

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

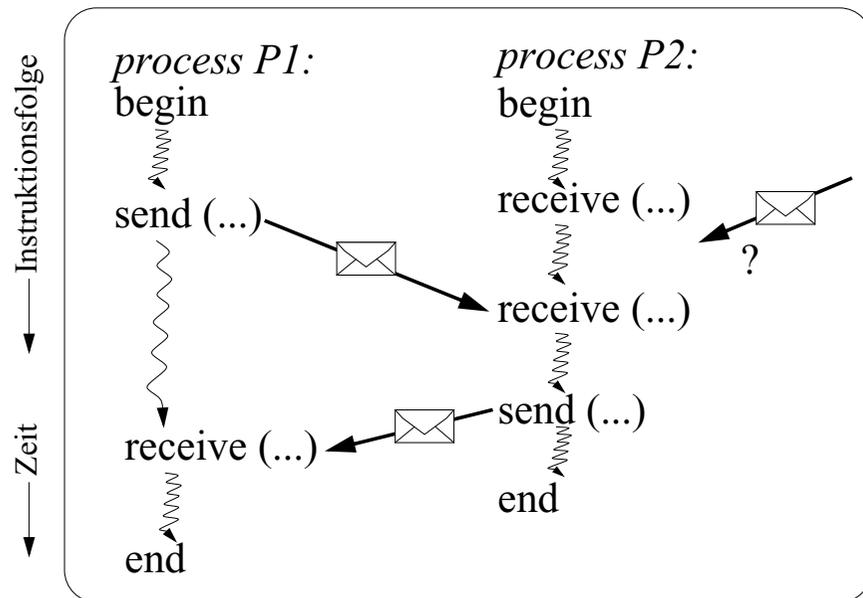
Also trotz Verteiltheit gewisse gemeinsame “Dinge”!

-
- Wir betrachten im folgenden den dritten Punkt genauer
 - Punkte 1) und 2) sind eher Themen einer Vorlesung über “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:
gegründet beim Senden, verbraucht beim Empfangen
- Interprozesskommunikation - naive Sicht:

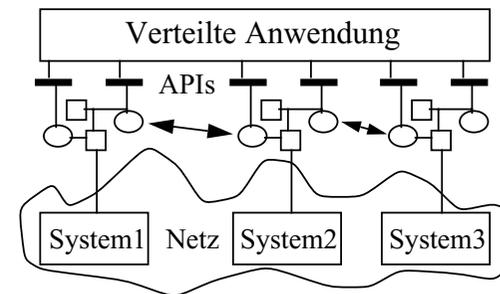


- Welche Kommunikationsanweisungen “matchen”?
- Empfangsbereitschaft, aber keine Nachricht?
- Nachricht, aber keine Empfangsbereitschaft?
- Sprachliche Ausprägung der Kommunikation?
- Wie wird adressiert?

Message Passing System

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

einfache Form von
“Middleware”
(→ später)



- Besteht aus Hilfsprozessen, Pufferobjekten...
- Verbirgt Details des zugrundeliegenden Netzes

- Verwendet vorhandene Protokolle; implementiert ggf. damit neue, “höhere” Protokolle
- Garantiert (je nach “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

⇒ erleichtert Portabilität!

⇒ Vielfältige Aspekte, Varianten, Probleme...

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,...)

Prioritäten von Nachrichten?

- Vgl. dies mit Prioritätsebenen von Unterbrechungen
- *Semantik* zunächst 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 *Rechnernetzen*: bei Rückstaus bei den Routern soll z.B. interaktiver Verkehr bevorzugt werden vor ftp etc.)

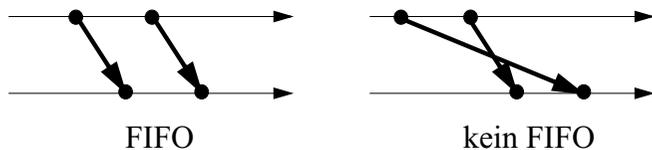
Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!

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

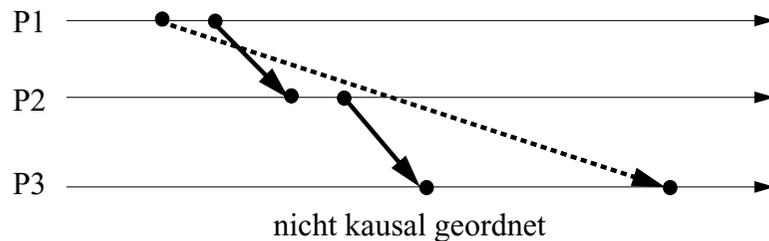
Derartige pragmatische Aspekte müssen in der Praxis neben der eigentlichen Kommunikationssemantik ebenfalls beachtet werden!

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

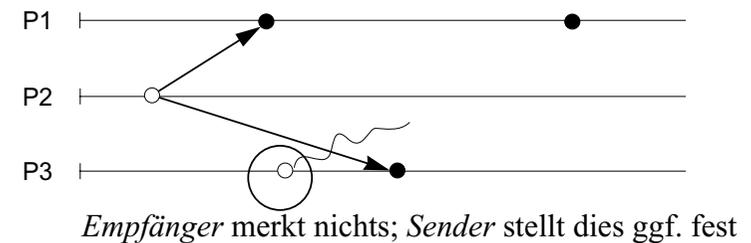


- Möchte man auch dies haben, so muss die Kommunikation "kausal geordnet" sein (Anwendungszweck?)
 - entspricht einer "Globalisierung" von FIFO auf mehrere Prozesse
 - "Dreiecksungleichung": Keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege
 - Denkübung: wie garantiert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

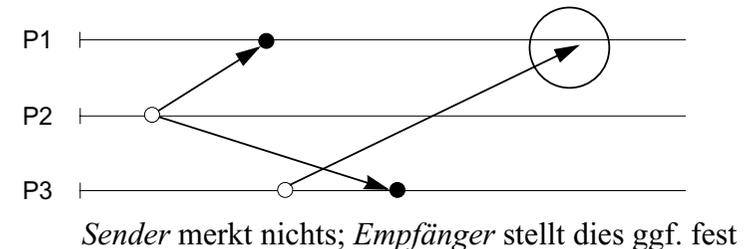
Fehlermodelle (1)

