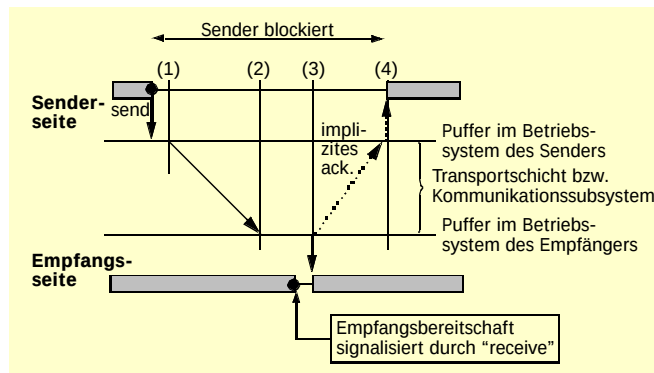


Blockierendes Senden

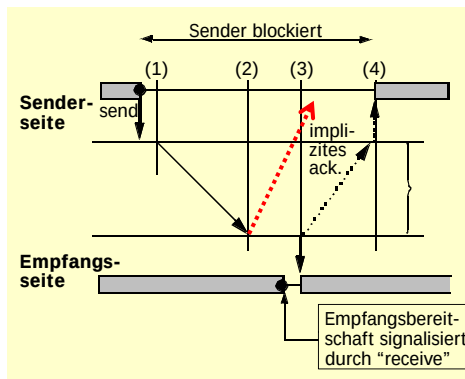
- **Blocking send:** Sender ist bis zum Abschluss der Nachrichtentransaktion blockiert
 - Sender hat eine **Garantie** (Nachricht wurde zugestellt / empfangen)

was genau ist das?



Blockierendes Senden (2)

- Verschiedene Ansichten einer adäquaten Definition von „**Abschluss der Transaktion**“ aus Sendersicht:



Zeitpunkt 4 (automatische Bestätigung, dass der Empfänger 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.)

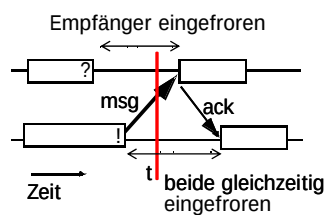
„gleich“ „zeitig“

Synchrone Kommunikation

- **Idealisierung:**
Send und receive geschehen **gleichzeitig**
- Wodurch ist diese Idealisierung gerechtfertigt?
(kann man auch mit einer Marssonde synchron kommunizieren?)

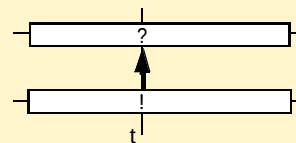
Synchrone Kommunikation mit „blocking send“ implementiert

a) „Receiver first“-Fall:



Bemerkung: „receive“ ist i.Allg. immer **blockierend** (d.h. Empfänger wartet so lange, bis eine Nachricht ankommt)

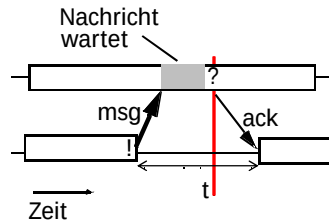
Idealisierung:
senkrechte Pfeile in den Zeitdiagrammen



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

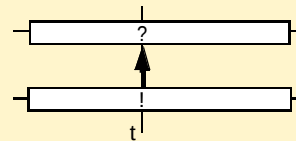
Synchrone Kommunikation mit „blocking send“ implementiert (2)

b) „Sender first“-Fall:



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

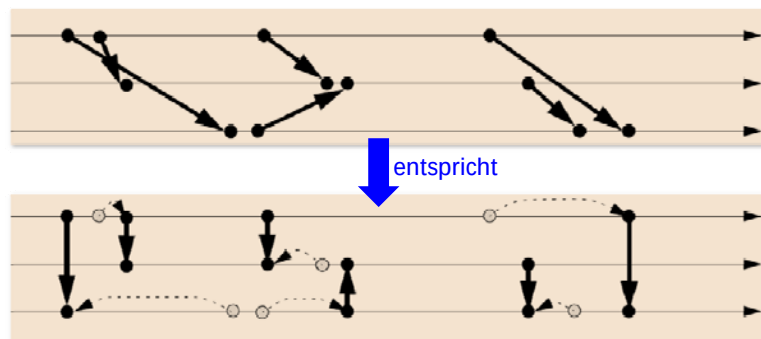
Idealisierung:
senkrechte Pfeile in den Zeitdiagrammen



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

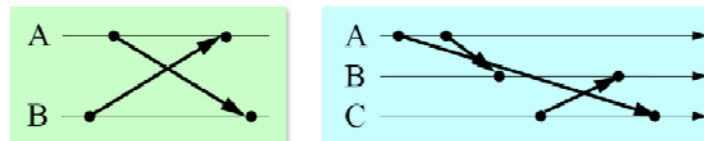
Virtuelle Gleichzeitigkeit

- 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**“)



Virtuelle Gleichzeitigkeit?

