Kommunikation

Kommunikation

- Prozesse sollen kooperieren, daher untereinander Information austauschen können - über
  - *gemeinsame Daten* in einem globalen Speicher (dieser kann physisch oder ggf. nur logisch vorhanden sein: “virtual shared memory”)
  - oder *Nachrichtenaustausch*: Kopie der Daten an eine entfernte Stelle

- Notwendig, damit die Kommunikation klappt ist jedenfalls:

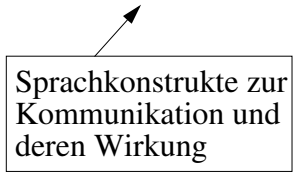
  1) dazwischenliegendes *physikalisches Medium*
     - z.B. elektrische Signale in Kupferkabeln

  2) einheitliche *Verhaltensregeln*
     - Kommunikationsprotokolle

  3) gemeinsame *Sprache* und gemeinsame *Semantik*
     - gleiches Verständnis der Bedeutung von Kommunikationskonstrukten und -Regeln

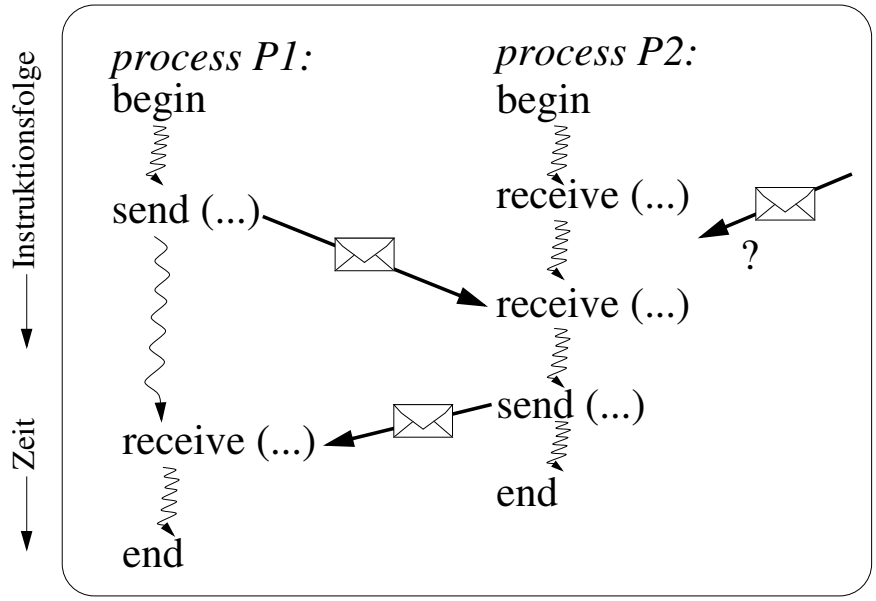
Also trotz Verteiltheit gewisse gemeinsame “Dinge”!

- Wir betrachten im folgenden den dritten Punkt genauer
  - Punkte 1) und 2) sind eher Themen einer Vorlesung über “Rechnernetze”



Nachrichtenbasierte Kommunikation

- "Austausch" von Nachrichten: send --> receive
- Implizite Synchronisation: Senden *vor* Empfangen
  Empfänger erfährt, wie weit der Sender mindestens ist
- Nachrichten sind dynamische Betriebsmittel:
  gegründet beim Senden, verbraucht beim Empfangen
- Interprozesskommunikation - naive Sicht:



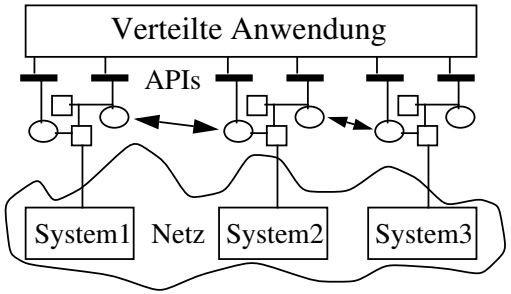
- Welche Kommunikationsanweisungen "matchen"?
- Empfangsbereitschaft, aber keine Nachricht? (verlorene Nachricht?)
- Nachricht, aber keine Empfangsbereitschaft?
- Sprachliche Ausprägung der Kommunikation?
- Wie wird adressiert?

==> Vielfältige Aspekte, Varianten, Probleme...

Message Passing System

einfache Form von "Middleware" (--> später)

- Organisiert den Nachrichtentransport
- Bietet Kommunikationsprimitive (APIs) an
  - z.B. send (...) bzw. receive (...)
  - ggf. auch ganze Bibliothek unterschiedlicher Kommunikationsdienste
  - Verwendbar in gängigen Programmiersprachen (oft zumindest C)



- Besteht aus Hilfsprozessen, Pufferobjekten...
- Verbirgt Details des zugrundeliegenden Netzes
- Verwendet vorhandene Protokolle; implementiert ggf. (damit) neue Protokolle
- Garantiert (abhängig von der "Semantik") gewisse Eigenschaften
  - z.B. Reihenfolgeerhalt
- Abstrahiert von Implementierungsdetails
  - wie z.B. Puffer, Low-level-Adressen etc.
- Maskiert gewisse Fehler
  - z.B. durch automatische Wiederholung nach einem timeout
- Verbirgt Heterogenität unterschiedlicher Rechner- bzw. Betriebssystemplattformen

===> Portabilität!

Nachrichtenkommunikation - pragmatische Aspekte

Vollständige Transparenz lässt sich kaum oder nur sehr teuer realisieren; gelegentlich schlagen Eigenschaften von tieferen Protokollschichten oder der Einsatzumgebung durch, dies betrifft z.B.:

- *Nachrichtenlänge*
  - fest
  - variabel aber begrenzt
  - (prinzipiell) unbegrenzt

  } dann muss man mit solchen Einschränkugen leben und darum herumprogrammieren

- *Zuverlässigkeitsgrad*: Nachrichtenverlust
  - nicht bemerkt
  - vermutet und gemeldet
  - vermieden

  } *qualitatives* Merkmal

- *Zuverlässigkeitsgrad*: Nachrichtenverfälschung
  - nicht bemerkt
  - erkannt und gemeldet
  - automatisch korrigiert

(*Techniken zur Erhöhung des Zuverlässigkeitsgrades:* Timeouts, Quittungen, Sequenznummern, Wiederholungen, Prüfsummen, fehlerkorrigierende Codes,...)

Derartige pragmatische Aspekte müssen in der Praxis neben der eigentlichen Kommunikationssemantik ebenfalls beachtet werden!

Prioritäten von Nachrichten?

- Vgl. dies mit Prioritätsebenen von Unterbrechungen

- *Semantik* zunächst nicht ganz klar:
  - Soll (kann?) Transportsystem Nachrichten höherer Priorität bevorzugt (=?) befördern?
  - Sollen (z.B. bei fehlender Pufferkapazität) Nachrichten niedrigerer Priorität überschrieben werden?
  - Wieviele Prioritätsstufen gibt es?
  - Sollen beim Empfang zunächst 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 *Rechnernetzen*: bei Rückstaus bei den Routern soll z.B. interaktiver Verkehr bevorzugt werden vor ftp etc.)

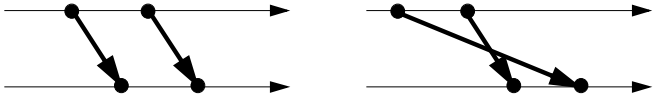
Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!

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

Ordnungserhalt

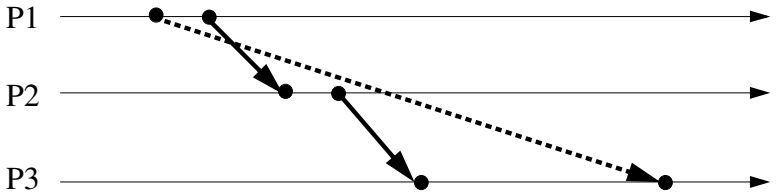
- Oft werden vom Kommunikationssystem keine Garantien bzgl. Nachrichtenreihenfolgen gegeben

- Eine mögliche Garantie stellt FIFO (First-In-First-Out) dar: Nachrichten zwischen zwei Prozessen überholen sich nicht: Sendereihenfolge = Empfangsreihenfolge



FIFO kein FIFO

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



nicht kausal geordnet

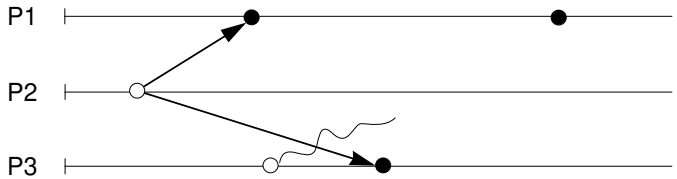
- Möchte man auch dies haben, so muss die Kommunikation "kausal geordnet" sein (Anwendungszweck?)
  - entspricht einer "Globalisierung" von FIFO auf mehrere Prozesse
  - "Dreiecksungleichung": Keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege
  - Denkübung: wie garantiert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

Fehlermodelle (1)

- Fehler sind leider eine Quelle vielfältiger Ärgernisse in verteilten Systemen

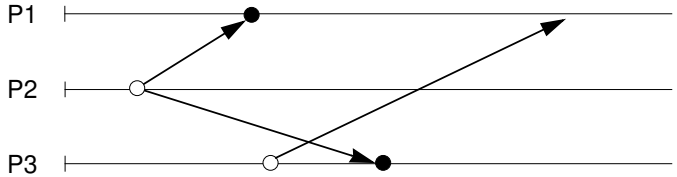
- Klassifikation von Fehlermöglichkeiten; Abstraktion von den konkreten Ursachen

- **Fehlerhaftes Senden**



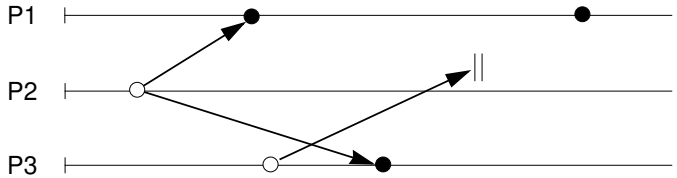
Empfänger merkt nichts; *Sender* stellt dies ggf. fest

- **Fehlerhaftes Empfangen**



Sender merkt nichts; *Empfänger* stellt dies ggf. fest

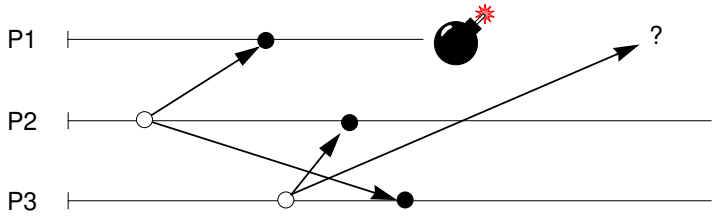
- **Fehlerhaftes Übertragen**



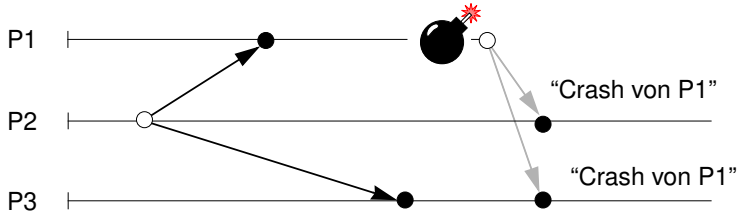
Weder Sender noch Empfänger merken unmittelbar etwas

Fehlermodelle (2)

- **Crash:** Ausfall eines Prozessors ohne Störverhalten



- **Fail-Stop:** Crash mit "Benachrichtigung"



- **Zeitfehler:** Ereignis erscheint zu spät (od. zu früh)
- **Byzantinische Fehler:** Beliebiges Fehlverhalten,
  - verfälschte Nachrichteninhalte
  - Prozess, der unsinnige Nachrichten sendet

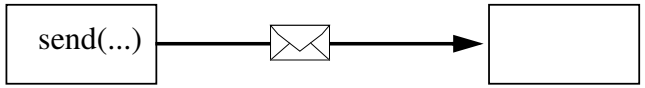
(derartige Fehler lassen sich höchstens bis zu einem gewissen Grad durch *Redundanz* erkennen)

Fehlertolerante Algorithmen sollen das "richtige" Fehlermodell berücksichtigen!

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

Kommunikationsmuster

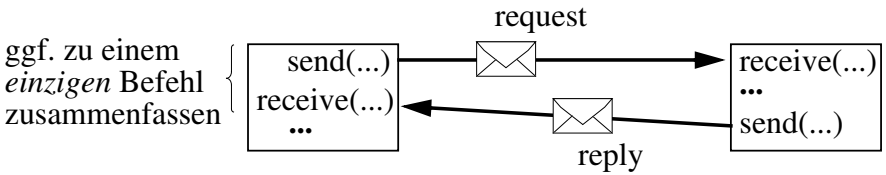
Mitteilungsortientiert:



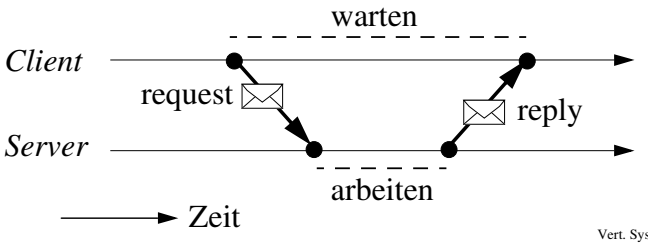
- Unidirektional
- Übermittelte Werte werden der Nachricht in Form von "Ausgabeparametern" beim send übergeben

---

Auftragsortientiert:



- Bidirektional
- "Antwort" (= Ergebnis eines Auftrags) wird zurückgeschickt

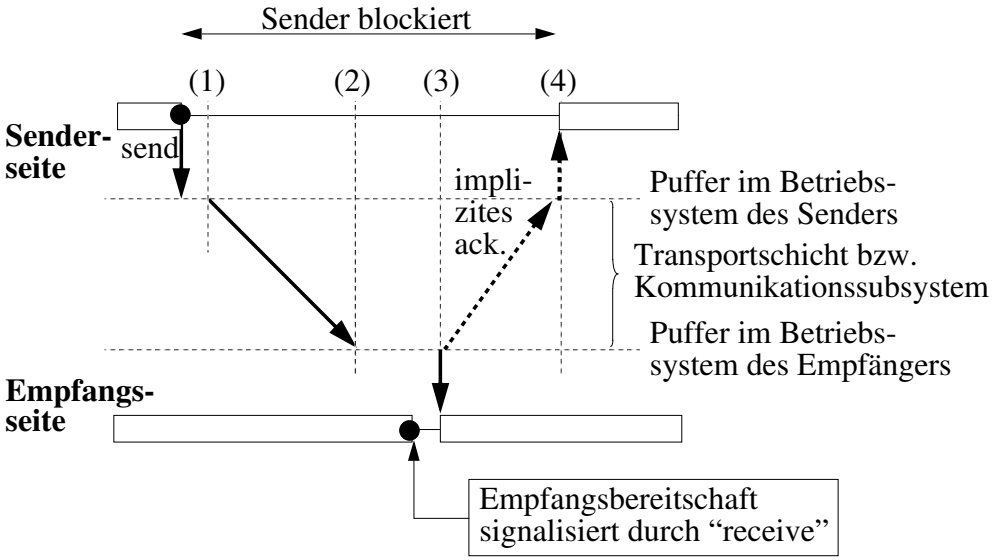


Synchrone Kommunikation

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

  was genau ist das?

