Kommunikation

Kommunikation

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

- Notwendig, damit die Kommunikation klappt ist jedenfalls:

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

2) einheitliche *Verhaltensregeln*
   - Kommunikationsprotokolle

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

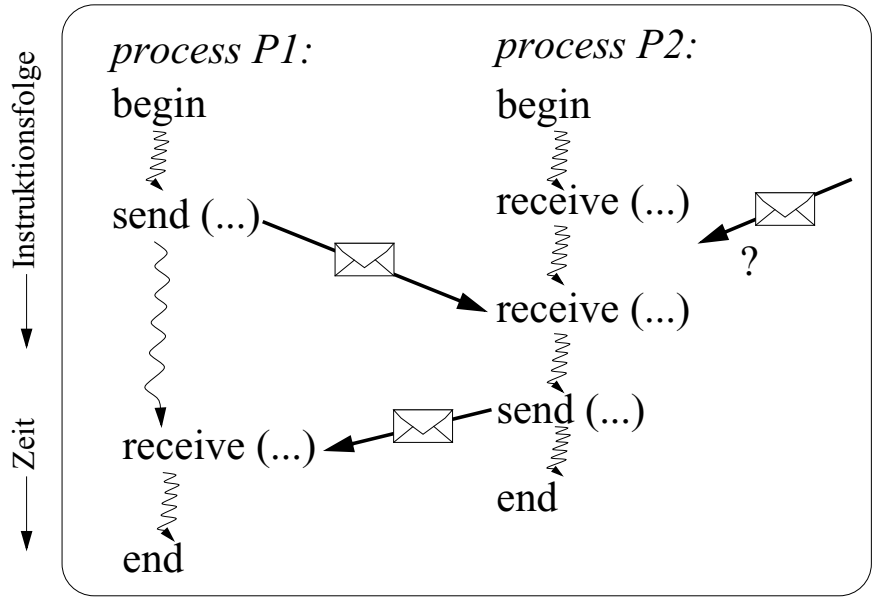
Also trotz Verteiltheit gewisse gemeinsame Aspekte!

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

Sprachkonstrukte zur Kommunikation und deren Wirkung

Nachrichtenbasierte Kommunikation

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



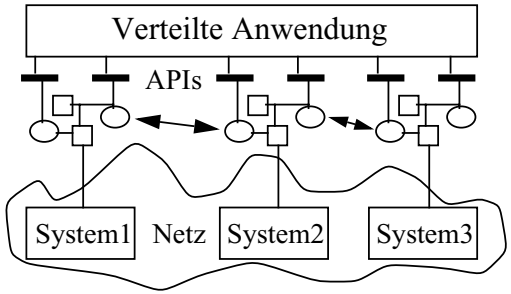
- Welche Kommunikationsanweisungen "matchen"?
- Empfangsbereitschaft, aber keine Nachricht?
- Nachricht, aber keine Empfangsbereitschaft?
- Wie wird adressiert?

⇒ Vielfältige Aspekte und Probleme

Message Passing System

als einfache Form von "Middleware"

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



- Besteht aus Hilfsprozessen, Pufferobjekten...
- Verbirgt Details des zugrundeliegenden Netzes
- Verwendet vorhandene Protokolle und implementiert damit neue, "höhere" Protokolle
- Garantiert (je nach "Semantik") gewisse Eigenschaften
  - z.B. Reihenfolgeerhalt
- Abstrahiert von Implementierungsdetails
  - wie z.B. Puffer, Geräteadressen etc.
- Maskiert gewisse Fehler
  - z.B. durch automatische Wiederholung nach einem timeout
- Verbirgt Heterogenität unterschiedlicher Rechner- bzw. Betriebssystemplattformen
  - erleichtert 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*
  - feste Paketlänge
  - variabel aber begrenzt
  - (theoretisch) unbegrenzt

  } dann muss man mit solchen Einschränkungen leben und darum herumprogrammieren

- *Zuverlässigkeitsgrad*: Nachrichtenverlust
  - nicht bemerkt
  - vermutet und gemeldet
  - automatisch korrigiert

  qualitatives Merkmal

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

(*Typische 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?

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

- Mögliche *Anwendungen*:
  - Unterbrechen laufender Aktionen (→ Interrupt)
  - Aufbrechen von Blockaden
  - Out-of-band-Signalisierung

  } Durchbrechung der FIFO-Reihenfolge!

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

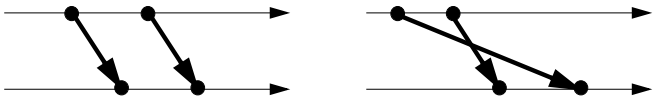
Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!

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

Ordnungserhalt

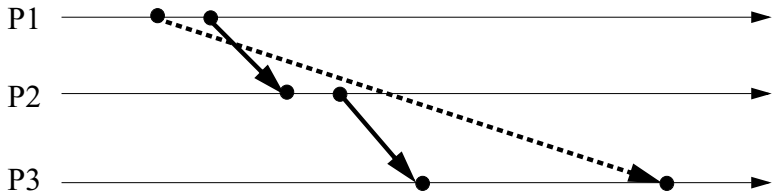
- Manchmal werden vom Kommunikationssystem Garantien bzgl. Nachrichtenreihenfolgen gegeben

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



FIFO kein FIFO

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



zwar FIFO, aber nicht kausal geordnet

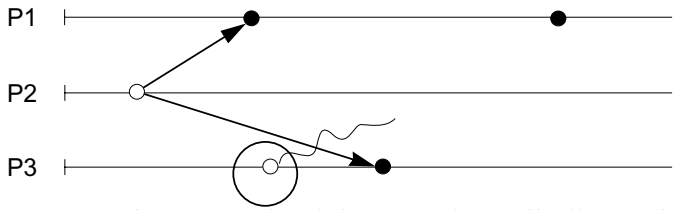
- Möchte man auch dies haben, so muss die Kommunikation "kausal geordnet" sein (Anwendungszweck?)
  - "Dreiecksungleichung": Keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege
  - entspricht einer "Globalisierung" von FIFO auf mehrere Prozesse
  - Denkübung: wie garantiert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

Fehlermodelle (1)

- Fehler sind leider eine Quelle vielfältiger Ärgernisse in verteilten Systemen

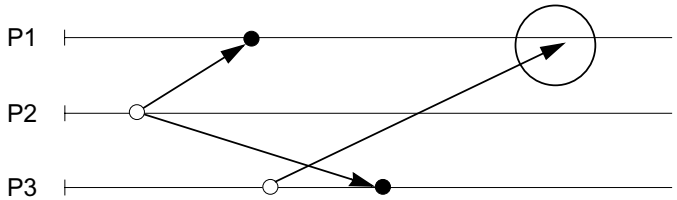
- Klassifikation von Fehlermöglichkeiten; Abstraktion von den konkreten, spezifischen Ursachen

• **Fehlerhaftes Senden**



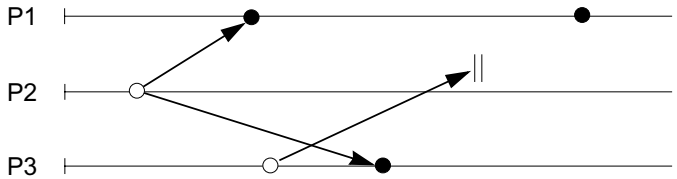
Empfänger merkt nichts; *Sender* stellt dies evtl. fest

