

K Kommunikationsmodelle

Kommunikationsmodelle auf der Basis 'Nachrichtenaustausch ohne gemeinsamen verteilten Speicher'

Fokus hier nun: Programme und Prozesse, die untereinander Informationen austauschen, nicht mehr Rechner bzw. Netzwerkkarten.

- Begriffe:
- Sender und Empfänger
 - Erzeuger und Verbraucher
 - Klient und Server
 - gleichberechtigte Teilnehmer (engl.: Peer)

Kriterium	Optionen	
Adressierung	direkt	indirekt
Blockierung	synchron	asynchron
Pufferung	ungepuffert	gepuffert, Mailbox
Kommunikationsform	meldungsorientiert	auftragsorientiert

Bild K.1 Kommunikationsmodelle

K.1 Adressierung

K.1.1 Direkte Adressierung

- *Symmetrisch* $\leftrightarrow$ gleiche Mechanismen wenn Sender und Empfänger die Rollen tauschen

P Empfangsadresse, Q Senderadresse, message

Dienstprimitive:

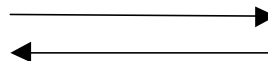
sendtoP (message); receiveFromQ (message);

z.B.:

Maschine A

```
Prozeß 4: sendtoMaschineB+Prozeß1(Nutzdaten)
         receiveFromMaschineB+Prozeß1(Nutzdaten)
....
Prozeß 5
Prozeß 6
....
```

z.B. serielle Ltg.



Maschine B

```
Prozeß 1: receiveFromMaschineA+Prozeß4(Nutzdaten)
         sendtoMaschineA+Prozeß4(Nutzdaten)
....
Prozeß 5
Prozeß 6
....
```

- *Asymmetrisch* $\leftrightarrow$ *Gateway zwischen Sender und Empfänger*

idS := Adresse des eigentlichen (entfernten) **Senders**,
S:= Empfangsadresse **Server** (Router, Gateway), der **lokal mehrere Empfänger bedient**:
 idE := Adresse des eigentlichen **Empfängers** im lokalen Zugriff des o.g. **Servers**,

Sender baut folgende **MESSAGE** := { idE , **SUBMESSAGE** } auf.
SUBMESSAGE:= { idS , Nutzdaten }

1. Weg: Sender => Server => Empfänger:

1. **Sender:** send (**S**, **MESSAGE**);
2. **Server:** receive (**MESSAGE**);
3. **Server:** idE aus der **MESSAGE**; sendlokal(idE , **SUBMESSAGE**);
4. **Empfänger:** receivelokal (**SUBMESSAGE**);

2. Rückweg: Empfänger => Server => Sender:

1. **alter Empfänger wird zum Sender:** sendlokal (**S**, **MESSAGE**);
2. **Server:** receivelokal (**MESSAGE**);
3. **Server:** idE aus der **MESSAGE**; send(idE , **SUBMESSAGE**);
4. **alter Sender wird zum Empfänger:** receive (**SUBMESSAGE**);

- Der Server stellt anhand der idE die Nachricht lokal zu. Er kann so mehrere lokale Empfänger bedienen.
- Der Server nimmt vom lokalen Empfänger die (Antwort-)Nachricht entgegen und stellt sie dem Sender anhand der idE zu.

Zusammenfassung direkte Adressierung:

- Der Sender spricht den Empfänger direkt und im allgemeinen hartcodiert an: *Adressen sind z.B. Maschinennr.+Prozeßnr., spezielle Hardwareausgänge des Systems: serielle oder parallele Leitungen*
- Sender- und Empfängeradresse sind fest im Programm hinterlegt
- Die Kommunikation kann nur erfolgen wenn Sender und Empfänger (ggf. Server) kommunikationsbereit sind.

K.1.2 Indirekte Adressierung

Sender spricht seinen Empfänger nicht direkt, sondern immer indirekt über eine Empfangsstelle an.

- ➔ Die Empfangsstelle hat i.d.R. eine eigenständige Adresse, meist unabhängig von Prozeßnummer und Maschine
- ➔ Sender- und Empfänger verabreden sich eine Empfangsstelle als Kommunikationsplatz zu verwenden
- ➔ Das unterlagerte Kommunikationssystem führt den Transport aus und 'weiß' wo und wie die Empfangsstellen zu erreichen sind.

Mailbox

Global bekannte FIFO-Puffer.

Es gibt auch Mailboxen für mehrere Sender bzw. Empfänger gleichzeitig.

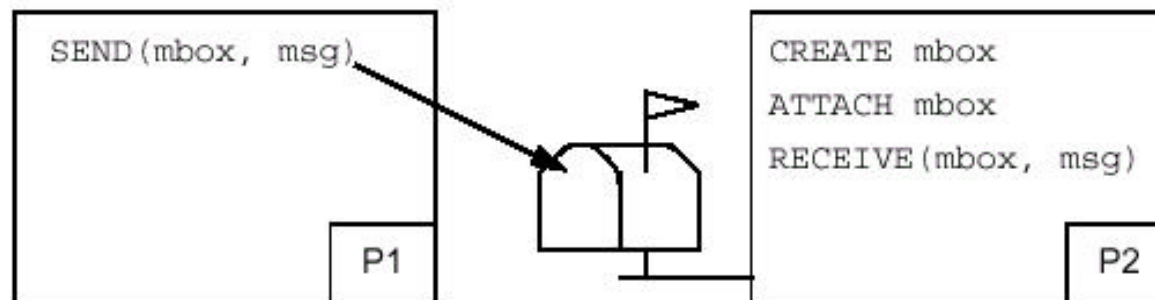
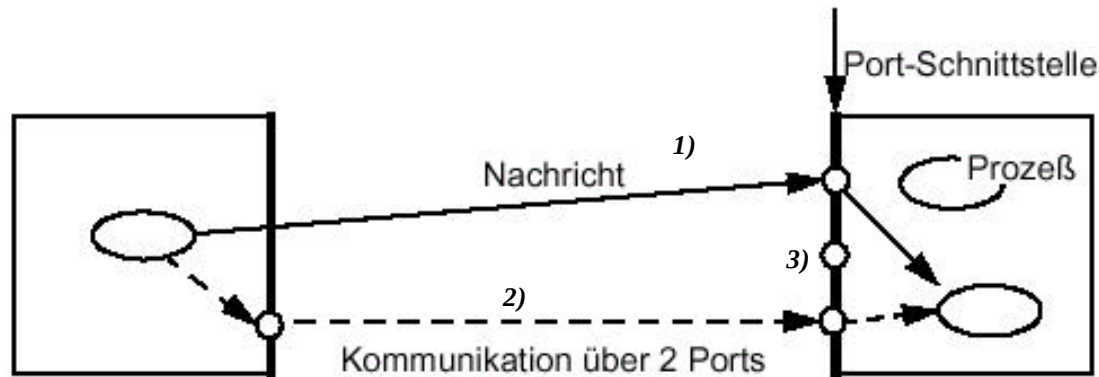


Bild K.2 Mailbox-Kommunikation

- Mailboxen können die Prozeßlebenszeit überdauern und bieten somit die Möglichkeit Nachrichten zwischenspeichern

Ports

spezielle Sende- und Empfangspunkte. → Ports werden meist vom Betriebssystem zur Verfügung gestellt.



