

Synchron [?] = blockierend

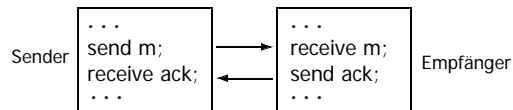
- Kommunikationsbibliotheken machen oft einen Unterschied zwischen **synchronem** und **blockierendem** Senden leider etwas verwirrend!
 - bzw. analog zwischen **asynchron** und **nicht-blockierend**
- **Blockierung** ist dann ein rein **senderseitiger** Aspekt
 - **blockierend**: Sender wartet, bis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

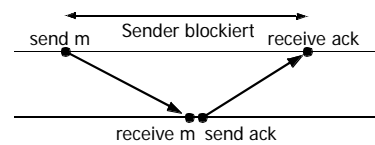
- Nicht-blockierende Operationen liefern oft einen „**handle**“
 - **handle = send(...)**
 - dieser kann in Test- bzw. Warteoperationen verwendet werden
 - z.B. **Test**, ob Send-Operation beendet:
`if msgdone(handle)...`
 - oder z.B. **warten** auf Beendigung der Send-Operation:
`msgwait(handle)`
-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

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:



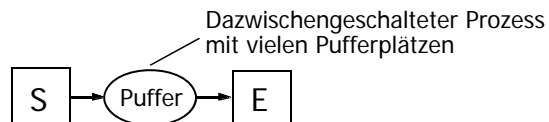
- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive



Dualität der Kommunikationsmodelle (2)

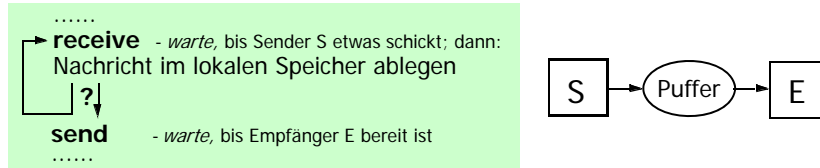
Asynchrone Kommunikation mittels synchroner:

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



→ Entkoppelung von Sender und Empfänger

Wie realisiert man einen Pufferprozess bei syn. Kommunikation?

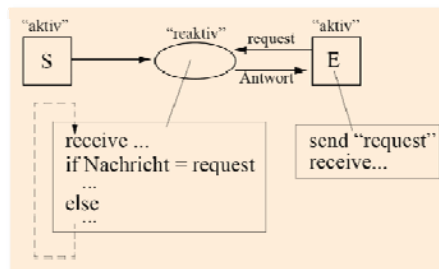


- **Dilemma**: Was tut der Pufferprozess nach dem Ablegen der Nachricht im lokalen Speicher?
 - wieder im receive auf den Sender warten, *oder*
 - in einem (blocking) send auf den Empfänger warten?
- Entweder Sender S oder Empfänger E könnte unnötigerweise **blockiert** sein!

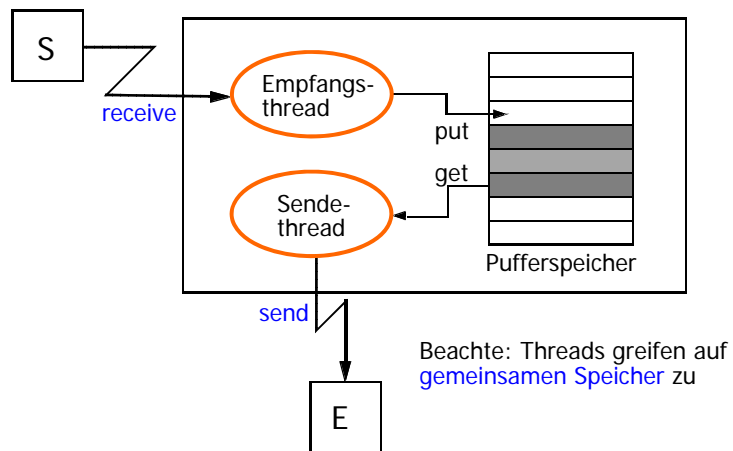
Bemerkung: **Puffer der Größe 1** lassen sich so realisieren → Kaskadierung im Prinzip möglich („Pufferpipeline“)

