Prioritäten von Nachrichten?

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

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

  } Durchbrechung der FIFO-Reihenfolge!

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

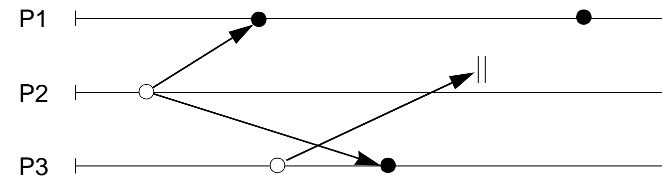
Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!

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

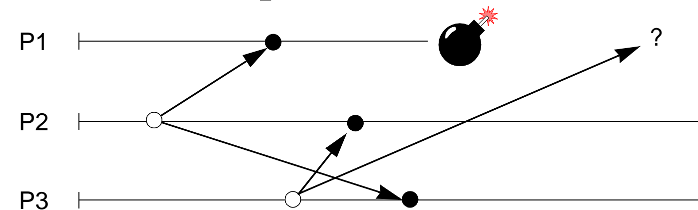
Fehlermodelle

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

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



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



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

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

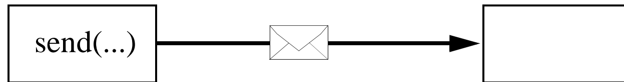
(solche Fehler lassen sich nur teilweise, z.B. durch *Redundanz,* erkennen)

Fehlertolerante Algorithmen sollen das "richtige" Fehlermodell berücksichtigen!

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

Mitteilung vs. Auftrag

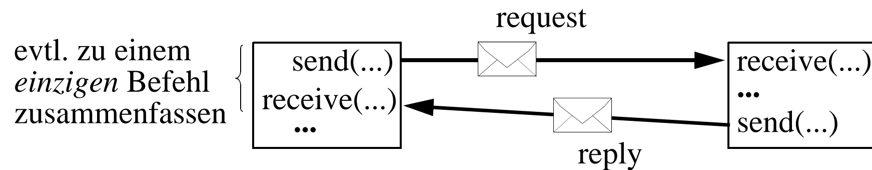
Mitteilungsortientiert:



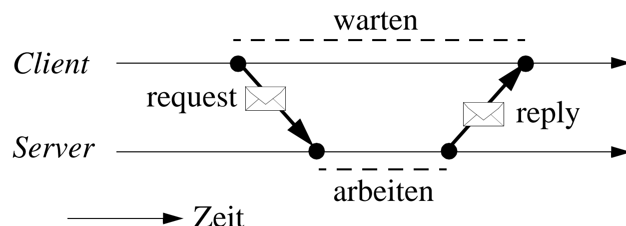
- Unidirektional
- Übermittelte Werte werden der Nachricht typw. als “Ausgabeparameter” beim send übergeben

---

Auftragsortientiert:

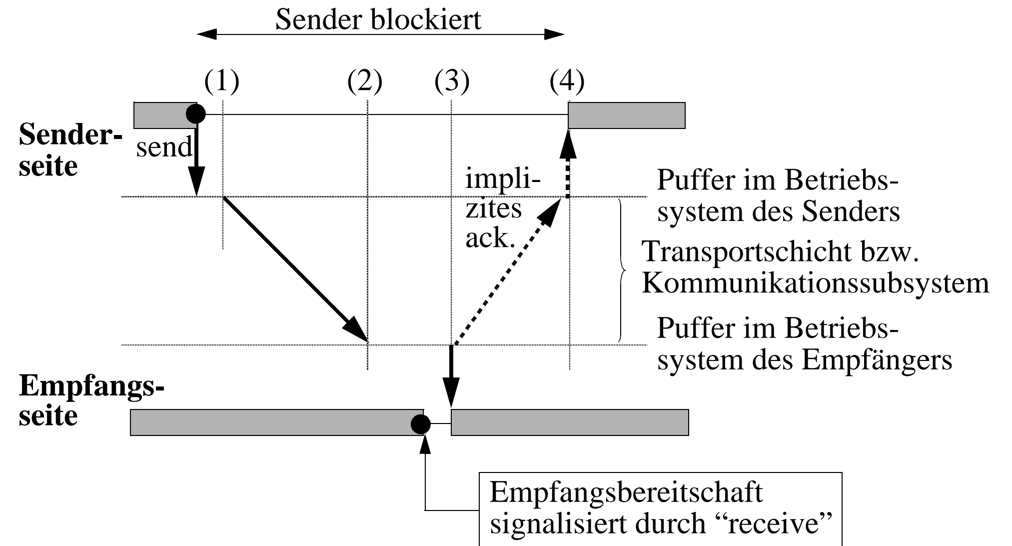


- Bidirektional
- Ergebnis eines Auftrags wird als “Antwortnachricht” zurückgeschickt
- Auftraggeber (“Client”) *wartet* bis Antwort eintrifft



Blockierendes Senden

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



- Verschiedene Ansichten einer adäquaten Definition von “Abschluss der Transaktion” aus Sendersicht:
  - *Zeitpunkt 4* (automatische Bestätigung, dass der Empfänger das receive ausgeführt hat) ist die höhere, anwendungsorientierte Sicht.
  - Falls eine Bestätigung bereits zum *Zeitpunkt 2* geschickt wird, weiss der Sender nur, dass die Nachricht am Zielort zur Verfügung steht und der Sendepuffer wieder frei ist. Vorher sollte der Sendepuffer nicht überschrieben werden, weil die Nachricht bei fehlerhafter Übertragung evtl. wiederholt werden muss.

