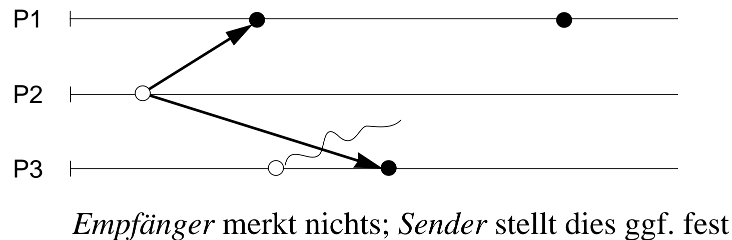
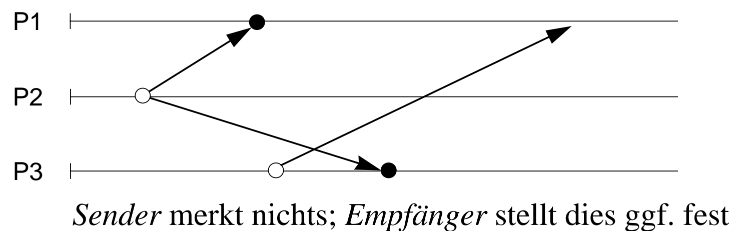
Fehlermodelle (1)

- Fehler sind leider eine Quelle vielfältiger Ärgernisse in verteilten Systemen
- Klassifikation von Fehlermöglichkeiten; Abstraktion von den konkreten Ursachen
- **Send-ommission:** Fehlerhaftes Senden



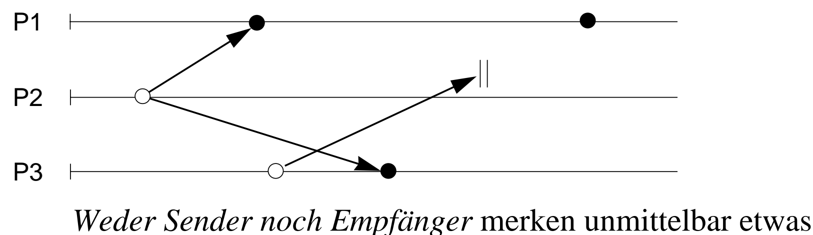
Empfänger merkt nichts; *Sender* stellt dies ggf. fest

- **Receive-ommission:** Fehlerhaftes Empfangen



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

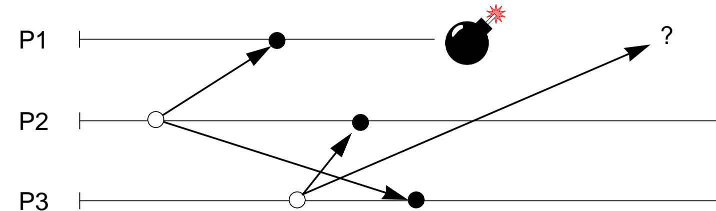
- **Link-ommission:** Fehlerhaftes Übertragen



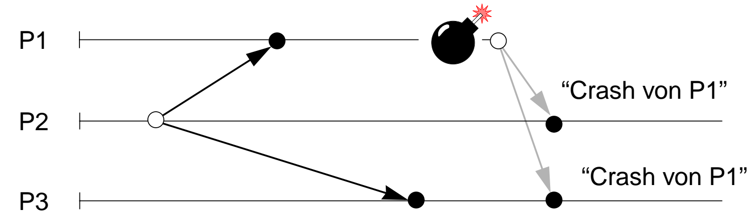
Weder Sender noch Empfänger merken unmittelbar etwas

Fehlermodelle (2)

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



- **Fail-Stop:** Crash mit “Benachrichtigung”



- **Timing-Fehler:** Ereignis erscheint zu früh / zu spät
- **Byzantinische Fehler:** Beliebiges Fehlverhalten, z.B.:
  - verfälschte Nachrichteninhalte
  - Prozess, der unsinnige Nachrichten sendet

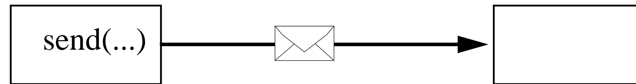
(derartige Fehler lassen sich höchstens bis zu einem gewissen Grad durch *Redundanz* erkennen)

Fehlertolerante Algorithmen sollen das “richtige” Fehlermodell berücksichtigen!

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

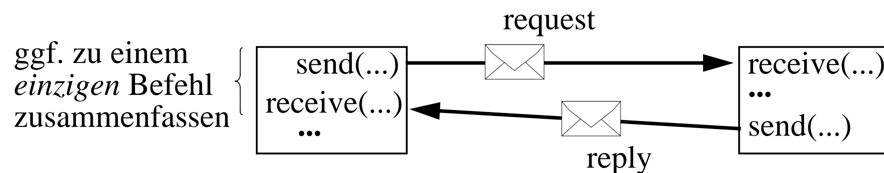
Kommunikationsmuster

Mitteilungsortientiert:

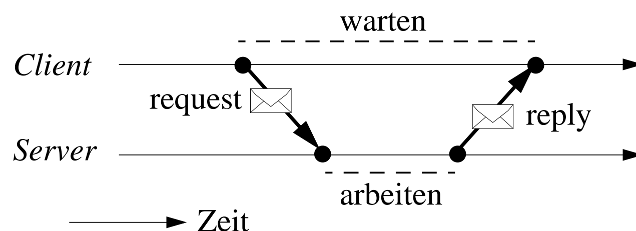


- Unidirektional
- Übermittelte Werte werden der Nachricht in Form von “Ausgabeparametern” beim send übergeben

Auftragsortientiert:



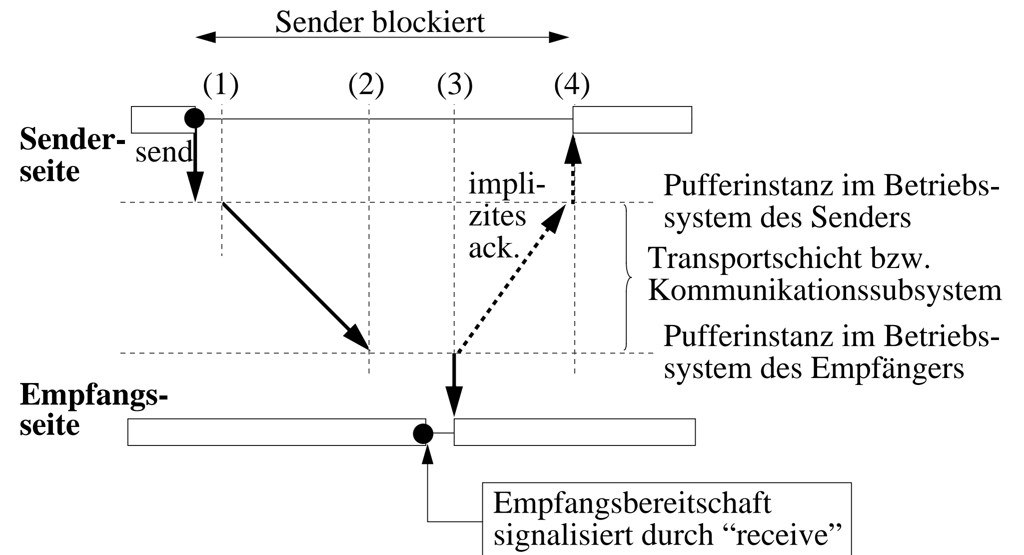
- Bidirektional
- “Antwort” (= Ergebnis eines Auftrags) wird zurückgeschickt



Synchrone Kommunikation

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

was genau ist das?

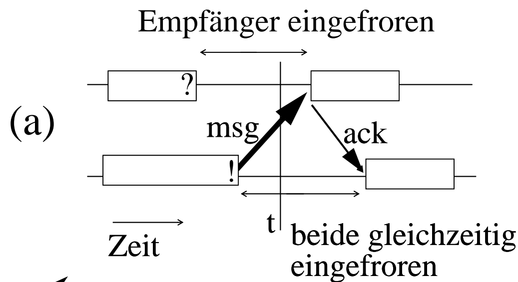


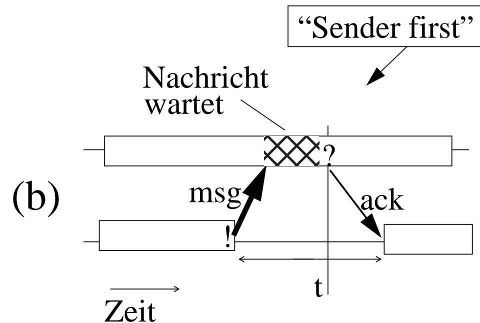
- Verschiedene Ansichten der “korrekten” Definition von “Abschluss der Transaktion” aus Sendersicht:
  - *Zeitpunkt 4* (automatische Bestätigung, dass der Empfänger das receive ausgeführt hat) ist die höhere, sprachorientierte Sicht.
  - Falls eine Bestätigung bereits zum *Zeitpunkt 2* geschickt wird, 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?

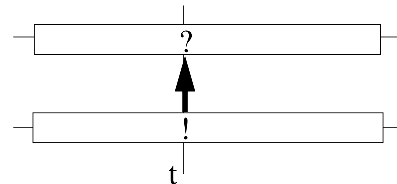
- *Syn-chron* = “gleich”-”zeitig”
- Idealisierung: Send und Receive geschehen *gleichzeitig*
- Wodurch ist diese Idealisierung gerechtfertigt?
  (Kann man auch mit einer Marssonde synchron kommunizieren?)
- Bem.: “Receive” ist i.a. blockierend (d.h. Empfänger wartet so lange, bis Nachricht eingetroffen)

Implementierung:





Idealisierung:
senkrechte Pfeile in den Zeitdiagrammen



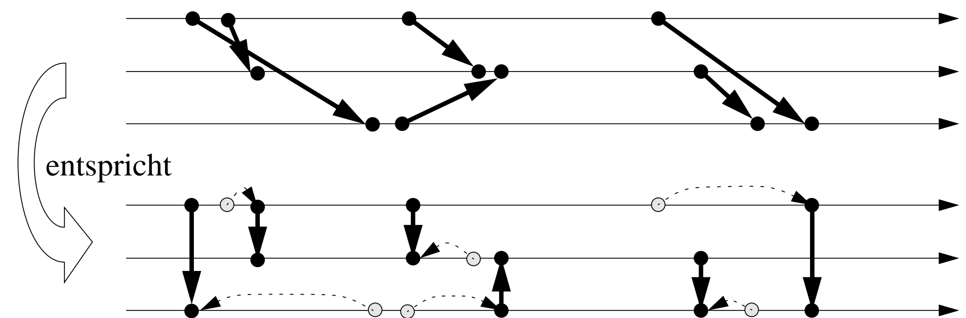
Als wäre die Nachricht zum Zeitpunkt t versendet und empfangen worden!

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

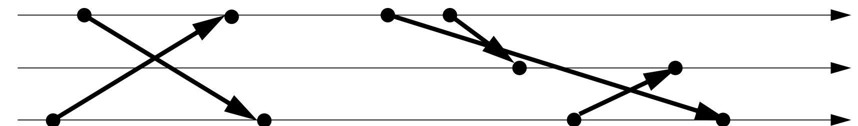
--> *“Rendez-Vous”*

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?)



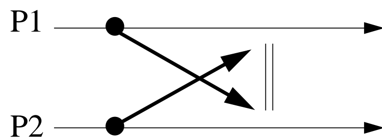
  - aber was geschieht denn, wenn man mit synchronen Kommunikationsprimitiven so programmiert, dass dies provoziert wird?

(Viel) mehr dazu für besonders Interessierte: Charron-Bost, Mattern, Tel: *Synchronous, Asynchronous and Causally Ordered Communication.* Distributed Computing, Vol. 9 No. 4, pp. 173 - 191, 1996

Blockaden bei synchroner Kommunikation

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

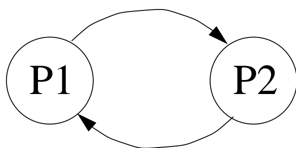
```
P2:
    send (...) to P1;
    receive...
    ...
```



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

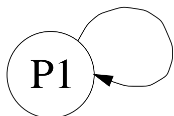
==> *Kommunikationsdeadlock!*

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



"Wait-for-Graph"

Genauso tödlich:

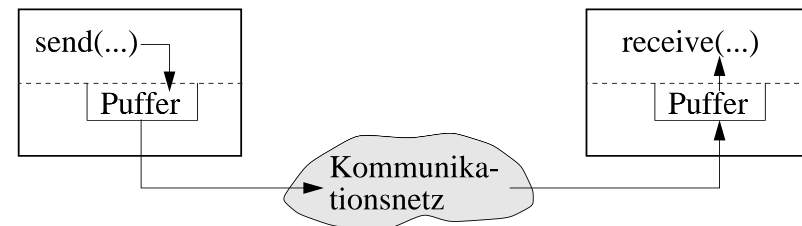



```
P1:
    send (...) to P1;
    receive...
    ...
```


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?)