- 1) Variante 'direkt' ohne Sendeport
- 2) Variante 'Port zu Port' mit Sendeport → connect erforderlich
- 3) Angebot an Ports, Lebensdauer an Prozeß gebunden

Bild K.3 Port-Kommunikation

Port-Eigenschaften:

- Sie definieren die Kommunikationsendpunkte und kapseln dadurch die interne Objekt- oder Prozeßstruktur ab. Der Port ist somit unabhängig von der Implementierung des dahinterliegenden Prozesses.
- Ports haben eine global eindeutige Identifikation, die sich meist aus der Rechneradresse und einer Portnummer zusammensetzt.
- Durch Ports wird ein selektiver Nachrichtenempfang unterstützt. Ein Prozeß sendet oder empfängt durchaus auf mehreren Ports.

typische Port-Operationen im Prozeß P

CreatePort (<i>PortId1</i>);	-- erzeugt einen neuen Port <i>PortId1</i>
OpenPort (<i>PortId1</i>);	-- öffnet den vorher erzeugten Port <i>PortId1</i> zur Verwendung
DeletePort (<i>PortId1</i>);	-- löscht im Prozeß P den Port <i>PortId1</i>
ClosePort (<i>PortId1</i>);	-- schließt einen verwendeten Port <i>PortId1</i> ohne ihn zu löschen
send (<i>PortId1</i> , message);	-- schickt Nachricht an den Port <i>PortId1</i>
receive (<i>PortId1</i> , message);	-- empfängt Nachricht vom Port <i>PortId1</i>

Bindevorgang zwischen zwei Prozeßports:

über einen externen Vermittlungsdienst oder im Betriebssystem verankert

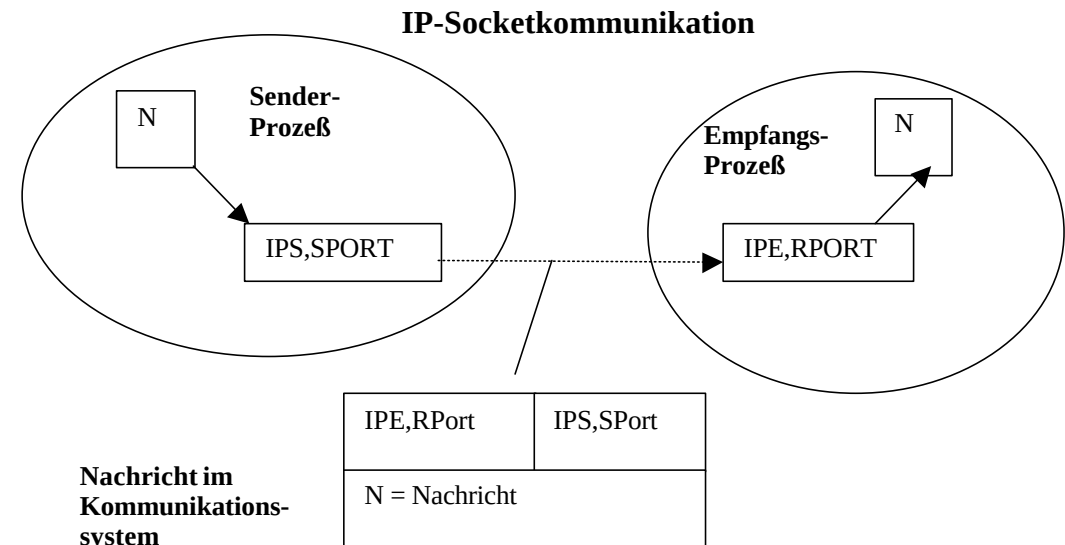
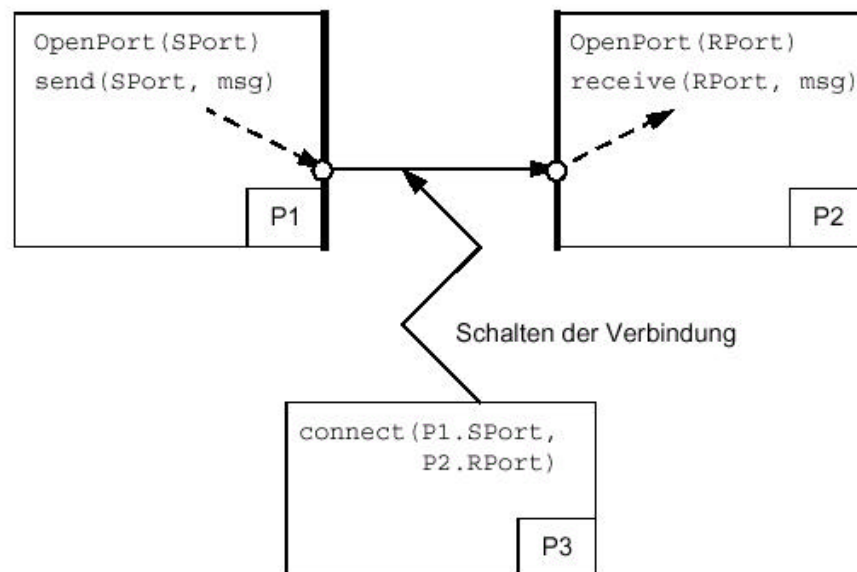


Bild K.4 Portorientierte Kommunikation

connect (P1.PortId1,P2.PortId2); -- Verbindet Prozeß P1 Port *PortID1* mit Prozeß P2 Port *PortID2*

Adressierungsvarianten

sowohl bei direkter und indirekter Adressierung:

- Adressen sind fest in den Programmcode integriert ☹
 - einfach zu programmieren
 - sehr unflexibel
 - Adreßänderung führt zur Editierung und Rekompilierung
- Adreßanfrage als Broadcast ins Netz ☺
 - Zugehöriger Server meldet sich
 - Orts-Transparenz
- Adreßanfrage an Namensdienst oder Trader ☺
 - Nachschauen in Adress-Datenbasis
 - Klient wendet sich dann direkt an den Server

K.2 Blockierung

K.2.1 Synchrone Kommunikation

Blockierende Kommunikation führt automatisch zu einer Synchronisation der Teilnehmerprozesse.

Ein Sender **blockiert** bis die Routine `send` zurückkehrt:

- d.h. er wartet bis die Nachricht über das Netz losgeschickt worden ist und eventuell eine Empfangsbestätigung vom Empfänger zurückgekommen ist.
- Ein korrespondierendes `receive` beim Empfänger blockiert bis eine Nachricht angekommen ist.