1. Lösung: 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?)

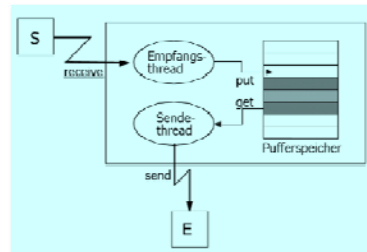


2. Lösung: Puffer als Multithread-Objekt



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 ist i.Allg. zyklisch organisiert (→ FIFO)
- Pufferspeicher liegt im gemeinsamen Adressraum
- ⇒ **Synchronisation** der beiden Threads notwendig!
 - konkurrenentes Programmieren
 - klassische Themen der Betriebssystemtheorie



Hauptklassifikation von Kommunikationsmechanismen

"orthogonal" → *Synchronisationsgrad*

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

- Häufigste Kombination: Mitteilung asynchron, Auftrag hingegen synchron

No-Wait Send

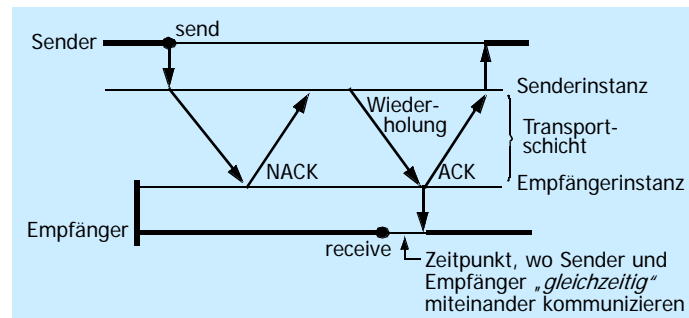
- **Asynchron-mitteilungsorientierte** Kommunikation



- **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
 - Gefahr des „Überrennens“ des Empfängers bei zu häufigen Nachrichten → Flusssteuerung („flow control“) notwendig

Rendezvous-Protokolle

- Synchron-mitteilungsorientierte Kommunikation



- Hier beispielhaft „Sender-first-Szenario“:
Sender wartet als Erster („Receiver-first-Szenario“ analog)

Rendezvous-Protokolle (2)

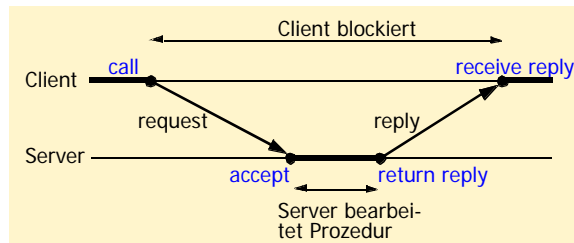
- **Rendezvous**: Der erste wartet auf den anderen („Synchronisationspunkt“)
- Mit **NACK / ACK** (vgl. Bild) sind weniger Puffer nötig → aber aufwändiges Protokoll („busy waiting“)
 - Alternative 1: Statt NACK („negative ACK“): Nachricht auf Empfängerseite puffern (Nachteil: Platzbedarf für Puffer)
 - Alternative 2: Statt laufendem Wiederholungsversuch: Empfängerinstanz 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-auftragsorientiertes Prinzip

Keine Parallelität
zwischen Client
und Server



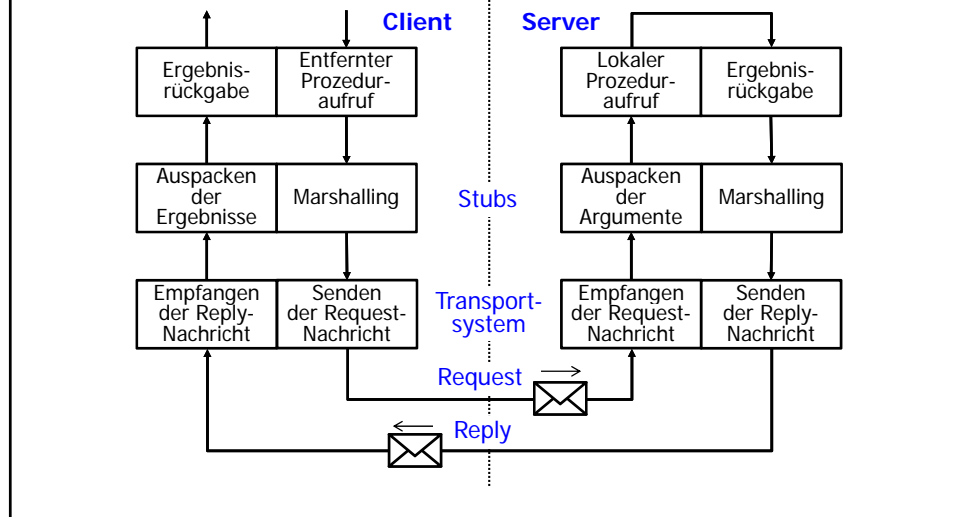
accept, return
reply, receive
reply etc. werden
„unsichtbar“
durch Compiler
bzw. Laufzeit-
system erledigt