  - Sender hat 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, sprachorientierte 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, wenn die Nachricht bei fehlerhafter Übertragung ggf. wiederholt werden muss. (Oft verwendet bei betriebssystemorientierten Betrachtungen.)

Virtuelle Gleichzeitigkeit?

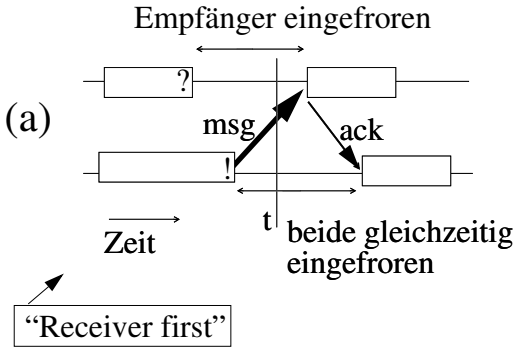
- *Syn-chron* = "gleich"-"zeitig"

- Idealisierung: Send und Receive geschehen *gleichzeitig*

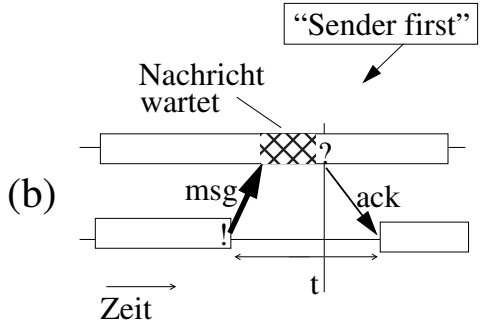
- Wodurch ist diese Idealisierung gerechtfertigt?
  (Kann man auch mit einer Marssonde synchron kommunizieren?)

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

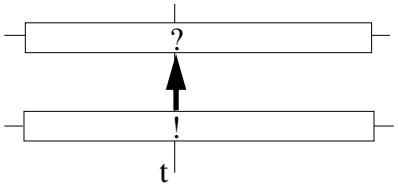
Implementierung:



"Receiver first"



Idealisierung: senkrechte Pfeile in den Zeitdiagrammen



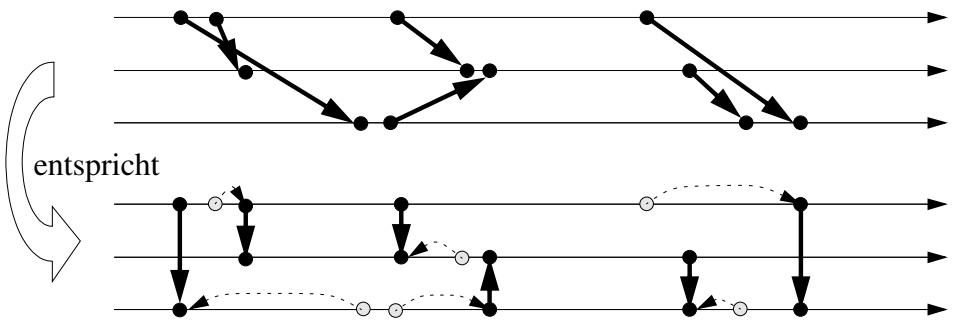
Als wäre die Nachricht zum Zeitpunkt t versendet und empfangen worden!

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

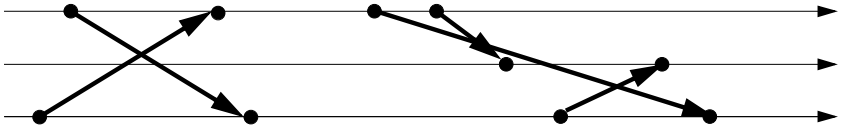
Virtuelle Gleichzeitigkeit

- Eine Berechnung (ohne globale Zeit), die synchrone Kommunikation benutzt, ist durch ein *äquivalentes* Raum-Zeit-Diagramm darstellbar, bei dem alle Nachrichtenpfeile senkrecht verlaufen

  - nur stetige Deformation (“Gummiband-Transformation”)



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



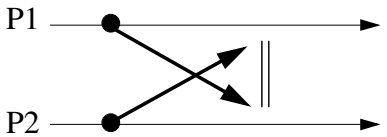
  - aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

(Viel) mehr dazu für besonders Interessierte: Charron-Bost, Mattern, Tel: S*ynchronous, Asynchronous and Causally Ordered Communication.* Distributed Computing, Vol. 9 No. 4, pp. 173 - 191, 1996
http://www.inf.ethz.ch/vs/publ/

Blockaden bei synchroner Kommunikation

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

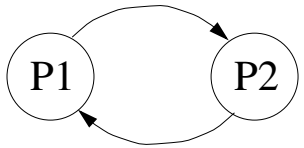
```
P2:
    send (...) to P1;
    receive...
    ...
```



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

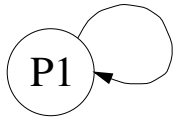
==> *Kommunikationsdeadlock!*

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



“Wait-for-Graph”

Genauso tödlich:

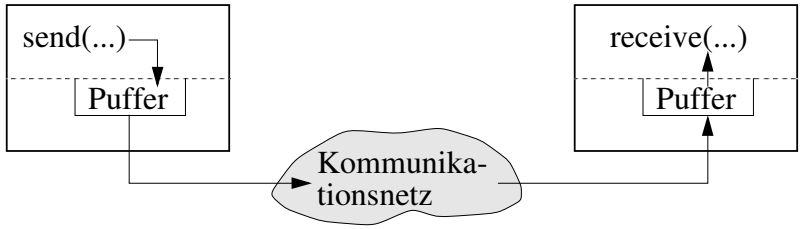



```
P1:
    send (...) to P1;
    receive...
    ...
```

Asynchrone Kommunikation

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

- Jedoch i.a. länger blockiert, falls Betriebssystem 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 angenehmer in der Anwendung)
  --> Sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird
  --> Höherer Grad an Parallelität möglich
  --> Stärkere Entkoppelung von Sender / Empfänger
  --> 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?)
  --> Betriebssystem muss Puffer verwalten (wieviele?)

Sendeoperationen in der Praxis

- Es gibt Kommunikationsbibliotheken, deren Routinen von verschiedenen Programmiersprachen (z.B. C) aus aufgerufen werden können
  - z.B. PVM (Parallel Virtual Machines)
  - oder MPI (Message Passing Interface)

  } Quasi-Standards; verfügbar auf vielen vernetzten Systemen / Parallelrechnern

- Typischer Aufruf einer solchen Send-Operation:

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

  - status: z.B. <0: Fehler; >0: Anzahl der tatsächlich gesendeten Bytes
  - buffer: Adresse des Puffers, in dem die zu verschickende Nachricht steht
  - size: Anzahl der zu versendenden Bytes
  - dest: Zieladresse

- Derartige Bibliotheken bieten i.a. 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 in einem Handbuch beschrieben

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

- Bei Bibliotheken wie MPI wird oft ein Unterschied zwischen synchronem und blockierendem Senden gemacht
  - 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 i.a. einen "handle"

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

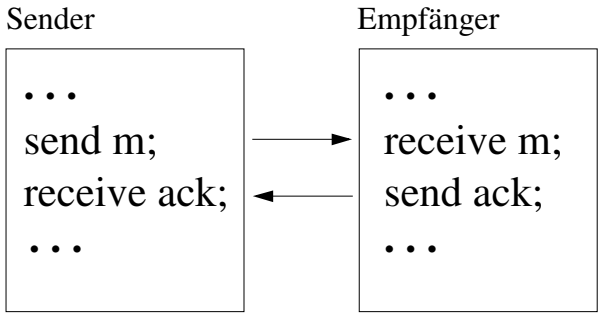
  - Dieser kann in Test- bzw. Warteoperationen verwendet werden
  - Z.B. Test, ob Send-Operation beendet: **`msgdone(handle)`**
  - Z.B. warten auf Beendigung der Send-Operation: **`msgwait(handle)`**

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

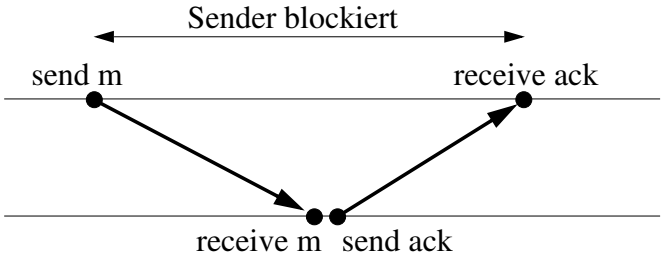
Denkübung: Man überlege, wie synchron+nicht-blockierend wirkt

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:

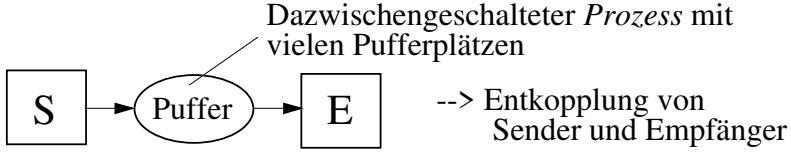


- Explizites Warten auf Acknowledgment im Sender direkt nach dem send (receive ist i.a. blockierend!)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive

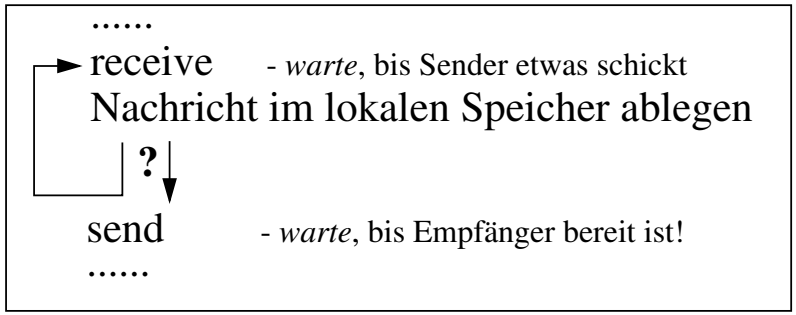


Asynchrone Kommunikation mittels synchroner Kommunikation

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



Wie realisiert man einen Pufferprozess?



Dilemma: Was tut der Pufferprozess nach dem Ablegen der Nachricht im lokalen Speicher?

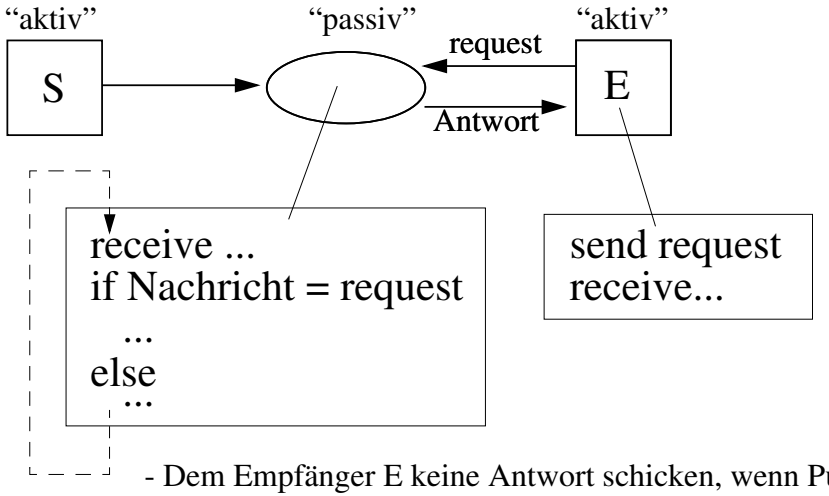
(1) wieder im receive auf den Sender warten, *oder*
(2) in einem send auf den Empfänger warten

--> Entweder Sender oder Empfänger könnte unnötigerweise blockiert sein!

Bemerkung: Puffer der Gösse 1 lassen sich so realisieren ==> Kaskadierung im Prinzip möglich ("Pipeline")

Inversion der Kommunikationsbeziehung

Lösung des zuvorgenannten Problems! (Puffer ist ein *Server*!)

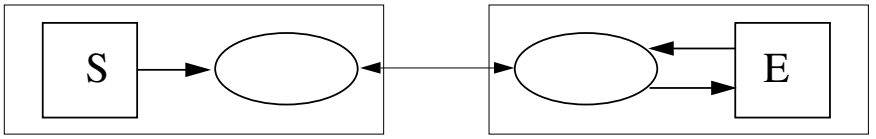


- Dem Empfänger E keine Antwort schicken, wenn Puffer leer.
- Empfänger E wird nur dann verzögert, wenn Puffer leer.
- Für Sender S ändert sich nichts.
- Was tun, wenn der Puffer *voll* ist?