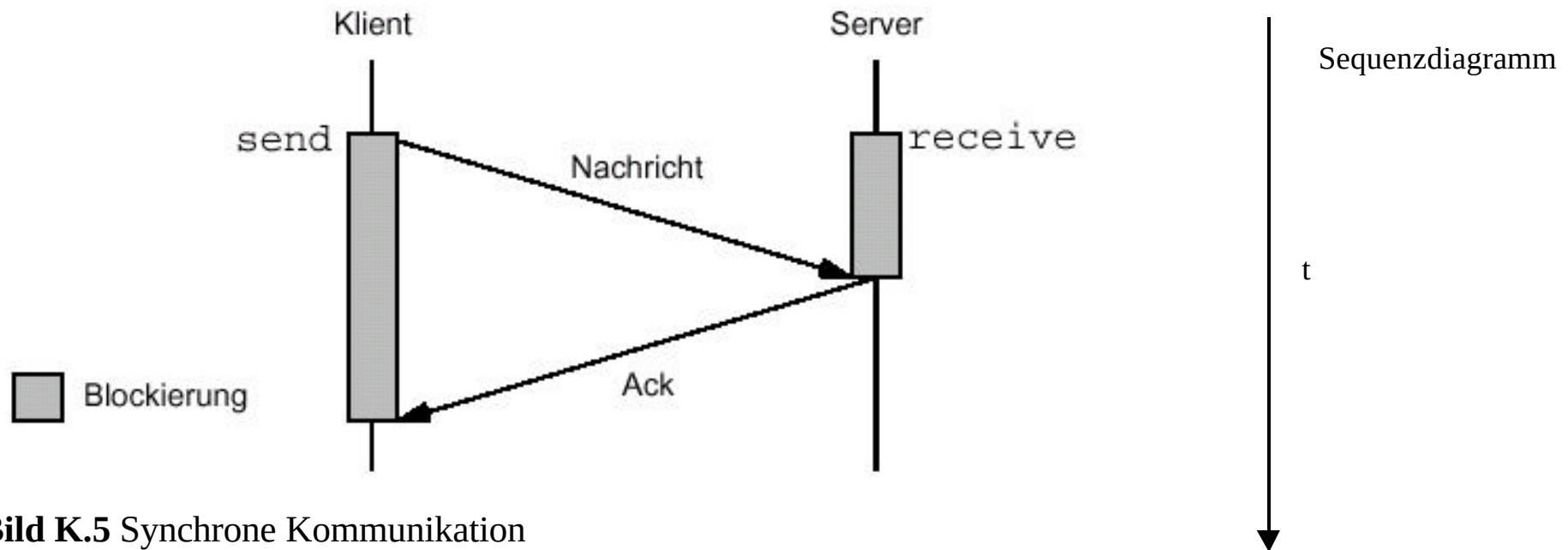


Bild K.5 Synchrone Kommunikation

Vorteile synchroner Kommunikation:

- keine zusätzlichen Mechanismen zur zeitlichen Synchronisation notwendig.
- Da auf jede eingehende Nachricht explizit gewartet wird, ist keine Pufferung von Nachrichten notwendig. Das spart sowohl Speicherplatz, als auch Verwaltungsaufwand.

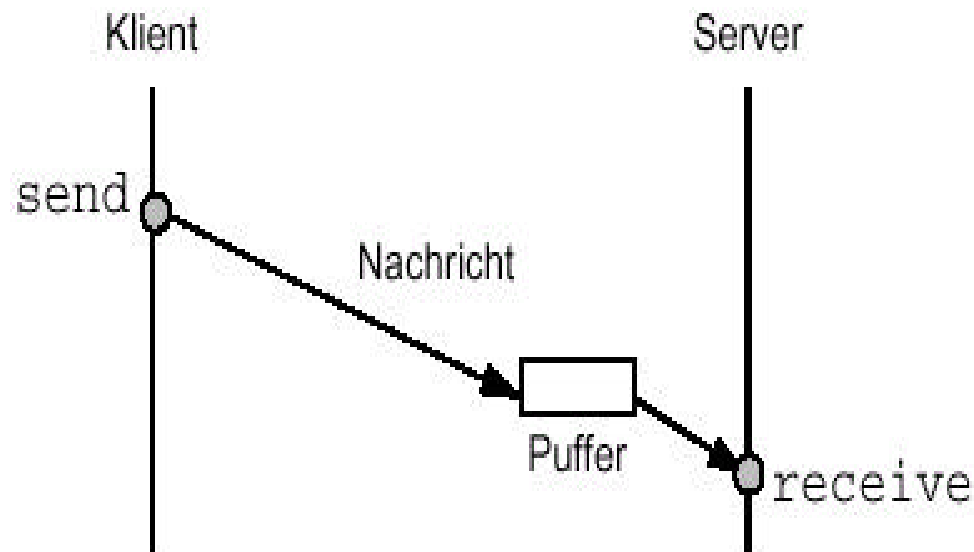
Nachteile:

- Durch die Blockierung ist die Parallelität von Sender und Empfänger stark eingeschränkt.
- Bei blockierendem `receive` kann der Empfänger nur auf eine Art von Nachricht warten.
- Ist der Sender bzw. Empfänger nicht bereit, so kann der entsprechende Empfänger bzw. Sender für immer blockieren.

Um dies zu vermeiden, muß man mit einem **Timeout** arbeiten.

K.2.2 Asynchrone Kommunikation

- Bei nicht-blockierender Kommunikation blockiert der Sender ebenfalls bis die Routine `send` zurückkehrt.
- `send` kehrt aber bereits zurück, wenn die Nachricht im Transportpuffer des Kommunikationssystems abgelegt ist.
- Ob und wann sie auf dem Netz erscheint, weiß der Sender damit nicht.
- Das korrespondierende `receive` liest in einem bereitgehaltenen Empfangspuffer.
- Ist eine Nachricht enthalten, wird sie gelesen, ansonsten kehrt `receive` in das rufende Programm zurück.



Wo ist der bzw.
Wer hat den
Puffer??
➔ Prozeß
➔ Betriebssystem

Bild K.6 Asynchrone Kommunikation

Vorteile asynchroner Kommunikation:

- zeitliche Entkopplung von Sender und Empfänger.
- Parallelarbeit möglich.

Nachteile:

- Pufferung auf der Sende- und Empfangsseite.
- Der Sender weiß nicht, ob und wann die Puffer frei sind.
- Sind die Puffer voll, so muß blockiert werden, was dem asynchronen Konzept widerspricht.

Einsatzgebiete: Echtzeit- oder ereignisgesteuerte Anwendungen

Mischform: Oft ist es ausreichend, daß nur der Sender asynchron läuft:

- dann kann der Empfänger im `receive` auch blockieren.

K.3 Pufferung

K.3.1 Ungepufferte Kommunikation

- **receive** stellt im Programm-Adreßraum eine einzelne Datenstruktur (i.a. als Array) für eine Nachricht zur Verfügung.
- Beim Eingang einer Nachricht kopiert das Kommunikationssystem im Betriebssystemkern die Nachricht in diese Datenstruktur und setzt den Empfangsprozess auf „ready“.

