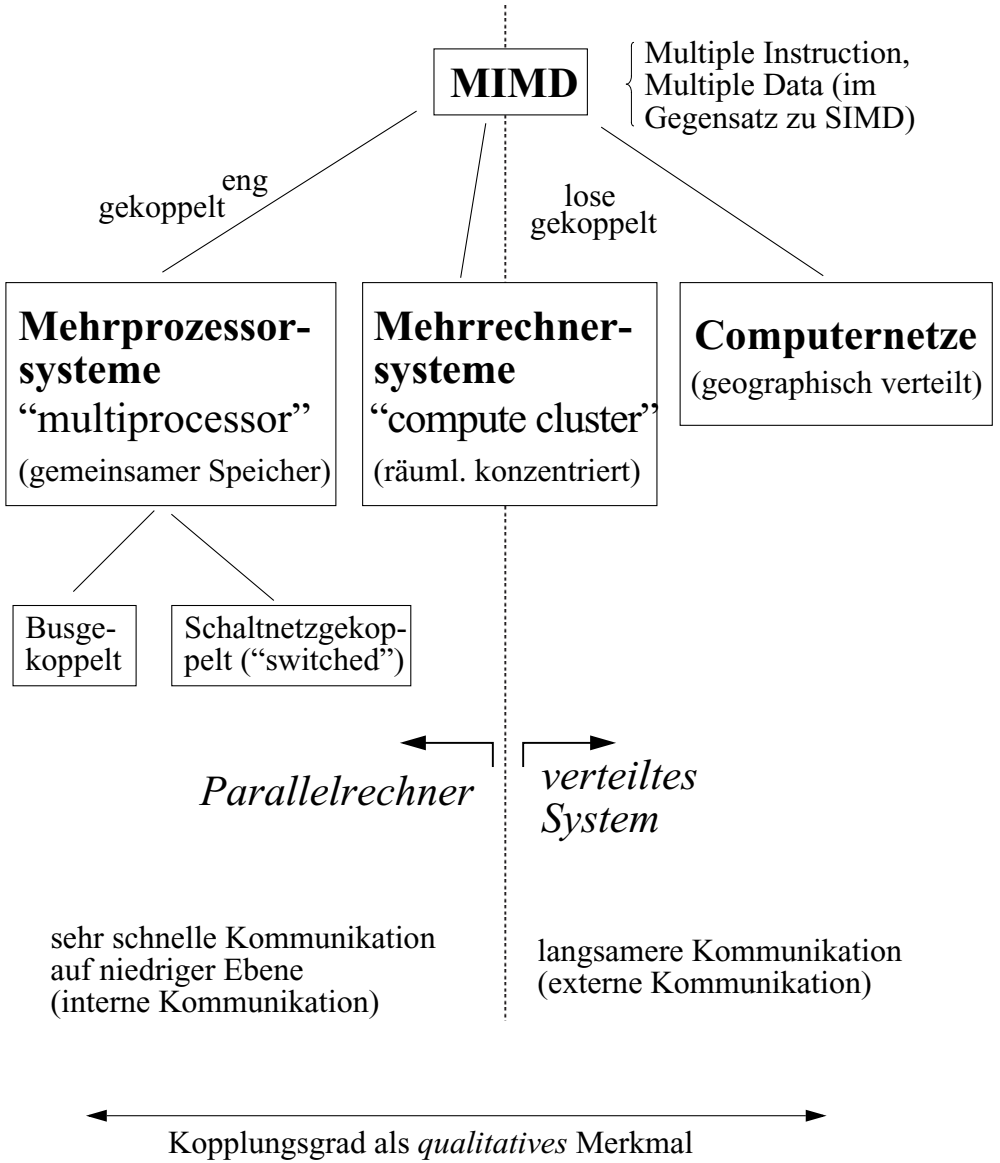
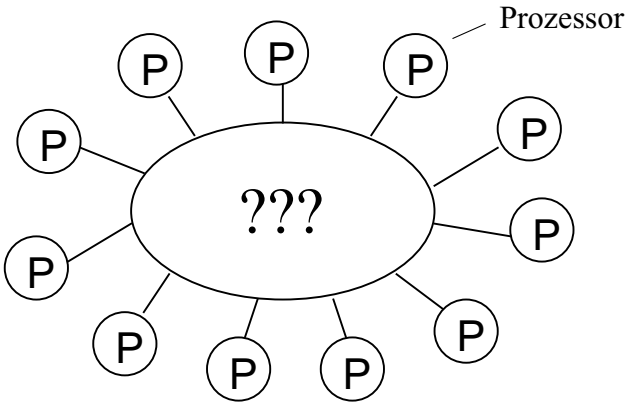
Multiprozessoren und Compute-Cluster

Abgrenzung Parallelrechner



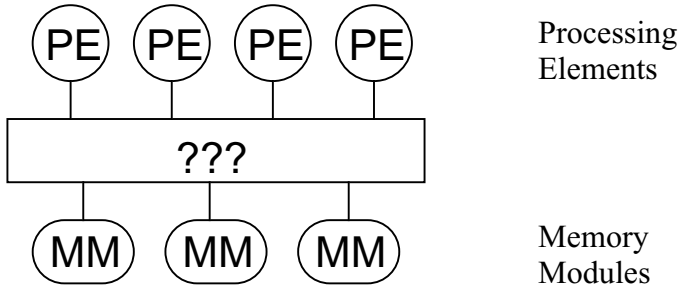
Prozessorverbund

- Autonome Prozessoren + Kommunikationsnetz
- Je nach Kopplungsgrad und Grad der Autonomie ergibt sich daraus ein
  - Mehrprozessorsystem
  - Compute Cluster
  - Computernetz



Speicherkopplung

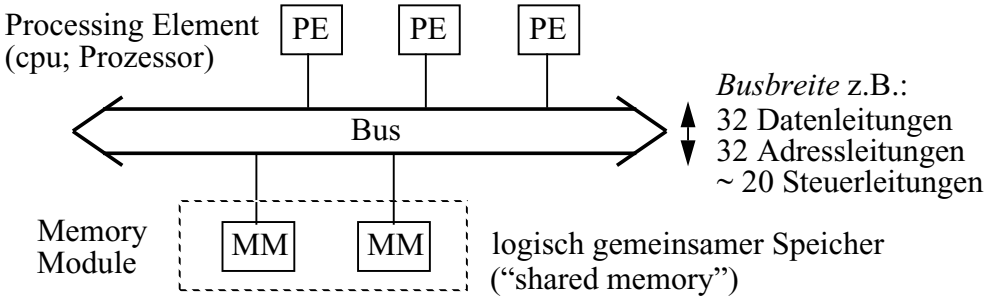
- Shared Memory
  - Kommunikation über gemeinsamen Speicher



- n Processing Elements teilen sich k Memory Modules
- Kopplung zwischen PE und MM, z.B.
  - Bus
  - Schaltnetz
  - Permutationsnetz
- UMA-Architektur (Uniform Memory Access) oder NUMA (Non-Uniform Memory Access)

wenn es “nahe” und “ferne” Speicher gibt: z.B. schneller Zugriff auf den eigenen Speicher, langsamer auf fremden

Busgekoppelte Multiprozessoren

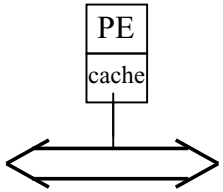


Problem:

Bus i.Allg. bereits bei wenigen (3 - 5) PEs überlastet

Lösung:

Lokale Caches
zwischen PE und Bus:



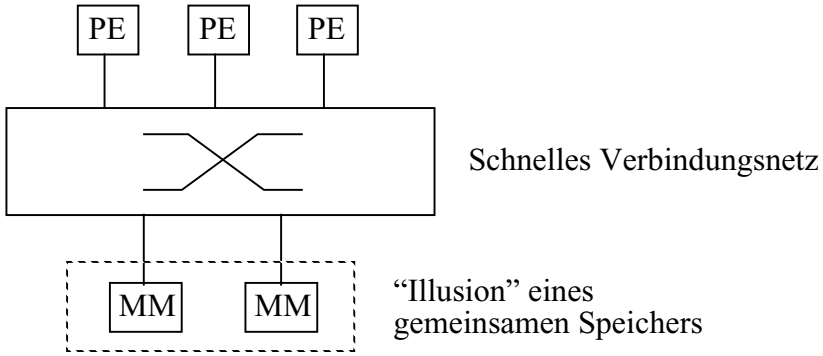
Cache gross genug (relativ zur Hauptspeichergrösse) wählen, um Hitraten > 90% erzielen

Probleme:

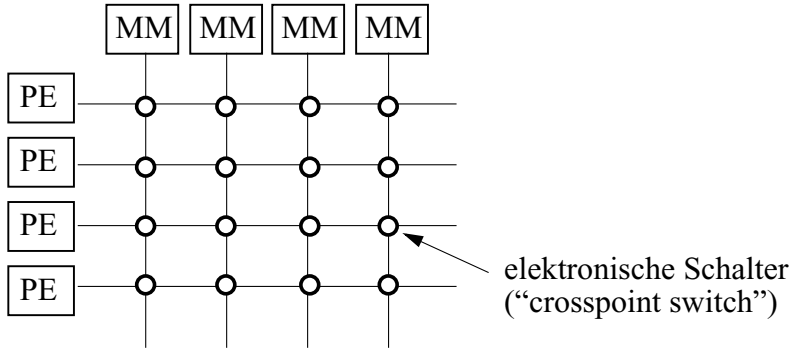
1) Kohärenz der caches
2) Damit Überlastungsproblem nur verschoben
(nicht wesentlich mehr Prozessoren möglich)

Generell: Busgekoppelte Systeme schlecht skalierbar!
(Übertragungsbandbreite bleibt “konstant” bei Erweiterung um Knoten)

Schaltnetzgekoppelte Multiprozessoren



Z.B. Crossbar-switch (Kreuzschienenverteiler):



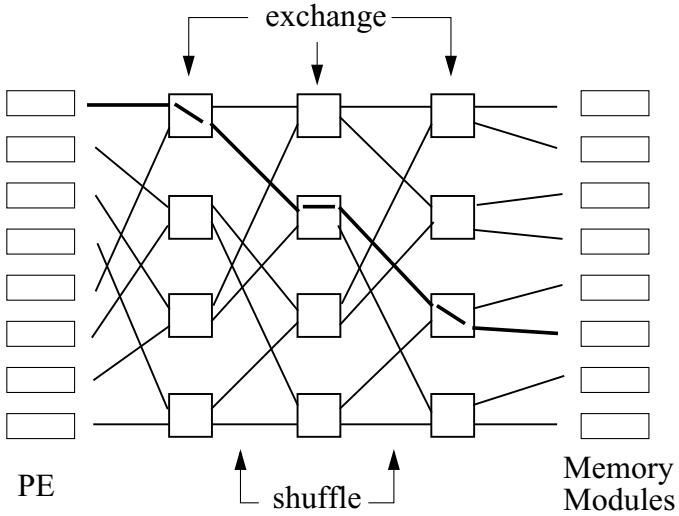
- Gleichzeitiger Zugriff von PEs auf Speichermodule (MM) zum Teil möglich
- Schlecht skalierbar (quadratisch viele Schalter)

Permutationsnetze

Mehrere Stufen von Schaltelementen ermöglichen die Verbindung jeden Einganges zu jedem Ausgang

Schaltelement (“interchange box”) kann zwei Zustände annehmen (durch ein Bit ansteuerbar):

straight through exchange



Beispiel:
*Shuffle-Exchange-*Netz
(Omega-Netz)

Hier: log n (identische!) Stufen mit je n/2 Schaltern

Es gibt weitere ähnliche dynamisch schaltbare Netze

Designkriterien:

z.B. Butterfly-Netze

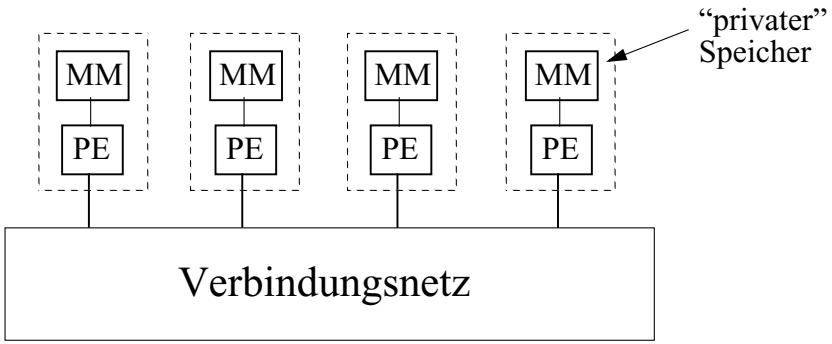
- wenig Stufen (“delay”)
- Parallele Zugriffe; Vermeidung von Blockaden

Multiprozessoren - Fazit

- Gemeinsamer Speicher, über den die Prozessoren Information austauschen (d.h. kommunizieren) können
  - Prozessoren müssen mit dem Speicher (bzw. den einzelnen Speichermodulen) gekoppelt werden
- Speicherkopplung begrenzt Skalierbarkeit und räumliche Ausdehnung
  - Untergliederung des Speichers in mehrere Module (Parallelität)
  - leistungsfähiges Kommunikationsnetz
- Bewertungskriterien für Verbindungsnetze
  - Realisierungsaufwand (Grösse, Kosten)
  - Skalierbarkeit (mit wachsender Anzahl PEs und MMs)
  - innere Blockadefreiheit (parallele Kommunikationsvorgänge?)
  - Anzahl der Stufen (Verzögerung)
  - Eingangsgrad und Ausgangsgrad der Bauelemente

Mehrrechnersysteme ("Compute Cluster")

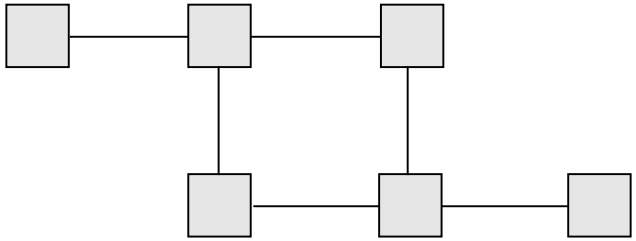
Vernetzung vollständiger Einzelrechner:



Zugriff auf andere Rechner (bzw. deren private Speicher) nur indirekt über *Nachrichten*

- kein globaler Speicher
- NORMA-Architektur (NO Remote Memory Access)

Verbindungstopologien für Mehrrechnersysteme



Zusammenhängender Graph mit

Knoten = Rechner (Prozessor mit privatem Speicher)
Kante = dedizierte Kommunikationsleitung

Ausdehnung: i.Allg. nur wenige Meter

Bewertungskriterien:

- Gesamtzahl der Verbindungen (bei n Knoten)
- maximale bzw. durchschnittliche Entfernung zweier Knoten
- Anzahl der Nachbarn eines Knotens ("fan out")
- Skalierbarkeit
- Routingkomplexität
- Zahl der alternativ bzw. parallel verfügbaren Wege

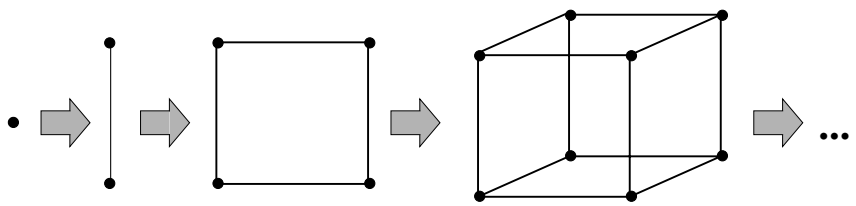
Technologische Faktoren:

- Geschwindigkeit, Durchsatz, Verzögerung, spezifische Kommunikationsprozessoren / Switches,...

Frage: Wieso kommuniziert man nicht einfach über Funk (indem z.B. jeder Knoten seine spezifische Empfangsfrequenz hat)?

Hypercube

- Hypercube = “Würfel der Dimension d”



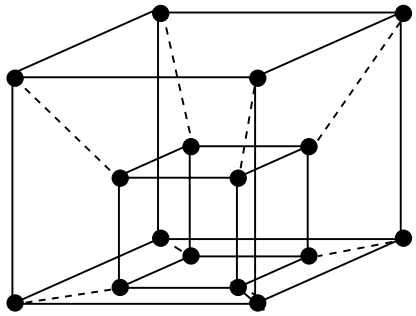
Draufsicht von der Seite liefert jeweils niedrigere Dimension

Entsprechend: Herausdrehen des Objektes aus der Blickebene zeigt, dass es sich “eigentlich” um ein Objekt der Dimension n+1 handelt!

- Rekursives Konstruktionsprinzip

  - Hypercube der Dimension 0: Einzelrechner
  - Hypercube der Dimension d+1:

    „Nimm zwei Würfel der Dimension d und verbinde korrespondierende Ecken“



4-dimensionaler Würfel

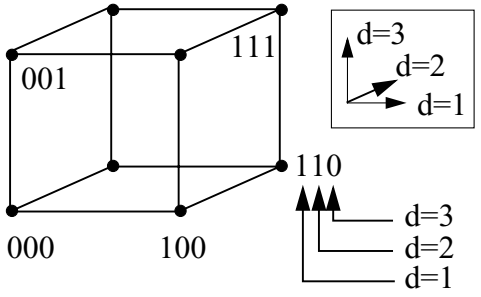
Hypercube der Dimension d

- $n = 2^d$ Knoten

- Anzahl der Nachbarn eines Knotens = d
  (Anzahl der “ports” in der Hardware)

- Gesamtzahl der Kanten (= Verbindungen): $d\ 2^d/2 = d\ 2^{d-1}$
  (Ordnung O(n log n))

- Einfaches Routing:
  - Knoten systematisch (entspr. rekursivem Aufbau) numerieren
  - 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 Übung!)