---

wenn S und E auf verschiedenen Prozessoren liegen

- Wo Puffer*prozess* anordnen (S, E, eigener Prozessor)?



- Vielleicht auch zwei kooperierende Pufferprozesse bei S und E?

Beschränkte Puffer

- Puffer haben (in der Praxis immer) *endliche Kapazität*

  --> Pufferprozess sollte Nachricht des Senders nicht entgegennehmen, wenn Pufferpeicher voll

  Dazu zwei getrennte receive-Anweisungen für Nachrichten vom Sender bzw. vom Empfänger

```
forever do
begin
    if Puffer ≠ voll then
    begin
        receive m from Sender;
        füge m in lokalen Pufferspeicher ein;
    end;
    if Puffer ≠ leer then
    begin
        receive request from Empfänger;
        entferne m' aus lokalem Pufferspeicher;
        send m' to Empfänger;
    end;
end
```

oder: **else begin**
receive...
send ("NACK: voll!") **to** Sender;
end;

- So geht es aber nicht: Es wird höchstens eine Nachricht gespeichert, dann ist der Puffer für den Sender blockiert!

Alternatives Empfangen ("select")

Boole'scher Ausdruck; zugehöriges receive kann nur ausgeführt werden, wenn dieser zu *true* evaluiert wird.

```
select
    guard_1 --> receive ...
                weitere Statements
    guard_2 --> receive ...
                weitere Statements
       :
    guard_n --> receive ...
                weitere Statements
endselect
```

Syntax jeweils leicht untersch. bei verschie-denen Sprachen

Hiermit kann der Puffer wie oben skizziert realisiert werden! (Als Denkübung)

==> gute / geeignete Kommunikationskonstrukte sind für die (elegante) verteilte Programmierung wichtig!

- Was geschieht, wenn kein guard "true" ist?
  ==> Blockieren, leere Anweisung, else-Fall...
  (hängt von der intendierten Semantik der Sprache ab)

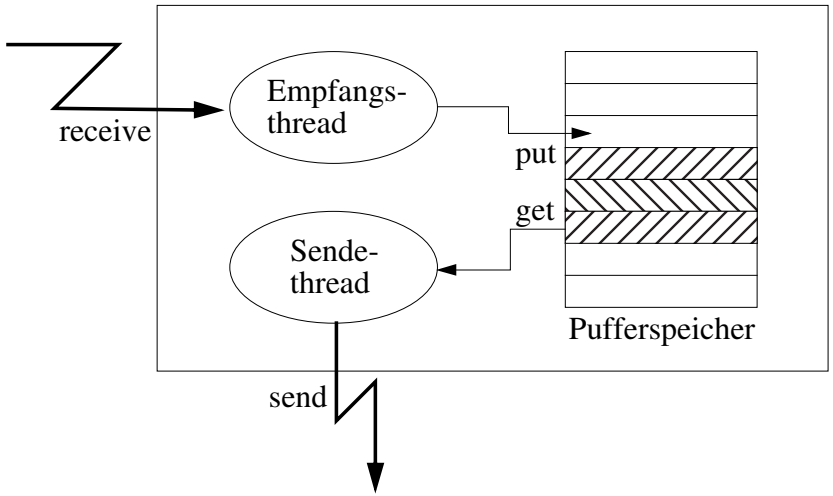
- Was geschieht, wenn mehrere guards "true" sind?
  ==> nichtdeterministische Auswahl, Wahl des ersten Falles...

- Wann genau sollten die guards evaluiert werden?

- Denkübung: Wie implementiert man select-Anweisungen?

Puffer bei Multi-thread-Objekten

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



- Empfangs-thread ist (fast) immer empfangsbereit
  - nur kurzzeitig anderweitig beschäftigt (put in lokalen Pufferspeicher)
  - nicht empfangsbereit, wenn lokaler Pufferspeicher voll

- Sende-thread ist (fast) immer sendebereit

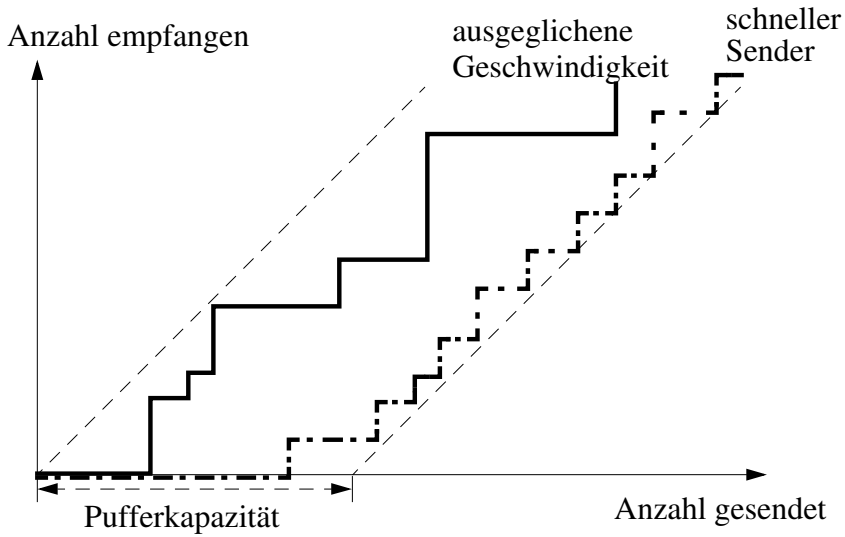
- Pufferspeicher wird i.a. zyklisch verwaltet (FIFO)

- Pufferspeicher liegt im gemeinsamen Adressraum
  ==> *Synchronisation* der beiden Threads notwendig!

  - z.B. Semaphore etc.
  --> “konkurrentes Programmieren”
  --> klassische Betriebssystem-Theorie!

Puffer

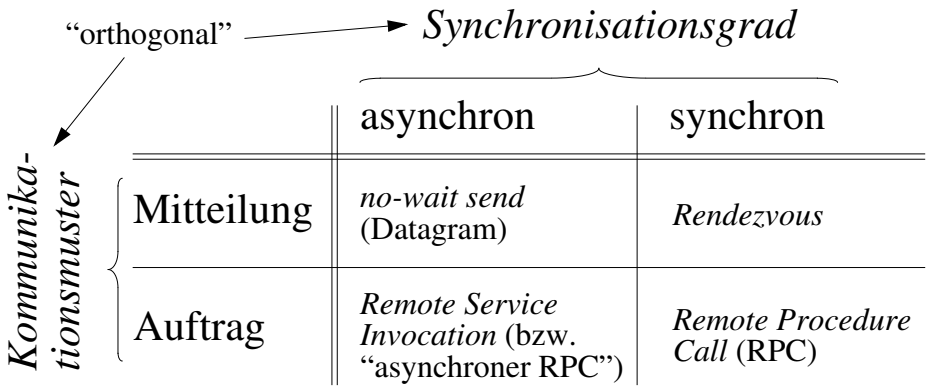
- Entkoppelung von Sender und Empfänger durch Puffer



- Anzahl der Pufferplätze bestimmt “Synchronisationsgrad”
  (Puffer der Grösse 0 ≈ synchrone Kommunikation?)

- Puffer gleicht *Varianz* in der Geschwindigkeit aus,
  nicht die Geschwindigkeiten selbst!

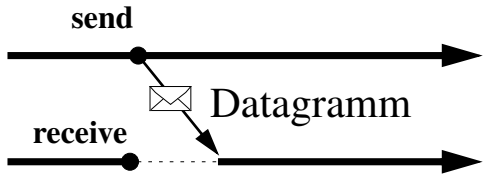
Klassifikation von Kommunikationsmechanismen



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

Datagramm

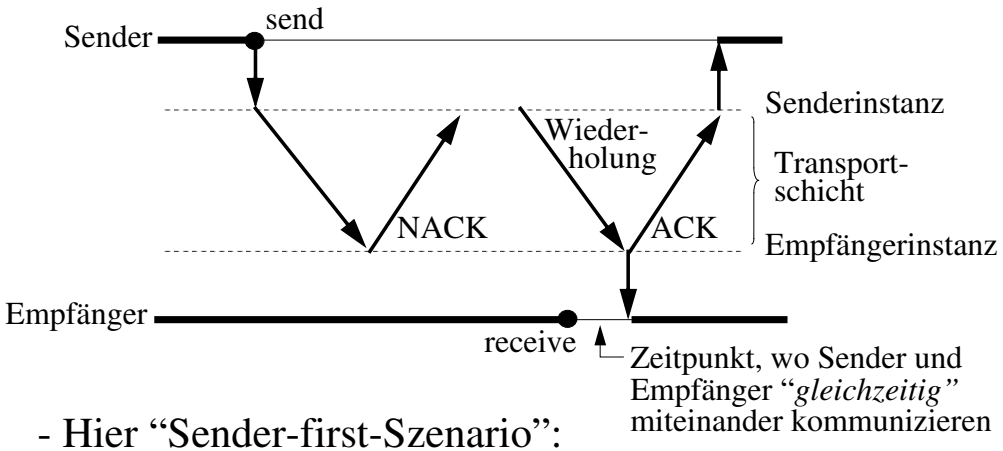
- Asynchron-mitteilungsorientierte Kommunikation



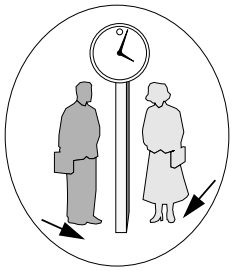
- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- Nachteil
  - keine Erfolgsgarantie für den Sender
  - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicherverwaltung …) im Unterschied etwa zur syn. Kommunikation
  - „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")

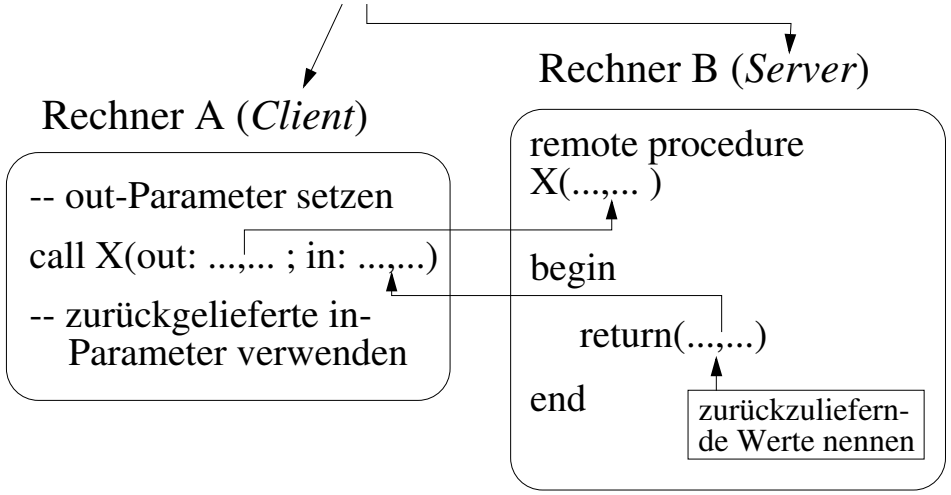
- Mit NACK / ACK ist keine Pufferverwaltung nötig!
  --> Aufwendiges Protokoll! ("Busy waiting")
  - Alternative: Statt NACK, Nachricht auf Empfängerseite puffern
  - Alternative: Statt laufendem Wiederholungsversuch: Empfängerinstanz meldet sich bei Senderinstanz, sobald Empfänger bereit

- Insbes. bei langen (zu paketisierenden) Nachrichten: vorheriges Anfragen, ob bei der Empfängerinstanz genügend Pufferplatz vorhanden ist, bzw. ob Empfänger bereits Synchronisationspunkt erreicht hat

Remote Procedure Call (RPC)

- "Entfernter Prozeduraufruf"

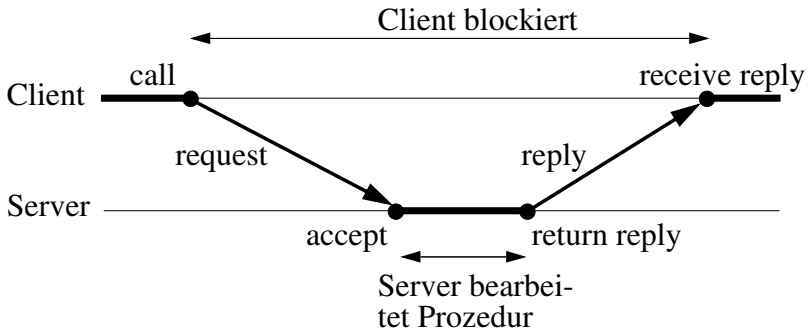
- Synchron; auftragsorientiertiertes Prinzip:



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

- Implementierungsproblem: Verteilungstransparenz
  - Verteiltheit so gut wie möglich verbergen

