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:

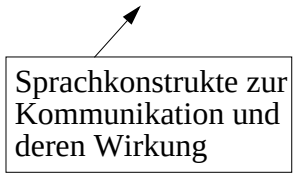
  1) dazwischenliegendes *physikalisches Medium*
     - z.B. elektrische Signale in Kupferkabeln

  2) einheitliche *Verhaltensregeln*
     - Kommunikationsprotokolle

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

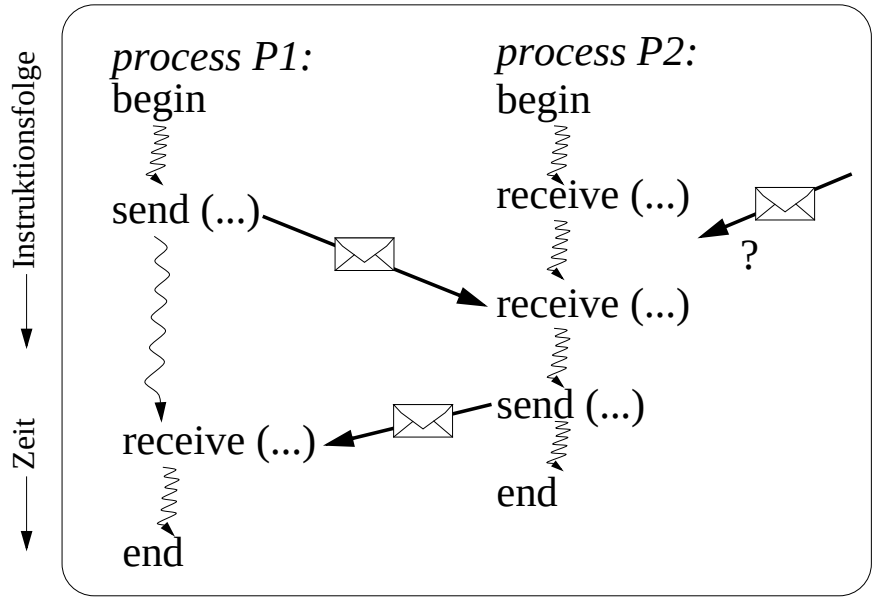
Also trotz Verteiltheit gewisse gemeinsame “Dinge”!

- 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:



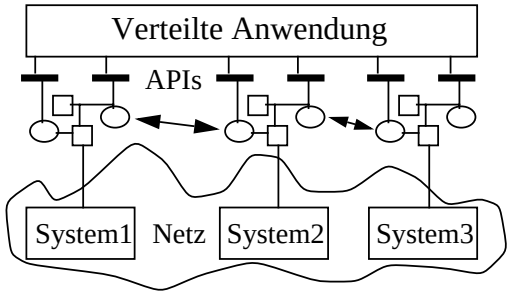
- Welche Kommunikationsanweisungen "matchen"?
- Empfangsbereitschaft, aber keine Nachricht? (verlorene Nachricht?)
- Nachricht, aber keine Empfangsbereitschaft?
- Sprachliche Ausprägung der Kommunikation?
- Wie wird adressiert?

==> 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.)

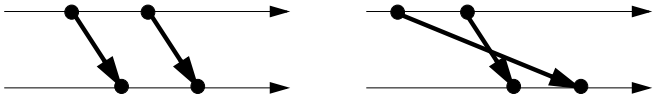
Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!

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

Ordnungserhalt

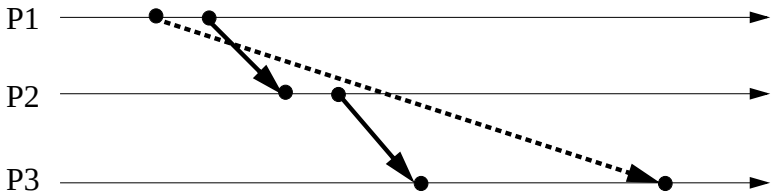
- Oft werden vom Kommunikationssystem keine Garantien bzgl. Nachrichtenreihenfolgen gegeben

- 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

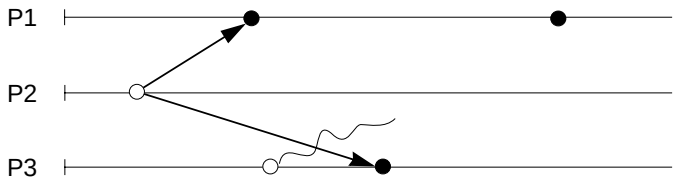
- Möchte man auch dies haben, so muss die Kommunikation "kausal geordnet" sein (Anwendungszweck?)
  - entspricht einer "Globalisierung" von FIFO auf mehrere Prozesse
  - "Dreiecksungleichung": Keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege
  - Denkübung: wie garantiert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