Synchrone Kommunikation

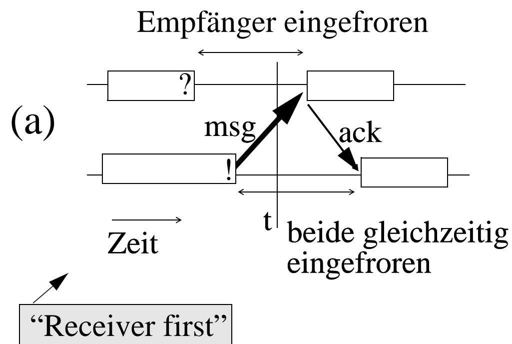
"gleich" "zeitig"

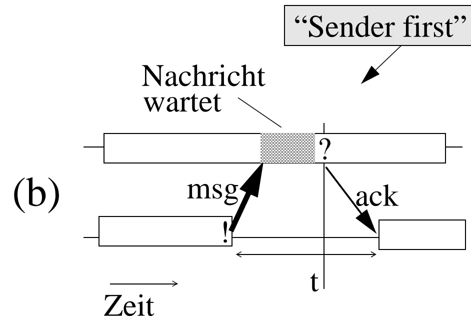
- Idealisierung: Send und receive geschehen *gleichzeitig*

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

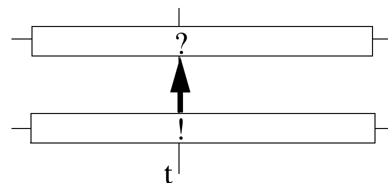
- Bem.: "Receive" ist i.Allg. blockierend (d.h. Empfänger wartet so lange, bis eine Nachricht ankommt)

Implementierung mit blocking send:





Idealisierung:
senkrechte Pfeile in den Zeitdiagrammen



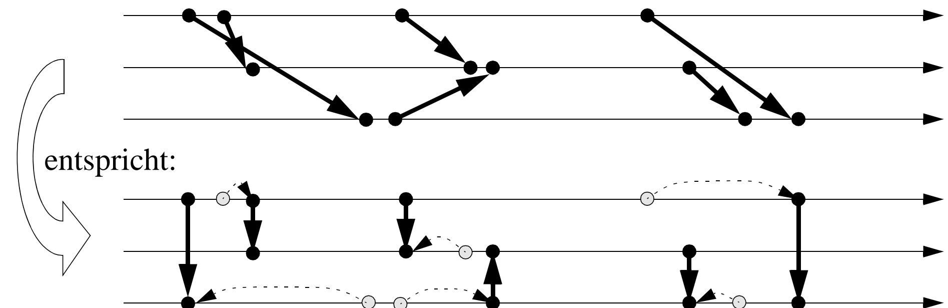
Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet ("!") *und* empfangen ("?") worden!

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

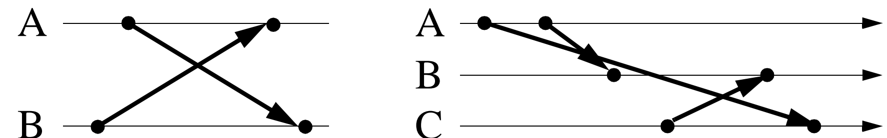
→ *"Rendezvous"*

Virtuelle Gleichzeitigkeit

- Ein Ablauf, der synchrone Kommunikation benutzt, ist (bei Abstraktion von der Realzeit) durch ein *äquivalentes* Zeitdiagramm darstellbar, bei dem alle Nachrichtenpfeile senkrecht verlaufen
  - nur stetige Deformation erlaubt ("Gummiband-Transformation")



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



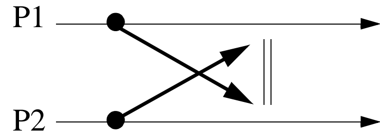
  - aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

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

Deadlocks bei synchroner Kommunikation

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

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

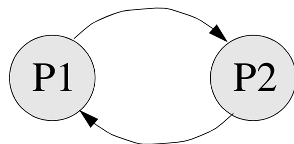


In beiden Prozessen muss zunächst das *send* ganz ausgeführt werden, bevor es zu einem *receive* kommt

⇒ *Kommunikationsdeadlock!*

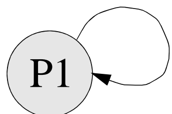
Zyklische Abhängigkeit der Prozesse voneinander: P1 wartet auf P2, und P2 wartet auf P1

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



"Wait-for-Graph"

Genauso tödlich:

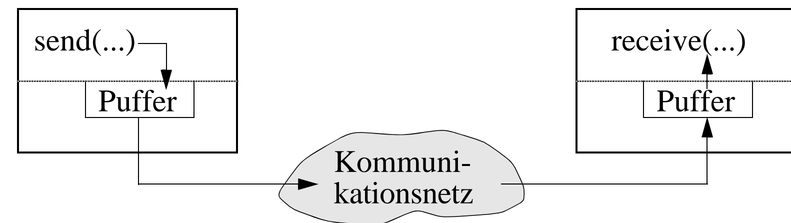


P1:
send (...) to P1;
receive...
...

Asynchrone Kommunikation

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

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



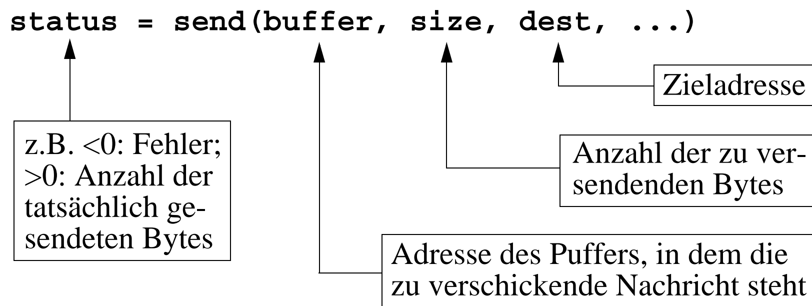
- *Vorteile:*
  (im Vgl. zur syn. Kommunikation oft angenehmer in der Anwendung)
  → Sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird
  → Stärkere Entkoppelung von Sender / Empfänger
  → Höherer Grad an Parallelität möglich
  → Geringere Gefahr von Kommunikationsdeadlocks

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

Sendeoperationen in der Praxis

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

- Typischer Aufruf einer solchen Send-Operation:



- 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)
  - Vorsicht: Semantik und Kontext der Anwendbarkeit ist oft nur informell beschrieben

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

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

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

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

- Nicht-blockierende Operationen liefern oft einen “handle”