Vorteile:

- kein Speicher- und Verwaltungsaufwand

Nachteile:

- Wurde **receive** noch nicht aufgerufen, kann das Kommunikationssystem eine eingehende Nachricht nicht zustellen und muß sie verwerfen.
- Rufen mehrere Klienten den Server, so ist dieser eventuell nicht empfangsbereit, da er noch eine alte Nachricht bearbeitet.

K.3.2 Gepufferte Kommunikation

- Der Betriebssystemkern hält Puffer vor, um Nachrichten zwischenzuspeichern, die vom Empfänger oder dem Netzwerk momentan nicht abgenommen werden können.
- Falls diese Puffer voll laufen, wird ein Wegwerfen von Nachrichten bzw. eine Blockierung des Senders notwendig.

K.4 Kommunikationsformen

	asynchron	synchron
meldungsorientiert	Datagramm	Rendezvous
auftragsorientiert	asynchroner entfernter Dienstaufruf	synchroner entfernter Dienstaufruf

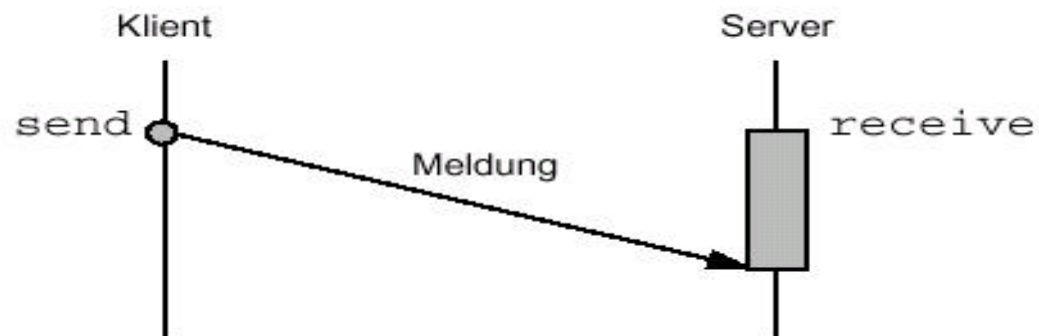
Bild K.7 Kommunikationsformen

K.4.1 Meldungsorientierte Kommunikation

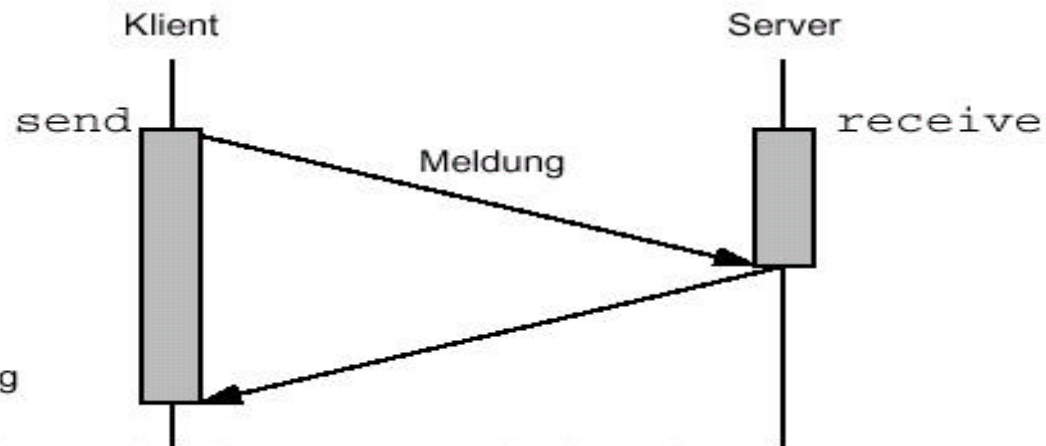
- immer Einwegnachrichten ('Idealbetrachtung')
- d.h. der Sender schickt eine Nachricht und erwartet **höchstens!** eine Empfangsbestätigung ('ACK')
- Eine echte Antwort ist nicht direkt mit der Nachricht verbunden

Asynchrone meldungsorientierte Kommunikation bezeichnet man als datagramm-basierte **Kommunikation**.
Synchrone meldungsorientierte Kommunikation bezeichnet man als **Rendezvous**.

Datagramm



Rendezvous



■ Blockierung

Bild K.8 Meldungsorientierte Kommunikationsformen

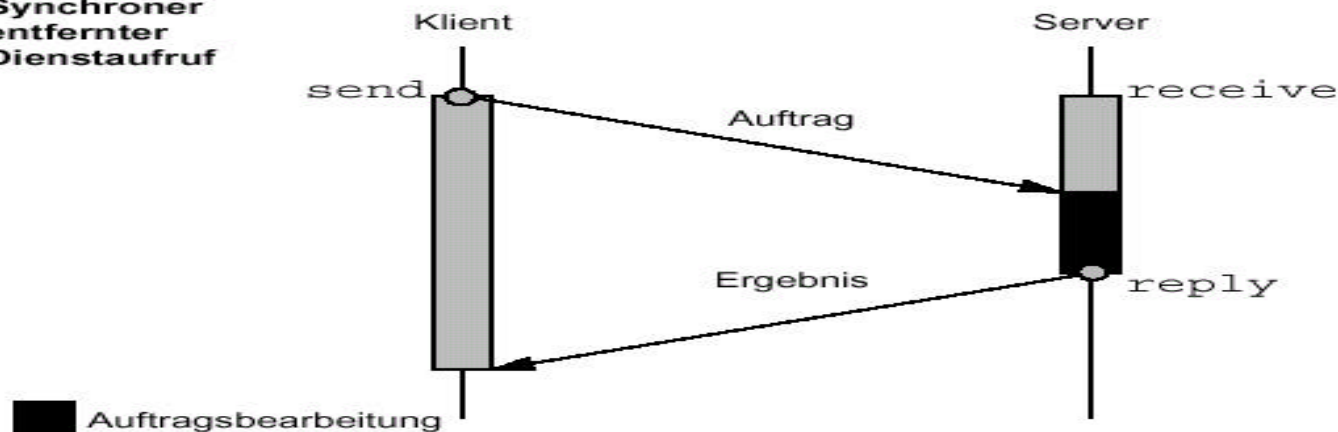
K.4.2 Auftragsorientierte Kommunikation

- Hin- und Rücknachrichten
- d.h. der Sender schickt eine Auftragsnachricht und erwartet die Rücksendung eines Ergebnisses.

Die Kommunikationsformen bezeichnet man als

- asynchronen entfernten Dienstaufruf (engl.: asynchronous remote service invocation, ARSI), bzw.
- synchronen entfernten Dienstaufruf (engl.: synchronous remote service invocation, SRSI).

