ nächste 2 slides

WSDL-Namensräume

- ExampleWebServices.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
  name="ExampleWebServices"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap">
  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://example.org/VS/WebServices/"
        schemaLocation="ExampleSchema.xsd"/>
      </xsd:schema>
    </types>
```

Namensraum der beschriebenen Web Services

Weitere Namensräume, aus denen Tags benötigt werden

Importiert die Typendefinitionen; können auch eigenen Namensraum haben

nächste slide

...

WSDL-Namensräume

■ ExampleSchema.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema
  version="1.0"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- Elementdefinitionen -->
</xs:schema>
```

Hier gleicher
Namensraum
wie Services

Definiert das Präfix „tns:“, um den
targetNamespace anzusprechen

Lässt „xs:“ auf den allgemeinen
XML-Schema-Namensraum zeigen

nächste
slide

WSDL-Beispiel: types

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

Diese Namen sind nun Teil des Namensraums

„type“ würde den xs-Namensraum nutzen, daher der Präfix

types

XML-Schema zur Typenbeschreibung

```
<xs:element name="myArgs" type="tns:myObject"/>
<xs:complexType name="myObject">
  <xs:sequence>
    <xs:element name="i" type="xs:int"/>
    <xs:element name="j" type="xs:float"/>
  </xs:sequence>
</xs:complexType>
```

„xs:“ definiert element, complexType, int etc.

„myArgs“ besteht also aus einem int und einem float

WSDL: messages

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

messages

- Abstrakte Definition der Nachrichten, mit denen ein Dienst angesprochen wird oder antwortet (können unter „bindings“ für spezielle Protokolle konkretisiert werden)
- Daten können in mehrere Teile („part“) gruppiert werden, die jeweils einem «type element» entsprechen
- Je ein Eintrag pro Nachrichtendefinition

WSDL-Beispiel: messages

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

messages

```
<message name="myRequest">
  <part name="parameters"
        element="tns:myArgs"/>
  <part name="optionalParameters"
        element="tns:myOpt"/>
</message>

<message name="myResponse">
  <part name="result" element="tns:myRet"/>
</message>
```

„myArgs“ wurde oben definiert

Die Gliederung in „parts“ ist optional

„myRequest“ hat also einen int und einen float als Parameter

WSDL: portType

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

portType

- Definition der einzelnen Web Services
- Jeder **Service** entspricht einer „**operation**“
 - hat typischerweise eine „**input**“-Nachricht und eine „**output**“-Nachricht
 - zusätzlich können Fehlerbenachrichtigungen angegeben werden („**fault**“)
- Services können auch **unidirektional** sein, (z.B. nur „output“ für Benachrichtigungen)

WSDL-Beispiel: portType

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

portType

```
<portType name="groupOfServices">
  <operation name="myService">
    <input message="tns:myRequest"/>
    <output message="tns:myResponse"/>
    <fault message="tns:someFault"/>
  </operation>
</portType>
```

Hier könnten weitere
„operation“ stehen

**„myService“ wird also mit einem
int und einem float aufgerufen**

WSDL: binding

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

binding

- Bindet „portType“ an ein Protokoll (z.B. HTTP)
- Es kann mehrere „binding“-Einträge für verschiedene Protokolle geben
- Im Normalfall genügen die Informationen aus „message“ und „portType“ für die Abbildung der Nachrichten auf ein konkretes Format (d.h. „binding“ enthält kaum Information)

WSDL-Beispiel: binding

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

(wo Service aufrufen)

binding

```
<binding name="myBinding"  
         type="tns:groupOfServices">  
  <soap:binding  
    transport="http://schemas.xmlsoap.org/soap/http"  
    style="document"/>  
</binding>
```

Allgemeiner als „rpc“
(veralteter Web Service „style“)

URI definiert HTTP als
Transportprotokoll
(mit anderen URIs
können beliebige
Protokolle zwischen
Client und Server
vereinbart werden)

WSDL: service

WSDL description

types, messages

(welche Nachrichten gibt es)

portType

(wie aufrufen → Signaturen)

binding

(welches Protokoll, z.B. HTTP)

service

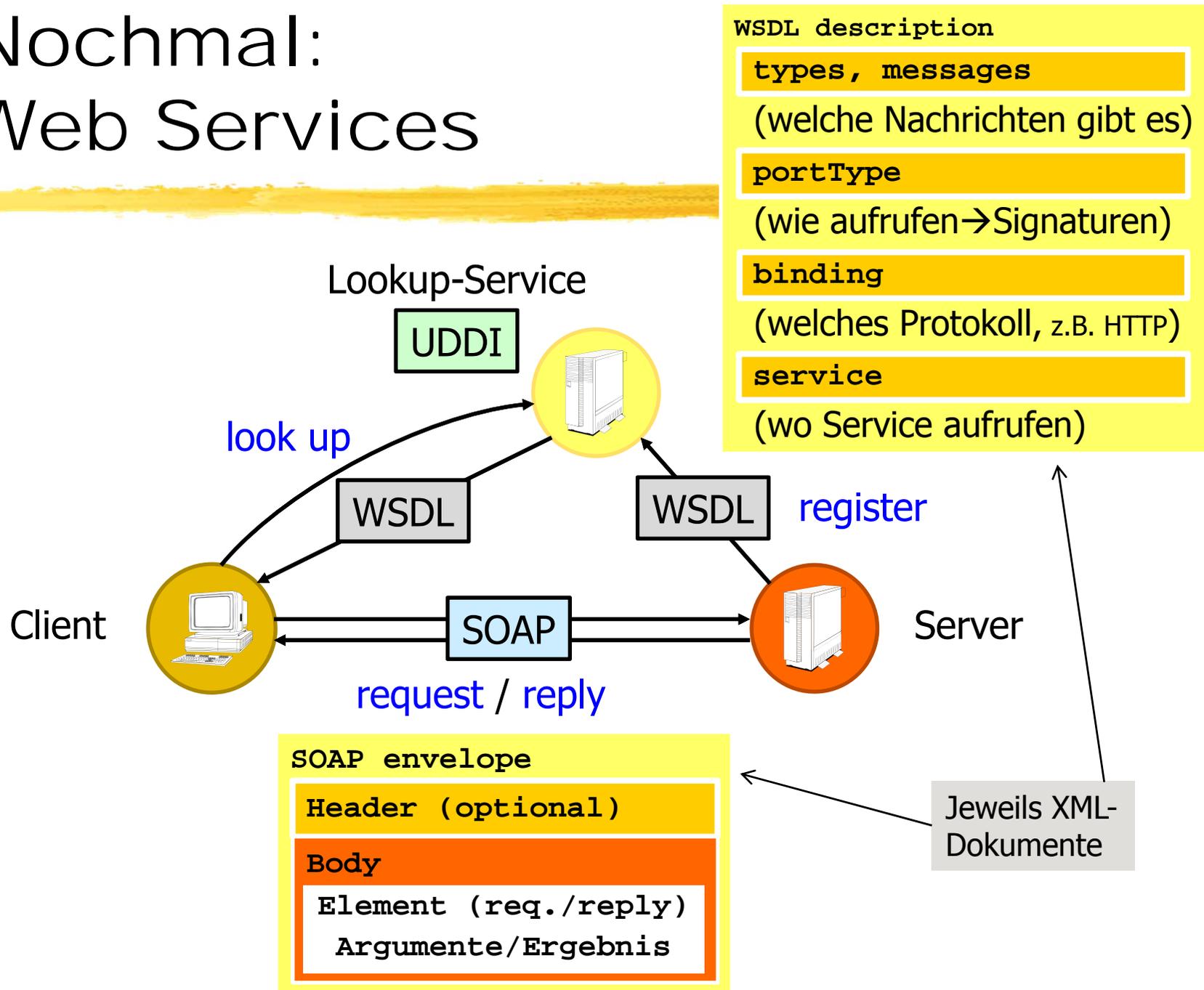
(wo Service aufrufen)

service

- Gibt **Adressen der Services** an (pro „portType“ und „binding“)
- Kann mehrere definierte „portTypes“ zu einem Service bündeln

```
<service name="specificService">  
  <port binding="tns:myBinding">  
    <soap:address  
      location="http://example.org/VS/service"/>  
    </port>  
</service>
```

Nochmal: Web Services



SOAP: envelope

SOAP envelope

Header (optional)

Body

Element (req./reply)
Argumente/Ergebnis

Envelope

Erforderliche Attribute:

- „namespace“ definiert das Dokument als SOAP-Nachricht
- „encodingStyle“ gibt Kodierungsregeln für die SOAP-Serialisierung an

Enthaltene Teilstrukturen:

- Header (optional)
- Body (erforderlich)

SOAP: header

SOAP envelope

Header (optional)

Body

Element (req./reply)
Argumente/Ergebnis

Header

Der SOAP-Header muss nicht enthalten sein

Durch ihn könnten zusätzliche Informationen über den Transaktionskontext, z.B. bezüglich Authentifizierung oder Bezahlung, angegeben werden

SOAP: body

SOAP envelope

Header (optional)

Body

Element (req./reply)
Argumente/Ergebnis

Body

SOAP-Nutzlast:

- Enthält die im WSDL definierte „message“ mit ihren Elementen
- Es wird der Namensraum aus dem WSDL angegeben, womit die dort definierten Tags verwendet werden können

SOAP-Beispiel: Request

Adresse des Service

HTTP-Header

```
POST /VS/service HTTP/1.1
Host: example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 355
```

SOAP envelope

Header (optional)

Body

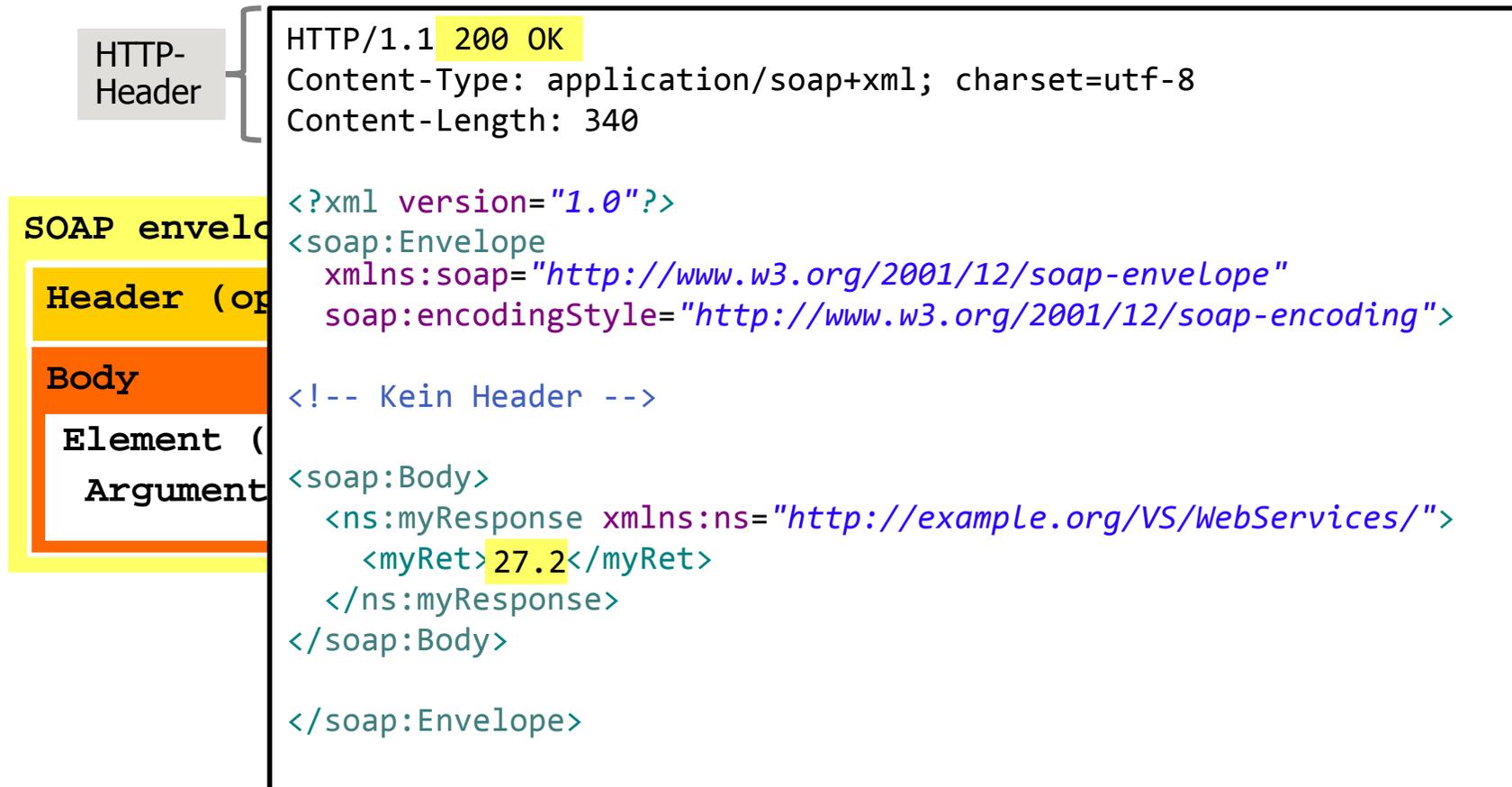
Element (Argument)

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <!-- Kein Header -->
  <soap:Body>
    <ns:myRequest xmlns:ns="http://example.org/Vs/WebServices/">
      <myArgs><i>23</i><j>4.2</j></myArgs>
    </ns:myRequest>
  </soap:Body>
</soap:Envelope>
```

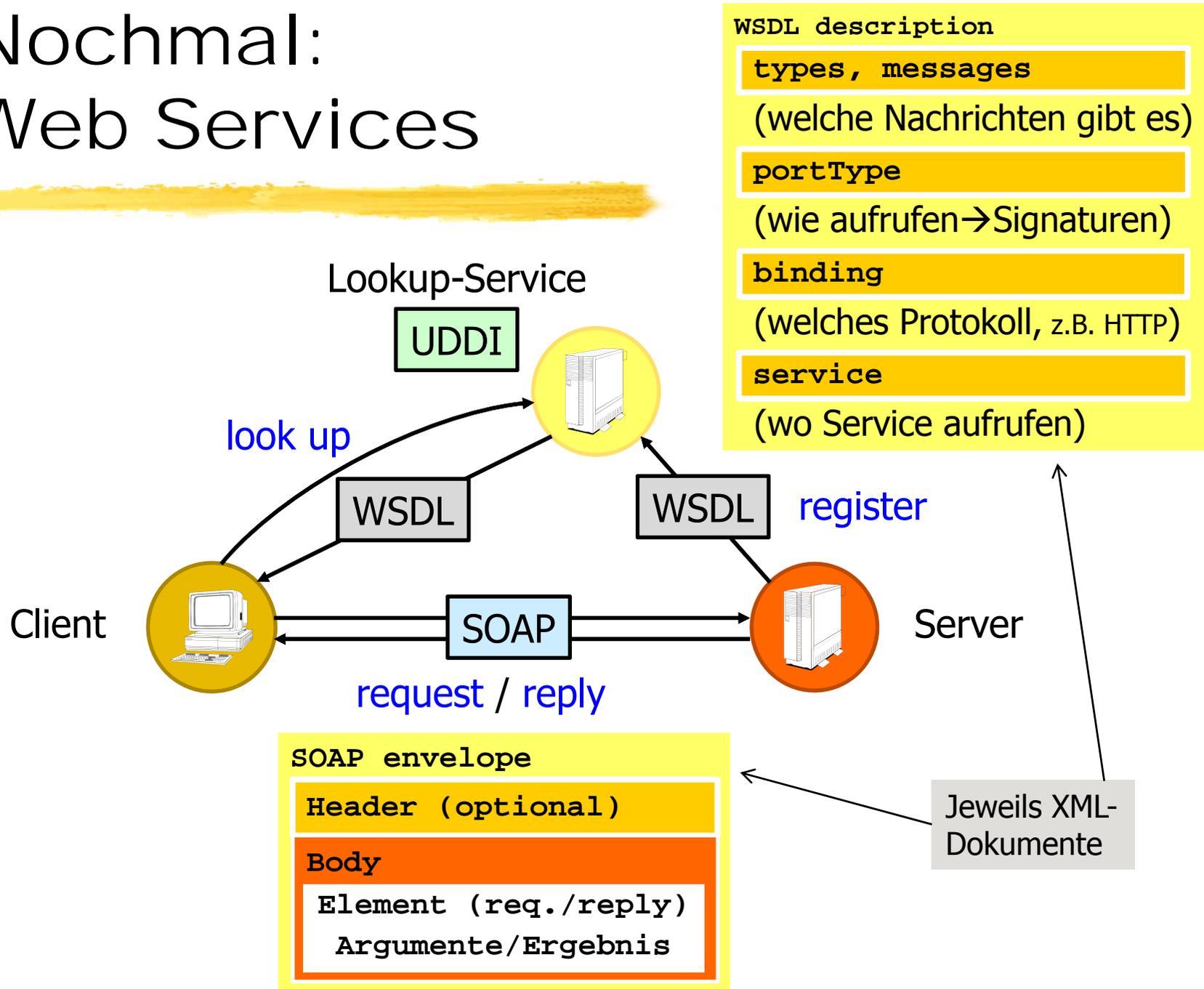
Das wird vom Client-Stub („SOAP engine“) generiert

Namensraum aus WSDL mit Präfix „ns:“

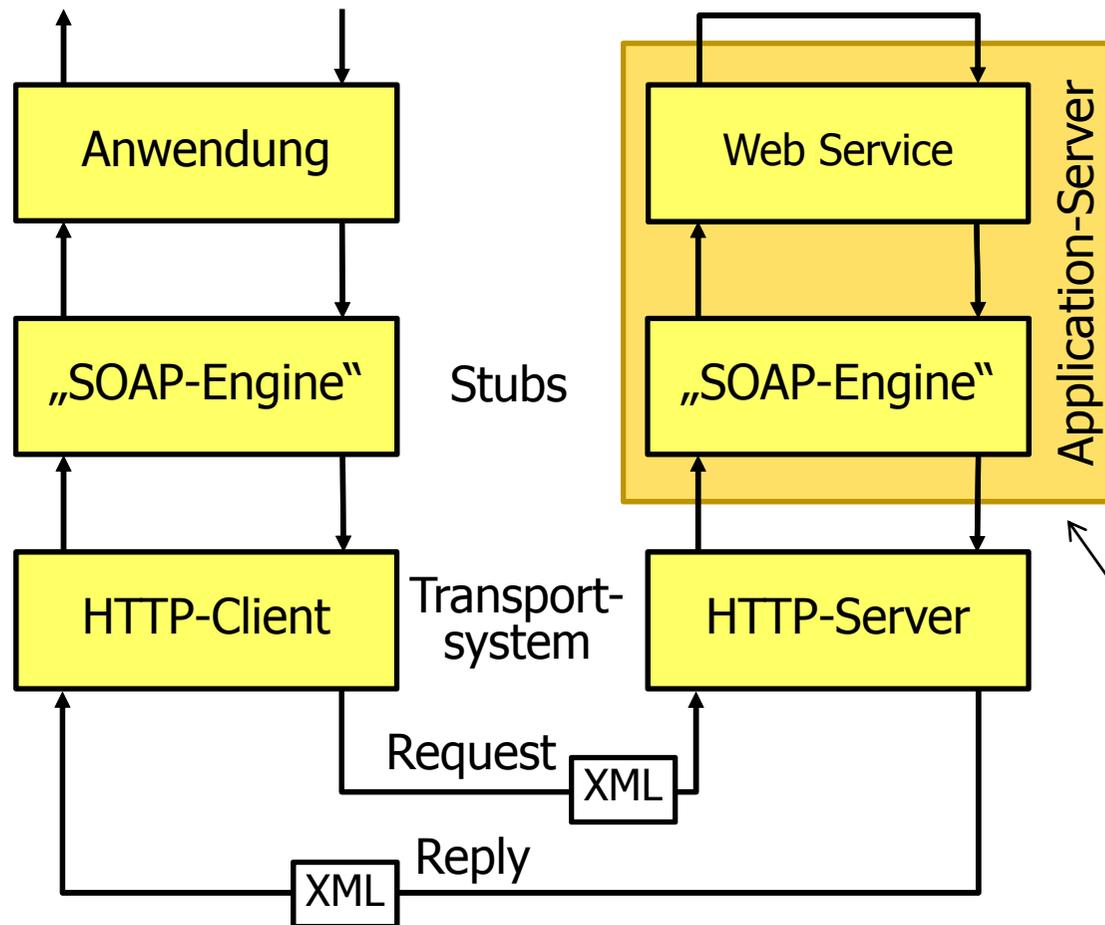
SOAP-Beispiel: Response



Nochmal: Web Services



Web Service Stubs



- **Server-Stubs** werden oft aus einer Web Service-Implementierung generiert (bottom up / „code first“)
- ...oder automatisch aus einer WSDL-Beschreibung des Interfaces (top down / „contract first“)
- **Client-Stubs** können ebenfalls aus WSDL generiert werden

Application-Server haben oft einen integrierten HTTP-Server, z.B. „Jakarta Tomcat“

vgl. frühere Folie „RPC: Implementierung“

Entwicklung von Web Service-Komponenten mit Java-IDE

- JAX-WS: Java API for XML Web Services (Beispiel: NetBeans)

The image displays two side-by-side screenshots from the NetBeans IDE. The left screenshot shows the source code of a Java class named `SimpleService`. The code includes package declarations, imports for `javax.jws` annotations, and a class definition for `SimpleService` with a public method `add`. A yellow highlight box contains the text "Erweiterung von einfachen Java-Objekten zu Web Services per Java Annotations (bottom up)", with arrows pointing to the `@WebService` and `@WebMethod` annotations in the code. The right screenshot shows the graphical user interface for adding a new operation to the `SimpleService` web service. It features a table for defining parameters and their types, and a dialog box for specifying the operation name and return type. The operation name is set to "add" and the return type is "int". A message at the bottom of the dialog indicates "Such method already exists".

```
1  /*
2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5  package ch.simple.service;
6
7  import javax.jws.WebService;
8  import javax.jws.WebMethod;
9  import javax.jws.WebParam;
10
11 /**
12  * JAX-WS Example
13  * "add" Web Service taking two Integers as input and returning an Integer
14  */
15 @WebService(serviceName = "SimpleService")
16 public class SimpleService {
17
18     /**
19     * Web service operation
20     */
21     @WebMethod(operationName = "add")
22     public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
23         return (i + j);
24     }
25 }
26
```

Parameters	Parameter Name	Output	Parameter Type	Description
	i		int	
	j		int	

Add Operation...

Name: add
Return Type: int
Parameters: i (int), j (int)

Such method already exists

Middleware-Entwicklung, die Web Services historisch voranging

1. RPC-Bibliotheken (z.B. von SUN für UNIX)

- Unterstützung des Client-Server-Paradigmas
- einfache Schnittstellenbeschreibungssprache, Stubgeneratoren
- Sicherheitskonzepte: Authentifizierung, Autorisierung, Verschlüsselung

2. Client-Server-Verteilungsplattformen (z.B. DCE)

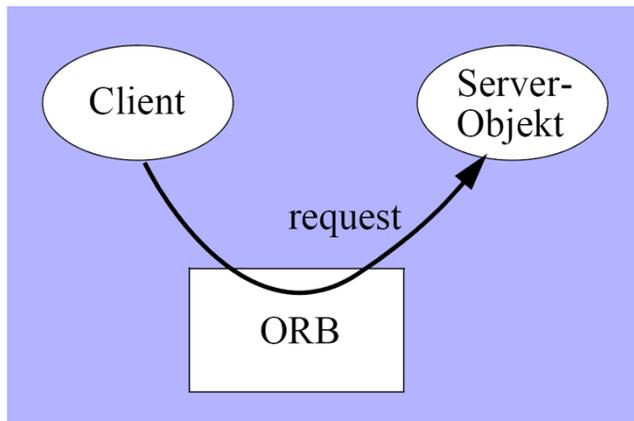
- Verzeichnisdienst, globaler Namensraum, globales Dateisystem
- Programmierhilfen: u.a. Synchronisation, Multithreading

3. Objektbasierte Verteilungsplattformen (z.B. CORBA)

- Kooperation zwischen verteilten Objekten
- objektorientierte Schnittstellenbeschreibungssprache
- „Object Request Broker“ als Vermittlungsinstanz

CORBA (Version 2.0 ca. ab 1997)

- **Common Object Request Broker Architecture**
 - **ORB** (Object Request Broker) als Vermittlungsinfrastruktur (Weiterleitung von Methodenaufrufen etc.)
 - **IDL** (Interface Description Language) mit Stub-Generatoren
 - Systemfunktionen und Basisdienste in Form von **Object Services**



Methodenaufruf unterschiedlicher Semantik:
synchron (mit Rückgabewerten; analog zu RPC)
verzögert synchron (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)
one way (asynchron: Aufrufer wartet nicht; keine Ergebnisrückgabe)

CORBA



- Ab ca. 2000 entstand der Wunsch nach einer wesentlichen **Erweiterung der CORBA-Funktionalität** aufgrund
 - neuer Anforderungen durch **E-Commerce**-Anwendungen
 - Ausbreitung des **WWW**
 - Aufkommen von **Java**
 - Aufkommen **mobiler Geräte**
- Allerdings geriet die **CORBA-Weiterentwicklung ins Stocken**
 - zu weitreichende Anforderungen → komplex / ineffizient
 - kommerzielle Implementierungen der Erweiterungen zögerlich
 - fehlende Unterstützung durch Microsoft (eigene Architektur: DCOM und .NET)
 - man versuchte, es jedem Recht zu machen (widersprüchliche Interessen, barocke Konstrukte durch Kompromisse)
 - aufkommende Konkurrenzsysteme (z.B. Web Services), die z.T. direkter und besser an die neuen Anforderungen angepasst waren

Man lese dazu auch: Michi Henning:
The rise and fall of CORBA. Commun. ACM, Vol. 51, No. 8 (Aug. 2008), 52-57

Zustandsändernde / -invariante Aufträge

- Verändern Aufträge den **Zustand des Servers**?
 - typische **zustandsinvariante Aufträge**: Anfrage bei Auskunftsdienst (z.B. Namensdienst oder Zeitservice)
 - typische **zustandsändernde Aufträge**: Schreiben bei Datei-Server
- **Idempotente Aufträge**
 - Wiederholung eines Auftrags führt zum gleichen Effekt
 - Beispiel: „Schreibe in Position 317 von Datei XYZ den Wert W“ (ist aber nicht zustandsinvariant!)
 - Gegenbeispiel: „Schreibe ans Ende der Datei XYZ den Wert W“
 - Gegenbeispiel: „Wie spät ist es?“ (ist aber zustandsinvariant!)
- Bei **Idempotenz** oder **Zustandsinvarianz** kann bei fehlgeschlagenem Auftrag (timeout beim Client) dieser problemlos erneut abgesetzt werden (→ **einfache Fehlertoleranz**)

Zustandslose („stateless“) / zustandsbehaftete („statefull“) Server

- Hält der Server **Zustandsinformation über Aufträge hinweg**?
 - z.B. (Protokoll)zustand des Clients
 - z.B. Information über frühere damit zusammenhängende Aufträge
- **Beispiel: Datei-Server**
 - ```
open("XYZ");
read;
read;
close;
```
  - In klassischen Systemen hält sich das Betriebssystem Zustandsinformation, z.B. über die Position des Dateizeigers geöffneter Dateien
- Bei **zustandslosen** Servern entfällt open/close; **jeder Auftrag muss vollständig beschrieben** sein (Position des Dateizeigers etc.)
  - zustandsbehaftete Server daher i.Allg. effizienter
  - Dateisperren bei echten zustandslosen Servern nicht einfach möglich
- Zustandsbehaftete Server können wiederholte Aufträge erkennen (z.B. Speichern von Sequenznummern) → **Idempotenz**
- **Crash eines Servers**: Weniger Probleme im zustandslosen Fall

# Sind Web-Server zustandslos?

- Beim **HTTP-Zugriffsprotokoll** wird über den Auftrag hinweg keine Zustandsinformation gehalten
  - jeder link, den man anklickt, löst eine neue „Transaktion“ aus
- Stellt z.B. ein Problem **E-Commerce-Anwendungen** dar
  - oft gewünscht: Transaktionen über mehrere Klicks hinweg und
  - Wiedererkennen von Kunden (beim nächsten Klick oder Tage später)
  - erforderlich z.B. für Realisierung von „Warenkörben“ von Kunden



# Wiedererkennung von Kunden?



- „URL rewriting“ und **dynamische Web-Seiten**
  - der Einstiegsseite eine eindeutige Identität anheften, wenn der Kunde diese erstmalig aufruft
  - diese Identität jedem link auf der Seite anheften und mit zurückübertragen
- „Cookie“ als **Context-Handle**
  - kleine Textdatei, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
  - der Server kann das Cookie später wieder lesen und damit den Kunden wiedererkennen
- Evtl. auch Wiedererkennung über **IP-Adresse**
  - aber oft Probleme bei dynamischen IP-Adressen, Proxies etc.

# Ressourcen-orientierte Architektur (ROA)

- Funktionalität wird nicht durch Services („SOA“), sondern durch (Web-) **Ressourcen** angeboten
- **Ressource?** Bezugsobjekt eines **Uniform Resource Identifiers**
  - RFC 1630 „URL“ (1994)      Implizit: „Etwas, das adressiert werden kann“
  - RFC 2396 „URI“ (1998)      *A resource can be **anything that has identity**. Familiar examples include an **electronic document**, an **image**, a **service** (e.g., "today's weather report for Los Angeles"), and a collection of other resources. **Not all resources are network "retrievable"**; e.g., human beings, corporations, and bound books in a library can also be considered resources...*
  - RFC 3986 „URI“ (2005)      *...Likewise, **abstract concepts** can be resources, such as the operators and operands of a mathematical equation, the types of a relationship...*

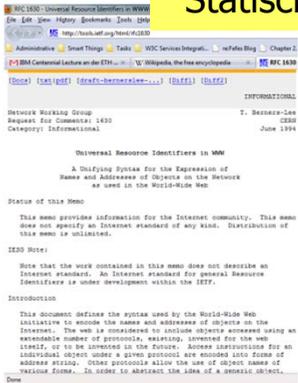
Warenkörbe/Repositories



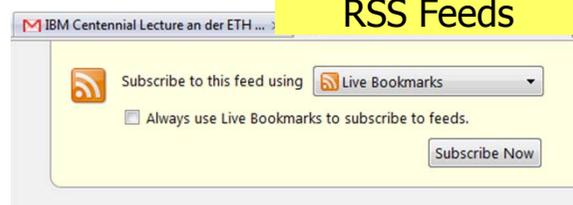
Web Dienste



Statische Websites



RSS Feeds

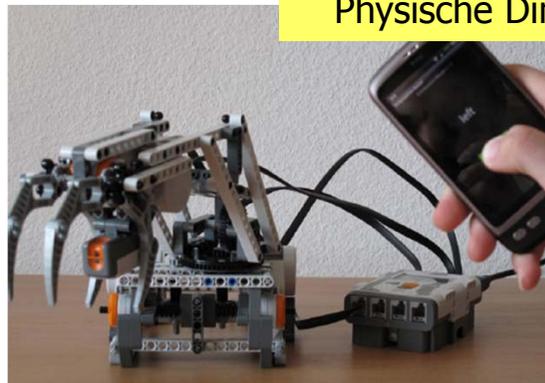


# (Web-) Ressourcen

Foren

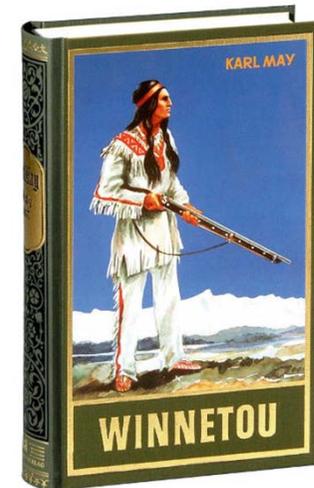


Physische Dinge



# Repräsentation – Beispiel „Buch“

- Das Buch als **abstraktes Konzept**
  - es gibt verschiedene Ausgaben, Exemplare etc.
  - identifiziert per ISBN: *ISBN-13: 978-3780200075*
- Was wir kaufen oder ausleihen ist eine **Repräsentation** des Buches
  - z.B. Hardcover, PDF, E-Book,...
  - auch ein Bild des Covers kann eine Repräsentation sein
  - oder ein maschinenlesbares XML-Dokument für das Bibliothekssystem



# REST



- REST (als **idealisierte Architektur des Web**) steht für
  - **Representational**: Nicht Ressourcen, sondern deren **Repräsentationen** werden übertragen
  - **State Transfer**: Die Übertragung löst **Zustandsübergänge** aus und verändert damit die Ressourcen
- Motivation und Entwicklung
  - **Erfolg des WWW** in technischer Hinsicht (z.B. bzgl. Skalierbarkeit) beruht auf Eigenschaften der zugrundeliegenden Protokolle und Mechanismen
  - Idealisierung dieser Architektur durch **Abstraktion von HTTP**
  - Mit REST wird des weiteren versucht, die Möglichkeiten, die das **Web** (bzw. HTTP) bietet, **optimal auszunutzen**

# Eigenschaften von REST



- **Zustandslosigkeit**
  - entschärft Crash-Problematik und Orphans
  - erlaubt Caching und bessere Skalierbarkeit
- Einheitliche und **a-priori bekannte Schnittstelle** für alle Ressourcen
  - **einheitliche Aufrufe**, z.B. GET, POST bei HTTP
  - **Adressierung** direkt durch URIs
  - **selbstbeschreibende Nachrichten**: alle benötigten Metadaten sind enthalten, z.B. im HTTP-Header
  - bekannte Repräsentationen, z.B. MIME-Typen
- Bevorzugte **Repräsentation wählbar**
  - HTML, XML, JSON,...

auch andere  
Protokolle möglich!

# Entwicklung von REST-Komponenten mit Java-IDE

- JAX-RS: Java API for RESTful Web Services (Beispiel: Eclipse)

```
ShoppingCartResource.java X
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