-Vorteile Hypercube:

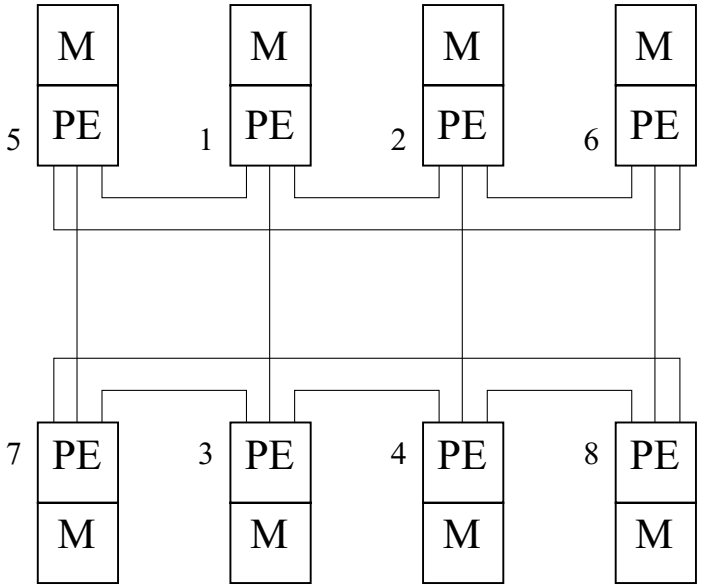
  - kurze Weglängen (max. log n)
  - einfaches Routing
  - viele Wegalternativen (Fehlertoleranz, Parallelität!)

wieviele verschiedene Wege der Länge k gibt es insgesamt?

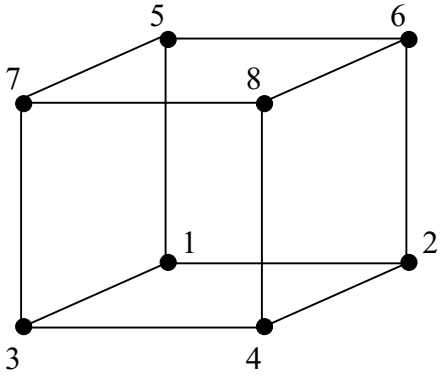
-Nachteile:

  - Anzahl der Nachbarn eines Knotens wächst mit der Dimension d
  - insgesamt relativ viele Verbindungen (“Kanten”): O(n log n) (eigentlich genügen n-1)

Layout eines Hypercube

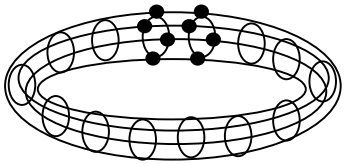


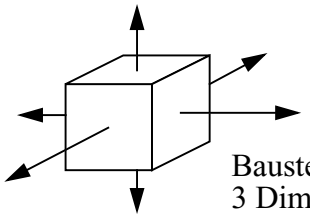
Obiger Topologie sieht man nicht unmittelbar an, dass es sich dabei um einen 3-dimensionalen Würfel handelt!



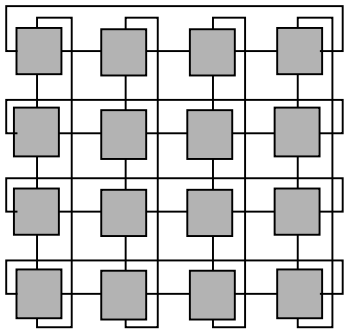
Eine andere Verbindungstopologie: der d-dimensionale Torus

= d-dimensionales Gitter mit “wrap-around”





Baustein für 3 Dimensionen



2 Dimensionen

- Rekursives Konstruktionsprinzip: „Nimm w_d gleiche Exemplare eines Torus der Dimension d -1 und verbinde korrespondierende Elemente zu einem Ring“

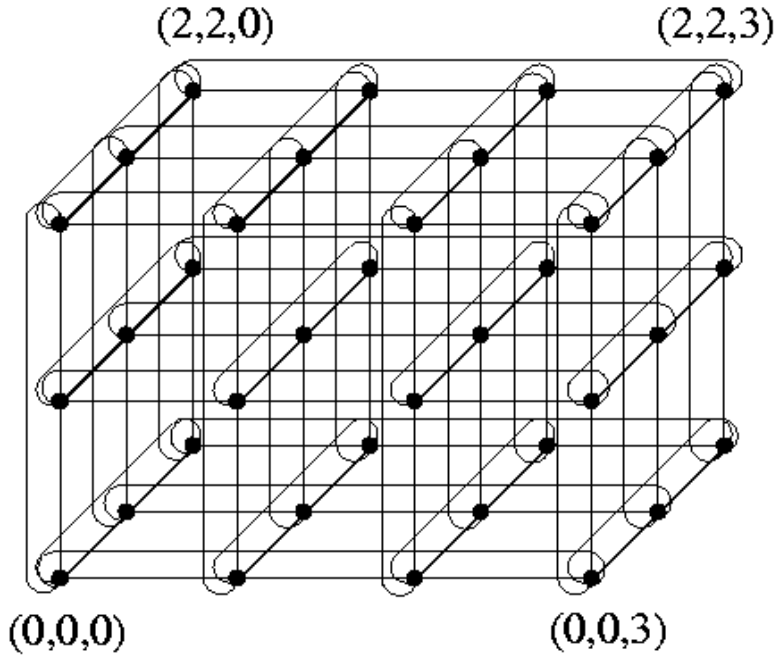
- Bei Ausdehnung w_i in Dimension i:
  $n = w_1 \times w_2 \times .. \times w_d$ Knoten;
  mittlere Entfernung zw. 2 Knoten: $\Delta \approx \frac{1}{4} \sum w_i$

- *Ring* als Sonderfall d = 1 !

- *Hypercube* der Dimension d ist d-dimensionaler Torus mit $w_i = 2$ für alle Dimensionen!

  $\rightarrow \quad \Delta = \frac{1}{4} \sum_d 2 = \frac{1}{4}(2\ d) = \frac{d}{2} = \underline{\underline{\frac{1}{2} \log_2 n}}$

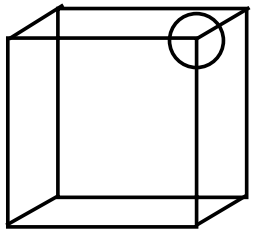
3-dimensionaler Torus

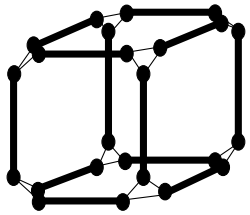


Mit $w_1 = 4$, $w_2 = 3$, $w_3 = 3$

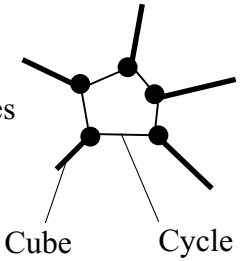
Cube Connected Cycles (CCC)

CCC = d-dimensionaler Hypercube mit aufgeschnittenen Ecken, die durch Gruppen von d ringförmig verbundenen Knoten ersetzt sind





"Ecke" eines CCC (d=5)



Beachte: Jeder Knoten hat immer *drei* Anschlüsse!

Bei Dimension d: $n = d\ 2^d$

Maximale / mittlerer Weglänge? ← Denkübung! (nicht ganz einfach)

Anzahl der Verbindungen = 3 n / 2
(statt O(n log n) wie beim Hypercube)

Es gibt viele weitere Verbindungstopologien
(wollen wir hier aber nicht betrachten)

Kommunikation

Kommunikation

- Prozesse sollen kooperieren, daher untereinander Information austauschen können; mittels
  - *gemeinsamer Daten* in einem globalen Speicher (dieser kann physisch oder evtl. nur logisch vorhanden sein: “virtual shared memory”)
  - oder *Nachrichten*: Daten an eine entfernte Stelle kopieren

Notwendig, damit die Kommunikation klappt, ist jedenfalls:

1) dazwischenliegendes *physikalisches Medium*
   - z.B. elektrische Signale in Kupferkabeln
2) einheitliche *Verhaltensregeln*
   - Kommunikationsprotokolle
3) gemeinsame *Sprache* und gemeinsame *Semantik*
   - gleiches Verständnis der Bedeutung von Kommunikationskonstrukten und -Regeln

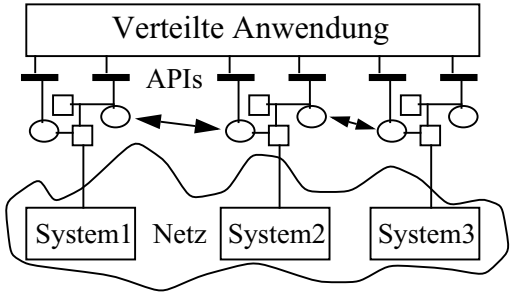
Also trotz Verteiltheit gewisse gemeinsame Aspekte!