**Synchroner
entfernter
Dienstaufruf**



**Asynchroner
entfernter
Dienstaufruf**

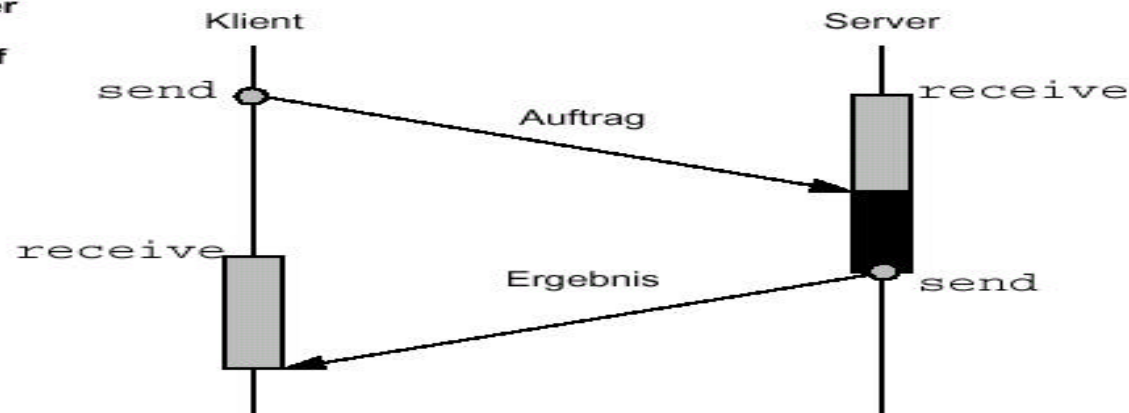


Bild K.9 Auftragsorientierte Kommunikation

K.4.3 Zuverlässigkeit auftragsorientierter Kommunikation

Situation:

Clientprozess ← Transportsystem → Serverprozess

Fehlerursachen:

Beispiele:

1. Verlust des Auftrags auf dem Weg zum Server
2. Verlust des Auftragsergebnisses auf dem Weg zum Client
3. Client stürzt nach Versenden des Auftrags vor Erhalt der Antwort ab
4. Server stürzt nach Erhalt des Auftrags bei der Bearbeitung ab
5. Server nicht erreichbar

Fehlererkennung:

Beispiele:

1. Transportsystem quittiert Auftragszustellung → Quittung bleibt aus Sender erkennt Auftragsverlust
2. Transportsystem quittiert nicht → Ausbleiben der Antwort auf einen Auftrag lässt die Erkennung der Fehlerursache nicht eindeutig zu:
 - Transport fehlerhaft, Server hat keinen Auftrag bekommen
 - Server erhielt Auftrag aber wurde während der Bearbeitung defekt
 - Antwort des Servers ging verloren

Frage: wie soll sich der Client verhalten?

- oft: **Timeouts im Antwortverhalten ⇔ Fehlererkennung**
- **Wiederholung von Aufträgen ⇔ Doppelausführung von Aufträgen**

➔ Es soll im folgenden gelten: Transportsystem signalisiert keine Fehler

Fehlersemantikklassen:

- *Maybe*

➔ Client sendet einmal seinen Auftrag und erwartet innerhalb eines Timeouts auf Antwort.

➔ Bei nicht Erhalten eines Ergebnisses erfolgt **keine automatische** Wiedeholung durch den Client.

- keine Fehlerbehandlungsmaßnahmen
- Dienst wird gar nicht oder höchstens einmal durchgeführt
- Im Fehlerfall hat man keine Hinweise, ob der Dienst tatsächlich ausgeführt wurde oder nicht
- Maybe-Semantik ist u.U. ausreichend für Auskunftsdienste

➔ Wenn nach einer gewissen Zeit keine Auskunft erteilt wurde, so probiert man es eben noch einmal.

Beispiel: News-Dienst

- ***At-Least-Once***

- ➔ Client sendet seinen Auftrag und erwartet innerhalb eines **Timeouts** auf Antwort.
 - ➔ Wahl des Timeout sollte ausreichend lange sein!!
 - ➔ Das Ausbleiben eines Ergebnisses wird durch Ablauf des Timeouts erkannt. Reaktion:
 - ➔ Wiederholung des Auftrags durch den Client solange bis eine Antwort erfolgt oder:
 - ➔ es wird in der Praxis nach einem vorgegebenen Maximum an Wiederholungen aufgegeben.

Fehlerursachen die zu einer Wiederholung des Sendens des Auftrags führen:

1. Transportsystemausfall auf dem Hinweg zum Server
2. Serverausfall
 - a. vor Ausführung des bereits erhaltenen Auftrags
 - b. nach Ausführung des Auftrags aber vor Versendung des Ergebnisses ➔ 2.Ausführung des Auftrags
3. Transportsystemausfall auf dem Rückweg vom Server ➔ 2.Ausführung des Auftrags
4. Durch Ergebnistransport- bzw. Serververarbeitungsverzögerungen: Ablauf des Timeouts beim Client, obwohl Auftrag korrekt zugestellt bzw. bearbeitet wird. ➔ 2...N. Ausführung des Auftrags

Zusammenfassung Eigenschaften:

- bei Nachrichtenverlusten wird der Auftrag bis zum Erfolg wiederholt oder abgebrochen.
- da keine Filterung von Duplikaten stattfindet, kann es vorkommen, daß ein Auftrag mehrfach ausgeführt wird

Verwendung:

- Bei idempotenten Operationen, d.h. mehrfache Ausführung verändert das Ergebnis nicht, ist diese Semantik ausreichend ➔ Das wiederholte Lesen einer Datei ist beispielsweise eine idempotente Operation.

Beispiel: mehrmals nach z.B. Timeout dieselbe Webadresse im Browser anwählen

- ***At-Most-Once und Exactly-Once***

→ Client sendet seinen Auftrag und erwartet innerhalb eines Timeouts auf Antwort.

→ wie ***At-Least-Once*** aber:

→ Erhält der Server einen Auftrag doppelt, so führt er ihn nur **einmal** aus schickt aber das **zwischengespeicherte** Ergebnis ggf. wiederholt an den Client.

→ **eindeutige Auftragskennung auf Client und Serverseite nötig**

→ jetzt: Atomarität der entfernten Operation garantiert denn: Ein Auftrag wird nun komplett durchgeführt, oder ein Fehler an den Klienten gemeldet, wobei dieser dafür dann Sorge tragen muß, mit dem nicht ausgeführten Auftrag umzugehen

→ ***At-Most-Once***: Hier führt nur der Server nur zur Laufzeit Buch über Aufträge pro Client. Damit lassen sich doppelte Aufträge herausfiltern.