@Path("/cart/{cartID}")
public class ShoppingCartResource {

 @GET @Produces("text/html")
 public Response getCartContent(
 @PathParam("cartID") String cartID) {

 return Response.ok("Books in your cart: ...").build();
 }

 @POST @Consumes("application/json") @Produces("text/html")
 public Response addBookToCart(
 @PathParam("cartID") String cartID,
 @QueryParam("bookID") String bookID) {

 return Response.created(c.bookURIInCart).build();
 }
}
```

Erweiterung von einfachen Java-Objekten zu Ressourcen per Java Annotations

@Path: Pfad zur Ressource

Definition der verwendeten Media Types (@Consumes und @Produces)

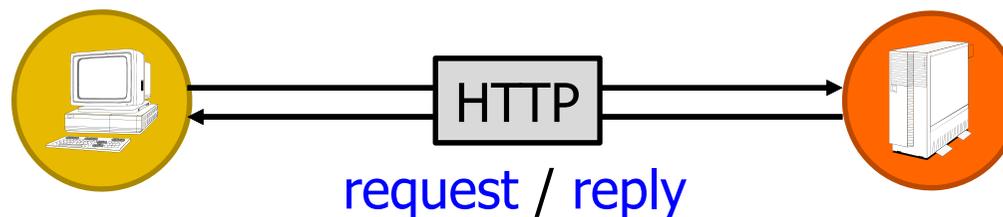
Extraktion von Parametern aus dem Request mittels:

@PathParam: z.B. {cartID}

@QueryParam: z.B. ?bookID=123

# REST-Anwendungsmodell

- „Hypermedia as the Engine of Application State“
  - Client kennt ausschliesslich die **Basis-URI des Dienstes**
  - Server leitet durch die Anwendungszustände durch Bekanntgabe von Wahlmöglichkeiten (**hyperlinks**)

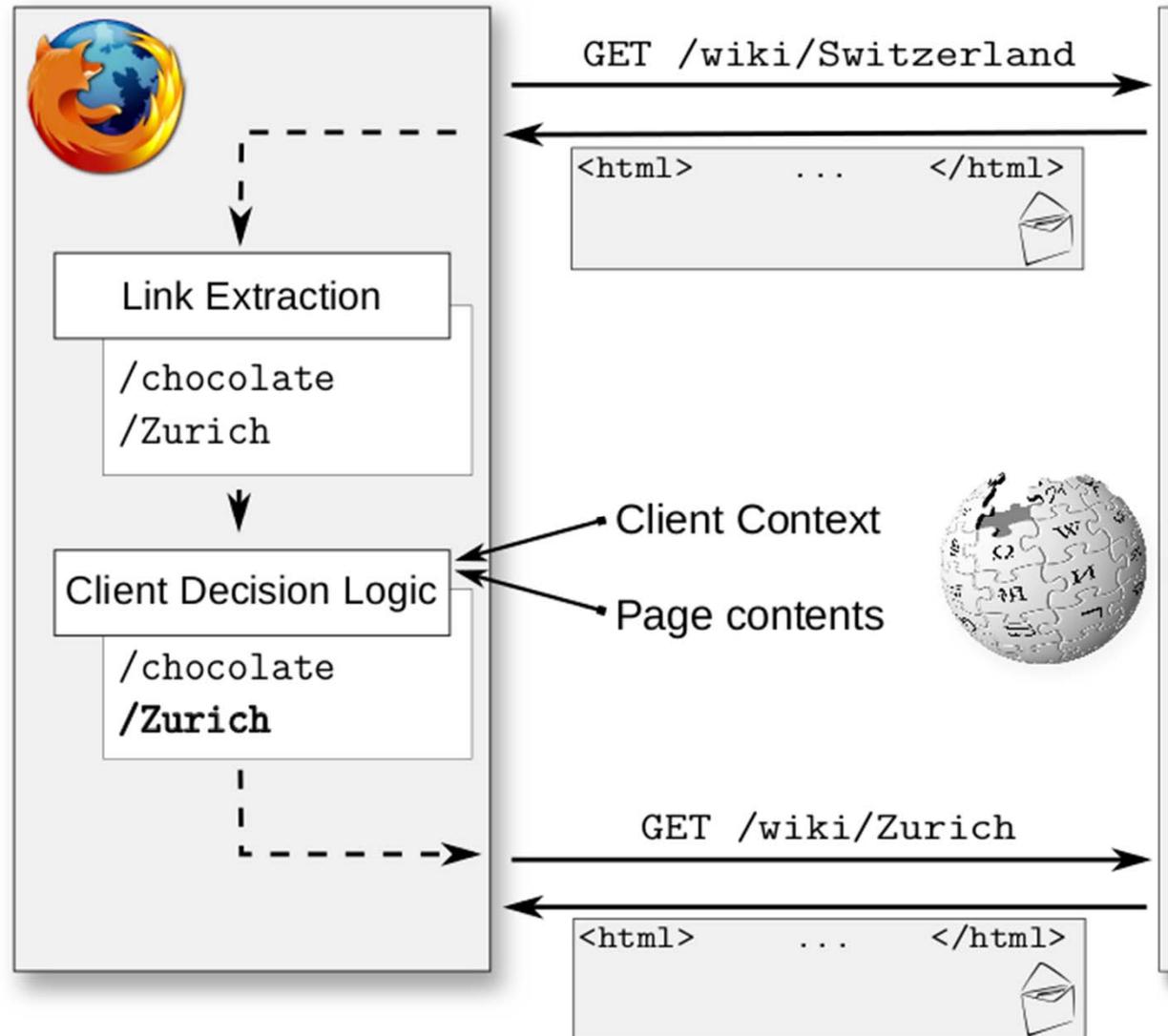


- Der **Anwendungszustand** kann beim **Client** oder beim **Server** liegen, oder auch über beide **verteilt** sein
- HTTP-Kommunikationsprotokoll selbst bleibt aber zustandslos

# Beispiel:

## Client

(ein Mensch an einem Browser)



## Server

# REST: Zustandsspeicherung

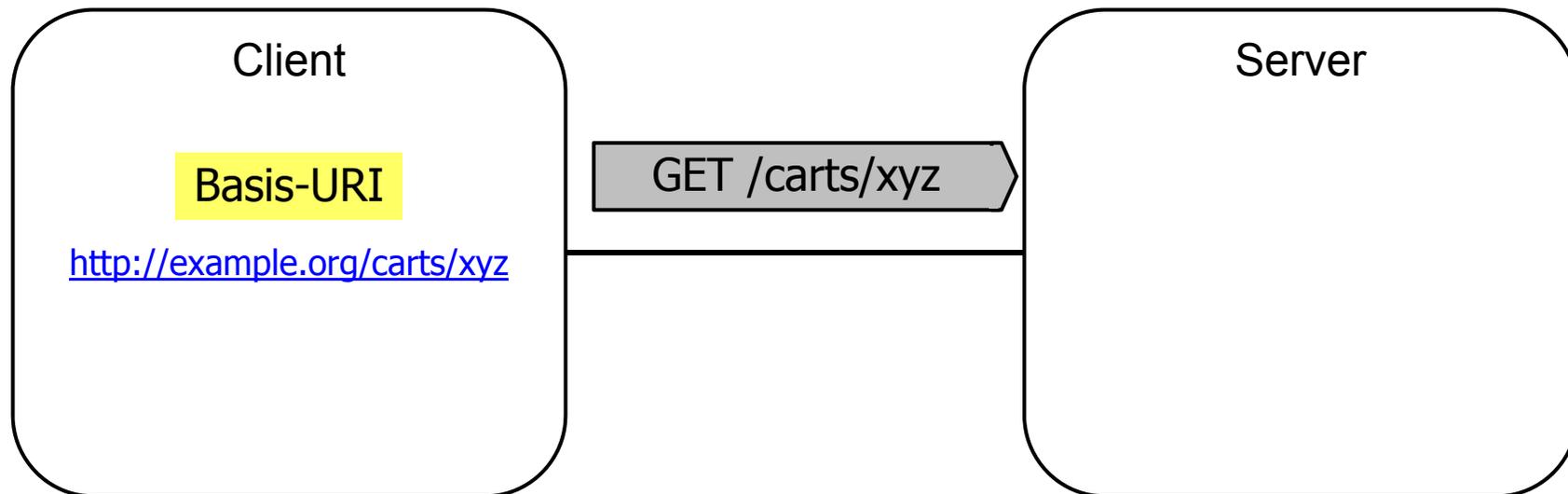
- Wenn der **Server keinen Zustand** hält, dann muss der Client alleine durch den Zustandsraum navigieren
  - Beispiel: man klickt sich durch eine **Hypertext-Struktur** (und merkt sich die früher besuchten Dokumente bzw. Zustände)
  - **Aktueller Zustand** = derzeitiges Hypertextdokument beim Client
  - Client und Server sind völlig **entkoppelt**
  - Dem Server ist es egal (und nicht bewusst), welches Hypertext-Dokument (d.h. Zustand) der Client vorher hatte
  - Der Client übergibt jeweils eine **vollständige URI**, die den Folgezustand bezeichnet
  - **Bookmarks ergeben Sinn** (sind vollständige URI, losgelöst von jeglichem Kontext)
  - **Back button im Browser ergibt Sinn**: er führt zu einem früheren (im Client gecachten) Zustand

# REST: Zustandsspeicherung

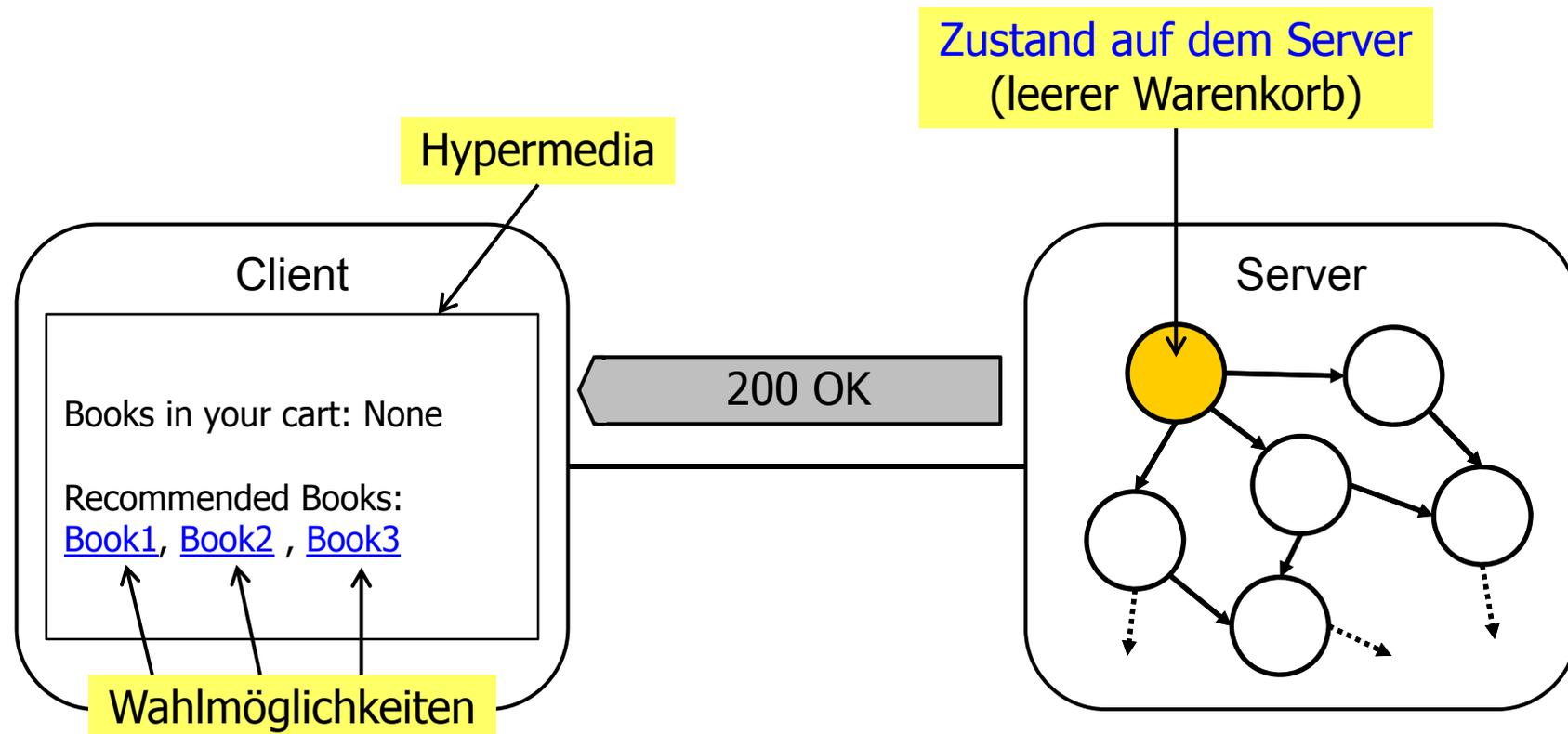


- Bei **komplexeren Anwendungen** residiert typischerweise der **Anwendungszustand beim Server**
  - z.B. Flugbuchung, wo Server mit externen Diensten kommuniziert
  - e-shop, wo der Warenkorb beim Server geführt wird
- Dann funktioniert eine **Kopie einer URI** (bookmark) später meistens nicht, weil dem Server der **Kontext** dazu fehlt
  - auch **back button** im Browser ist **problematisch**: führt zu einer früheren Zustandskopie, ohne dass der Server dies mitbekommt – Client meint fälschlicherweise, in einem gewissen Zustand zu sein, der tatsächliche Zustand wird aber auf dem Server gehalten

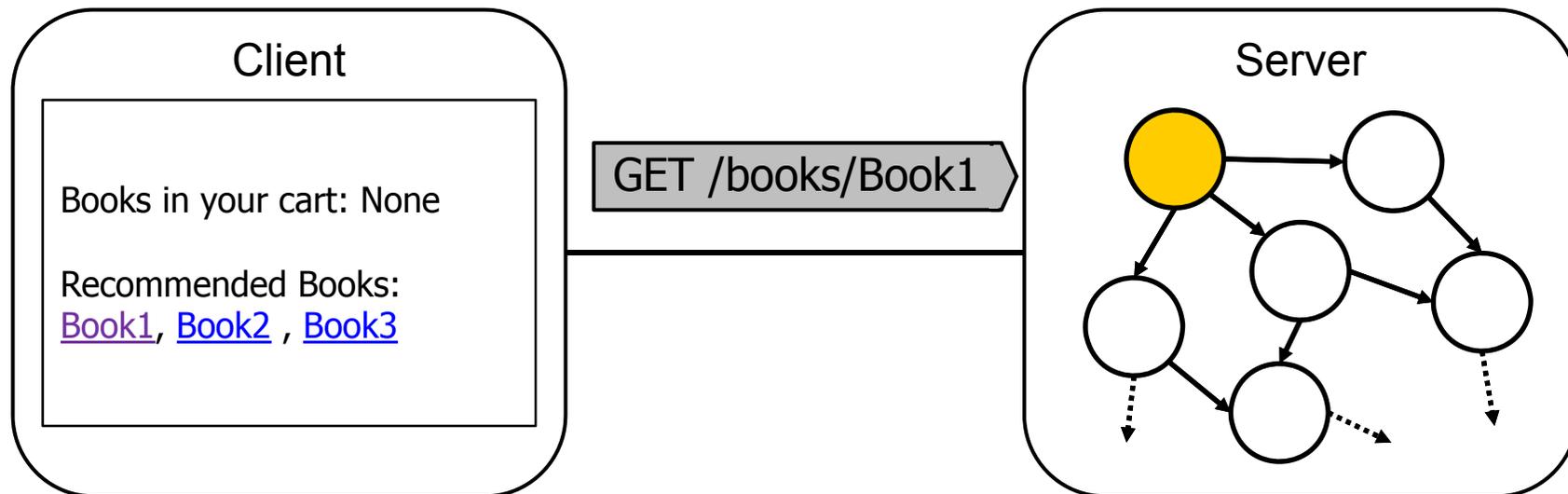
# Zustandsspeicherung beim Server



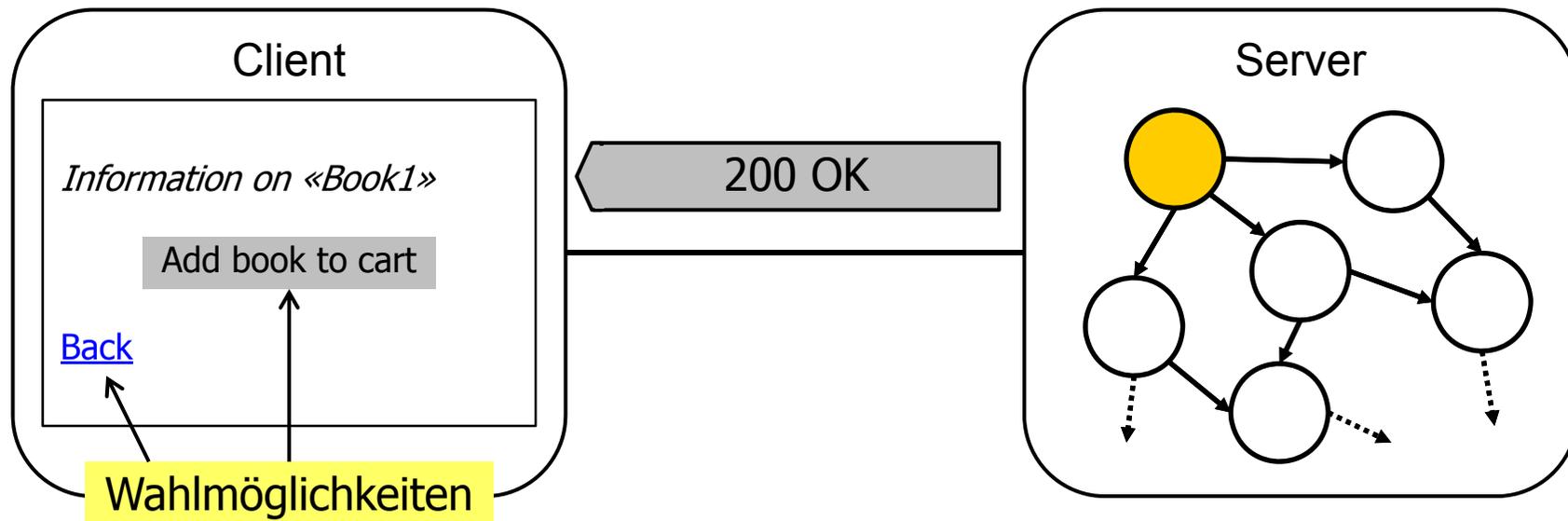
# Zustandsspeicherung beim Server



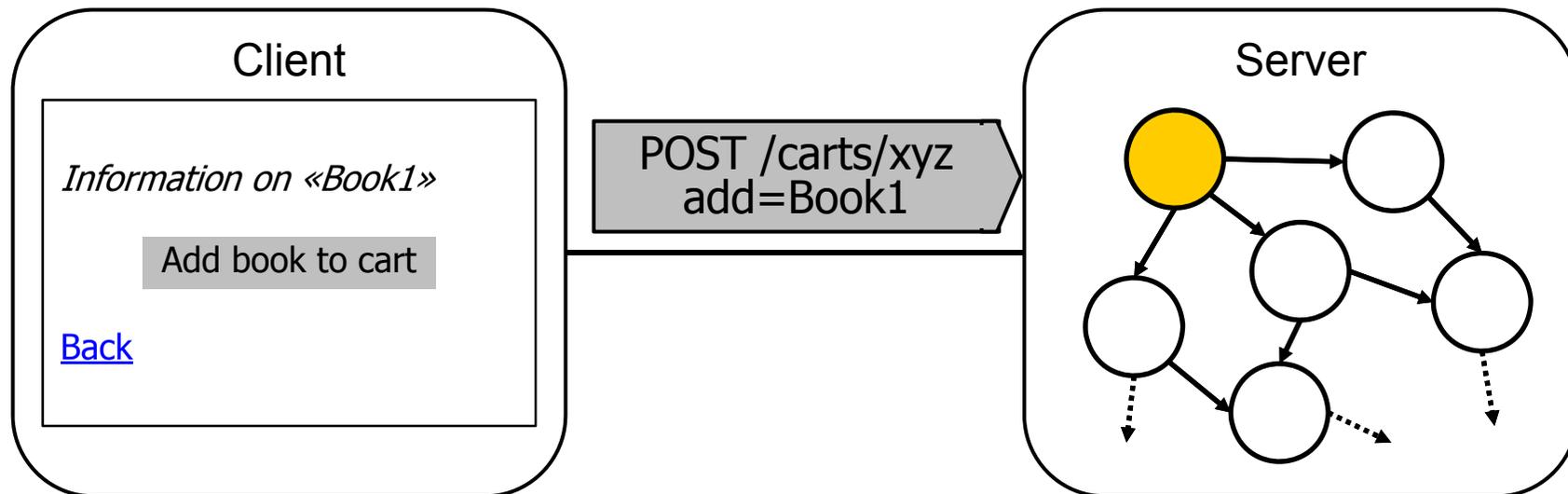
# Zustandsspeicherung beim Server



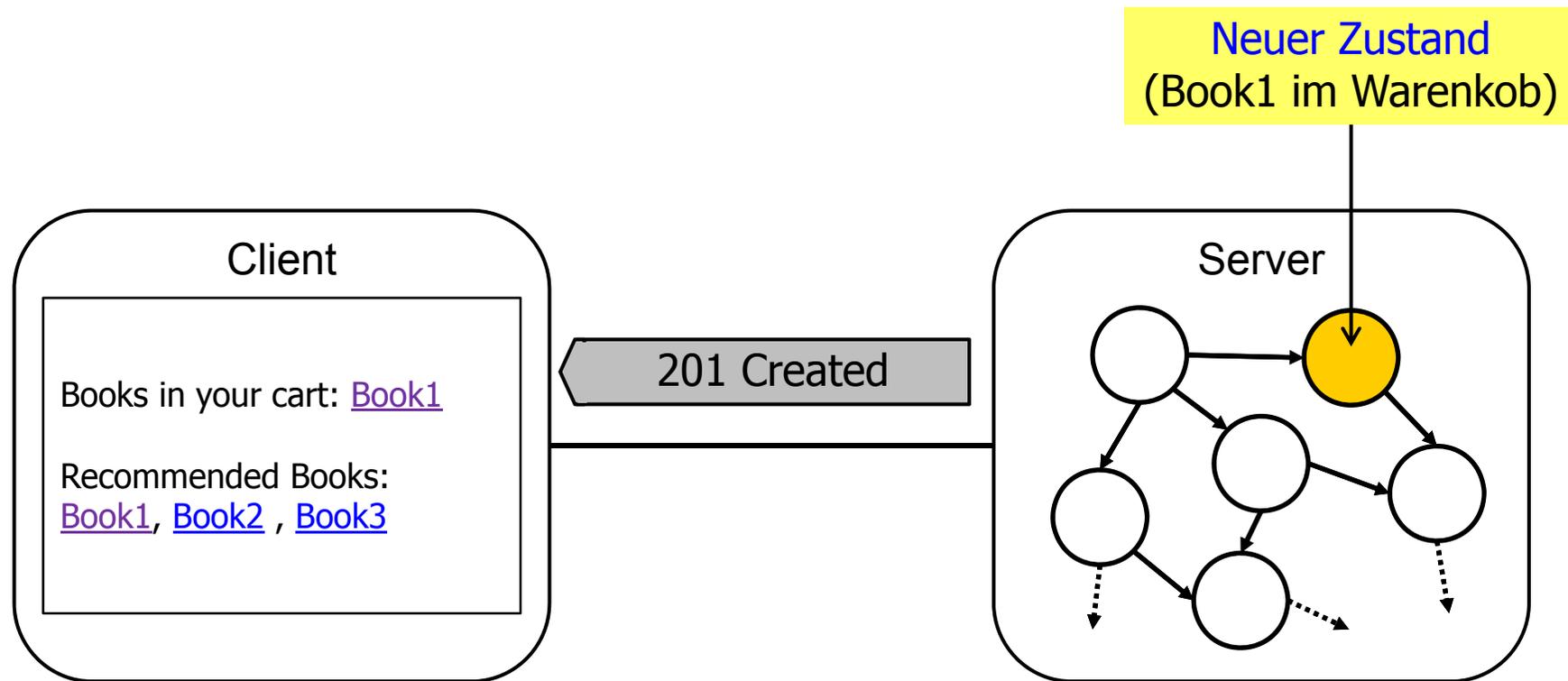
# Zustandsspeicherung beim Server



# Zustandsspeicherung beim Server



# Zustandsspeicherung beim Server



Zustand wird auf dem Server gemerkt, indem die **Ressource** `/carts/xyz/Book1` angelegt wird

# Jini



- **Infrastructure** (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications

Jini serves as an example for a number of similar platforms (UPnP, Bluetooth SDP, SLP, HAVi, Salutation, e-speak, HP Chai,...)

# Jini

- **Infrastructure** (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications



- framework of APIs with useful functions / services
- helper services (discovery, lookup,...)
- suite of standard protocols and conventions

# Jini



- **Infrastructure** (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications

- 
- services, devices, ... find each other automatically (“plug and play”)
  - dynamically added / removed components
  - changing communication relationships
  - mobility

# Jini



- **Infrastructure** (“middleware”) for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications
- **Based on Java**
  - uses RMI (Remote Method Invocation)
  - code shipping
  - requires JVM / bytecode everywhere
- **Service-oriented**
  - (almost) everything is considered a service
  - service access through mobile proxy objects

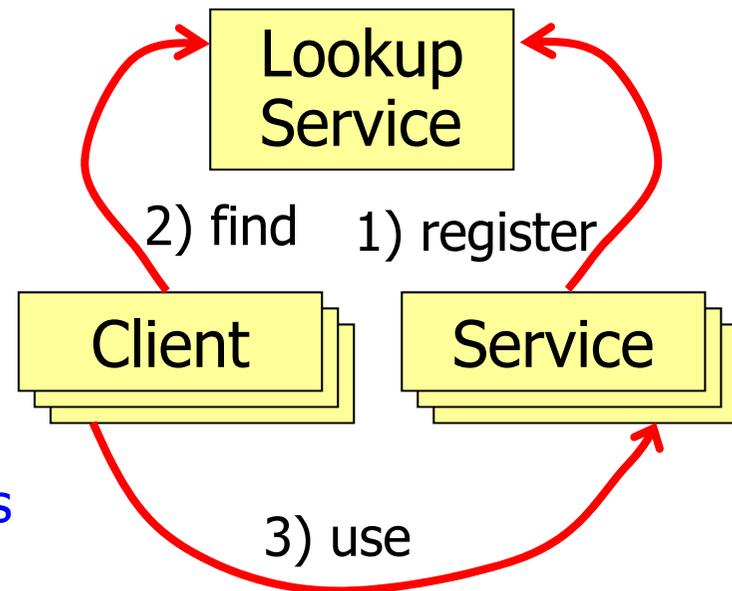
# Service Paradigm



- Almost everything relevant is a **service**
- Jini's run-time infrastructure offers mechanisms for **adding, removing, finding, and using services**
- Services are defined by **interfaces** and provide their functionality via their interfaces
  - services are characterized by their **type** and their **attributes** (e.g., "600 dpi", "version 21.1")
- Services (and service users) may "spontaneously" form a so-called **federation**

# Jini: Global Architecture

- **Lookup Service (LUS)**
  - main registry entity and brokerage service for services and clients
  - maintains information about available services
- **Services**
  - specified by Java interfaces
  - **register** together with **proxy objects** and attributes at the LUS
- **Clients**
  - know the Java interfaces of the services, but not their implementation
  - **find** services via the LUS
  - **use** services via proxy objects



# Network Centric



- Jini is based on the **network paradigm**
  - network = hardware and software infrastructure
- View is “network to which devices are connected to”, not “devices that get networked”
  - network always exists, devices and services are transient
- Jini supports **dynamic** networks and adaptive systems
  - adding and removing components or communication relations should only minimally affect other components

# Spontaneous Networking



- Objects in an open, distributed, dynamic world find each other and form a **transitory community**
  - cooperation, service usage, ...
- Typical scenario: client wakes up (device is switched on, plugged in, ...) and asks for services in its vicinity
- Finding each other and establishing a connection should be **fast, easy, and automatic**

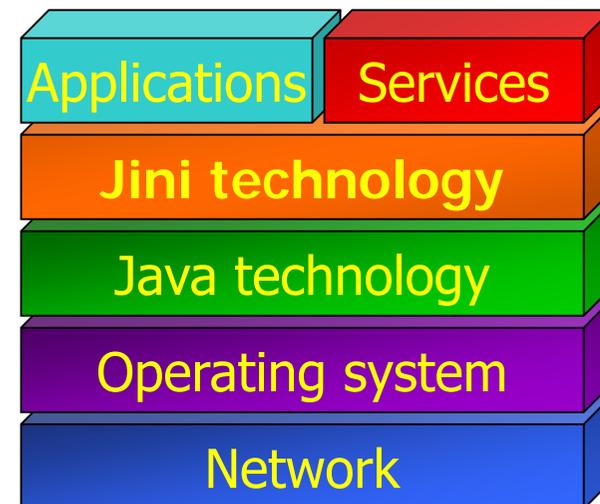
# Some Fallacies of Common Distributed Computing Systems



- The “classical” **idealistic view**...
  - the network is reliable
  - latency is zero
  - bandwidth is infinite
  - the network is secure
  - the topology is stable
  - there is a single administrator
- ...**isn't true** in practice
  - Jini acknowledges and addresses some of these issues

# Bird's-Eye View on Jini as a Middleware Infrastructure

- Jini consists of a number of **APIs**
- Is an extension to the **Java** platform dealing with distributed computing
- Is an **abstraction layer** between the application and the underlying infrastructure (network, OS)



# Jini's Use of Java



- Jini **requires JVM** (as bytecode interpreter)
  - homogeneity in a heterogeneous world
- Devices that are **not "Jini-enabled"** have to be managed by a Jini-enabled **software proxy** (somewhere in the net)

run protocols for discovery and join; have a JVM

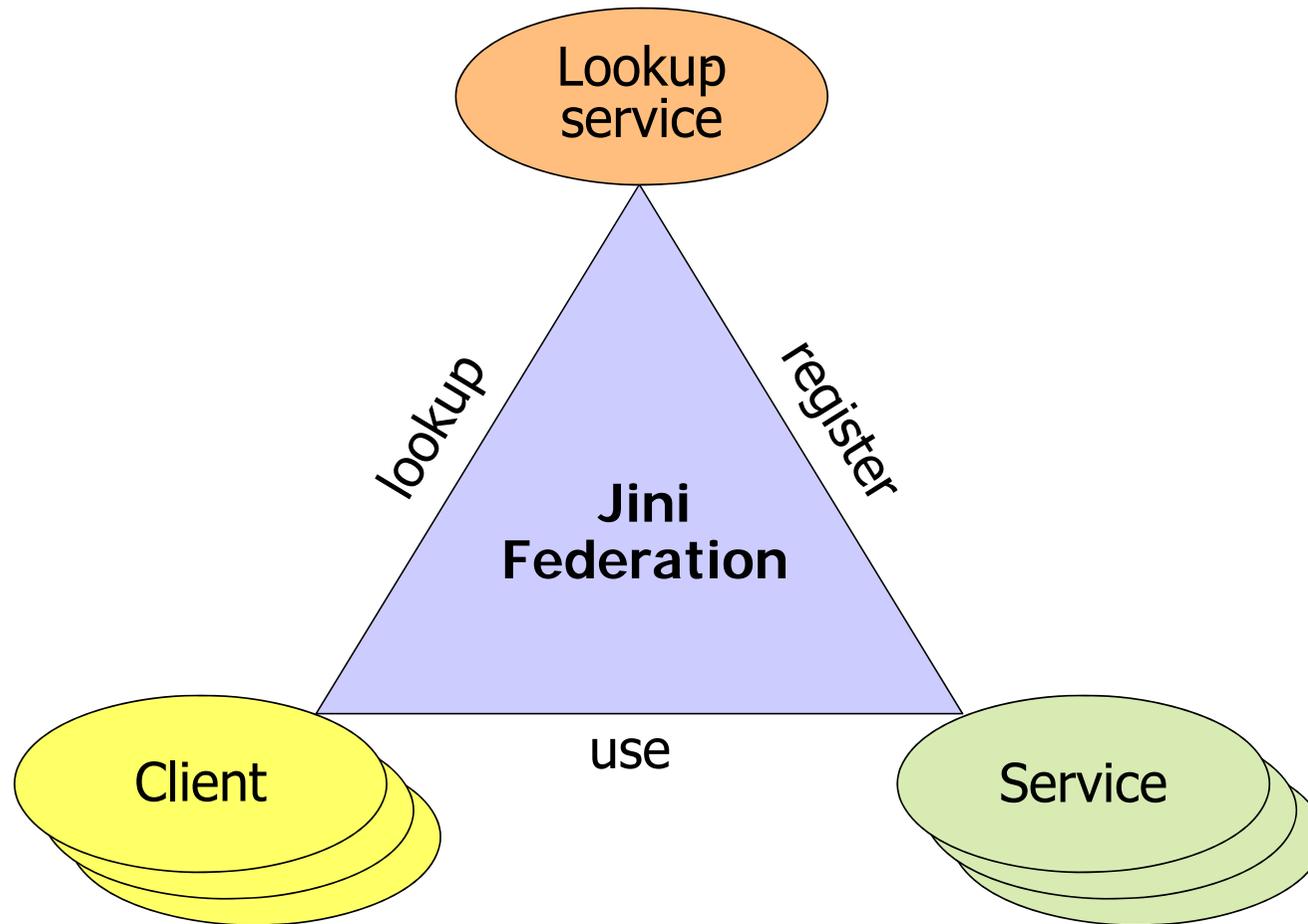


# Main Components of the Jini Infrastructure

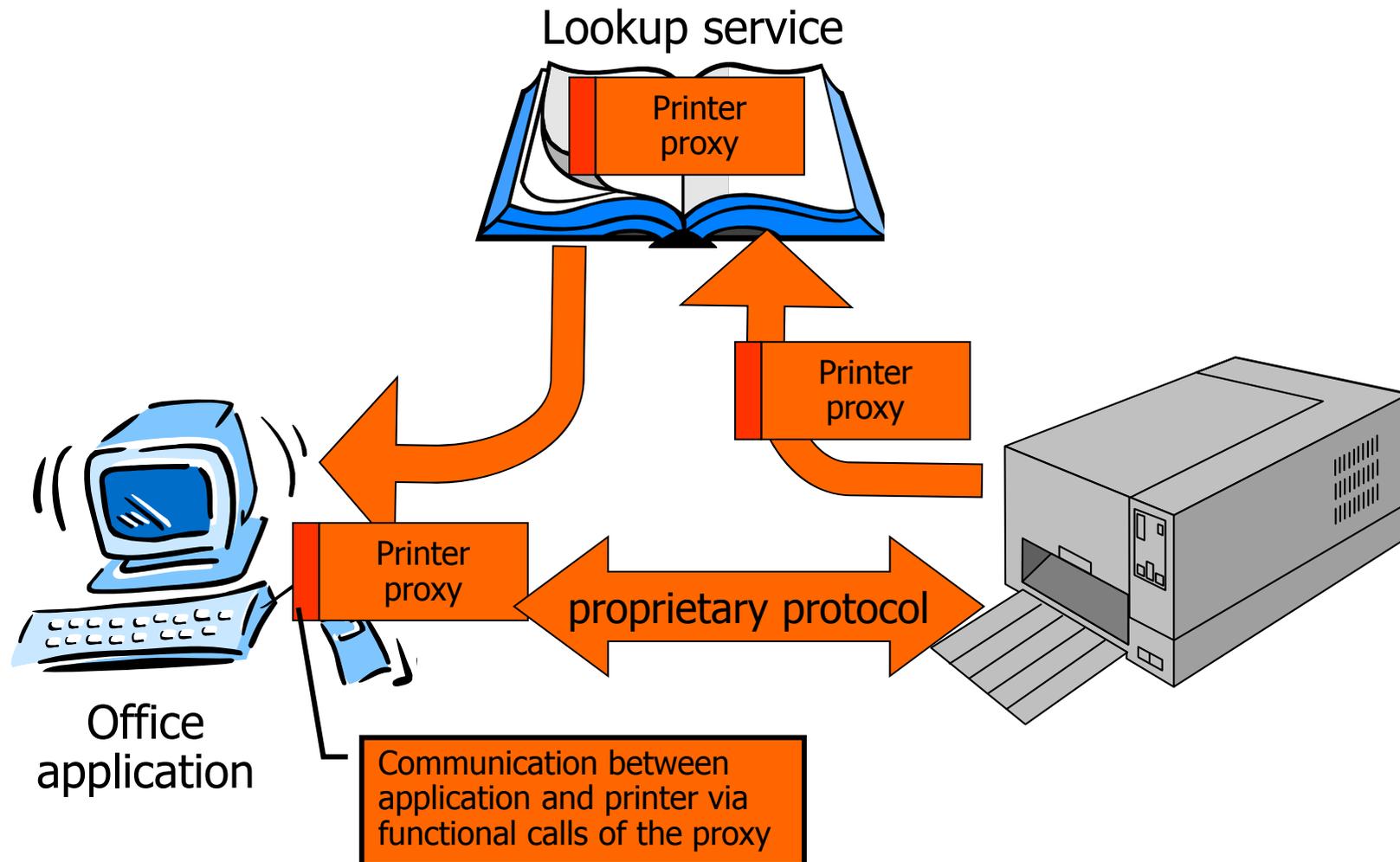


- **Lookup service (LUS)**
  - as repository / naming service / trader
- **Protocols**
  - discovery & join, lookup of services
  - based on TCP/UDP/IP
- **Proxy objects**
  - transferred from service to clients (via LUS)
  - represent the service locally at the client

# Lookup Service



# Example



# Lookup Service



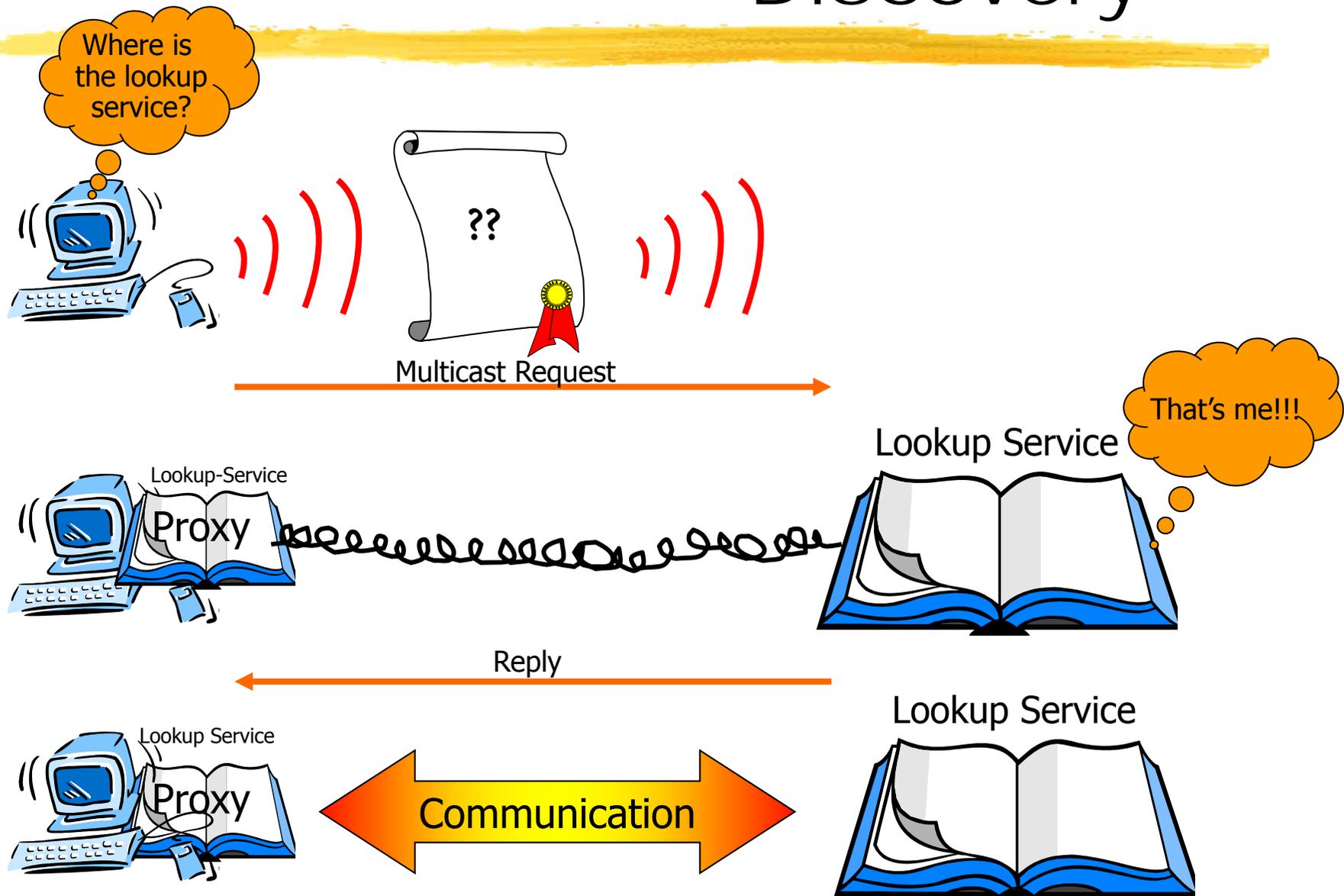
- Uses **Java RMI** for communication
  - objects („proxies“) can migrate over the network
- Stores besides the **name/address** of a service:
  - set of **attributes**
    - e.g., printer(color: true, dpi: 600, ...)
  - **proxies**, which may be complex classes
    - e.g., user interfaces
- Further possibilities:
  - responsibility can be distributed to a number of (logically separated) lookup services
  - increase robustness by running **redundant lookup services**

# Discovery: Finding a LUS



- Goal: **Find a lookup service** (without knowing anything about the network) to
  - advertise (register) an application service, or
  - find (look up) an existing application service
- **Discovery protocol:**
  - **multicast** to well-known address/port
  - lookup service replies with a serialized object (its **proxy**)
    - from then on communication with the LUS is via this proxy

# Discovery



# Multicast Discovery Protocol



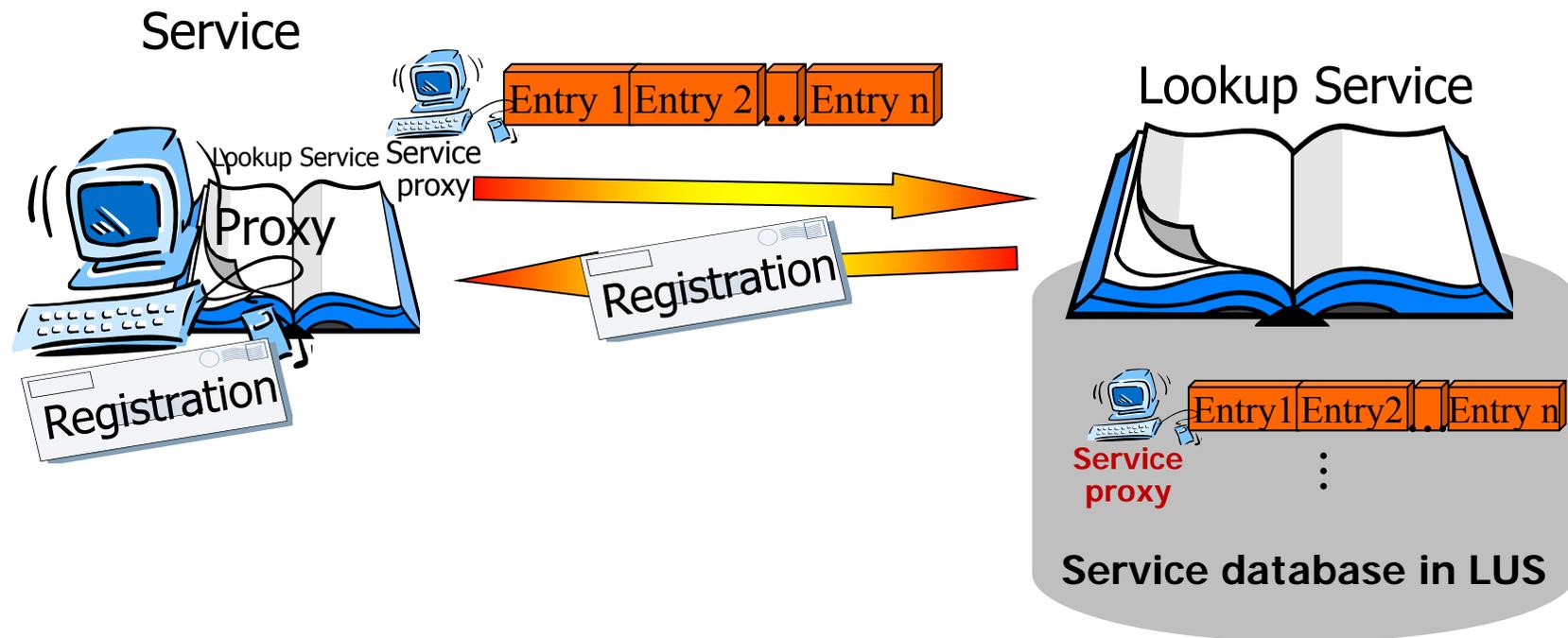
- Search for lookup services
  - no information about the host network needed
- Discovery request uses multicast **UDP** packets
  - **multicast address** for discovery (224.0.1.85)
  - default **port number** of lookup services (4160)
  - recommended **time-to-live** is 15
  - usually does not cross **subnet boundaries**
- Discovery **reply** is establishment of a **TCP connection**
  - port for reply is included in multicast request packet

# Join: Registering a Service



- Assumption: Service provider already has a proxy of the lookup service (→ discovery)
- It uses this proxy to **register its service**
- Gives to the lookup service
  - its **service proxy**
  - **attributes** that further describe the service
- Service provider can now be found and its service be used in this Jini federation

# Join



# Join: More Features



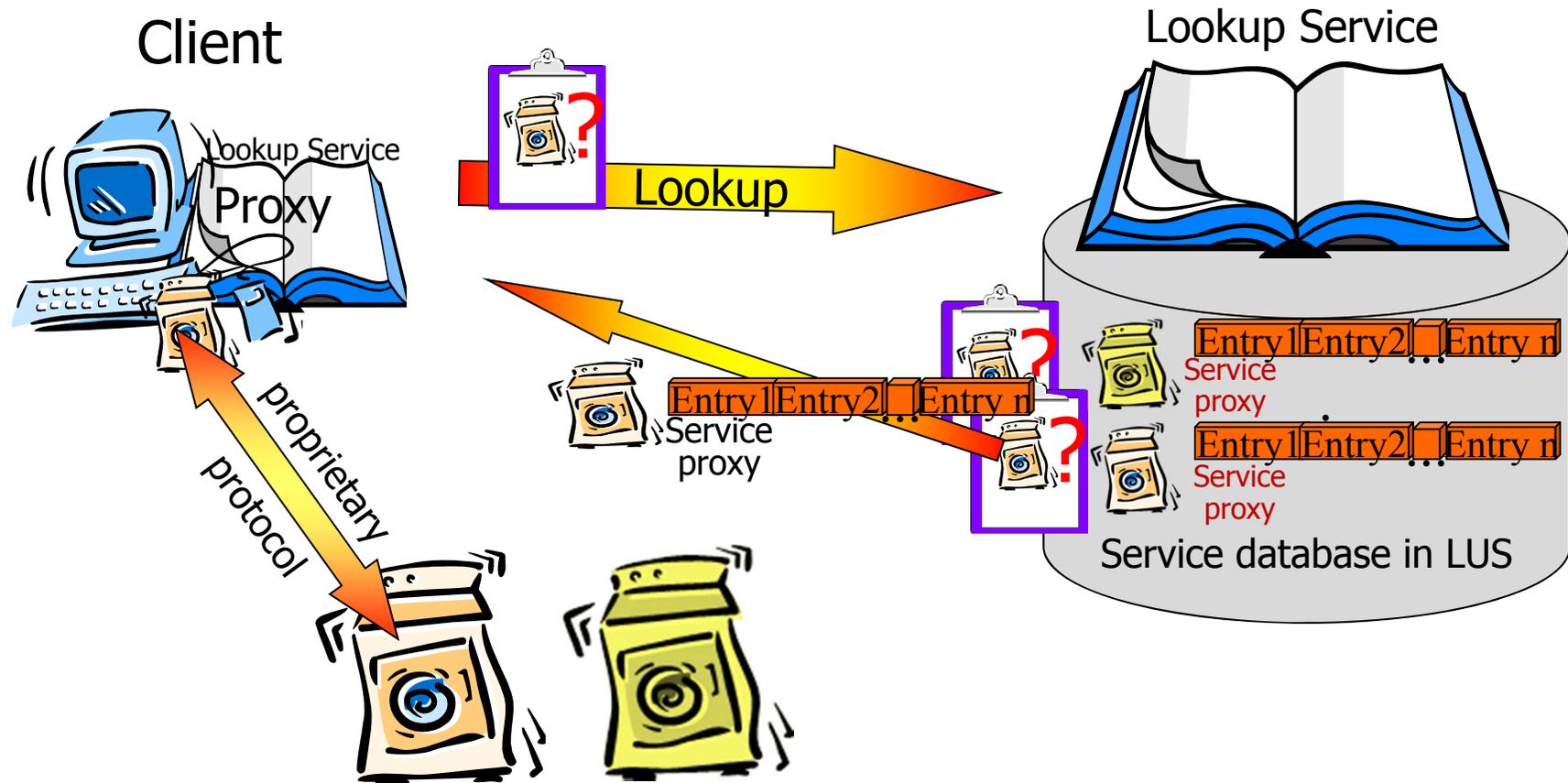
- To join, a service supplies:
  - its **proxy**
  - a **ServiceID** (a “universally unique identifier”)
  - a set of **attributes**
- Service waits a random amount of time after start-up
  - prevents packet storms after restarting a network segment
- Registration with a lookup service is bound to a **lease**
  - service has to **renew** its lease periodically

# Lookup: Searching Services



- Client creates query for lookup service
    - matching by registration number of service (**ServiceID**) and/or service **type** and/or **attributes**
    - **wildcards** are possible („null“)
  - Via its proxy at the client, the lookup service returns zero, one or more **matches** (i.e., **server proxies**)
  - Selection among several matches is done by client
- 
- Client uses the service by calling functions of the **service proxy**
  - Any proprietary protocol between service proxy and service provider is possible

# Lookup



# Proxies



- Proxy object is **stored in the LUS** upon registration
  - serialized object
  - implements the service interfaces
- Upon request, service proxy is **sent to the client**
  - client communicates with service implementation via its proxy:  
**client invokes local methods of the proxy object**
  - proxy **implementation hidden** from client

# Smart Proxies

- Parts of (or the whole) service functionality may be **executed by the proxy** at the client
  - example: when dealing with large volumes of data, it usually makes sense to **preprocess** parts of the data (e.g., compressing video data before transfer)
- Partition of service functionality depends on service implementer's choice
  - client needs appropriate **resources**



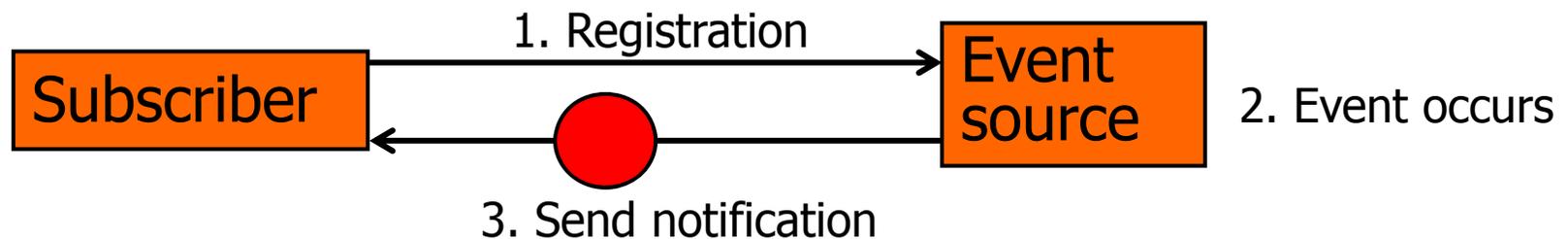
# Leases



- Leases are **contracts** between two parties
- Leases introduce a notion of **time**
  - resource usage is restricted to a certain time frame
- Repeatedly express interest in some resource:
  - I'm **still interested** in X
    - renew lease periodically
    - lease renewal can be denied
  - I **don't need** X anymore
    - cancel lease or let it expire

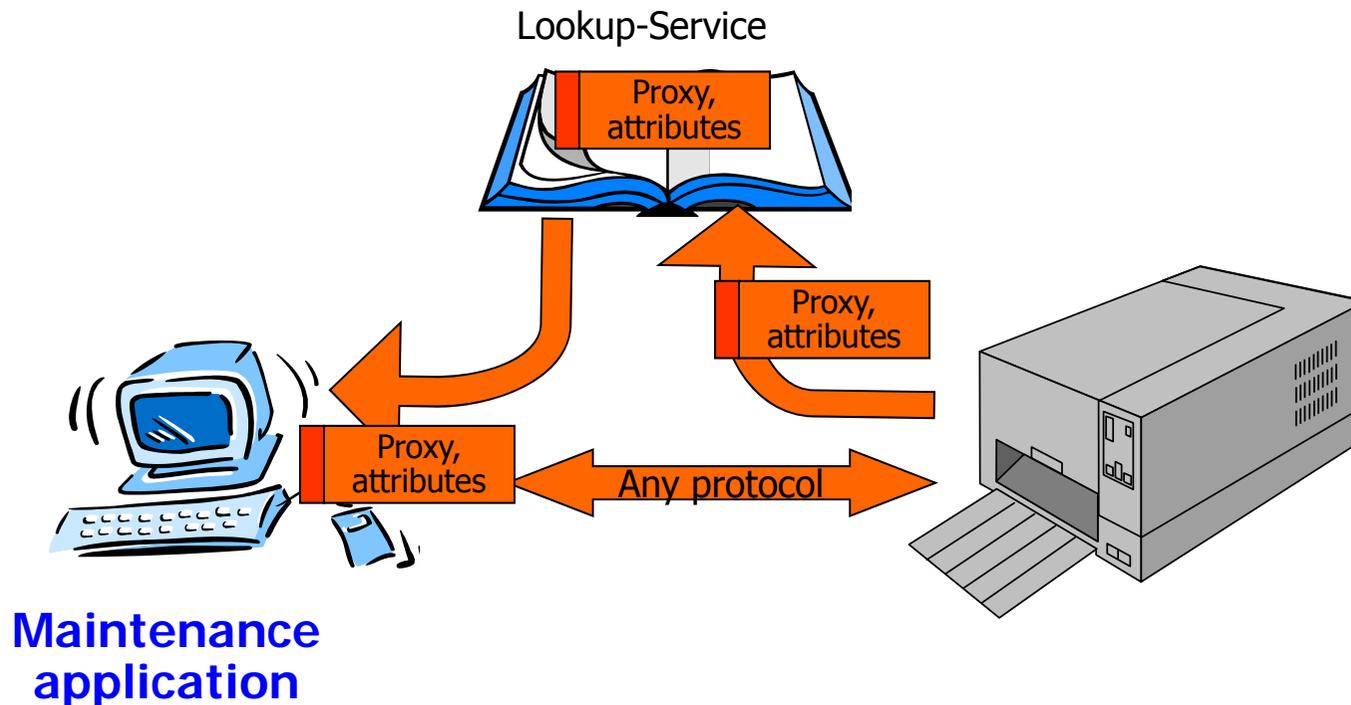
# Distributed Events

- Objects on one JVM can **register interest** in certain events of another object on a different JVM
- **“Publisher/subscriber”** model



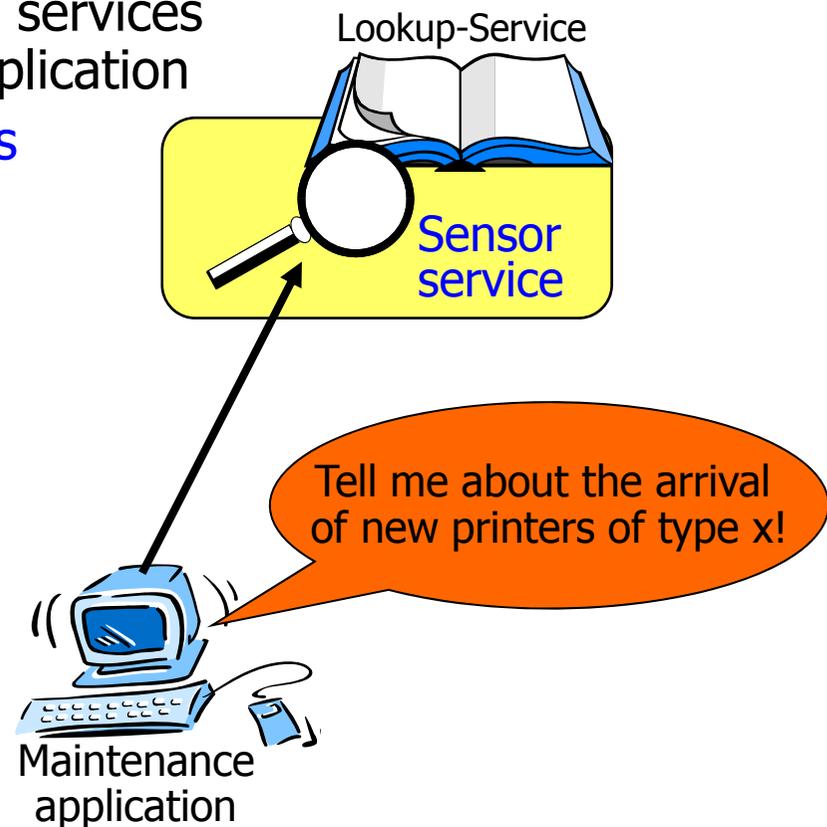
# Distributed Events - Example

- Example: printer is **plugged in**
  - printer **registers** itself with local lookup service
- **Maintenance application** wants to update software



# Distributed Events - Example

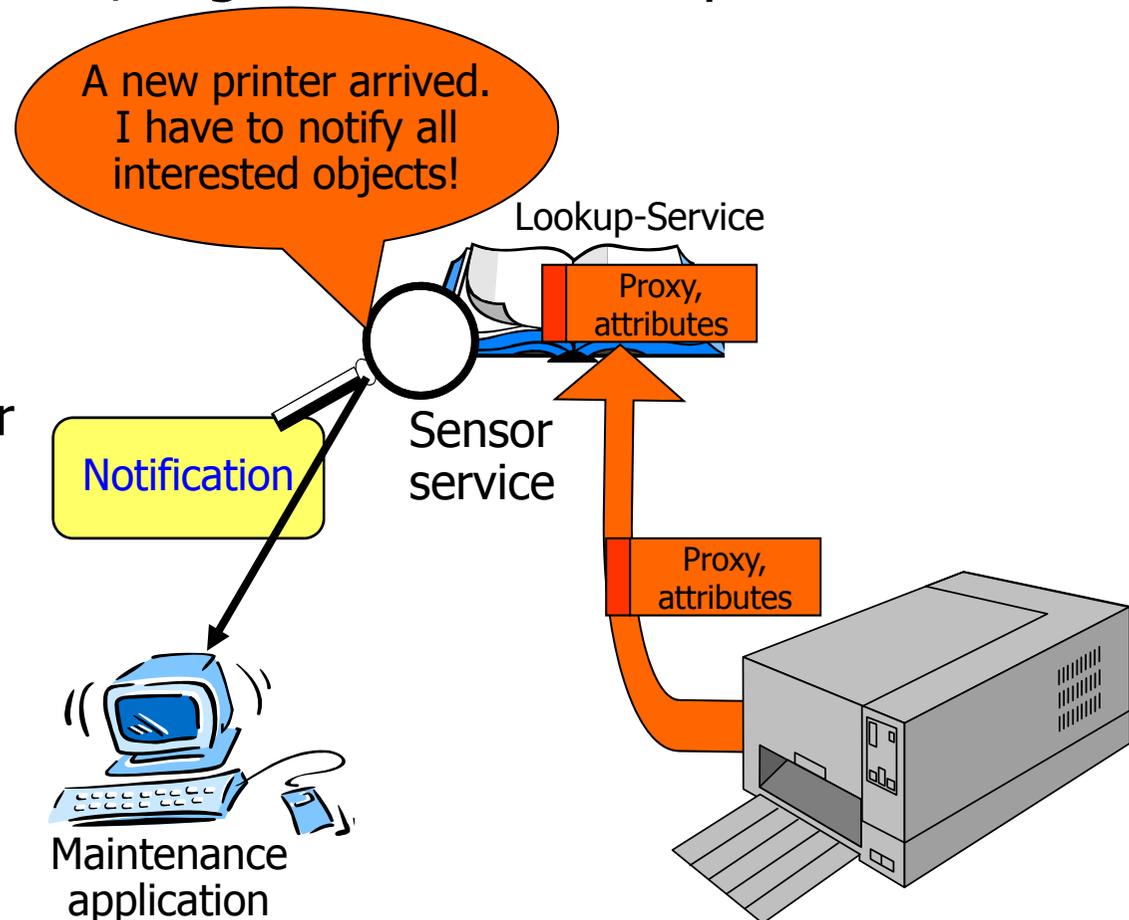
- Search for printers is “outsourced” to the lookup service
  - “**sensor service**” looks for certain services on behalf of the maintenance application
  - maintenance application **registers for events** showing the arrival of certain types of printers