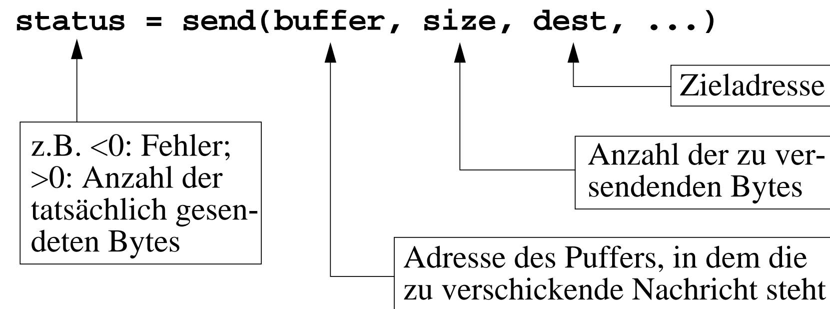
Sendeoperationen in der Praxis

- 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, ...)
```



- 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(...)
```

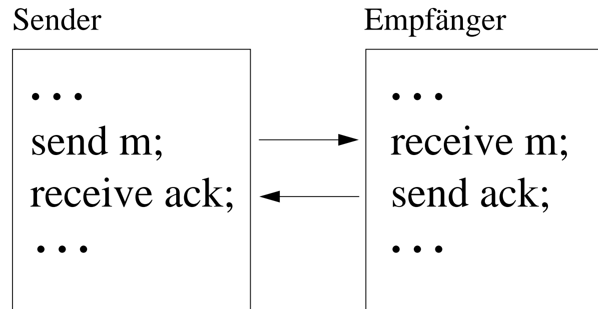
  - Dieser kann in Test- bzw. Warteoperationen verwendet werden
  - Z.B. Test, ob Send-Operation beendet: `msgdone(handle)`
  - Z.B. warten auf Beendigung der Send-Operation: `msgwait(handle)`

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

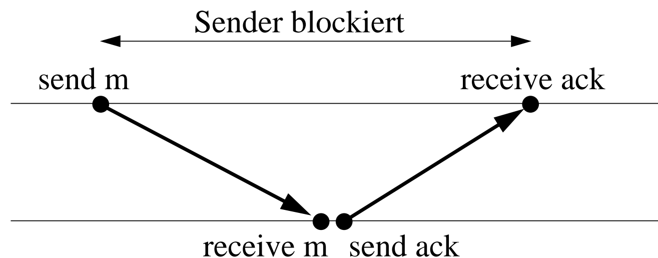
Denkübung: Man überlege, wie synchron+nicht-blockierend wirkt

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:

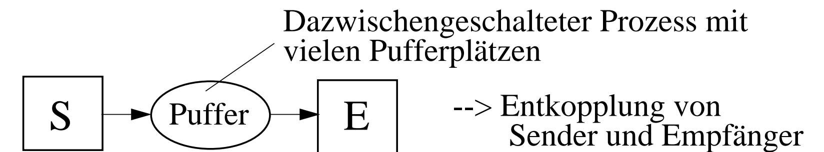


- Explizites Warten auf Acknowledgment im Sender direkt nach dem send (receive ist i.a. blockierend!)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive

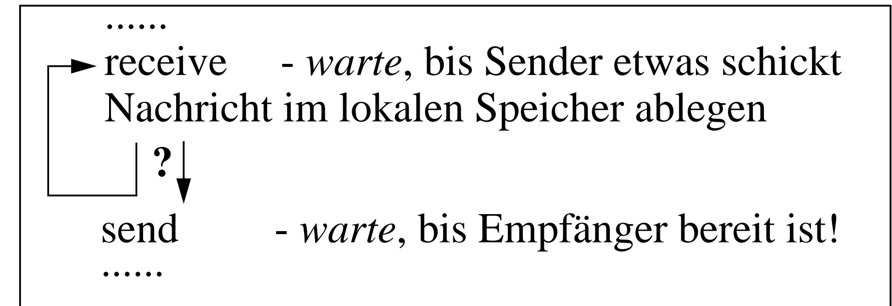


Asynchrone Kommunikation mittels synchroner Kommunikation

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?

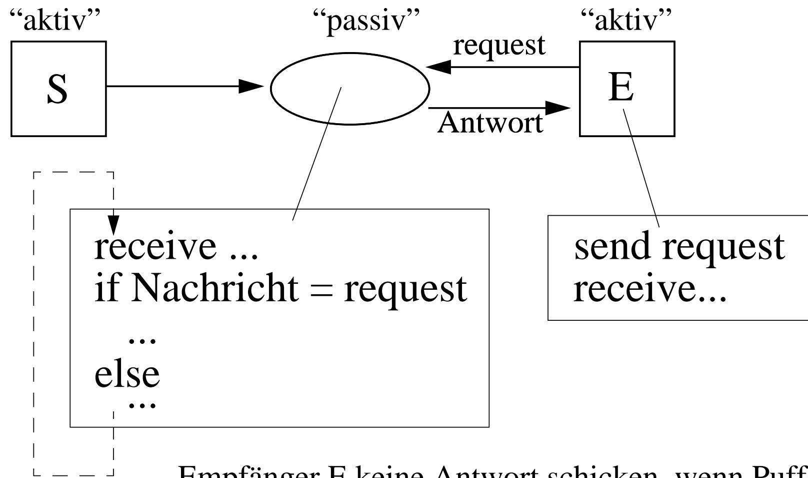
(1) wieder im receive auf den Sender warten, *oder*
(2) in einem send auf den Empfänger warten

--> 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!