→ ***Exactly-Once***: **Server und Client führen persistent** Buch (→ Transaktionen) über erfolgreich erledigte und ausstehende Aufträge. Nach Ausfall und Wiederanlauf des Servers/Clients ist weiterhin **alle** Informationen zu den Aufträgen verfügbar. Die **eindeutige Auftragskennung auf Client und Serverseite** ist 'weltweit' und 'zeitlos' → keine ID wiederholt sich je!! → **Transaktionssteuerung erforderlich!!!**

At-Most_once - Verwendung in RPC-Implementierungen (Corba, DCOM, Java RMI)

→ Beispiel: Lagerverwaltung → Entnahme eines Teils wird von der Workstation per RPC-Call dem Lagerserver gemeldet. Z.B: Durch Netzwerkstörung erfolgt der RPC-Aufruf mehrfach, wird aber nur einmal ausgeführt.

	Fehlerfreier Ablauf	Nachrichten-verluste	Ausfall des Servers
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-Least-Once	Ausführung: 1 Ergebnis: 1	Ausführung: ≥ 1 Ergebnis: ≥ 1	Ausführung: ≥ 0 Ergebnis: ≥ 0
At-Most-Once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0
Exactly-Once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

Bild K.10 Fehlersemantikklassen bei auftragsorientierter Kommunikation

K 5 Remote Procedure Call

Kommunikation ursprünglich über Ein-/Ausgabe-Primitive wie `send` oder `receive` ist umständlich und aufwendig zu programmieren

Gewünscht ist oft eine virtuelle Monoprozessorsicht, bei der eine Funktion oder Prozedur des Servers vom Klienten einfach aufgerufen werden kann, als ob es eine lokale Funktion wäre.

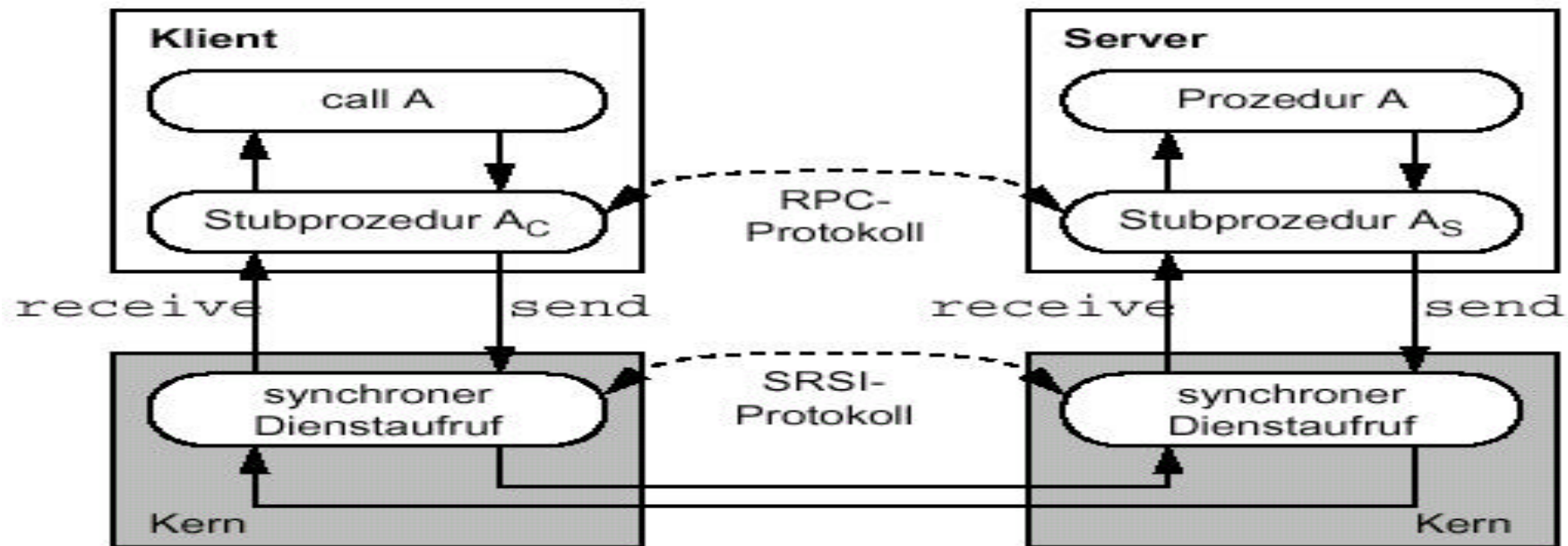
Ein RPC-System erlaubt den Aufruf von Prozeduren auf anderen Rechnern, inklusive der Übergabe von Parametern, basierend auf den Mechanismen eines synchronen entfernten Dienstaufrufs

Der Sender blockiert bis der Aufruf komplett abgeschlossen ist, wie bei einer lokalen Prozedur.

- Ein Programmierer kann also nach dem gleichen Paradigma programmieren wie er es in einem lokalen Programm verwendet.

RPC Besonderheiten und Einschränkungen

- Da ein Kommunikationssystem involviert ist, ergeben sich zusätzliche Fehlerquellen.
- Da auf zwei verschiedenen Rechnern gearbeitet wird, gibt es keinen gemeinsamen Adreßraum. Dies hat großen Einfluß auf die Parameterübergabe.
- Die beteiligten Prozesse haben separate Lebenszyklen. Es ist nicht sicher, ob die entfernte Prozedur immer ansprechbar ist.
- Durch den Kommunikations-Overhead ist die Aufrufdauer um Größenordnungen länger als bei lokalen Aufrufen.

**Bild K.13** RPC Funktionsweise

Aufgaben der Stubprozedur A_C • Parameter der aufzurufenden Prozedur zu einer Nachricht zusammensetzen • Serialisierungsprozeß = Marshalling	Aufgaben der Stubprozedur A_S • Übergabeparameter rekonstruieren • Deserialisierung = Demarshalling • Aktivierung der Prozedur A • Die Übertragung der Rückgabeparameter erfolgt analog.
Prozeduren-Multiplexing • mehrere entfernte Prozeduren auf gleichem Server • Prozedurkennung mitschicken	➔ Ausführlich bei Koll. Dr. Thomas RMI/JAVA und Agentensysteme

K6 Gruppenkommunikation

Die bisher betrachteten Kommunikationsmodelle waren Punkt-zu-Punkt-Szenarien.

Die Kommunikationsform, bei der ein Sender mehrere Empfänger erreichen kann, nennt man Gruppenkommunikation (1:m Kommunikation).

Die Versendung einer Nachricht an die Gruppe erfolgt durch einen Multicast.