- 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

als einfache Form von “Middleware”

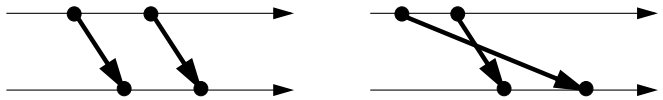
- Organisiert den Nachrichtentransport
- Bietet Kommunikationsprimitive (als APIs) an
  - z.B. *send* (...) bzw. *receive* (...)
  - evtl. auch ganze Bibliothek unterschiedlicher Kommunikationsdienste
  - verwendbar mit gängigen Programmiersprachen (oft zumindest C)



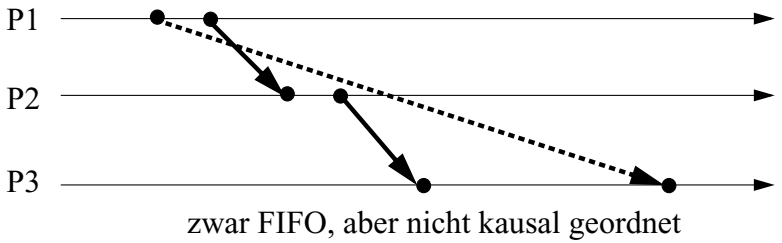
- Besteht aus Hilfsprozessen, Pufferobjekten,...
- Verbirgt Details des zugrundeliegenden Netzes
- Verwendet vorhandene Netzprotokolle und implementiert damit neue, “höhere” Protokolle
- Garantiert (je nach “Semantik”) gewisse Eigenschaften
  - z.B. Reihenfolgeerhalt oder Prioritäten von Nachrichten
- Abstrahiert von Implementierungsaspekten
  - wie z.B. Geräteadressen, Längenbeschränkung von Nachrichten etc.
- Maskiert gewisse Fehler
  - mit typischen Techniken zur Erhöhung des Zuverlässigkeitsgrades: Timeouts, Quittungen, Sequenznummern, Wiederholungen, Prüfsummen, fehlerkorrigierende Codes,...
- Verbirgt Heterogenität unterschiedlicher Rechner- bzw. Betriebssystemplattformen
  - erleichtert Portabilität von Anwendungen

Ordnungserhalt von Nachrichten?

- Manchmal werden vom Kommunikationssystem Garantien bzgl. Nachrichtenreihenfolgen gegeben
- Eine mögliche Garantie stellt FIFO (First-In-First-Out) dar: Nachrichten zwischen zwei Prozessen überholen sich nicht: Empfangsreihenfolge = Sendereihenfolge



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



- Möchte man auch dies haben, so muss die Kommunikation “kausal geordnet” sein (Anwendungszweck?)
  - “Dreiecksungleichung”: Keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege
  - 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?

- Achtung: *Semantik* ist a priori nicht ganz klar:
  - Soll (kann?) das Transportsystem Nachrichten höherer Priorität bevorzugt (=?) befördern?
  - Sollen (z.B. bei fehlender Pufferkapazität) Nachrichten niedrigerer Priorität überschrieben werden?
  - Wieviele Prioritätsstufen gibt es?
  - Sollen auf Empfangsseite zuerst Nachrichten mit höherer Priorität angeboten werden?

- Mögliche *Anwendungen*:
  - Unterbrechen 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.)

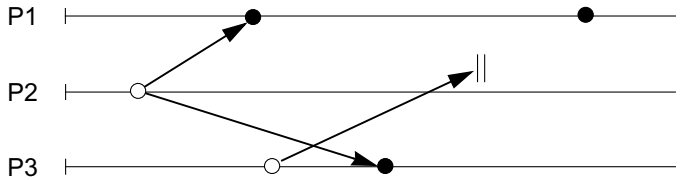
Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!

- Inwiefern ist denn eine (faule) Implementierung, bei der "eilige" Nachrichten (insgeheim) wie normale Nachrichten realisiert werden, tatsächlich nicht korrekt?

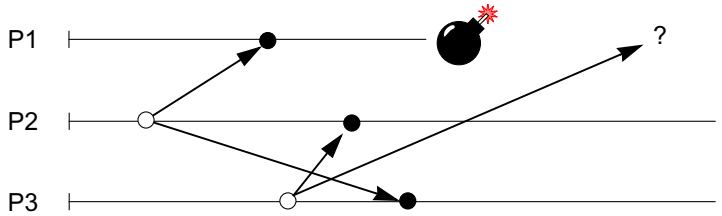
Fehlermodelle

- Klassifikation von Fehlermöglichkeiten; Abstraktion von den konkreten, spezifischen Ursachen

• **Fehler beim Senden / Übertragen / Empfangen**



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



• **Zeitfehler:** Ereignis erscheint zu spät (oder zu früh)

• **"Byzantinische" Fehler:** Beliebiges Fehlverhalten, z.B.:
  - verfälschte Nachrichteninhalte
  - Prozess, der unsinnige Nachrichten sendet

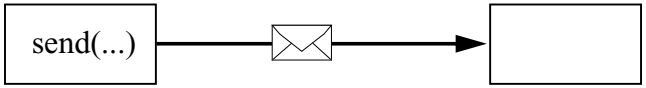
(solche Fehler lassen sich nur teilweise, z.B. durch *Redundanz,* erkennen)

Fehlertolerante Algorithmen sollen das "richtige" Fehlermodell berücksichtigen!

- adäquate Modellierung der realen Situation / des Einsatzgebietes
- Algorithmus verhält sich korrekt nur *relativ* zum Fehlermodell

Mitteilung vs. Auftrag

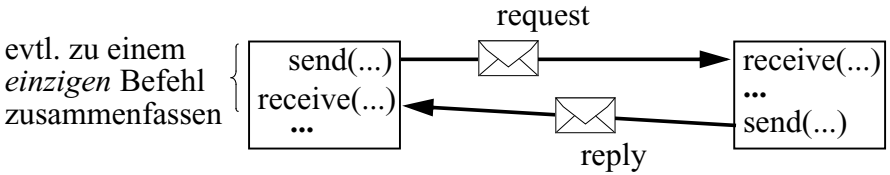
Mitteilungsortientiert:



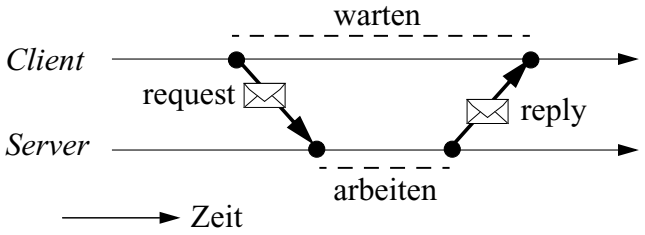
- Unidirektional
- Übermittelte Werte werden der Nachricht typw. als “Ausgabeparameter” beim send übergeben

---

Auftragsortientiert:



- Bidirektional
- Ergebnis eines Auftrags wird als “Antwortnachricht” zurückgeschickt
- Auftraggeber (“Client”) *wartet* bis Antwort eintrifft

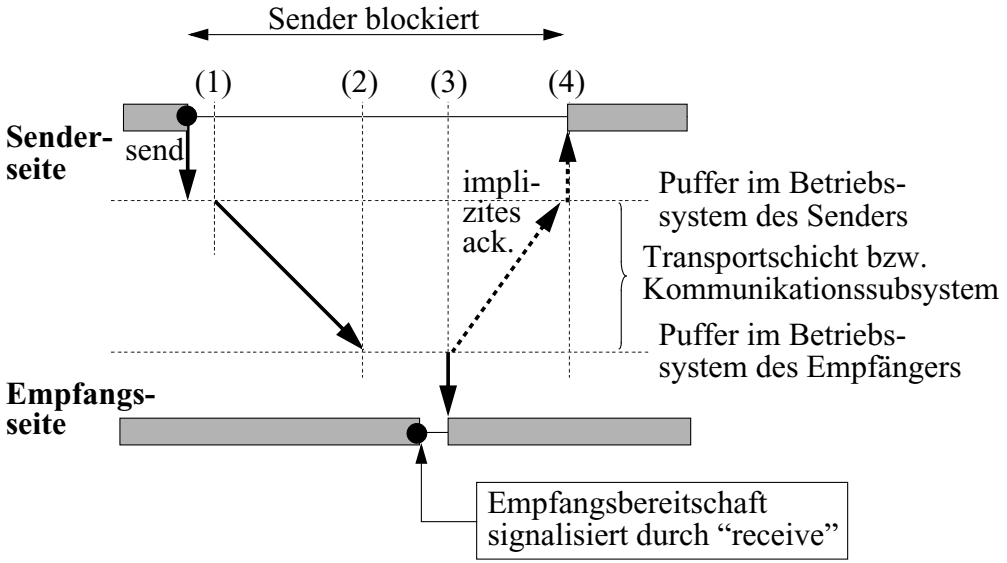


Blockierendes Senden

- *Blocking send:* Sender ist bis zum Abschluss der Nachrichtentransaktion blockiert

was genau ist das?