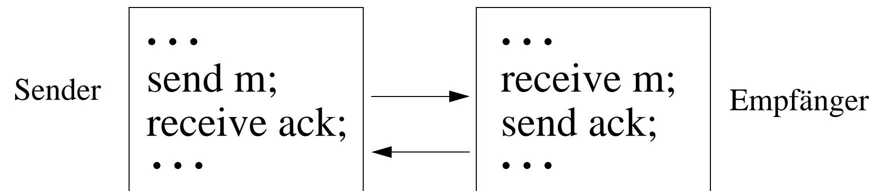
```
handle = send(...)
```

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

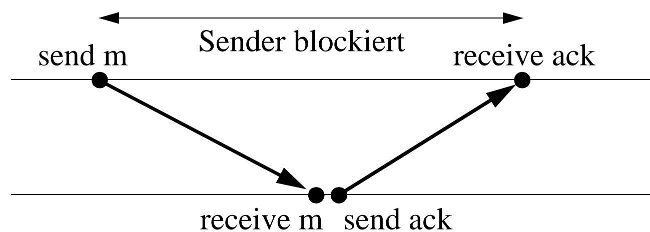
- Nicht-blockierend ist effizienter aber u.U. unsicherer und umständlicher (evtl. Test; warten) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:

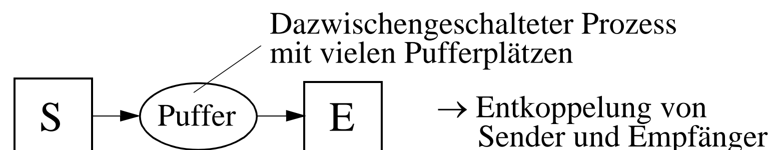


- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive



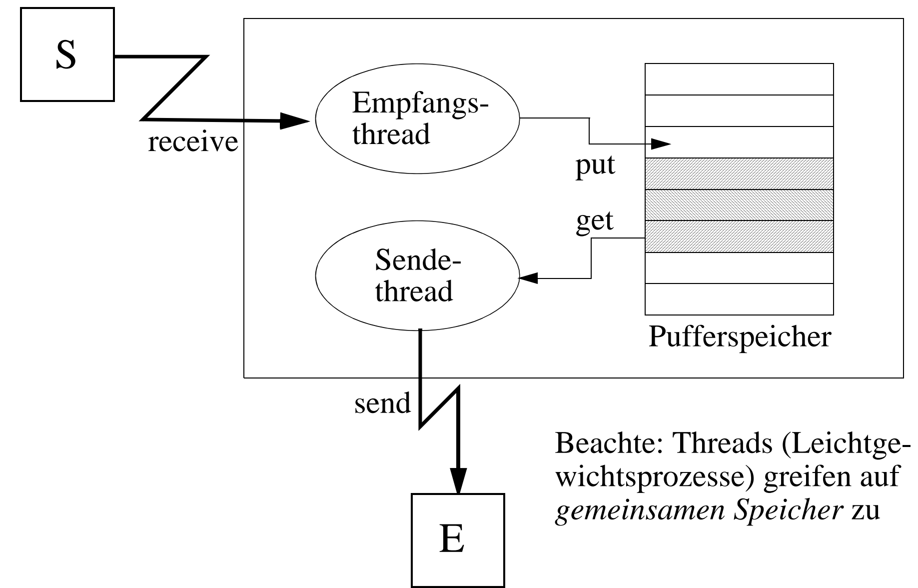
Asynchrone Kommunikation mittels synchroner:

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



→ Entkoppelung von Sender und Empfänger

Puffer als Multithread-Objekt



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

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

Klassifikation von Kommunikationsmechanismen

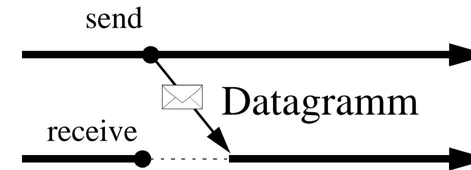
"orthogonal" → *Synchronisationsgrad* / *Kommunikationsmuster*

Kommunikationsmuster \ *Synchronisationsgrad*	asynchron	synchron
Mitteilung	*no-wait send* (Datagramm)	*Rendezvous*
Auftrag	"asynchroner RPC"	*Remote Procedure Call* (RPC)

- 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

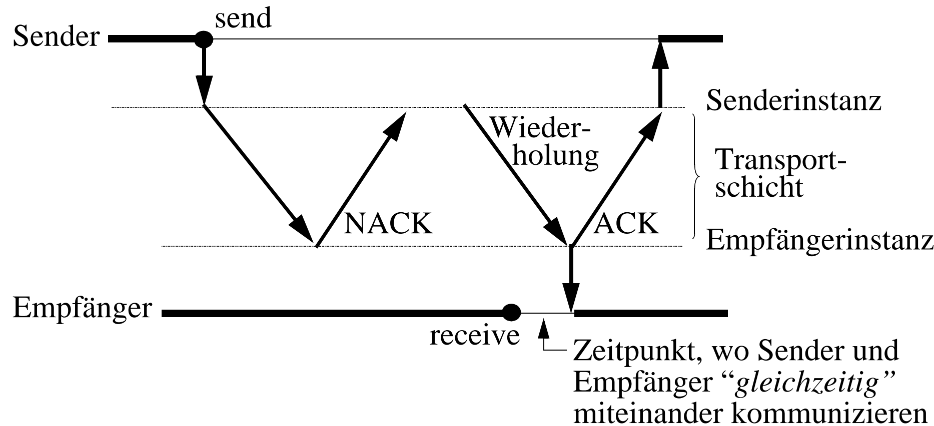
- Asynchron-mitteilungsorientierte Kommunikation



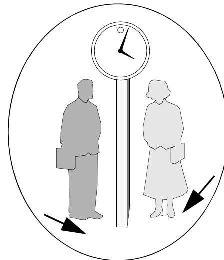
- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- Nachteil
  - keine Erfolgsgarantie für den Sender
  - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicherverwaltung …) im Unterschied etwa zur synchronen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

Rendezvous-Protokolle

- Synchron-mitteilungsorientierte Kommunikation



- Hier beispielhaft "Sender-first-Szenario": Sender wartet als Erster
  - "Receiver-first-Szenario" analog