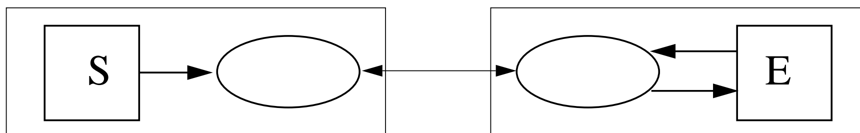


- Empfänger E keine Antwort schicken, wenn Puffer leer.
- Empfänger E wird nur dann verzögert, wenn Puffer leer.
- Für Sender S ändert sich nichts.
- Was tun, wenn der Puffer *voll* ist?

---

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 belegt

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

```
while true do
begin
    if Puffer ≠ voll then
    begin
        receive m from Sender;
        füge m in lokalen Pufferspeicher ein;
    end;  <--  oder: else begin
                          receive...
                          send ("NACK: voll!") to Sender;
                     end;
    if Puffer ≠ leer then
    begin
        receive request from Empfänger;
        füge m' aus lokalem Pufferspeicher aus;
        send m' to Empfänger;
    end;
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)

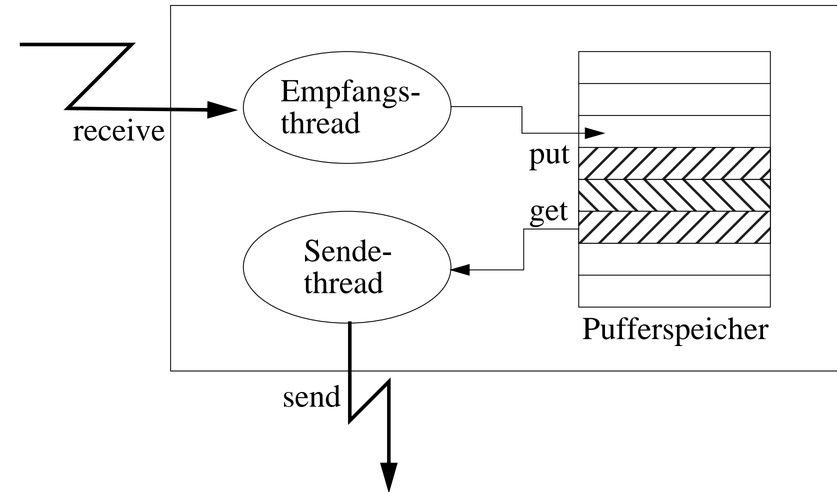
- Was geschieht, wenn mehrere guards "true" sind?
  ==> nichtdeterministische Auswahl, Wahl des ersten Falles...

- 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

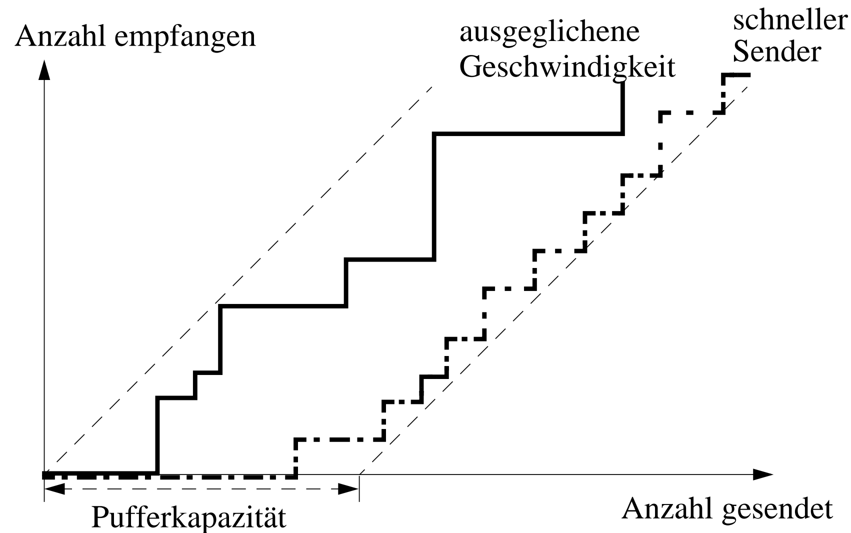
- Sende-thread ist (fast) immer sendebereit

- 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

"orthogonal" → *Synchronisationsgrad* / *Kommunikationsmuster*

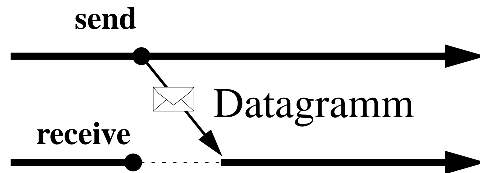
Kommunikationsmuster \ *Synchronisationsgrad*	asynchron	synchron
Mitteilung	*no-wait send* (Datagram)	*Rendezvous*
Auftrag	*Remote Service Invocation* (bzw. "asynchroner RPC")	*Remote Procedure Call* (RPC)

- Hiervon gibt es Varianten, Abarten...
  - bei verteilten objektorientierten Systemen z.B. "Remote Method Invocation" (RMI) statt RPC

- Weitere Klassifikation nach Adressierungsart möglich (Prozess, Port, Mailbox, Broadcast...)