• **Fehlerhaftes Empfangen**



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

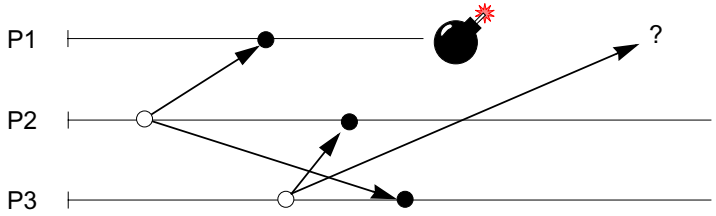
• **Fehlerhaftes Übertragen**



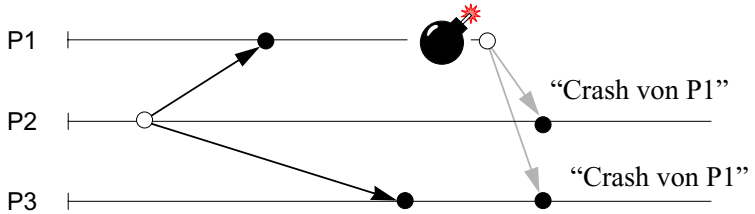
Weder Sender noch Empfänger merken unmittelbar etwas

Fehlermodelle (2)

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



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



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

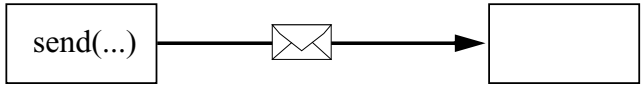
(solche Fehler lassen sich nur teilweise, z.B. durch *Redundanz,* erkennen)

Fehlertolerante Algorithmen sollen das “richtige” Fehlermodell berücksichtigen!

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

Kommunikationsmuster

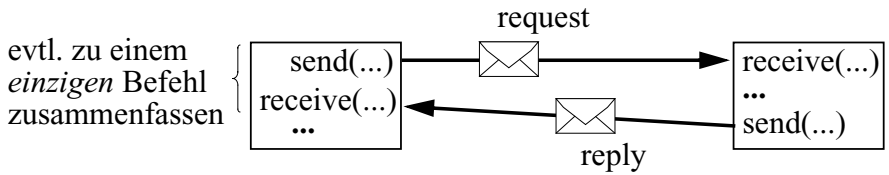
Mitteilungsortientiert:



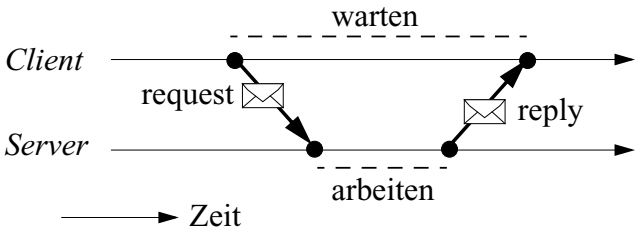
- Unidirektional
- Übermittelte Werte werden der Nachricht typw. als “Ausgabeparameter” beim send übergeben

---

Auftragsortientiert:



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

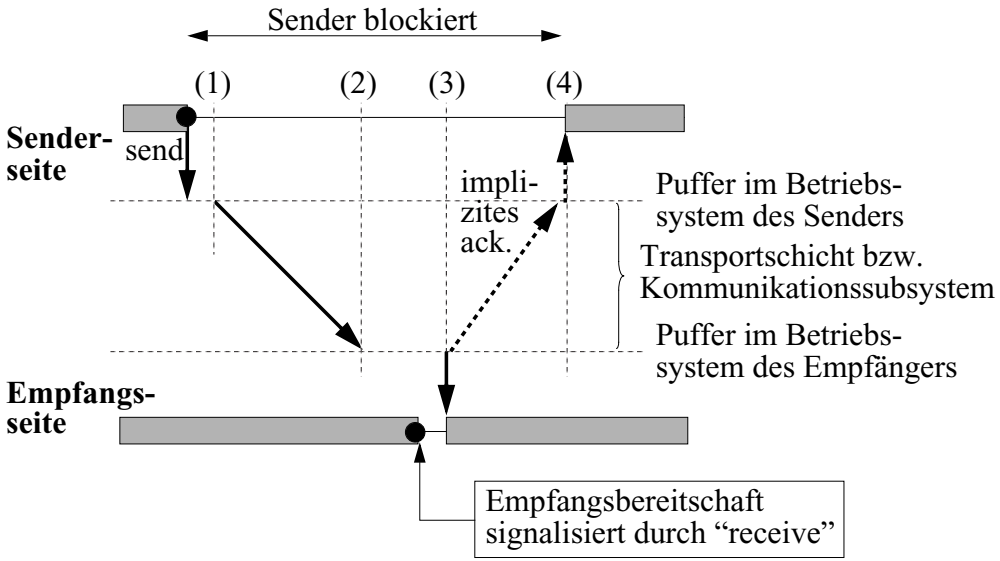


Blockierendes Senden

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

  was genau ist das?

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



- Verschiedene Ansichten der "korrekten" Definition von "Abschluss der Transaktion" aus Sendersicht:

  - *Zeitpunkt 4* (automatische Bestätigung, dass der Empfänger das receive ausgeführt hat) ist die höhere, anwendungsorientierte Sicht.

  - Falls eine Bestätigung bereits zum *Zeitpunkt 2* geschickt wird, weiss der Sender nur, dass die Nachricht am Zielort zur Verfügung steht und der Sendepuffer wieder frei ist. Vorher sollte der Sendepuffer nicht überschrieben werden, wenn die Nachricht bei fehlerhafter Übertragung evtl. wiederholt werden muss. (Oft verwendet bei systemorientierten Betrachtungen.)

Synchrone Kommunikation

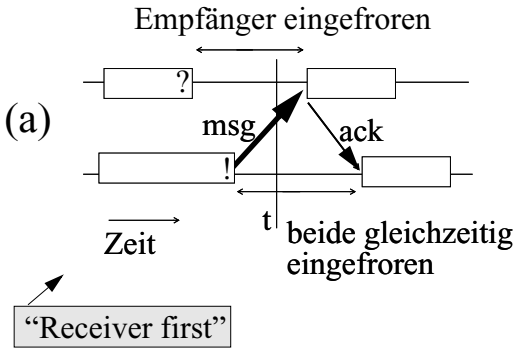
- *Syn-chron* = "gleich"-"zeitig"

- Idealisierung: Send und receive geschehen *gleichzeitig*

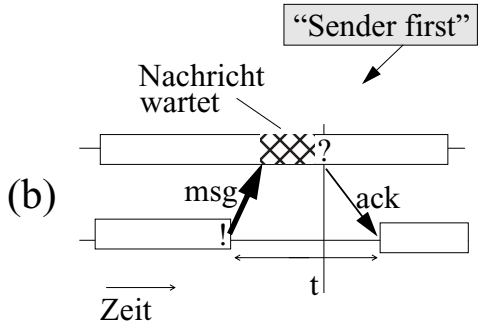
- Wodurch ist diese Idealisierung gerechtfertigt?
  (Kann man auch mit einer Marssonde synchron kommunizieren?)

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

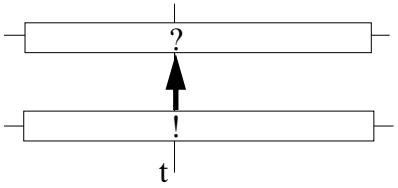
Implementierung mit blocking send:



"Receiver first"



Idealisierung: senkrechte Pfeile in den Zeitdiagrammen



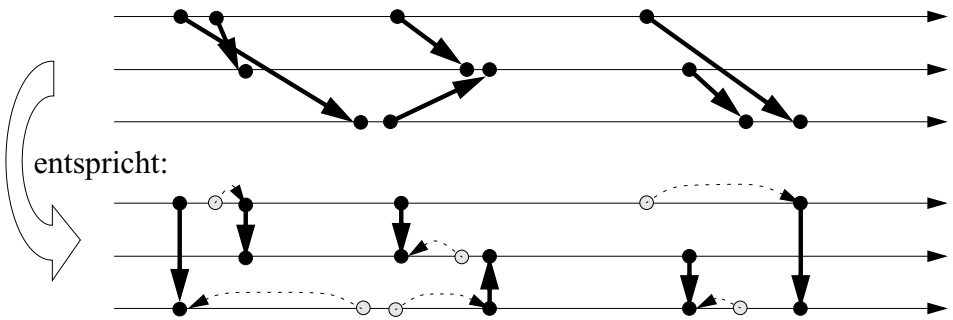
Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet ("!") *und* empfangen ("?") worden!