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

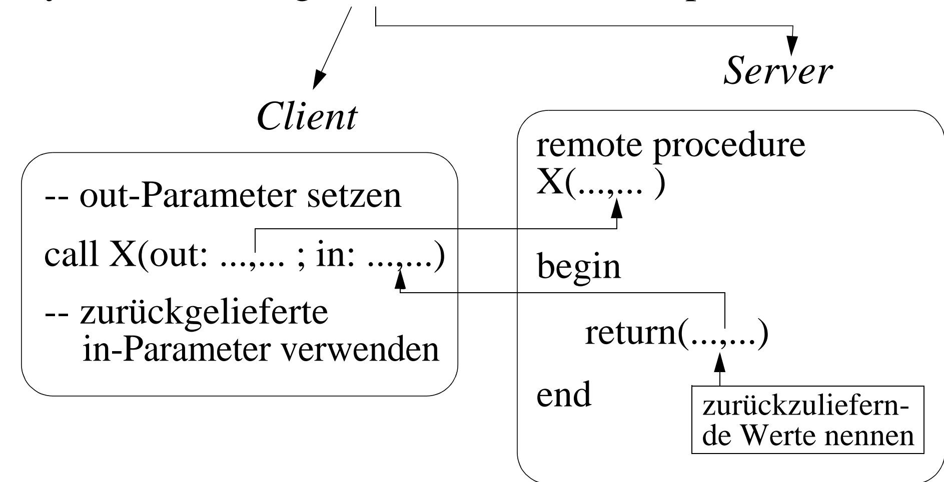
- Mit NACK / ACK sind weniger Puffer nötig
  → Aber aufwändiges Protokoll! ("busy waiting")
  - Alternative 1: Statt NACK: Nachricht auf Empfängerseite puffern
  - Alternative 2: Statt laufendem Wiederholungsversuch: Empfänger-instanz meldet sich bei Senderinstanz, sobald Empfänger bereit

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

Remote Procedure Call (RPC)

- Aufruf einer "entfernten Prozedur"

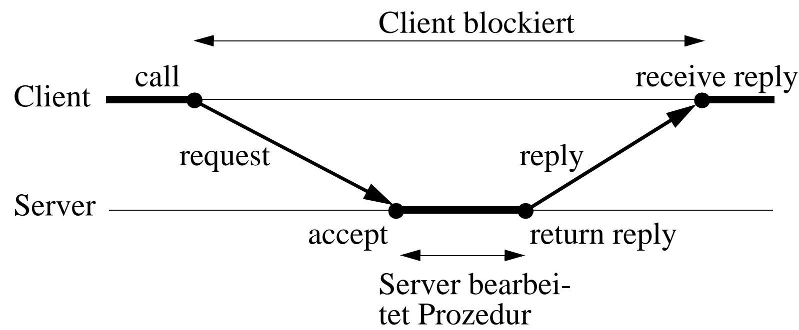
- Synchron-auftragsorientiertiertes Prinzip:

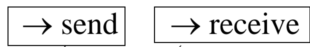


- Soll dem klassischen Prozeduraufruf möglichst gleichen
  - klare Semantik für den Anwender (Auftrag als „Unterprogramm")
  - einfaches Programmieren
    - kein Erstellen von Nachrichten, kein Quittieren,... auf Anwendungsebene
    - Syntax analog zu bekanntem lokalen Prozeduraufruf
    - Verwendung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

- Implementierungsproblem: Verteilungstransparenz
  - Verteiltheit so gut wie möglich verbergen