Datagramm

- Asynchron-mitteilungsorientierte Kommunikation

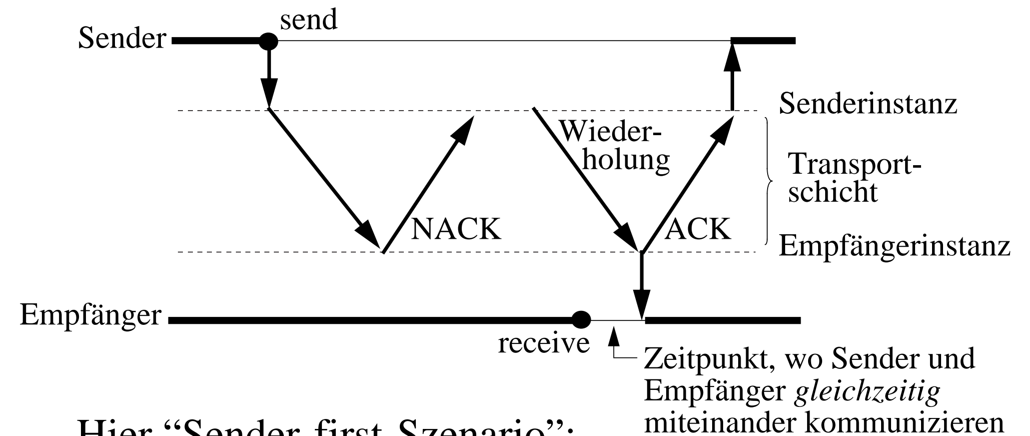


- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)

- 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

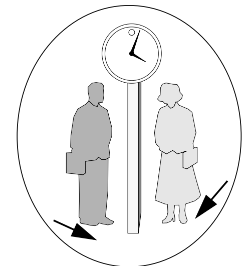
- Synchron-mitteilungsorientierte Kommunikation



- Hier “Sender-first-Szenario”: Sender wartet zuerst

- “Receiver-first-Szenario” analog

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



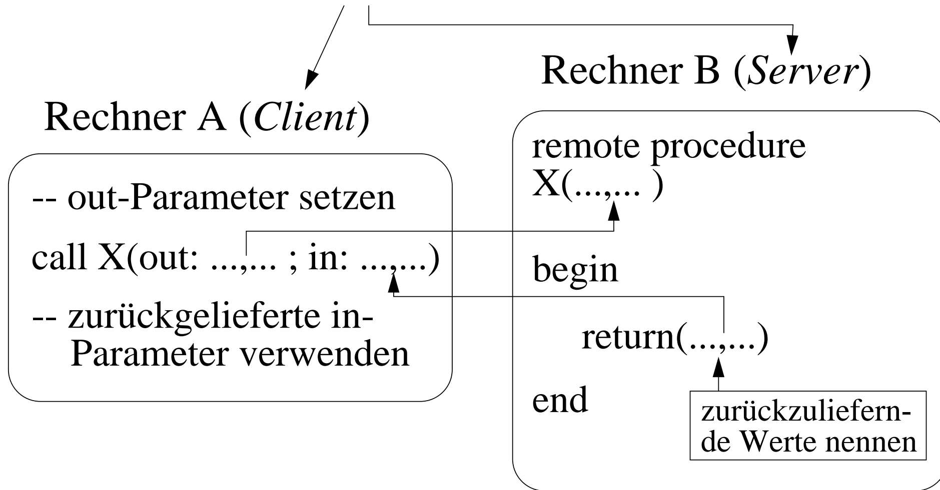
- Mit NACK / ACK ist keine Pufferverwaltung nötig! --> Aufwendiges Protokoll! (“Busy waiting”)
  - Alternative: Statt NACK, Nachricht auf Empfängerseite puffern
  - Alternative: Statt laufendem Wiederholungsversuch: Empfängerinstanz meldet sich bei Senderinstanz, sobald Empfänger bereit

- 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"

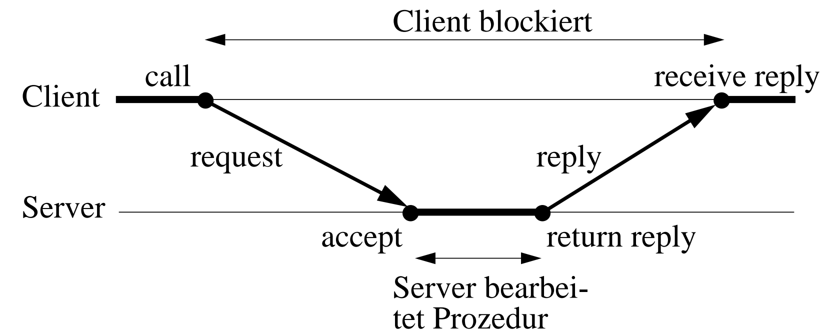
- Synchron; auftragsorientiertiertes Prinzip:

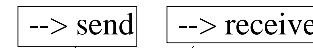


- Soll dem klassischen Prozeduraufruf möglichst gleichen
  - klare Semantik für den Anwender (Auftrag als „Unterprogramm")
  - einfaches Programmieren
    - kein Packen von Nachrichten, kein Quittieren...
    - Syntax analog zu bekanntem lokalen Prozeduraufruf
    - Kommunikation mit lokalen / entfernten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

- Implementierungsproblem: Verteilungstransparenz
  - Verteiltheit so gut wie möglich verbergen

RPC: Prinzipien



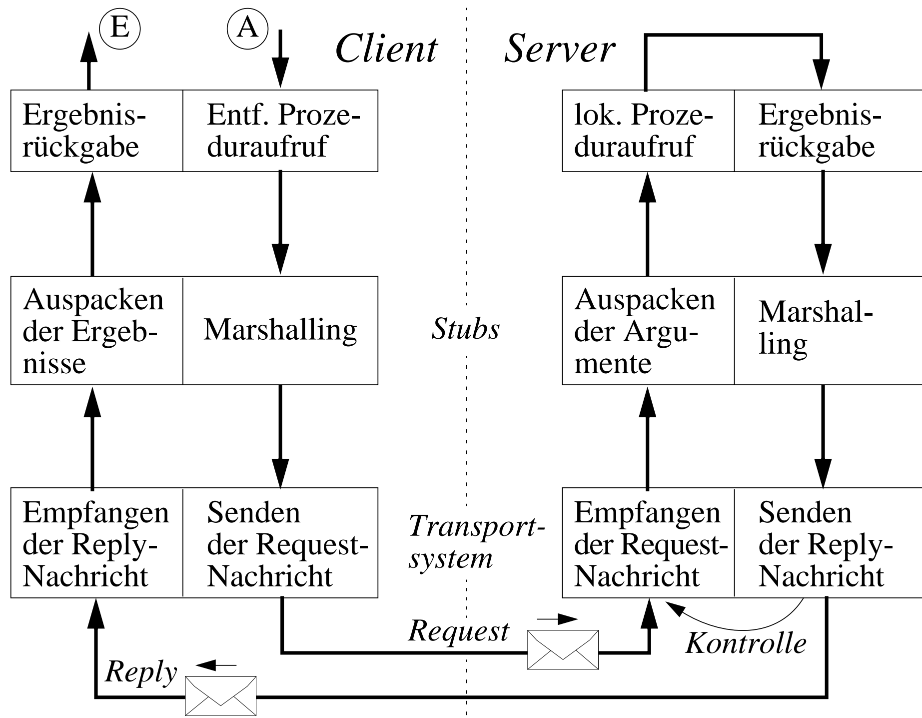


- call; accept; return; receive: interne Anweisungen
  - nicht sichtbar auf Sprachebene --> Compiler

- Call-by-value/result-Parameterübergabe

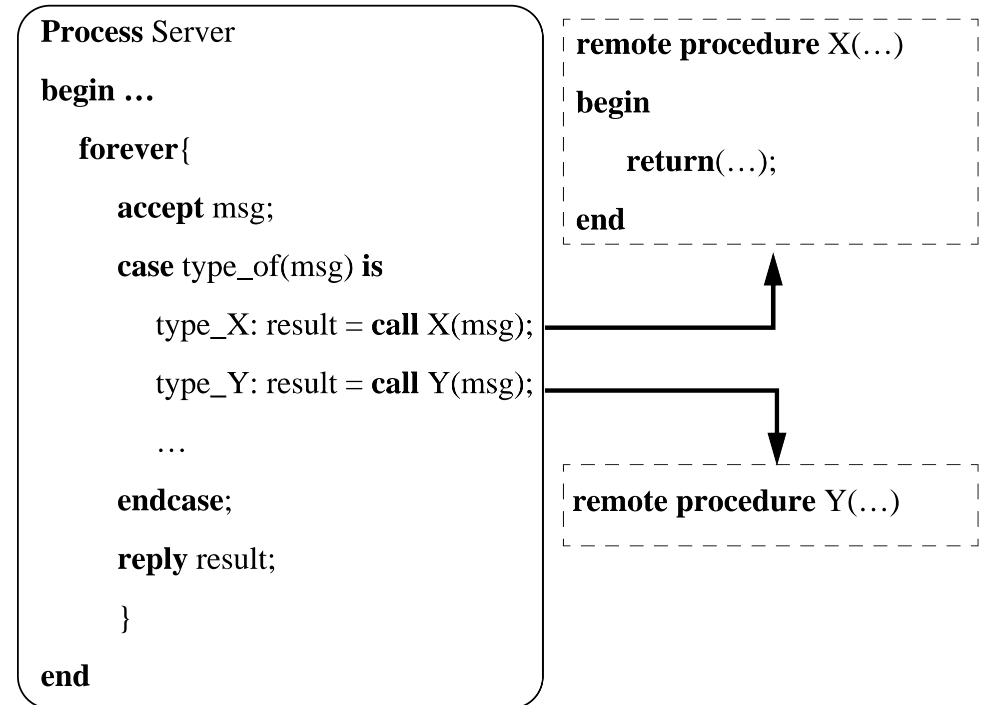