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

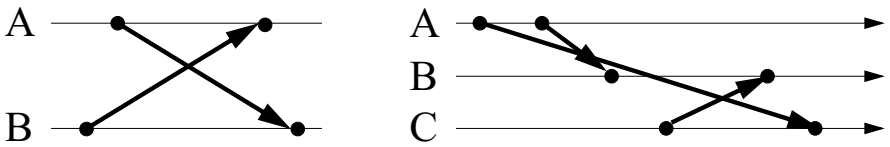
Virtuelle Gleichzeitigkeit

- Ein Ablauf, der synchrone Kommunikation benutzt, ist (bei Abstraktion von der Realzeit) durch ein *äquivalentes* Zeitdiagramm darstellbar, bei dem alle Nachrichtenpfeile senkrecht verlaufen

  - nur stetige Deformation ("Gummiband-Transformation")



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



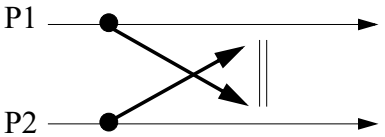
  - aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

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

Deadlocks bei synchroner Kommunikation

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

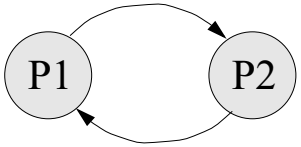
```
P2:
    send (...) to P1;
    receive...
    ...
```



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

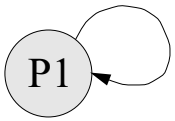
⇒ *Kommunikationsdeadlock!*

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



"Wait-for-Graph"

Genauso tödlich:

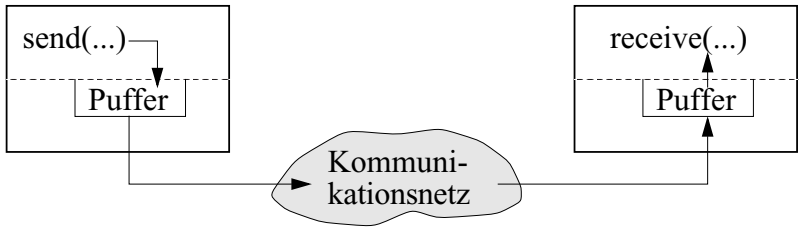



```
P1:
    send (...) to P1;
    receive...
    ...
```

Asynchrone Kommunikation

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

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



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

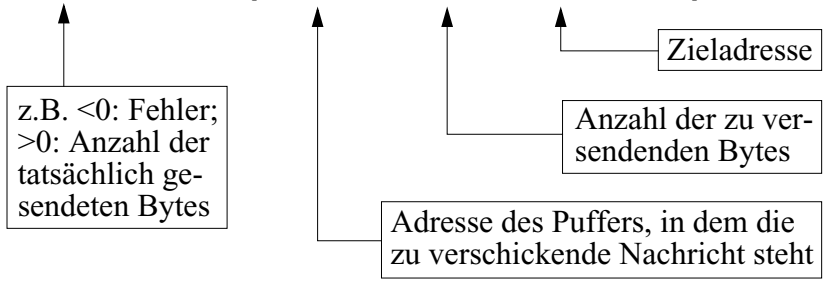
- *Nachteile:*
  (im Vgl. zur synchronen Kommunikation aufwendiger zu realisieren)
  - → Sender weiss nicht, ob / wann Nachricht angekommen
  - → Debugging der Anwendung oft schwierig (wieso?)
  - → System muss Puffer verwalten (wieviele?)

Sendeoperationen in der Praxis

- Es gibt Kommunikationsbibliotheken, deren Dienste von verschiedenen Programmiersprachen (z.B. C) aus aufgerufen werden können
  - z.B. MPI (Message Passing Interface) { Quasi-Standard; verfügbar auf vielen vernetzten Systemen / Compute-Clustern

- Typischer Aufruf einer solchen Send-Operation:

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



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

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

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

- Blockierung ist dann ein rein *senderseitiger* Aspekt
  - *blockierend*: Sender wartet, bis die Nachricht vom Kommunikationssystem abgenommen wurde (und der Puffer wieder frei ist)
  - *nicht blockierend*: Sender informiert Kommunikationssystem lediglich, wo bzw. dass es eine zu versendende Nachricht gibt (Gefahr des Überschreibens des Puffers!)

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

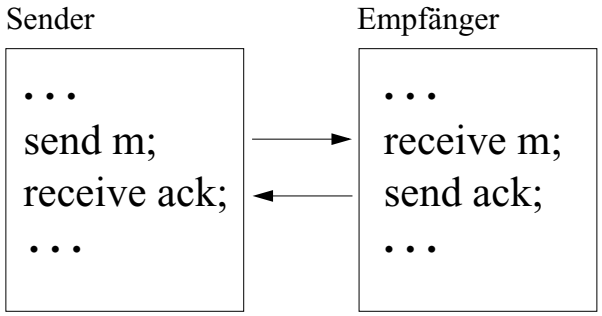
- Nicht-blockierende Operationen liefern oft einen "handle"

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

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

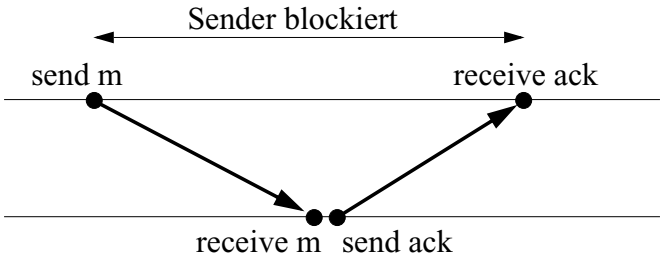
Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:



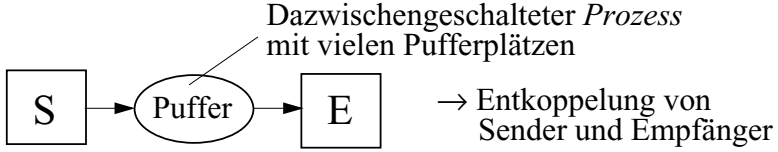
- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)

- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive

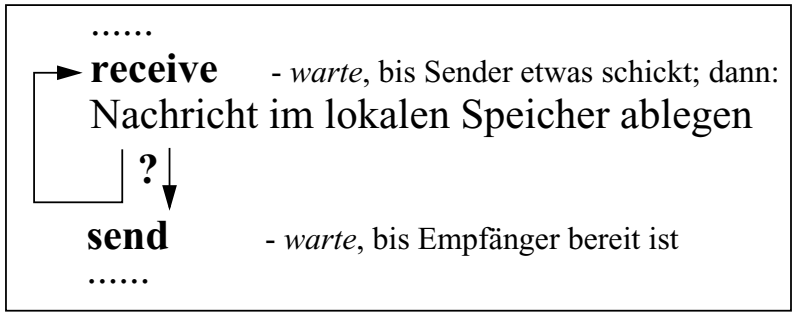


Asynchrone Kommunikation mittels synchroner Kommunikation

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?