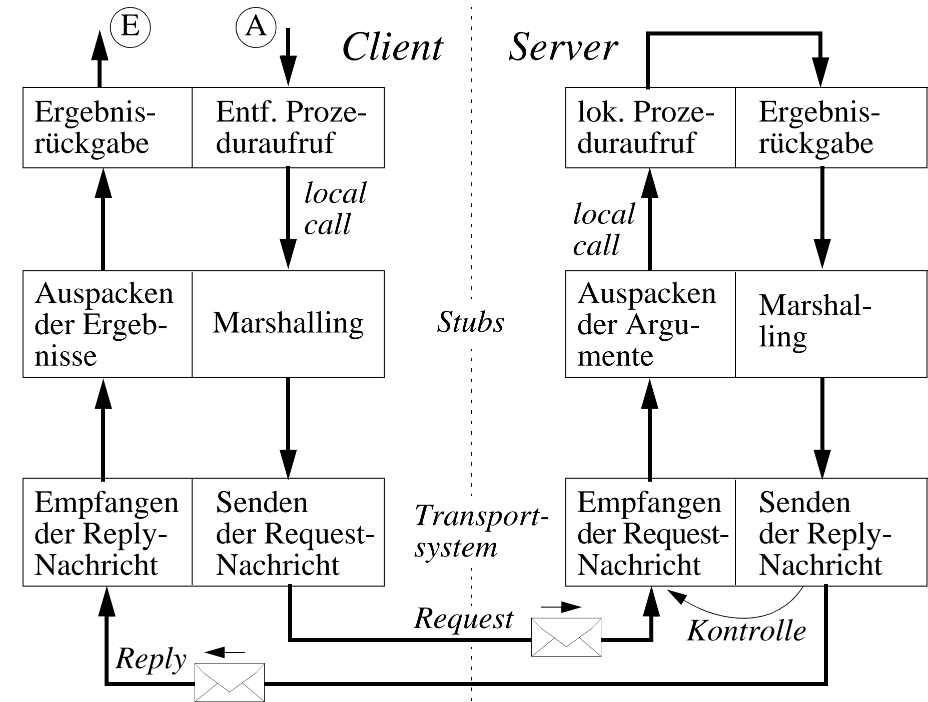
RPC: Prinzipien

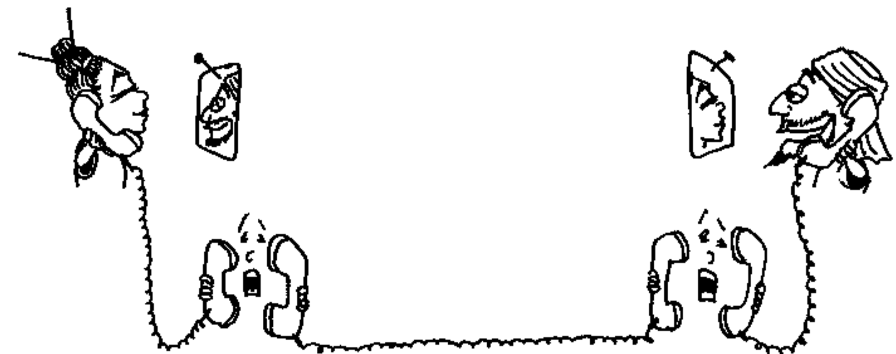




- call; accept; return; receive: interne Anweisungen
  - nicht sichtbar auf Sprachebene → Compiler bzw. realisiert im API
- Parameterübergabe-Semantik: call-by-value/result
- Keine Parallelität zwischen Client und Server
  - RPC-Aufrufe sind blockierend

RPC: Implementierung

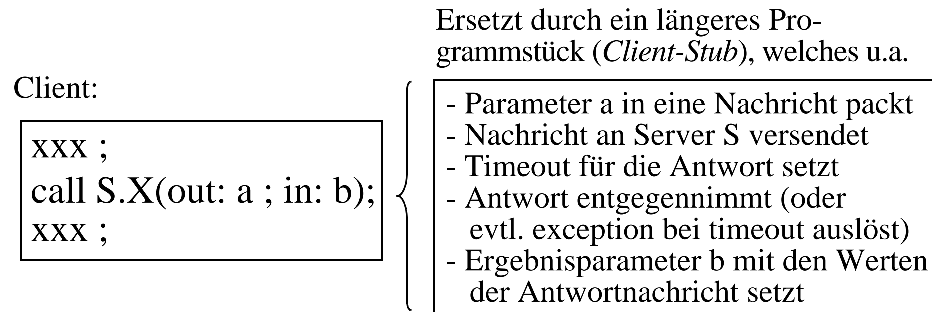




"Kommunikation mit Proxies" (Bild aus dem Buch: "Java ist auch eine Insel")

RPC: Stubs

- *Stub* = Stummel, Stumpf



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

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

  wird nicht generiert, sondern dazugebunden

- Stubs sorgen also für *Transparenz*

RPC: Marshalling

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

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

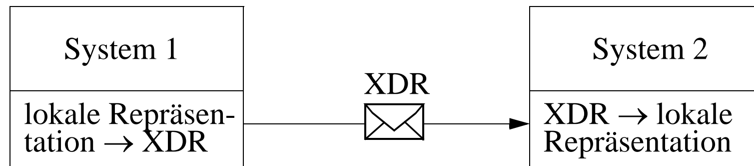
- Problem: RPCs werden oft in *heterogenen* Umgebungen eingesetzt mit unterschiedlicher Repräsentation z.B. von
  - Strings (Längenfeld ↔ '\0')
  - Character (ASCII ↔ Unicode)
  - Arrays (zeilen- ↔ spaltenweise)
  - niedrigstes Bit einer Zahl vorne oder hinten

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

RPC: Datenkonversion

1) Umwandlung in eine gemeinsame Standardrepräsentation

- z.B. "XDR" (eXternal Data Representation)



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

2) Oder lokale Datenrepräsentation verwenden und dies in der Nachricht vermerken

- "receiver makes it right"
- Vorteil: bei gleichen Systemumgebungen / Computertypen ist keine (doppelte) Umwandlung nötig
- Empfänger muss aber mit der Senderrepräsentation umgehen können

---

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

RPC: Transparenzproblematik

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

bekanntes Programmier-paradigma!

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

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

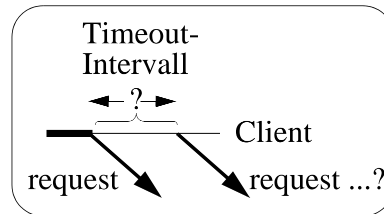
- Ortstransparenz?
  - evtl. muss Server ("Zielort") bei Adressierung explizit genannt werden
  - erkennbare Trennung der Adressräume von Client und Server
  - keine Kommunikation über globale Variablen möglich
  - i.Allg. keine Pointer/Referenzparameter als Parameter möglich

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

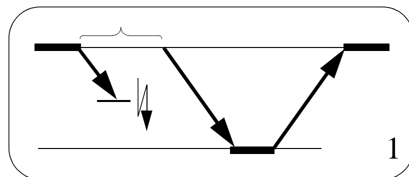
⇒ Fehlerproblematik ist also "kompliziert"!

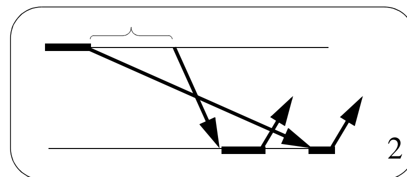
Typische Fehlerursachen bei RPC: I. Verlorene Request-Nachricht

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

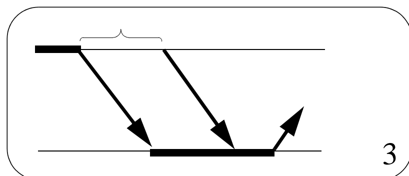


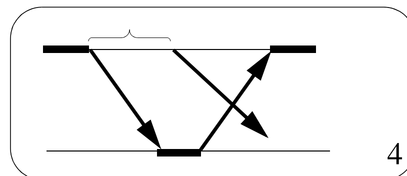
- *Probleme*:
  - Wieviele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
    - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten serverseitigen Operationen!)
  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





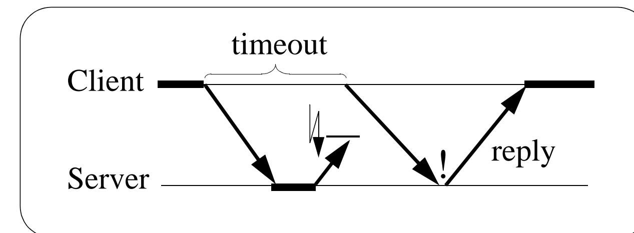
?