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



- Aber was geschieht denn, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies **proviziert** wird?

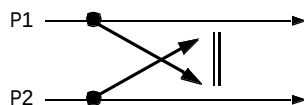
Mehr dazu für besonders Interessierte: B. Charron-Bost, F. Mattern, G. Tel:
Synchronous, Asynchronous and Causally Ordered Communication.
Distributed Computing 9(4), pp. 173-191, www.vs.inf.ethz.ch/publ/

Deadlocks bei synchroner Kommunikation

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

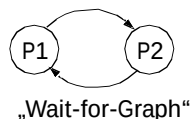
P2:
send (...) to P1;
receive...
...

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



„Wait-for-Graph“

Genauso tödlich:

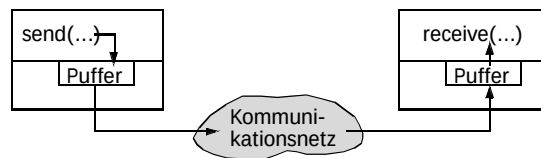


P1:
send (...) to P1;
receive...
...

Gleichnishaft entspricht der *synchronen* Kommunikation das Telefonieren, der *asynchronen* Kommunikation der Briefwechsel

Asynchrone Kommunikation

- **No-wait send**: Sender ist nur (kurz) bis zur lokalen Ab-
lieferung der Nachricht an das Transportsystem blockiert
 - diese kurzzeitige Blockade sollten für die Anwendung transparent sein
- Jedoch i.Allg. länger blockiert, falls das System mo-
mentan keinen **Pufferplatz** für die Nachricht frei hat
 - Alternative: Sendenden Prozess nicht blockieren, aber mittels
„return value“ über Misserfolg des send informieren

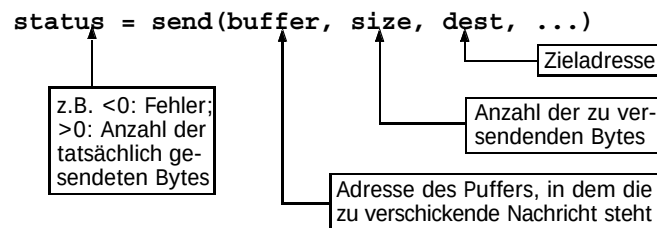


Asynchrone ↔ synchrone Kommunikation

- **Vorteile asynchroner Kommunikation** (im Vgl. zur syn. Komm.):
 - 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**
 - Sender weiss nicht, ob / wann Nachricht angekommen ist
 - 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 diversen Plattformen
- Typischer Aufruf einer solchen Send-Operation:



Sendeoperationen in der Praxis (2)

- Derartige Systeme bieten im Allgemeinen 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 [?] = blockierend

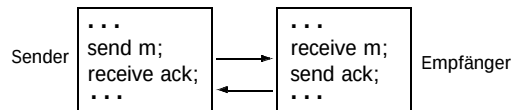
- Kommunikationsbibliotheken machen oft einen Unterschied zwischen **synchronem** und **blockierendem** Senden
■ bzw. analog zwischen asynchron und nicht-blockierend leider etwas verwirrend!
- **Blockierung** ist dann ein rein **senderseitiger** Aspekt
 - **blockierend**: Sender wartet, bis die Nachricht 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-blockierend

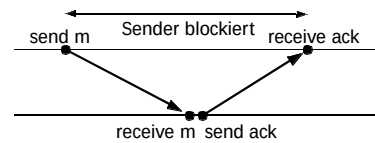
- Nicht-blockierende Operationen liefern oft einen „**handle**“
handle = send(...)
 - dieser kann in Test- bzw. Warteoperationen verwendet werden
 - z.B. **Test**, ob Send-Operation beendet:
msgdone(handle)
 - oder z.B. **warten** auf Beendigung der Send-Operation:
msgwait(handle)
- Nicht-blockierend ist **effizienter** aber evtl. **unsicherer** und **umständlicher** (evtl. Test; warten) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:



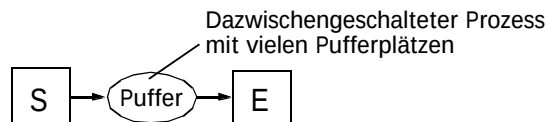
- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive



Dualität der Kommunikationsmodelle (2)

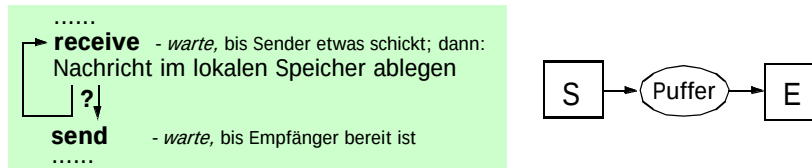
Asynchrone Kommunikation mittels synchroner:

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



→ **Entkoppelung** von Sender und Empfänger

Wie realisiert man einen Pufferprozess bei syn. Kommunikation?

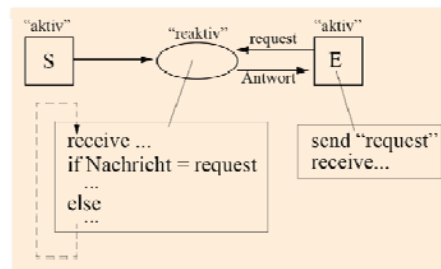


- **Dilemma:** Was tut der Pufferprozess nach dem Ablegen der Nachricht im lokalen Speicher?
 - wieder im receive auf den Sender warten, *oder*
 - in einem (blocking) send auf den Empfänger warten?
- Entweder Sender S oder Empfänger E könnte unnötigerweise **blockiert** sein!

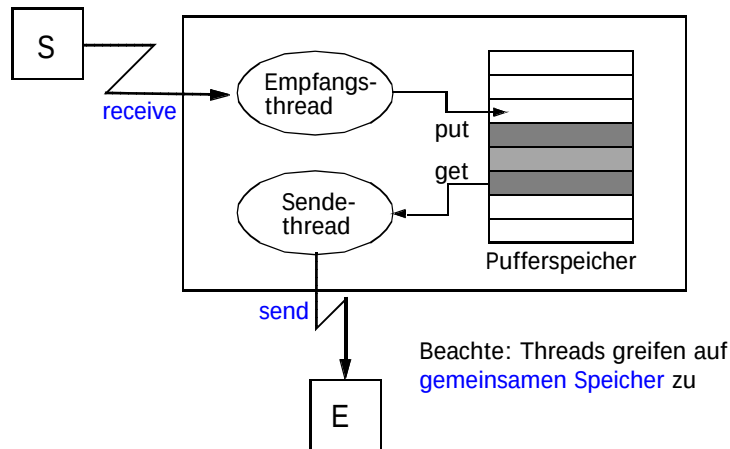
Bemerkung: Puffer der Größe 1 lassen sich so realisieren → Kaskadierung im Prinzip möglich („Pufferpipeline“)

Puffer als Server!

- E schickt „seinem“ Puffer einen „**request**“ (und muss dazu die Adresse des Puffers kennen)
- Puffer schickt E keine Antwort, wenn er **leer** ist
- Empfänger E wird nur dann verzögert, wenn Puffer leer
- Für Sender S ändert sich nichts
- Was tun bei **vollem** Puffer?
 - Dann sollte der Puffer keine Nachricht von S (wohl aber von E!) annehmen (Denkübung: wie programmiert man das?)

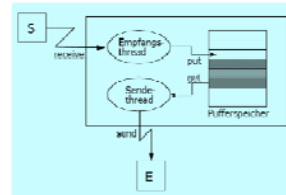


Puffer als Multithread-Objekt



Puffer als Multithread-Objekt (2)

- **Empfangsthread** ist (fast) immer empfangsbereit
 - nur kurzzeitig anderweitig beschäftigt (put in lokalen Pufferspeicher)
 - 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 Betriebssystemtheorie!