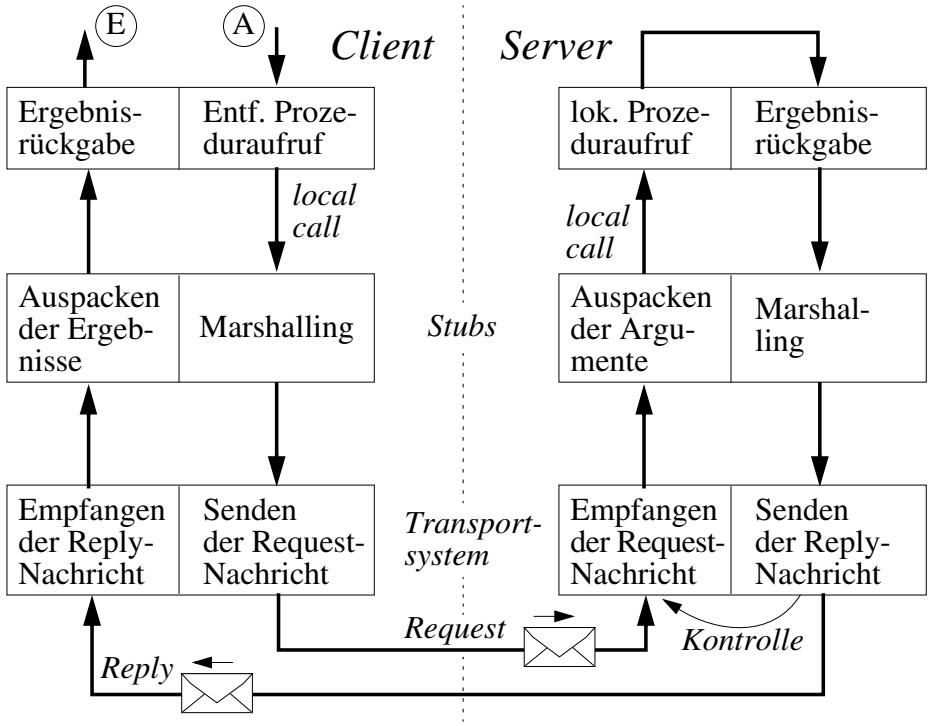
RPC: Prinzipien



--> send --> receive

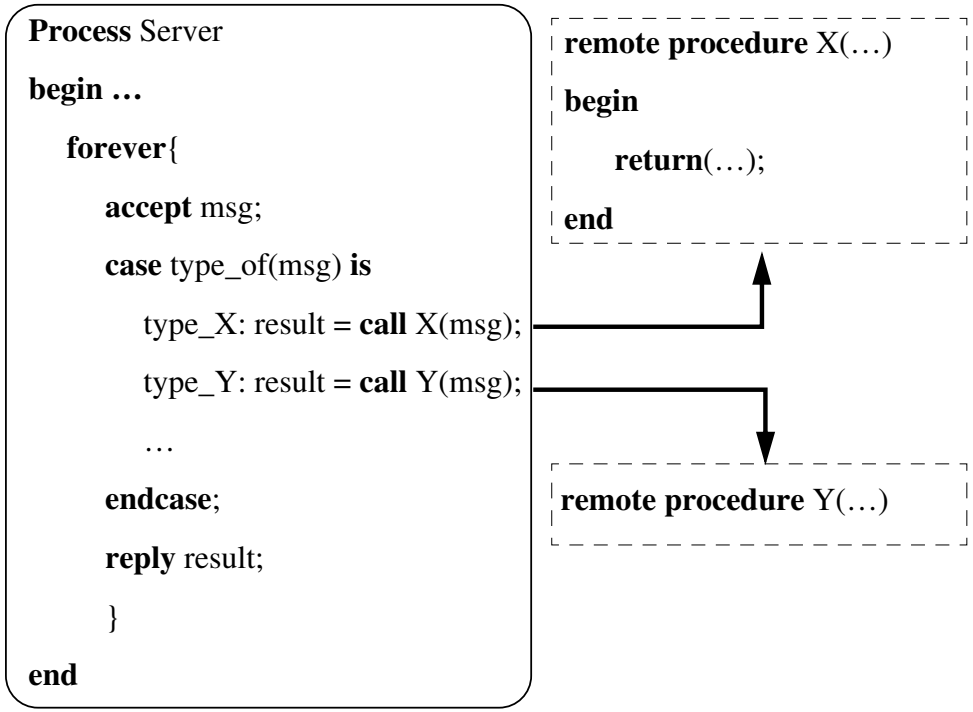
- call; accept; return; receive: interne Anweisungen
  - nicht sichtbar auf Sprachebene --> Compiler
- Call-by-value/result-Parameterübergabe
- Keine Parallelität zwischen Client und Server
  - RPC-Aufrufe sind blockierend

RPC: Implementierung



RPC: Server-Kontrollzyklus

- Warten auf Request, Verzweigen zur richtigen Prozedur:



"Dispatcher"

RPC-Stubs

- *Stub* = Stummel, Stumpf

Client:

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

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

- Parameter in eine Nachricht packt
- Nachricht an Rechner H versendet
- Timeout für die Antwort setzt
- Antwort entgegennimmt (oder ggf. exception bei timeout auslöst)
- Ergebnisparameter mit den Werten der Antwortnachricht setzt

- *Lokale Stellvertreter* des entfernten Gegenübers
  - Client-Stub / Server-Stub
  - simulieren einen lokalen Aufruf
  - sorgen für Packen und Entpacken von Nachrichten
  - konvertieren Datenrepräsentationen bei heterogenen Netzen
  - steuern das Übertragungsprotokoll (z.B. zur fehlerfreien Übertragung)
  - bestimmen ggf. 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 ggf. einer eigenständigen Schnittstellenbeschreibung ("sprachneutral", z.B. IDL oder ASN.1)
  - Compiler kennt (bei streng getypten Sprachen) Datenformate
  - Schnittstelle zum verfügbaren Transportsystem sind auch bekannt
  - Nutzung fertiger Bibliotheken für Formatkonversion, Multithreading usw.
  - Nutzung von Routinen der *RPC-Laufzeitumgebung* (z.B. zur Kommunikationssteuerung, Fehlerbehandlung etc.)

wird nicht generiert, sondern dazugebunden

- Stubs sorgen also für *Transparenz*

RPC: Marshalling

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

- Problem: RPCs werden oft in *heterogenen* Umgebungen eingesetzt mit unterschiedlicher Repräsentation z.B. von
  - Integer (1er <--> 2er Komplement)
  - Gleitkommazahlen
  - Strings (Längenfeld <--> '\0')
  - Character (ASCII <--> Unicode)
  - Arrays (zeilen- <--> spaltenweise)
  - niedrigstes Bit einer Zahl vorne oder hinten im Wort

- Client und Server kennen Typ der Parameter, falls das Programm in Quellform vorliegt oder vom Compiler generierte Typtabellen existieren.
  (Problematisch ggf. bei un- / schwach-getypten Sprachen!)

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

2) Oder Datenrepräsentation in der Nachricht vermerken
   - "receiver makes it right"
   - Vorteil: bei gleichen Maschinentypen ist keine (doppelte) Umwandlung nötig

RPC: Transparenzproblematik

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

bekanntes Programmier-paradigma!

  - Client- / Serverprozesse haben ggf. unterschiedliche Lebenszyklen: Server mag noch nicht oder nicht mehr oder in einer "falschen" Version existieren

- Leistungstransparenz
  - RPC i.a. wesentlich langsamer
  - begrenzte Kommunikationsbandbreite bei umfangreichen Parametern
  - ungewisse, variable Verzögerungen

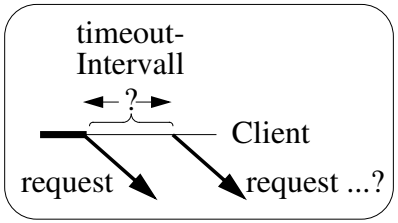
- Ortstransparenz
  - Standort des Servers bei Adressierung u.U. anzugeben
  - erkennbare Trennung der Adressräume von Client und Server
  - i.d.R. keine Pointer/Referenzparameter als Parameter möglich
  - auch keine Kommunikation über globale Variablen möglich

- Fehlertransparenz
  - es gibt mehr Fehlerfälle (beim klassischen Prozeduraufruf gilt: Client = Server --> "fail-stop"-Verhalten: alles oder nix)
  - partielle ("einseitige") Systemausfälle: Server-Absturz, Client-Absturz
  - Nachrichtenverlust (Ununterscheidbar von zu langsamer Nachricht!)
  - Anomalien durch Nachrichtenverdopplung (z.B. nach Timeout)
  - Crash kann zu "ungünstigen Momenten" erfolgen (kurz vor / nach Senden / Empfangen einer Nachricht etc.)
  - Client / Server haben zumindest zwischenzeitlich eine unterschiedliche Sicht des Zustandes einer "RPC-Transaktion"

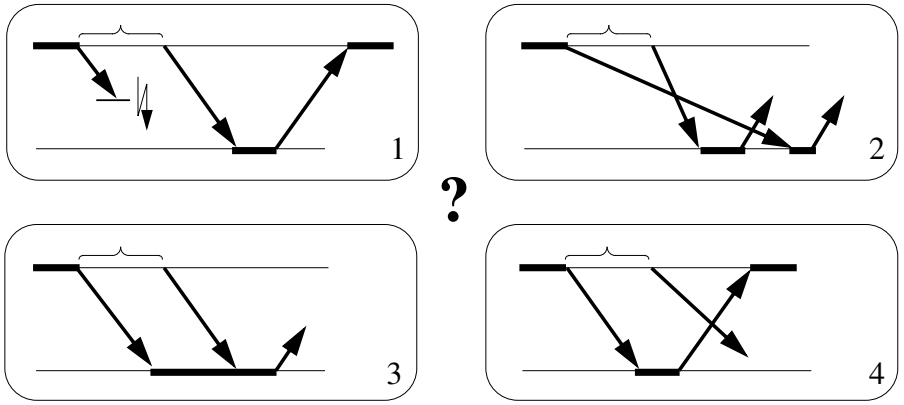
==> Fehlerproblematik ist also "kompliziert"!

Typische Fehlerursachen: I. Verlorene Request-Nachricht

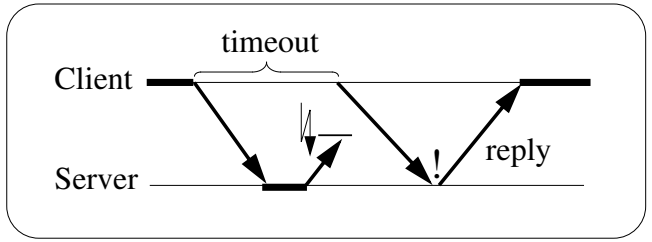
- *Gegenmassnahme*:
  - Nach Ablauf eines Timers ohne Reply die Request-Nachricht erneut senden



- *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?



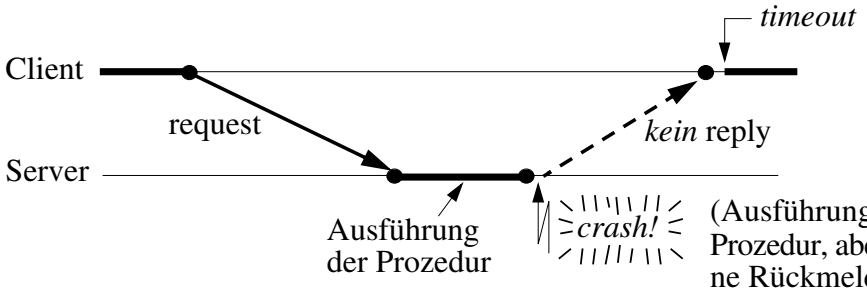
II. Verlorene Reply-Nachricht

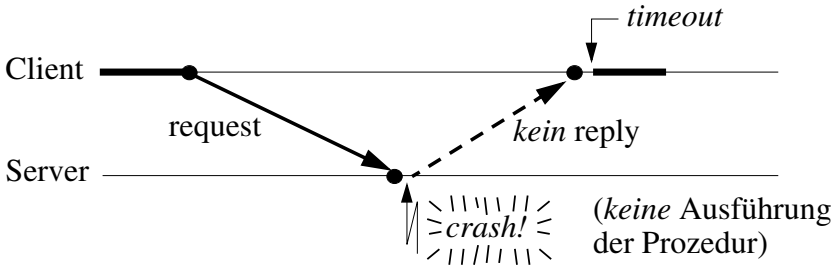


- *Gegenmassnahme 1:* analog zu verlorener Request-Nachricht
  - Also: Anfrage nach Ablauf des Timeouts wiederholen
- *Probleme:*
  - Vielleicht ging aber tatsächlich der Request verloren?
  - Oder der Server war nur langsam und arbeitet noch?
  - Ist aus Sicht des Clients nicht unterscheidbar!

- *Gegenmassnahme 2:*
  - Server könnte eine “Historie” der versendeten Replies halten
    - Falls Server Duplikate erkennt und den Auftrag bereits ausgeführt hat: letztes Reply erneut senden, ohne das Resultat nochmals zu berechnen!
    - 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

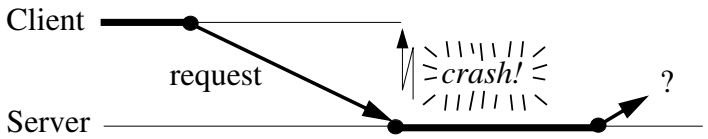




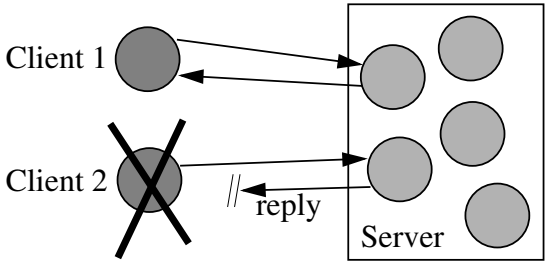
Probleme:

- Wie soll der Client dies unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinnhaftigkeit von Gegenmassnahmen hängt ggf. davon ab
  - Client *meint* u.U. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (--> falsche Sicht des Zustandes!)
- Ggf. 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

IV. Client-Crash



- 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.a. Ressourcen beim Server!
- “Orphans” (Waisenkinder) beim Server
  - Prozesse, deren Auftraggeber nicht mehr existiert



Beispiel: Pro RPC-Auftrag eines Client gründet der Server einen eigenen Prozess

- Nach Neustart des Client dürfen alte Replies nicht stören
  - “Antworten aus dem Nichts” (Gegenmassnahme: Epochen-Zähler)
- Nach Restart könnte ein Client versuchen, Orphans zu killen (z.B. durch Benachrichtigung der Server)
  - dadurch bleiben aber u.U. locks etc. bestehen
  - Orphans könnten 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)

RPC-Fehlersemantik

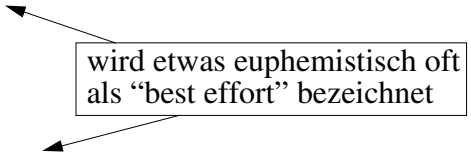
Operationale Sichtweise:

- Wie wird auf (vermeintlich?) nicht eintreffende Requests oder Replies nach einem Timeout und auf wiederholte Requests reagiert?
- Und wie auf gecrashte Server / Clients?