Fehlermodelle (1)

- Fehler sind leider eine Quelle vielfältiger Ärgernisse in verteilten Systemen

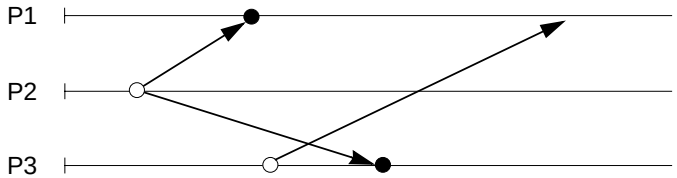
- Klassifikation von Fehlermöglichkeiten; Abstraktion von den konkreten Ursachen

- **Fehlerhaftes Senden**



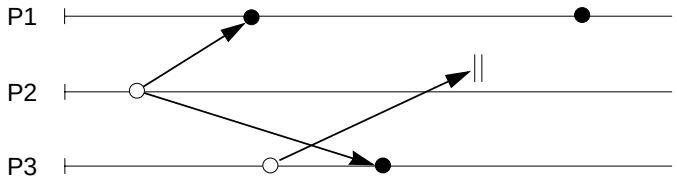
Empfänger merkt nichts; *Sender* stellt dies ggf. fest

- **Fehlerhaftes Empfangen**



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

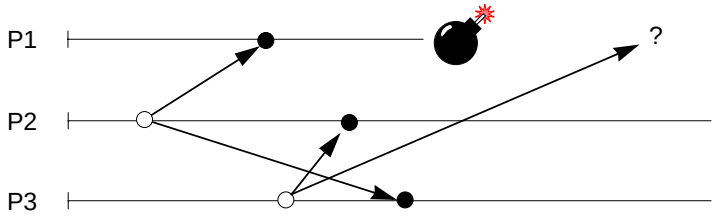
- **Fehlerhaftes Übertragen**



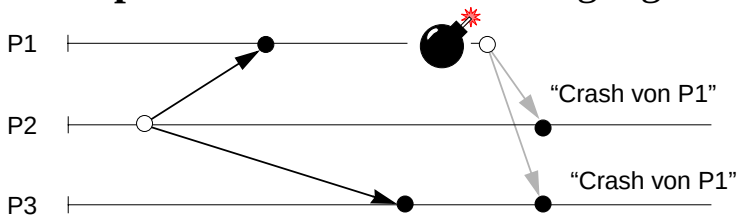
Weder Sender noch Empfänger merken unmittelbar etwas

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

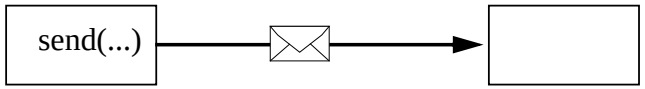
(derartige Fehler lassen sich höchstens bis zu einem gewissen Grad durch *Redundanz* erkennen)

Fehlertolerante Algorithmen sollen das “richtige” Fehlermodell berücksichtigen!

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

Kommunikationsmuster

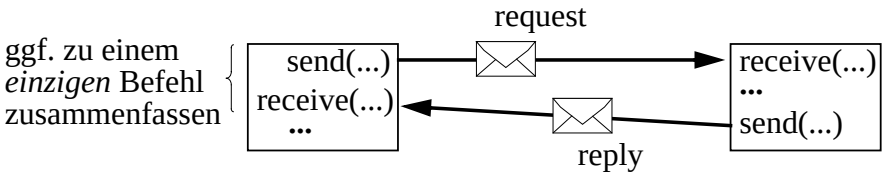
Mitteilungsortientiert:



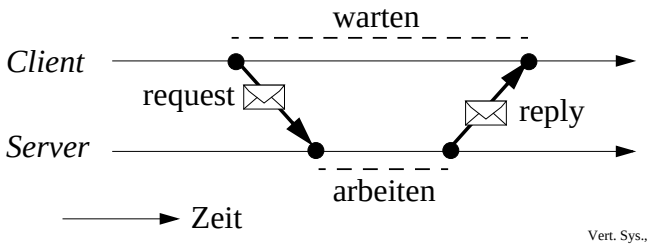
- Unidirektional
- Übermittelte Werte werden der Nachricht in Form von “Ausgabeparametern” beim send übergeben

---

Auftragsortientiert:



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

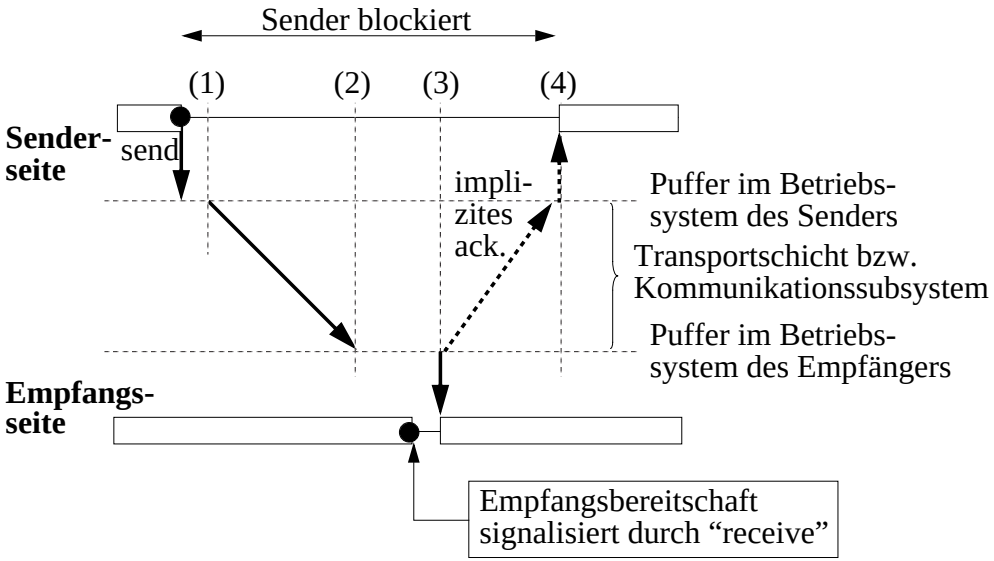


Synchrone Kommunikation

- *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?

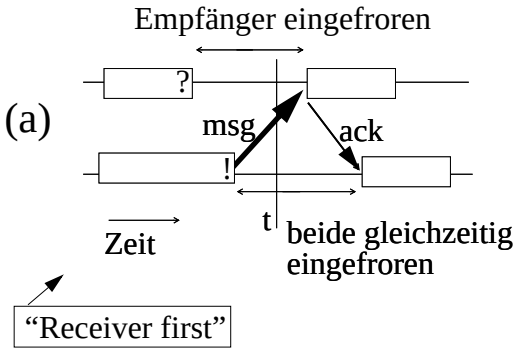
- *Syn-chron* = "gleich"-"zeitig"

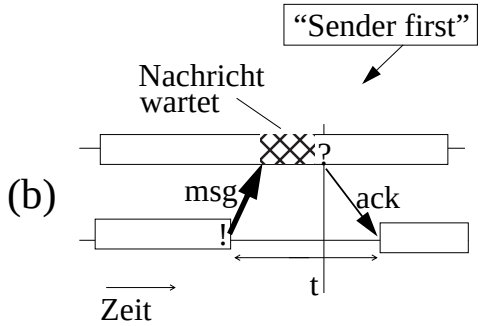
- Idealisierung: Send und Receive geschehen *gleichzeitig*

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

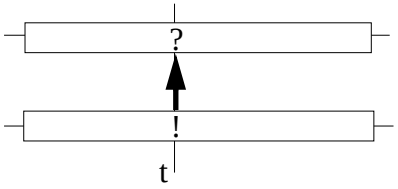
- Bem.: "Receive" ist i.a. blockierend (d.h. Empfänger wartet so lange, bis Nachricht angekommen)

Implementierung:





Idealisierung: senkrechte Pfeile in den Zeitdiagrammen



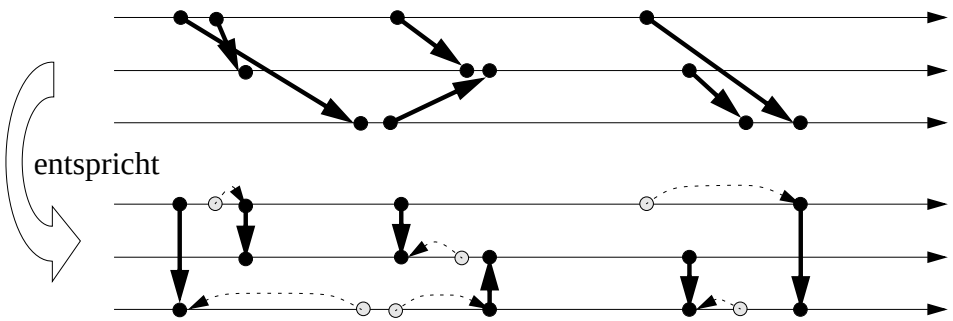
Als wäre die Nachricht zum Zeitpunkt t versendet und empfangen worden!