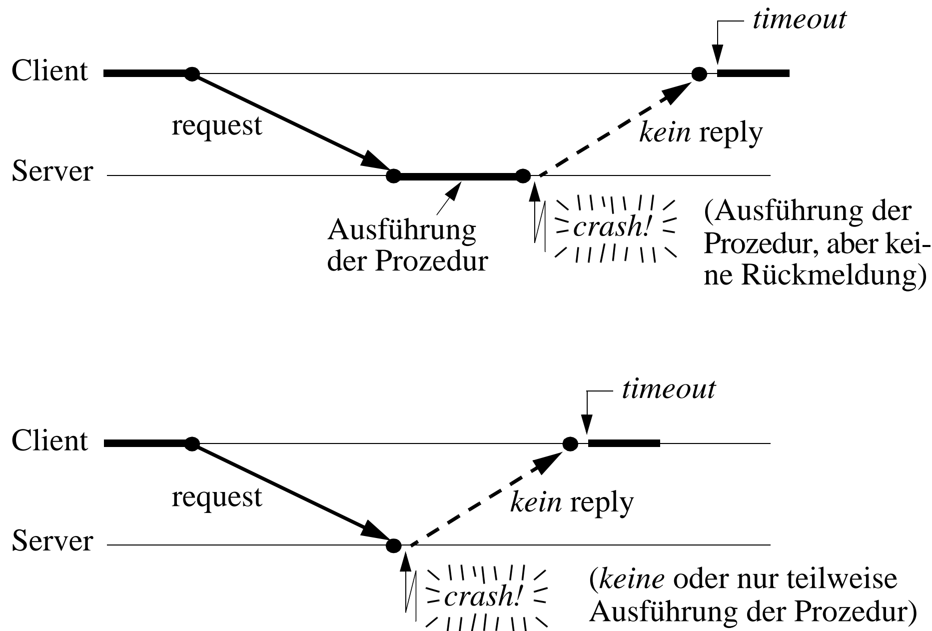
II. Verlorene Reply-Nachricht

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



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

III. Server-Crash



Probleme:

- Wie soll der Client obige Fälle unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinn und Erfolg konkreter Gegenmassnahmen hängt u.U. davon ab
  - Client *meint* evtl. 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

Typische RPC-Fehlersemantik-Klassen

Operationale Sichtweise:

- Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?

---

1) Maybe-Semantik:

- Keine Wiederholung von Requests
- *Einfach und effizient*
- Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
  Mögliche Anwendungsklasse: Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

2) At-least-once-Semantik:

1) und 2) werden etwas euphemistisch oft als "best effort" bezeichnet

- Hartnäckige automatische 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

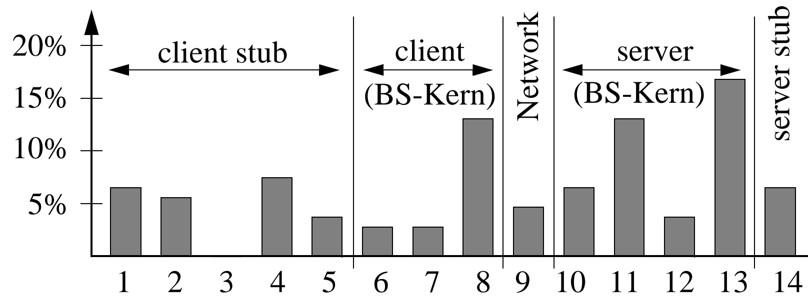
ist "exactly once" machbar?

- May-be → At-least-once → At-most-once → ... ist zunehmend aufwändiger 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: Effizienz

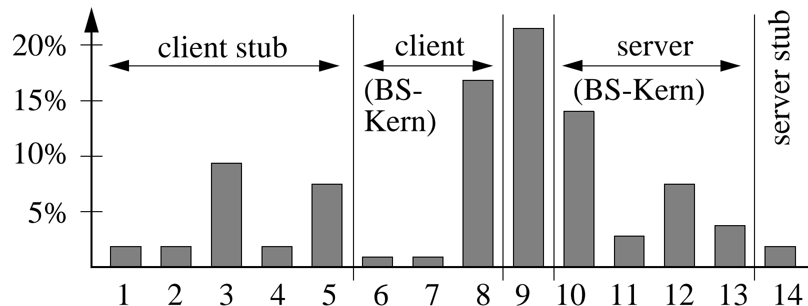
Analyse eines RPC-Protokolls (zitiert nach A. Tanenbaum)

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. Network 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

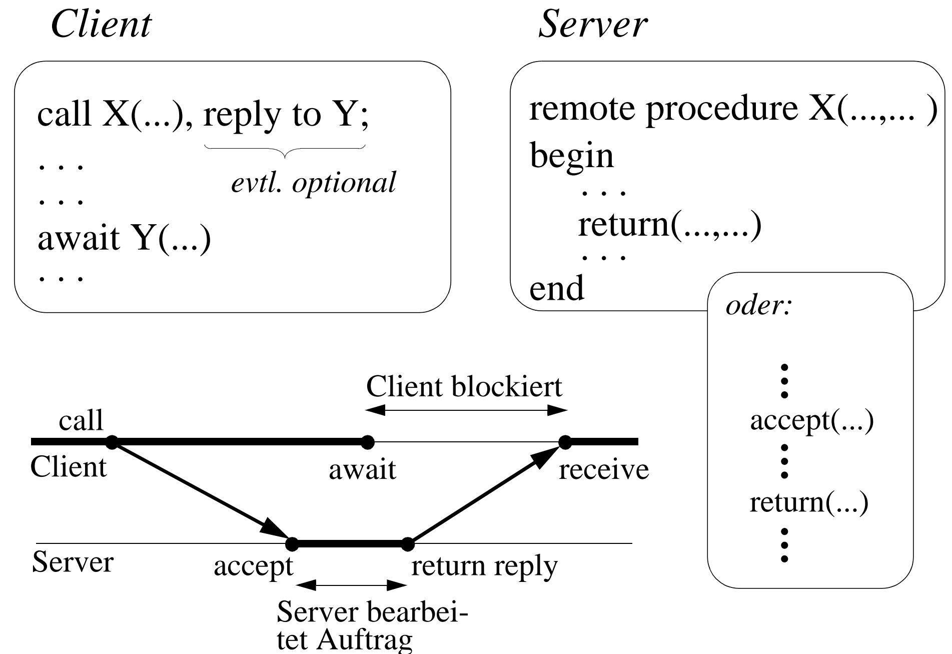
b) 1440 Byte Nutznachricht (ebenfalls keine Auftragsbearbeitung):