- Fehler sind leider eine Quelle vielfältiger Ärgernisse in verteilten Systemen
- Klassifikation von Fehlermöglichkeiten; Abstraktion von den konkreten, spezifischen Ursachen

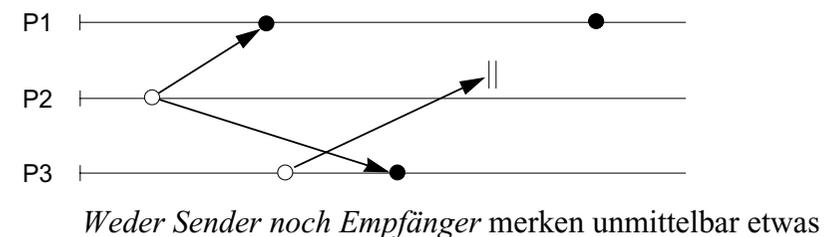
• Fehlerhaftes Senden



• Fehlerhaftes Empfangen

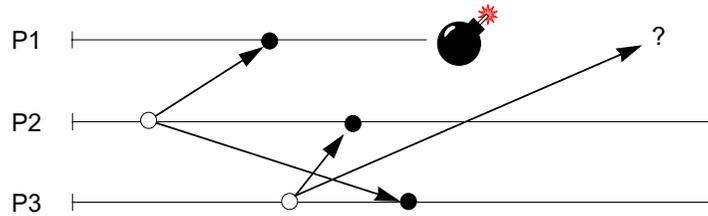


• Fehlerhaftes Übertragen

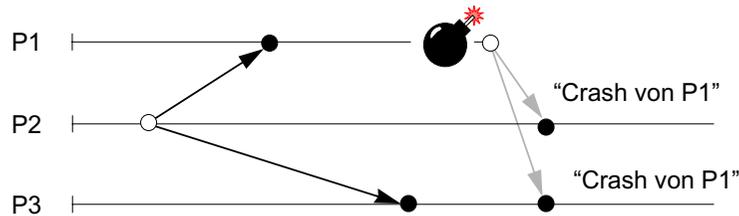


Fehlermodelle (2)

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



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



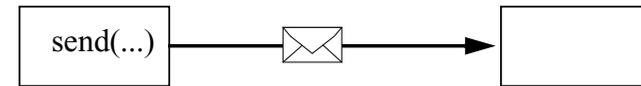
- **Zeitfehler:** Ereignis erscheint zu spät (oder zu früh)
- **Byzantinische Fehler:** Beliebiges Fehlverhalten, z.B.
 - verfälschte Nachrichteninhalte
 - Prozess, der unsinnige Nachrichten sendet
 (solche Fehler lassen sich nur teilweise, z.B. durch *Redundanz*, erkennen)

Fehlertolerante Algorithmen sollen das “richtige” Fehlermodell berücksichtigen!

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

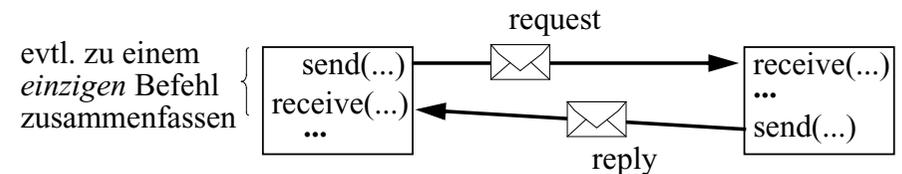
Kommunikationsmuster

Mitteilungsorientiert:

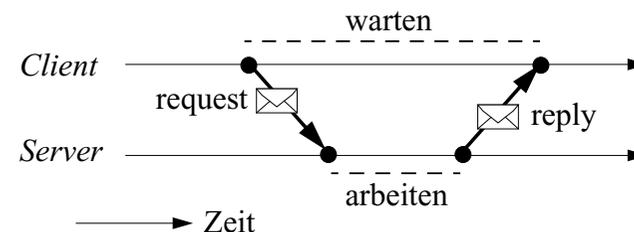


- Unidirektional
- Übermittelte Werte werden der Nachricht in Form von “Ausgabeparametern” beim send übergeben

Auftragsorientiert:

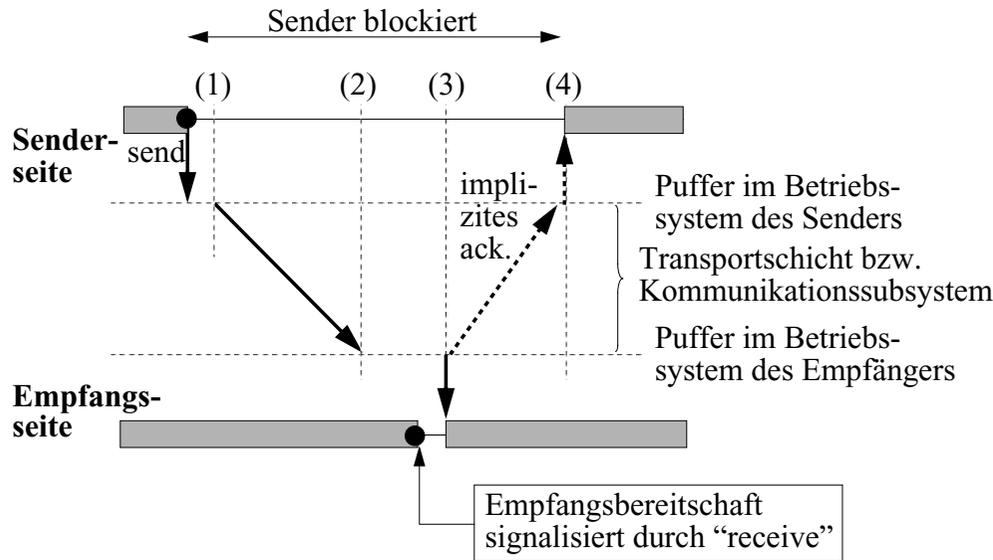


- Bidirektional
- “Antwort” (= Ergebnis eines Auftrags) wird zurückgeschickt



Blockierendes Senden

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



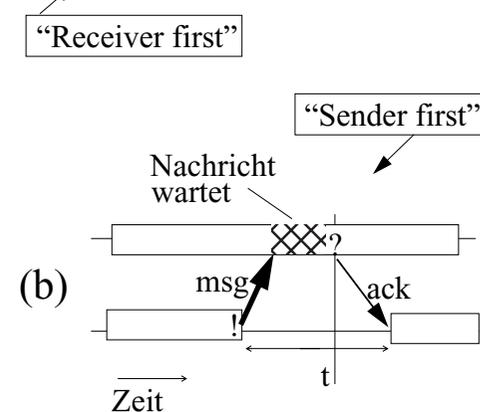
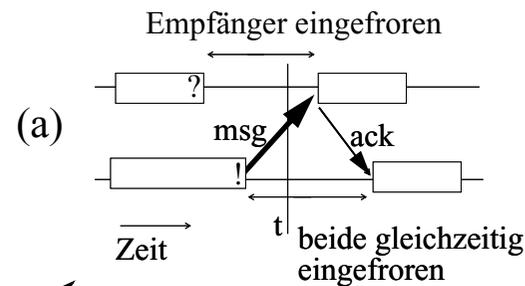
- Verschiedene Ansichten der "korrekten" Definition von "Abschluss der Transaktion" aus Sendersicht:

- *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.)

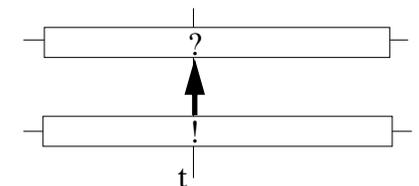
Synchrone Kommunikation

- *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.a. 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!

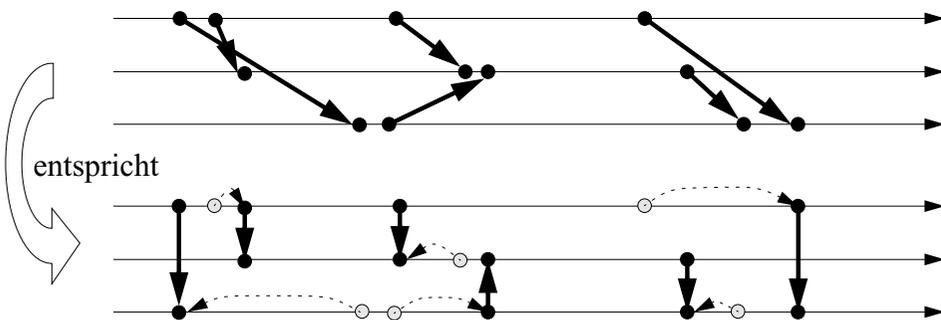
Zeit des Senders steht still → es gibt einen *gemeinsamen Zeitpunkt*, wo die beiden Kommunikationspartner sich treffen.

→ "Rendezvous"

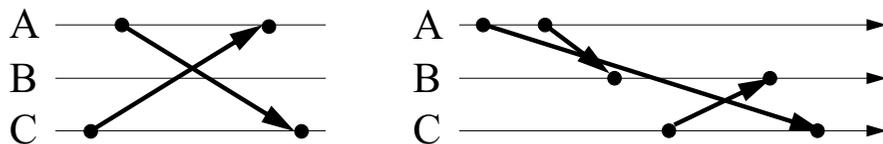
Virtuelle Gleichzeitigkeit

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

- nur stetige Deformation ("Gummiband-Transformation")



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

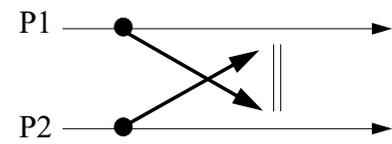


- aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

Blockaden bei synchroner Kommunikation

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

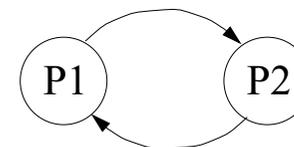
P2:
send (...) to P1;
receive...
...



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

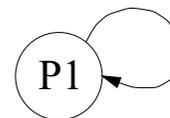
⇒ *Kommunikationsdeadlock!*

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



“Wait-for-Graph”

Genauso tödlich:

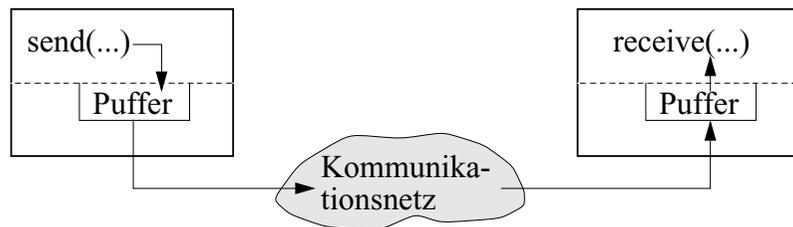


P1:
send (...) to P1;
receive...
...

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

Asynchrone Kommunikation

- *No-wait send*: Sender ist nur (kurz) bis zur Ablieferung der Nachricht an das Transportsystem blockiert (diese kurzzeitigen 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 oft angenehmer in der Anwendung)

- Sender 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)

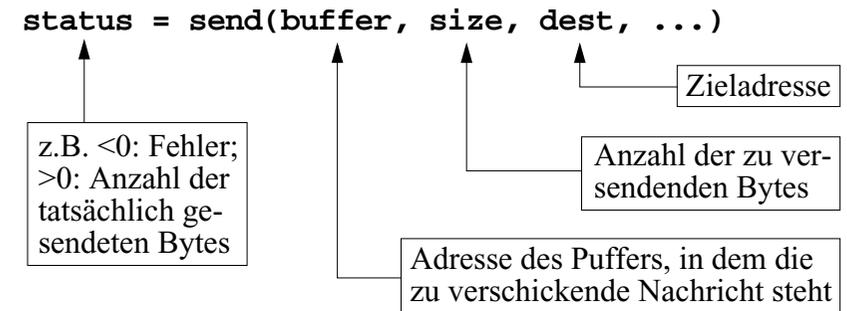
- Sender weiss nicht, ob / wann Nachricht angekommen
- Debugging der Anwendung oft schwierig (wieso?)
- Betriebssystem muss Puffer verwalten (wieviele?)