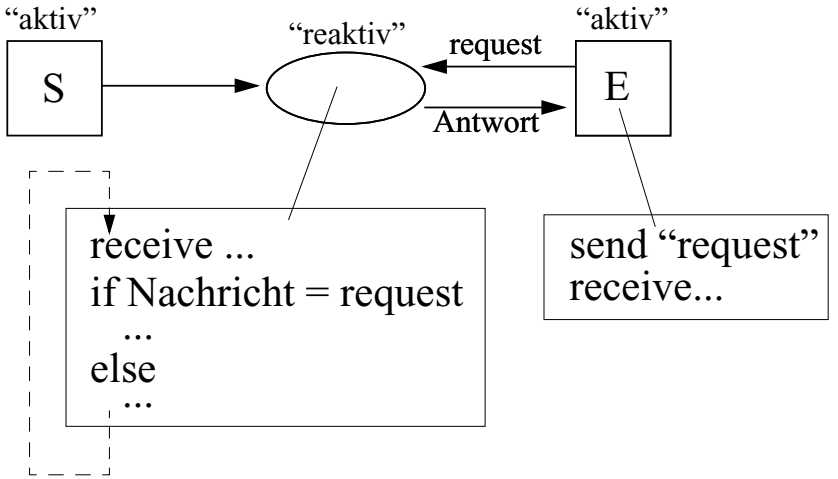
(1) wieder im receive auf den Sender warten, *oder*
(2) in einem send auf den Empfänger warten?

→ Entweder Sender S oder Empfänger E könnte unnötigerweise blockiert sein!

Bemerkung: *Puffer der Gösse 1* lassen sich so realisieren → Kaskadierung im Prinzip möglich ("Pufferpipeline")

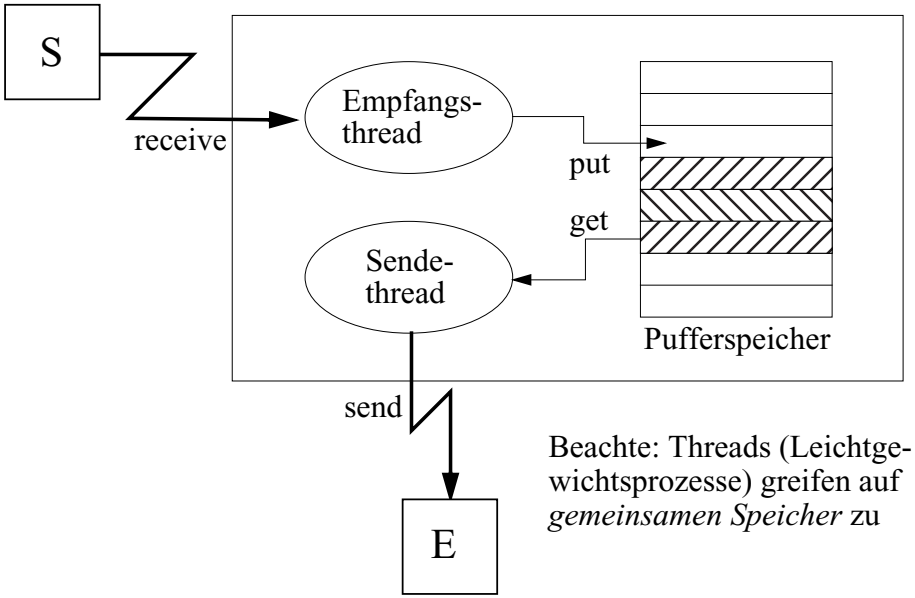
Inversion der Kommunikationsbeziehung

Lösung des zuvorgenannten Problems: (Puffer als *Server*!)



- Puffer schickt E keine Antwort, wenn er leer ist
- Empfänger E wird nur dann verzögert, wenn Puffer leer
- Für Sender S ändert sich nichts
- E muss die Adresse "seines" Puffers kennen
- Was tun, wenn der Puffer *voll* ist?
  - dann sollte der Puffer keine Nachricht von S (aber von E!) annehmen
  - Denkübung: Wie programmiert man das?

Puffer als Multithread-Objekt

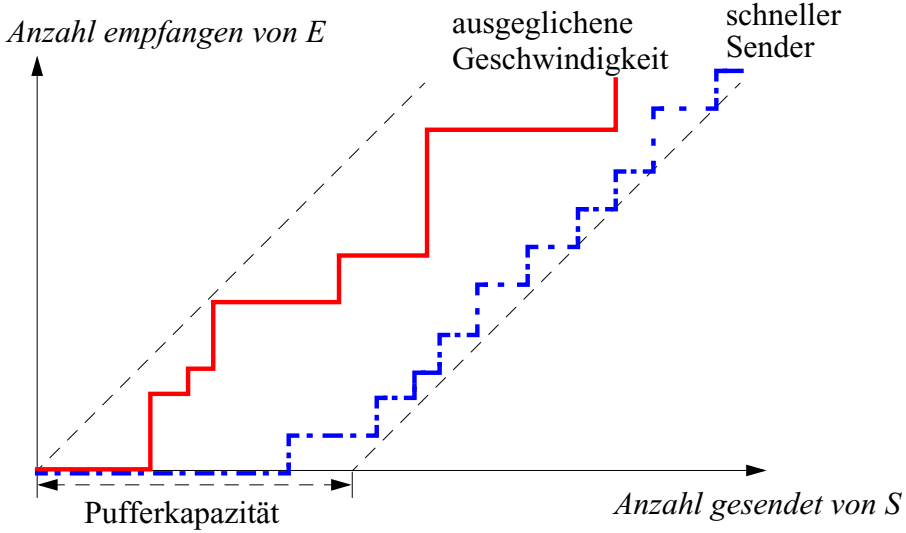


Beachte: Threads (Leichtgewichtsprozesse) greifen auf *gemeinsamen Speicher* zu

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

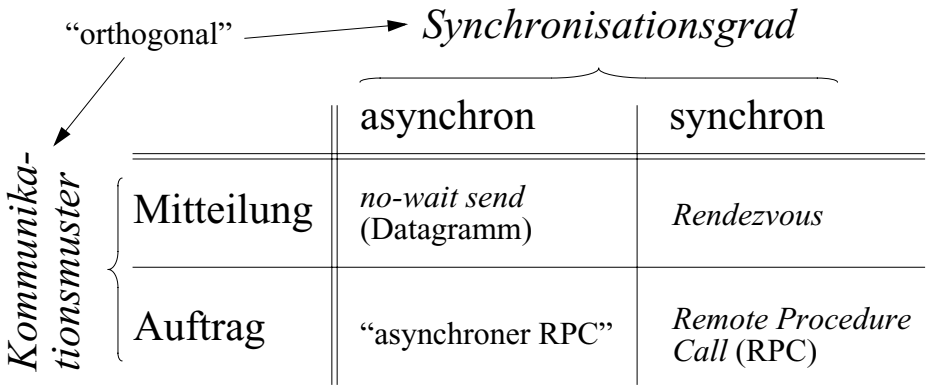
Puffer

- Entkoppelung von Sender und Empfänger durch Puffer



- Anzahl der Pufferplätze bestimmt "Synchronisationsgrad"
  (entspräche ein "Puffer der Grösse 0" der synchronen Kommunikation?)
- Puffer gleicht *Varianz* in der Geschwindigkeit aus, nicht die Geschwindigkeiten selbst!