---

1) Maybe-Semantik:

- Keine Wiederholung von Requests
- *Einfach und effizient*
- Keinerlei Erfolgsgarantien --> oft nicht anwendbar
  Mögliche Anwendungsklasse: Auskunftsdienste (noch einmal probieren, wenn keine Antwort kommt)



2) At-least-once-Semantik:

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

RPC-Fehlersemantik (2)

3) At-most-once-Semantik:

- Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
- Keine wiederholte Ausführung der Prozedur, sondern ggf. erneutes Senden des Reply
- Geeignet auch für *nicht-idempotente* Operationen
- Kein Ergebnis bei abgestürztem Server

4) Exactly-once-Semantik:

- Wunschtraum?
- Oder geht es zumindest unter der *Voraussetzung*, dass der Server nicht crasht und ein reply letztlich auch durchkommt? (Z.B. durch hartnäckige Wiederholung von Requests?)
- Was ist mit verteilten Transaktionen? (--> Datenbanken! Stichworte: Checkpoint; persistente Datenspeicherung; Recovery...)

- Nochmals: Fehlertransparenz bei RPC?
  - Problem: Client / Server haben u.U. (temporär?) eine inkonsistente Sicht
  - Einige Fehler sind bei gewöhnlichen Prozeduraufrufen nicht möglich
  - Timeout beim Client kann *verschiedene* Ursachen haben (verlorener Request, verlorenes Reply, langsamer Request bzw. Reply, langsamer Server, abgestürzter Server...) --> Fehlermaskierung schwierig
  - Vollständige Transparenz ist kaum erreichbar
  - Hohe Fehlertransparenz = hoher Aufwand

Wirkung der RPC-Fehlersemantik

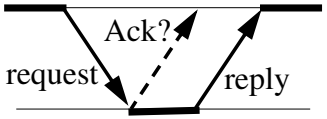
	Fehlerfreier Ablauf	**Nachrichten-verluste**	**Ausfall des Servers**
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-least-once	Ausführung: 1 Ergebnis: 1	Ausführung: ≥1 Ergebnis: ≥1	Ausführung: ≥0 Ergebnis: ≥0
At-most-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0
Exactly-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

May-be ---> At-least-once ---> At-most-once ---> ...
ist zunehmend aufwendiger zu realisieren!

- man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt“ wie nötig

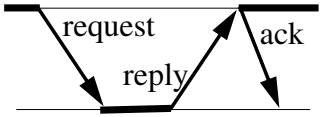
RPC-Protokolle

- *RR-Protokoll* (“Request-Reply”):



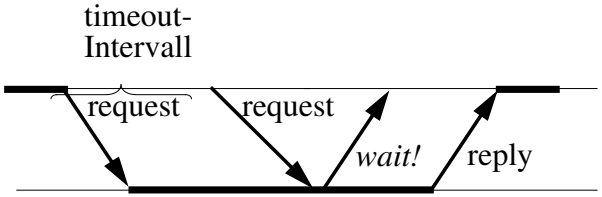
- Reply ist implizite Quittung für Request
- lohnt sich ggf. eine unmittelbare Bestätigung des Request?

- *RRA-Protokoll* (“Request-Reply-Acknowledge”):



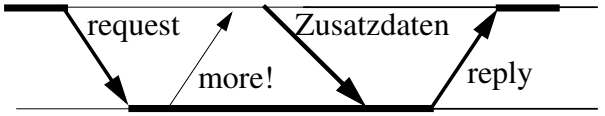
- “pessimistischer” als das RR-Protokoll
- Vorteil: Server kann ggf. gespeicherte Replies frühzeitig löschen (und natürlich Replies bei Ausbleiben des ack wiederholen)

- *Sinnvoll bei langen Aktionen / überlasteten Servern:*



“wait” = Bestätigung eines erkannten Duplikats

- *Parameter-Übertragung „on demand“*



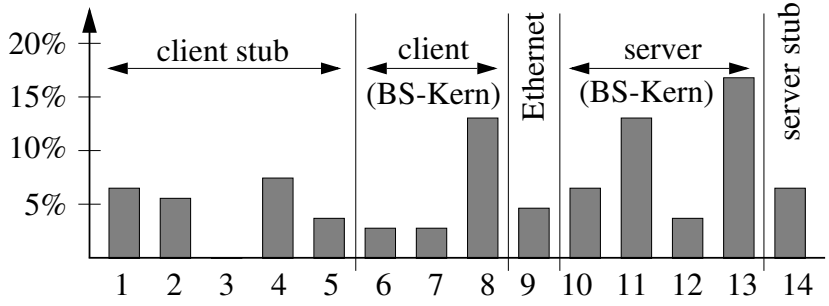
- spart Pufferkapazität
- bessere Flusssteuerung
- Zusatzdaten abhängig vom konkreten Ablauf

- *Weitere RPC-Protokollaspekte:*
  - effiziente Implementierung einer geeigneten (=?) Fehlersemantik
  - geeignete Nutzung des zugrundeliegenden Protokolls (ggf. aus Effizienzgründen eigene Paketisierung der Daten, Flusssteuerung, selektive Wiederholung einzelner Nachrichtenpakete bei Fehlern, eigene Fehlererkennung / Prüfsummen, kryptogr. Verschlüsselung...)

RPC: Effizienz

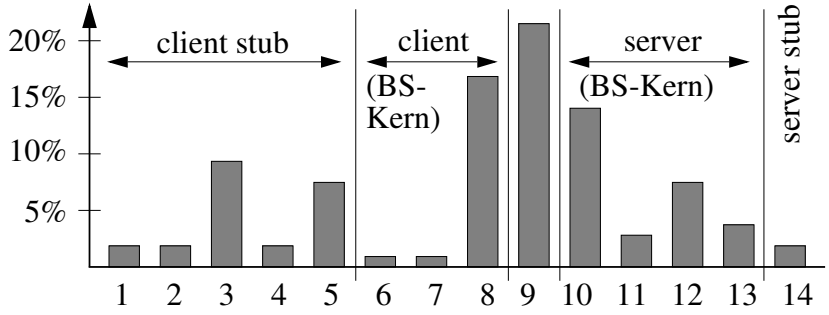
Analyse eines RPC-Protokolls durch Schroeder
(zitiert nach A. Tanenbaum)

a) Null-RPC (Nutznachricht der Länge 0, kein Auftragsbearbeitung):



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

b) 1440 Byte Nutznachricht (ebenfalls kein Auftragsbearbeitung):

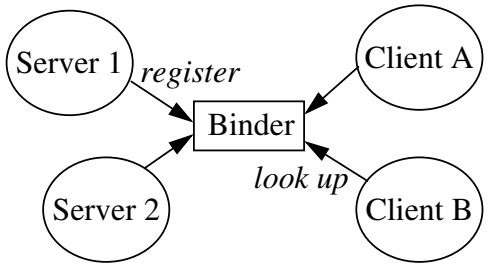


- Eigentliche Übertragung kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zw. Anwendung und BS wichtig
- Mehrfaches Kopieren kostet viel

RPC: Binding

- Problem: Wie werden Client und Server "gematcht"?
  - Verschiedene Rechner und i.a. verschiedene Lebenszyklen --> kein gemeinsames Übersetzen / statisches Binden (fehlende gem. Umgebung)

Dynamisches Binden mittels eines "Binders"



- Server (-stub) gibt den Namen etc. seines Services (RPC-Routine) dem Binder bekannt
  - "*register*"; "exportieren" der RPC-Schnittstelle (Typen der Parameter...)
  - ggf. auch wieder abmelden
- Client erfragt beim Binder die Adresse eines geeigneten Servers
  - "*look up*"; "importieren" der RPC-Schnittstelle

oft auch "registry" oder "look-up service" genannt

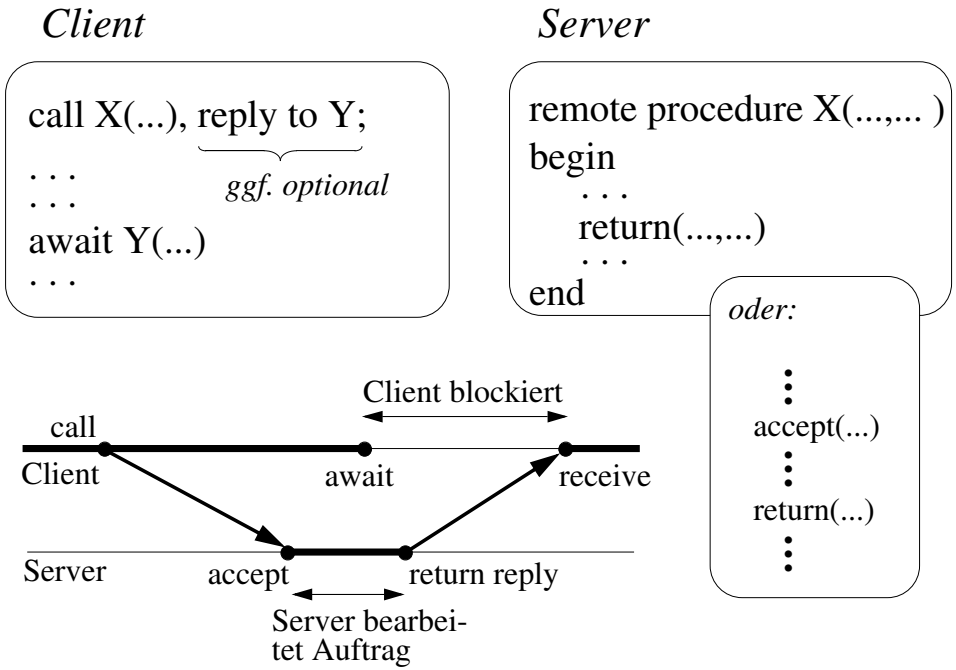
- *Vorteile*: im Prinzip kann Binder
  - mehrere Server für den gleichen Service registrieren (--> Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Existenz eines Services testen
  - verschiedene Versionen eines Dienstes verwalten

dann eher "Trader" oder "Broker"

- *Probleme*:
  - zentraler Binder ist ein potentieller Engpass (Binding-Service geeignet verteilen? Konsistenz!)
  - dynamisches Binden kostet Ausführungszeit

Asynchroner RPC: “Remote Service Invocation”

- auftragsorientiert --> Antwortverpflichtung



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

Future-Variablen

- Zuordnung Auftrag / Ergebnisempfang bei der asynchron-auftragsorientierten Kommunikation?
  - 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);)
  - ggf. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock (vgl. Interrupt-Routine) empfangen werden

- Spracheinbettung ggf. auch durch “Future-Variablen”
  - Future-Variable = handle, der wie ein Funktionsergebnis in Ausdrücke eingesetzt werden kann
  - Auswertung der Future-Variable erst, wenn unbedingt nötig
  - Blockade nur dann, falls Inhalt bei Auswertung noch nicht feststeht
  - Beispiel:

```
FUTURE future: integer;
some_value: integer;
…
future = RSI_call(…);
…
some_value = 4711;
print(some_value + future);
```

Die Socket-Programmierschnittstelle

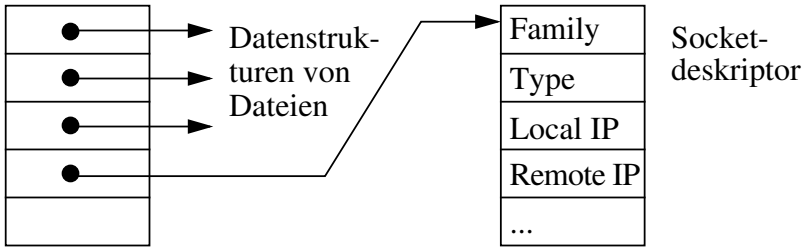
- Zu TCP (bzw. UDP) gibt es keine festgelegten “APIs”
- Bei UNIX ist dafür entstanden: “sockets” als Zugangspunkte zum Transportsystem
  - etwas modernere Alternative: TLI (Transport Layer Interface)
- Semantik eines sockets: analog zu Datei-Ein/Ausgabe
  - ist insbesondere bidirektional (“schreiben” und “lesen”)
  - ein socket kann aber auch mit mehreren Prozessen verbunden sein
- Programmiersprachliche Einbindung (typw. in C)
  - sockets werden wie Variablen behandelt (können Namen bekommen)
  - Beispiel in C (Erzeugen eines sockets):

```
int s;
s = socket(int PF_INET, int SOCK_STREAM, 0);
```

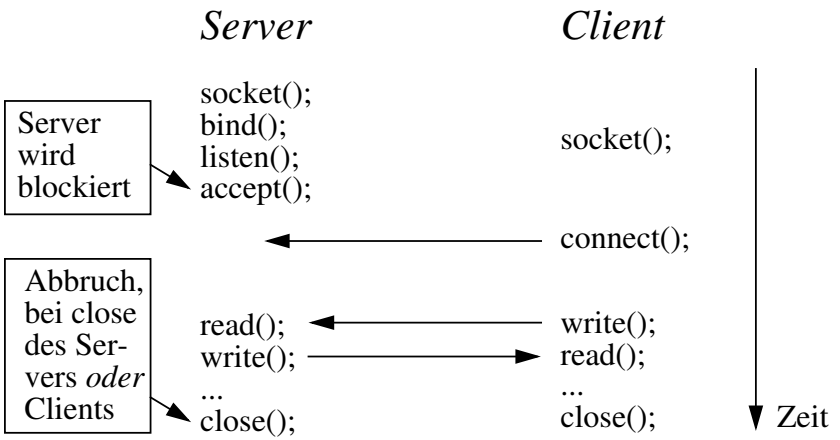
“Family”: Internet oder nur lokale Domäne

“Type”:Angabe, ob TCP verwendet (“stream”); oder UDP (“datagram”)

- Bibliotheksfunktion “socket” erzeugt einen Deskriptor
  - wird innerhalb der Filedeskriptortabelle des Prozesses angelegt
  - Datenstruktur wird allerdings erst mit einem nachfolgenden “bind”-Aufruf mit Werten gefüllt (binden der Adressinformation aus Host-Adresse und einer “bekannten” lokaler Portnummer an den socket)



Client-Server mit Sockets (Prinzip)



- Voraussetzung: Client “kennt” die IP-Adresse des Servers sowie die Portnummer (des Dienstes)
  - muss beim connect angegeben werden
- Mit “listen” richtet der Server eine Warteschlange für Client-connect-Anforderungen ein
  - Auszug aus der Beschreibung: *“If a connection request arrives with the queue full, tcp will retry the connection. If the backlog is not cleared by the time the tcp times out, the connect will fail”*
- Accept / connect implementieren ein “Rendezvous”
  - mittels des 3-fach-Handshake von TCP
  - bei “connect” muss der Server bereits listen / accept ausgeführt haben
- Rückgabewerte von read bzw. write: Anzahl der tatsächlich gesendeten / empfangenen Bytes
- Varianten: Es gibt ein select, ein nicht-blockierendes accept etc., vgl. dazu die UNIX-Bibliothek (“man”...)