Hauptklassifikation von Kommunikationsmechanismen

"orthogonal" → *Synchronisationsgrad*

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

- Häufigste Kombination:
Mitteilung asynchron, Auftrag hingegen synchron

No-Wait Send

- **Asynchron-mitteilungsorientierte** Kommunikation

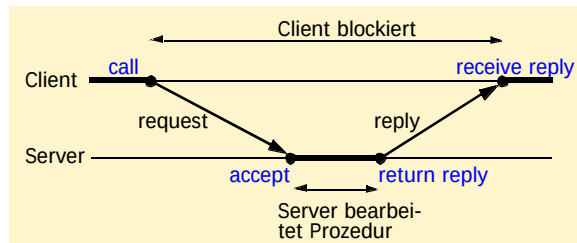


- **Vorteile**
 - weitgehende zeitliche Entkopplung von Sender und Empfänger
 - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- **Nachteile**
 - keine Erfolgsgarantie für den Sender
 - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicher-verwaltung,...) im Unterschied etwa zur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

Remote Procedure Call (RPC)

- Aufruf einer „**entfernten Prozedur**“
- Synchron-auftragsorientiertes Prinzip

Keine Parallelität
zwischen Client
und Server

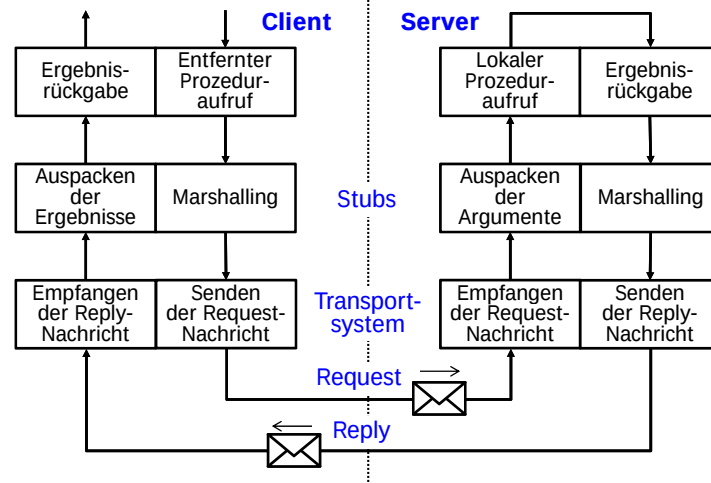


- Bei verteilten objektorientierten Systemen:
„Remote Method Invocation“ (**RMI**), z.B. Java RMI

RPC: Pragmatik

- Soll dem klassischen Prozeduraufruf möglichst gleichen
 - **klare Semantik** für den Anwender (Auftrag als „Unterprogramm“)
- **Einfaches Programmieren**
 - kein Erstellen von Nachrichten, kein Quittieren etc. auf Anwendungsebene
 - Syntax analog zu bekanntem lokalen Prozeduraufruf
 - Verwendung von lokalen / entfernten Prozeduren „identisch“
 - Typsicherheit (Datentypprüfung auf Client- und Serverseite möglich)
- Implementierungsproblem: „**Verteilungstransparenz**“
 - Verteiltheit so gut wie möglich verbergen

RPC: Implementierung



RPC: Stubs

- **Stub** = Stummel, Stumpf

Client:

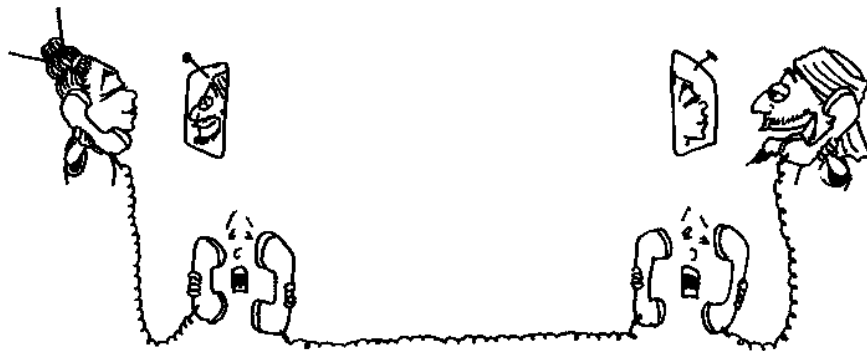
```
xxx ;
call S.X(out: a ; in: b);
xxx ;
```

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

- Parameter a in eine Nachricht packt
- Nachricht an Server S versendet
- Timeout für die Antwort setzt
- Antwort entgegennimmt (oder evtl. exception bei timeout auslöst)
- Ergebnisparameter b mit den Werten der Antwortnachricht setzt

RPC: Stubs (2)