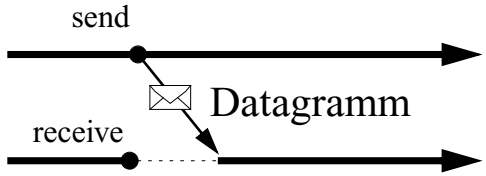
Klassifikation von Kommunikationsmechanismen



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

Datagramm

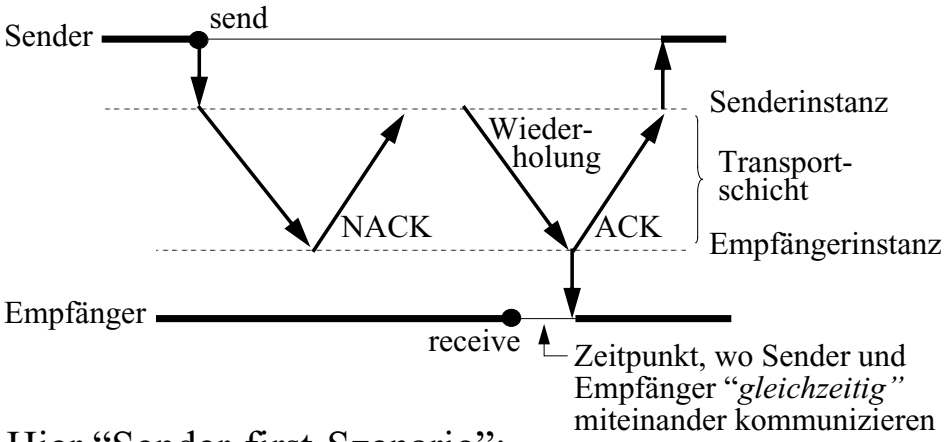
- Asynchron-mitteilungsorientierte Kommunikation



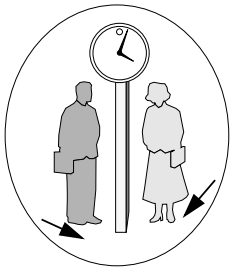
- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- Nachteil
  - keine Erfolgsgarantie für den Sender
  - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicherverwaltung …) im Unterschied etwa zur synchronen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

Rendezvous-Protokolle

- Synchron-mitteilungsorientierte Kommunikation



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



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

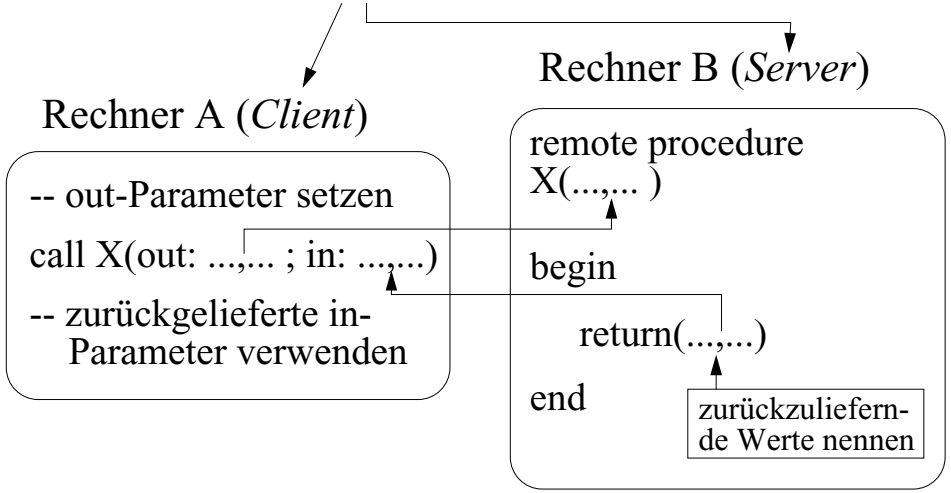
- Mit NACK / ACK ist keine Pufferverwaltung nötig!
  → Aufwendiges Protokoll! ("Busy waiting")
  - Alternative 1: Statt NACK: Nachricht auf Empfängerseite puffern
  - Alternative 2: Statt laufendem Wiederholungsversuch: Empfänger-instanz meldet sich bei Senderinstanz, sobald Empfänger bereit

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

Remote Procedure Call (RPC)

- "Entfernter Prozeduraufruf"

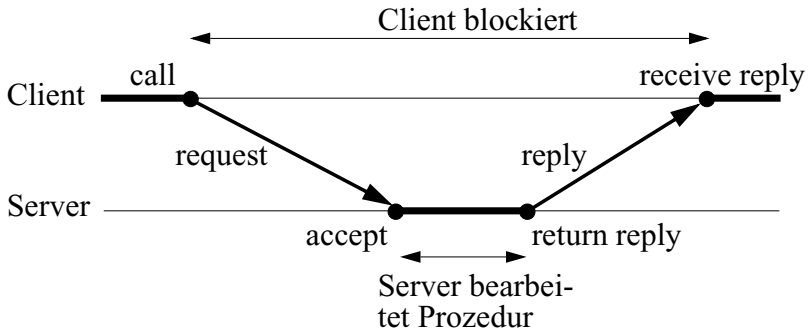
- Synchron-auftragsorientiertiertes Prinzip:



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

- Implementierungsproblem: Verteilungstransparenz
  - Verteiltheit so gut wie möglich verbergen