Sendoperationen 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-Standard; verfügbar auf vielen vernetzten Systemen / Compute-Clustern

- Typischer Aufruf einer solchen Send-Operation:



- 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 beschrieben

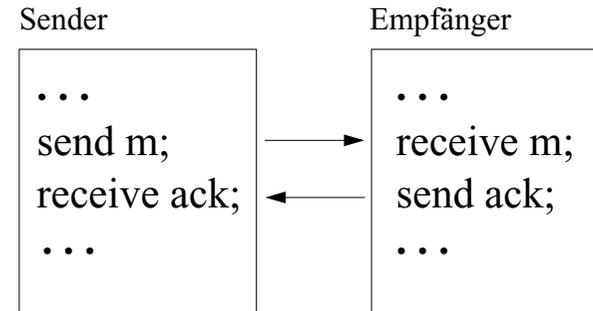
Synchron ? blockierend

- 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)

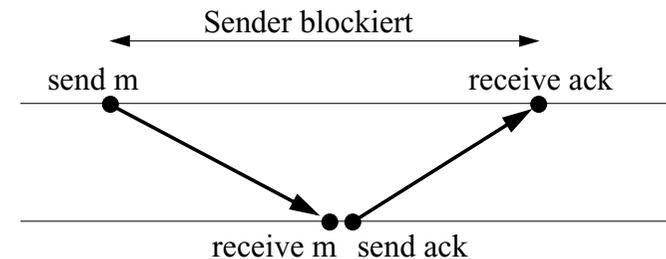
-
- Nicht-blockierende Operationen liefern i.a. einen “handle”
`handle = send(...)`
 - Dieser kann in Test- bzw. Warteoperationen verwendet werden
 - Z.B. Test, ob Send-Operation beendet: `msgdone(handle)`
 - Z.B. warten auf Beendigung der Send-Operation: `msgwait(handle)`
 - Nicht-blockierend ist effizienter aber u.U. unsicherer und umständlicher (ggf. Test, warten) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:

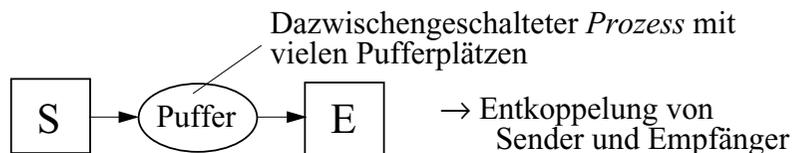


- Warten auf explizites Acknowledgment im Sender direkt nach dem `send` (`receive` ist i.a. blockierend!)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem `receive`

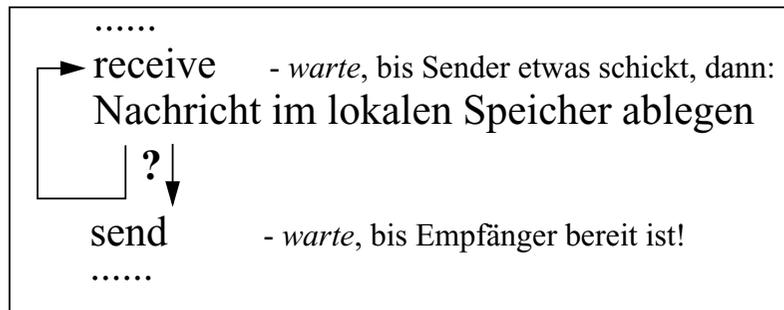


Asynchrone Kommunikation mittels synchroner Kommunikation

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



Wie realisiert man einen Pufferprozess?



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

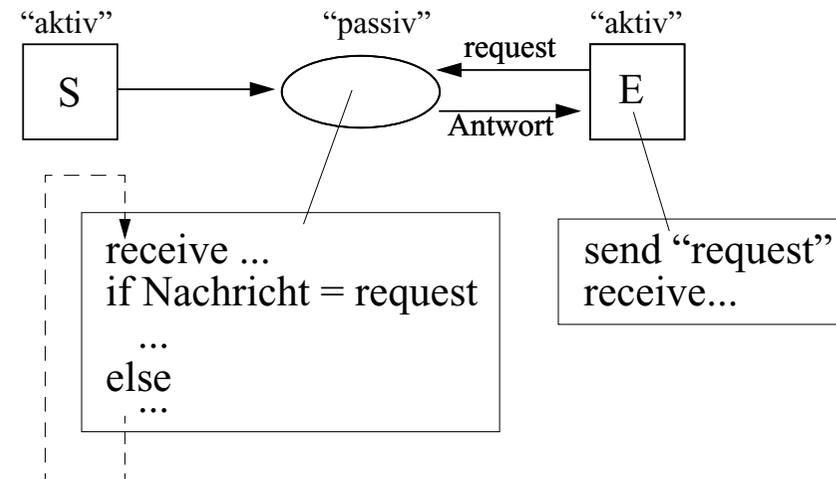
- (1) wieder im receive auf den Sender warten, *oder*
- (2) in einem send auf den Empfänger warten

→ Entweder Sender oder Empfänger könnte unnötigerweise blockiert sein!