- Keine Parallelität zwischen Client und Server
  - RPC-Aufrufe sind blockierend

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
  - “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 (> 1 ms)
  - begrenzte Kommunikationsbandbreite bei umfangreichen Parametern
  - ungewisse, variable Verzögerungen

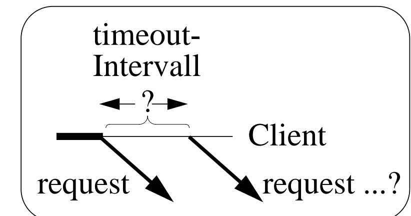
- Ortstransparenz
  - Standort des „zuständigen“ Servers bei Adressierung u.U. anzugeben
  - erkennbare Trennung der Adressräume von Client und Server
  - i.d.R. keine Pointer/Referenzparameter als Parameter möglich
  - auch keine Kommunikation über globalen Variablen möglich

- 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”

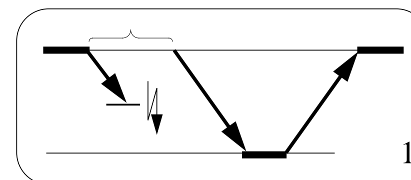
==> Fehlerproblematik ist also “kompliziert”!

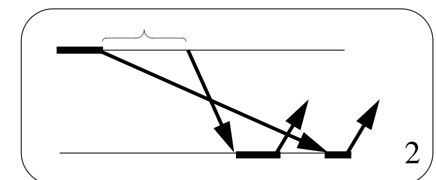
Typische Fehlerursachen: I. Verlorene Request-Nachricht

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

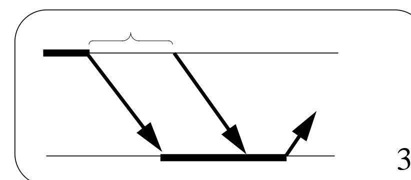


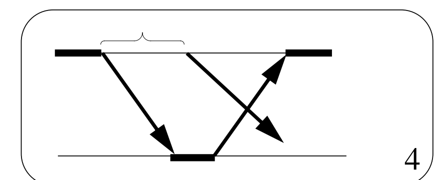
- *Probleme*:
  - Wieviele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
    - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten Operationen!)
    - Server sollte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



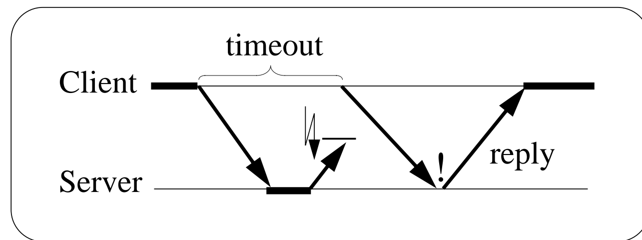


?





II. Verlorene Reply-Nachricht

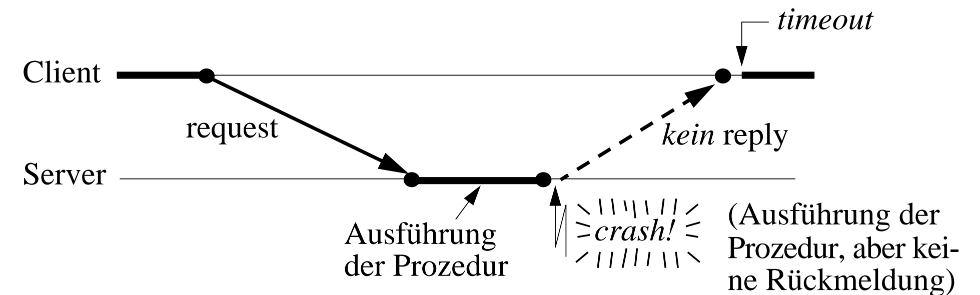


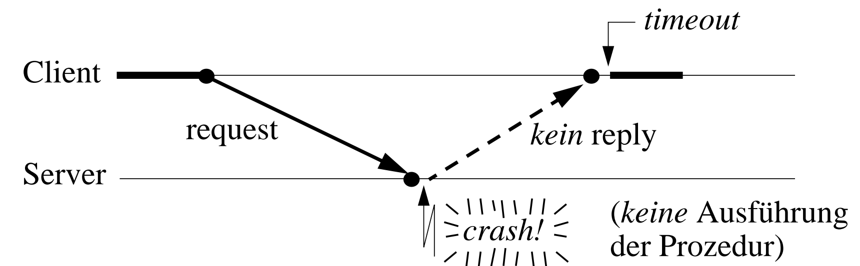
- *Gegenmassnahme 1:* analog zu verlorener Request-Nachricht
  - Also: Anfrage nach Ablauf des Timeouts wiederholen

- *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



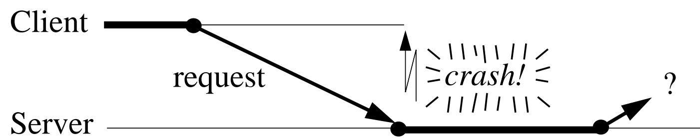


Probleme:

- 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!)

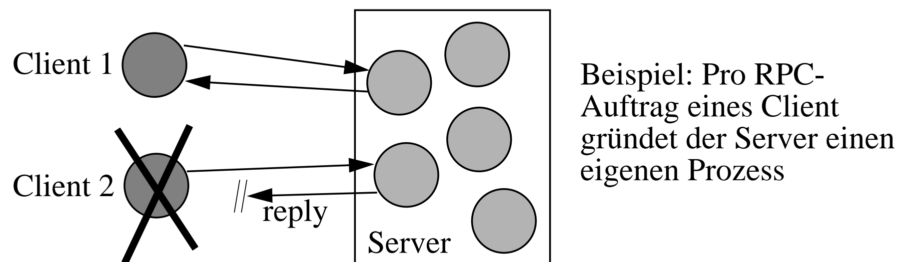