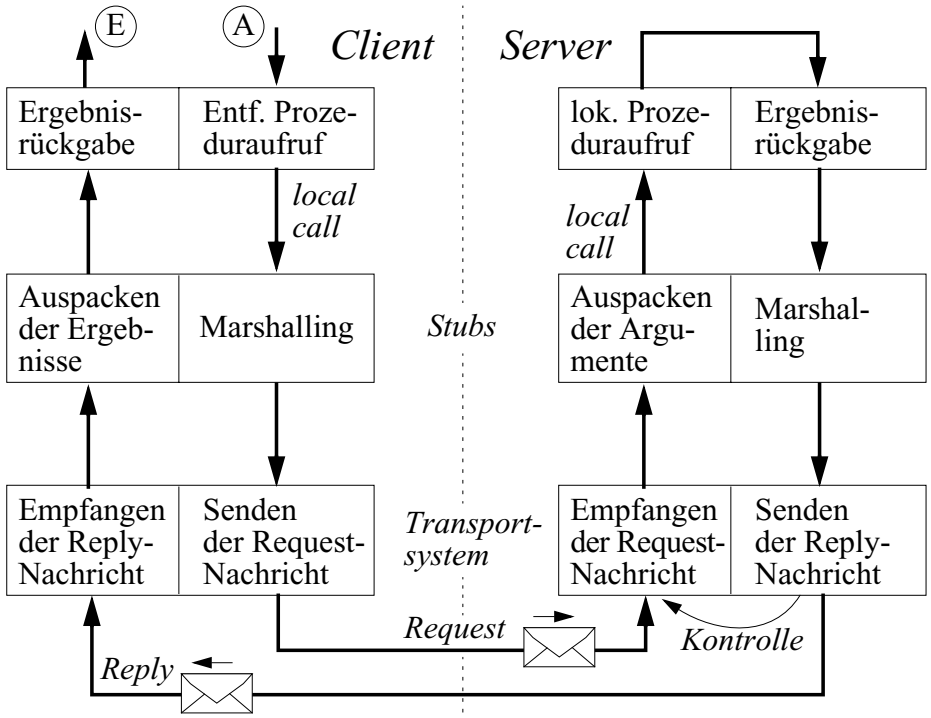
RPC: Prinzipien

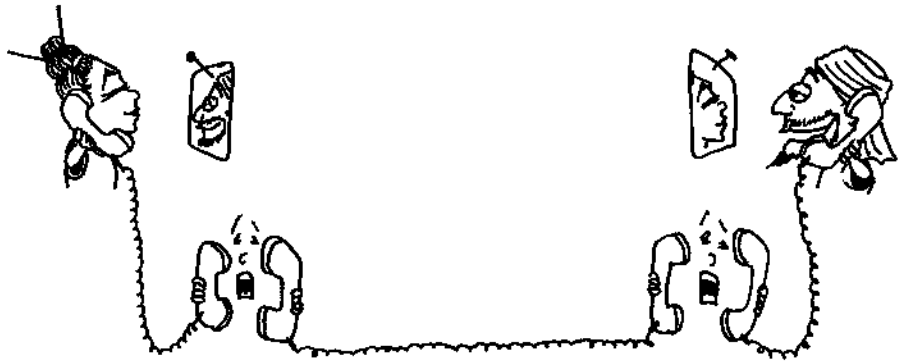


→ send → receive

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

RPC: Implementierung

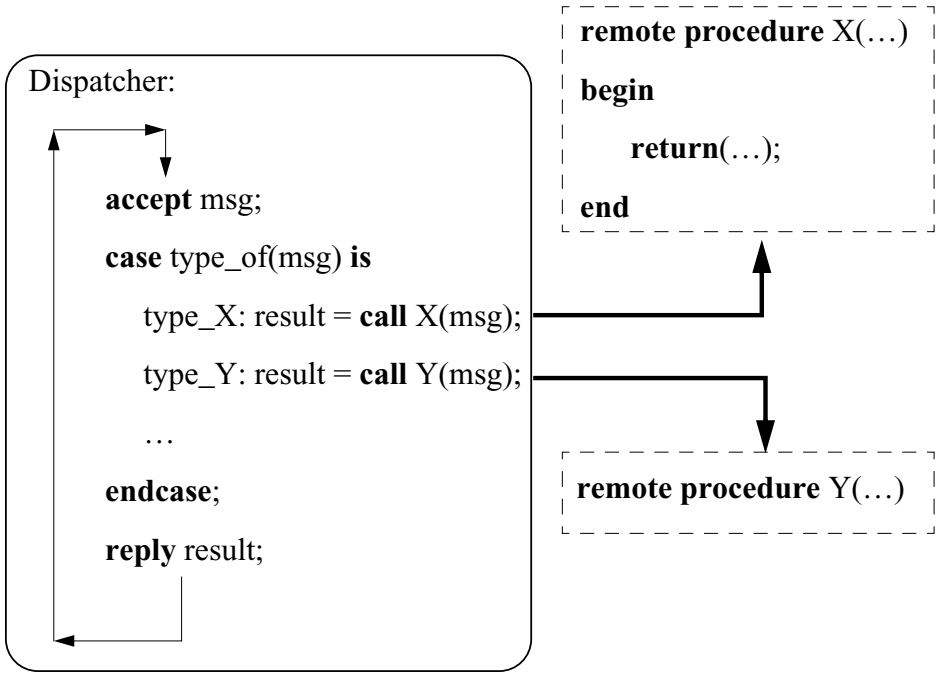




"Kommunikation mit Proxies" (Bild aus dem Buch: "Java ist auch eine Insel")

RPC: Server-Kontrollzyklus

Warten auf Request, Verzweigen zur richtigen Prozedur:



RPC: Stubs

- *Stub* = Stummel, Stumpf

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

Client:

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

- Parameter in eine Nachricht packt
- Nachricht an Server S versendet
- Timeout für die Antwort setzt
- Antwort entgegennimmt (oder evtl. 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 evtl. Zuordnung zwischen Client und Server („Binding")

- Können oft weitgehend *automatisch generiert* werden
  - z.B. mit einem "RPC-Compiler" aus dem Client- oder Server-Code und evtl. einer "sprachneutralen" Schnittstellenbeschreibung (z.B. IDL)
  - Compiler kennt (bei streng getypten Sprachen) Datenformate
  - Schnittstelle zum verfügbaren Transportsystem sind auch bekannt
  - Nutzung von Hilfsbibliotheken für Formatkonversion, Multithreading usw.
  - Nutzung von Routinen einer *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
  - evtl. dabei geeignete Codierung (komplexer) Datenstrukturen
  - Glätten ("flattening") komplexer (evtl. verzeigerter) Datenstrukturen zu einer Sequenz von Basistypen (mit Strukturinformation)
  - umgekehrte Tranformation auch als "unmarshalling" bezeichnet

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

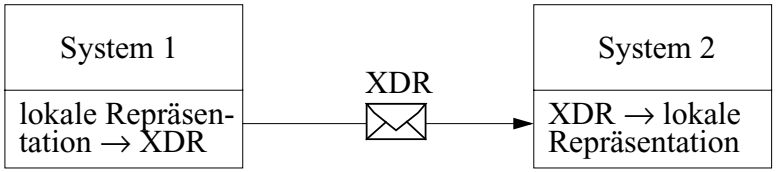
- Falls das Programm in Quellform vorliegt oder vom Compiler generierte Typtabellen existieren, kennen Client und Server den Typ der Parameter und können Typkompatibilitätsprüfungen vornehmen (bzw. die Konversion in ein einheitliches Repräsentationsformat vornehmen)

  (Problematisch evtl. bei ungetypten / schwach getypten Sprachen)

RPC: Datenkonversion

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

   - Prinzip der Datenkonversion:



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

2) Oder lokale Datenrepräsentation verwenden und dies in der Nachricht vermerken
   - "receiver makes it right"
   - Vorteil: bei gleichen Systemumgebungen / Computertypen ist keine (doppelte) Umwandlung nötig
   - Empfänger muss aber mit der Senderrepräsentation umgehen können

---

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

RPC: Transparenzproblematik

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

  bekanntes Programmier-paradigma!

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

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

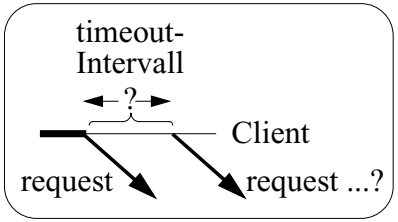
- Ortstransparenz
  - evtl. muss Server ("Zielort") bei Adressierung explizit genannt werden
  - erkennbare Trennung der Adressräume von Client und Server
  - keine Kommunikation über globale Variablen möglich
  - i.Allg. keine Pointer/Referenzparameter als Parameter möglich

- Fehlertransparenz
  - es gibt mehr Fehlerfälle (beim klassischen Prozeduraufruf gilt: Client = Server → "fail-stop"-Verhalten, also "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"

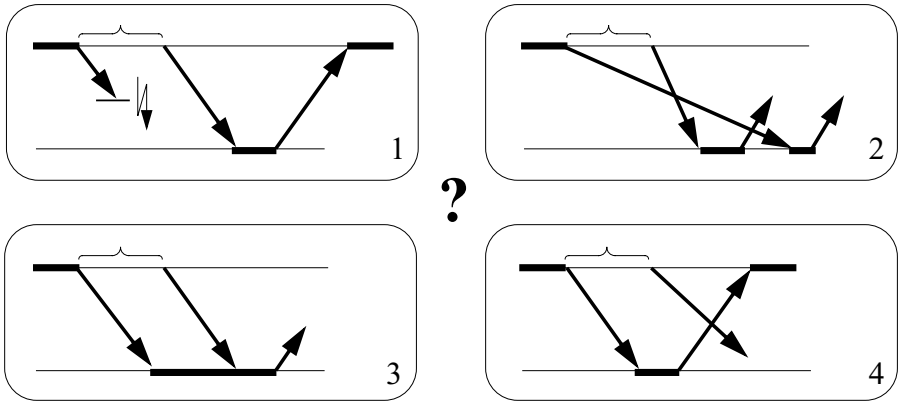
⇒ Fehlerproblematik ist also "kompliziert"!