  - Sender hat so eine *Garantie* (Nachricht wurde zugestellt / empfangen)



- Verschiedene Ansichten der “korrekten” Definition von “Abschluss der Transaktion” aus Sendersicht:
  - *Zeitpunkt 4* (automatische Bestätigung, dass der Empfänger das 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.

Synchrone Kommunikation

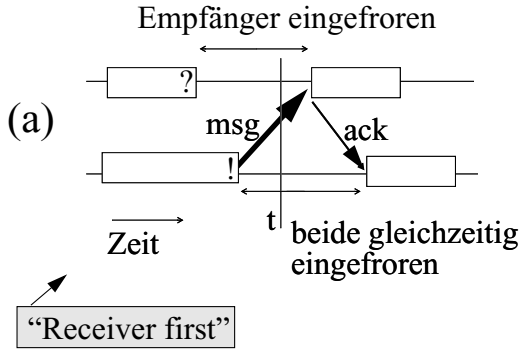
"gleich" "zeitig"

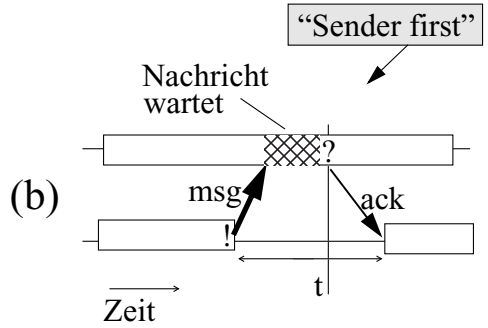
- Idealisierung: Send und receive geschehen *gleichzeitig*

- Wodurch ist diese Idealisierung gerechtfertigt?
  (Kann man auch mit einer Marssonde synchron kommunizieren?)

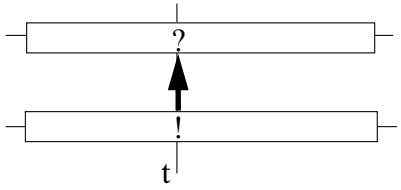
- Bem.: "Receive" ist i.Allg. blockierend (d.h. Empfänger wartet so lange, bis eine Nachricht ankommt)

Implementierung mit blocking send:





Idealisierung: senkrechte Pfeile in den Zeitdiagrammen



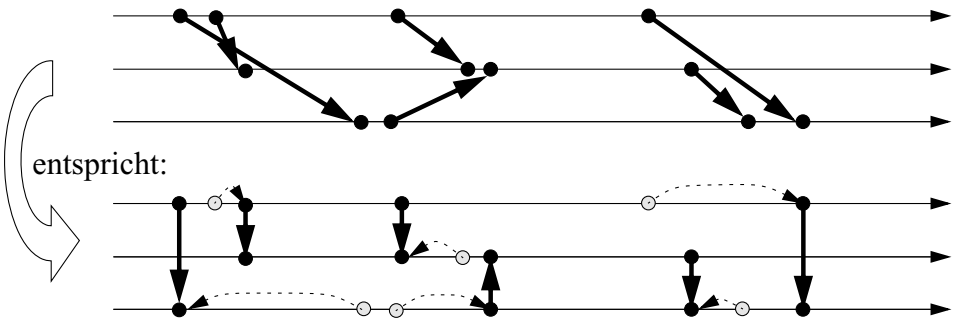
Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet ("!") *und* empfangen ("?") worden!

Zeit des Senders steht still → es gibt einen *gemeinsamen Zeitpunkt t,* wo die beiden Kommunikationspartner sich treffen.
→ *"Rendezvous"*

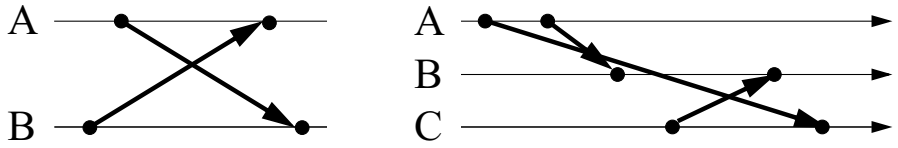
Virtuelle Gleichzeitigkeit

- 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 ("Gummiband-Transformation")



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



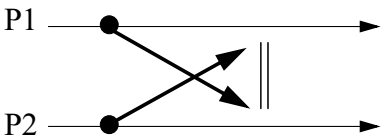
  - aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

Mehr dazu nur für besonders Interessierte: Charron-Bost, Mattern, Tel: *Synchronous, Asynchronous and Causally Ordered Communication.* Distributed Computing, Vol. 9 No. 4 (173-191), www.vs.inf.ethz.ch/publ/

Deadlocks bei synchroner Kommunikation

```
P1:
    send (...) to P2;
    receive...
    ...
```

```
P2:
    send (...) to P1;
    receive...
    ...
```

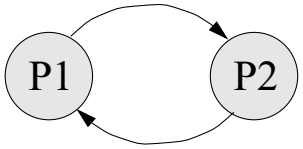


In beiden Prozessen muss zunächst das *send* ganz ausgeführt werden, bevor es zu einem *receive* kommt

⇒ *Kommunikationsdeadlock!*

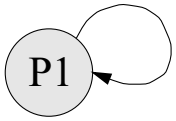
Zyklische Abhängigkeit der Prozesse voneinander: P1 wartet auf P2, und P2 wartet auf P1

Gleichnishaft entspricht der syn. Kommunikation das Telefonieren, der asy. Komm. der Briefwechsel



"Wait-for-Graph"

Genauso tödlich:

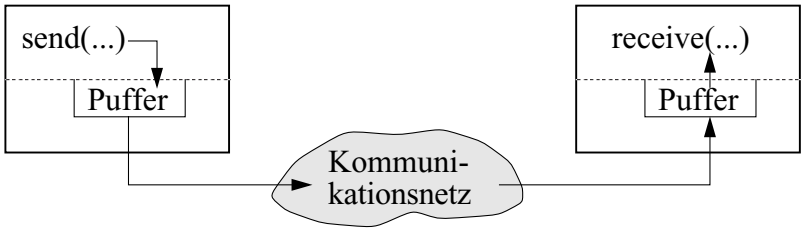



```
P1:
    send (...) to P1;
    receive...
    ...
```


Asynchrone Kommunikation

- *No-wait send:* Sender ist nur (kurz) bis zur lokalen Ablieferung der Nachricht an das Transportsystem blockiert
  (diese kurzzeitigen Blockaden sollten für die Anwendung transparent sein)

- Jedoch i.Allg. länger blockiert, falls das System z.Z. keinen Pufferplatz für die Nachricht frei hat
  (Alternative: Sendenden Prozess nicht blockieren, aber mittels "return value" über Misserfolg des send informieren)



- *Vorteile:*
  (im Vgl. zur syn. Kommunikation oft angenehmer in der Anwendung)
  - → Sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird
  - → Stärkere Entkoppelung von Sender / Empfänger
  - → Höherer Grad an Parallelität möglich
  - → Geringere Gefahr von Kommunikationsdeadlocks

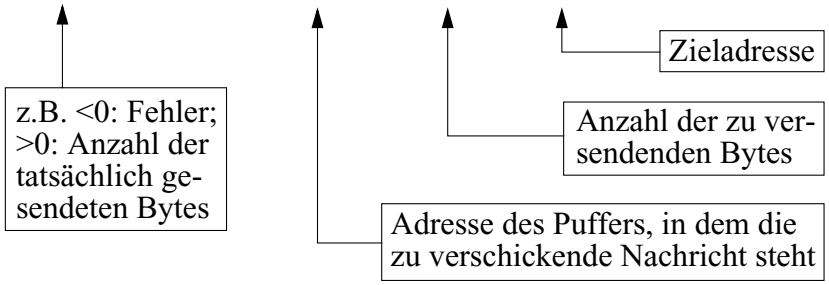
- *Nachteile:*
  (im Vgl. zur synchronen Kommunikation aufwendiger zu realisieren)
  - → Sender weiss nicht, ob / wann Nachricht angekommen
  - → Debugging der Anwendung oft schwierig (wieso?)
  - → System muss Puffer verwalten

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 vielen vernetzten Systemen / Compute-Clustern

- Typischer Aufruf einer solchen Send-Operation:

```
status = send(buffer, size, dest, ...)
```



- Derartige Systeme bieten i.Allg. 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)
  - Problem: Semantik und Kontext der Anwendbarkeit ist oft nur informell beschrieben

Synchron $\stackrel{?}{=}$ blockierend

- Kommunikationsbibliotheken machen oft einen Unterschied zwischen *synchronem* und *blockierendem* Senden
  - bzw. analog zwischen asynchron und nicht-blockierend
  - leider etwas verwirrend!