- Wirken als „**proxy**“
 - lokale Stellvertreter des entfernten Gegenübers
- Client-Stub bzw. Server-Stub
- **Simulieren** einen **lokalen Aufruf**
- Sorgen für das **Packen** und **Entpacken** von Nachrichten
- **Konvertieren** Datenrepräsentationen
 - bei heterogenen Umgebungen
- Steuern das Übertragungsprotokoll
 - z.B. zur Behebung von Übertragungsfehlern
- Bestimmen evtl. Zuordnung zwischen Client und Server
 - „binding“



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

RPC: Stubs (3)

- Können oft weitgehend **automatisch generiert** werden
 - z.B. aus dem Client- oder Server-Code sowie evtl. einer **Schnittstellenbeschreibung**
- Nutzen Hilfsroutinen aus Bibliotheken für Formatkonversion usw.
- Nutzen Routinen einer **RPC-Laufzeitumgebung**
 - z.B. zur Kommunikationssteuerung, Fehlerbehandlung etc.
 - werden nicht generiert, sondern dazugebunden

RPC: Kompatibilität von Datenformaten und -strukturen?

- 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?
 - 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

RPC: Marshalling

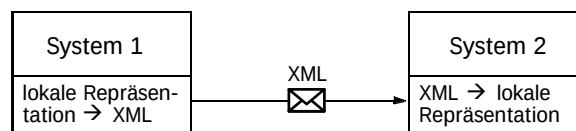
Marshalling: Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern

- evtl. dabei geeignete „standardisierte“ **Codierung** (komplexer) Datenstrukturen
- **Glätten** („flattening“) komplexer (evtl. verzweigter) Datenstrukturen zu einer Sequenz von Basistypen (evtl. mit Strukturinformation)
- umgekehrte Transformation wird oft als „**unmarshalling**“ bezeichnet

RPC: Datenkonversion

1) Umwandlung in eine gemeinsame **Standardrepräsentation**

- z.B. **XML** bei „XML-RPCs“ (in Web-Applikationen)



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

RPC: Datenkonversion (2)

2) Oder sendeseitig 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

Generell: Datenkonversion ist überflüssig, wenn sich alle Kommunikationspartner an einen **gemeinsamen Standard** halten

RPC: Transparenzproblematik

- RPCs sollten so weit wie möglich **lokalen Prozedur-aufrufen** (als bekanntes Programmierparadigma) gleichen, es gibt aber einige subtile **Unterschiede**
 - z.B. bei Leistungs-, Orts- und Fehlertransparenz
- Beachte auch: Client-/Serverprozesse haben evtl. **unterschiedliche Lebenszyklen**
 - Server könnte z.B. noch nicht oder nicht mehr oder in einer „falschen“ (z.B. veralteten) Version existieren
 - Anwender oder Programmierer eines Clients hat typischerweise keine Kontrolle über den Server

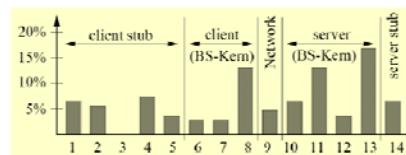
RPC: Leistungstransparenz?

- RPC i.Allg. wesentlich **langsamer** als lokaler Prozeduraufruf
- **Netzbandbreite** ist bei umfangreichen Datenmengen relevant
- Oft ungewisse, variable **Verzögerungen**

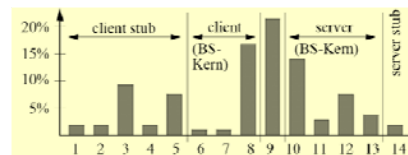
Effizienzanalyse eines RPC-Protokolls

(zitiert nach
A. Tanenbaum)

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



1440 Byte-Nutznachricht
(ebenfalls keine Auftragsbearbeitung)



- | | | |
|-------------------------|----------------------------------|----------------------------------|
| 1. Call stub | 6. Trap to kernel | 10. Get packet from controller |
| 2. Get message buffer | 7. Queue packet for transmission | 11. Interrupt service routine |
| 3. Marshal parameters | 8. Move packet to controller | 12. Compute UDP checksum |
| 4. Fill in headers | 9. Network transmission time | 13. Context switch to user space |
| 5. Compute UDP checksum | | 14. Server stub code |

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

RPC: Ortstransparenz?

- Evtl. muss Server („Zielort“) bei **Adressierung** explizit genannt werden
- Erkennbare **Trennung der Adressräume** von Client und Server
 - keine Kommunikation über **globale Variablen** möglich
 - typischerweise keine **Pointer** / **Referenzparameter** als Parameter möglich

RPC: Fehlertransparenz?