- 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 lokaler / entfernter Prozeduren analog
 - Typsicherheit (Datentypprüfung auf Client- und Serverseite möglich)
- Implementierungsproblem: „Verteilungstransparenz“
 - Verteiltheit so gut wie möglich verbergen

RPC: Implementierung



RPC: Stubs

- **Stub** = Stummel, Stumpf

Client:

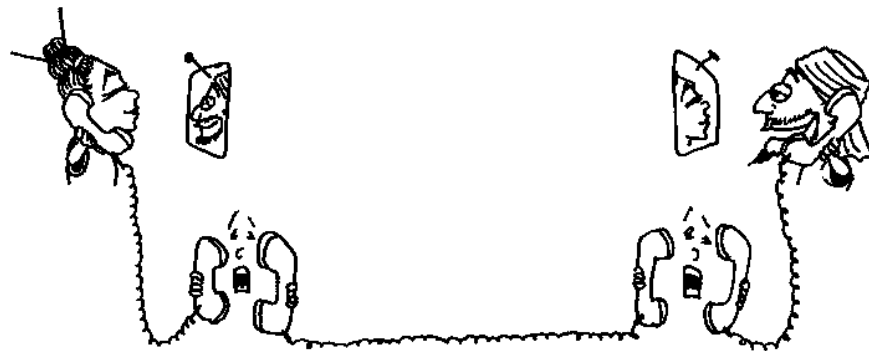
```
xxx ;
call S.X(out: a ; in: b);
xxx ;
```

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 **Zusammenbau** und **Entpacken** von Nachrichten
- **Konvertieren** Datenrepräsentationen
 - bei heterogenen Umgebungen
- Können oft weitgehend **automatisch generiert** werden
 - z.B. aus dem Client- oder Server-Code sowie evtl. einer **Schnittstellenbeschreibung**
- Steuern das **Übertragungsprotokoll**
 - z.B. zur Behebung von Übertragungsfehlern



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

RPC: Kompatibilität von Datenformaten und -strukturen?

- Problem: Parameter **komplexer 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 (Längelfeld ↔ '\0' als Endekennung)
 - Character (ASCII ↔ Unicode)
 - Arrays (zeilen- ↔ spaltenweise)
 - Zahlen (niedrigstes Bit vorne oder hinten)

RPC: Marshalling

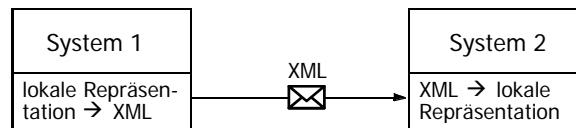
Marshalling: Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern

- evtl. dabei geeignete „standardisierte“ **Codierung** (komplexer) Datenstrukturen
- **Glätten** („flattening“) komplexer (evtl. verzeigert) 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 Dekodieringsroutinen 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 Prozeduraufrufen** (als bekanntes Programmierparadigma) gleichen, es gibt aber einige subtile **Unterschiede**
 - z.B. bei Nichterreichbarkeit oder Absturz des Servers; RPCs dauern auch länger als lokale Prozeduraufrufe
- 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

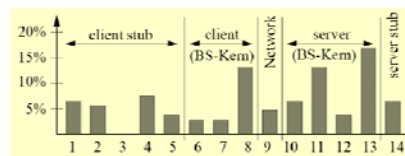
RPC: Leistungstransparenz?