- Blockierung ist dann ein rein *senderseitiger* Aspekt
  - *blockierend*: Sender wartet, bis die Nachricht 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!)

- Synchron / asynchron nimmt Bezug auf den *Empfänger*
  - *synchron*: Nach Ende der Send-Operation wurde die Nachricht dem Empfänger zugestellt (*asynchron*: dies ist nicht garantiert)

- Nicht-blockierende Operationen liefern oft einen “handle”

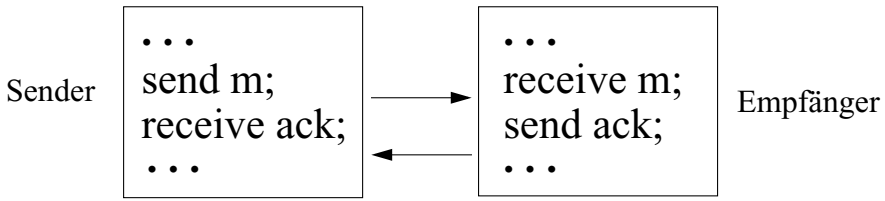
```
handle = send(...)
```

  - dieser kann in Test- bzw. Warteoperationen verwendet werden
  - z.B. Test, ob Send-Operation beendet: **`msgdone(handle)`**
  - z.B. warten auf Beendigung der Send-Operation: **`msgwait(handle)`**

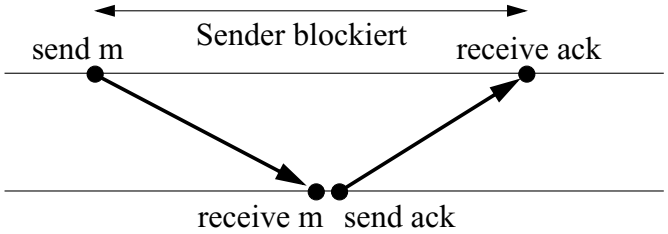
- Nicht-blockierend ist effizienter aber u.U. unsicherer und umständlicher (evtl. Test; warten) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:

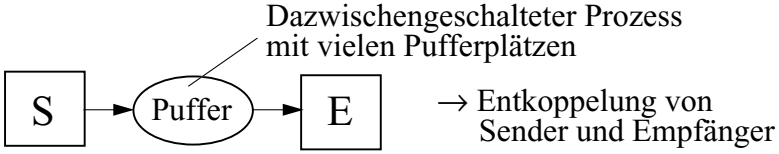


- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive

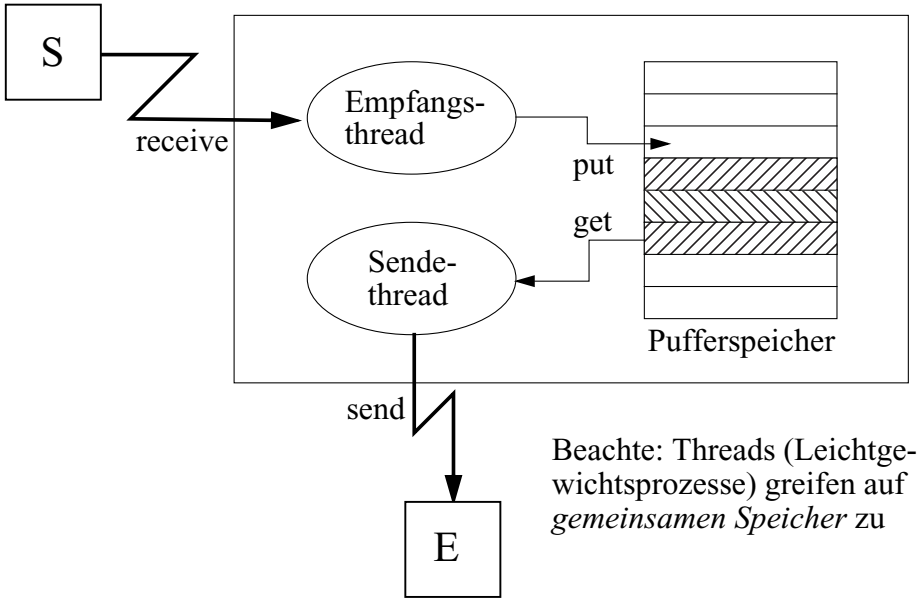


Asynchrone Kommunikation mittels synchroner:

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



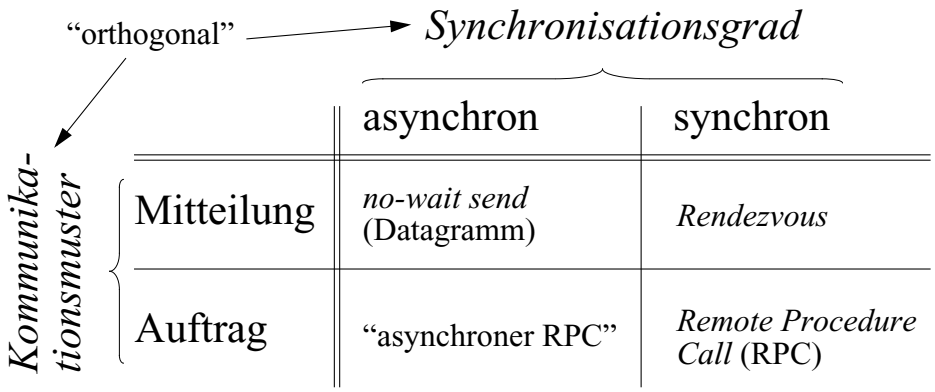
Puffer als Multithread-Objekt



Beachte: Threads (Leichtgewichtsprozesse) greifen auf *gemeinsamen Speicher* zu

- Empfangsthread ist (fast) immer empfangsbereit
  - nur kurzzeitig anderweitig beschäftigt (put in lokalen Pufferspeicher) (aber evtl. nicht empfangsbereit, wenn lokaler Pufferspeicher voll)
- Sendethread ist (fast) immer sendebereit
- Pufferspeicher (FIFO) wird i.Allg. zyklisch verwaltet
- Pufferspeicher liegt im gemeinsamen Adressraum
  ⇒ *Synchronisation* der beiden Threads notwendig!
  - z.B. Semaphore etc.
  → "konkurrentes Programmieren"
  → klassische Themen der Betriebssystem-Theorie!

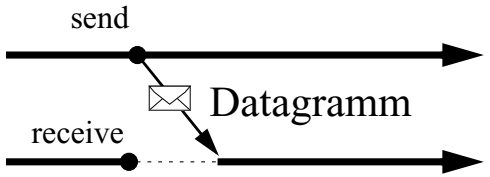
Klassifikation von Kommunikationsmechanismen



- Hiervon gibt es diverse Varianten
  - bei verteilten objektorientierten Systemen z.B. "Remote Method Invocation" (RMI) statt RPC
- Weitere Klassifikation nach Adressierungsart möglich (Prozess, Port, Mailbox, Broadcast...)
- Häufigste Kombination: Mitteilung asynchron, Auftrag hingegen synchron

Datagramm

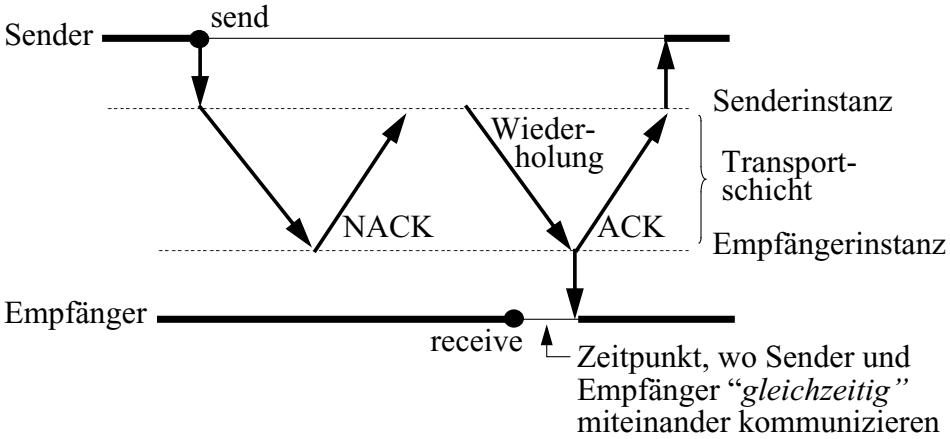
- Asynchron-mitteilungsorientierte Kommunikation