Ein Socket-Beispiel in C

- Verwendung von sockets in C erfordert u.a.
  - Header-Dateien mit C-Strukturen, Konstanten etc.
  - Programmcode zum Anlegen, Füllen etc. von Strukturen
  - Fehlerabfrage und Behandlung

- Socket-Programmierung ist ziemlich “low level”
  - umständlich, fehleranfällig bei der Programmierung
  - aber dicht am Netz und dadurch ggf. manchmal von Vorteil (vgl. Assembler-Programmierung)

- Zunächst der Quellcode für den Client:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711
#define BUF_SIZE 1024

main(argc,argv)
int   argc;
char *argv[];
{
    int                 sock, run;
    char                buf[BUF_SIZE];
    struct sockaddr_in  server;
    struct hostent     *hp;
    if(argc != 2)
    {
        fprintf(stderr,"usage: client
                       <hostname>\n");
        exit(2);
    }
```


Socket-Beispiel: Client

```
    /* create socket */
    sock = socket(AF_INET,SOCK_STREAM,0);
    if(sock < 0)
    {
        perror("open stream socket");
        exit(1);
    }
    server.sin_family = AF_INET;
    /* get internet address of host specified by command line */
    hp = gethostbyname(argv[1]);
    if(hp == NULL)
    {
       fprintf(stderr,"%s unknown host.\n",argv[1]);
       exit(2);
    }
    /* copies the internet address to server address */
    bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
    /* set port */
    server.sin_port = PORT;
    /* open connection */
    if(connect(sock,&server,sizeof(struct sockaddr_in)) < 0)
    {
       perror("connecting stream socket");
       exit(1);
    }
    /* read input from stdin */
    while(run=read(0,buf,BUF_SIZE))
    {
       if(run<0)
       {
          perror("error reading from stdin");
          exit(1);
       }
       /* write buffer to stream socket */
       if(write(sock,buf,run) < 0)
       {
          perror("writing on stream socket");
          exit(1);
       }
    }
    close(sock);
}
```

Socket-Beispiel: Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711                    /* random port number */
#define MAX_QUEUE 1
#define BUF_SIZE 1024

main()
{
    int sock_1,sock_2;     /* file descriptors for sockets */
    int rec_value, length;
    char buf[BUF_SIZE];
    struct sockaddr_in server;

    /* create stream socket in internet domain*/
    sock_1 = socket(AF_INET,SOCK_STREAM,0);
    if (sock_1 < 0)
    {
       perror("open stream socket");
       exit(1);
    }
    /* build address in internet domain */
    server.sin_family = AF_INET;
    /* everyone is allowed to connet to server */
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = PORT;
    /* bind socket */
    if(bind(sock_1,&server,sizeof(struct sockaddr_in)))
    {
       perror("bind socket to server_addr");
       exit(1);
    }
```


Socket-Beispiel: Server (2)

```
    listen(sock_1,MAX_QUEUE);
    /* start accepting connection */
    sock_2 = accept(sock_1,0,0);
    if(sock_2 < 0)
    {
       perror("accept");
       exit(1);
    }
    /* read from sock_2 */
    while (rec_value=read(sock_2,buf,BUF_SIZE))
    {
       if(rec_value<0)
       {
          perror("reading stream message");
          exit(1);
       }
       else
          write(1,buf,rec_value);
    }
    printf("Ending connection.\n");
    close(sock_1); close(sock_2);
}
```

- Sinnvolle praktische Übungen:

1) Beispiel genau studieren; Semantik der socket-Operationen etc. nachlesen (Online-Dokumentation von UNIX oder Bücher)

2) Varianten und andere Beispiele implementieren, z.B.:
   - Server, der zwei Zahlen addiert und Ergebnis zurücksendet
   - Produzent / Konsument mit dazwischenliegendem Pufferprozess (unter Vermeidung von Blockaden bei vollem Puffer)
   - Server, der mehrere Clients gleichzeitig bedienen kann
   - Chat- bzw. Konferenzserver analog zu IRC
   - Trader, der geeignete Clients und Server zusammenbringt
   - Messung des Durchsatzes im LAN; Nachrichtenlängen in mehreren Experimenten jeweils verdoppeln

Sockets unter Java

- Auch unter Java lassen sich Sockets verwenden
  - sogar bequemer als unter C
  - Paket java.net.* enthält u.a. die Klasse “Socket”
  - Streamsockets (verbindungsorientiert) bzw. Datagrammsockets

- Beispiel:

```
DataInputStream in;
PrintStream out;
Socket server;
...
server = new Socket(getCodeBase().getHost(),7);
  // Klasse Socket besitzt Methoden
  // getInputStream bzw. getOutputStream, hier
  // Konversion zu DataInputStream / PrintStream:
in = new DataInputStream(server.getInputStream());
out = new PrintStream(server.getOutputStream());
...
  // Etwas an den Echo-Server senden:
out.println(...)
...
  // Vom Echo-Server empfangen; vielleicht
  // am besten in einem anderen Thread:
String line;
while((line = in.readLine()) != null)
  // line ausgeben
...
server.close;
```

Herstellen einer Verbindung

Hostname

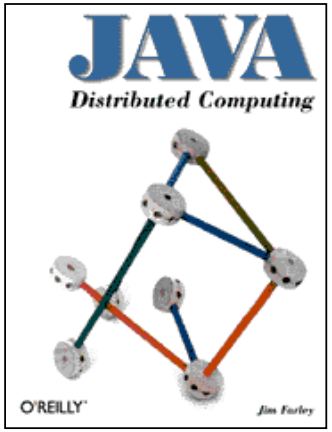
Echo-Port

Port Nummer 7 sendet alles zurück

- Zusätzlich: Fehlerbedingungen mit Exceptions behandeln (“try”; “catch”)
  - z.B. “UnknownHostException” beim Gründen eines Socket

Client-Server mit Sockets in Java

- Beispiel aus dem Buch *Java Distributed Computing* von Jim Farley (O’Reilly, 1998, ISBN 1-56592-206-9)



- Hier der Client:

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

public class SimpleClient
{
  // Our socket connection to the server
  protected Socket serverConn;

    public SimpleClient(String host, int port)
      throws IllegalArgumentException {
    try {
      System.out.println("Trying to connect to "
      + host + " " + port);
      serverConn = new Socket(host, port);
    }
    catch (UnknownHostException e) {
      throw new IllegalArgumentException
        ("Bad host name given.");
    }
    catch (IOException e) {
      System.out.println("SimpleClient: " + e);
      System.exit(1);
    }

    System.out.println("Made server connection.");
  }
```

Konstruktor

```
  public static void main(String argv[]) {
    if (argv.length < 2) {
      System.out.println ("Usage: java \
         SimpleClient <host> <port>");
      System.exit(1);
    }

    int port = 3000;
    String host = argv[0];
    try { port = Integer.parseInt(argv[1]); }
    catch (NumberFormatException e) {}

    SimpleClient client  =  new SimpleClient(host, port);
    client.sendCommands();
  }

  public void sendCommands() {
    try {
      DataOutputStream dout =
        new DataOutputStream(serverConn.getOutputStream());
      DataInputStream din =
        new DataInputStream(serverConn.getInputStream());

      // Send a GET command...
      dout.writeChars("GET goodies ");
      // ...and receive the results
      String result = din.readLine();
      System.out.println("Server says: \"" + result + "\"");
    }
    catch (IOException e) {
      System.out.println("Communication SimpleClient: " + e);
      System.exit(1);
    }
  }

  public synchronized void finalize() {
    System.out.println("Closing down SimpleClient...");
    try { serverConn.close(); }
    catch (IOException e) {
      System.out.println("Close SimpleClient: " + e);
      System.exit(1);
    }
  }
}
```

Host- und Port-nummer von der Kommandozeile

Wird vom Garbage-Collector aufgerufen, wenn keine Referenzen auf den Client mehr existieren ('close' ggf. am Ende von 'sendCommands')

Der Server

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

public class SimpleServer {
  protected int portNo = 3000;
  protected ServerSocket clientConnect;

  public SimpleServer(int port) throws
      IllegalArgumentException {
    if (port <= 0)
      throw new IllegalArgumentException(
       "Bad port number given to SimpleServer constructor.");

    // Try making a ServerSocket to the given port
    System.out.println("Connecting server socket to port");
    try { clientConnect = new ServerSocket(port); }
    catch (IOException e) {
     System.out.println("Failed to connect to port " + port);
      System.exit(1);
    }

    // Made the connection, so set the local port number
    this.portNo = port;
  }

  public static void main(String argv[]) {
    int port = 3000;
    if (argv.length > 0) {
      int tmp = port;
      try {
        tmp = Integer.parseInt(argv[0]);
      }
      catch (NumberFormatException e) {}
      port = tmp;
    }

    SimpleServer server = new SimpleServer(port);
    System.out.println("SimpleServer running on port " +
       port + "...");
    server.listen();
  }
```

Default-Port, an dem der Server auf eine Client-Verbindung wartet

Socket, der Verbindungs-wünsche entgegennimmt

Konstruktor

Portnummer von Kommandozeile

Aufruf der Methode "listen" (siehe unten)

```
public void listen() {
    try {
      System.out.println("Waiting for clients...");
      while (true) {
        Socket clientReq = clientConnect.accept();
        System.out.println("Got a client...");
        serviceClient(clientReq);
      }
    }
    catch (IOException e) {
      System.out.println("IO exception while listening.");
      System.exit(1);
    }
  }

  public void serviceClient(Socket clientConn) {
    SimpleCmdInputStream inStream = null;
    DataOutputStream outStream = null;
    try {
      inStream = new SimpleCmdInputStream
        (clientConn.getInputStream());
      outStream = new DataOutputStream
        (clientConn.getOutputStream());
    }
    catch (IOException e) {
      System.out.println("SimpleServer: I/O error.");
    }
    SimpleCmd cmd = null;
    System.out.println("Attempting to read commands...");
    while (cmd == null || !(cmd instanceOf DoneCmd)) {
      try { cmd = inStream.readCommand(); }
      catch (IOException e) {
        System.out.println("SimpleServer (read): " + e);
        System.exit(1);
      }

      if (cmd != null) {
        String result = cmd.Do();
        try { outStream.writeBytes(result); }
        catch (IOException e) {
          System.out.println("SimpleServer (write): " + e);
          System.exit(1);
        }
      }
    }
  }
}
```

Warten auf connect eines Client, dann Gründen eines Sockets

Von DataInput-Stream abgeleitete Klasse

Klasse SimpleCmd hier nicht gezeigt

Schleife zur Entgegennahme und Ausführung von Kommandos

finalize-Methode hier nicht gezeigt

Java als "Internet-Programmiersprache"

- Java hat eine Reihe von Konzepten, die die Realisierung verteilter Anwendungen erleichtern, z.B.:
  - Socket-Bibliothek zusammen mit Input- / Output-Streams
  - Remote Method Invocation (RMI): Entfernter Methodenaufruf mit Transport (und dabei Serialisierung) auch komplexer Objekte
  - CORBA-APIs
  - eingebautes Thread-Konzept
  - java.security-Paket
  - plattformunabhängiger Bytecode mit Klassenlader (Java-Klassen können über das Netz transportiert und geladen werden; Bsp.: Applets)

Im Vergleich zu RPC: Nicht notw. Master-Slave, sondern peer-to-peer

Damit z.B. Realisierung eines "Meta-Protokolls": Über einen Socket vom Server eine Klasse laden (und Objekt-Instanz gründen), was dann (auf Client-Seite) ein spezifisches Protokoll realisiert. (Vgl. "mobiler Code", "mobile Agenten", Jini...)