  - sensor service observes the lookup service
  - **notifies application** as soon as matching printer arrives via distributed events



# Distributed Events - Example

- Example: **printer arrives**, registers with lookup service

- printer performs **discovery and join**
- sensor finds new printer in lookup service
- checks if there is an **event registration** for this type of printer
- notifies** all interested objects
- maintenance application** retrieves printer proxy and updates software



# Mehr zu allgemeinen Kommunikationsprinzipien

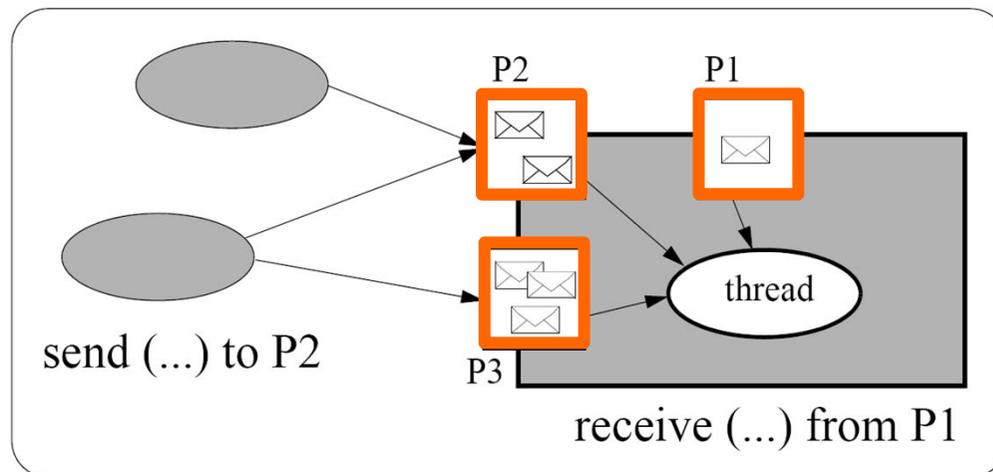


## Im Folgenden:

- Port-Konzept
- Kommunikationskanäle
- Ereigniskanäle
- Timeouts bei der Kommunikation
- Broadcast / Multicast

# Das Port-Konzept

- **Port** = adressierbarer **Kommunikationsendpunkt**, der die interne Struktur eines Nachrichtenempfänger abkapselt
- Ein Prozess kann mehrere (evtl. typisierte) Ports haben

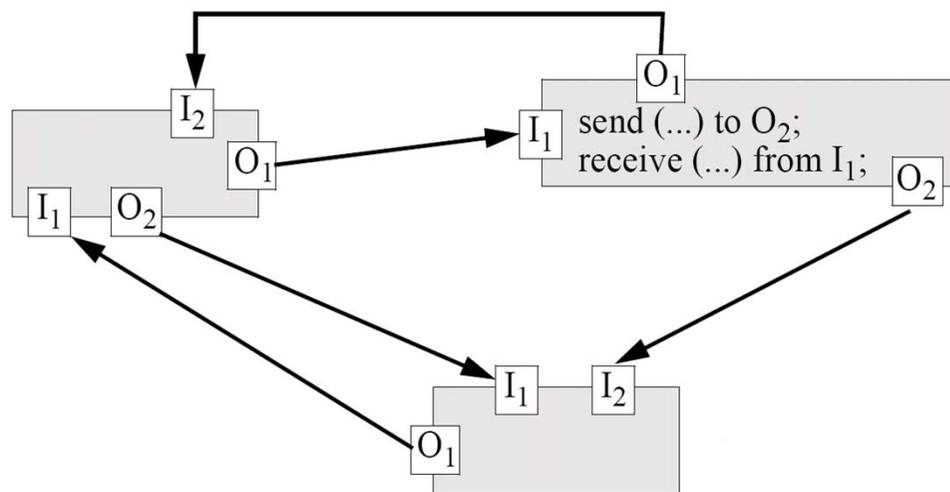


- Manchmal bilden Ports **Stauräume** („message queues“) für Nachrichten
- Manchmal können Ports **dynamisch** gegründet oder auch **geschlossen / geöffnet** werden

- Neben **Eingangsports** („In-Port“) sind manchmal auch **Ausgangsports** („Out-Port“) möglich

# Kommunikationskanäle

- **Kanäle**, z.B. eingerichtet mit Ports als Endpunkten
  - dazu je einen In- und Out-Port miteinander **verbinden**

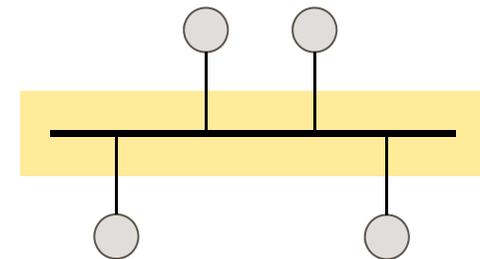


- Alternativ: **Kanäle benennen** und etwas *auf den Kanal senden* bzw. von ihm lesen
- Evtl. **Broadcast-Kanäle**: Nachricht geht an alle angeschlossenen Empfänger

- Flexibilität durch **Umkonfiguration** der Verbindungsstruktur
  - eigentlicher Adressat wird den Prozessen **verborgen** („virtualisiert“)

# Ereigniskanäle für autonome Software-Komponenten

- Kooperierende **autonome Software-Komponenten**
  - nicht notwendigerweise geographisch weit verteilt
  - mit i.Allg. getrennten Lebenszyklen
  - **anonym**: kennen nicht die Identität der anderen
  - **Auslösen** von „Ereignissen“ durch Sender
  - **Reagieren** auf **Ereignisse** beim Empfänger
- Ereigniskanal als „**Softwarebus**“
  - agiert als Zwischeninstanz und verknüpft die Komponenten
  - **registriert** Interessenten (vgl. LUS)
  - **Dispatching** eingehender Ereignisse
  - evtl. **Puffern**, **Filtern**, Umlenken von Ereignissen



# Ereigniskanäle (2)



## ■ Probleme

- Ereignisse können „jederzeit“ ausgelöst werden, werden von Empfängern aber i.Allg. nicht jederzeit entgegengenommen (→ **Pufferung?**)
- falls Komponenten nicht lokal, sondern **geographisch verteilt** sind  
→ „übliche“ Probleme nachrichtenbasierter Kommunikation:  
Verzögerungen, evtl. verlorene Ereignisse, falsche Reihenfolge,...

## ■ Beispiele

- Microsoft-Komponentenarchitektur (**.NET** etc.)
- „Distributed Events“ bei **JavaBeans** und **Jini**  
(event generator bzw. remote event listener)

# Zeitüberwacher Nachrichteneingang

- Idee: **Receive soll max. eine gewisse Zeit lang blockieren**
  - z.B. über Rückgabewert abfragen, ob Kommunikation geklappt hat oder aber der **Timeout** zugeschlagen hat
  - Timeout-Wert adäquat setzen (oft schwierig)
- Im Timeout-Fall geeignete Recovery-Massnahmen treffen oder **Exception** auslösen
- Verwendung bei:
  - **Echtzeitprogrammierung**
  - Aufheben von **Blockaden im Fehlerfall** (z.B. bei abgestürztem Kommunikationspartner)
- Timeout evtl. auch beim **blockierenden Senden** sinnvoll



# Zeitüberwacher Nachrichteneingang (2)

- Sprachliche Einbindung z.B. so:

**receive ... delay t**

{...}

**else**

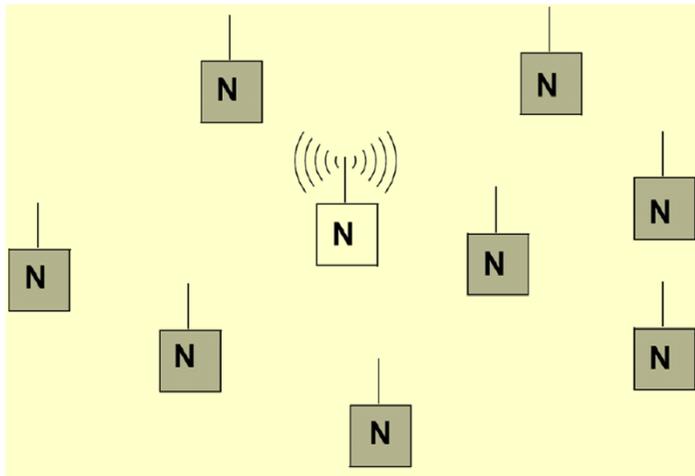
{...}

**end**

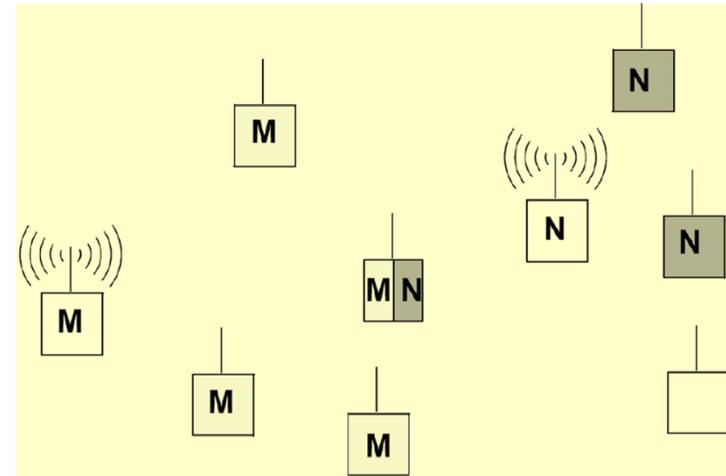
Wird nach mindestens t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen

- Beachte: Es wird **mindestens so lange** auf Kommunikation **gewartet** – danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Ablaufs verstreichen
- Was könnte „delay 0“ bedeuten? Ist das sinnvoll?

# Gruppenkommunikation (Broadcast / Multicast)



**Broadcast:** Senden an die Gesamtheit aller Teilnehmer



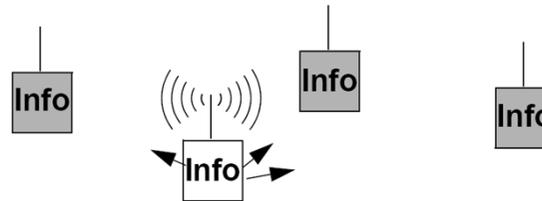
**Multicast:** Senden an eine Untergruppe aller Teilnehmer

- **Multicast** entspricht Broadcast bezogen auf die **Gruppe**
  - verschiedene Gruppen können sich evtl. **überlappen**
  - jede Gruppe hat eine **Multicastadresse**

# Anwendungen von Gruppenkommunikation

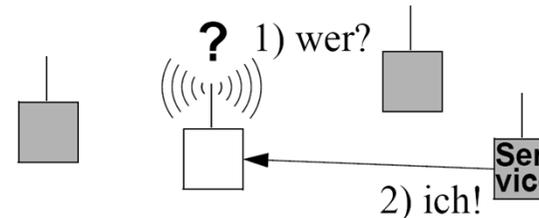
- Informieren

- z.B. Newsdienste



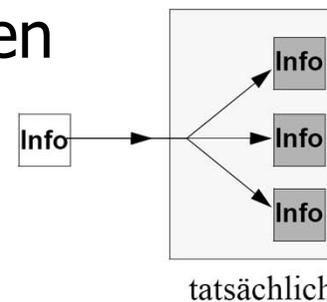
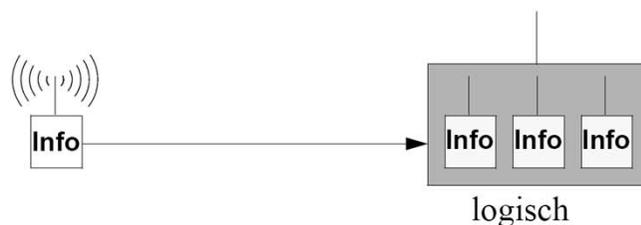
- Suchen

- z.B. Finden von Objekten und Diensten



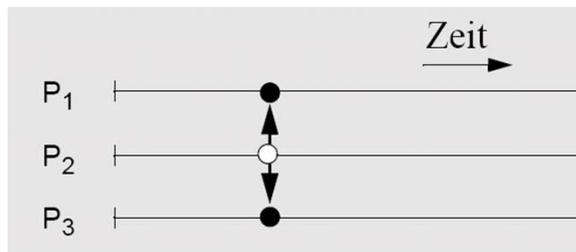
- „Logischer Unicast“

- an replizierte Komponenten



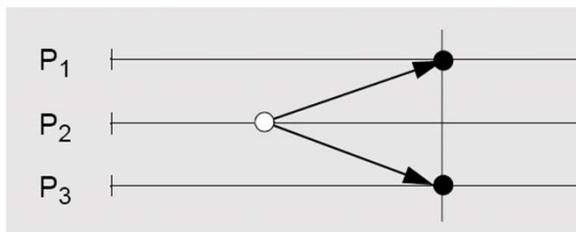
Typische Anwendungs-  
klasse von Replikation:  
Fehlertoleranz

# Gruppenkommunikation – idealisierte Semantik



## 1. Modellhaftes Vorbild: Speicherbasierte Kommunikation

- augenblicklicher „Empfang“
- vollständige Zuverlässigkeit (kein Nachrichtenverlust etc.)



## 2. Idealisierte Sicht bei nachrichtensbasierter Kommunikation:

- gleichzeitiger Empfang
- vollständige Zuverlässigkeit

- In verteilten Systemen ist aber beides **nicht realistisch** (2. kann allerdings in der Praxis bei Vorhandensein einer globalen Zeit evtl. durch eine „Sperrfrist“ als Zeitpunkt in der Zukunft erreicht werden)

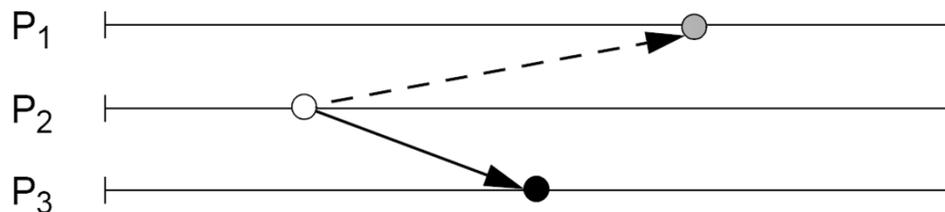
# Gruppenkommunikation – tatsächliche Situation



- **Zugrundeliegendes Medium (Netz) oft nicht multicastfähig**
  - LANs höchstens innerhalb von Teilstrukturen; WLAN als Funknetz a priori anfällig für Übertragungsstörungen
  - multicastfähige Netze sind typischerweise nicht verlässlich (keine Empfangsgarantie)
- Daher typischerweise „Simulation“ von Multicast durch ein Protokoll aus vielen **Punkt-zu-Punkt-Einzelnachrichten**
  - z.B. Multicast-Server, der die Information an alle Empfänger einzeln weiterverteilt

# Gruppenkommunikation – tatsächliche Situation (2)

- **Nachrichtenkommunikation entspricht nicht dem „Ideal“**
  - nicht-deterministische **Zeitverzögerung**  
→ Empfang zu unterschiedlichen Zeiten
  - **keine garantierte Zuverlässigkeit** der Nachrichtenübermittlung



- **Ziel von Broadcast- / Multicast-Protokollen**
  - möglichst gute **Approximation der Idealsituation**
- **Hauptproblem bei der Realisierung von Broadcast also:**
  1. **Zuverlässigkeit**
  2. **garantierte Empfangsreihenfolge**

Das soll dem Problem nicht-deterministischer Nachrichtenverzögerungen begegnen

# Typische Fehlerfälle bei der Gruppenkommunikation



- Während des Protokolls: **Verlust von Einzelnachrichten** oder **Ausfall des Senders**
  - Nachrichten können aus unterschiedlichen Gründen verloren gehen (z.B. Netzüberlastung, Empfänger hört gerade nicht zu, Hilfsprozess in der Kommunikationsinfrastruktur abgestürzt,...)
- Problem dann: **Empfänger sind sich nicht einig**, da manche, aber nicht alle, informiert werden
  - Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche **Folgeprobleme** sein (da wäre es manchmal besser, gar kein Prozess hätte die Nachricht empfangen!)
  - **partielle Abhilfe** durch **aufwändigere Protokolle** und mehr Redundanz

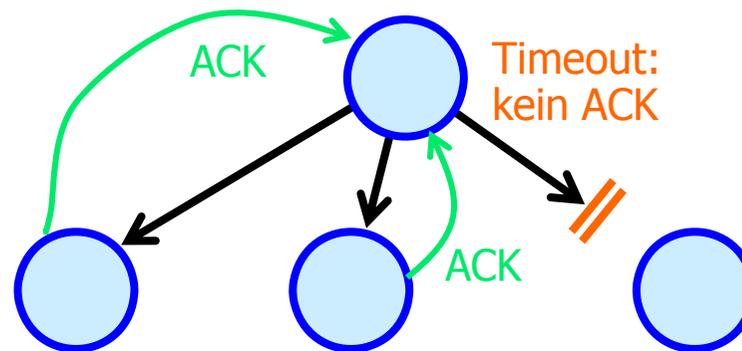
# „Best Effort“ Broadcast



- **Euphemistische Bezeichnung**, da keine extra Anstrengung
  - typischerweise einfache Realisierung ohne Ack etc.
- **Keinerlei Garantien**
  - unbestimmt, wie viele / welche Empfänger erreicht wurden
  - unbestimmte Empfangsreihenfolge
- Allerdings **effizient** (im Erfolgsfall)
- Geeignet für die Verbreitung **unkritischer Informationen**
  - z.B. Informationen, die Einfluss auf die Effizienz oder Qualität der Anwendung haben, nicht aber die Korrektheit in Frage stellen
- Evtl. als **Grundlage zur Realisierung höherer Protokolle**
  - wenn Fehlerfall selten → aufwändiges Recovery auf höherer Ebene tolerierbar

# „Reliable“ Broadcast

- Ziel: Unter gewissen (welchen?) Fehlermodellen einen „möglichst zuverlässigen“ Broadcast-Dienst realisieren



- 1. Idee: **Quittung** („positives Acknowledgement“: ACK) für jede Einzelnachricht
  - im Verlustfall z.B. einzeln nachliefern oder (z.B. bei broadcastfähigem Medium) einen zweiten Broadcast-Versuch (→ Duplikaterkennung!)
  - viele ACKs → teuer, Belastung des Senders (→ schlechte Skalierbarkeit)

# „Reliable“ Broadcast mit negativen Acknowledgements („NACK“)

- Alle broadcasts werden vom Sender **nummeriert**
  - Empfänger erkennt so eine **Lücke** beim nächstem Empfang
  - Dann wird ein **NACK** bzgl. fehlender Nachricht gesendet
  - Fehlende Nachricht wird **nachgeliefert**
    - Sender muss daher Kopien von Nachrichten aufbewahren (wie lange?)
  - Gelegentl. Broadcast einer „**Nullnachricht**“ ist evtl. sinnvoll
    - → schnelles Erkennen von Lücken
  - Kombination von **ACK / NACK** mag sinnvoll sein
- 
- **Fehlermodell?** Verfahren hilft gegen **Verlust von Nachrichten**, aber nicht gegen den **Crash** des Senders „mittendrin“

# Ein weiterer Reliable-Broadcast-Algorithmus

- **Zweck:** Jeder nicht gecrashte und zumindest indirekt erreichbare Prozess soll die Broadcast-Nachricht erhalten
- **Voraussetzung:** zusammenhängendes „gut vermaschtes“ Punkt-zu-Punkt-Netz

## Denkübungen:

- Bidirektionale Kommunikationskanäle notwendig?
- Wie effizient ist das Verfahren (Anzahl der Einzelnachrichten)?
- Wie fehlertolerant? (Wie viel darf kaputt sein / verloren gehen...?)

## Sender $s$ : Realisierung von broadcast( $N$ )

- $send(N, s, sequ\_num)$  an alle Nachbarn
- $sequ\_num ++$

## Empfänger $r$ : Realisierung des Nachrichtenempfangs

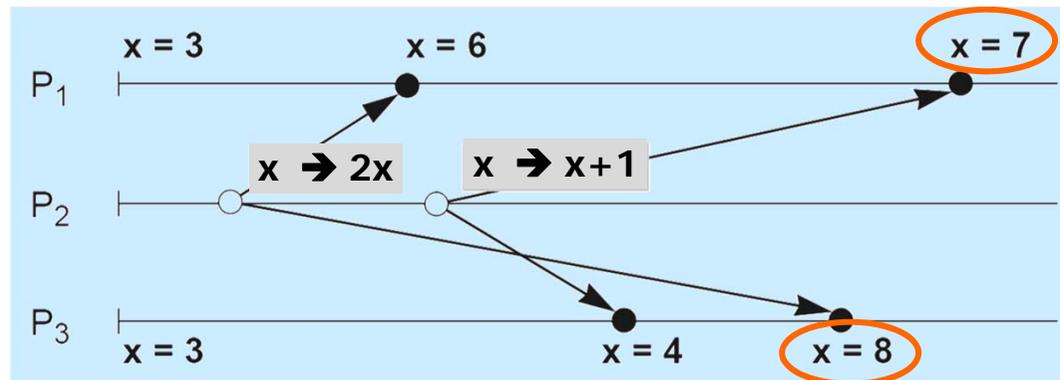
- $receive(N, s, sequ\_num);$   
wenn  $r$  noch kein  $deliver(N)$  für  $sequ\_num$  ausgeführt hat, dann:  
wenn  $r \neq s$  dann  $send(N, s, sequ\_num)$  an alle Nachbarn von  $r$  ;  
Nachricht an die Anwendungsebene ausliefern („ $deliver(N)$ “);

Beachte:  $receive \neq deliver$   
(unterscheide Anwendungsebene und Transportebene)

Prinzip: „Fluten“ des Netzes (→ Vorlesung „Verteilte Algorithmen“)

# Broadcast: Empfangsreihenfolge

- Wie ist die **Empfangsreihenfolge** von Gruppennachrichten?
  - problematisch wegen der i.Allg. unterschiedlichen Übermittlungszeiten
  - **Bsp.:** Update einer replizierten Variablen mittels „function shipping“:

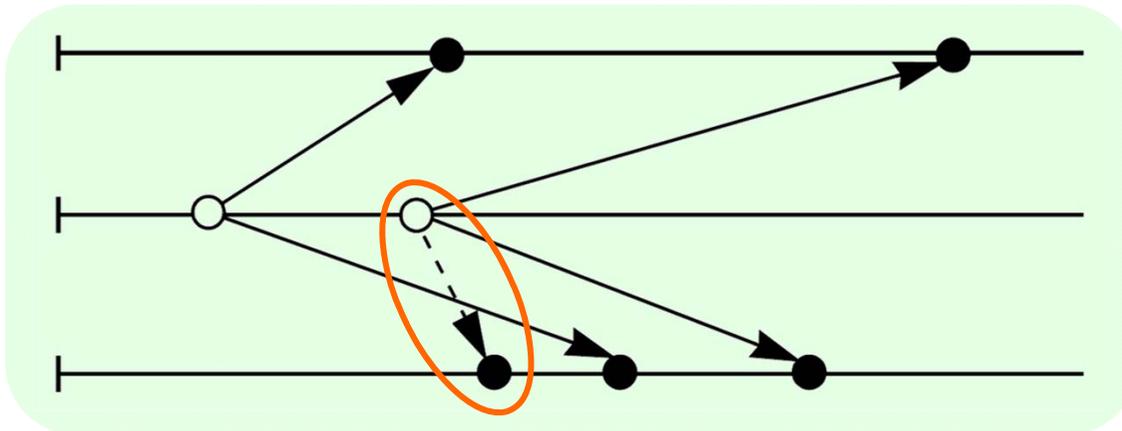


- Es sind diverse Semantiken zum **Ordnungserhalt** denkbar, z.B.:
  - **FIFO**
  - **kausal geordnet**
  - **total geordnet**

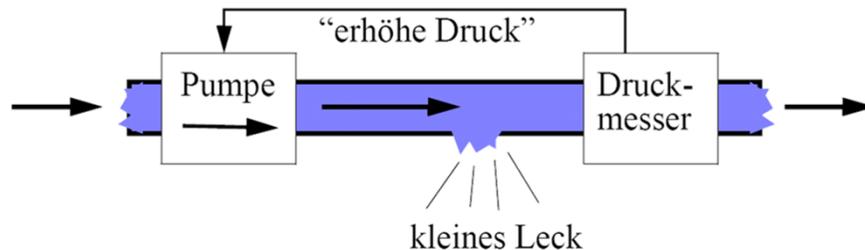
Wir besprechen dies  
nun der Reihe nach

# FIFO-Ordnung bei Multicast

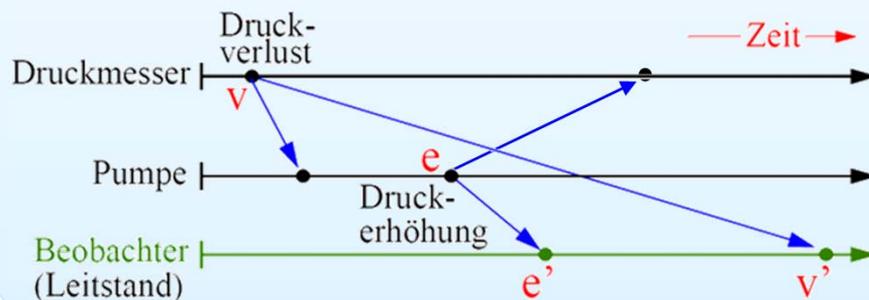
- Alle Broadcast-Nachrichten eines (d.h.: **ein und des selben**) **Senders** an eine Gruppe kommen bei allen Mitgliedern der Gruppe **in FIFO-Reihenfolge** an
  - Denkübung: wie dies mittels eines Protokolls garantieren?



# FIFO-Broadcasts sind aber nicht immer „gut genug“



Annahme: Steuerelemente kommunizieren über **FIFO-Broadcasts**:



Falsche Schlussfolgerung des Beobachters:

„Aufgrund einer unbegründeten Pumpenaktivität wurde ein Leck erzeugt, wodurch schliesslich der Druck absank.“

„Irgendwie“ kommt beim Beobachter die Reihenfolge durcheinander!

Man sieht also:

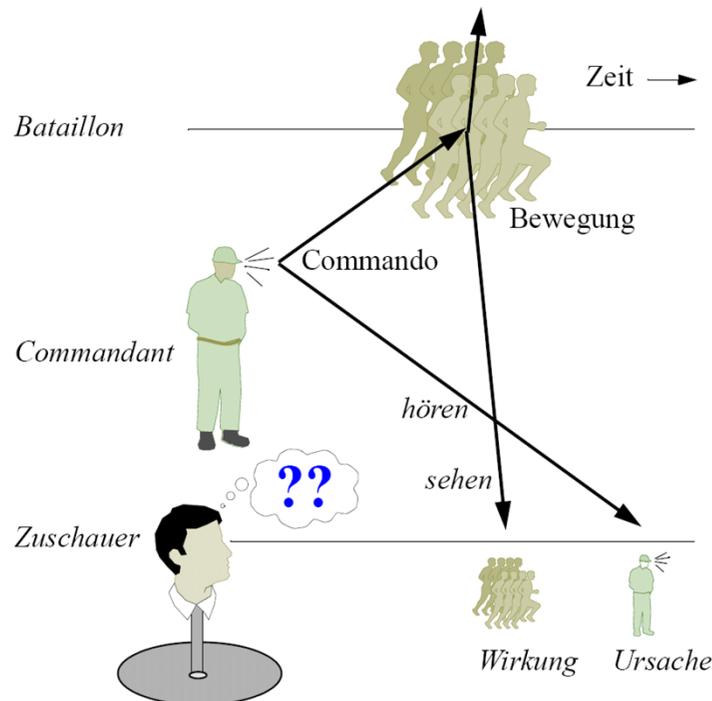
- FIFO-Reihenfolge ist nicht immer ausreichend, um Semantik zu wahren
- eine Nachricht verursacht oft das Senden einer anderen (→ Kausalität!)

# Das „Broadcastproblem“ ist nicht neu

wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, ehe er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der Causalzusammenhänge weiss er, dass die Bewegungen die Wirkung des gehörten Commandos sind, dieses also jenen objectiv vorangehen muss, und er wird sich sofort der Täuschung bewusst, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.

# Das „Broadcastproblem“ ist nicht neu

- Natürlicher Broadcast bei **Licht- und Schallwellen**
  - wann handelt es sich dabei um FIFO-Broadcasts?
  - wie ist es mit dem Kausalitätserhalt?



„Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so **sieht** er übereinstimmende Bewegungen desselben plötzlich eintreten, **ehe** er die Commandostimme oder das Hornsignal **hört**; aber aus seiner Kenntnis der **Causalzusammenhänge** weiss er, dass die Bewegungen die Wirkung des gehörten Commandos sind, dieses also jenen **objectiv** vorangehen muss, und er wird sich sofort der Täuschung bewusst, die in der **Umkehrung der Zeitfolge in seinen Perceptionen** liegt.“