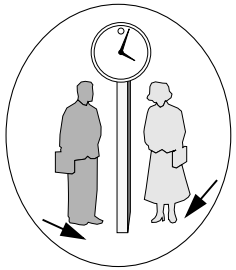
- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- Nachteil
  - keine Erfolgsgarantie für den Sender
  - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicherverwaltung …) im Unterschied etwa zur synchronen Kommunikation
  - „Überrennen" des Empfängers bei langen/ häufigen Nachrichten → Flusssteuerung notwendig

Rendezvous-Protokolle

- Synchron-mitteilungsorientierte Kommunikation



- Hier "Sender-first-Szenario":
  Sender wartet zuerst
  - "Receiver-first-Szenario" analog



- *Rendezvous*: Der erste wartet auf den anderen... ("Synchronisationspunkt")

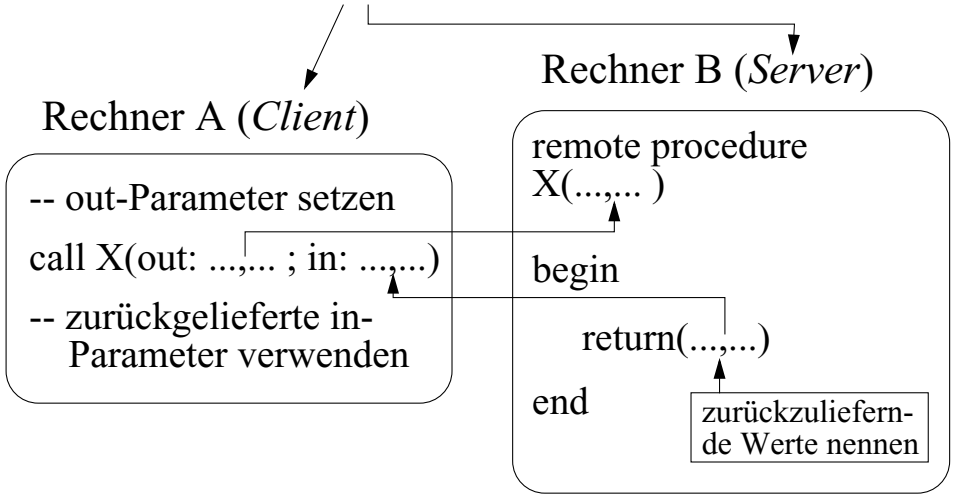
- Mit NACK / ACK sind weniger Puffer nötig
  → Aber aufwändiges Protokoll! ("busy waiting")
  - Alternative 1: Statt NACK: Nachricht auf Empfängerseite puffern
  - Alternative 2: Statt laufendem Wiederholungsversuch: Empfänger-instanz meldet sich bei Senderinstanz, sobald Empfänger bereit

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

Remote Procedure Call (RPC)

- "Entfernter Prozeduraufruf"

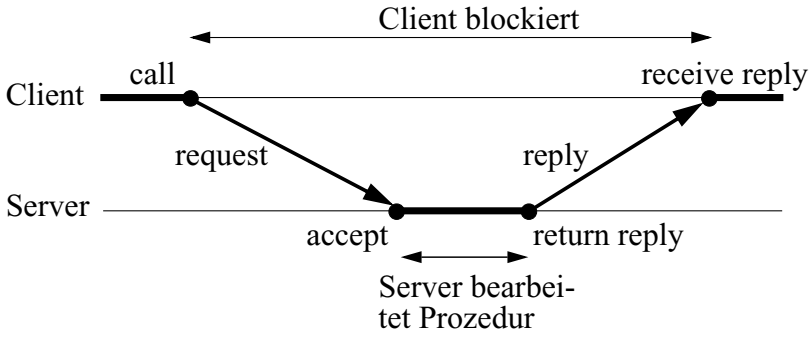
- Synchron-auftragsorientiertiertes Prinzip:



- Soll dem klassischen Prozeduraufruf möglichst gleichen
  - klare Semantik für den Anwender (Auftrag als „Unterprogramm")
  - einfaches Programmieren
    - kein Erstellen von Nachrichten, kein Quittieren,... auf Anwendungsebene
    - Syntax analog zu bekanntem lokalen Prozeduraufruf
    - Verwendung von lokalen / entfernten Prozeduren "identisch"
  - Typsicherheit (Datentypüberprüfung auf Client- und Serverseite möglich)

- Implementierungsproblem: Verteilungstransparenz
  - Verteiltheit so gut wie möglich verbergen

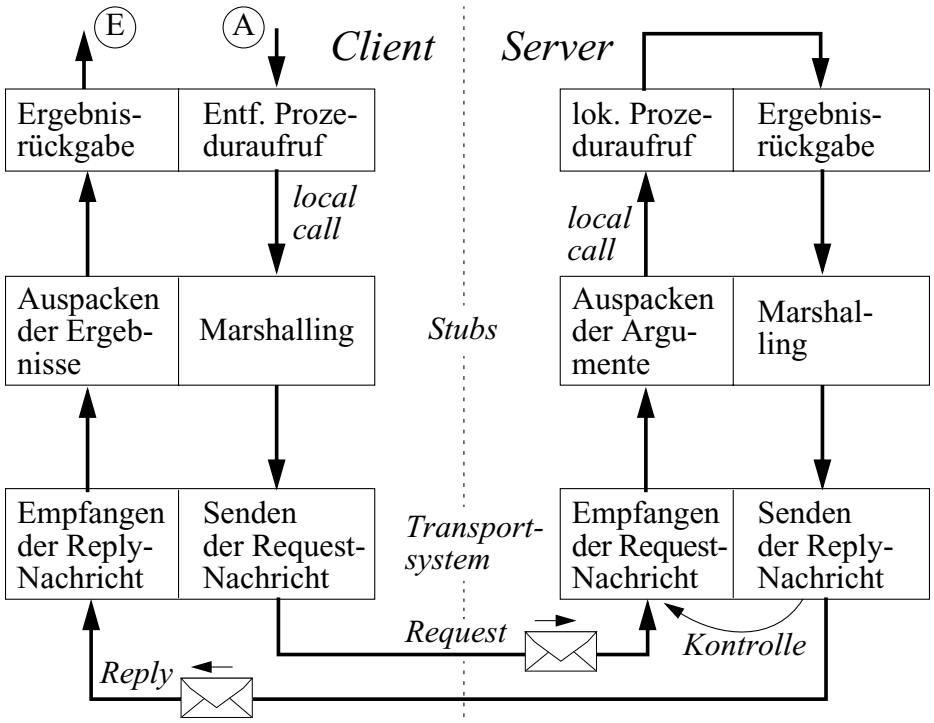
RPC: Prinzipien

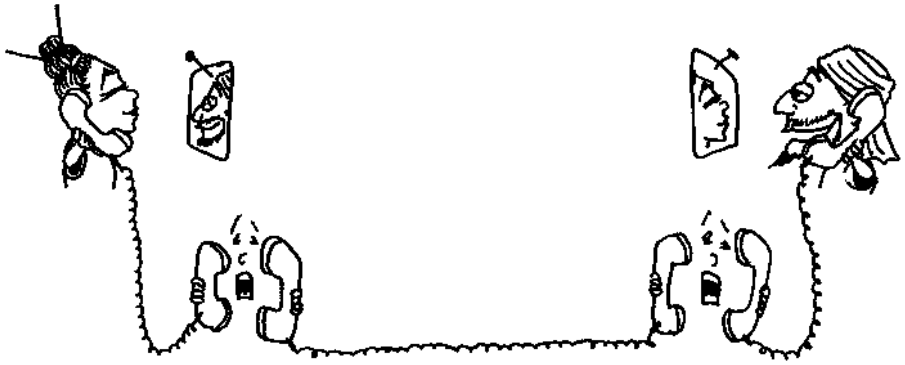


→ send → receive

- call; accept; return; receive: interne Anweisungen
  - nicht sichtbar auf Sprachebene → Compiler bzw. realisiert im API
- Parameterübergabe: call-by-value/result
- Keine Parallelität zwischen Client und Server
  - RPC-Aufrufe sind blockierend

RPC: Implementierung





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

RPC: Stubs

- *Stub* = Stummel, Stumpf

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

Client:

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

- 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

- *Lokale Stellvertreter* ("proxy") des entfernten Gegenübers
  - Client-Stub bzw. Server-Stub
  - simulieren einen lokalen Aufruf
  - sorgen für Packen und Entpacken von Nachrichten
  - konvertieren Datenrepräsentationen bei heterogenen Umgebungen
  - steuern das Übertragungsprotokoll (z.B. zur fehlerfreien Übertragung)
  - bestimmen evtl. Zuordnung zwischen Client und Server („Binding")

- Können oft weitgehend *automatisch generiert* werden
  - z.B. mit einem "RPC-Compiler" aus dem Client- oder Server-Code und evtl. einer Schnittstellenbeschreibung (z.B. formuliert in IDL = "Interface Description Language")
  - nutzen Hilfsroutinen aus Bibliotheken für Formatkonversion usw.
  - nutzen Routinen einer *RPC-Laufzeitumgebung* (z.B. zur Kommunikationssteuerung, Fehlerbehandlung etc.)

wird nicht generiert, sondern dazugebunden

- Stubs sorgen also für *Transparenz*

RPC: Marshalling

- Problem: Parameter aus *komplexen Datentypen* wie
  - Records, Strukturen
  - Objekte
  - Referenzen, Zeiger
  - Zeigergeflechte

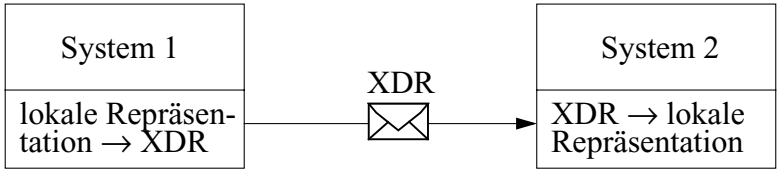
  → sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein?
  → sollen Referenzen symbolisch, relativ... interpretiert werden? Ist das stets möglich?
  → wie wird Typkompatibilität sichergestellt?