- Das UDP-Protokoll kann mit "Datagram-Sockets" verwendet werden, z.B. so:

```
try {
  DatagramSocket s = new DatagramSocket();
  byte[] data = {'H','e','l','l','o'};
  InetAddress addr = InetAddress.getByName("my.host.com");
  DatagramPacket p = new DatagramPacket(data,
    data.length, addr, 5000);
  s.send(p);
}
catch (Exception e) {
  System.out.println("Exception using datagrams:");
  e.printStackTrace();
}
```

Port-Nummer

  - entsprechend zu "send" gibt es ein "receive"
  - InetAddress-Klasse repräsentiert IP-Adressen
  - diese hat u.a. Methoden "getByName" (klassenbezogene Methode) und "getAddress" (instanzbezogene Methode)
  - UDP ist verbindungslos und unsicher (aber effizient)

URL-Verbindungen in Java

- Java bietet einfache Möglichkeiten, auf "Ressourcen" (i.w. Dateien) im Internet mit dem HTTP-Protokoll lesend und schreibend zuzugreifen
  - falls auf diese mittels einer URL verwiesen wird
- Klasse "URL" in java.net.*
  - auf höherem Niveau als die Socket-Programmierung
  - Sockets (mit TCP) werden vom Anwender verborgen benutzt
- Beispiel: zeilenweises Lesen einer Textdatei
  - aber auch hier noch diverse Fehlerbedingungen abfangen!

```
// Objekt vom Typ URL anlegen:
URL myURL;
myURL = new URL("http", ..., "/Demo.txt");
...
DataInputStream instream;
instream = new DataInputStream(myURL.openStream());
String line = "";
while((line = instream.readLine()) != null)
  // line verarbeiten
...
```

hier Hostname angeben

Name der Datei

- Es gibt auch Möglichkeiten, Daten an eine URL zu senden (POST-Methode, z.B. an ein CGI-Skript)
- Ferner: Information über das Objekt ermitteln
  - z.B. Grösse, Kodierung, letztes Änderungsdatum, HTTP-Header etc.
- Es existiert eine ContentHandlerFactory, die den MIME-Typ prüft und einen ContentHandler dafür liefert
  - ContentHandler liest ein Objekt über eine URL-Verbindung und konstruiert lokal eine geeignete Objekt-Instanz

Übungsbeispiel: Ein Bookmark-Checker

```
import java.io.*;
import java.net.*;
import java.util.Date;
import java.text.DateFormat;

public class CheckBookmark {

  public static void main (String args[]) throws
       java.io.IOException, java.text.ParseException {

    if (args.length != 2) System.exit(1);

    // Create a bookmark for checking...
    CheckBookmark bm = new CheckBookmark(args[0], args[1]);

    bm.checkit(); // ...and check

    switch (bm.state) {
      case CheckBookmark.OK:
        System.out.println("Local copy of " +
           bm.url_string + " is up to date"); break;
      case CheckBookmark.AGED:
        System.out.println("Local copy of " +
           bm.url_string + " is aged"); break;
      case CheckBookmark.NOT_SUPPORTED:
        System.out.println("Webserver does not support \
           modification dates"); break;
      default: break;
    }
  }

  String url_string, chk_date;
  int state;

  public final static int OK = 0;
  public final static int AGED = 1;
  public final static int NOT_SUPPORTED = 2;

  CheckBookmark(String bm, String dtm) // Constructor
  { url_string = new String(bm);
    chk_date = new String(dtm);
    state = CheckBookmark.OK;
  }
```

```
  public void checkit() throws java.io.IOException,
       java.text.ParseException {

    URL checkURL = null;
    URLConnection checkURLC = null;

    try { checkURL = new URL(this.url_string); }
    catch (MalformedURLException e) {
      System.err.println(e.getMessage() + ": Cannot \
          create URL from " + this.url_string);
      return;
    }

    try {
      checkURLC = checkURL.openConnection();
      checkURLC.setIfModifiedSince(60);
      checkURLC.connect();
    }

    catch (java.io.IOException e) {
      System.err.println(e.getMessage() + ": Cannot \
          open connection to " + checkURL.toString());
      return;
    }

    // Check whether modification date is supported
    if (checkURLC.getLastModified() == 0) {
      this.state = CheckBookmark.NOT_SUPPORTED;
      return;
    }

    // Cast last modification date to a "Date"
    Date rem = new Date(checkURLC.getLastModified());

    // Cast stored date of bookmark to Date
    DateFormat df = DateFormat.getDateInstance();
    Date cur = df.parse(this.chk_date);

    // Compare and set flag for outdated bookmark
    if (cur.before(rem)) this.state = CheckBookmark.AGED;
  }
}
```


Adressierung

- *Sender* muss in geeigneter Weise spezifizieren, wohin die Nachricht gesendet werden soll
  - ggf. mehrere Adressaten zur freien Auswahl (Lastverteilung, Fehlertoleranz)
  - ggf. mehrere Adressaten gleichzeitig (Broadcast, Multicast)

- *Empfänger* ist ggf. nicht bereit, jede beliebige Nachricht von jedem Sender zu akzeptieren
  - selektiver Empfang (Spezialisierung)
  - Sicherheitsaspekte, Überlastabwehr

- Probleme
  - *Ortstransparenz*: Sender weiss *wer*, aber nicht *wo* (sollte er i.a. auch nicht!)
  - *Anonymität*: Sender und Empfänger kennen einander nicht (sollen sie oft auch nicht)

Kenntnis von Adressen?

- Adressen sind u.a. Rechneradressen (z.B. IP-Adresse oder Netzadresse auf Ethernet-Basis), Portnamen, Socketnummern, Referenzen auf Mailboxes...

- Woher kennt ein Sender die Adresse des Empfängers?

  1) Fest in den Programmcode integriert --> unflexibel

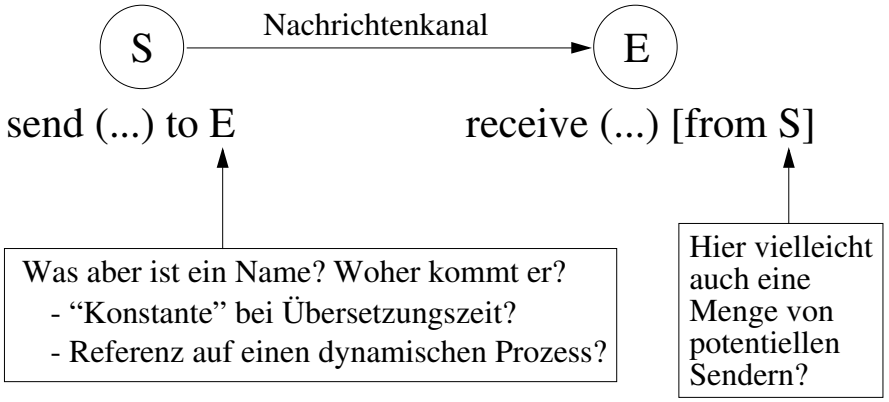
  2) Über Parameter erhalten oder von anderen Prozessen mitgeteilt

  3) Adressanfrage per Broadcast “in das Netz”
     - häufig bei LANs: Suche nach lokalem Nameserver, Router etc.

  4) Auskunft fragen (Namensdienst wie z.B. DNS; Lookup-Service)
     - wie realisiert man dies effizient und fehlertolerant?

Direkte Adressierung

- *Direct Naming* (1:1-Kommunikation):

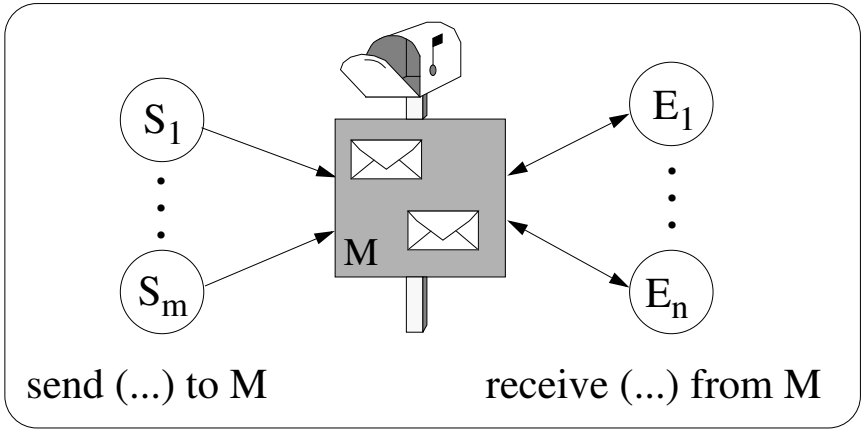


- Direct naming ist insgesamt relativ unflexibel

- Empfänger (= Server) sollten nicht gezwungen sein, potentielle Sender (= Client) explizit zu nennen
  - Symmetrie ist also i.a. gar nicht erwünscht

Indirekte Adressierung - Mailbox

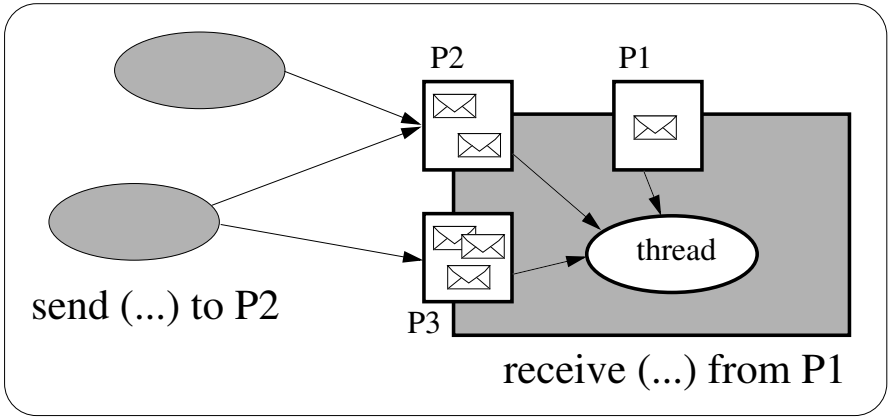
- m:n-Kommunikation möglich



- Eine Nachricht besitzt i.a. mehrere potentielle Empfänger
  - Mailbox spezifiziert damit eine *Gruppe* von Empfängern
- Kann jeder Empfänger die Nachricht bearbeiten?
  - Mailbox i.a. typisiert: nimmt nur bestimmte Nachrichten auf
  - Empfänger kann sich u.U. Nachrichten der Mailbox ansehen / aussuchen...
  - aber wer garantiert, dass jede Nachricht irgendwann ausgewählt wird?
- Wo wird die Mailbox angesiedelt? (--> Implementierung)
  - als ein einziges Objekt auf irgendeinem (geeigneten) Rechner?
  - repliziert bei den Empfängern? Abstimmung unter den Empfängern notwendig (--> verteiltes Cache-Kohärenz-Problem)
  - Nachricht verbleibt in einem Ausgangspuffer des Senders: Empfänger müssen sich bei allen (welche sind das?) potentiellen Sendern erkundigen
- Mailbox muss gegründet werden: Wer? Wann? Wo?

Indirekte Adressierung - Ports

- m:1-Kommunikation
- Ports sind Mailboxes mit genau einem Empfänger
  - Port gehört diesem Empfänger
  - Kommunikationsendpunkt, der die interne Struktur abkapselt
- Ein Objekt kann i.a. mehrere Ports besitzen

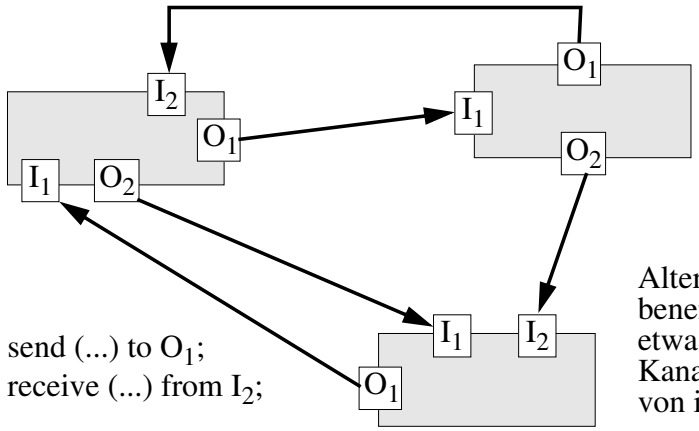


Pragmatische Aspekte (Sprachdesign etc.):

- Sind Ports statische oder dynamische Objekte?
- Wie erfährt ein Objekt den Portnamen eines anderen (dynamischen) Objektes?
  - können Namen von Ports verschickt werden?
- Sind Ports typisiert?
  - unterstützt den selektiven Nachrichtenempfang
- Grösse des Nachrichtenpuffers?
- Können Ports geöffnet und geschlossen werden?
  - genaue Semantik?

Kanäle und Verbindungen

- Neben *Eingangsports* (“in-port”) lassen sich auch *Ausgangsports* (“out-port”) betrachten



Alternativ: Kanäle benennen und etwas auf den Kanal senden bzw. von ihm lesen

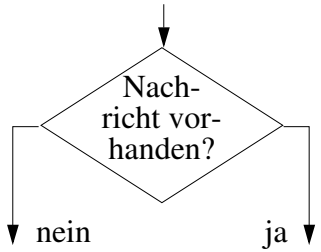
- Ports können als Ausgangspunkte für das Einrichten von *Verbindungen* (“Kanäle”) gewählt werden
- Dazu werden je zwei in- / out-Ports miteinander verbunden. Dies kann z.B. mit einer connect-Anweisung geschehen: **connect** p1 **to** p2
  - denkbar sind auch broadcastfähige Kanäle
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen

Konfigurationsphase

- Grössere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
  - dadurch auch Anonymisierung von Sender und Adressat
- Kommunikationsbeziehung: wahlweise 1:1, n:1, 1:n, n:m

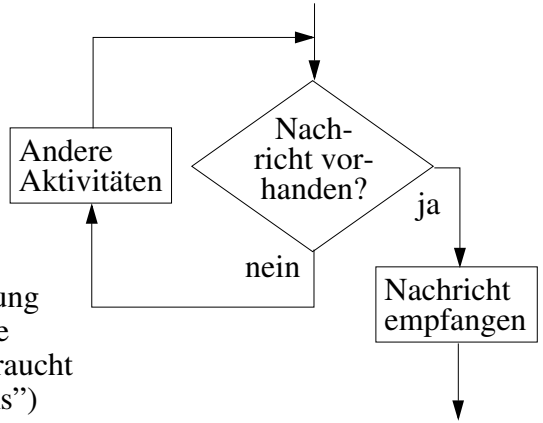
Varianten beim Empfangen von Nachrichten - Nichtblockierung

- Typischerweise ist ein “receive” blockierend
- Aber auch nichtblockierender Empfang denkbar:



- “Non-blocking receive”
- Sprachliche Realisierung z.B. durch “Returncode” eines als Funktionsaufruf benutzten “receive”

- Aktives Warten:
  (“busy waiting”)
  - Nachbildung des blockierenden Wartens wenn “andere Aktivitäten” leer
  - Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte (“spin locks”)



- Weitere Möglichkeit: unterbrechungsgesteuertes (“asynchrones”) Empfangen der Nachricht (--> nicht unproblematisch!)