- Ggf. Probleme nach einem Server-Restart
  - z.B. “Locks”, die noch bestehen (Gegenmassnahmen?) bzw. allgemein: “verschmutzer” Zustand durch frühere Inkarnation
  - typischerweise ungenügend Information (“Server Amnesie”), um in alte Kommunikationszustände problemlos wieder einzusteigen

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 exisitiert)

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)

wird etwas euphemistisch oft als "best effort" bezeichnet

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

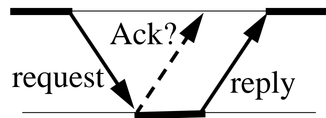
	Fehlerfreier Ablauf	**Nachrichten-verluste**	**Ausfall des Servers**
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-least-once	Ausführung: 1 Ergebnis: 1	Ausführung: ≥1 Ergebnis: ≥1	Ausführung: ≥0 Ergebnis: ≥0
At-most-once	Ausführung: 1 Ergebnis: 1	Ausführung: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

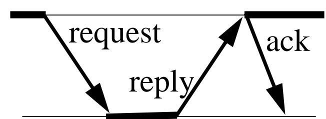
RPC-Protokolle

- *RR-Protokoll* ("Request-Reply"):



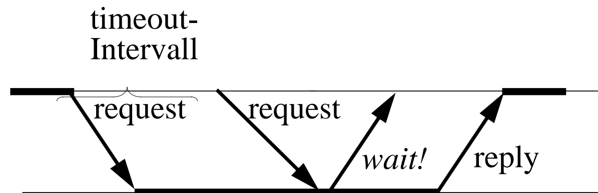
- Reply ist implizite Quittung für Request
- lohnt sich ggf. eine unmittelbare Bestätigung des Request?

- *RRA-Protokoll* ("Request-Reply-Acknowledge"):



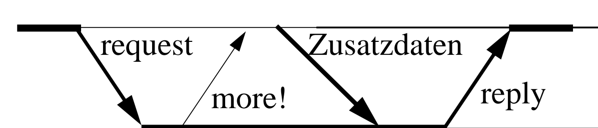
- "pessimistischer" als das RR-Protokoll
- Vorteil: Server kann ggf. gespeicherte Replies frühzeitig löschen (und natürlich Replies bei Ausbleiben des ack wiederholen)

- *Sinnvoll bei langen Aktionen / überlasteten Servern:*



"wait" = Bestätigung eines erkannten Duplikats

- *Parameter-Übertragung „on demand"*



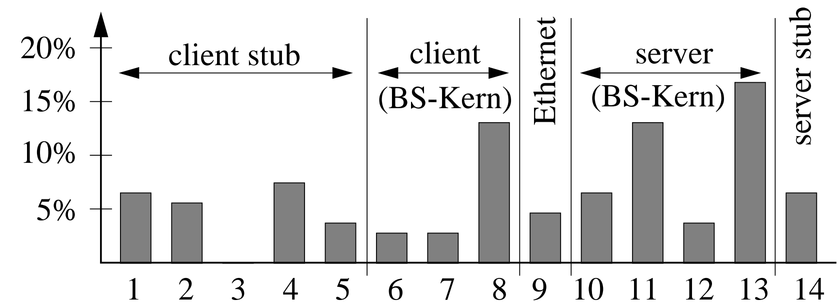
- spart Pufferkapazität
- bessere Flusssteuerung
- Zusatzdaten abhängig vom konkreten Ablauf

- *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