*Christoph von Sigwart (1830-1904): Logik. Zweiter Band. Die Methodenlehre (1878)*



Lotte Sigwart  
geb. Georgii  
geb. den 30. Okt. 1836  
gest. den 31. März 1896.

Christoph Sigwart  
Professor der Philosophie  
geb. den 28. März 1830.  
gest. den 5. August 1904.

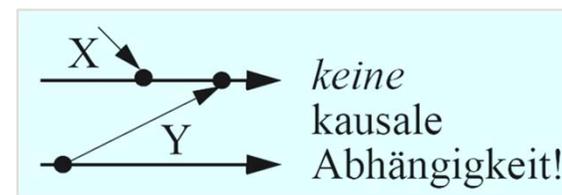
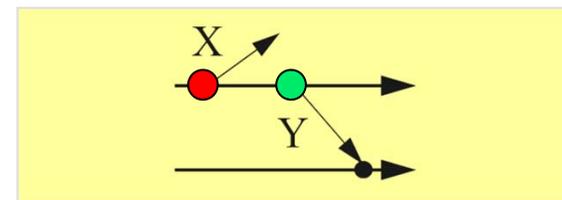
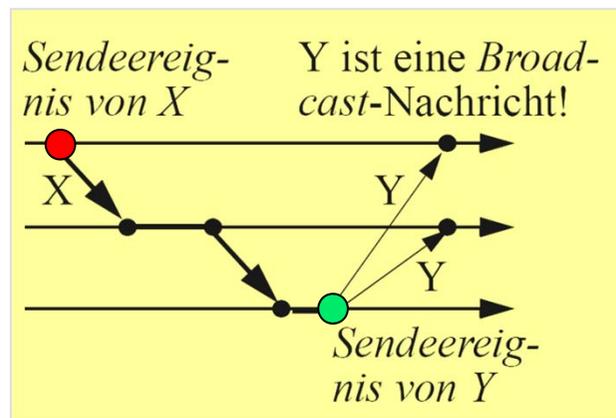
Karl Sigwart Architekt  
geb. 29. Mai 1874, gest. 11. Dez. 1912  
Georg Sigwart Pfarrer  
geb. 26. März 1817, gest. 16. Febr. 1915

FRIEDERIKE PFEIFFER  
1855-1925  
FRIEDRICH PFEIFFER  
1853-1900  
EMILIE P.  
1880  
KLARA  
1800

Friederike Jeller  
geb. Schallheiss  
geb. 3. Mai 1818, gest. 2. Nov. 1878  
Christian Jeller  
Oberrevisor  
geb. 28. Aug. 1826, gest. 20. Febr. 1898  
Mathilde Jeller  
geb. 21. Okt. 1828, gest. 1. Febr. 1898  
Marie  
geb. 8. März 1830, gest. 1. Febr. 1898

# Kausale Nachrichtenabhängigkeit

- **Definition:** Nachricht **Y** hängt kausal von Nachricht **X** ab, wenn es im Raum-Zeit-Diagramm einen von links nach rechts verlaufenden **Pfad** gibt, der vom Sendeereignis von **X** zum Sendeereignis von **Y** führt.



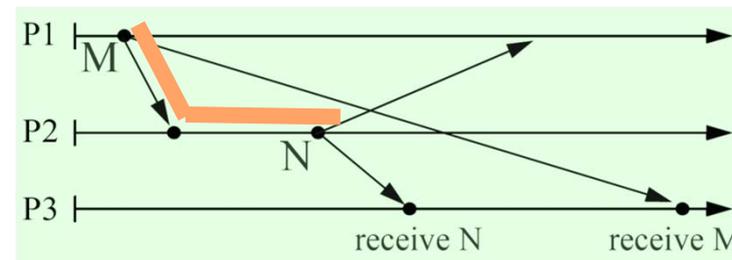
- **Beachte:** Dies lässt sich bei geeigneter Modellierung auch abstrakter ohne Rückgriff auf „Diagramme“ etc. fassen ( $\rightarrow$  „logische Zeit“ später in der Vorlesung, vor allem aber Vorlesung „Verteilte Algorithmen“)

# Kausaler Broadcast

Zweck: **Wahrung von Kausalität** bei der Kommunikation

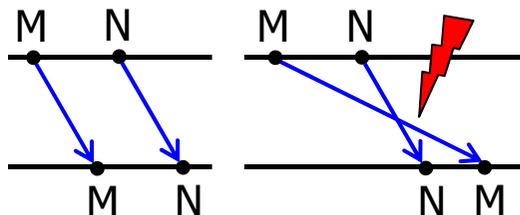
- Definition: Kausale Reihenfolge** („causal order“): Wenn eine Nachricht **N** kausal von einer Nachricht **M** abhängt, und ein Prozess **P** die Nachrichten **N** und **M** empfängt, dann muss er **M vor N** empfangen haben

- „Kausale Reihenfolge“ (bzw. „kausale Abhängigkeit“) lassen sich insbesondere auch auf Broadcasts anwenden → **kausaler Broadcast**



Gegenbeispiel:  
*Keine* kausalen Broadcasts!

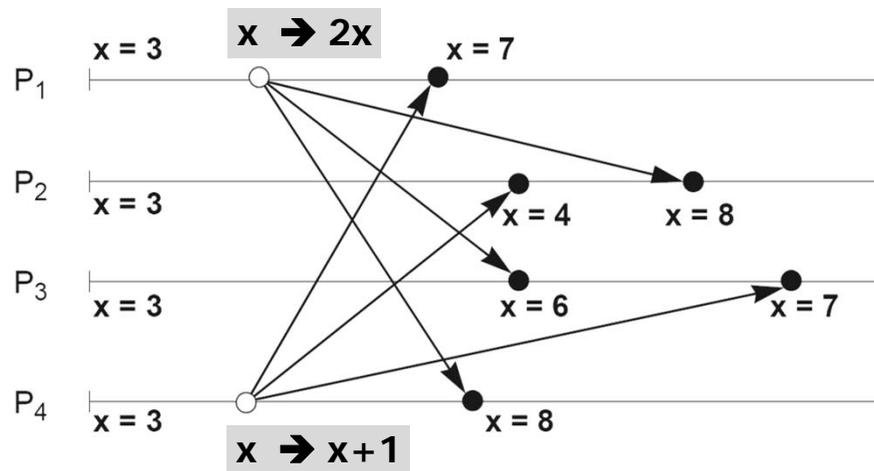
- Kausale Reihenfolge impliziert FIFO-Reihenfolge: kausale Reihenfolge ist eine Art „**globales FIFO**“



Das **Erzwingen der kausalen Reihenfolge** ist mittels geeigneter Algorithmen möglich (→ Vorlesung „Verteilte Algorithmen“, z.B. Verallgemeinerung der Sequenzzählermethode für FIFO)

# Probleme mit (kausalen) Broadcasts?

- **Beispiel:** Aktualisierung einer replizierten Variablen  $x$ :



## Konkrete Problemursache:

- Broadcasts werden nicht überall „gleichzeitig“ empfangen
- dies führt lokal zu verschiedenen Empfangsreihenfolgen

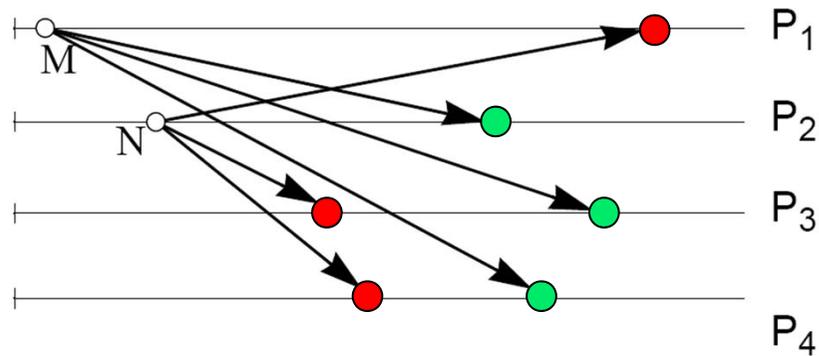
- **Abstrakte Ursache:**
  - die Nachrichtenübermittlung erfolgt (erkennbar!) „nicht-atomar“
- $\rightarrow$  Auch kausale Broadcasts haben **keine „ideale“ Semantik** (im Sinne einer Illusion von speicherbasierter Kommunikation)

# Atomarer Broadcast

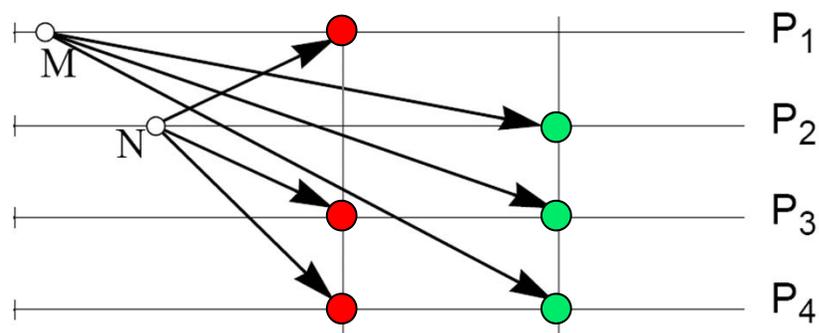


- *Definition:* Wenn zwei Prozesse  $P_1$  und  $P_2$  beide die Nachrichten  $M$  und  $N$  empfangen, dann empfängt  $P_1$  die Nachricht  $M$  vor  $N$  genau dann, wenn  $P_2$  die Nachricht  $M$  vor  $N$  empfängt
- Anschaulich: Nachrichten eines Broadcasts werden „überall quasi gleichzeitig“ empfangen
- Beachte: „Atomar“ heisst hier nicht „alles oder nichts“ (wie etwa beim Transaktionsbegriff von Datenbanken)

# Atomarer Broadcast: Beispiel



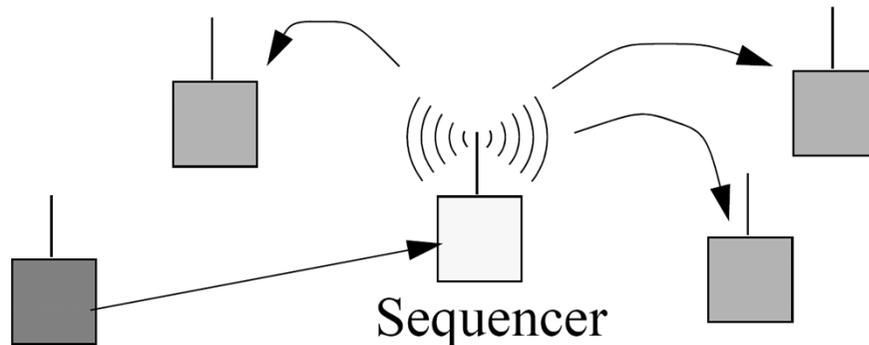
Äquivalent bzgl. „Gummiband-Transformation“



Beachte: das Senden wird nicht als Empfang der Nachricht beim Sender selbst gewertet

# Realisierung von atomarem Broadcast

## 1. Lösung: Zentraler „Sequencer“, der Reihenfolge festlegt



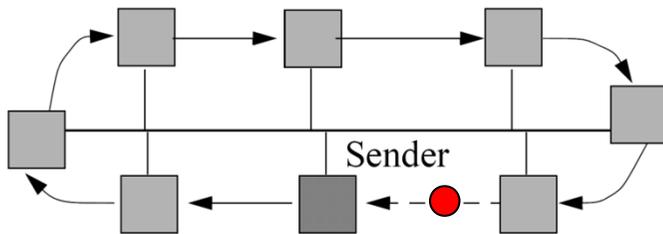
Sequencer ist allerdings ein potentieller Engpass!

- „Unicast“ vom Sender zum Sequencer
- **Multicast vom Sequencer** an alle
- Sequencer wartet jew. auf **Acknowledgements** von allen
  - Denkübung: genügt anstelle von Acknowledgements auch ein **FIFO-Multicast** (ohne Acknowledgements)?

# Realisierung von atomarem Broadcast

Verfahren berücksichtigt auch Nachrichtenverluste und gecrashte Prozesse

## 2. Lösung: **Token**, das auf einem (logischen) Ring kreist

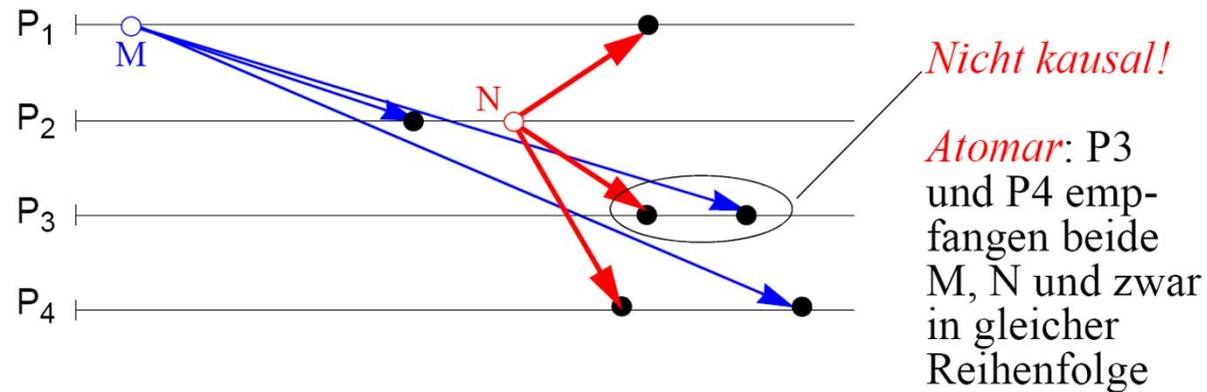


- **Token = Senderecht** (Token weitergeben!)
- Broadcast selbst z.B. über ein zugrundeliegendes broadcastfähiges Medium (oder Protokoll aus Einzelnachrichten, z.B. „fluten“)

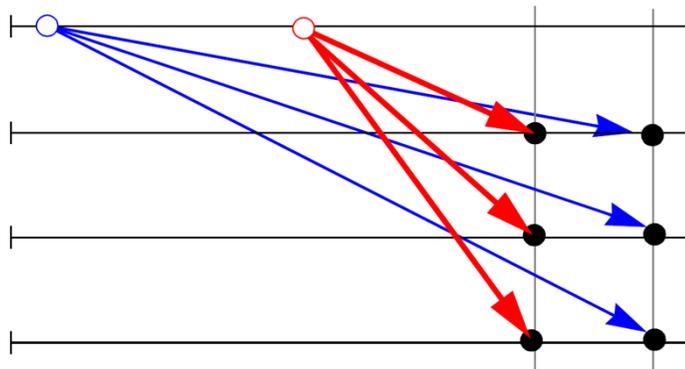
- Token führt eine **Sequenznummer** (inkrementiert beim Senden); so werden alle **Broadcasts global nummeriert**
- Empfänger wissen, dass Nachrichten **entsprechend** der (in den Nachrichten mitgeführten **Sequenznummer**) **ausgeliefert** werden müssen
- Bei **Lücken** in den Nummern: dem Token einen **Wiederholungswunsch** mitgeben (Sender erhält damit implizit ein NACK bzw. ACK)
- **Tokenverlust** (z.B. durch Prozessor-Crash) durch **Timeouts** feststellen (Vorsicht: Gefahr, dass dabei Token unabsichtlich verdoppelt wird!)
- Einen **gecrashten Prozess** (der z.B. das Token nicht entgegennimmt) aus dem logischen Ring entfernen
- **Variante** (z.B. bei vielen Teilnehmern): Statt Ring: Token auf Anforderung direkt zusenden (**broadcast: „Token bitte zu mir“**), dabei aber Fairness beachten

# Wie „gut“ ist atomarer Broadcast?

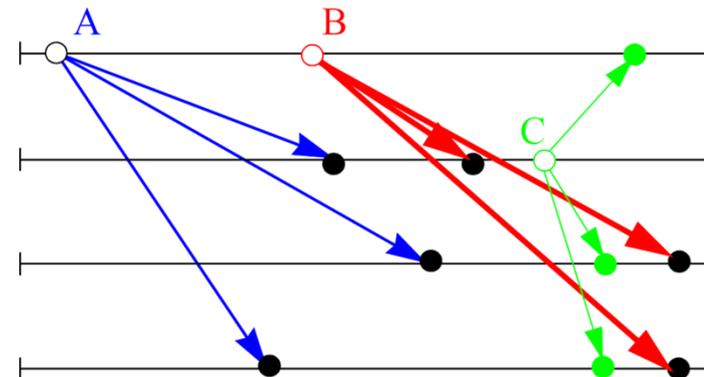
1) Ist **atomar** auch **kausal**?



2) Ist **atomar** wenigstens **FIFO**?



3) Ist **atomar + FIFO** evtl. **kausal**?



# Fazit: Semantik von Broadcast

- Atomare Übermittlung  $\not\Rightarrow$  kausale Reihenfolge
- Atomare Übermittlung  $\Rightarrow$  FIFO-Reihenfolge
- Atomare Übermittlung + FIFO  $\not\Rightarrow$  kausale Reihenfolge
  - Bemerkung zu vorheriger Seite: nicht nur 3), sondern auch 1) ist ein (Gegen)beispiel, da M, N FIFO-Broadcast ist
- Vergleich mit („idealer“) **speicherbasierter Kommunikation**:
  - 1) Kommunikation über gemeinsamen Speicher ist **atomar** (alle „sehen“ das Geschriebene gleichzeitig – so sie hinschauen)
  - 2) Kommunikation über gemeinsamen Speicher **wahrt Kausalität** (Wirkung tritt unmittelbar mit dem Schreibereignis als Ursache ein)

# Kausaler atomarer Broadcast



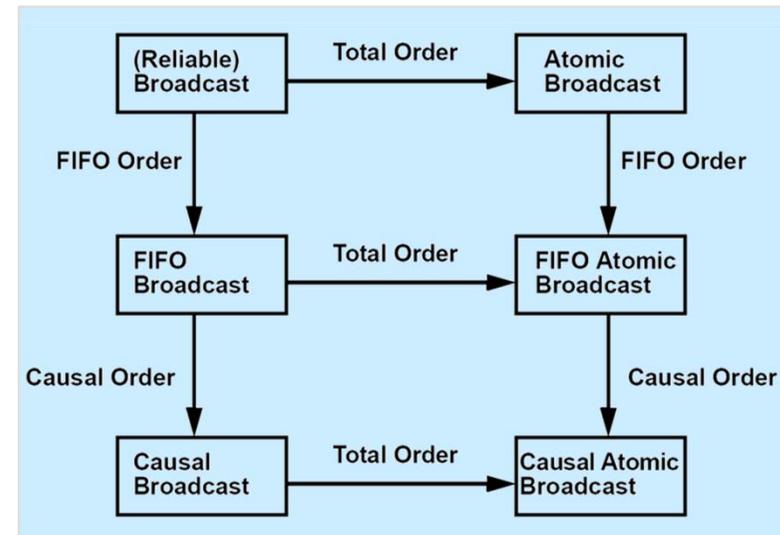
- Der speicherbasierten Kommunikation vergleichbares Kommunikationsmodell per Nachrichten:  
**kausaler atomarer Broadcast**
    - = kausaler Broadcast + atomarer Broadcast
  - Man nennt daher kausale, atomare Übermittlung auch **virtuell synchrone Kommunikation**
- 
- **Denkübung**: realisieren die beiden Implementierungen „zentraler Sequencer“ bzw. „Token auf Ring“ die virtuell syn. Kommunikation?

# Stichwort: Virtuelle Synchronität

- Idee: Ereignisse finden zu **verschiedenen Realzeitpunkten** statt, aber zur **gleichen logischen Zeit**
  - logische Zeit berücksichtigt nur die **Kausalstruktur** der Nachrichten und Ereignisse; die exakte Lage der **Ereignisse** auf dem „Zeitstrahl“ ist **verschiebbar** (Dehnen / Stauchen wie auf einem Gummiband)
- **Innerhalb** des Systems ist synchron (im Sinne von „gleichzeitig“) und virtuell synchron **nicht unterscheidbar**
  - identische Kausalbeziehungen
  - identische totale Ordnung aller Ereignisse
- Konsequenz: Nur mit Hilfe **Realzeit** / echter Uhr könnte man (z.B. als externer Beobachter) den **Unterschied** feststellen
- Den Begriff „**logische Zeit**“ werden wir später noch genauer fassen (mehr dazu dann wieder in der Vorlesung „Verteilte Algorithmen“)

# Broadcast – schematische Übersicht

- Warum nicht **einzigster Broadcast**, der alles kann? „Stärkere Semantik“ hat auch **Nachteile**:
  - Leistungseinbussen
  - weniger potentielle Parallelität
  - aufwändiger zu implementieren
- Bekanntes Prinzip:
  - man begnügt sich daher, falls es der Anwendungskontext gestattet, oft mit einer **billigeren**, aber **weniger perfekten** Lösung
  - Motto: so billig wie möglich, so „perfekt“ wie nötig
  - man sollte aber die **Einschränkungen einer Billiglösung kennen!**
- ⇒ grössere Vielfalt ⇒ komplexer bzgl. Verständnis und Anwendung

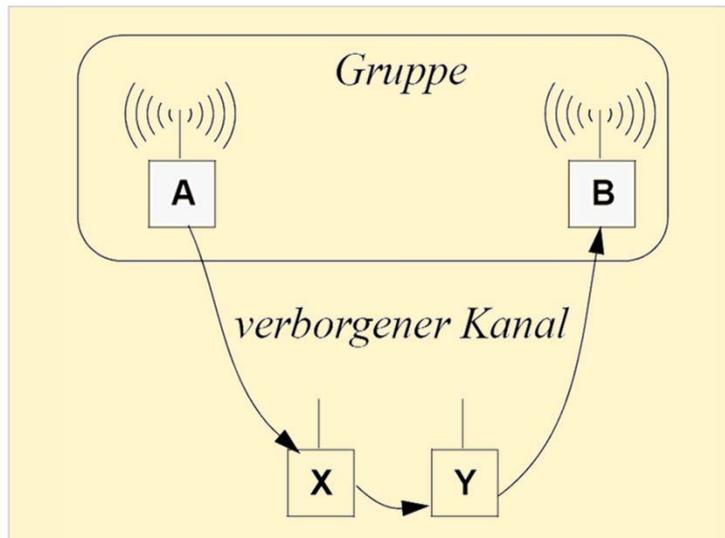


# Multicast



- Multicast = Broadcast an eine **Teilmenge von Prozessen**
  - diese Teilmenge wird „Multicast-Gruppe“ genannt
- Zweck von **Multicast-Gruppen**
  - „selektiver Broadcast“
  - Vereinfachung der Adressierung (z.B. statt Liste von Einzeladressen)
  - Verbergen der Gruppenzusammensetzung nach aussen
  - „logischer Unicast“: Gruppen ersetzen Individuen (z.B. für transparente Replikation)
- Alles, was zur Broadcastsemantik gesagt wurde, gilt (innerhalb der Gruppe) auch bzgl. **Multicastsemantik**:
  - zuverlässiger Multicast, FIFO-Multicast, kausaler Multicast, atomarer Multicast, kausaler atomarer Multicast

# Problem der „Hidden Channels“ beim Multicast



Kausalitätsbezüge verlassen  
(z.B. durch Gruppenüberlap-  
pung) die **Multicastgruppe**  
und kehren später wieder

- Soll nun das Senden von **B** als **kausal abhängig** vom Senden von **A** gelten?
- **Global** gesehen ist das der Fall, **innerhalb der Gruppe** ist eine solche Abhängigkeit jedoch nicht erkennbar

# Dynamische Multicastgruppen

- Bei **dynamischen Gruppen** können Prozesse jederzeit der Gruppe **beitreten** oder aus der Gruppe **austreten**
  - **Crash** kann als eine besondere Austrittsform modelliert werden
- Und wenn dies **während** des Ablaufs einer Multicast-Operation geschieht?
  - haben dann z.B. verschiedene Sender an die Gruppe die **gleiche Sicht** bzgl. der Gruppenzusammensetzung?
- **Man wünscht sich** (Realisierung besprechen wir hier nicht!):
  - die Gruppe soll bei allen (potentiellen) Sendern an die Gruppe hinsichtlich der (logischen) Beitritts- und Austrittszeitpunkte jedes Gruppenmitglieds immer **übereinstimmen**
  - Beitritt und Austritt sollen **atomar** erfolgen

# Weitere Kommunikationsparadigmen



Wir besprechen nachfolgend 3 weitere Aspekte:

1. Push-Paradigma
2. Publish & Subscribe
3. Tupelräume (und JavaSpaces)

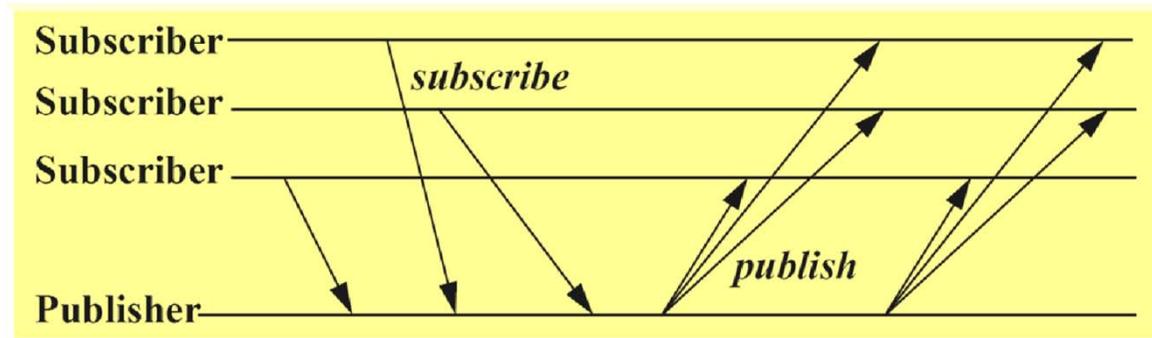
# Push-Paradigma



- „Push“ im Unterschied zum klassischen „Pull-“ (bzw. „Request / Reply“)-Paradigma, bei dem
  - Clients die gewünschte **Information aktiv anfordern** müssen,
  - sie aber oft nicht wissen, **ob bzw. wann** sich eine Information geändert hat,
  - dadurch periodisches Nachfragen („**polling**“) beim Server notwendig ist
- **Unangefragt** (vom Client) schickt der Server etwas
  - der Client hat den Server höchstens pauschal beauftragt
- Push: „**event driven**“ ↔ Pull: „**demand driven**“

# Publish & Subscribe

- **Subscriber** (= Client) meldet sich für den Empfang des gewünschten Typs von Information („channel“) an
- Subscriber erhält **automatisch** (aktualisierte) Information vom **Publisher** (= Server), sobald diese zur Verfügung steht
  - **push** vom Publisher bzw. „**callback**“ des Subscribers durch Publisher



- „**Publish**“ entspricht **Multicast**
  - „**subscribe**“ entspricht so gesehen dem **Beitritt** einer Multicast-Gruppe

# Tupelräume



- Gemeinsam genutzter („virtuell globaler“) Speicher
- **Blackboard-** oder **Marktplatz-**Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkoppelung der Teilnehmer : „Sender“ kennt / adressiert „Empfänger“ nicht und umgekehrt
- **Tupel** = geordnete Menge typisierter Datenwerte
- Entworfen 1985 von **David Gelernter** (für die Sprache Linda)
- **Operationen:**
  - **out (t)**: Einfügen eines Tupels t in den (globalen) Tupelraum
  - **in (t)**: Lesen und Löschen von t aus dem Tupelraum
  - **read (t)**: Lesen („zerstörungsfrei“) von t im Tupelraum

# Tupelräume (2)



- **Inhaltsadressiert** („Assoziativspeicher“)
  - Vorgabe eines Zugriffsmusters (bzw. „Suchmaske“) beim Lesen; damit Ermittlung der restlichen Datenwerte eines Tupels („wild cards“)
  - z.B.: `int p; in(„Artikelpreis“, 4711, ?p)` liefert ein „passendes“ Tupel
  - analog zu einigen relationalen Datenbankabfragesprachen
- **Synchrone** und **asynchrone** Leseoperationen
  - ``in`` und ``read`` blockieren, bis ein passendes Tupel vorhanden ist
  - ``inp`` und ``readp`` blockieren nicht, sondern liefern als Prädikat („passendes Tupel vorhanden?“) ``wahr`` oder ``falsch`` zurück

# Tupelräume (3)

- Mit Tupelräumen sind auch die üblichen Kommunikationsmuster realisierbar (d.h. simulierbar), z.B. **Client-Server**:

```
/* Client */
...
out("Anfrage" client_Id, Parameterliste);
in("Antwort", client_Id, ?Ergebnisliste);
...
/* Server*/
...
while (true)
{ in("Anfrage", ?client_Id, ?Parameterliste);
 ...
 out("Antwort", client_Id, Ergebnisliste);
}
```

Zuordnung des  
„richtigen“ Clients  
über die client\_Id

# Tupelräume (4)



- Kanonische **Erweiterungen** des Modells hinsichtlich
  - **Persistenz** (Tupel bleiben nach Programmende erhalten)
  - **Transaktionseigenschaft** (wichtig, wenn mehrere Prozesse parallel auf den Tupelraum bzw. gleiche Tupel zugreifen)
- Problem: **effiziente, skalierbare Implementierung?**
  - 1) **zentrale Lösung**: Engpass
  - 2) **replizierter Tupelraum** (jeder Teilnehmer hat vollständige Kopie des Tupelraums; schnelle Zugriffe, jedoch hoher Synchronisationsaufwand)
  - 3) **aufgeteilter Tupelraum** (jeder hat disjunkten Teil des Tupelraums; 'out' kann z.B. lokal ausgeführt werden, 'in' evtl. mit Broadcast)
- **Kritik**: Konzept des globalen Speichers ist der strukturierten Programmierung und der Verifikation abträglich
  - unüberschaubare potentielle Seiteneffekte

# JavaSpaces



- „Tupelraum“ für Java
  - gespeichert werden Objekte → neben Daten auch „Verhalten“
  - Tupel entspricht Gruppen von Objekten
- Operationen
  - **write**: mehrfache Anwendung erzeugt verschiedene Kopien
  - **read**
  - **readifexists**: blockiert (im Gegensatz zu read) nicht; liefert u.U. 'null'
  - **takeifexists**
  - **notify**: Benachrichtigung (mittels eines Ereignisses), wenn ein passendes Objekt in den JavaSpace geschrieben wird

# JavaSpaces (2)

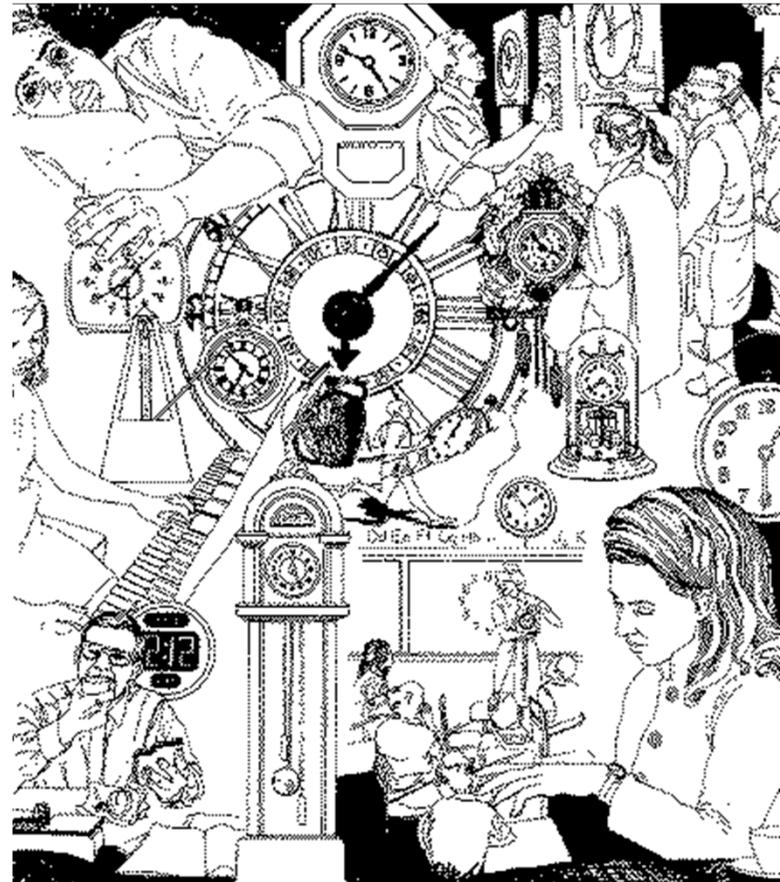


- **Teil von Jini** (Infrastrukturplattform und Middleware für Java)
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode vom Sender zum Empfänger
  - gemeinsame Nutzung von Objekten
  - **persistente Speicherung** von Objekten (aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt)
- **Semantik**: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn ein write vor einem read beendet wird, muss read nicht notwendigerweise das lesen, was write geschrieben hat

# Logische Zeit

# Zeit?

*Ich halte ja eine Uhr für überflüssig. Sehen Sie, ich wohne ja ganz nah beim Rathaus. Und jeden Morgen, wenn ich ins Geschäft gehe, da schau ich auf die Rathausuhr hinauf, wie viel Uhr es ist, und da merke ich's mir gleich für den ganzen Tag und nütze meine Uhr nicht so ab. -- Karl Valentin*

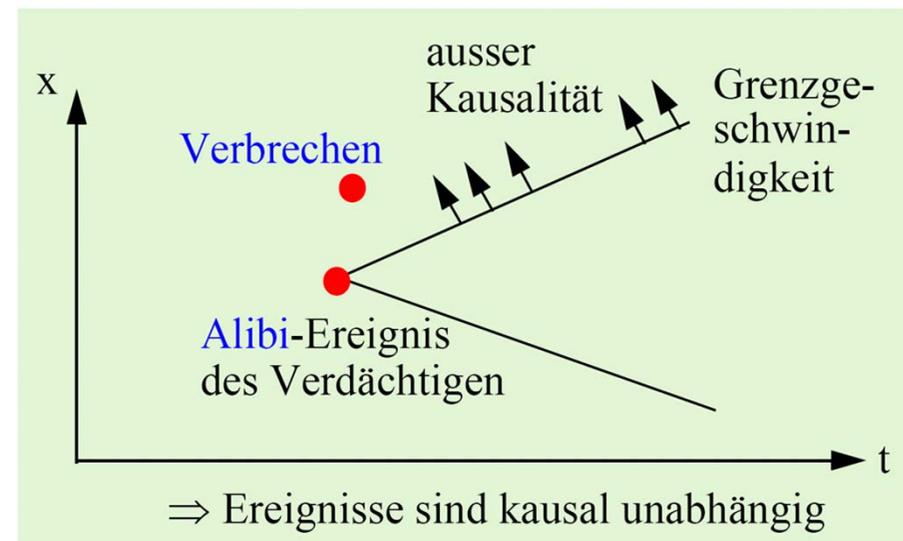


# Zeit ist nützlich

Beispiele:

1. Volkszählung: **Stichzeitpunkt** in der Zukunft
  - liefert eine gleichzeitige „Beobachtung“ im Nachhinein

2. **Kausalitätsbeziehung** zwischen Ereignissen („**Alibi-Prinzip**“)
  - Ausschluss potentieller Kausalität
  - wurde Y später als X geboren, dann kann Y unmöglich Vater von X sein



# Zeit ist nützlich (2)



## 3. Fairer **wechselseitiger Ausschluss**

- bedient wird derjenige, der am längsten wartet

## 4. Viele **weitere nützliche Anwendungen** von „Zeit“ in unserer verteilten realen Welt

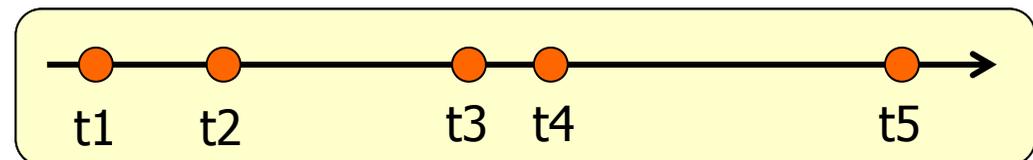
- z.B. **kausaltreue Beobachtung** durch „Zeitstempel“ der Ereignisse

- 
- Zeit in verteilten Systemen ist vor allem auch dann wichtig, wenn es um Interaktionen mit der **realen Welt** geht
    - Prozesssteuerung, Realzeitsysteme, Cyber-Physical Systems,...

# Eigenschaften der „Realzeit“

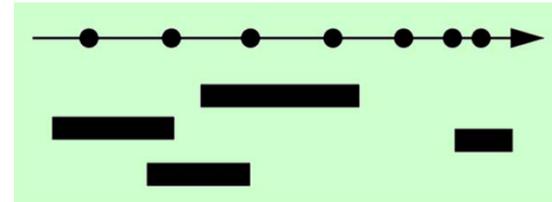
Struktureigenschaften eines „natürlichen“ Zeitpunktmodells:

- asymmetrisch (Zeit ist „gerichtet“)
  - transitiv
  - irreflexiv
  - linear
- } lineare Ordnung („später als“)
- unbeschränkt („Zeit ist ewig“: Kein Anfang oder Ende)
  - dicht (es gibt immer einen Zeitpunkt dazwischen)
  - kontinuierlich
  - metrisch
  - vergeht „von selbst“  
→ jeder Zeitpunkt wird schliesslich erreicht



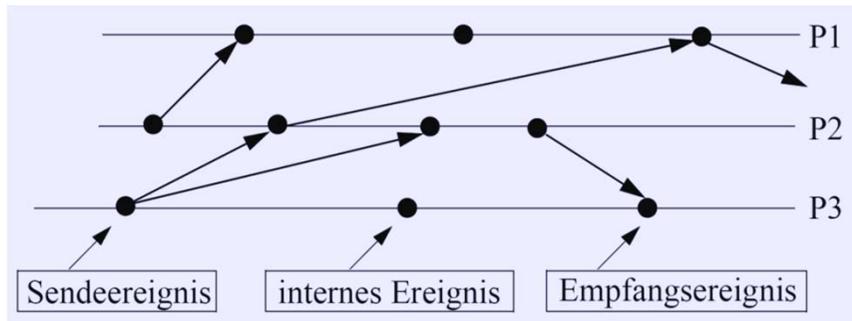
# Eigenschaften der „Realzeit“ (2)

- Ist das **Zeitpunktmodell** adäquat? Oder sind **Zeitintervalle** besser?



- wann tritt das Ereignis (?) „Sonne wird rot“ am Abend ein?
- Welche **Eigenschaften** benötigen wir überhaupt?
  - Idee: „billigeren“ Ersatz für fehlende globale Realzeit konstruieren (z.B.: sind die reellen / rationalen / ganzen Zahlen gute Modelle?)
  - wann genügt welche Form „logischer“ statt „echter“ Zeit?
  - dazu vorher klären: was wollen wir mit „Zeit“ anfangen?

# Raum-Zeitdiagramme und die Kausalrelation



Interessant dabei: von links nach rechts verlaufende „Kausalitätspfade“

Bezeichnung oft: „happened before“

- eingeführt von Leslie Lamport (1978)
- Vorsicht: damit ist nicht direkt eine „zeitliche“ Aussage getroffen!

Definiere eine Kausalrelation  $\prec$  auf der Menge aller Ereignisse:

- Es sei  $x \prec y$  genau dann, wenn:
  - $x$  und  $y$  auf dem gleichen Prozess stattfinden und  $x$  vor  $y$  kommt, oder
  - $x$  ist ein Sendereignis und  $y$  korrespondierendes Empfangsereignis, oder
  - $\exists z$  mit  $x \prec z \wedge z \prec y$  (Transitivität)

links von

zur gleichen Nachricht gehörend

# Logische Zeitstempel von Ereignissen

- Zweck: Ereignissen  $E$  eine Zeit geben
  - welche Zeit *zwischen* Ereignissen herrscht, ist irrelevant
  - gesucht ist also eine Abbildung  $C: E \rightarrow \mathbb{N}$  („C“ steht für „Clock“)
  - $\mathbb{N}$  genügt hier,  $\mathbb{Z}$  oder  $\mathbb{R}$  ist nicht nötig, wie wir sehen werden
- $C(e)$  nennt man den **Zeitstempel** von  $e$ 
  - wenn  $C(e) < C(e')$ , dann nennt man  $e$  **früher als**  $e'$

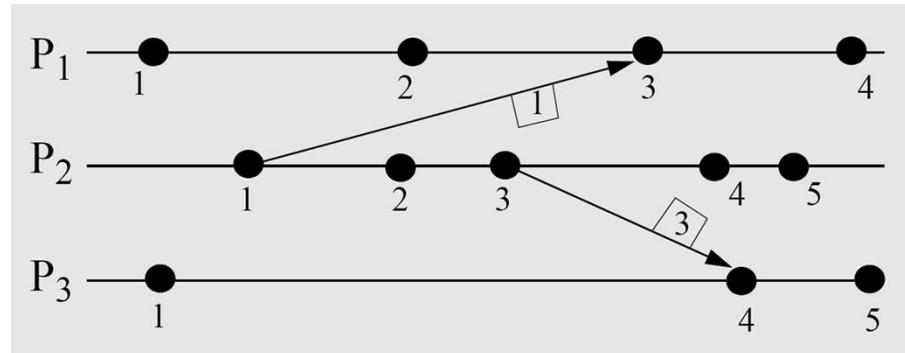
Ordnungshomomorphismus

- 
- Sinnvolle Forderung: **Uhrenbedingung**:  $e \prec e' \Rightarrow C(e) < C(e')$ 
    - Interpretation („Zeit ist kausaltreu“):

Kann ein Ereignis  $e$  ein anderes Ereignis  $e'$  beeinflussen, dann muss  $e$  einen kleineren Zeitstempel als  $e'$  haben

# Logische Uhren von Lamport

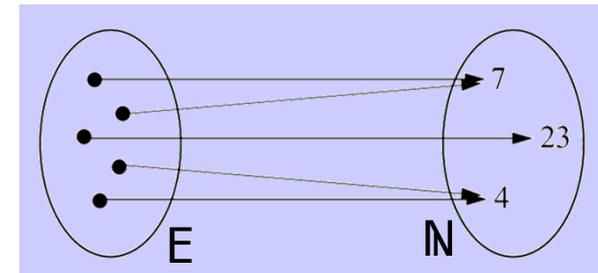
- $C: (E, \prec) \rightarrow (\mathbb{N}, <)$   
Zeitstempel-Zuordnung
- $e \prec e' \Rightarrow C(e) < C(e')$   
Uhrenbedingung



- Protokoll zur **Implementierung der Uhrenbedingung**:
  - bei jedem Ereignis tickt die **lokale Uhr** (= „Zähler“) des Prozesses
  - **Sendeereignis**: Uhrwert mitsenden ( $\rightarrow$  Zeitstempel der Nachricht)
  - **Empfangsereignis**:  $\text{Uhr} = \max(\text{Uhr}, \text{Zeitstempel der Nachricht})$   
(zuerst max, unmittelbar danach erst tickt die Uhr)
- **Behauptung: Protokoll respektiert Uhrenbedingung**
  - *Beweis*: Entlang von Kausalitätspfaden wächst logische Zeit monoton...

# Lamport-Zeit: injektive Variante

- Die durch Lamports Protokoll definierte Abbildung ist **nicht injektiv**
  - wäre wichtig z.B. für: „Wer die kleinste Zeit hat, der gewinnt“



- Lösung: **lexikographische Ordnung**  $(C(e),i)$ , wobei  $i$  die Prozessnummer bezeichnet, auf dem  $e$  stattfindet
  - Def.:  $(a,b) < (a',b') \Leftrightarrow a < a' \vee a = a' \wedge b < b'$  (ist eine lineare Ordnung!)
- Alle Ereignisse haben nun **verschiedene Zeitstempel**
  - Abbildung ist injektiv
  - jede (nicht-leere) Menge von Ereignissen hat nun ein „frühestes“
- Abb.  $(E, \prec) \rightarrow (\mathbb{N} \times \mathbb{N}, <)$  respektiert die **Uhrenbedingung**

# Umkehrung der Uhrenbedingung?

- Wieso gilt folgendes eigentlich nicht?

$$e \prec e' \iff C(e) < C(e')$$

- Was kann man überhaupt über die beiden **Ereignisse**  $e$  und  $e'$  sagen, wenn man die **Zeitstempel**  $C(e)$  und  $C(e')$  vergleicht?
- Kann man eine andere Art von Zeitstempeln finden, für die die **Umkehrung der Uhrenbedingung** gilt?
  - wofür wäre das **nützlich**?
  - → Vorlesung „verteilte Algorithmen“

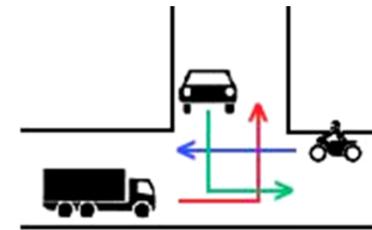
# Wechselseitiger Ausschluss



# Wechselseitiger Ausschluss

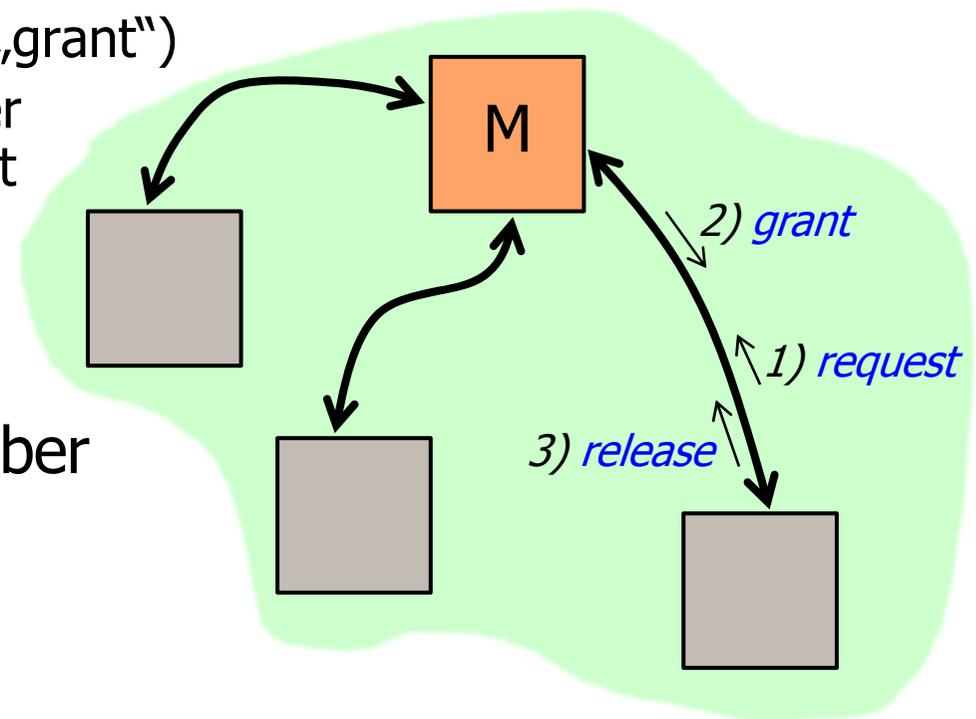
(Mutual Exclusion, „Mutex“)

- Koordination, wenn viele wollen, aber **nur einer darf**
- „Streit“ um **exklusives Betriebsmittel**, z.B.:
  - konkrete Ressource (z.B. gemeinsamer Datenbus)
  - abstrakte Ressource (z.B. ein „Termin“ in einem verteilten Terminkalendersystem)
  - „**kritischer Abschnitt**“ in einem nebenläufigen Programm
- Es gibt klassische Lösungen bei **shared memory**
  - z.B. **Semaphore** und **Monitore** (→ Betriebssystemtheorie)
  - sind in unserem verteilten Kontext aber nicht relevant



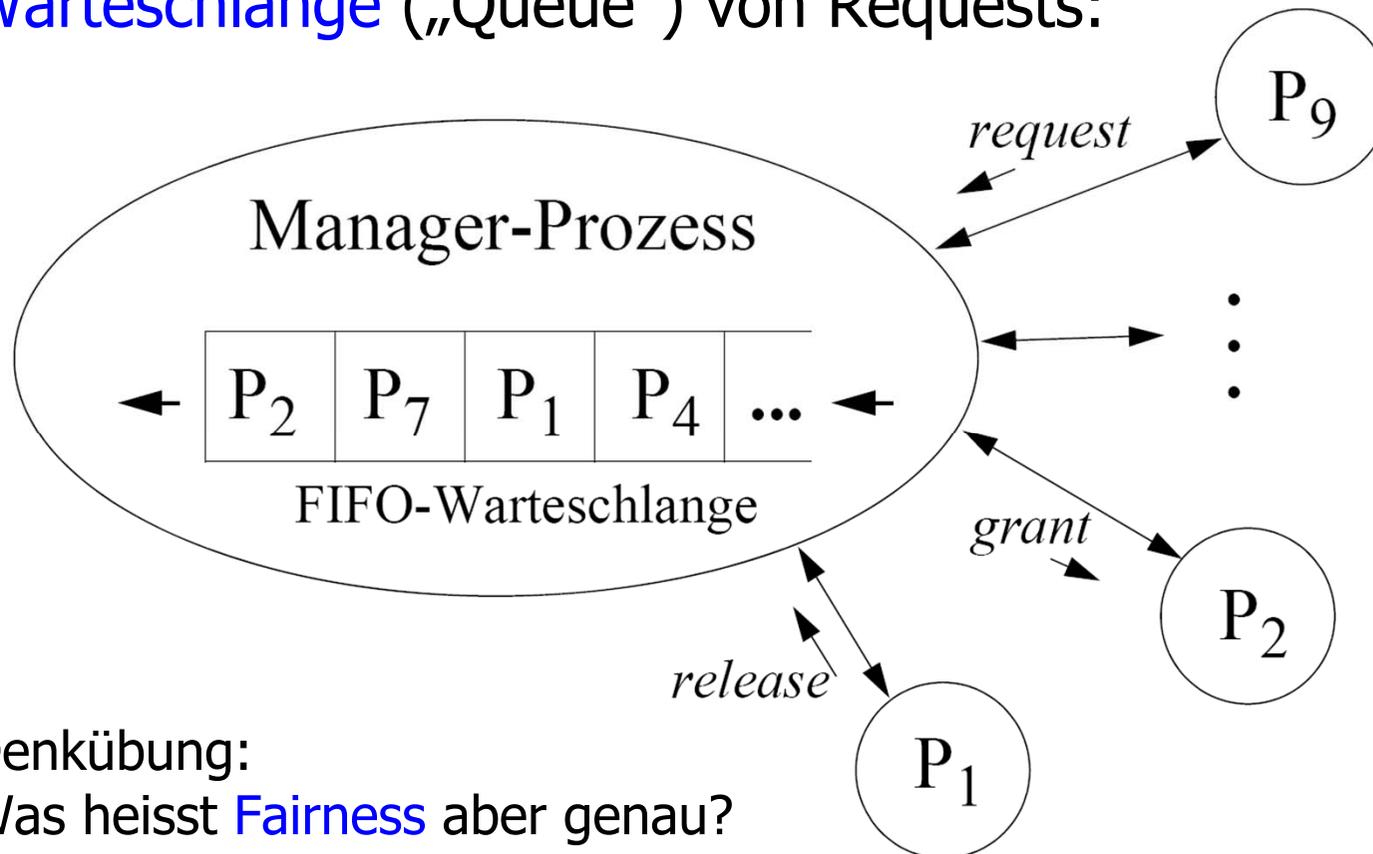
# Zentraler Manager?

- Hier: Nachrichtenbasiertes System konkurrierender Prozesse
- Idee: **Manager**, der Ressourcen (exklusiv aber fair!) zuordnet
  - ein Prozess **bewirbt** sich um die Ressource mit „request“
  - wartet dann auf **Erlaubnis** („grant“)
  - teilt schliesslich **Freigabe** der Ressource dem Manager mit „release“ mit
- Vergleichsweise einfach und wenige Nachrichten, aber
  - potentieller **Engpass**
  - **single point of failure**



# Globale Warteschlange garantiert Fairness

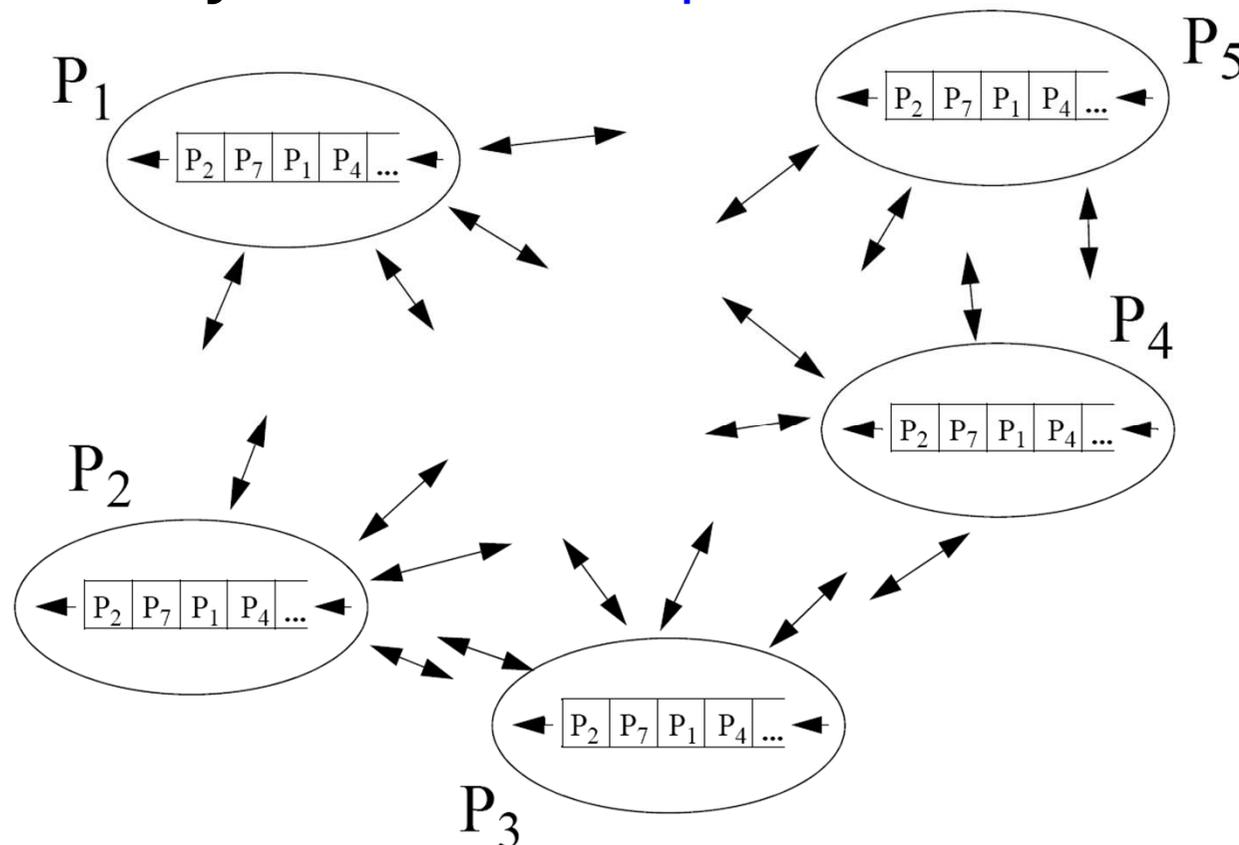
- Zentraler Manager-Prozess hält eine (zeitlich geordnete) **Warteschlange** („Queue“) von Requests:



- Denkübung:  
Was heisst **Fairness** aber genau?

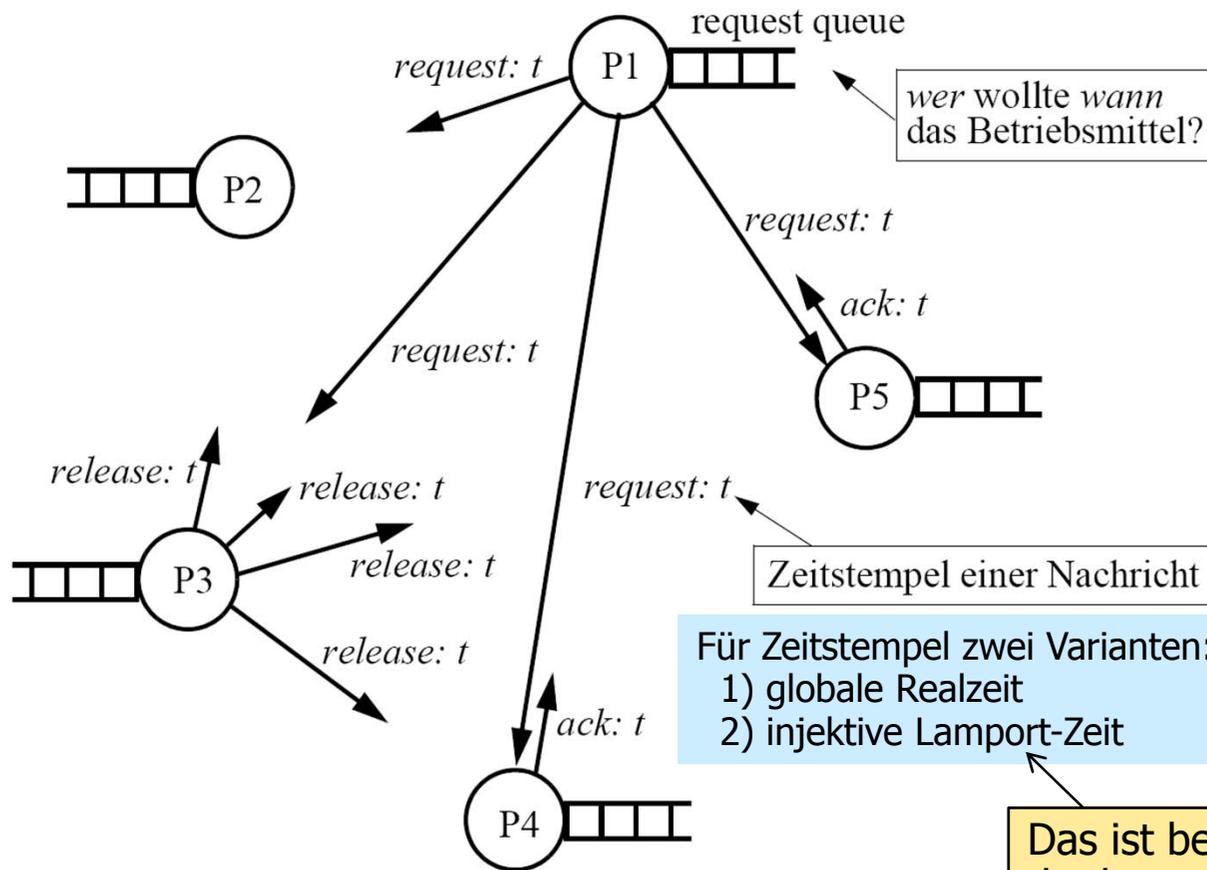
# Replizierte Warteschlange?

- Idee für eine dezentrale Lösung: globale **Warteschlange** bei jedem Prozess **replizieren**



- Alle Prozesse sollen die **gleiche Sicht** der „virtuell globalen“ Warteschlange haben
- Konsistenz** wird mit (vielen) Nachrichten und **logischer Zeit** erreicht (→ nächste zwei slides)

# Synchronisation der Warteschlangen mit Zeitstempeln



- Voraussetzung: **FIFO-Kommunikationskanäle**
- Alle Nachrichten tragen (eindeutige!) **Zeitstempel**
- Request- und Release-Nachrichten immer an alle senden (**FIFO-Broadcast**)
- Requests werden bestätigt („ack“)

Das ist besonders interessant, da dann auf synchronisierte Uhren verzichtet werden kann

# Der Algorithmus (Lamport 1978)

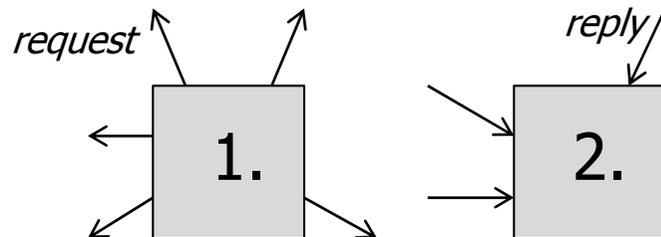
- 1) **Bewerbung** um Betriebsmittel: Request mit Zeitstempel und Absender an alle senden und in eigene Queue einfügen Denkübungen:
- 2) Bei **Empfang eines Request**: Request in eigene Queue einfügen, ack versenden ▪ Wieso ist **FIFO** notwendig?  
▪ Wo geht (bei Lamport-Zeit) die **Uhrenbedingung** ein?
- 3) Bei **Freigabe** des Betriebsmittels: Aus eigener Queue entfernen und Release an alle versenden Wieso sind garantiert:  
1) **Safety** (zu jedem Zeitpunkt höchstens einer),  
2) **Fairness** (jeder Request wird „schliesslich“ erfüllt)?
- 4) Bei **Empfang eines Release**: Zugehörigen Request aus eigener Queue entfernen
- 5) Ein Prozess darf das **Betriebsmittel nutzen, wenn:**
  - a) der eigene Request der früheste in seiner Queue ist
  - b) und er bereits von jedem anderen Prozess (irgendeine) spätere Nachricht bekommen hat

(Frühester Request ist global eindeutig  $\Rightarrow$  die beiden Bedingungen garantieren, dass kein früherer Request mehr kommt (wieso?))

$3(n-1)$  Nachrichten pro Bewerbung  
( $n$  = Zahl der Prozesse)

# Ein anderer verteilter Mutex-Algorithmus (Ricart / Agrawala, 1981)

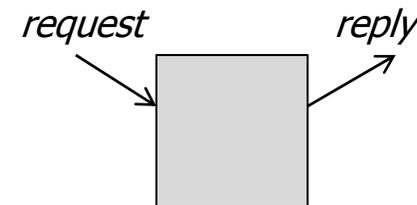
- Nur  $2(n-1)$  Nachrichten (Reply übernimmt Rolle von Release und ack)



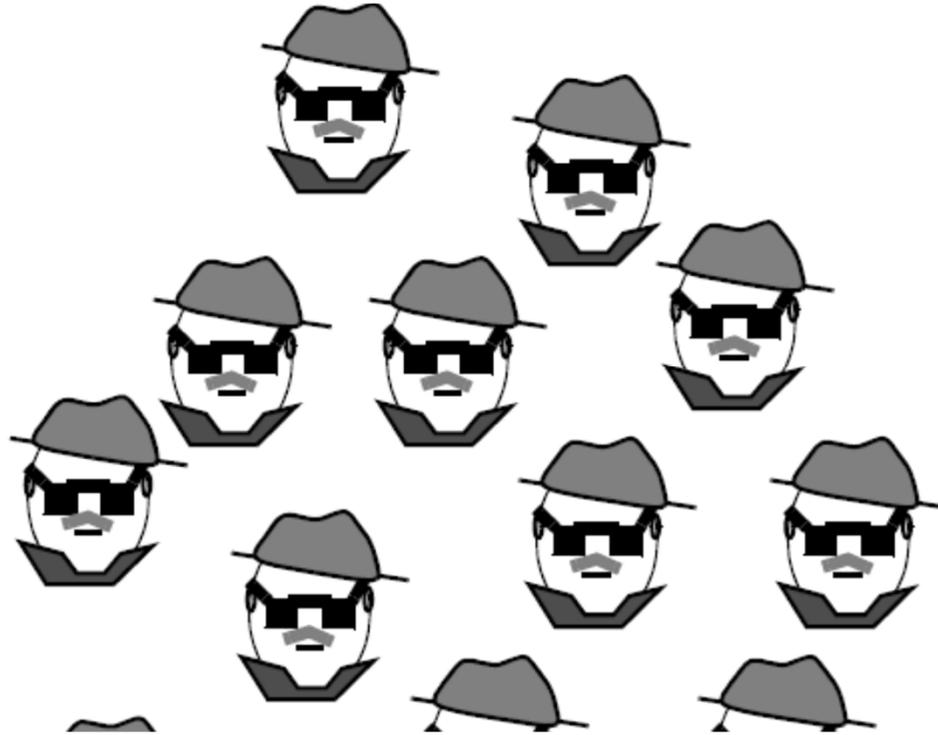
1. **Sende Request** (mit log. Zeitstempel!) an alle  $n-1$  anderen
2. Dann auf  $n-1$  **Replies warten**, danach Betriebsmittel nutzen

- Bei **Eintreffen einer Request-Nachricht**:

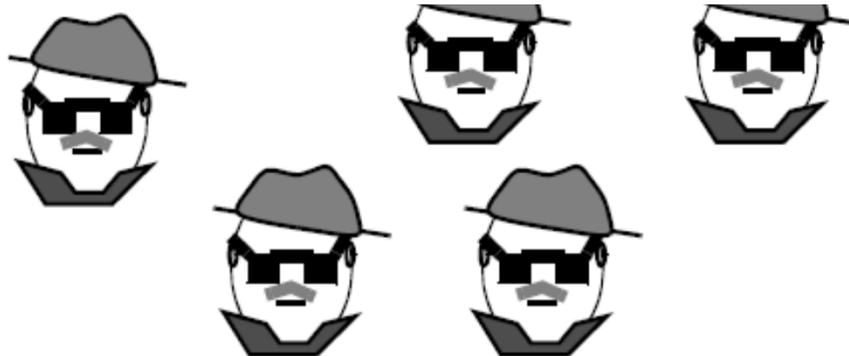
- wenn nicht selbst beworben oder der Sender „ältere Rechte“ (bzgl. **logischer Zeit!**) hat, dann **Reply sofort** schicken
  - ansonsten **Reply erst später** (im Sinne von Release, s.o.) schicken, nach Erfüllen des eigenen Requests (d.h. dem exklusivem Zugriff)
- Nur älteste Bewerbung setzt sich überall durch!



Denkübungen: Safety? Fairness? Deadlockfreiheit? FIFO notwendig?

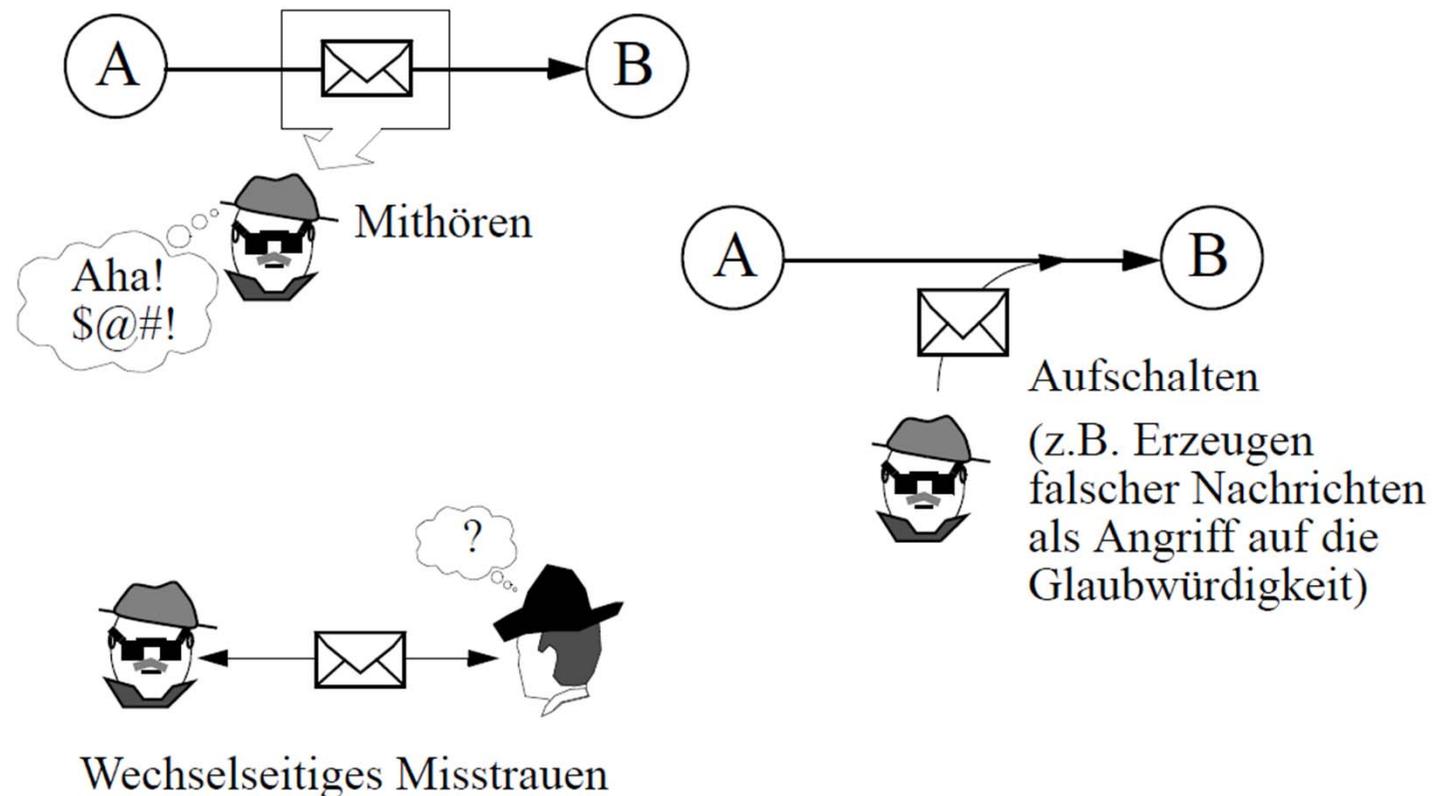


# Sicherheit

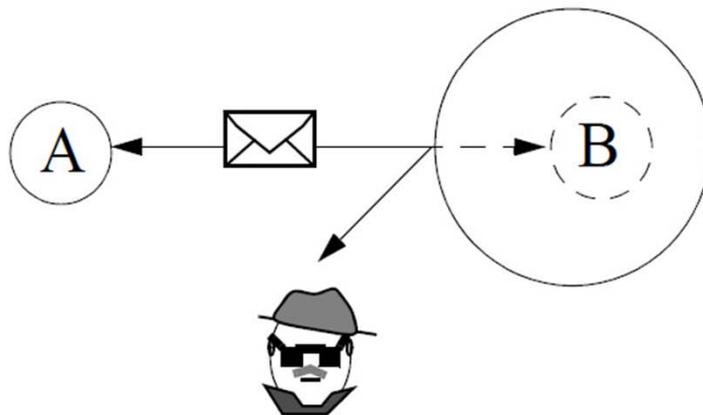
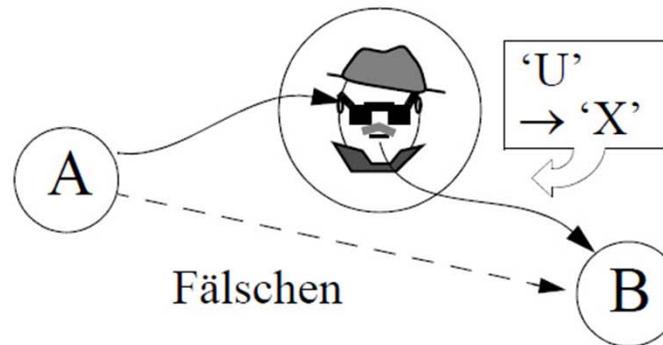


# Sicherheit in verteilten Systemen

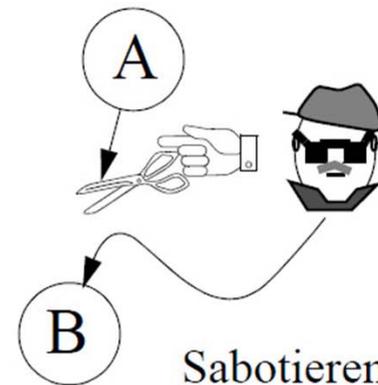
## Problemaspekte und Angriffsszenarien:



# Sicherheit in verteilten Systemen (2)



Vorspiegeln falscher Identität



Sabotieren

# Sicherheit: Anforderungen



- **Autorisierung / Zugriffsschutz**
  - Einschränkung der Nutzung auf den Kreis der Berechtigten
- **Vertraulichkeit**
  - Daten / Nachrichteninhalte gegen Lesen Unberechtigter schützen
  - Kommunikationsverhalten (wer mit wem etc.) geheim halten
- **Authentizität**
  - Absender "stimmt" (z.B. Server ist der, für den er sich ausgibt)
  - Daten sind "echt" und aktuell (→ Integrität)
- **Integrität**
  - Wahrung der Unversehrtheit von Nachrichten, Programmen, Daten
- **Verfügbarkeit** der wichtigsten Dienste
  - keine Zugangsbehinderung ("denial of service") durch andere
  - kein provozierter Absturz ("Sabotage")

# Weitergehende Anforderungen



Wie zum Beispiel:

- Nichtabstreitbarkeit
- Accountability
- Strafrechtliche Verfolgbarkeit
  - Protokollierung
  - „Key Escrow“
- Konformität zu rechtlich / politischen Vorgaben
- ...