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

Inversion der Kommunikationsbeziehung

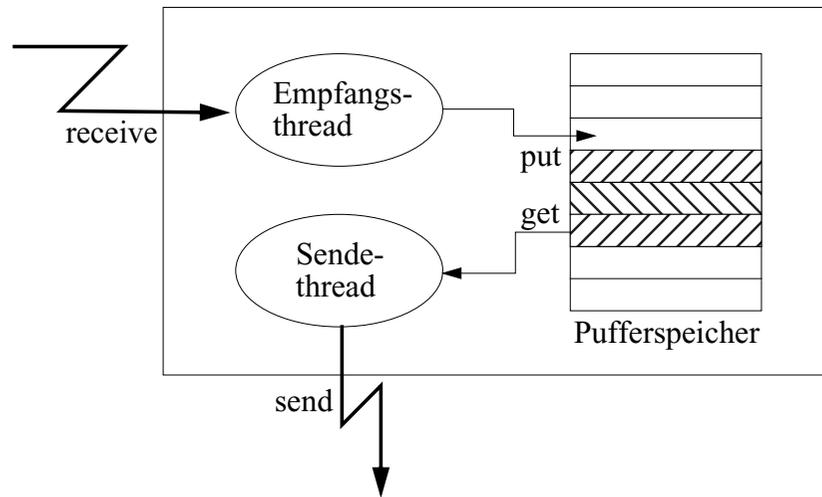
Lösung des zuvor genannten Problems: (Puffer ist ein *Server!*)



- 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 bei Multi-thread-Objekten

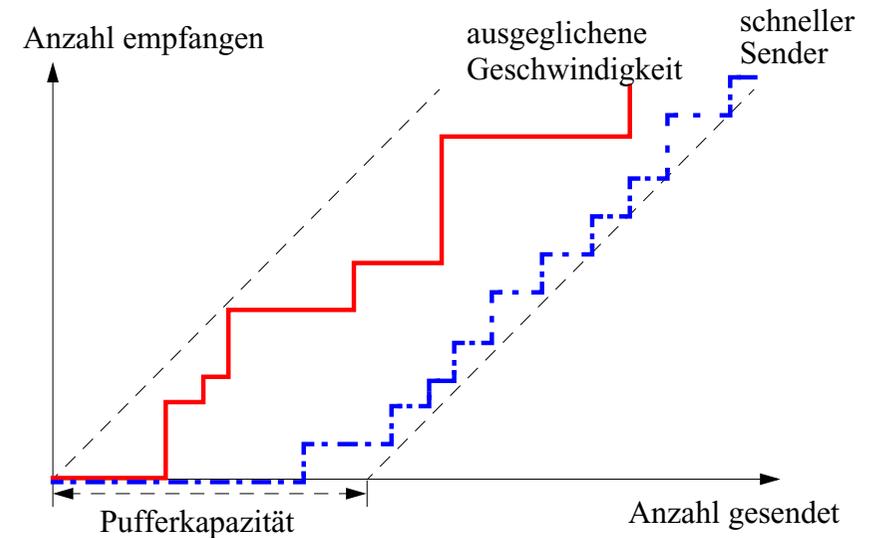
Beachte: Threads (Leichtgewichtsprozesse) greifen auf *gemeinsamen Speicher* zu



- Empfangs-Thread ist (fast) immer empfangsbereit
 - nur kurzzeitig anderweitig beschäftigt (put in lokalen Pufferspeicher)
 - aber: 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 Themen der Betriebssystem-Theorie!

Puffer

- Entkoppelung von Sender und Empfänger durch Puffer



- Anzahl der Pufferplätze bestimmt “Synchronisationsgrad” (entspricht ein “Puffer der Größe 0” der synchronen Kommunikation?)
- Puffer gleicht *Varianz* in der Geschwindigkeit aus, nicht die Geschwindigkeiten selbst!

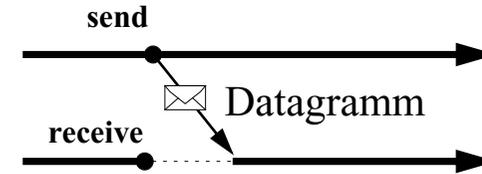
Klassifikation von Kommunikationsmechanismen

“orthogonal” → *Synchronisationsgrad*

Kommunikationsmuster	<i>Synchronisationsgrad</i>	
	asynchron	synchron
Mitteilung	<i>no-wait send</i> (Datagramm)	<i>Rendezvous</i>
Auftrag	“asynchroner RPC”	<i>Remote Procedure Call</i> (RPC)

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 synchronen Kommunikation
- „Überrennen“ des Empfängers bei langen/ häufigen Nachrichten
→ Flusssteuerung notwendig

- Hiervon gibt es diverse Varianten

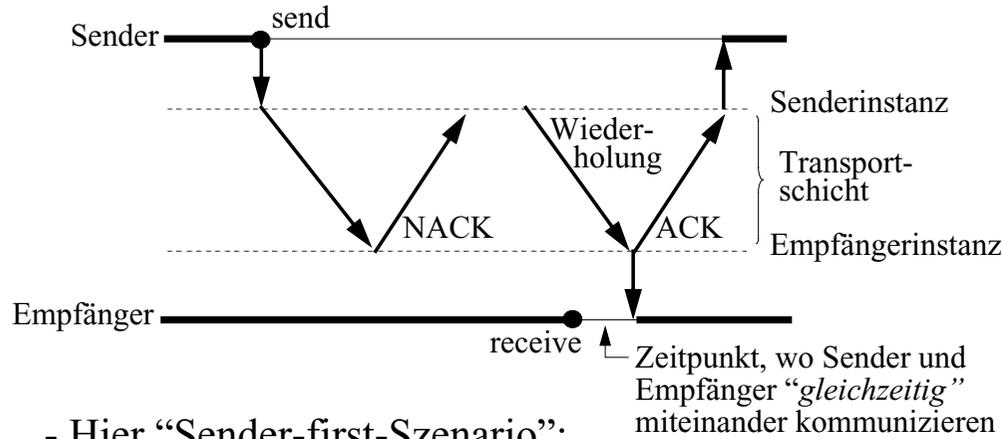
- bei verteilten objektorientierten Systemen z.B.
“Remote Method Invocation” (RMI) statt RPC