- Eigentliche Übertragung (9) kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zwischen Anwendung und Betriebssystem relevant
- Mehrfaches Kopieren kostet viel

Asynchroner RPC

- Andere Bezeichnung: “Remote Service Invocation”
- Auftragsorientiert → Antwortverpflichtung



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

Asynchroner RPC: Ergebnisempfang

- 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:

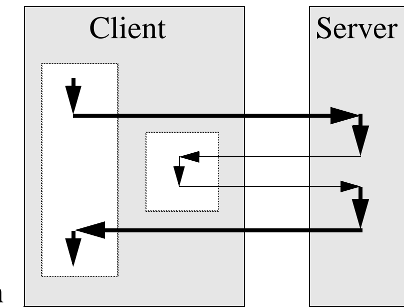
```
FUTURE f: integer;
...
f = call(...);
...
some_value = 4711;
print(some_value + f);
```


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 bei DCE-RPCs 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 faktisch ein-, kein-, oder mehrmals ausgeführt
- effizienter als (a), aber nur für wiederholbare Dienste geeignet

(c) *maybe*

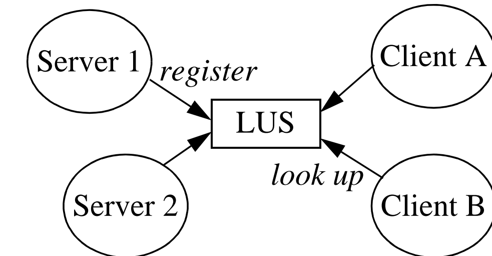
- wie (b), aber ohne Rückmeldung über Erfolg oder Fehlschlag
- noch effizienter, aber nur in speziellen Fällen anwendbar

- 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
  - Zusätzlich Schutz gegen Verfälschung (verschlüsselte Prüfsumme)

Lookup-Service

- Problem: Wie finden sich Client und Server ?
  - haben i.Allg. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gemeinsame Umgebung)
  - Lookup Service (LUS) oder "registry"



- Server (-stub) gibt den Namen etc. seines Services (RPC-Routine) dem LUS bekannt
  - "*register*"; "exportieren" der RPC-Schnittstelle (Typen der Parameter,...)
  - evtl. auch wieder abmelden

- Client erfragt beim LUS die Adresse eines geeigneten Servers (aber wie genau?)
  - "*look up*"; "discovery"; "importieren" der RPC-Schnittstelle

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

- *Probleme*:
  - lookup kostet Ausführungszeit (gegenüber statischem Binden)
  - zentraler LUS ist ein potentieller Engpass / "single point of failure" (Lookup-Service geeignet replizieren / verteilen?)

Adressierung

- Adressen sind u.a. Geräteadressen (z.B. IP-Adresse oder Netzadresse in einem LAN), Portnamen, Socketnummern, Referenzen auf Mailboxes,...

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

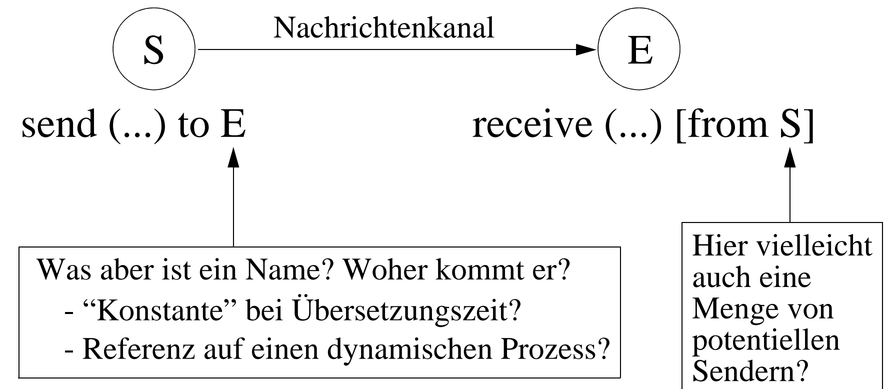
- *Empfänger* ist evtl. nicht bereit, jede beliebige Nachricht von jedem Sender zu akzeptieren
  - selektiver Empfang
  - Sicherheitsaspekte, Überlastabwehr

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

- Woher kennt ein Sender die Adresse des Empfängers?
  1) Fest in den Programmcode integriert → unflexibel
  2) Über Parameter erhalten oder von anderen Prozessen mitgeteilt
  3) Adressanfrage per Broadcast “in das Netz”
     - häufig bei LANs: Suche nach lokalem Nameserver, Router etc.
  4) Auskunft fragen (Namensdienst wie z.B. DNS; Lookup-Service)

Direkte Adressierung

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

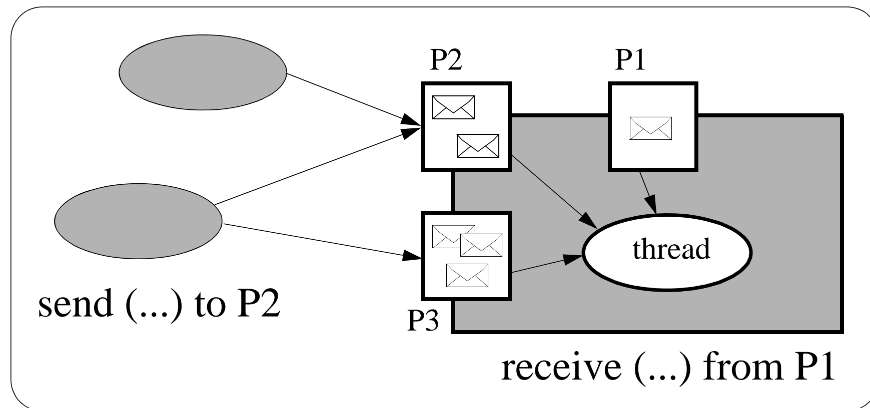


- Direct naming ist insgesamt relativ unflexibel

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

Indirekte Adressierung - Ports

- m:1-Kommunikation
- Ports "gehören" einem Empfänger
  - Kommunikationsendpunkt, der die interne Empfängerstruktur abkapselt
- Ein Objekt kann i.Allg. mehrere Ports besitzen