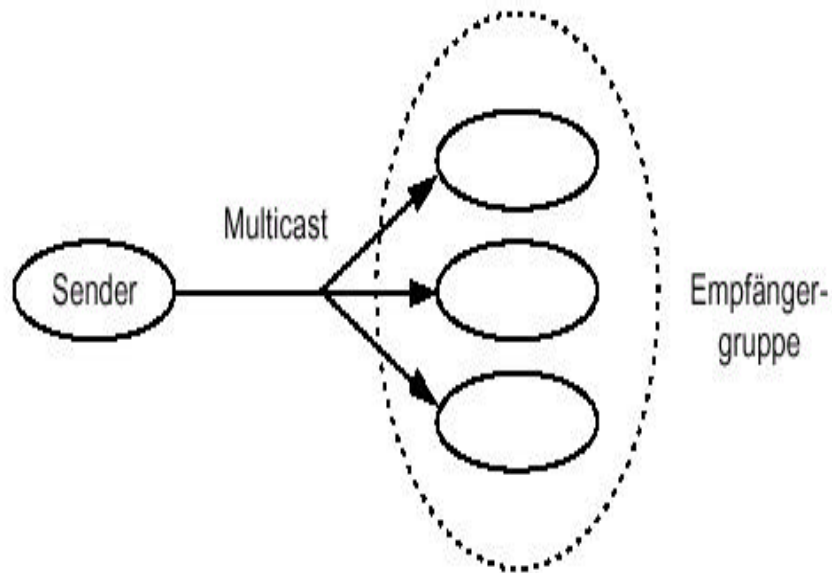


Bild K.17: Gruppenkommunikation

typische Anwendungen für Gruppenkommunikation

- Fehlertoleranz wird durch replizierte Dienste gewährleistet, wobei alle Server identische Aufträge erhalten.
- Um die Lokation von Objekten zu bestimmen, schickt man eine Nachricht an eine Gruppe potentieller Server.
- Bei Datenreplikation werden die Änderungen über Gruppenkommunikation an die replizierten Objekte bekannt gegeben.
- In Newsdiensten werden die Daten über Gruppenkommunikation verteilt.
- Konferenzsysteme und Gruppenanwendungen wie IRC, JVTOS und MBone bedienen sich einer Gruppenkommunikations-Semantik.

Realisierungsvarianten

Realisierung über multicastfähige Netzwerke (z.B.: Ethernet) ist relativ einfach.

- jede Gruppe erhält eine Multicastadresse

Eine 1:m Kommunikation kann auch mittels Broadcast erreicht werden, aber

- keine Möglichkeit einer dynamischen Gruppenadressierung.
- alle Rechner im Netzwerk erhalten die Nachricht
- unbeteiligte Empfänger müssen die Nachricht verwerfen

Steht im Netzwerk weder Multicast noch Broadcast zur Verfügung

- Multicast durch Folge von Unicasts nachbilden

K.6.1 Entwurfskriterien

Wahl der Gruppenform

- *Geschlossene Gruppen*
nur Mitglieder der Gruppe dürfen Gruppennachrichten verschicken.
- *Offene Gruppen*
Hier können auch Sender, die nicht der Gruppe angehören, Nachrichten an die Gruppe schicken.

Basisprimitive für die Gruppenverwaltung

```
GroupId = CreateGroup (ListOfPortIds);  
JoinGroup (IndividualPortId, GroupId);  
LeaveGroup (IndividualPortId, GroupId);  
DeleteGroup (GroupId);
```

Gruppenverwaltung

- dedizierter Gruppen-Server oder
- verteilt unter den Mitgliedern
- `join` und `leave` sollten synchron mit den eigentlichen Multicastnachrichten abgewickelt werden.

Kommunikationsprimitive:

```
SendtoGroup (GroupId, Message);  
ReceivefromGroup (GroupId, Message);
```

K.6.2 Semantik der Gruppenkommunikation

Zuverlässigkeitsgrade

Der Zuverlässigkeitsgrad gibt an, wieviele Empfänger aus der Gruppe mit Nachrichten bedient werden.

- *keine Zuverlässigkeit* : Es gibt keine Garantie darüber, wieviele Empfänger und welche Empfänger eine Gruppennachricht tatsächlich erhalten.
- *k-zuverlässig*: Es wird garantiert, daß mindestens k Mitglieder der Empfängergruppe eine Gruppennachricht erhalten.
- *atomar*: Es wird sichergestellt, daß entweder alle Mitglieder der Empfängergruppe eine Gruppennachricht erhalten oder keines.

Ordnungsgrade

Der Ordnungsgrad beschreibt die Reihenfolge, in der Gruppennachrichten von den Gruppenmitgliedern empfangen werden.

- *ungeordnet*: Es besteht keine festgeschriebene Empfangsreihenfolge der Nachrichten. D.h. jede Teilnachricht einer Gruppennachricht kann jede andere Teilnachricht des gleichen Senders oder anderer Sender überholen.
- *FIFO-geordnet* :Alle Gruppennachrichten ein und desselben Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in der gesendeten Reihenfolge an.
- *kausal geordnet* : Falls eine Gruppennachricht m_b kausal abhängig von m_a ist ($m_a < m_b$), so erhalten alle Mitglieder der Gruppe die Nachricht m_b erst nach dem Empfang von m_a . Kausal unabhängige Gruppennachrichten sind weiterhin ungeordnet.
- *total geordnet*: Zusätzlich zur kausalen Ordnung werden alle nicht kausal abhängigen Nachrichten in einer beliebigen, aber für alle Mitglieder der Gruppe gleichen Reihenfolge empfangen.