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

- Häufigste Anwendung: Mitteilung asynchron,
Auftrag hingegen synchron

Rendezvous-Protokolle

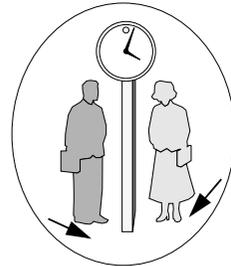
- Synchron-mitteilungsorientierte Kommunikation



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

- "Receiver-first-Szenario" analog

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



- Mit NACK / ACK ist keine Pufferverwaltung nötig!
→ Aufwendiges Protokoll! ("Busy waiting")

- Alternative: Statt NACK: Nachricht auf Empfängerseite puffern

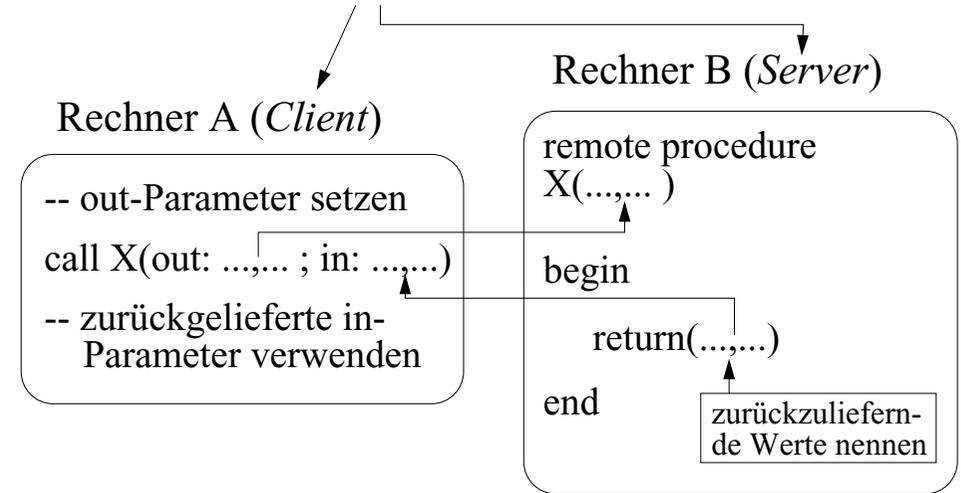
- Alternative: Statt laufendem Wiederholungsversuch: Empfängerinstanz meldet sich bei Senderinstanz, sobald Empfänger bereit

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

Remote Procedure Call (RPC)

- "Entfernter Prozeduraufruf"

- Synchron-auftragsorientiertes 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

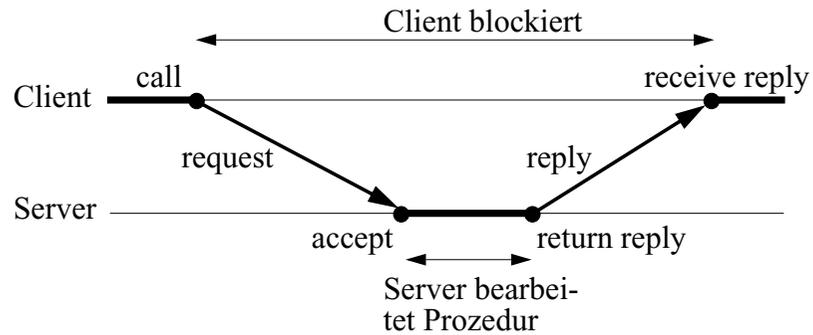
- Kommunikation mit lokalen / entfernten Prozeduren "identisch"