Nichtblockierendes, alternatives Empfangen

- Sprachliche Realisierung z.B. so:

```
select

    guard_1 --> receive
                    Statements
        ⋮
    guard_n --> receive
                    Statements
    [ else . . . ]

endselect
```

else-Fall wird gewählt, wenn kein guard 'true' ist bzw. dort momentan keine Nachricht anliegt

- Aktives Warten durch umschliessende while-Schleife
  - im else-Fall könnte dann die while-Bedingung auf false gesetzt werden, falls das aktive Warten abgebrochen werden soll, oder es könnte mittels timer ("wait") eine kurze Zeit gewartet werden...
  - else-Fall kann auch einfach das leere Statement enthalten

- Typischerweise blockierend, wenn else-Alternative ganz fehlt

Zeitüberwachter Nachrichtenempfang

- Empfangsanweisung soll maximal (?) eine gewisse Zeit lang blockieren ("timeout")
  - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat

- Sinnvoll bei:
  - Echtzeitprogrammierung
  - Vermeidung von Blockaden im Fehlerfall (etwa: abgestürzter Kommunikationspartner)
    --> dann sinnvolle Recovery-Massnahmen treffen ("exception")
    --> timeout-Wert "sinnvoll" setzen!

Quelle vielfältiger Probleme...

- Timeout-Wert = 0 kann ggf. genutzt werden, um zu testen, ob eine Nachricht "jetzt" da ist

---

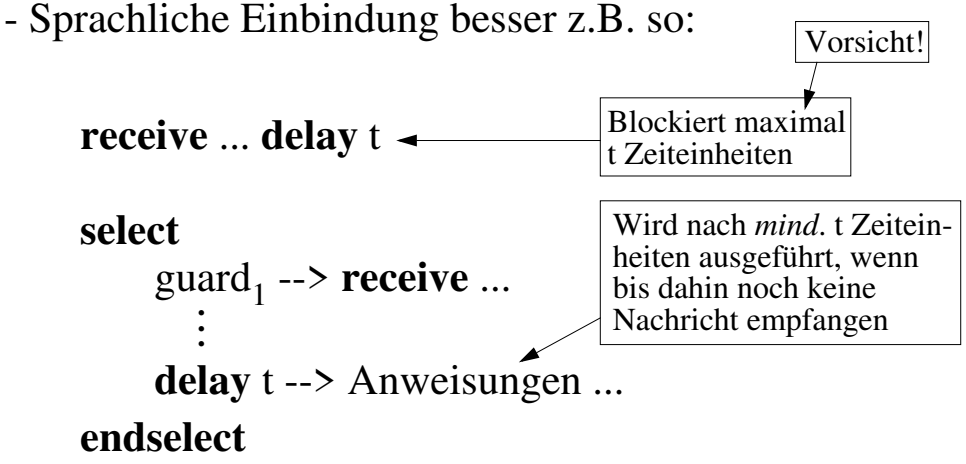
- Analog ggf. auch für synchrones (!) *Senden* sinnvoll
  --> Verkompliziert zugrundeliegendes Protokoll: Implizite Acknowledgements kommen nun "asynchron" an...

Zeitüberwachter Empfang

- Möglicher Realisierung:
  - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
  - Sprungziel könnte z.B. eine Unterbrechungsroutine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive

  --> "systemnahe", unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

- Sprachliche Einbindung besser z.B. so:



- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!

- Frage: ist "delay 0" äquivalent zu "else"?

Selektives Empfangen

≠ alternatives!

- Bedingung an den *Inhalt* (bzw. Typ, Format,...) der zu empfangenden Nachricht

- Dadurch werden gewisse ("unpassende") Nachrichten einfach ausgeblendet

- Bedingung wird oft vom aktuellen Zustand des Empfängers abhängen

- Implementierung:
  - Eingangspuffer wird nach passender Nachricht durchsucht, *oder:*
  - Sender und Empfänger "verhandeln"

- Vorteil bei der Anwendung:

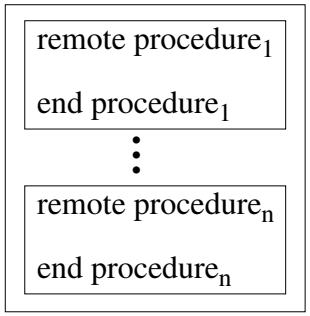
  Empfänger muss nicht mehr alles akzeptieren und zwischenspeichern, sondern bekommt nur das, was ihn momentan interessiert

Implizites Empfangen

- Keine receive, select...-Anweisung, sondern Spezifikation von Routinen, die bei Vorliegen einer Nachricht ausgeführt (“angesprungen”) werden

- z.B. RPC:

bzw. asynchrone Variante oder “Remote Method Invocation” bei objektorientierten Systemen



- Analog auch für den “Empfang” einer Nachricht *ohne Antwortverpflichtung* denkbar

Gelegentlich ist allerdings ein explizites Empfangen (“receive”) wesentlich bequemer als ein implizites Empfangen!

- *Semantik*:
  - Interne Parallelität?
    - Mehr als eine gleichzeitig aktive Prozedur, Operation, thread... (“Routine”) im Empfänger?
    - Vielleicht sogar mehrere Instanzen der gleichen Routine?
  - Atomare Routinen?
    - Wird eine aktive Routine ggf. unterbrochen, um eine andere aktivierte auszuführen?

Sprachaspekte

- Kommunikation (und Synchronisation) soll in geeigneter Weise sprachlich unterstützt werden

---

1) Aufruf von Primitiven des Betriebs- oder Kommunikationssystems
   - bzw. allgemein gebräuchliche Kommunikationsbibliothek

2) Ergänzung einer existierenden Sprache
   - zusätzliche Kommunikationsanweisungen (Aufruf von Methoden / Prozeduren)
   - oder gar syntaktische Erweiterungen (--> Änderungen am Compiler)

   } + ggf. neues Laufzeitsystem

3) Eigene Programmiersprache
   - “aus einem Guss” (Homogenität der Konzepte; syntaktisch und semantisch harmonische Integration)
   - sichereres Programmieren --> bessere (robustere) Software
   - Beispiele: CSP, Occam, Ada (allerdings eher weniger gebräuchlich)

Dazu festlegen:

- Syntax und Semantik der Kommunikation
- Art der kommunizierenden Objekte
- Harmonisierung mit “sequentiellen” Konzepten

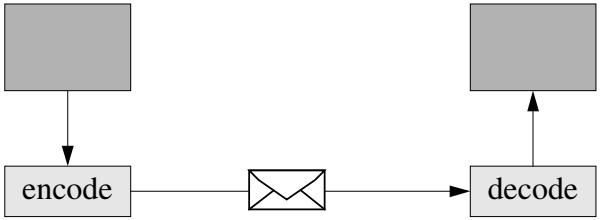
Sprachdesign: Modularisierung

- Einheiten der
  - Strukturierung
    - bezüglich Softwaretechnologieaspekten (information hiding, Schnittstellen, getrennte Übersetzung...)
  - Verteilung
    - gemeinsamer Namensraum
    - Cluster “lokaler” und zusammengehöriger Objekte
    - nur als ganze Einheit auf Rechner plazierbar
  - Parallelität
    - mehrere Module arbeiten nebenläufig (“gleichzeitig”)
    - z.B. thread, Prozess,...

- Getrennte oder einheitliche Sprachmittel dafür?
  - Gesichtspunkte wie Zweckmässigkeit, Anwendungsbereich, Effizienz, Sicherheit, Einfachheit...
- Hierarchien, Schachtelungen von Modulen?
- Granularität; interne Parallelität?
- Dynamik <--> Statik ?
  - Ports, Kanäle, Mailboxen... dynamisch erzeugbar?
  - Dynamik ist flexibler, aber aufwendiger in der Realisierung und problematischer in der Benutzung (Benennung, “dangling references”)

Kommunizierbare Datentypen?

- Werte von “klassischen” einfachen Datentypen
  - int, character, string, floating point,...
- Kompatibilität in heterogenen Systemen?
  - Grösse von int
  - Format von floating point
  - höherwertiges Bit links oder rechts
  - ...



- Vereinbarung einer *Standardrepräsentation* (z.B. XDR)
- marshalling (encode / decode) kostet Zeit

- Was ist mit *komplexen Datentypen* wie
  - Records, Strukturen
  - Objekte, ADT
  - Referenzen, Zeiger
  - Zeigergeflechte

  } sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein?
  } sollen Referenzen symbolisch, relativ... interpretiert werden? Ist das stets möglich?
  } wie wird Typkompatibilität sichergestellt?
- Ggf. “Linearisieren” und ggf. Strukturbeschreibung mitschicken (u.U. sprachunabhängig, z.B. mit ASN.1)
- Sind (Namen von) Ports, Prozessen... eigene Datentypen, deren Werte versendet werden können?

“first class objects”

Ausflug: ASN.1

ASN.1 ist eine Sprache zur Beschreibung strukturierter Information, die vor allem zur Spezifikation von Kommunikationsprotokollen eingesetzt wird.

Auszug aus: http://whatis.com/asn1.htm:

ASN.1 (Abstract Syntax Notation One) is a standard way to describe a message (a unit of application data) that can be sent or received in a network. ASN.1 is divided into two parts:

(1) the rules of syntax for describing the contents of a message in terms of data types and content sequence or structure and

(2) how you actually encode each data item in a message.

ASN.1 is defined in two ISO standards for applications intended for the Open Systems Interconnection (OSI) framework.

Here's an example of a message definition specified with ASN.1 notation:

```
Report ::= SEQUENCE {
    author        OCTET STRING,
    title         OCTET STRING,
    body          OCTET STRING,
    biblio        Bibliography
}
Bibliography ::=  SEQUENCE  {
      author        OCTET STRING
      title         OCTET STRING
      publisher     OCTET STRING
      year          OCTET STRING
}
```

Auszug aus: http://www.techapps.co.uk/asn1gloss.html:

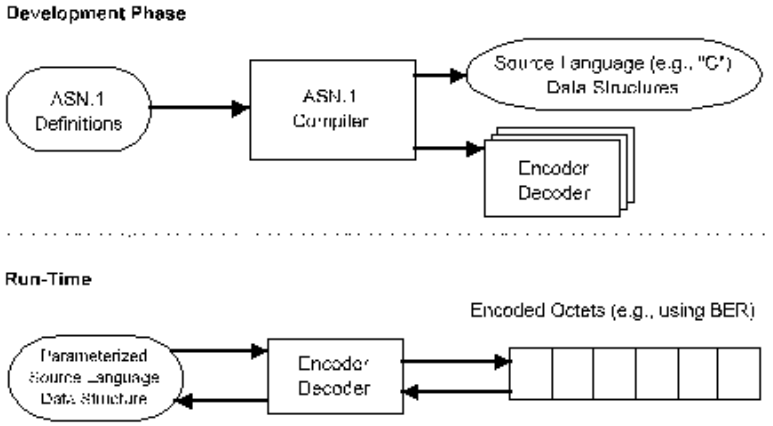
Prior to ASN.1, information to be conveyed in communication protocols was typically specified by ascribing meanings to particular bits and bytes in protocol messages, much as programmers, before the advent of high level languages, had to deal with the bits and bytes of storage layout.

With ASN.1, the protocol designer can view and describe the relevant information and its structure at a high level and need not be unduly concerned with how it is represented while in transit.

ASN.1 comes into its own when the information being described is complex. This is because the language allows arbitrarily complex structures to be built up in a uniform way from simpler components, and ultimately from a few simple information types. ASN.1 is, in effect, a data definition language, allowing a designer to define the parameters in a protocol data unit without concern as to how they are encoded for transmission.

Given any ASN.1 description of a message, a representation can be derived mechanically by applying a set of encoding rules. While many such sets could be imagined, initially only a single set, the Basic Encoding Rules (BER), were standardised as a companion standard to ASN.1.

ASN.1 (2)



The Use of ASN.1 Compilers

ASN.1 notation is recognisable as a high level definition language. It is constructed in modules with unique identifiers. There are over 20 built data types such as Simple data types (BOOLEAN, INTEGER, ENUMERATED, REAL, BIT STRING, OCTET STRING, NULL, ...), Character strings (NumericString, PrintableString...)...

Arbitrarily complex structures can be built up from these data types using constructors such as:

SET{} - order not significant

SEQUENCE{} - fixed order

A type may be simple or structured. The simple types are the basic building blocks of ASN.1, and include types like Boolean and integer. A simple type will generally be used to describe a single aspect of something. A structured type, on the other hand, is defined in terms of other types - its components - and its values are made up of values of the component types. Each of these components may itself be simple or structured, and this nesting can proceed to an arbitrary depth, to suite the needs of the application. All structured types are ultimately defined in terms of simple types.

ASN.1 makes available to the abstract syntax designer a number of simple types, as well as techniques for defining structured types and subtypes. The designer employs these types by using the type notation which ASN.1 provides for each such type.

Earlier standards such as ASCII and EBCDIC specified both the abstract syntax (the letter A) and the encoding, or transfer syntax, (hexadecimal 21 or 41). ASN.1 separates these two concepts, such that at connect time you can chose to encode the data. You can chose an encoding which is efficient on the line or reliable or easy to decode. The first defined for ASN.1 was the Basic Encoding Rules (BER). The BER allow the automatic derivation of a transfer syntax for every abstract syntax defined using ASN.1.