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

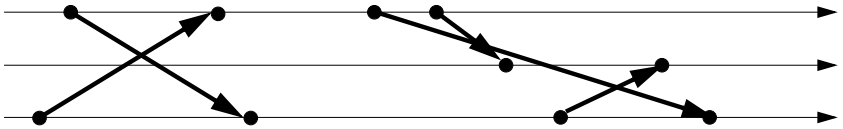
Virtuelle Gleichzeitigkeit

- 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?)



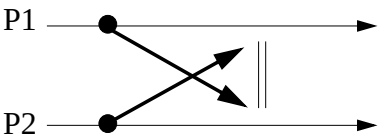
  - aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

(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...
    ...
```

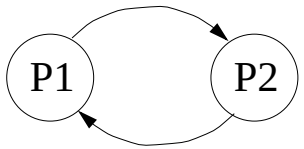
```
P2:
    send (...) to P1;
    receive...
    ...
```



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

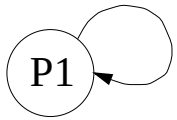
==> *Kommunikationsdeadlock!*

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



“Wait-for-Graph”

Genauso tödlich:

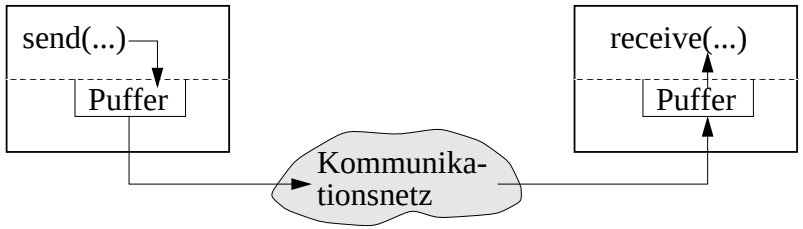