- 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

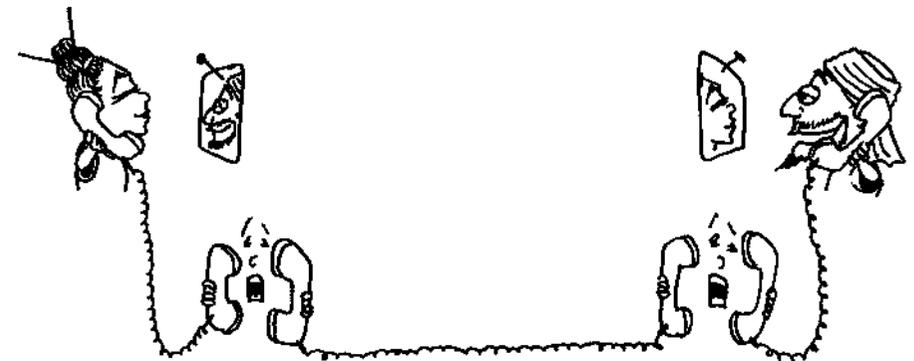
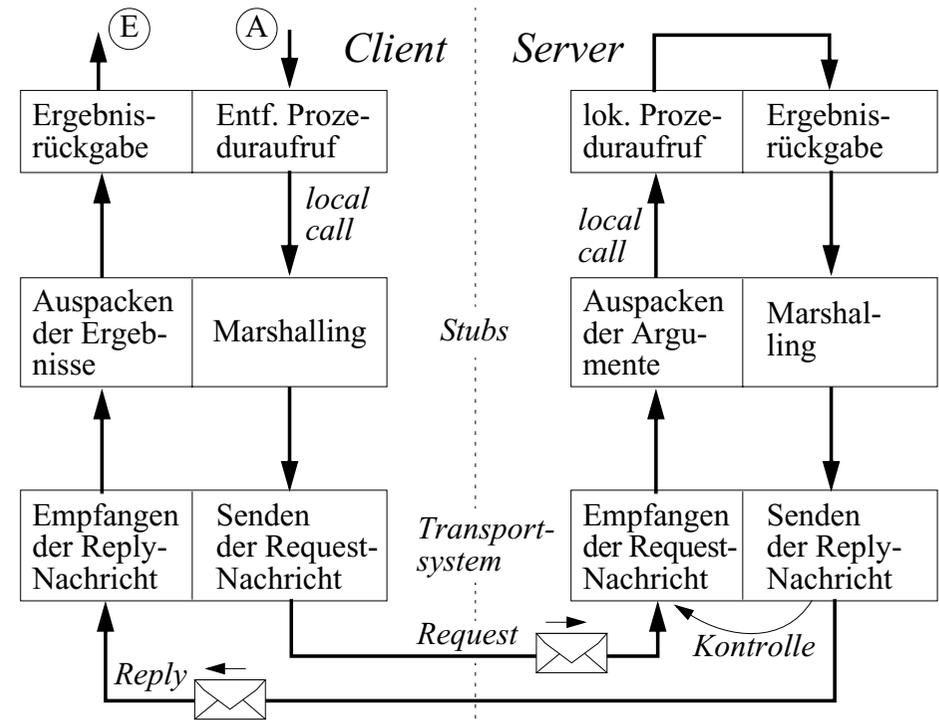
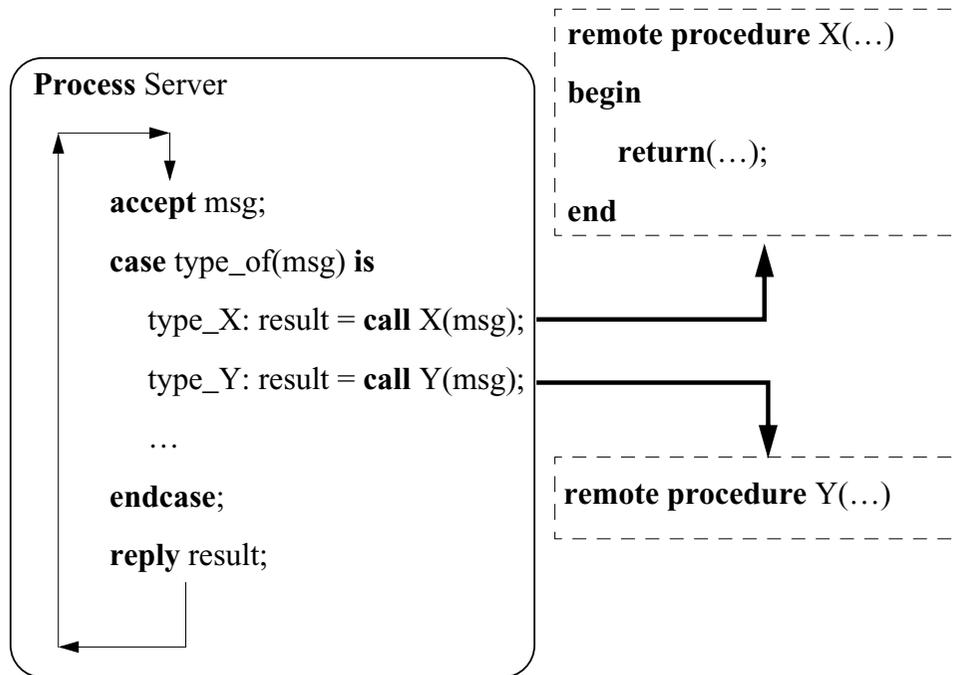


Bild aus dem Buch: "Java ist auch eine Insel" von Christian Ullendorn

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 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 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 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 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*

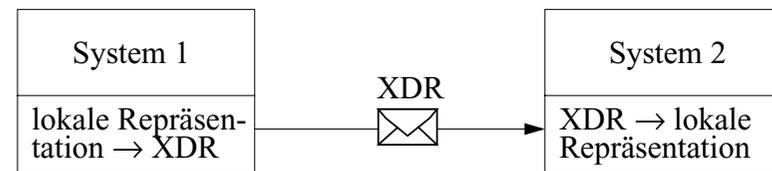
RPC: Marshalling

- Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern
 - evtl. dabei geeignete Codierung (komplexer) Datenstrukturen
 - Glätten (“flattening”) komplexer (evtl. verzeigeter) Datenstrukturen zu einer Sequenz von Basistypen (mit Strukturinformation)
 - umgekehrte Transformation auch als “unmarshalling” bezeichnet
- Problem: RPCs werden oft in *heterogenen* Umgebungen eingesetzt mit unterschiedlicher Repräsentation z.B. von
 - Strings (Längelfeld ↔ ‘\0’)
 - Character (ASCII ↔ Unicode)
 - Arrays (zeilen- ↔ spaltenweise)
 - niedrigstes Bit einer Zahl vorne oder hinten
- Allerdings vorteilhaft: 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!)

RPC: Marshalling

- 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

RPC: Transparenzproblematik

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

bekanntes Programmierparadigma!
- Client- / Serverprozesse haben evtl. unterschiedliche Lebenszyklen: Server könnte 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

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

- Fehlertransparenz

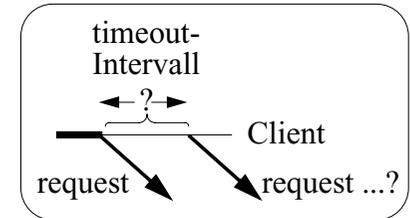
- es gibt mehr Fehlerfälle (beim klassischen Prozeduraufruf gilt: Client = Server → "fail-stop"-Verhalten, also alles oder nix)
- partielle ("einseitige") Systemausfälle: Server-Absturz, Client-Absturz
- Nachrichtenverlust (ununterscheidbar von zu langsamer Nachricht!)
- Anomalien durch Nachrichtenverdopplung (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