Typische Fehlerursachen bei RPC: I. Verlorene Request-Nachricht

- *Gegenmassnahme*:
  - Nach Timeout ohne Reply die Request-Nachricht erneut senden

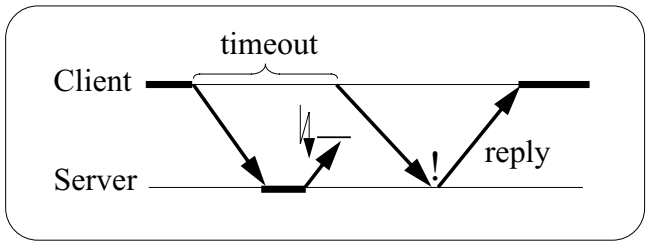


- *Probleme*:
  - Wieviele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
    - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten Operationen!)
    - Server sollte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



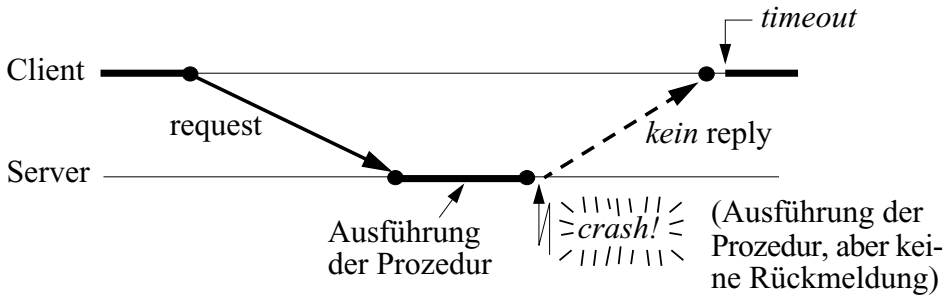
II. Verlorene Reply-Nachricht

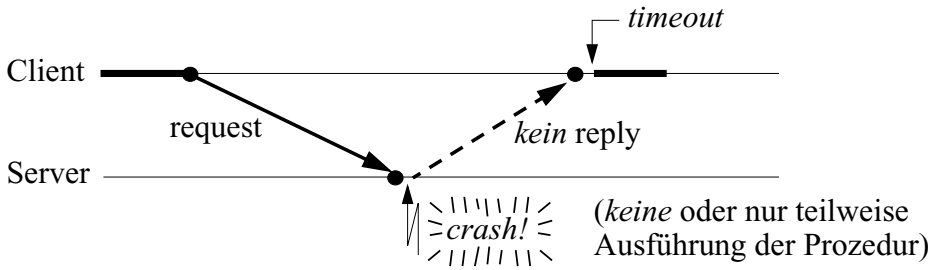
- *Gegenmassnahme 1:* analog zu verlorener Request-Nachricht
  - Also: Anfrage nach Ablauf des Timeouts wiederholen
- *Probleme:*
  - Vielleicht ging aber tatsächlich der Request verloren?
  - Oder der Server war nur langsam und arbeitet noch?
  - Ist aus Sicht des Clients nicht unterscheidbar!



- *Gegenmassnahme 2:*
  - Server hält eine “Historie” versendeter Replies
    - Falls Server Request-Duplikate erkennt und den Auftrag bereits ausgeführt hat: letztes Reply erneut senden, ohne die Prozedur erneut auszuführen
    - Pro Client muss nur das neueste Reply gespeichert werden
    - Bei vielen Clients u.U. dennoch Speicherprobleme:
      → Historie nach “einiger” Zeit löschen
      (Ist in diesem Zusammenhang ein ack eines Reply sinnvoll?)
      Und wenn man ein gelöschtes Reply später dennoch braucht?

III. Server-Crash



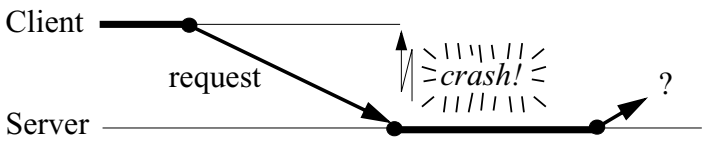


Probleme:

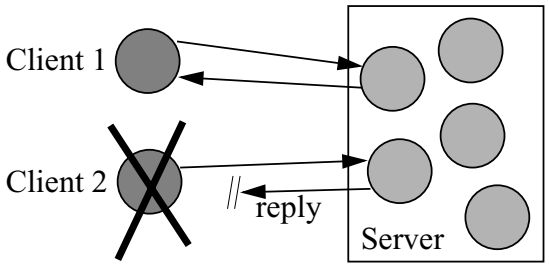
- Wie soll der Client obige Fälle unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinn und Erfolg konkreter Gegenmassnahmen hängt u.U. davon ab
  - Client *meint* u.U. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (→ falsche Sicht des Zustandes!)
- Evtl. Probleme nach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

IV. Client-Crash

- Oder auch: Client schon nicht mehr am reply interessiert



- Reply des Servers wird nicht abgenommen
  - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
  - blockiert i.Allg. Ressourcen beim Server!
- "Orphans" (Waisenkinder) beim Server
  - Prozesse, deren Auftraggeber nicht mehr existiert



Beispiel: Pro RPC-Auftrag eines Client gründet der Server einen eigenen Prozess, der erst terminiert, wenn reply abgenommen wurde

- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)
  - Orphans könnten aber bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben...
- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)

RPC-Fehlersemantik

Operationale Sichtweise:

- Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Requests oder Replies sowie 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:

wird etwas euphemistisch oft als "best effort" bezeichnet

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

3) At-most-once-Semantik:

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

RPC-Fehlersemantik (2)

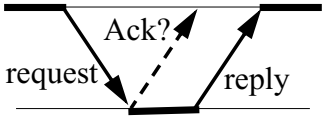
- *Exactly-once-Semantik?*
  - Wunschtraum?
  - Oder geht es zumindest unter der *Voraussetzung*, dass der Server nicht crasht und ein reply letztlich auch durchkommt? (Z.B. durch hartnäckige Wiederholung von Requests?)
  - Was ist mit verteilten Transaktionen? (→ Datenbanken! Stichworte: Checkpoint;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

- Problem der 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

- 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
    (Aber dieses Motto gilt natürlich nicht in allen sonstigen Lebenssituationen! Ein Sicherheitsabstand durch “besser als notwendig” ist oft angebracht!)

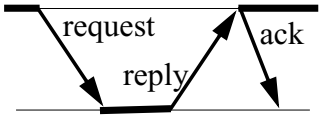
RPC-Protokolle

- *RR-Protokoll* (“Request-Reply”):



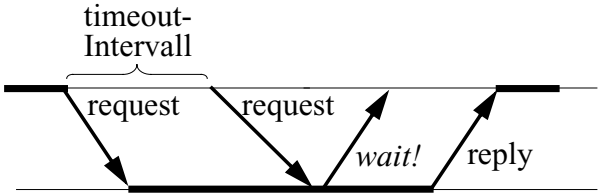
  - Reply ist implizite Quittung für Request
  - lohnt sich ggf. eine unmittelbare Bestätigung des Request?

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



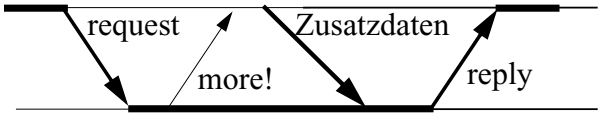
  - “pessimistischer” als das RR-Protokoll
  - Vorteil: Server kann evtl. gespeicherte Replies frühzeitig löschen (und natürlich Replies bei Ausbleiben des ack wiederholen)

- *Sinnvoll bei langen Aktionen / überlasteten Servern:*



“wait” = Bestätigung eines erkannten Duplikats

- *Parameter-Übertragung „on demand“*



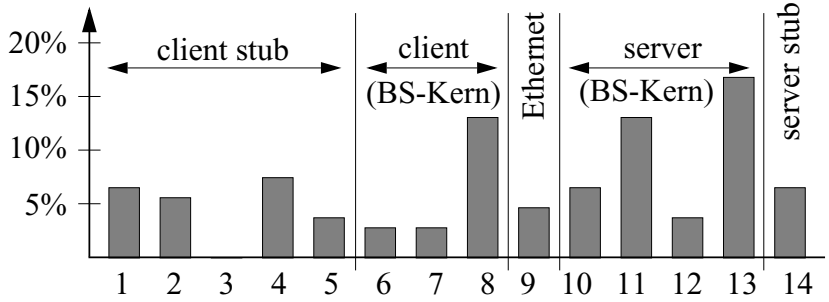
  - spart Pufferkapazität
  - bessere Flusssteuerung
  - Zusatzdaten abhängig vom konkreten Ablauf