```
P1:
    send (...) to P1;
    receive...
    ...
```

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. 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)

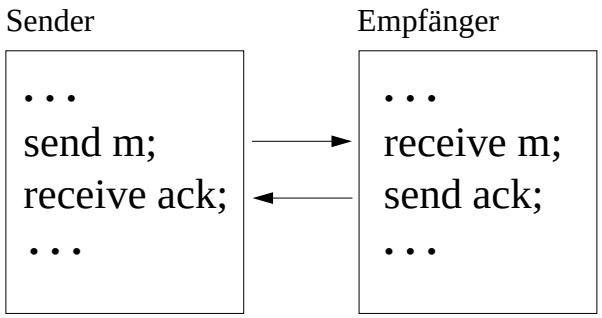
- Nicht-blockierende Operationen liefern i.a. einen “handle”

  `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)`

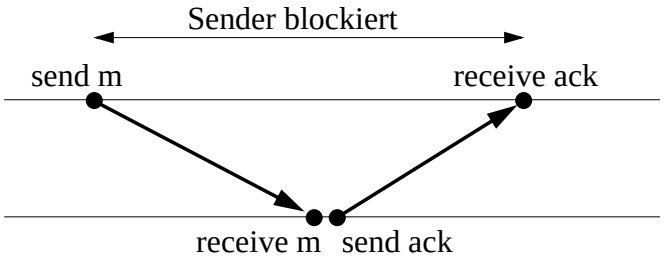
- Nicht-blockierend ist effizienter aber u.U. unsicherer und umständlicher (ggf. Test, warten) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:

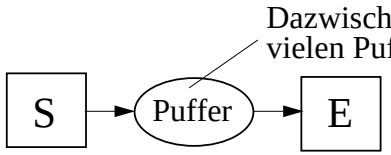


- Explizites Warten auf Acknowledgment im Sender direkt nach dem send (receive ist i.a. blockierend!)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive



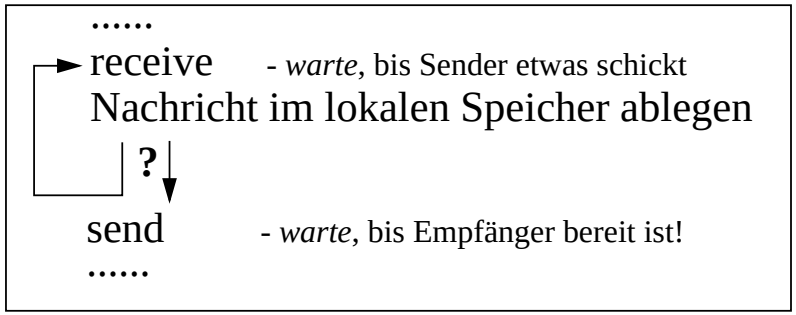
Asynchrone Kommunikation mittels synchroner Kommunikation

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



--> Entkopplung von Sender und Empfänger

Wie realisiert man einen Pufferprozess?



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

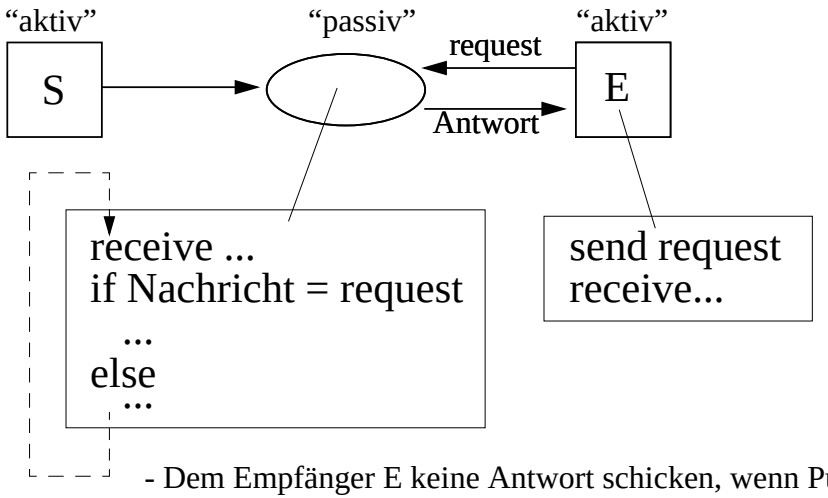
(1) wieder im receive auf den Sender warten, *oder*