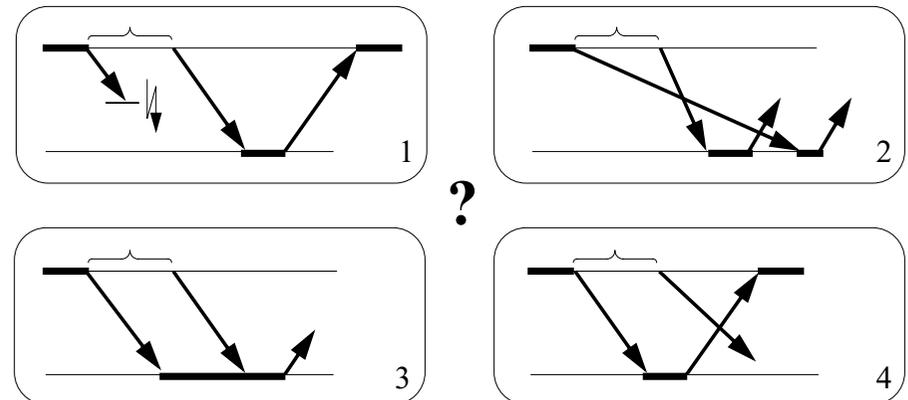
- Gegenmassnahme:

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



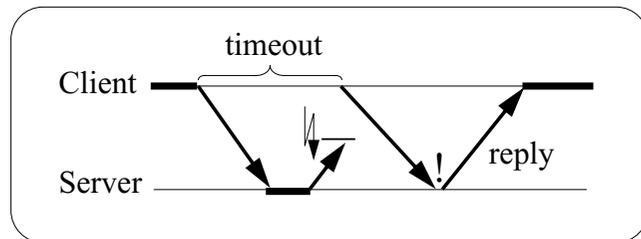
- Probleme:

- Wieviele Wiederholungsversuche maximal?
- Wie gross soll der Timeout sein?
- Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
 - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten Operationen!)
 - Server sollte solche Duplikate erkennen. (Wie? Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet? Genügen Sequenznummern? Zeitmarken?)
 - Würde das Quittieren der Request-Nachricht etwas bringen?



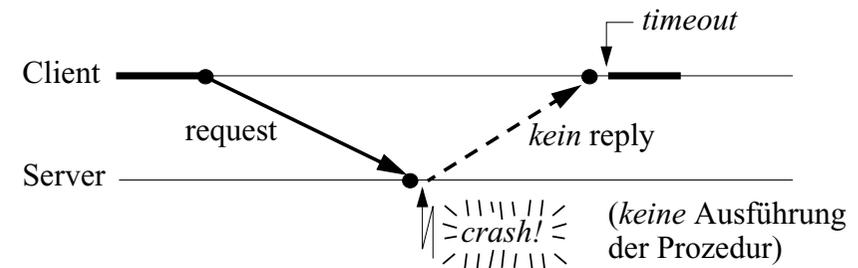
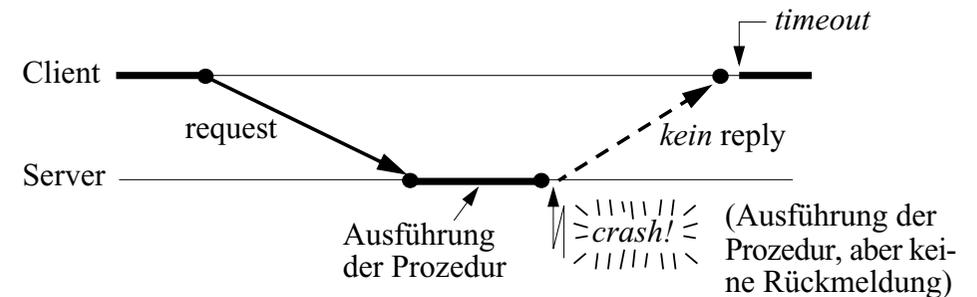
II. Verlorene Reply-Nachricht

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



- *Gegenmassnahme 2*:
 - Server könnte eine "Historie" der versendeten Replies halten
 - Falls Server 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 gelöschttes Reply später dennoch braucht?

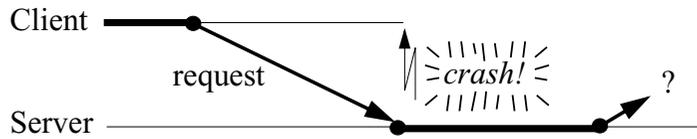
III. Server-Crash



Probleme:

- Wie soll der Client dies unterscheiden?
 - ebenso: Unterschied zu verlorenem request bzw. reply?
 - Sinn und Erfolg konkreter Gegenmassnahmen hängt ggf. 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: "verschmutzter" Zustand durch frühere Inkarnation
 - typischerweise ungenügend Information ("Server Amnesie"), um in alte Kommunikationszustände problemlos wieder einzusteigen

IV. Client-Crash

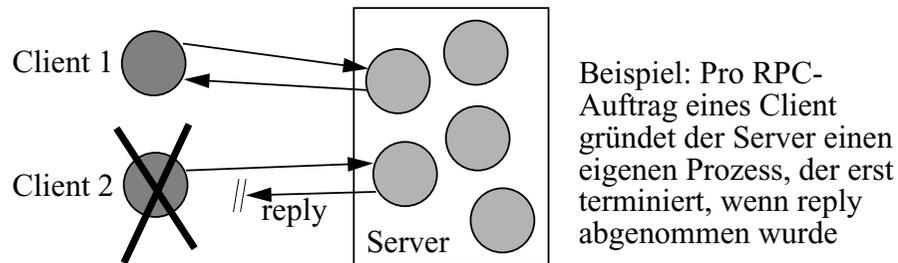


- Reply des Servers wird nicht abgenommen

- Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
- blockiert i.a. Ressourcen beim Server!

- “Orphans” (Waisenkinder) beim Server

- Prozesse, deren Auftraggeber nicht mehr existiert



- Nach Neustart des Client dürfen alte Replies nicht stören

- “Antworten aus dem Nichts” (Gegenmassnahme: Epochen-Zähler)

- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)

- dadurch bleiben aber u.U. locks etc. bestehen
- Orphans könnten bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben...

- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)

RPC-Fehlersemantik

Operationale Sichtweise:

- Wie wird 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)

wird etwas euphemistisch oft als “best effort” bezeichnet

2) At-least-once-Semantik:

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

RPC-Fehlersemantik (2)

3) *At-most-once-Semantik:*

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

4) *Exactly-once-Semantik:*

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

- Nochmals: Fehlertransparenz bei RPC?

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