# Sicherheit: Verteilungsaspekte



- **Offenheit** in verteilten Systemen begünstigt Angriffe
  - grosse Systeme → **vielfältige Angriffspunkte**
  - standardisierte Kommunikationsprotokolle → Angriff **einfach**
  - räumliche Distanz → Ortung des Angreifers schwierig, Angriff **sicher**
  - breiter Einsatz, allgemeine Verwendung → Angriff **reizvoller**
  - physische Abschottung oft nicht durchsetzbar
  - technologische Gegebenheiten: z.B. Wireless LAN ("broadcast")
- **Heterogenität**
  - sorgt für zusätzliche Schwachstellen
  - erschwert Durchsetzung einer einheitlichen Schutzphilosophie
- **Dezentralität**
  - fehlende netzweite Sicherheitsautorität

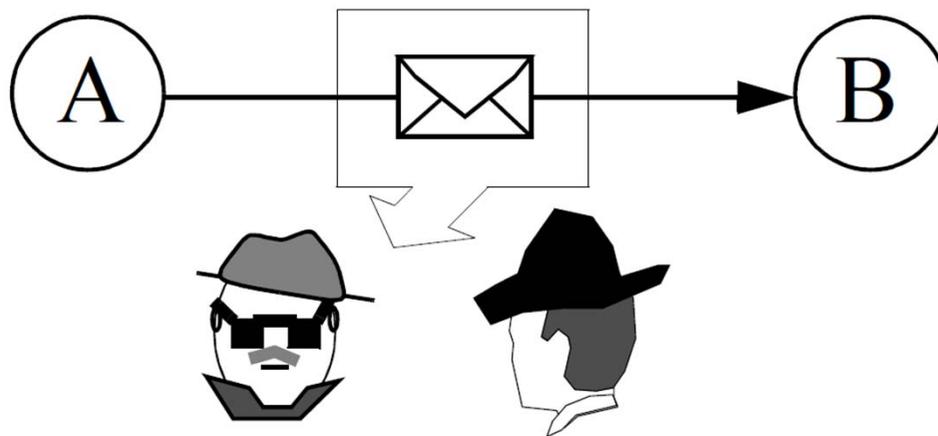
# Sicherheit: Verteilungsaspekte

- → Gewährleistung der Sicherheit ist in verteilten Systemen **wichtiger** und **schwieriger** als in alleinstehenden Systemen
- Typische **Techniken** und **“Sicherheitsdienste”**:
  - *Verschlüsselung*
  - *Autorisierung* (“der darf das!”)
  - *Authentisierung* (“X ist wirklich X!”)

Hierfür Kryptosysteme  
und Protokolle als  
“Security Service”

# Angriffsformen – Passive Angriffe

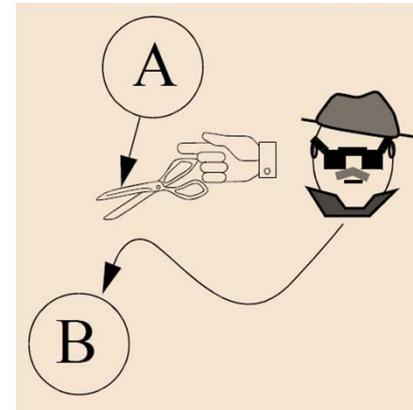
- Beobachten der Kommunikation
  - **Nachrichteninhalt** in Erfahrung bringen
  - **Kommunikationsverhalten** analysieren (“wer mit wem wie oft?”)



→ Verschlüsselung  
→ Anonymisierung

# Aktive Angriffe: Eindringen, Vorsätzliche Täuschung etc.

- **Einbruch** (Zugangsschranken durchbrechen)
  - Diebstahl von Daten, Nachrichteninhalten,...
- **Verändern** des Nachrichtenstroms
  - Ändern, Vernichten, Erzeugen, Vertauschen, Verzögern, Wiederholen ("replay") von Nachrichten
- **Falsche Identität** vorspiegeln
  - Maskerade: Nachahmen anderer Prozesse oder Nutzung eines fremden Passwortes
- **Missbräuchliche Nutzung** von Diensten
- **Denial of Service** durch Sabotage oder Verhindern des Dienstzugangs, z.B. durch Überfluten mit Nachrichten



# Authentizität



Seid auf eurer Hut vor dem Wolf; wenn er hereinkommt, so frisst er euch alle mit Haut und Haar. Der Bösewicht verstellt sich oft, aber an seiner rauhen Stimme und seinen schwarzen Füßen werdet ihr ihn gleich erkennen.  
(*Der Wolf und die sieben Geisslein*, grimmsche Märchen)

- **Authentizität** ist **elementar** und essentiell für die Sicherheit eines verteilten Systems
  - zu authentischen Nachrichten / Daten vgl. auch "Integrität"
- Authentizität eines **Subjekts**
  - Kommunikationspartner, Client, Server,...
  - ist der andere wirklich der, der er vorgibt zu sein?
  - z.B.: darf ich als Server daher ihm (?) den Zugriff gewähren?

# Authentizität (2)



- Authentizität eines **Dienstes**
  - Bsp.: Handelt es sich wirklich um den Druckdienst oder um einen böswilligen Dienst, der die Datei ausserdem noch heimlich kopiert?
- Authentizität einer **Nachricht**
  - hat mein Kommunikationspartner dies wirklich so gesagt?
  - soll ich als Geldautomat wirklich so viel Geld ausgeben?
- Authentizität **gespeicherter Daten**
  - ist dies wirklich der Vertragstext, den wir gemeinsam elektronisch hinterlegt haben?
  - hat der Autor Casimir von Hinkelstein wirklich *das* geschrieben?
  - ist das Foto nicht eine Fälschung?
  - ist dieser elektronische Schlüssel wirklich echt?

# Hilfsmittel zur Authentifizierung



- Wahrung der **Nachrichten-Authentizität**
  - **Verschlüsselung**, so dass inhaltliche Änderungen auffallen (Signatur, kryptograph. Hashcode / Prüfsumme)
  - Beachte: Authentizität des Nachrichteninhalts garantiert nicht Authentizität der Nachricht als solche! (z.B. **Replay-Attacke**: Neuversenden einer früher abgehörten Nachricht)
  - Massnahmen gegen Replays: z.B. mitcodierte Sequenznummer
- **Peer-Authentifizierung** z.B. mit Frage-Antwort-Spiel
  - "**challenge / response**": Antworten sollte nur der echte Kommunikationspartner kennen
  - idealerweise stets neue Fragen verwenden (Replay-Attacken!)

# Passwörter zur Authentifizierung

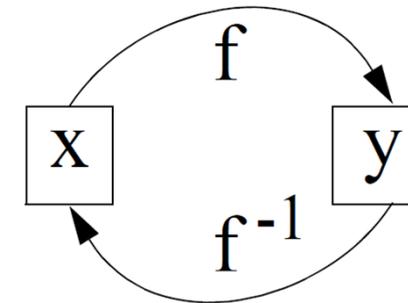


- **Authentifizierung mit Passwort**
  - typischerweise zur Authentifizierung eines Benutzers ("Client") zum Schutz des Dienstes vor unbefugter Benutzung (Autorisierung)
  - Kenntnis des Passworts gilt als **Identitätsbeweis** (gerechtfertigt?)
- **Potentielle Schwächen von Passwörtern**
  - **Geheimhaltung** (Benutzer kann Passwörter "verleihen" etc.)
    - Angriffsformen: Raten bzw. systematische Suche ("dictionary attack")
    - Gegenmassnahmen: Zurückweisung zu "simpler" Passwörter,
    - Zeitverzögerung nach jedem Fehlversuch und Maximalzahl von Fehlversuchen,
    - security logs
  - **Abhörgefahr** (möglichst keine Übermittlung im Klartext; Speicherung nur in codierter Form, so dass Invertierung prakt. unmöglich)
  - **Replay-Attacke** (Gegenmassnahme: Einmalpasswörter)

# Einwegfunktionen

- Bilden die **Basis vieler kryptographischer Verfahren**

- Prinzip:  $y = f(x)$  **einfach** aus  $x$  berechenbar, aber  $x = f^{-1}(y)$  ist extrem **schwierig** aus  $y$  zu ermitteln
  - $f^{-1}$  typw. exponentielle Zeitkomplexität
  - zeitaufwändig ( $\rightarrow$  praktisch undurchführbar)



z.B.  $f = O(n), O(n \log n), \dots$   
aber  $f^{-1} = O(2^n)$

- Es gibt (noch) **keinen mathematischen Beweis**, dass Einwegfunktionen überhaupt **existieren**
  - aber einige Funktionen sind es *allem Anschein nach*

# Einwegfunktionen (2)

- Einwegfunktionen mögen zunächst **sinnlos erscheinen** :



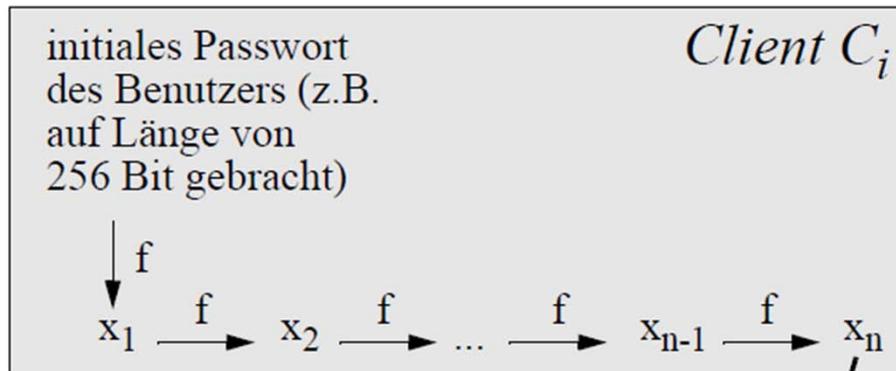
Ein zu  $y = f(x)$  verschlüsselter Text  $x$  kann nie wieder entschlüsselt werden!

- Aber: Einwegfunktionen mit **“trap-door”**  
(ein **Geheimnis**, das es erlaubt,  $f^{-1}$  **effizient** zu berechnen)
  - Idee: Nur der “Besitzer” oder “Erfinder” von  $f$  kennt dieses
  - Beispiel **Briefkasten**: Einfach etwas hineinzutun; schwierig etwas herauszuholen; mit Schlüssel (= Geheimnis) ist das aber einfach!
  - Anwendung z.B. **Public-Key-Verschlüsselung** oder **Einmalpasswörter**

# Prinzipien typischer (vermuteter) Einwegfunktionen

- Das **Multiplizieren** zweier (grosser) Primzahlen  $p, q$  ist effizient; das Zerlegen einer Zahl (z.B.  $n = pq$ ) in ihre **Primfaktoren** i.Allg. schwierig (d.h., sehr aufwändig)
- In einem Restklassenring (mod  $m$ ) ist die Bildung der **Potenz  $a^k$**  einfach; die  **$k$ -te Wurzel** oder den (diskreten) **Logarithmus** zu berechnen, ist i.Allg. schwierig.
  - aber:  $k$ -te Wurzel einfach, wenn Primzerlegung von  $m = pq$  bekannt  $\rightarrow$  trap-door!

# Einmalpasswörter mit Einwegfunktionen

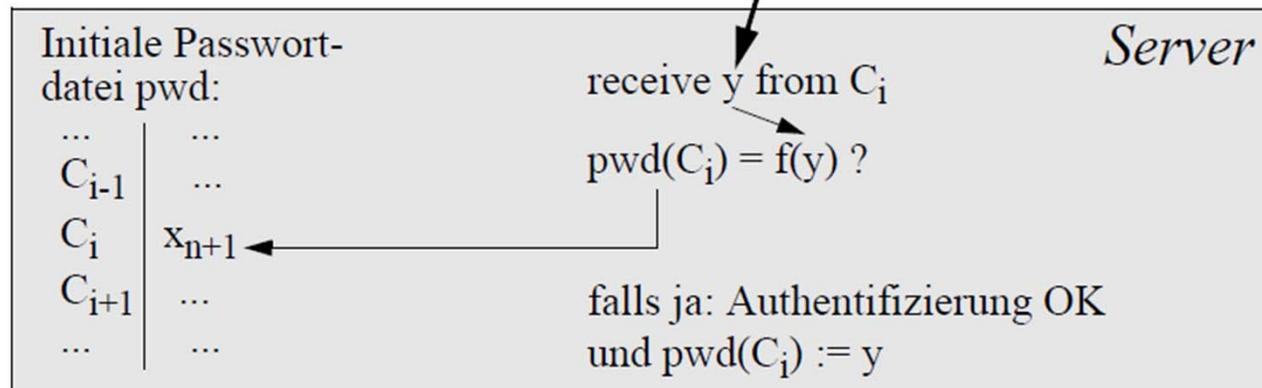


Durch iterierte Anwendung einer (bijektiven) Einwegfunktion  $f$  wird (quasi auf Vorrat) eine **Liste von  $n$  Einmalpasswörtern  $x_1 \dots x_n$**  erzeugt

Kommunikation über das Netz ist unsicher!



Zunächst wird  $x_n$  verwendet, beim nächsten **Mal  $x_{n-1}$** , dann  $x_{n-2}$  etc.



# Einmalpasswörter mit Einwegfunktionen: Eigenschaften

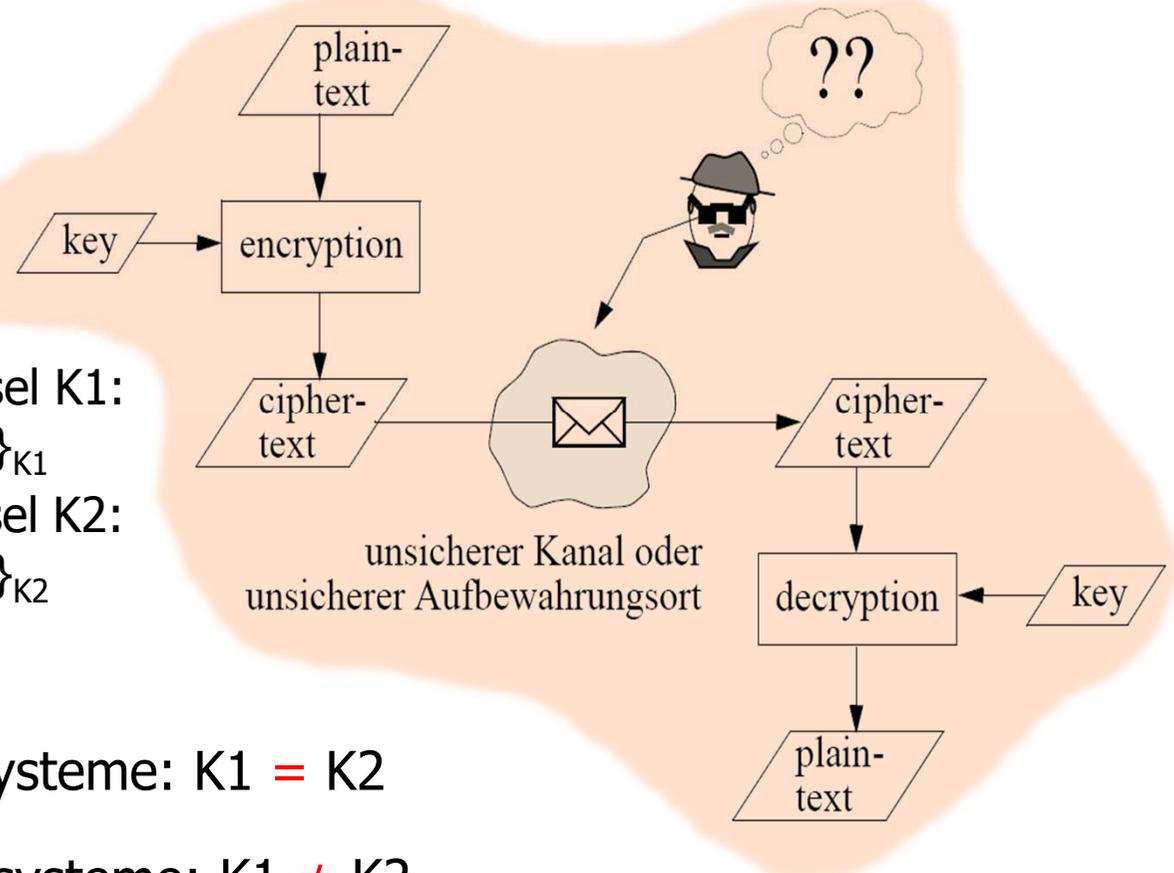
- Ein **abgehörtes Passwort**  $x_i$  **nützt nicht viel**
  - Berechnung von  $x_{i-1}$  aus  $x_i$  ist (praktisch) nicht möglich
- Ein **Lesen der Passwortdatei** des Servers ist **nutzlos**
  - dort ist nur das *vergangene* Passwort vermerkt
- Einwegfunktion **f muss nicht geheimgehalten werden**

Verfahren ist z.B. im **S/KEY-System** (RFC 1760) realisiert

# Kryptosysteme

## Schreibweisen:

- **Verschlüsseln** mit Schlüssel K1:  
Schlüsseltext = {Klartext}<sub>K1</sub>
- **Entschlüsseln** mit Schlüssel K2:  
Klartext = {Schlüsseltext}<sub>K2</sub>
  
- **Symmetrische** Kryptosysteme: K1 = K2
- **Asymmetrische** Kryptosysteme: K1 ≠ K2



# Kryptosysteme (2)



- **Forderung an das Verfahren:**  
Verschlüsselung ist ohne Kenntnis der Schlüssel höchstens mit unverhältnismässig hohem Rechenaufwand umkehrbar
- **Geheimhalten eines konkreten Verschlüsselungsverfahrens** stellt i.Allg. keinen Sicherheitsgewinn dar
  - organisatorisch oft nicht lange durchhaltbar
  - kein öffentliches Feedback über erkannte Schwächen des Verfahrens
  - Verfahren, die Geheimhaltung nötig hätten, erscheinen "verdächtig"

# Symmetrische Schlüssel



- **Nachteile symmetrischer Schlüssel:**
  - Schlüssel muss geheimgehalten werden (da Verfahren i.Allg. bekannt)
  - mit allen Kommunikationspartnern separaten Schlüssel vereinbaren
  - hohe Komplexität der **Schlüsselverwaltung** bei vielen Teilnehmern
  - Problem des **geheimen Schlüsselaustausches**
- **Vorteile symmetrischer Schlüssel:**
  - ca. 100 bis 1000 Mal schneller als typ. asymmetrische Verfahren
- **Beispiele** für symmetrische Verfahren:
  - **IDEA** (International Data Encryption Algorithm):  
128-Bit Schlüssel, Einsatz in PGP
  - **DES** (Data Encryption Standard) bzw. „Triple DES“ (TDES)
  - **AES** (Advanced Encryption Standard) als Nachfolger von DES

# One-Time Pads

- “Perfektes” symmetrisches Kryptosystem
  - Denkübung: Wieso und unter welchen Voraussetzungen?

|           |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|-----------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Klartext  | V   | E  | R  | T  | E  | I  | L  | T  | E  |    | S  | Y  | S  | T  | E  | M  | E  |
| in ASCII  | 56  | 45 | 52 | 54 | 45 | 49 | 4C | 54 | 45 | 20 | 53 | 59 | 53 | 54 | 45 | 4D | 45 |
|           | XOR |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Schlüssel | 4C  | 93 | EF | 20 | B7 | 55 | 92 | 7C | DA | 69 | 23 | F8 | BB | 72 | 0E | 81 | 00 |
| = Chiffre | 1A  | D6 | BD | 74 | F2 | 1C | DE | 28 | 9F | 49 | 70 | A1 | E8 | 26 | 4B | CC | 45 |

- Prinzip: Wähle zufällige Sequenz von Schlüsselbits
  - Verschlüsselung: Schlüsseltext = Klartext XOR Schlüsselbitsequenz
  - Entschlüsselung: Klartext = Schlüsseltext XOR Schlüsselbitsequenz
  - Begründung:  $(a \text{ XOR } b) \text{ XOR } b = a$  (für alle Bitbelegungen von a, b)

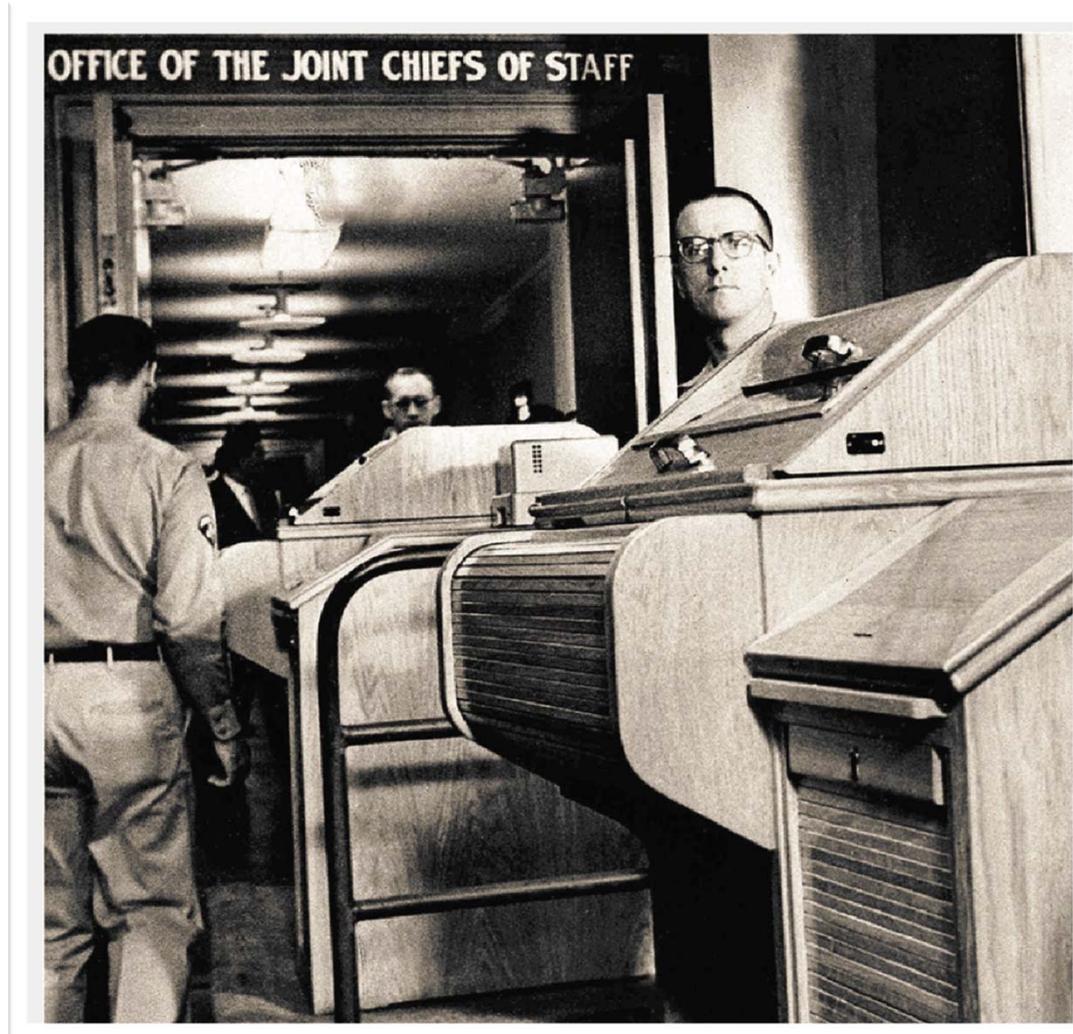
# One-Time Pads (2)



- Anforderungen an Schlüsselbitsequenz:
  - keine **Wiederholung** von Bitmustern (→ Schlüssellänge = Klartextlänge)
  - Schlüsselbitsequenz **ohne Bildungsgesetz** ("echte" Zufallsfolge )
  - Schlüsselbitsequenz wirklich "one-time" (**keine Mehrfachverwendung!**)
- Kryptoanalyse ohne Schlüsselkenntnis unmöglich
- **Nachteile** von One-Time Pads:
  - Verwendung **unhandlich** (hoher Bedarf an frischen Schlüsselbits, dadurch aufwändiger Schlüsselaustausch)
  - **Synchronisation**problem bei Übertragungsstörungen (wenn Empfang ausser Takt gerät, ist Folgetext verloren)
  - → nur für hohe Sicherheitsanforderungen üblich (z.B. "rotes Telefon")



# Rotes „Telefon“



Der „heisse Draht“ zwischen Washington und Moskau wird am **26. August 1963** installiert; es handelt sich um Fernschreib-einrichtungen (**Telex**) mit Kabel via London und Helsinki.

Ständige Prüfnachricht war „**the quick brown fox jumps over the lazy dog 1234567890**“.

Ab 1971 auch Kommunikation via **Satellit**, seit 2008 auch **Glaserfaserkabel**.

Seit 1980 wird **Fax** benutzt, seit 2010 auch **E-Mail**.



# Rotes „Telefon“



Der „heisse Draht“ zwischen Washington und Moskau wird am **26. August 1963** installiert; es handelt sich um Fernschreib-einrichtungen (**Telex**) mit Kabel via London und Helsinki.

Ständige Prüfnachricht war „**the quick brown fox jumps over the lazy dog 1234567890**“.

Ab 1971 auch Kommunikation via **Satellit**, seit 2008 auch **Glaserfaserkabel**.

Seit 1980 wird **Fax** benutzt, seit 2010 auch **E-Mail**.

Die Kenngruppen dürfen nur einmal zur Bildung eines Spruchschlüssels verwendet werden.

(1) Der Tagesschlüssel ist ein Zeitschlüssel und hat jeweils 24 Stunden Gültigkeit. Der Schlüsselwechsel erfolgt 00.01 Uhr.

(2) Der Spruchschlüssel hat nur Gültigkeit für jeweils einen Spruch.

(3) Tagesschlüsseltabellen und Schlüssellockkarten, die auf Grund von Beschädigungen oder Kompromittierung nicht benutzt wurden, sind der Leitstelle (für die Chiffrier-  
verbindung verantwortliche Chiffrierstelle) des Schlüsselbereiches chiffriert oder mittels Kurier zu melden. Der neue Tagesschlüssel (bzw. die neue Folge der Tagesschlüssel) wird durch die Leitstelle chiffriert oder mittels Kurier angewiesen.

(4) Alle Schlüsselmittel, einschließlich entnommene, zur Bearbeitung nichtbenutzte bzw. nichtverwendbare, sind bis zur Vernichtung so aufzubewahren, daß ein Zugriff durch unbefugte Personen ausgeschlossen ist.

(5) Wenn nicht anders angewiesen, sind zur Bearbeitung benutzte, entnommene unbenutzte bzw. nichtverwendbare Tages-  
schlüsseltabellen, Schlüssellockkarten, Spruchschlüssel-  
tabellen und vollständig aufgebrauchte Kenngruppentafeln  
innerhalb von 48 Stunden zu vernichten.

(6) Die Vernichtung von Schlüsselmitteln ist lückenlos mit  
Datum und durch zwei Unterschriften nachzuweisen. *Der Nachweis der*  
*vernichtung der Tagesschlüsseltabellen, Schlüssellockkarten und Spruchschlüssel-*  
*tabellen hat in der Entnahmetabelle zu erfolgen.*

# Asymmetrische Kryptosysteme

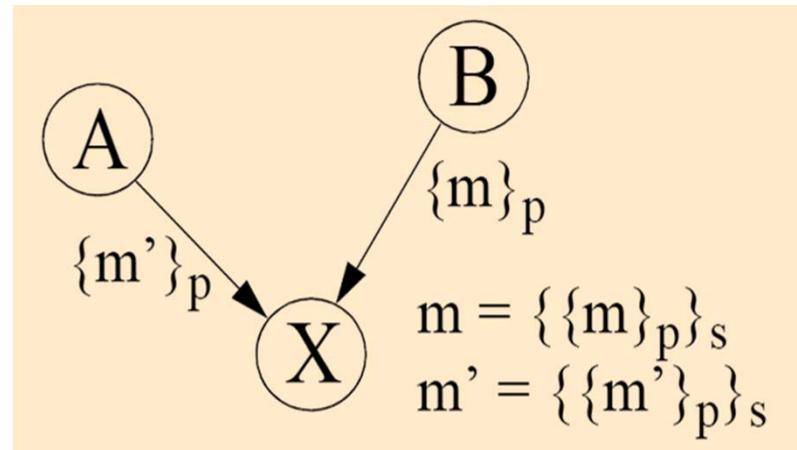
Schlimm sind die Schlüsseln,  
die nur schliessen auf, nicht zu;  