- Problem: RPCs werden oft in *heterogenen* Umgebungen eingesetzt mit unterschiedlicher Repräsentation z.B. von
  - Strings (Längenfeld ↔ '\0')
  - Character (ASCII ↔ Unicode)
  - Arrays (zeilen- ↔ spaltenweise)
  - niedrigstes Bit einer Zahl vorne oder hinten

→ *Marshalling:* Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern
  - evtl. dabei geeignete Codierung (komplexer) Datenstrukturen
  - Glätten ("flattening") komplexer (evtl. verzeigerter) Datenstrukturen zu einer Sequenz von Basistypen (evtl. mit Strukturinformation)
  - umgekehrte Tranformation auch als "unmarshalling" bezeichnet

RPC: Datenkonversion

1) Umwandlung in eine gemeinsame Standardrepräsentation
   - z.B. “XDR” (eXternal Data Representation)



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

2) Oder 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 mit der Senderrepräsentation umgehen können

---

Datenkonversion überflüssig, wenn sich alle Kommunikationspartner an einen gemeinsamen Standard halten

RPC: Transparenzproblematik

- RPCs sollten so weit wie möglich lokalen Prozeduraufrufen gleichen, es gibt aber einige subtile Unterschiede

bekanntes Programmier-paradigma!

  - Client- / Serverprozesse haben evtl. unterschiedliche Lebenszyklen: Server könnte noch nicht oder nicht mehr oder in einer “falschen” Version existieren

- Leistungstransparenz?
  - RPC i.Allg. wesentlich langsamer
  - Bandbreite ist bei umfangreichen Datenmengen relevant
  - ungewisse, variable Verzögerungen

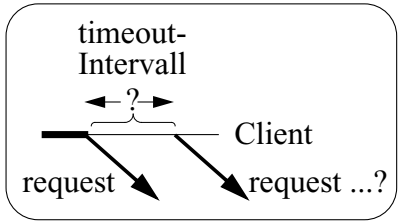
- Ortstransparenz?
  - evtl. muss Server (“Zielort”) bei Adressierung explizit genannt werden
  - erkennbare Trennung der Adressräume von Client und Server
  - keine Kommunikation über globale Variablen möglich
  - i.Allg. keine Pointer/Referenzparameter als Parameter möglich

- Fehlertransparenz?
  - es gibt mehr Fehlerfälle (beim klassischen Prozeduraufruf gilt: Client = Server → “alles oder nix”)
  - partielle (“einseitige”) Systemausfälle: Server-Absturz, Client-Absturz
  - Nachrichtenverlust (ununterscheidbar von zu langsamer Nachricht!)
  - ein Crash kann im “ungünstigen Moment” der Transaktion erfolgen
  - Client / Server haben zumindest zwischenzeitlich eine unterschiedliche Sicht des Zustandes einer “RPC-Transaktion”

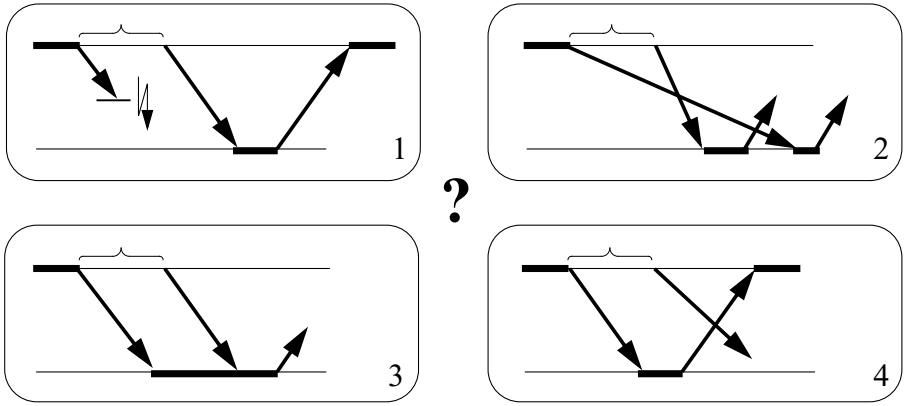
⇒ Fehlerproblematik ist also “kompliziert”!

Typische Fehlerursachen bei RPC: I. Verlorene Request-Nachricht

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

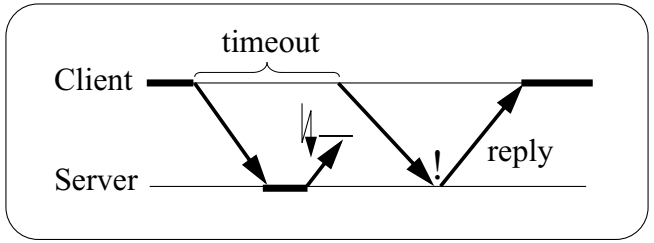


- *Probleme*:
  - Wieviele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
    - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten Operationen!)
    - Server sollte solche Duplikate erkennen. (Wie? 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?



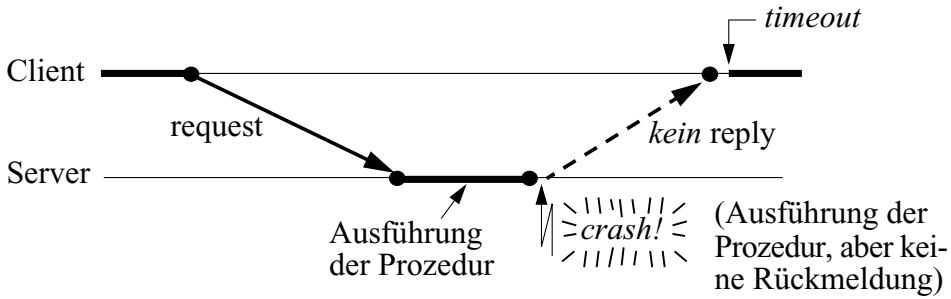
II. Verlorene Reply-Nachricht

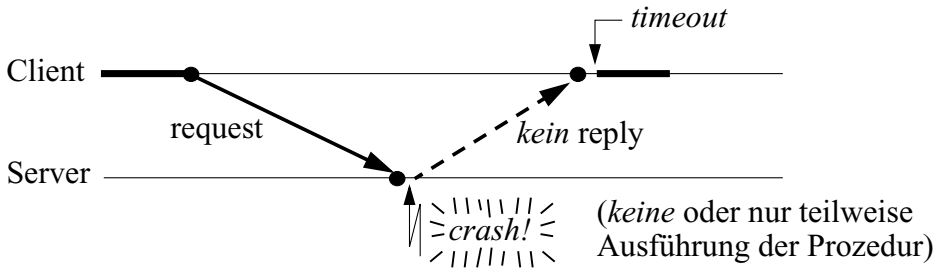
- *Gegenmassnahme 1:* analog zu verlorener Request-Nachricht
  - Also: Anfrage nach Ablauf des Timeouts wiederholen
- *Probleme:*
  - Vielleicht ging aber tatsächlich der Request verloren?
  - Oder der Server war nur langsam und arbeitet noch?
  - Ist aus Sicht des Clients nicht unterscheidbar!



- *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 Speicherprobleme:
      → Historie nach “einiger” Zeit löschen
      (Ist in diesem Zusammenhang ein ack eines Reply sinnvoll?)
      Und wenn man ein gelöschtes Reply später dennoch braucht?

III. Server-Crash





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* u.U. 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: “verschmutzer” Zustand durch frühere Inkarnation
  - typischerweise ungenügend Information (“Server Amnesie”), um in alte Kommunikationszustände problemlos wieder einzusteigen