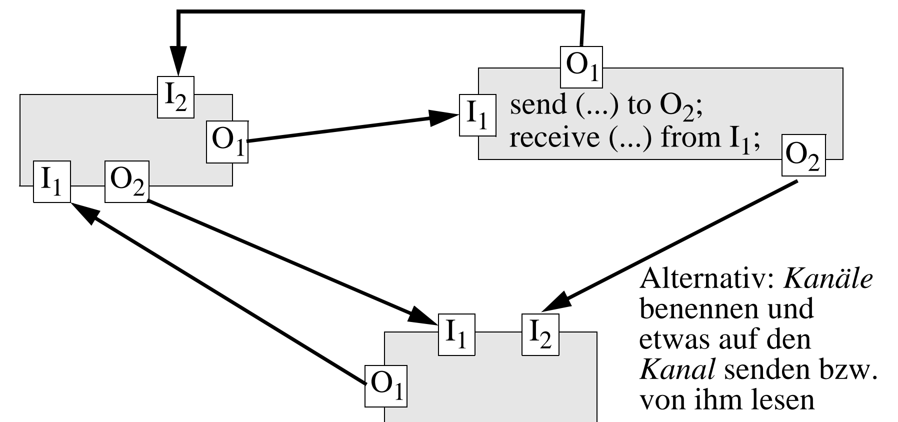


Pragmatische Aspekte (Sprachdesign etc.):

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

Ausgangsports und Kommunikationskanäle

- Neben *Eingangsports* ("In-Port") lassen sich auch *Ausgangsports* ("Out-Port") betrachten



- Ports können als Ausgangspunkte für das Einrichten von *Kanälen* gewählt werden
- Dazu werden je ein In- und Out-Port miteinander verbunden (z.B.: **connect** p1 **to** p2)
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen

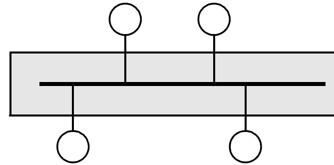
  Konfigurationsphase
- Grössere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
- Evtl. Broadcast-Kanäle: Nachricht geht an alle angeschlossenen Empfänger (z.B. "Event-Bus")

Ereigniskanäle für Software-Komponenten

- Stark entkoppelte Kommunikation
  - Software-Komponenten haben oft getrennte Lebenszyklen
  - Entkoppelung fördert bessere Wiederverwendbarkeit und Wartbarkeit
  - anonym: Sender / Empfänger erfahren nichts über die Identität des anderen
  - Auslösen von “Ereignissen” bei Sendern
  - Reagieren auf Ereignisse bei Empfängern
  - dazwischenliegende “third party objects” können Ereignisse evtl. speichern, filtern, umlenken,...

oder sogar die Existenz

- Ereigniskanal als “Softwarebus”
  - agiert als Zwischeninstanz und verknüpft die Komponenten
  - registriert Interessenten (vgl. LUS)
  - Dispatching eingehender Ereignisse
  - evtl. Pufferung von Ereignisen



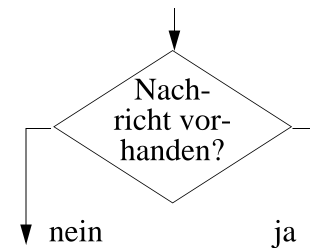
- Probleme
  - Ereignisse können “jederzeit” ausgelöst werden, von Empfängern aber i.Allg. nicht jederzeit entgegengenommen werden
  - falls Komponenten nicht lokal, sondern geographisch verteilt sind, die “üblichen” Probleme: verzögerte Meldung, evtl. verlorene Ereignisse, Reihenfolge (insbes. bei Broadcast / Multicast),...

- Beispiele
  - Microsoft-Komponentenarchitektur (DCOM / ActiveX / OLE / .NET / ...)
  - “Distributed Events” bei JavaBeans und Jini (event generator bzw. remote event listener)
  - event service von CORBA: typisierte und untypisierte Kanäle; Schnittstellen zur Administration von Kanälen; Semantik (z.B. Pufferung des Kanals) jedoch nicht genauer spezifiziert

Nichtblockierendes Empfangen

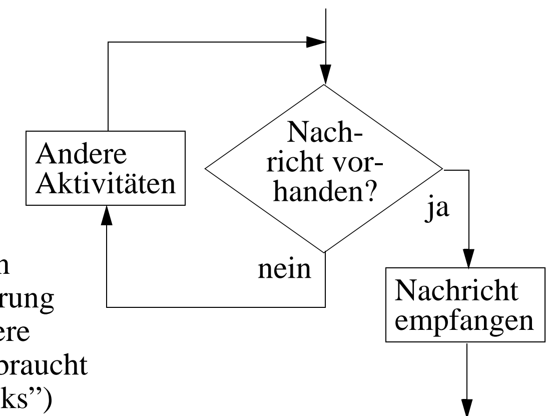
- Typischerweise ist ein “receive” blockierend

- Aber auch *nichtblockierender* Empfang ist denkbar:



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

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



Zeitüberwachte Kommunikation

- *Receive* soll maximal eine gewisse Zeit lang blockieren
    - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat oder der Timeout zugeschlagen hat

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

Quelle vielfältiger Probleme...

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

---

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

Zeitüberwachter Nachrichtenempfang

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

    → "systemnahe", unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

- Sprachliche Einbindung besser z.B. so:

```
receive ... delay t      ← Blockiert maximal t Zeiteinheiten (Vorsicht!)
    {...}
else
    {...}                ← Wird nach mind. t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen
end
```

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

- Frage: Was könnte / sollte "delay 0" bedeuten?