Mit solchem Schlüsselbund im Haus  
verarmst du.

*Friedrich Rückert, Die Weisheit des Brahmanen*

- Schlüssel zum Ver- / Entschlüsseln sind verschieden
  - z.B. **RSA-Verfahren** (Rivest, Shamir, Adleman, 1978),  
beruht auf der Schwierigkeit von Faktorisierung
  - andere Verfahren beruhen z.B. auf diskreten Logarithmen
- Für jeden Prozess X existiert ein Paar **(p,s)**
  - **p** = *public key*  
(zum *Verschlüsseln* von Nachrichten an X)
  - **s** = *secret key* (oder: private key)  
(zum *Entschlüsseln* von mit p verschlüsselten Nachrichten)

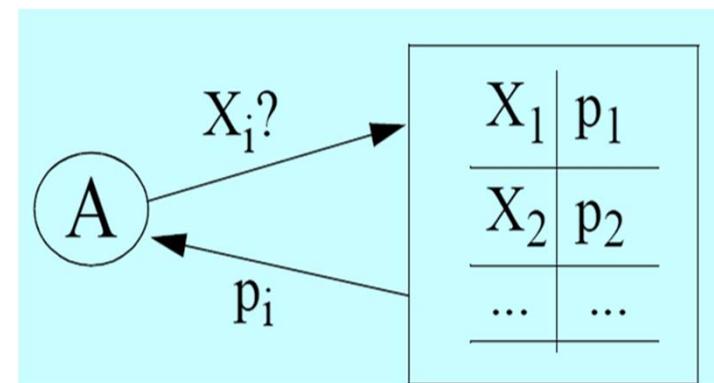
# Asymmetrische Kryptosysteme

- Jeder Prozess, der an X sendet, kennt  $p$
- Nur X selbst kennt  $s$



- **Public-Key-Server**

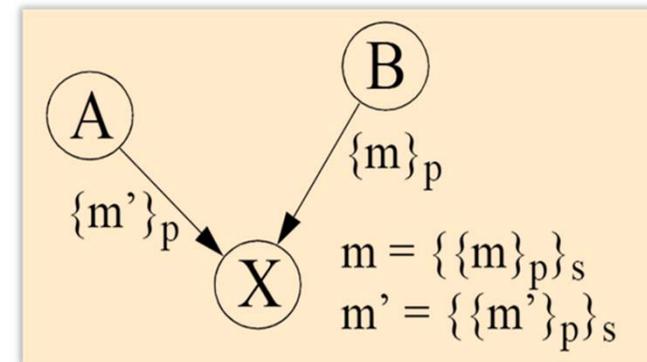
- Welchen Schlüssel hat Prozess  $X_i$ ?
- Server muss **vertrauenswürdig** sein
- Kommunikation zum Server darf nicht manipuliert sein



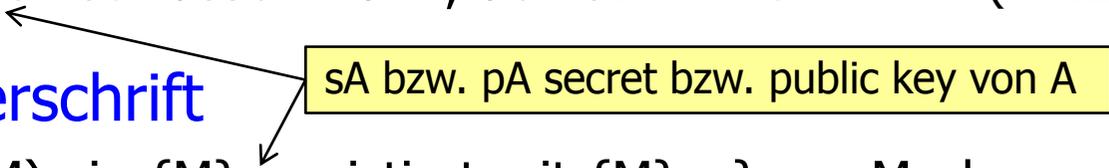
# Asymmetrische Kryptosysteme

## Gewünschte Eigenschaften:

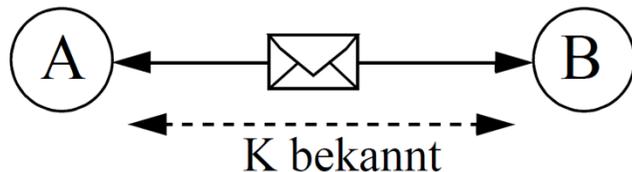
- 1)  $m$  lässt sich nicht allein aus  $\{m\}_p$  ermitteln
- 2)  $s$  lässt sich aus  $p$  oder einer verschlüsselten, bekannten Nachricht nicht (mit vertretbarem Aufwand) ableiten
- 3)  $m = \{\{m\}_p\}_s$
- 4) Evtl. zusätzlich:  $m = \{\{m\}_s\}_p$   
(Rolle von Verschlüsselung und Entschlüsselung austauschbar)
- Resistenz gegen „Chosen-Plaintext“-Angriff (beliebige Nachrichten  $M$  und deren Verschlüsselung  $\{M\}_p$  sind jederzeit generierbar)



# Vorteil asymmetrischer Verfahren gegenüber symmetrischen

- Vereinfachter Schlüsselaustausch
    - jeder darf den übermittelten public key  $p$  mithören
    - secret key  $s$  braucht grundsätzlich nie Dritten mitgeteilt zu werden
    - bei  $n$  Teilnehmern genügen  $2n$  Schlüssel (statt  $O(n^2)$  bei sym. Verf.)
  - Kenntnis von  $s$  authentifiziert zugleich den Besitzer
    - "wer  $\{M\}_{pA}$  entschlüsseln kann, der ist wirklich A" (wirklich?)
  - Digitale Unterschrift
    - "wenn (zu  $M$ ) ein  $\{M\}_{sA}$  existiert mit  $\{\{M\}_{sA}\}_{pA} = M$ , dann muss dies ( $M$  bzw.  $\{M\}_{sA}$ ) von A erzeugt worden sein" (wieso?)
- 

# Authentifizierung mit symmetrischen Schlüsseln



Sei  $K$  der zwischen A und B vereinbarte (und geheimzuhaltende!) Schlüssel

- Problem: B soll die **Authentizität von A** feststellen
  - *Idee* (Geheimdienstprinzip): "Wenn X *das* weiss und kann, dann muss X wirklich X sein, denn sonst weiss und kann das niemand"
  - *Bemerkung*: Oft ist eine **gegenseitige Authentifizierung** nötig

- **1. Verfahren:**

A:  $m :=$  "Ich bin A"

$m' := \{m\}_K$

A  $\rightarrow$  B:  $m', m$

B: überprüfe, ob  $\{m\}_K = m'$

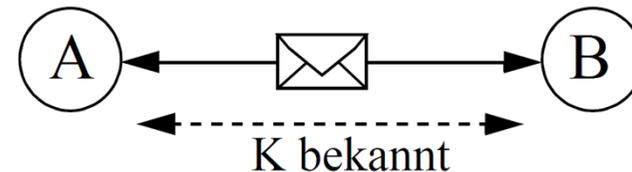
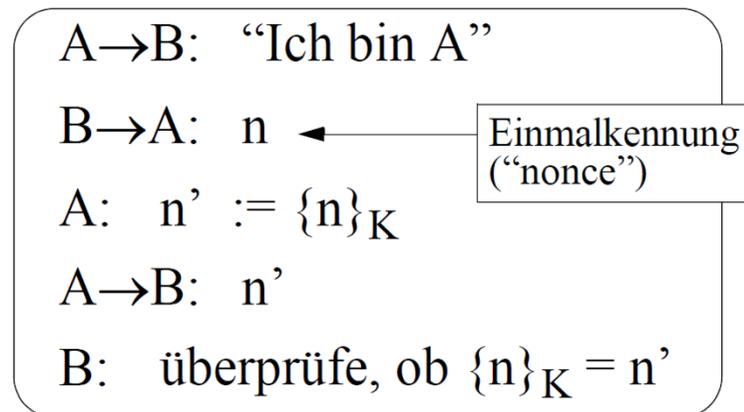
Damit B den richtigen Schlüssel (für A) wählt

- *Idee*: Überprüfe die Fähigkeit, Nachrichten mit einem geheimen Schlüssel zu kodieren

- *Nachteil*: Möglichkeit von replays durch Abhören

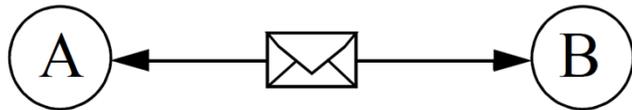
# Authentifizierung mit symmetrischen Schlüsseln

- 2. Verfahren:

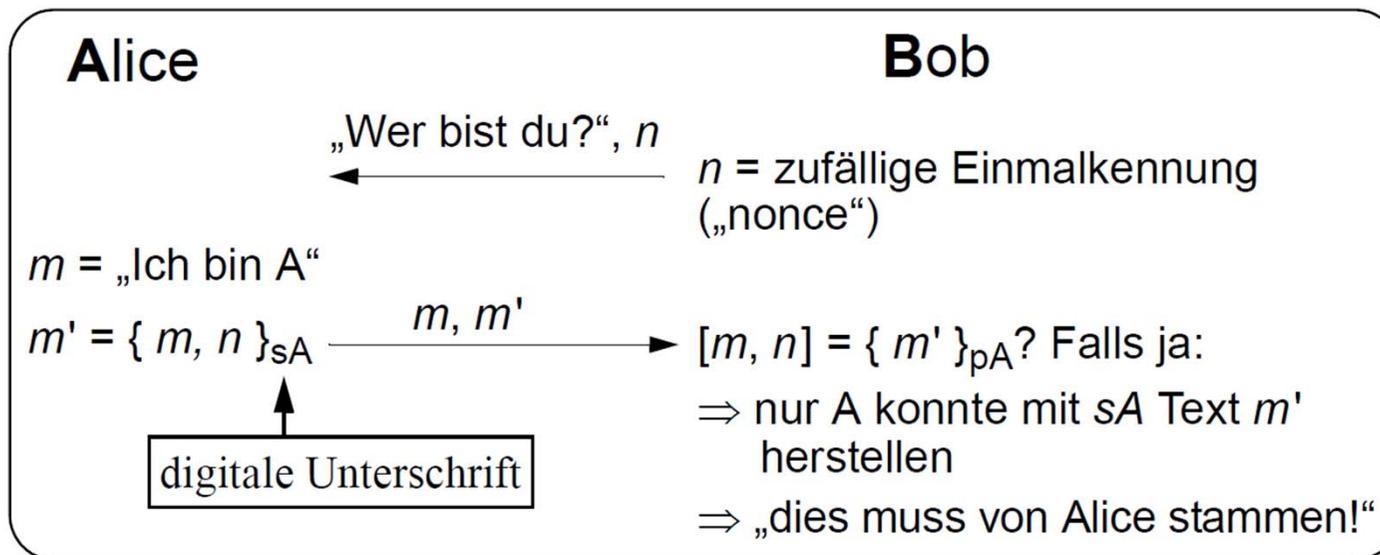


Genereller **Nachteil** bei symmetrischen Schlüsseln: **Viele individuelle Schlüsselpaare** für jede Client / Server-Beziehung

# Authentifizierung mit asymmetrischen Schlüsseln



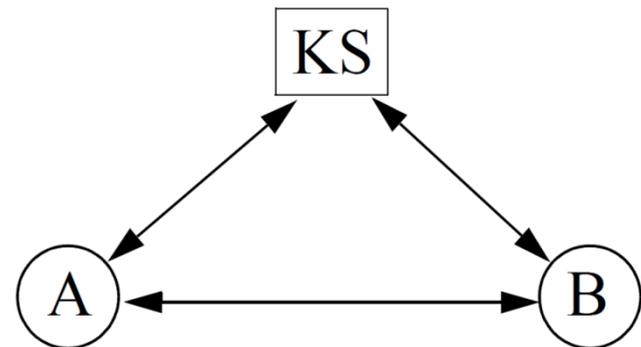
Notation:  $sX$  = secret key von X;  
 $pX$  = public key von X



- Geschützt gegen **Replays** (wieso?)
- Vorsicht: **“Man in the middle”**-Angriff möglich (wie?)

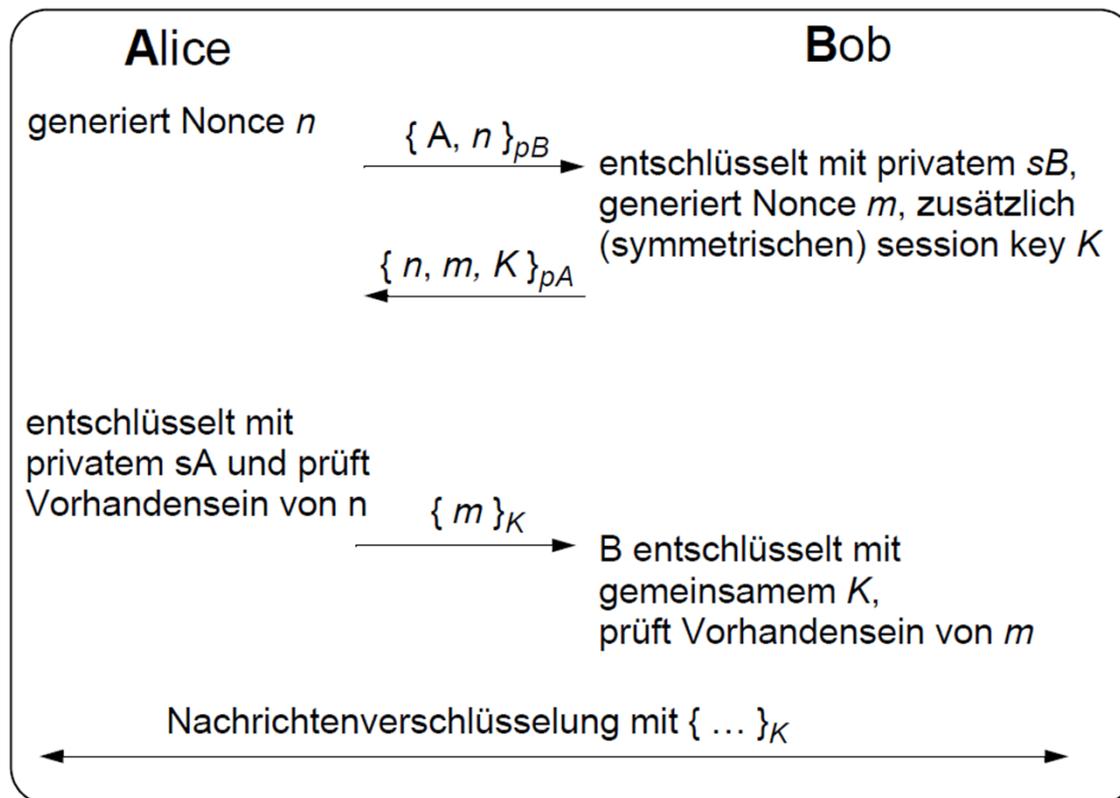
# Einbezug eines Schlüsselservers

- Nachteil obiger Lösung: B muss viele public keys speichern
- Alternativ mit **Key Server** KS:  
dieser kennt alle public keys
  - B erfragt public key von A bei KS
  - **KS signiert** alle seine Nachrichten
  - jeder kennt public key von KS (um Unterschrift von KS zu verifizieren)
- Angriff auf den Schlüsselservers KS liefert **keine Geheimnisse**; erlaubt aber u.U., in dessen Rolle zu schlüpfen und falsche Auskünfte zu geben!
- KS ist evtl. **repliziert** oder **verteilt**



# Gegenseitige Authentifizierung (mit Schlüsselvereinbarung)

- Im Prinzip wie oben beschrieben nacheinander in beide Richtungen möglich; effizienter **beides zusammen** erledigen



- Voraussetzung: A und B kennen die public keys  $p_B$  bzw.  $p_A$  des jeweiligen Partners
- Hier zusätzlich: Vereinbarung eines **symmetrischen "session keys"  $K$** , der nach der Authentifizierung zur effizienten Verschlüsselung benutzt wird

# Replays



- Generelles Problem: Auch wenn ein Angreifer eine **Nachricht** nicht entschlüsseln kann, kann er sie dennoch evtl. kopieren und **später wieder einspielen**
  - Autorisierungscode für Geldautomaten,...

Mögliche Gegenmassnahmen:

- 1) Verwendung von **Einmalkennungen**, die vom Empfänger vorgegeben werden ("nonce")
  - etwas aufwändigeres Challenge-Response-Protokoll
  - → alle relevanten Nachrichten sind verschieden

# Replays (2)



## 2) Verwendung von mitkodierte Sequenznummern

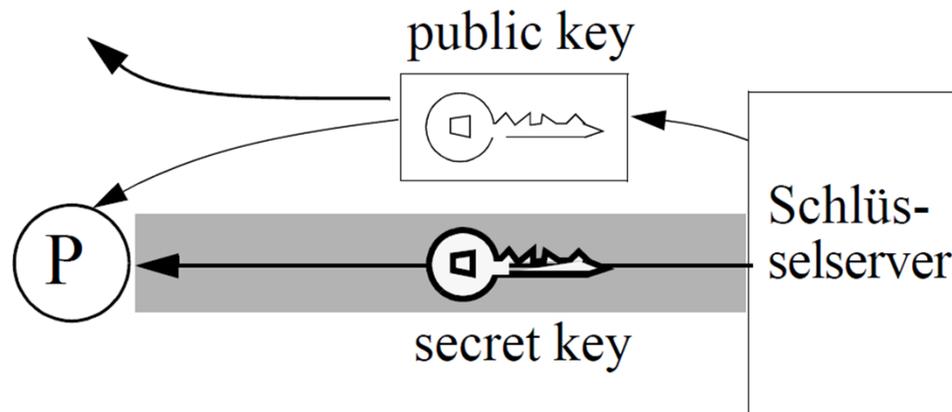
- nur bei einer Nachrichtenfolge zwischen 2 Prozessen möglich

## 3) Mitverschlüsseln der Absendezeit

- Empfänger akzeptiert Nachricht nur, wenn seine Zeit max.  $\Delta t$  abweicht
  - globaler Zeitservice bzw. gut synchronisierte Uhren nötig
  - Angreifer darf Zeitservice nicht manipulieren können
- Zeitfenster  $\Delta t$  geschickt wählen
  - Nachrichtenlaufzeiten berücksichtigen
  - *zu gross* → unsicher durch mögliche Replays
  - *zu klein* → falscher Alarm

# Schlüsselvergabe (public key)

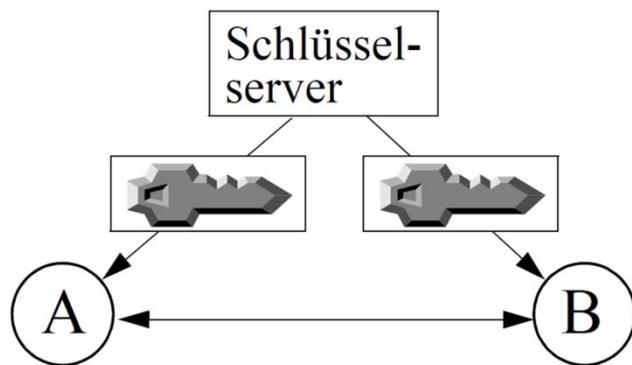
- Zur Erzeugung und **Verwaltung von Schlüsseln** existiert typischerweise ein eigener Dienst (mit **Schlüsselservers**)
- Zum Bsp. Vergabe eines Paares von **public / secret keys**:



- **Secret key** muss auf **sicherem Kanal** zu P gelangen
- Public key von P kann an beliebige Prozesse offen verteilt werden (jedoch i.Allg. "**zertifiziert**", dass der Schlüssel authentisch ist)

# Schlüsselvergabe (session key)

- Zur Generierung von temporären symmetrischen Schlüsseln ("session key"):



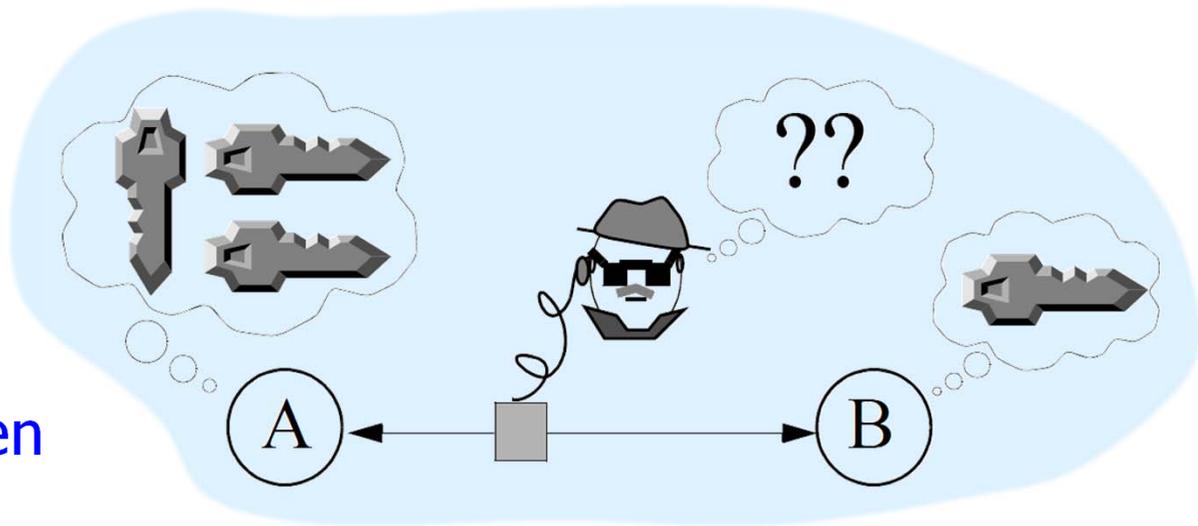
Z.B. bittet A den Schlüssels-server, ihm und B einen session key für die Kommunikation zwischen ihnen zu schicken

Session keys werden **sicher** und **authentisch**, z.B. mit einem Public-Key-Verfahren, an die Kommunikationspartner übertragen

- Schlüssels-server kann session key nach Übertragung bei sich löschen
- Aufwändiges Public-Key-Verfahren nur ein einziges Mal pro "Session", tatsächliche **Nachrichtenschlüsselung** auf dem Kanal zwischen A und B **dann effizient per symmetrischem Schlüssel**

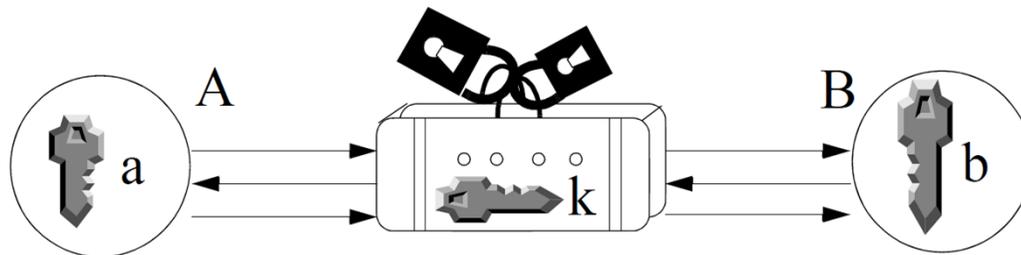
# Direkte Schlüsselvereinbarung?

- Problem: A und B wollen sich **ohne Schlüsselservers** über einen unsicheren Kanal auf einen (geheimen) **gemeinsamen Schlüssel** einigen



- Sinnvoll z.B. bei dynamisch gegründeten Prozessen, die vorher noch nie kommuniziert haben
  - z.B. wenn keine public keys vorhanden bzw. nicht bekannt
- Wie geht dies?
  - wir erinnern uns an die "Schatzkiste mit zwei Vorhängeschlössern"

# Direkte Schlüsselvereinbarung?



1. A generiert einen Sitzungsschlüssel  $k$
2. A verschlüsselt  $k$  mit einem geheimen Schlüssel  $a$
3.  $A \rightarrow B: \{k\}_a$  a und b sind "lokal erfunden"
4. B verschlüsselt dies mit seinem Schlüssel  $b$
5.  $B \rightarrow A: \{\{k\}_a\}_b$
6. A entschlüsselt mit seinem Schlüssel  $a$ :  
 $\{\{\{k\}_a\}_b\}_a = \{\{\{k\}_a\}_a\}_b = \{k\}_b$ 

Forderung!

Bezeichne  $\bar{x}$  den zu  $x$  inversen Schlüssel (oft:  $\bar{\bar{x}} = x$ )
7.  $A \rightarrow B: \{k\}_b$  gemeinsames Geheimnis
8. B entschlüsselt mit seinem Schlüssel:  $\{\{k\}_b\}_b = k$

Beachte:  $k$  wird nie offen transportiert!

Frage: Geht hier xor mit "one-time pads"  $a, b$  ?

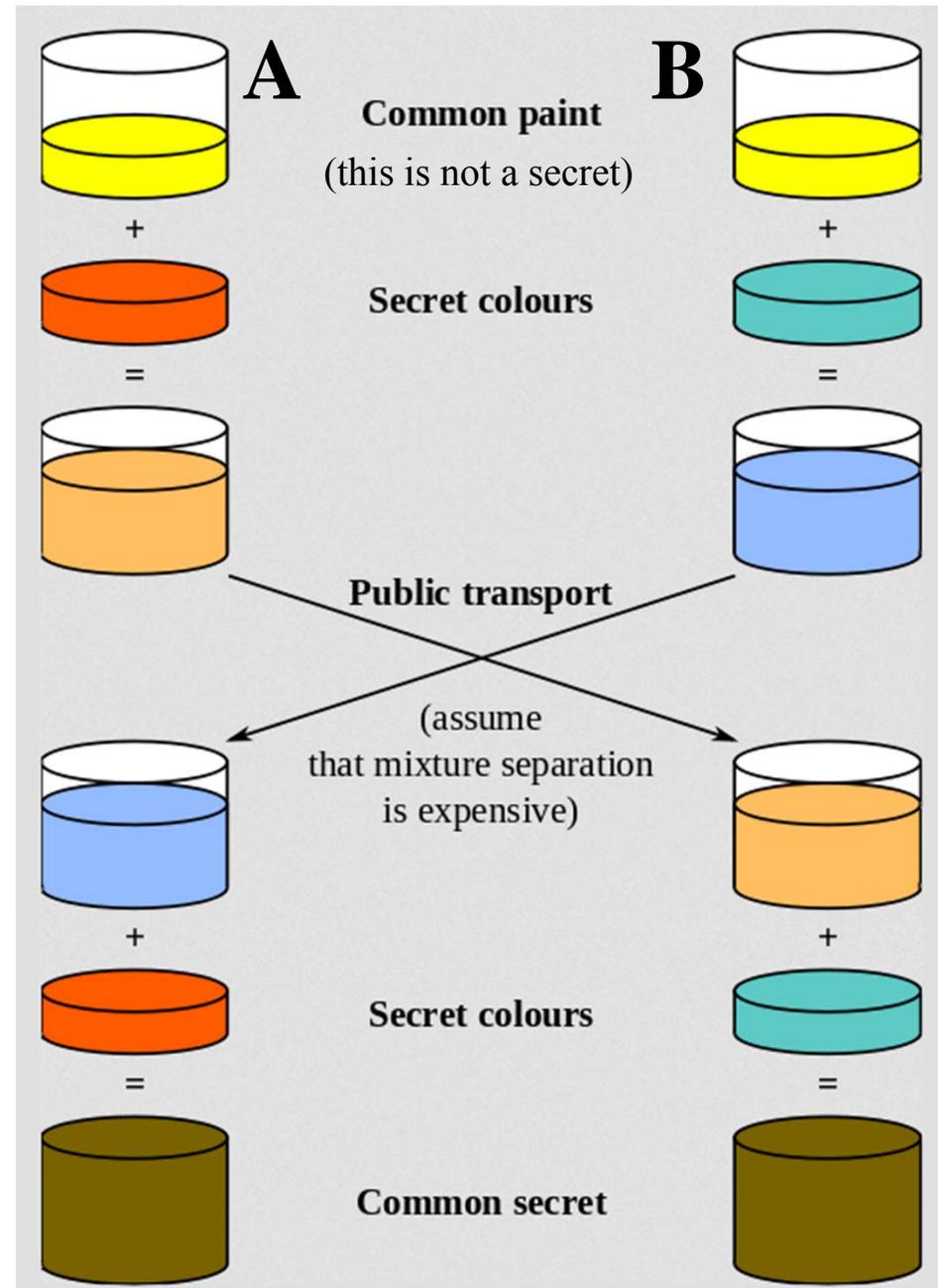
- xor erfüllt die Forderung (ist assoziativ und kommutativ)
- xor mit one-time pads ist sicher (wirklich?) und effizient

■ **Aber:** Wenn Schritt 3  $\{k\}_a$  und Schritt 5  $\{\{k\}_a\}_b$  abgehört wird, dann kann daraus der Schlüssel  $b$  ermittelt werden, so dass in Schritt 7 aus dem abgehörten  $\{k\}_b$  das geheime  $k$  ermittelt werden kann!

■ Geht anstelle von xor **etwas anderes**, das sicher ist?  
 → Idee des Diffie-Hellman-Verfahrens

# Diffie-Hellman-Verfahren

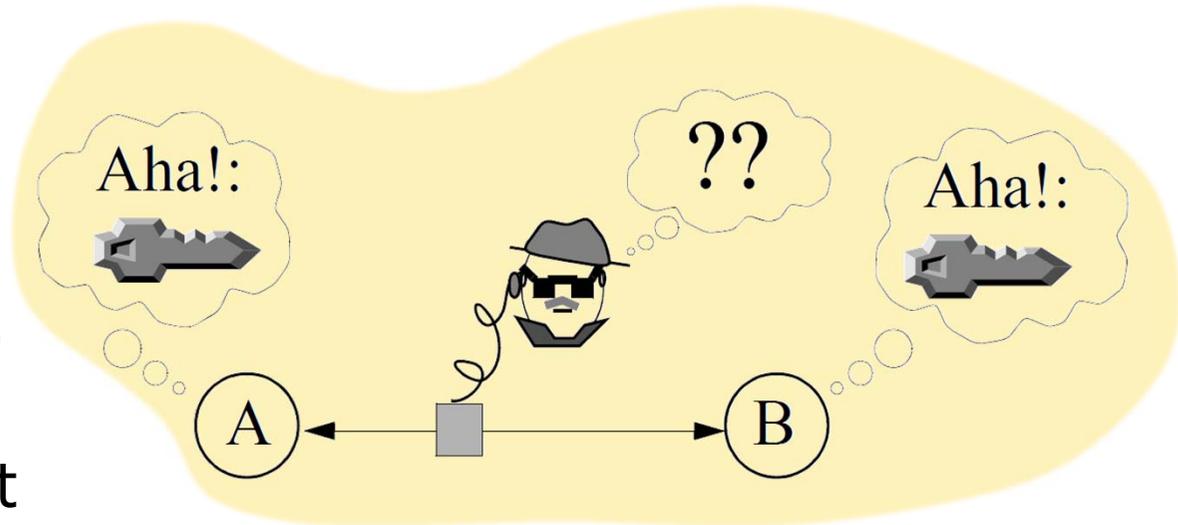
- Hier als **Gleichnis** in Form von Farbmischen
- Ziel: **A** und **B** sollen sich über einen unsicheren Kanal auf ein **gemeinsames "Geheimnis"** (hier: eine Farbe) einigen, ohne dass ein Angreifer es erfährt
  - Auch wenn der **Angreifer** aus den beiden übermittelten Farben eine **Probe mischt**?



Bildquelle: Wikipedia

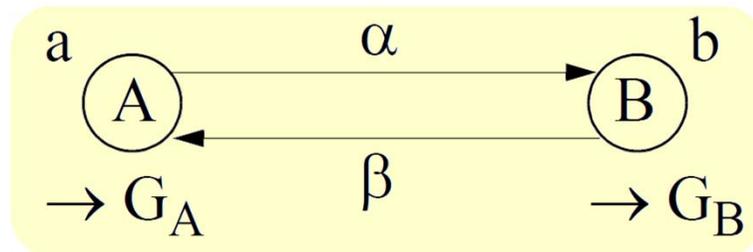
# Schlüsselvereinbarung mit dem Diffie-Hellman-Verfahren

- Ziel: **A** und **B** sollen sich über einen unsicheren Kanal auf ein **gemeinsames "Geheimnis" G** einigen, ohne dass ein Angreifer es erfährt



- Nutzung einer **Einwegfunktion**:  $f(x) = c^x \bmod p$   
( $1 < c < p$ ; wobei  $p$  eine grosse Primzahl ist)
  - in einem Restklassenring ist die Bestimmung diskreter Logarithmen (als **Umkehrfunktion**) **viel schwieriger** als die Bildung von Potenzen

# Der Diffie-Hellman-Algorithmus



- wenig Nachrichten
- effizient

Bem.: a und b sind nur lokal bekannt und bleiben geheim

1. A wählt eine Zufallszahl a
2. A berechnet  $\alpha = f(a)$
3. A  $\rightarrow$  B:  $\alpha$

4. B wählt eine Zufallszahl b
5. B berechnet  $\beta = f(b)$
6. B  $\rightarrow$  A:  $\beta$

7. A berechnet  $G_A = \beta^a \bmod p$
8. B berechnet  $G_B = \alpha^b \bmod p$

**Behauptung:**  $G_A = G_B$   
(gemeinsames Geheimnis!)

**Beispiel** (für  $c = 5$  und unrealistisch kleines  $p = 7$ ):

$$f(x) = 5^x \bmod 7$$

$$a = 3 \rightarrow \alpha = 6$$

$$b = 4 \rightarrow \beta = 2$$

$$\rightarrow G_B = 6^4 \bmod 7 = 1$$

$$\rightarrow G_A = 2^3 \bmod 7 = 1$$

$$G_A = G_B$$

Zu zeigen:  $\beta^a \bmod p = \alpha^b \bmod p$ , also:

$$(c^b \bmod p)^a \bmod p = (c^a \bmod p)^b \bmod p$$

*Lemma:*  $(k \bmod p)^n \bmod p = k^n \bmod p$  ← Restklassenarithmetik...

$$\begin{aligned} (c^b \bmod p)^a \bmod p &= (c^b)^a \bmod p && \text{[Lemma]} \\ &= c^{(b \cdot a)} \bmod p \\ &= c^{(a \cdot b)} \bmod p \\ &= (c^a)^b \bmod p && \text{[Lemma]} \\ &= (c^a \bmod p)^b \bmod p \end{aligned}$$

### Bemerkungen:

- Lässt sich auch auf  $k > 2$  Benutzer verallgemeinern
- Der Algorithmus (entdeckt 1976) ist patentiert (U.S.-Patent Nummer 4200770, Sept. 1977)

# Sweet Little Secret G

- A und B könnten  $G = G_A = G_B$  nun als symmetrischen **Schlüssel** zur Kodierung ihrer Nachrichten verwenden
  - Besser: G nur als „**master key**“ verwenden, um daraus (mittels Einwegfunktion) einen **session key** zu erzeugen
  - Motivation: G selbst so selten wie möglich benutzen
- Einzusehen bliebe noch, dass aus Kenntnis von  $\alpha$  und  $\beta$  (sowie von  $c$  und  $p$  aus  $f$ )  $G$  von einem Mitlauscher (in effizienter Weise) **nicht ermittelt werden kann**
  - $\alpha$  und  $\beta$  sind unabhängig voneinander (wieso ist das ein Argument?)
  - Bem.: nicht jedes  $p$  ist „gut“ und sollte auch einige 100 Bit lang sein

# United States Patent [19]

[11]

4,200,770

Hellman et al.

[45]

Apr. 29, 1980

## [54] CRYPTOGRAPHIC APPARATUS AND METHOD

[75] Inventors: **Martin E. Hellman**, Stanford; **Bailey W. Diffie**, Berkeley; **Ralph C. Merkle**, Palo Alto, all of Calif.

[73] Assignee: **Stanford University**, Palo Alto, Calif.

[21] Appl. No.: **830,754**

[22] Filed: **Sep. 6, 1977**

[51] Int. Cl.<sup>2</sup> ..... **H04L 9/04**

[52] U.S. Cl. .... **178/22; 340/149 R; 375/2; 455/26**

[58] Field of Search ..... **178/22; 340/149 R**

## [56] References Cited

### PUBLICATIONS

"New Directions in Cryptography", Diffie et al., *IEEE Transactions on Information Theory*, vol. IT-22, No. 6, Nov. 1976.

Diffie & Hellman, "Multi-User Cryptographic Techniques", *AFIPS Conference Proceedings*, vol. 45, pp. 109-112, Jun. 8, 1976.

*Primary Examiner*—Howard A. Birmiel  
*Attorney, Agent, or Firm*—Flehr, Hohbach, Test

## [57] ABSTRACT

A cryptographic system transmits a computationally secure cryptogram over an insecure communication channel without prearrangement of a cipher key. A secure cipher key is generated by the conversers from transformations of exchanged transformed signals. The conversers each possess a secret signal and exchange an initial transformation of the secret signal with the other converser. The received transformation of the other converser's secret signal is again transformed with the receiving converser's secret signal to generate a secure cipher key. The transformations use non-secret operations that are easily performed but extremely difficult to invert. It is infeasible for an eavesdropper to invert the initial transformation to obtain either conversers' secret signal, or duplicate the latter transformation to obtain the secure cipher key.