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

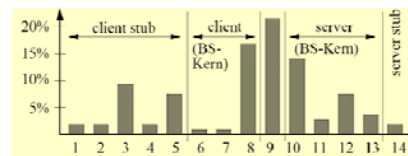
Effizienzanalyse eines RPC-Protokolls

(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 over the bus | 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 nicht 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
- **Getrennte 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 Zustands einer „RPC-Transaktion“
 - crash kann im „ungünstigsten Moment“ des RPC-Protokolls erfolgen

⇒ Fehlerproblematik ist also „kompliziert“!

Typische RPC-Fehlerursachen

Wir besprechen nachfolgend 4 **typische Fehlerursachen**:

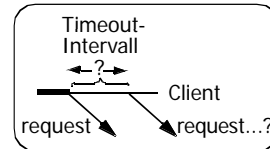
1. Verlorene Request-Nachricht
2. Verlorene Reply-Nachricht
3. Server-Crash
4. Client-Crash

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

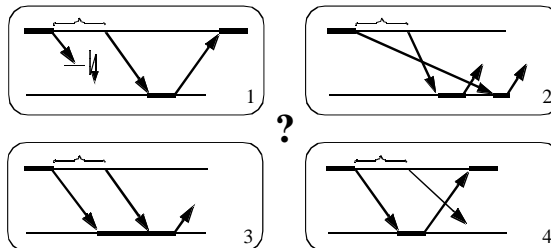
- **Gegenmassnahme:**
Nach Timeout (kein Reply) die Request-Nachricht erneut senden



- **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)

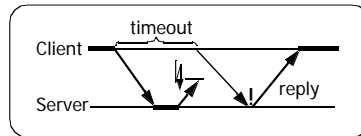
- **Probleme**, wenn Nachricht tatsächlich gar nicht verloren:
 - Doppelte Request-Nachricht!
(Gefährlich bei nicht-idempotenten serverseitigen Operationen!)
 - Server sollte solche Duplikate erkennen (Denkübung: Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet?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

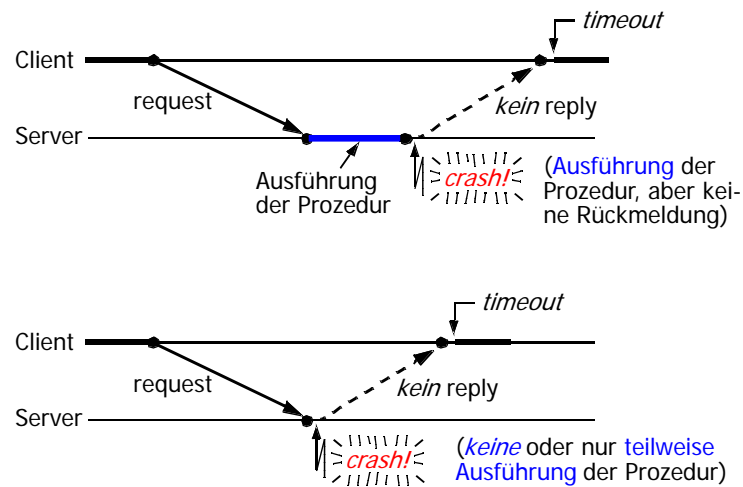


- **Probleme:**
 - vielleicht ging aber tatsächlich der **Request verloren?**
 - oder der **Server** war nur **langsam** und arbeitet noch?
 - ist aus Sicht des Clients nicht unterscheidbar!

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?

Server-Crash

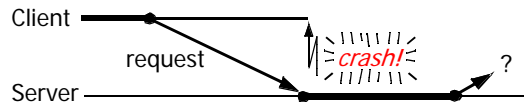


Server-Crash (2)

- **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: „verschmutzter“ Zustand durch frühere Inkarnation
 - typischerweise ungenügend Information, um in alte Transaktionszustände problemlos wieder einzusteigen

Client-Crash

- Oder auch:
Client mittlerweile nicht mehr am reply interessiert



- Reply des Servers wird **nicht abgenommen**
 - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
 - blockiert i.Allg. Ressourcen beim Server!

Client-Crash (2)

- Problem: „Orphans“ (Waisenkinder) beim Server
 - Prozesse, deren Auftraggeber nicht mehr existiert
- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)
 - Orphans könnten aber bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben,...
- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)
- „Sehr“ alte Prozesse, die für einen Auftrag gegründet wurden, werden als Orphans angesehen und terminiert

Resümee (3a)

- **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