(2) in einem send auf den Empfänger warten

--> 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*!)

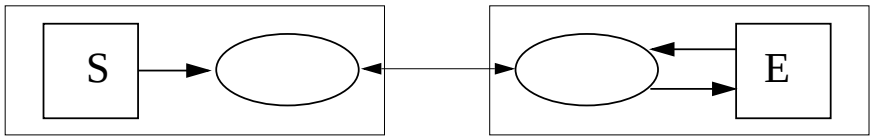


- Dem Empfänger E keine Antwort schicken, wenn Puffer leer.
- Empfänger E wird nur dann verzögert, wenn Puffer leer.
- Für Sender S ändert sich nichts.
- Was tun, wenn der Puffer *voll* ist?

---

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)

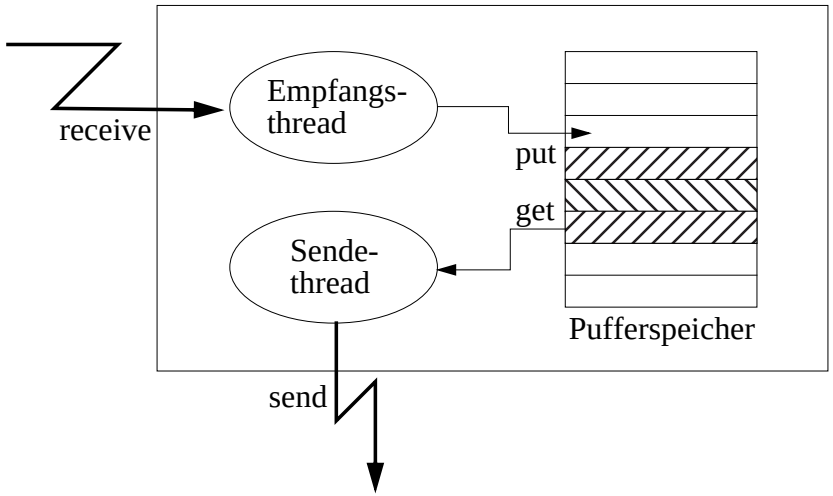
- Was geschieht, wenn mehrere guards "true" sind?
  ==> nichtdeterministische Auswahl, Wahl des ersten Falles...

- Wann genau sollten die guards evaluiert werden?

- Denkübung: Wie implementiert man select-Anweisungen?

Puffer bei Multi-thread-Objekten

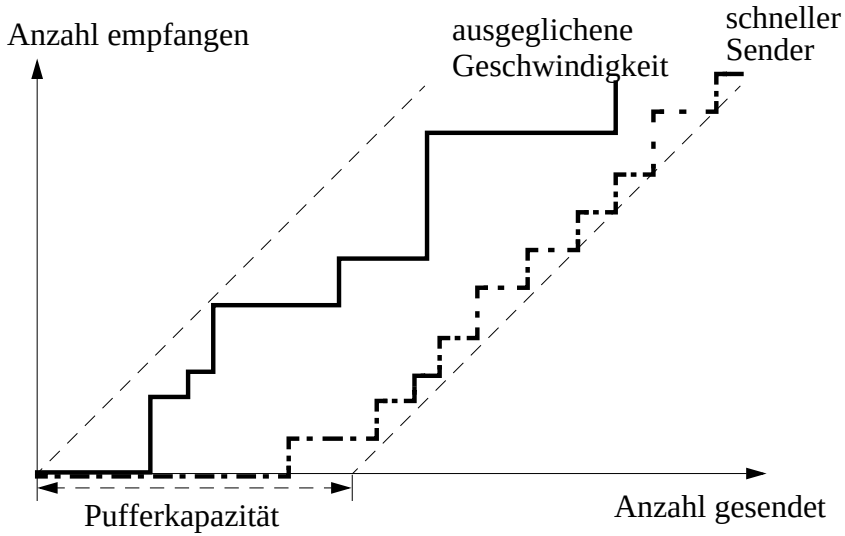
Beachte: Threads (Leichtgewichtsprozesse) greifen auf
gemeinsamen Speicher zu.



- 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!

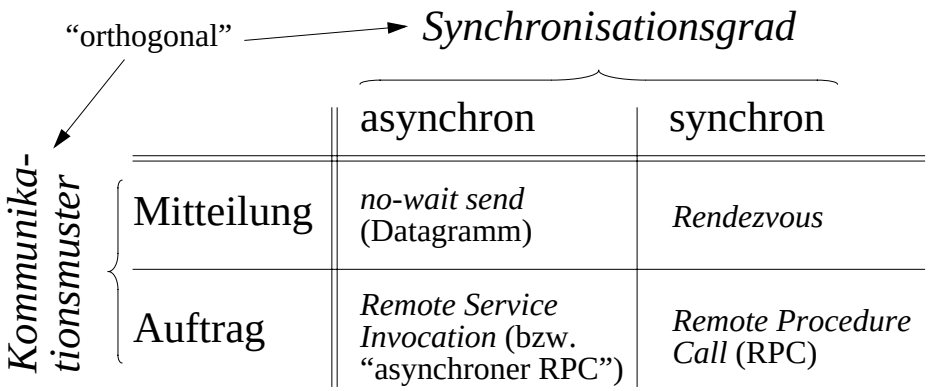
Puffer

- 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!

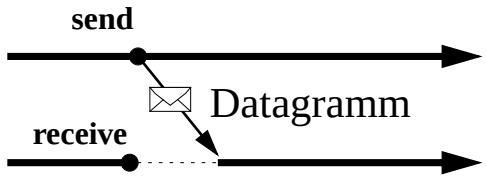
Klassifikation von Kommunikationsmechanismen



- 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

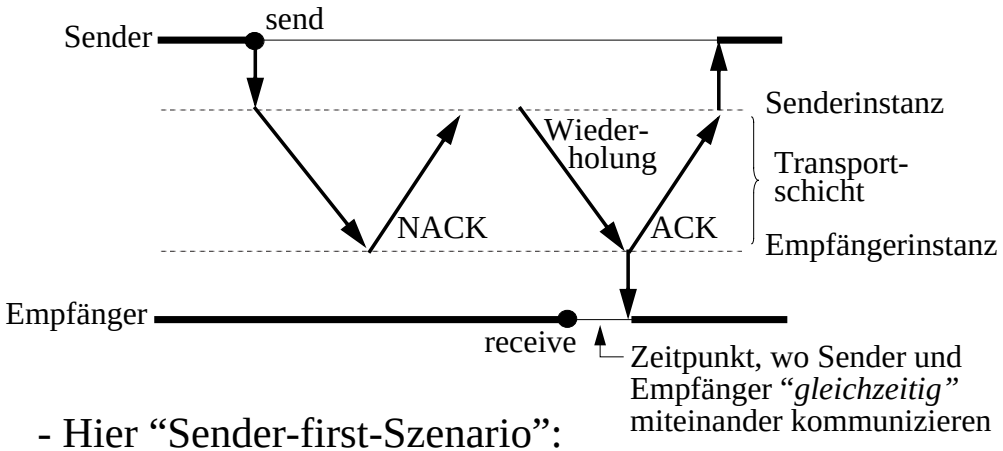