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

RPC: Effizienz

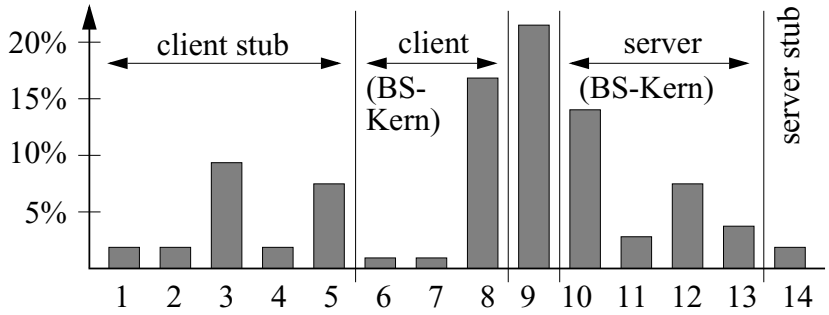
Analyse eines RPC-Protokolls (zitiert nach A. Tanenbaum)

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

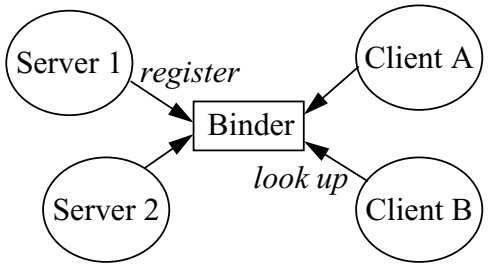


- Eigentliche Übertragung kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zw. Anwendung und BS wichtig
- Mehrfaches Kopieren kostet viel

RPC: Binding

- Problem: Wie werden Client und Server "gematcht"?
  - Verschiedene Rechner und i.Allg. 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

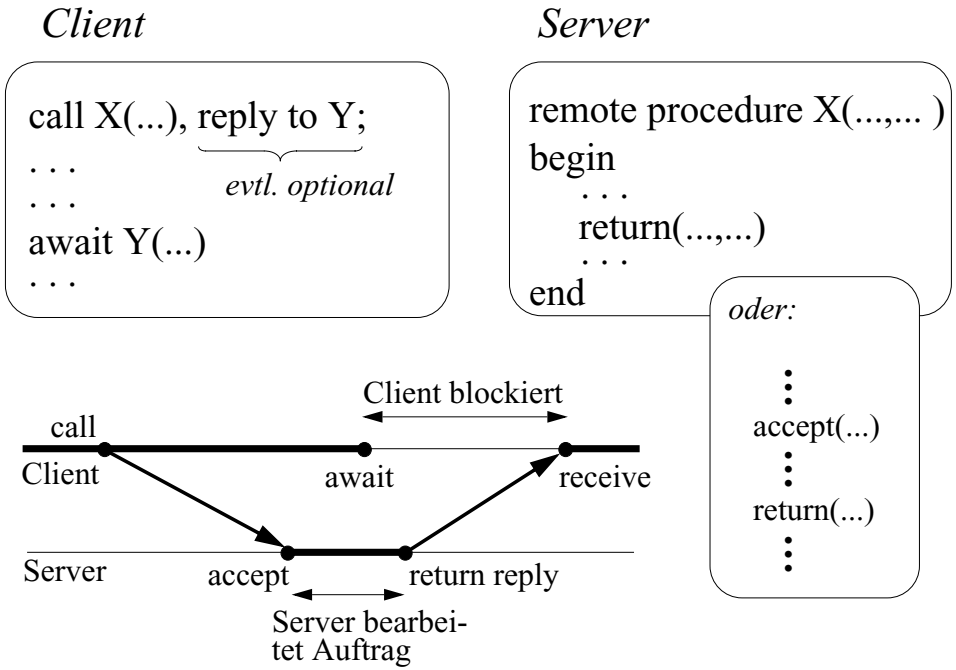
- *Vorteile*: im Prinzip kann Binder
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Existenz eines Services testen
  - verschiedene Versionen eines Dienstes verwalten

dann eher "Trader" oder "Broker"

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

Asynchroner RPC

- andere Bezeichnung: “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);
  - evtl. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt- oder Exception-Routine)

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

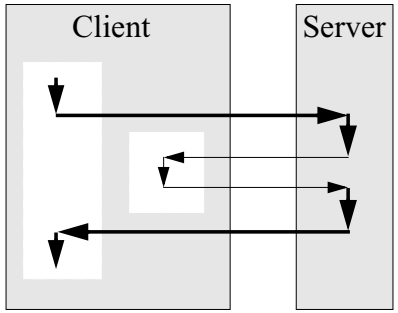
```
FUTURE future: integer;
some_value: integer;
…
future = call(…);
…
some_value = 4711;
print(some_value + future);
```

Beispiel: RPC bei DCE

- DCE (“Distributed Computing Environment”) ist eine Middleware, die in den 1990er-Jahren von einem herstellerübergreifenden Konsortium entwickelt wurde

- RPCs weisen dort einige interessante Besonderheiten auf:

- *Rückrufe* (“call back RPC”)
  - temporärer Rollentausch von Client und Server
  - um evtl. bei langen Aktionen Zwischenresultate zurückzumelden
  - um evtl. weitere Daten vom Client anzufordern
  - Client muss Rückrufadresse übergeben



- *Pipes* als spezielle Parametertypen
  - sind selbst keine Daten, sondern ermöglichen es, Daten stückweise zu empfangen (“pull”-Operation) oder zu senden (“push”)
  - evtl. sinnvoll bei der Übergabe grosser Datenmengen
  - evtl. sinnvoll, wenn Datenmenge erst dynamisch bekannt wird (“stream”)

- *Context-handles* zur aufrufglobalen Zustandsverwaltung
  - werden vom Server dynamisch erzeugt und an Client zurückgegeben
  - Client kann diese beim nächsten Aufruf unverändert wieder mitsenden
  - Kontextinformation zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg z.B. bei Dateiserver (“read; read”) sinnvoll
  - vgl. “cookies”
  - Vorteil: Server arbeitet “zustandslos“

Beispiel: RPC bei DCE (2)

- Semantik für den *Fehlerfall* ist wählbar:

  (a) *at most once*
  - bei temporär gestörter Kommunikation wird Aufruf automatisch wiederholt; eventuelle Aufrufduplikate werden gelöscht
  - Fehlermeldung an Client bei permanentem Fehler

  (b) *idempotent*
  - keine automatische Unterdrückung von Aufrufduplikaten
  - Aufruf wird ein-, kein-, oder mehrmals ausgeführt
  - effizienter als (a), aber nur für wiederholbare Dienste geeignet

  (c) *maybe*
  - wie (b), aber ohne Rückmeldung über Erfolg oder Fehlschlag
  - noch effizienter, aber nur in speziellen Fällen anwendbar

- Optionale *Broadcast*-Semantik
  - Nachricht wird an mehrere lokale Server geschickt
  - RPC ist beendet mit der ersten empfangenen Antwort

- Wählbare Sicherheitsstufen bei der Kommunikation
  - Authentifizierung nur bei Aufbau der Verbindung (“binding”)
  - Authentifizierung pro RPC-Aufruf
  - Authentifizierung pro Nachrichtenpaket
  - Zusätzlich Verschlüsselung jedes Nachrichtenpaketes
  - Schutz gegen Verfälschung (verschlüsselte Prüfsumme)