- Es gibt **mehr Fehlerfälle**
 - beim klassischen Prozeduraufruf gilt: Client = Server → „alles oder nichts“
 - hier: partielle („einseitige“) Systemausfälle typisch (Server-Absturz, Client-Absturz)
- **Nachrichtenverlust**
 - ununterscheidbar von zu langsamer Nachricht!
- Client/Server haben zumindest zwischenzeitlich eine **unterschiedliche Sicht** des Zustandes einer „RPC-Transaktion“
 - crash kann im „ungünstigen Moment“ des RPC-Protokolls erfolgen

⇒ Fehlerproblematik ist also „kompliziert“!

Typische RPC-Fehlerursachen

Wir besprechen nachfolgend 4 **typische Fehlerursachen**:

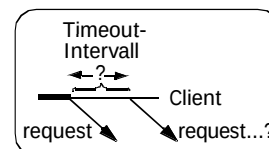
1. Verlorene Request-Nachricht
2. Verlorene Reply-Nachricht
3. Server-Crash
4. Client-Crash

Grundprobleme:

- Client / Server haben **temporär** eine **inkonsistente Sicht**
- **Timeout** beim Client kann **verschiedene Ursachen** haben (verlorener Request, verlorenes Reply, langsamer Request bzw. Reply, langsamer Server, abgestürzter Server,...) → Fehlermaskierung schwierig

Verlorene Request-Nachricht

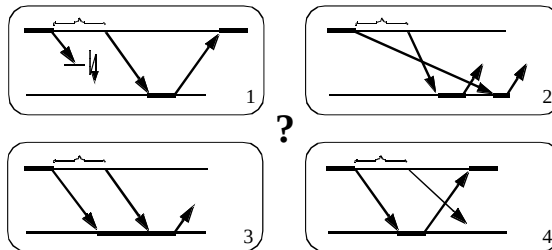
- **Gegenmassnahme:**
Nach Timeout (kein Reply) die Request-Nachricht erneut senden



- **Probleme:**
 - Wie viele Wiederholungsversuche maximal?
 - Wie gross soll der Timeout sein?
 - Was, wenn die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam?

Verlorene Request-Nachricht (2)

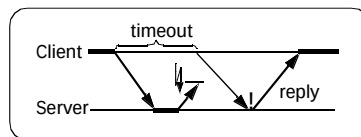
- **Probleme**, wenn Nachricht tatsächlich gar nicht verloren:
 - Doppelte Request-Nachricht!
(Gefährlich bei nicht-idempotenten serverseitigen Operationen!)
 - Server sollte solche Duplikate erkennen (Denkübung: Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet? Genügen Sequenznummern? Zeitmarken?)
 - Würde das Quittieren der Request-Nachricht etwas bringen?



Bei automatischer Wiederholung von Request-Nachrichten ist eine **Vielzahl möglicher Fälle** zu unterscheiden

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!

Verlorene Reply-Nachricht (2)

- **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 **Speicherplatzprobleme:**
→ Historie nach „einiger“ Zeit löschen bzw. kürzen
 - und wenn man ein gelöscht Reply später dennoch braucht?
 - ist in diesem Zusammenhang ein ack eines Reply sinnvoll?

Resümee (3a)

- **Synchrone Kommunikation und blocking send**
 - Realisierung synchroner Kommunikation
 - Kommunikationsdeadlocks
 - virtuelle Gleichzeitigkeit („Gummibandtransformation“)
- **Kommunikationsmuster**
 - synchron ↔ asynchron
 - Vor- / Nachteile asynchroner bzw. synchroner Kommunikation
- **Synchrone Kommunikation mit asynchroner simulieren**
 - Warten auf ein explizites Acknowledgement
- **Asynchrone Kommunikation mit synchroner simulieren**
 - Pufferprozess zur Entkoppelung dazwischenschalten

Resümee (3b)



- Implementierung von Pufferprozessen
 - z.B. durch Multithreading
- Rendezvous-Protokoll (syn., mitteilungsorientiert)
- RPC-Implementierung
 - Stubs
 - Parameter-Marshalling
 - Transparenzproblematik
 - RPC-Effizienz
- RPC-Fehlerproblematik
 - Fehlerursachen (verlorene Nachrichten, Crash von Server / Client)
 - Gegenmassnahmen