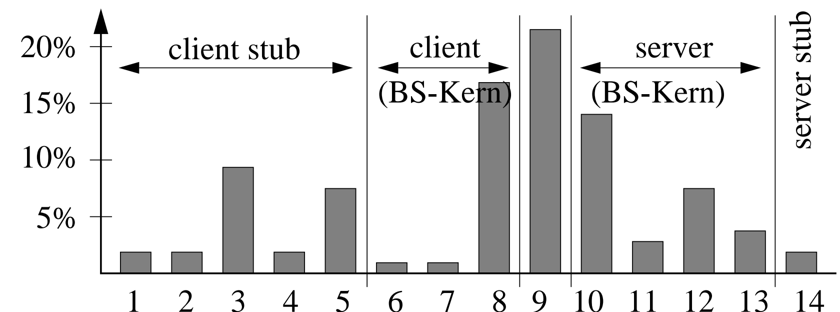
Analyse eines RPC-Protokolls durch Schroeder et al. ('90):
(zitiert nach A. Tanenbaum)

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

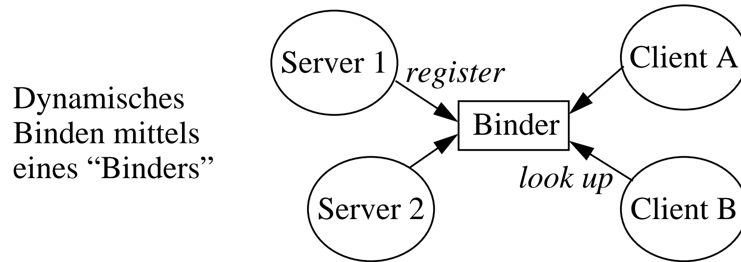
b) 1440 Byte Nutznachricht (ebenfalls kein Auftragsbearbeitung):



- 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

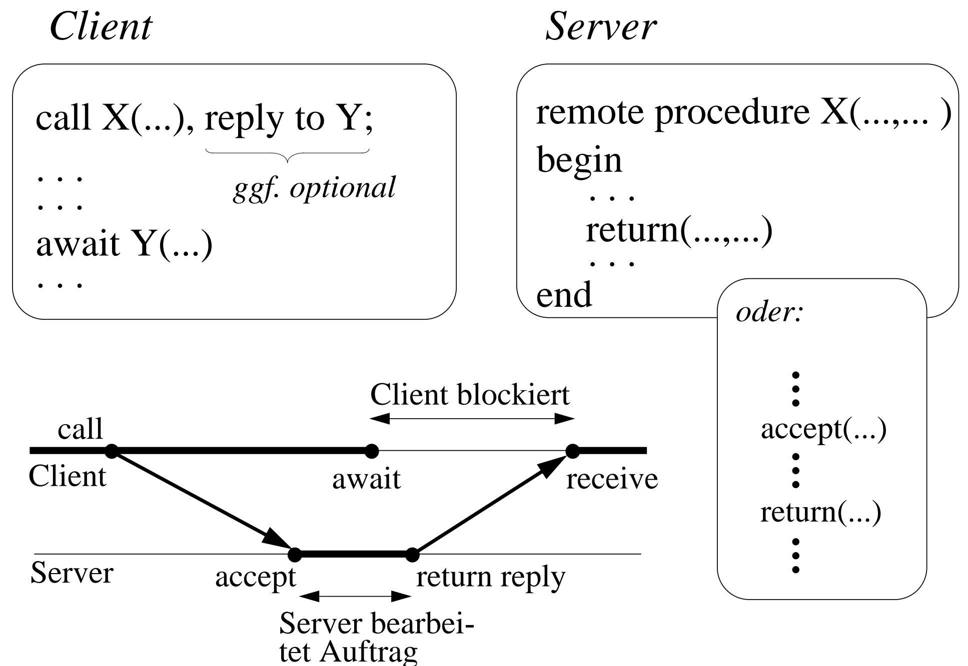
- Problem: Wie werden Client und Server "gematcht"?
  - Verschiedene Rechner und i.a. verschiedene Lebenszyklen --> kein gemeinsames Übersetzen / statisches Binden (fehlende gem. Umgebung)



- 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 die Adresse eines geeigneten Servers beim Binder
  - "*look up*"; "importieren" der RPC-Schnittstelle
- *Vorteile*: Binder kann (im Prinzip): ← --> "Trader" bzw. "Broker"
  - 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
- *Probleme*:
  - zentraler Binder ist ein potentieller Engpass (Binding-Service geeignet verteilen? Konsistenz!)
  - dynamisches Binden kostet Ausführungszeit

Remote Service Invocation

- "asynchroner RPC"
- 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

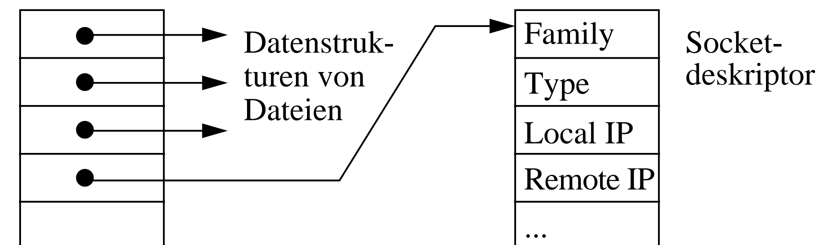
- Zu TCP (bzw. UDP) gibt es keine festgelegten “APIs”
- Bei UNIX ist dafür entstanden: “sockets” als Zugangspunkte zum Transportsystem
  - etwas modernere Alternative: TLI (Transport Layer Interface)
- Semantik eines sockets: analog zu Datei-Ein/Ausgabe
  - ist insbesondere bidirektional (“schreiben” 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);
```

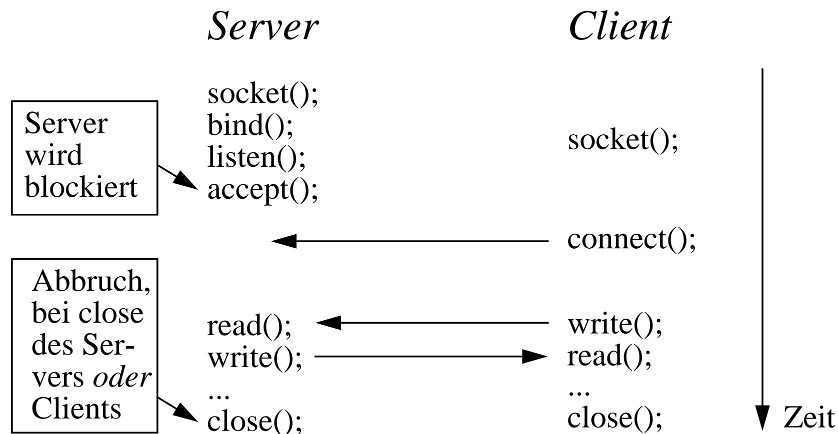
“Family”: Internet oder nur lokale Domäne

“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;
```


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