- Asynchron-mitteilungsorientierte Kommunikation



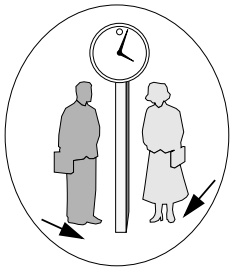
- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- Nachteil
  - keine Erfolgsgarantie für den Sender
  - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicher-verwaltung …) im Unterschied etwa zur syn. Kommunikation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

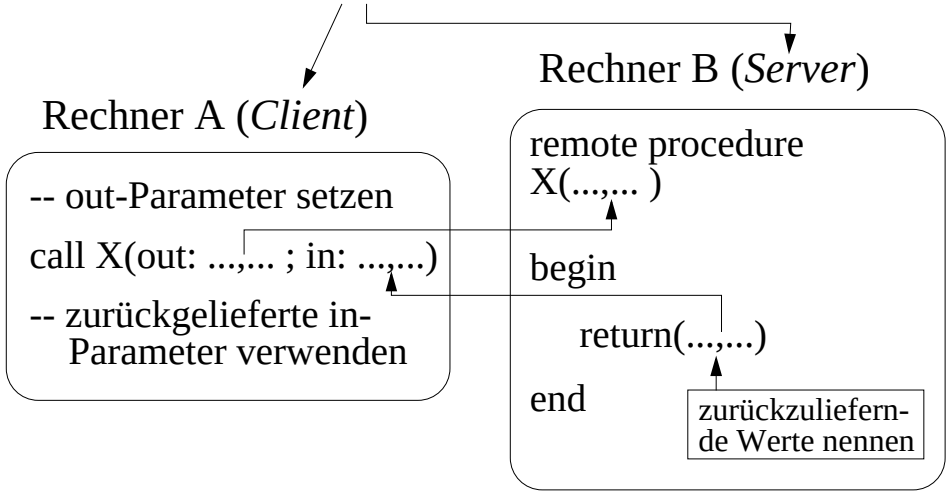
- Mit NACK / ACK ist keine Pufferverwaltung nötig!
  --> Aufwendiges Protokoll! ("Busy waiting")
  - Alternative: Statt NACK: Nachricht auf Empfängerseite puffern
  - Alternative: Statt laufendem Wiederholungsversuch: Empfänger-instanz meldet sich bei Senderinstanz, sobald Empfänger bereit

- 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"

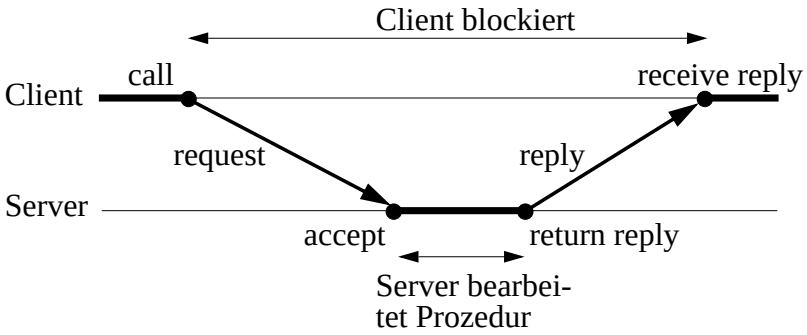
- Synchron-auftragsorientiertiertes Prinzip:



- 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

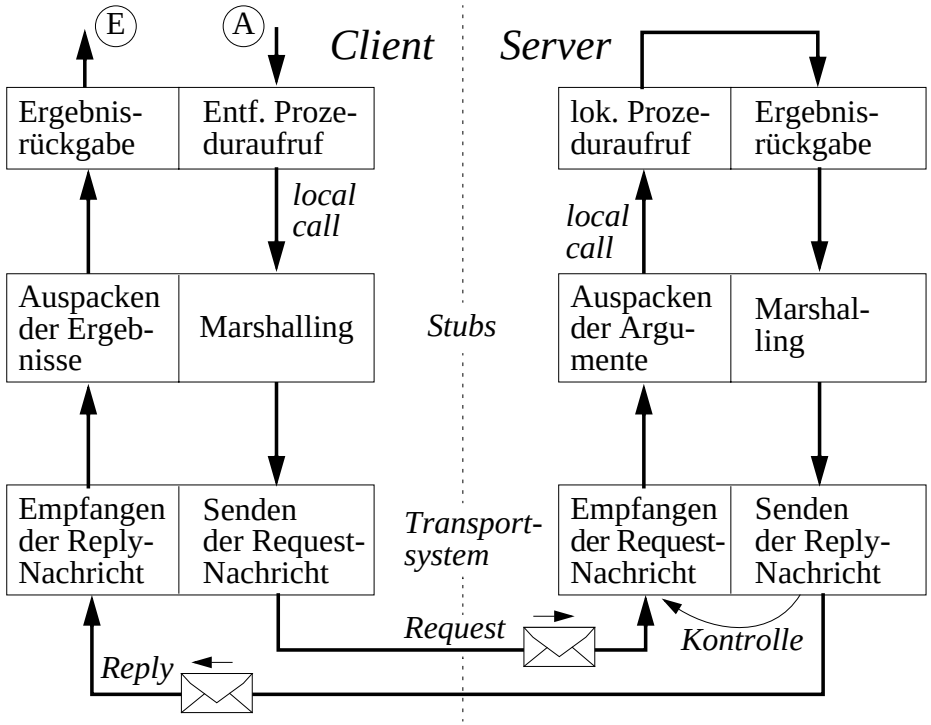
RPC: Prinzipien



--> send --> receive

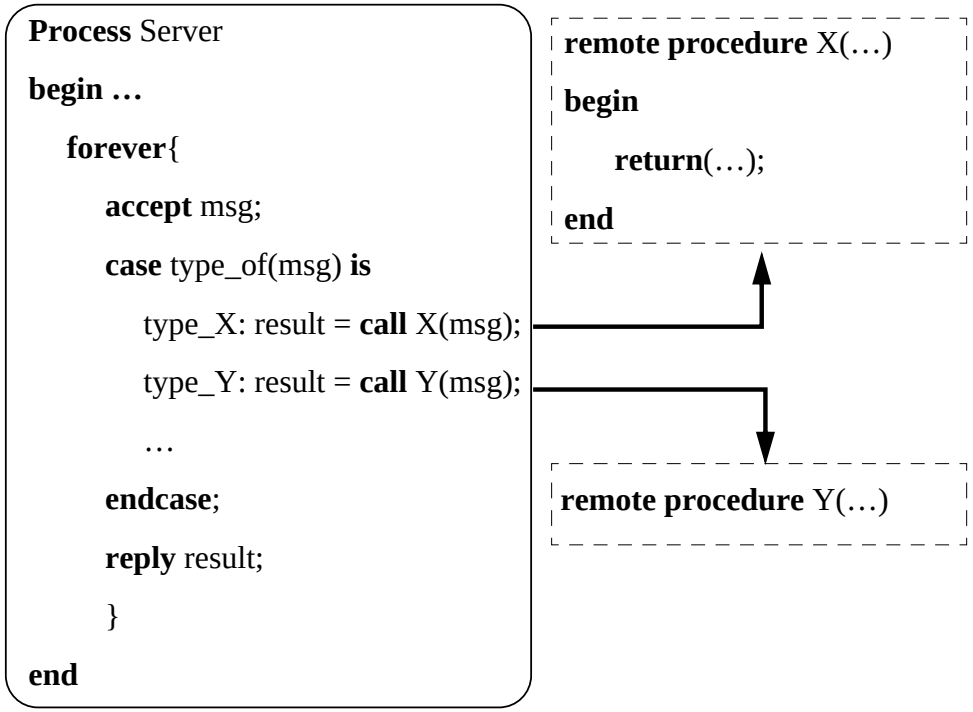
- call; accept; return; receive: interne Anweisungen
  - nicht sichtbar auf Sprachebene --> Compiler
- Call-by-value/result-Parameterübergabe
- Keine Parallelität zwischen Client und Server
  - RPC-Aufrufe sind blockierend

RPC: Implementierung



RPC: Server-Kontrollzyklus

- Warten auf Request, Verzweigen zur richtigen Prozedur:



"Dispatcher"

RPC-Stubs

- *Stub* = Stummel, Stumpf

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

Client:

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

- 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* ("proxy") des entfernten Gegenübers
  - Client-Stub / Server-Stub
  - simulieren einen lokalen Aufruf
  - sorgen für Packen und Entpacken von Nachrichten
  - konvertieren Datenrepräsentationen bei heterogenen Umgebungen
  - steuern das Übertragungsprotokoll (z.B. zur fehlerfreien Übertragung)
  - bestimmen 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
  - Bandbreite bei umfangreichen Parametern beachten
  - ungewisse, variable Verzögerungen

- Ortstransparenz
  - Standort des Servers bei Adressierung u.U. anzugeben
  - erkennbare Trennung der Adressräume von Client und Server
  - i.a. 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”!