**8 Claims, 6 Drawing Figures**

## US4218582: Public key cryptographic apparatus and method

Issued/Filed Dates: Aug. 19, 1980 / Oct. 6, 1977

Abstract:

A cryptographic system transmits a **computationally secure** cryptogram that is generated from a **publicly known transformation** of the message sent by the transmitter; the cryptogram is again transformed by the authorized receiver using a **secret reciprocal transformation** to reproduce the message sent. The authorized receiver's transformation is known only by the authorized receiver and is used to generate the transmitter's transformation that is made publicly known. The publicly known transformation uses operations that are **easily performed but extremely difficult to invert**. It is infeasible for an unauthorized receiver to invert the publicly known transformation or duplicate the authorized receiver's secret transformation to obtain the message sent.

What is claimed is:

1. In a method of **communicating securely over an insecure communication channel** of the type which communicates a message from a transmitter to a receiver, the improvement characterized by: providing random numbers at the receiver; generating from said random numbers a public enciphering key at the receiver; generating from said random numbers a secret deciphering key at the receiver such that the secret deciphering key is directly related to and computationally infeasible to generate from the public enciphering key; communicating the public enciphering key from the receiver to the transmitter; processing the message and the public enciphering key at the transmitter and generating an enciphered message by an enciphering transformation, such that the enciphering transformation is easy to effect but computationally infeasible to invert without the secret deciphering key; transmitting the enciphered message from the transmitter to the receiver; and processing the enciphered message and the secret deciphering key at the receiver to transform the enciphered message with the secret deciphering key to generate the message.

2. ...

Project 2 looks more reasonable, maybe  
because your description of Project 1 is ~~hurdle~~  
terribly. Talk to me about these today.  
Ralph Merkle

### Project Proposal

Establishing secure communications between separate  
secure sites over insecure communication lines.

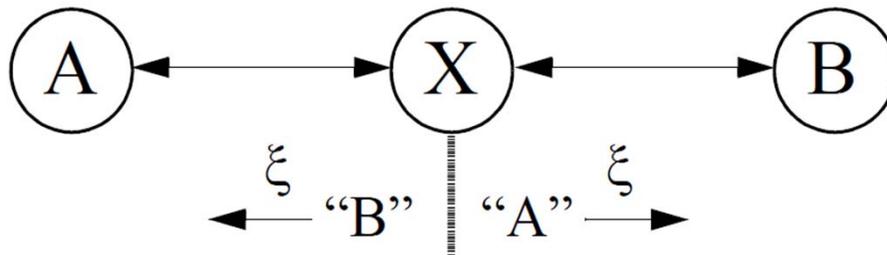
No prior arrangements have been made between the two  
sites, and it is assumed that any information known  
at either site is known to the enemy. The sites,

had the opportunity to prearrange an encryption method, then they  
will be unable to communicate securely over an insecure channel.

While this might seem intuitively obvious, I believe it is false.

I believe that it is possible for two people to communicate securely  
without having made any prior arrangements that are not completely  
public.

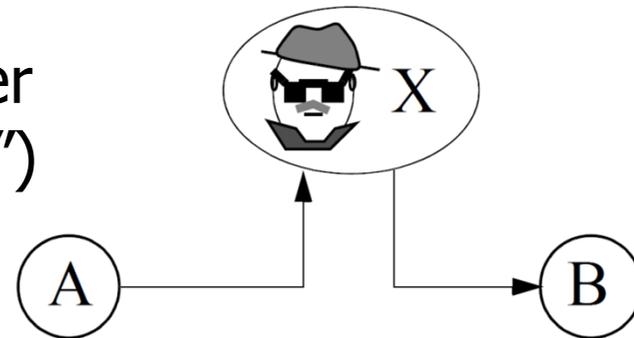
# Ist das Diffie-Hellman-Prinzip gefeit gegen einen Eindringling?



- X ist ein sogen. „**man in the middle**“
  - mimt die Identität des jeweils anderen
- X kann unter Vortäuschung falscher Identitäten jeweils **eigene Schlüssel** für **Teilstrecke AX** bzw. **XB** vereinbaren!
  - „ein  $\xi$  für ein  $\alpha$  bzw.  $\beta$  vormachen“

# Aktive Angriffe: Man in the Middle

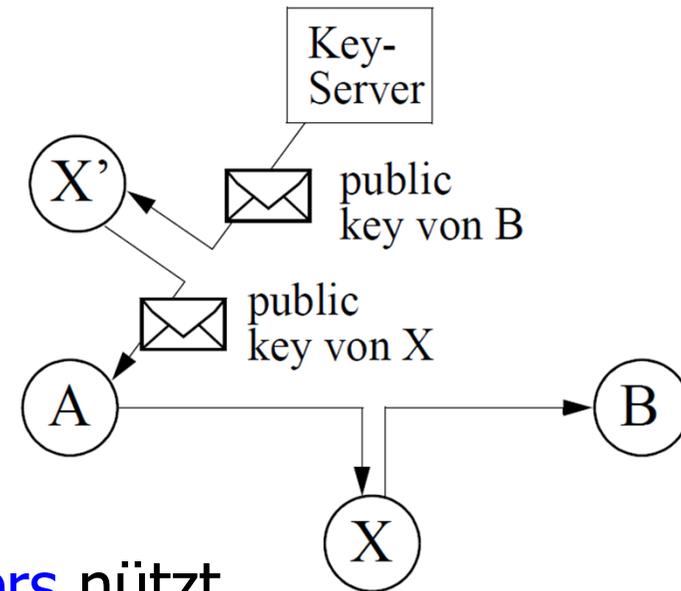
- Ein **generelles Problem**: X verhält sich gegenüber A wie B, gegenüber B wie A ( $\rightarrow$  X arbeitet "transparent")
  - z.B. eigene Schlüssel für die Teilstrecken vereinbaren



- **Challenge-Response-Test nützt so nichts**: X reicht Challenges einfach an den von ihm vorgetäuschten Partner weiter und mimt mit der abgefangenen Antwort die angenommene Identität

# Aktive Angriffe: Schlüssel-fälschung beim Key-Server

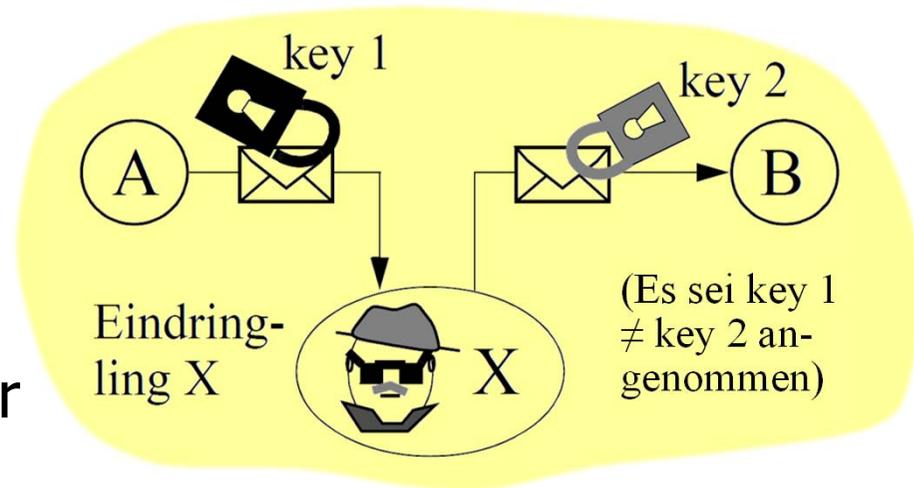
- Kompromittierter Key-Server bzw. **Verschwörung** von  $X, X'$
- $X$  kann alle von  $A$  mit dem **falschen Schlüssel verschlüsselten** Nachrichten an  $B$  entziffern
  - $X$  verschlüsselt danach die Nachricht neu mit dem richtigen Schlüssel für  $B$
- **Digitale Unterschrift des Key-Servers** nützt nichts, wenn  $A$  den Prozess  $X'$  für den Key-Server hält und dessen Unterschrift akzeptiert
- Ist es überhaupt möglich,  $X$  in diesen Szenarien zu erkennen?
- Nützt die allgemeine Bekanntgabe des public keys des Key-Servers?



# Erkennen von Eindringlingen?

Hier: Das sogen. „interlock protocol“

- 1) B stellt eine Anfrage, die nur A beantworten kann
- 2) A generiert die Antwort und verschlüsselt diese
- 3) A sendet zunächst aber nur die "Hälfte" davon zurück
  - z.B. nur jedes zweite Bit (also die "geraden" Bits)
  - B erwartet diese Hälfte der Antwort in weniger als  $t$  Zeiteinheiten
- 4) Ohne die andere Hälfte kann X dies nicht entschlüsseln und neu (mit key 2) verschlüsseln
- 5) Erst nach  $t$  Zeiteinheiten sendet A die andere Hälfte
  - B setzt Schlüsseltexthälften zusammen und überprüft Antwort



# Das Dilemma des Eindringlings

- Gibt X die halbe Nachricht sofort unverändert weiter, kann B das Ganze nicht entschlüsseln → Fälschung erkannt
- Behält X die halbe Nachricht bis zum Eintreffen der anderen Hälfte, um alles zu entschlüsseln und neu mit key 2 zu verschlüsseln (und verzögert dann die Hälfte der ungeraden Bits um t Zeiteinheiten), dann arbeitet X nicht mehr zeittransparent → Eindringling vermutet
- **Fragen:** Wird in (in Schritt 1) nicht schon ein gemeinsames Geheimnis vorausgesetzt?
- Können (im Kontext des Diffie-Hellman-Verfahrens) A und B nicht dieses benutzen, um einen von X nicht ermittelbaren gemeinsamen Schlüssel zu finden? Oder genügt in 1) eine schwächere Eigenschaft ("originelle" Antwort; Fähigkeit, die nur A hat...)?

# Zertifikate im gläsernen Tresor?

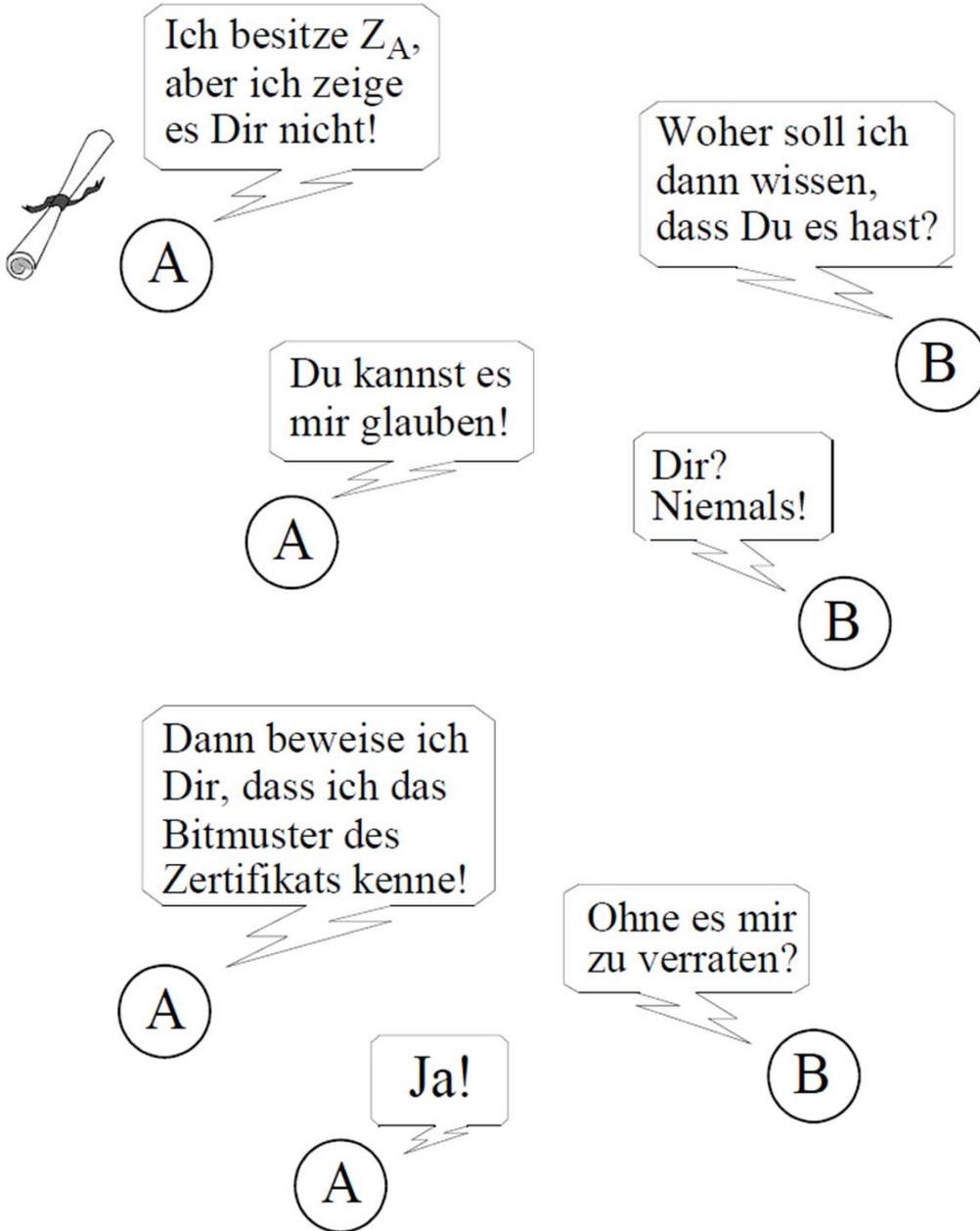


- **Anschauen:** ja
- **Stehlen oder kopieren:** nein

# Authentifizierung mit geheimen Zertifikaten?

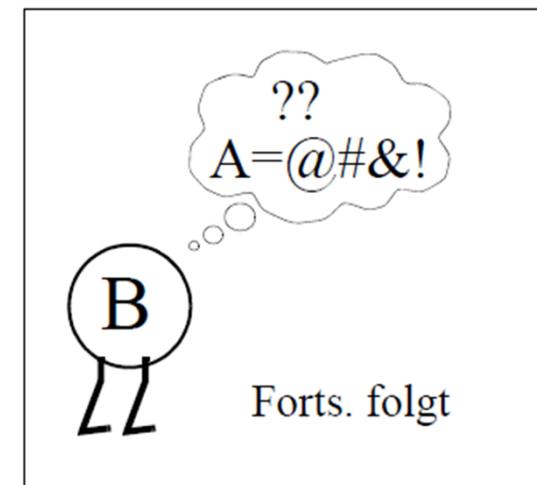


- A lässt sich von einer **Autorität** ein **Zertifikat  $Z_A$**  geben
  - $Z_A$  soll von der Autorität signiert sein
  - Autorität gilt als **vertrauenswürdig** und hat A evtl. persönlich in Augenschein genommen (oder einem fremden Zertifikat vertraut)
- Wenn B an der Identität von A zweifelt, wird B von A auf sein Zertifikat  $Z_A$  hingewiesen
  - **Besitz des Zertifikates = Authentifizierung**
- Aber: A darf  $Z_A$  **nie vorzeigen** – sonst könnte B es sich kopieren und sich fortan als A ausgeben!
  - „Dokumente“ (Bitfolgen) der digitalen Welt lassen sich perfekt kopieren
  - wie vermeidet man daher **„raubkopierte Zertifikate“**?
- Offenbar muss  $Z_A$  ein **Geheimnis** bleiben, das niemand ausser der Autorität und A kennt!
- Taugt ein solches **Geheimnis als Zertifikat??**
  - wie beweist man den Besitz eines Zertifikates, ohne es zu zeigen?



# Geheime Zertifikate?

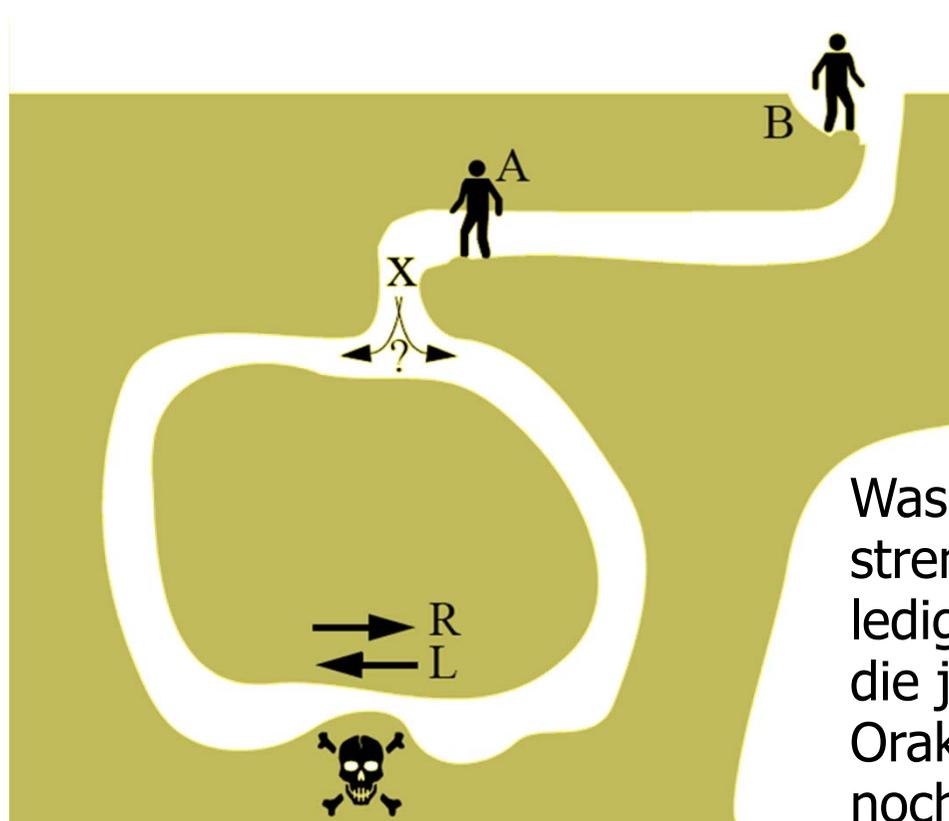
Nachweisen ohne  
vorzuweisen?



# Geheime Zertifikate!

- Im Prinzip wissen wir schon, dass das geht:  
Der **secret key**  $s_A$  eines asymmetrischen Verfahrens stellt ein solches Zertifikat dar
  - braucht von A **nicht verraten** zu werden
  - B kann **dennoch überprüfen**, ob A das Zertifikat hat (z.B. indem sich B von A etwas mit  $s_A$  verschlüsseln lässt und anschliessend durch Anwenden von  $p_A$  prüft; bzw. indem B eine Challenge  $\{M\}_{p_A}$  an A schickt und sich dies von A mit  $s_A$  entschlüsseln lässt)
- Eine andere Realisierung geht mit **“zero knowledge”**
  - beweist **Kenntnis eines Geheimnisses G**, ohne relevante Information preiszugeben

# Ein Höhlengleichnis



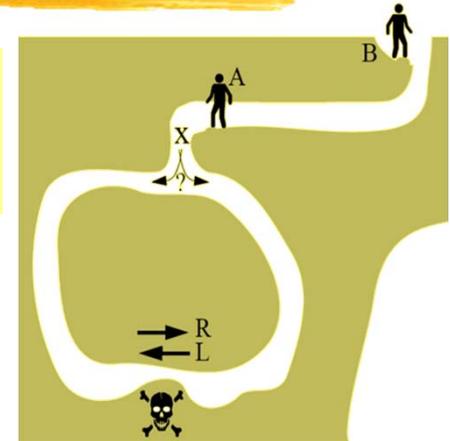
Der **Höhlendämon** lässt nur diejenigen die Engstelle lebendig passieren, die dort in der **richtigen Richtung** (L oder R) vorbeigehen.

Was die **richtige Richtung** ist, ist ein streng gehütetes **Geheimnis**; es ist lediglich bekannt, dass der Dämon die jeweilige Entscheidung einer Orakelbox entnimmt, die bisher noch niemand öffnen konnte.

- A sagt zu B: "Ich **kenne das Geheimnis**. Das beweise ich Dir, ohne das Geheimnis zu verraten!"

# Ein Höhlengleichnis

- A sagt zu B: "Ich **kenne das Geheimnis**. Das beweise ich Dir, ohne das Geheimnis zu verraten!"
- A begibt sich in die Höhle bis zur Engstelle; erst danach folgt B bis zur Stelle x
  - B weiss nicht, welche Richtung A dort eingeschlug
- B ruft A *entweder*
  - - "komm links heraus!" *oder*
  - - "komm rechts heraus!" zu
- A tut dies, indem A ggf. die Engstelle (in der richtigen Richtung) passiert
- A und B verlassen zusammen die Höhle
- Nachdem A das ganze **n Mal überlebt** hat, **ist B überzeugt**, dass A das Geheimnis (= Funktion der Orakelbox) kennt!

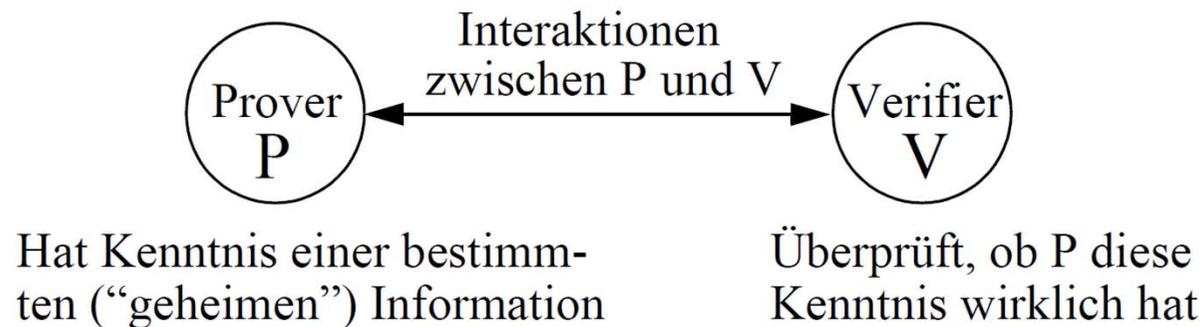


Die Irrtumswahrscheinlichkeit beträgt nur  $2^{-n}$

B hat in diesem "interaktiven Beweis" das Geheimnis nicht erfahren!  
→ "Zero knowledge proof"

# Zero-Knowledge-Beweis

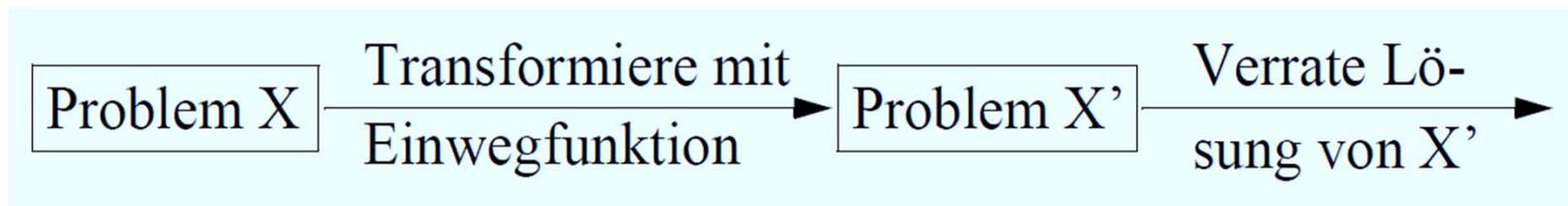
- “**Beweis**” = Nachweis, dass P eine bestimmte Folge von Bits (= Zahl, Algorithmus, Zertifikat,...) kennt



- P soll V (praktisch) **nicht betrügen** können: Wenn P die Information nicht hat, sollen seine Chancen, V zu überzeugen, verschwindend gering sein
  - V soll über die eigentliche Kenntnis von P **nichts erfahren**
  - V erfährt auch sonst nichts Relevantes von P, was V nicht auch alleine in Erfahrung bringen könnte

# Zero-Knowledge-Beweis

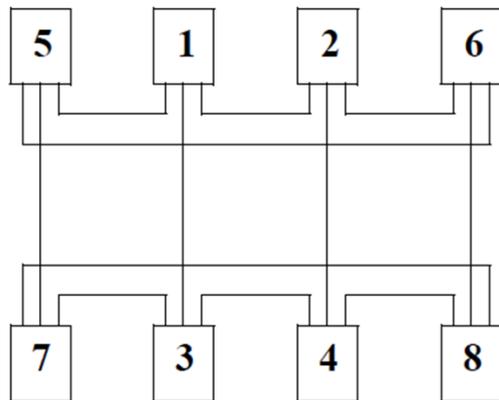
- **Idee:** geheime Information = Lösung eines schwierigen Problems  $X$



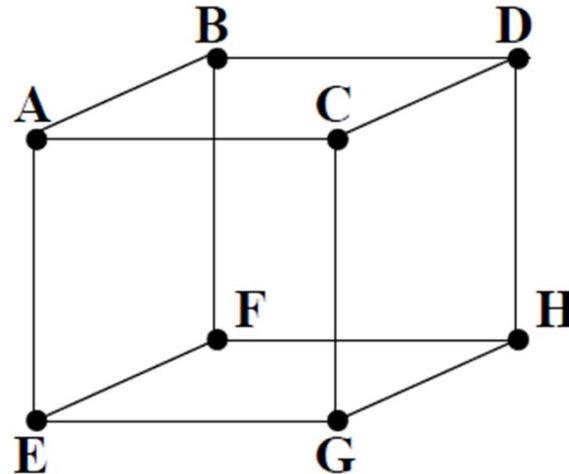
- Wobei die Lösung von  $X'$  die Lösung von  $X$  **logisch impliziert**, sie jedoch **nicht effektiv-konstruktiv** liefert

# Beispiel: Isomorphie von Graphen

Bemerkung: Ob zwei grosse (z.B. in Form von Adjazenzmatrizen) gegebene Graphen  $G_1, G_2$  **topologisch isomorph** ( $G_1 \sim G_2$ ) sind (d.h. bis auf Umbenennung von Knoten und evtl. Kanten identisch sind), ist ein **schwieriges Problem**



Hier nur ein kleines und daher einfaches (also unrealistisches) Beispiel

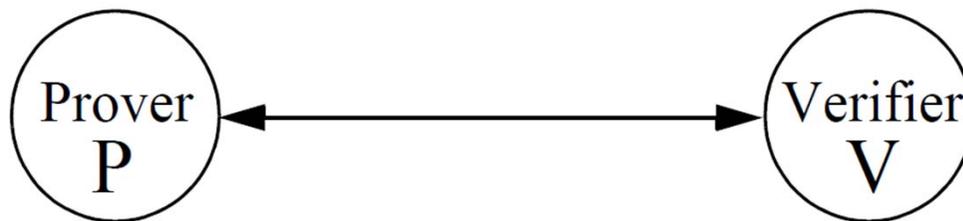


A = 7  
B = 5  
C = 8  
D = 6  
E = 3  
F = 1  
G = 4  
H = 2

**Überprüfung** eines (durch eine Knotenzuordnung gegebenen) Isomorphismus ist allerdings **"einfach"**!

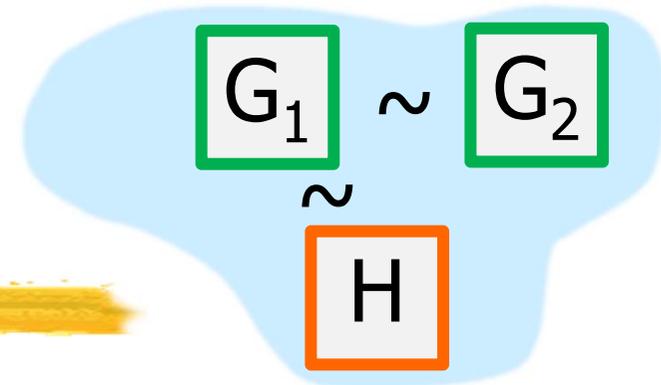
# Zero-Knowledge bei Graphisomorphie

- **P behauptet**, einen Beweis zu haben, dass zwei gegebene Graphen  $G_1$ ,  $G_2$  isomorph sind, möchte den Beweis aber nicht verraten
  - etwa weil diese **Isomorphie** seinen „**Identitätsausweis**“ darstellt

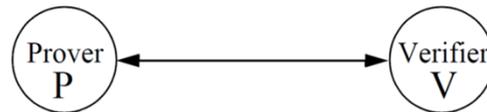


- Folgendes Protokoll **überzeugt V** davon, wobei der Isomorphismus selbst ein **Geheimnis von P bleibt** →

# Zero-Knowledge bei Graphisomorphie



- P erzeugt durch zufällige Umbenennung der Knoten einen Graphen  $H$  mit  $H \sim G_1$  (und damit  $H \sim G_2$ )
  - für P ist dies einfach



Unvorhersehbar für P

- P sendet  $H$  an V
- V bittet dann P, entweder  $H \sim G_1$  nachzuweisen, oder  $H \sim G_2$

- Da P den Graphen  $H$  konstruiert hat, kann **P** das Gewünschte **einfach** tun

- Für **Andere** aber ist  $H \sim G_1$  und  $H \sim G_2$  genauso **schwierig** wie  $G_1 \sim G_2$
- P hütet sich allerdings davor, auch noch die andere, von V nicht gewünschte, Alternative nachzuweisen – wieso?

P und V **wiederholen** alles **n Mal**, wobei von P jedes Mal ein anderer "Zeuge"  $H$  konstruiert wird  
 → Beweissicherheit =  $1-2^{-n}$

- V kann den gelieferten Isomorphiebeweis **einfach verifizieren**

# Zero-Knowledge bei Graphisomorphie – Eigenschaften

- Falls P lügt, also keinen Isomorphismus zwischen  $G_1$  und  $G_2$  kennt, kann P keinen Graphen  $H$  konstruieren, der nachweislich isomorph zu *beiden* ist
  - verschiedene  $H_1, H_2$  zu finden mit  $H_1 \sim G_1$  und  $H_2 \sim G_2$  ist einfach; mit 50% Wahrscheinlichkeit wird P so allerdings der Lüge überführt!
- V lernt nichts über die Isomorphie  $G_1 \sim G_2$ 
  - glaubt aber schliesslich, dass P eine solche kennt
- Zur Minimierung der Interaktionen lassen sich die “Runden” parallelisieren: P sendet mehrere “isomorphe Zeugen” an V, und V sendet einen Bitvektor zurück, der die Einzelnachweise auswählt

# Zero-Knowledge bei Graphisomorphie – Eigenschaften

- V kann **einem Dritten W gegenüber nicht beweisen**, dass P den Isomorphismus kennt: Selbst ein exaktes Protokoll der Kommunikationsvorgänge muss W nicht überzeugen: P und V könnten sich verschworen haben!
- Da V nichts Relevantes gelernt hat, kann V sich anderen gegenüber auch nicht mit der Kenntnis schmücken
  - **sich also nicht für P ausgeben** (Kenntnis ist Identitätsausweis von P)
- Grosse Graphen sind in der Praxis etwas unhandlich; es gibt praktischere Ausprägungen des Zero-Knowledge-Verfahrens, z.B. das **Protokoll von Fiat und Shamir**; dieses beruht auf der Schwierigkeit, die  $k$ -te Wurzel in einem Restklassenring zu berechnen