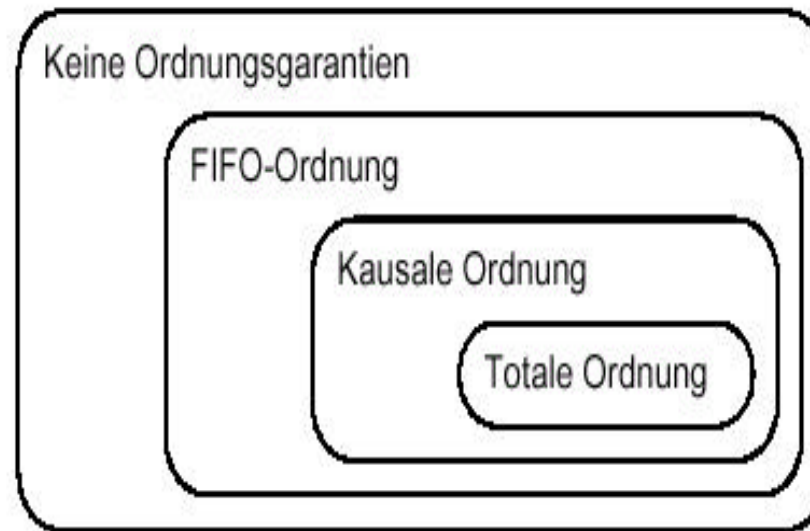
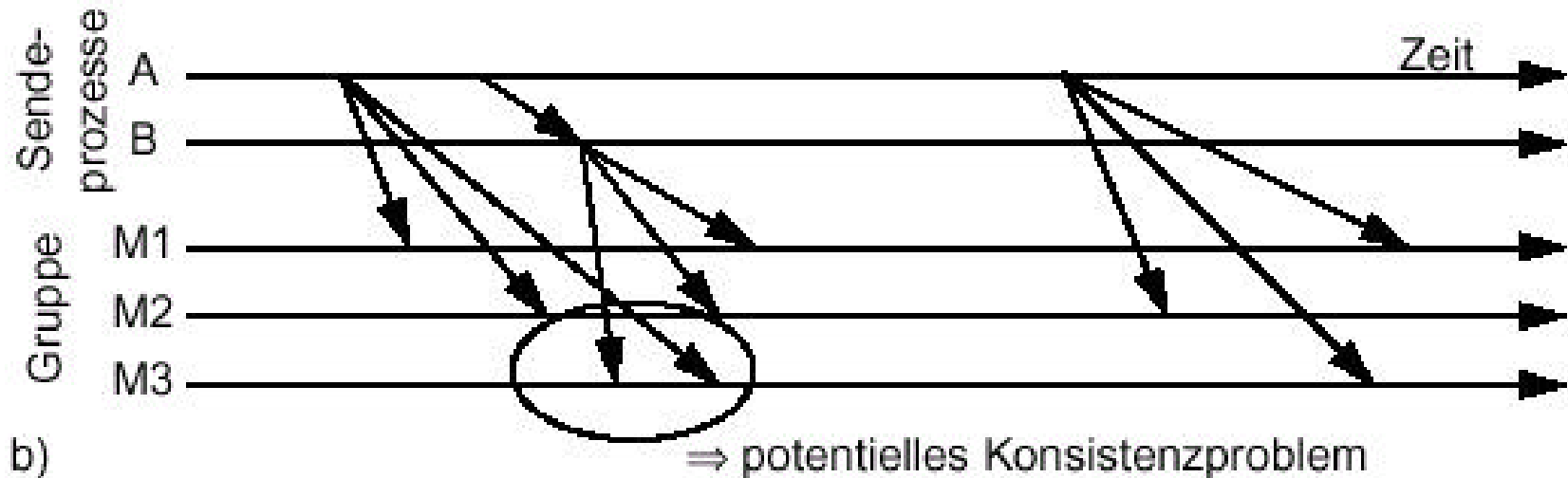
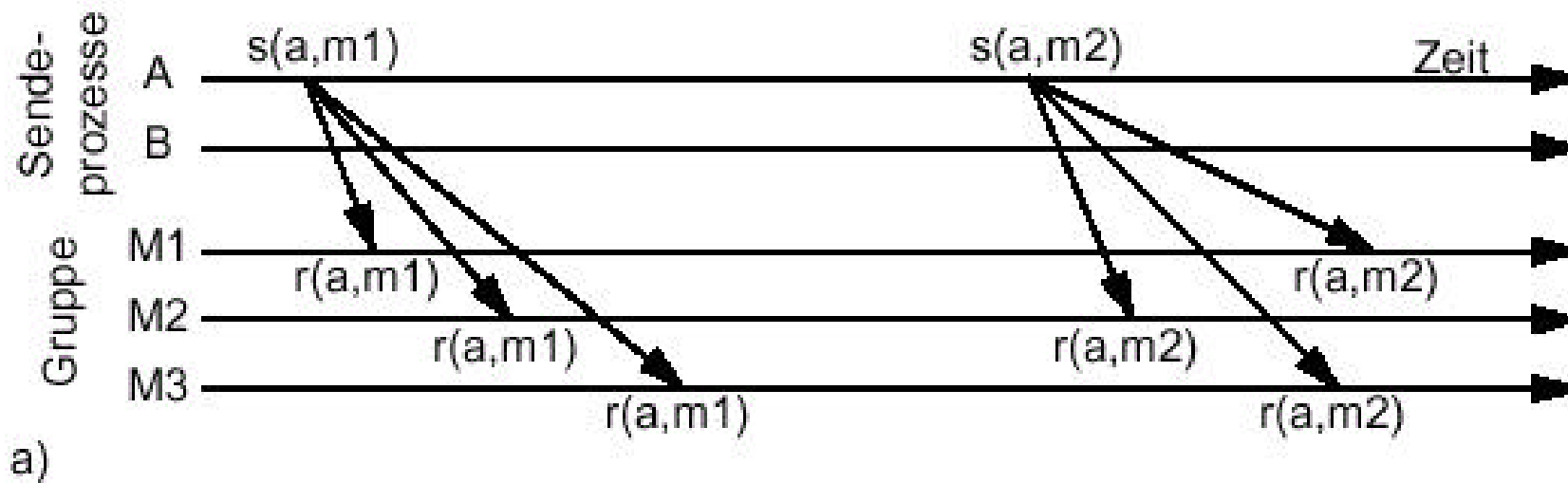


Bild K.19: Ordnungsgrade von Gruppen-Nachrichten

**Bild K.18:** FIFO-geordnete Gruppenkommunikation

Kausale Ordnung

Den Begriff der kausalen Ordnung auf Ereignissen definiert man über die Ereignisse, deren Ursache vor ihrer Wirkung beobachtbar sein sollen.

Definition: Kausalität

- $e_1 <_p e_2$: e_2 ist von e_1 kausal abhängig genau dann, wenn e_1 Auswirkungen auf e_2 haben kann.
- Zwei Ereignisse e_1 und e_2 sind dann kausal unabhängig, wenn weder $e_1 <_p e_2$ noch $e_2 <_p e_1$ gilt.
- Es gilt die Transitivität: Wenn $e_1 <_p e_2$ und $e_2 <_p e_3$, dann gilt: $e_1 <_p e_3$

Die Relation $<_p$ definiert eine **partielle** Ordnung auf der Menge der Ereignisse.

Kausalität und Gruppenkommunikation

Innerhalb eines sequentiell ablaufenden Prozesses sind die nacheinander stattfindenden Ereignisse kausal geordnet. Jede kausalitätserhaltende Ordnung ist somit auch eine FIFO-Ordnung.

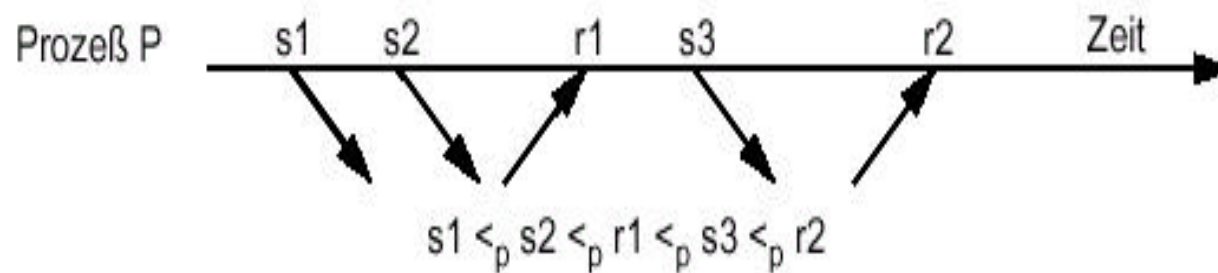


Bild K.20: Kausalitäten innerhalb eines Prozesses

Das Empfangen einer Nachricht ist kausal von ihrem Absenden abhängig.

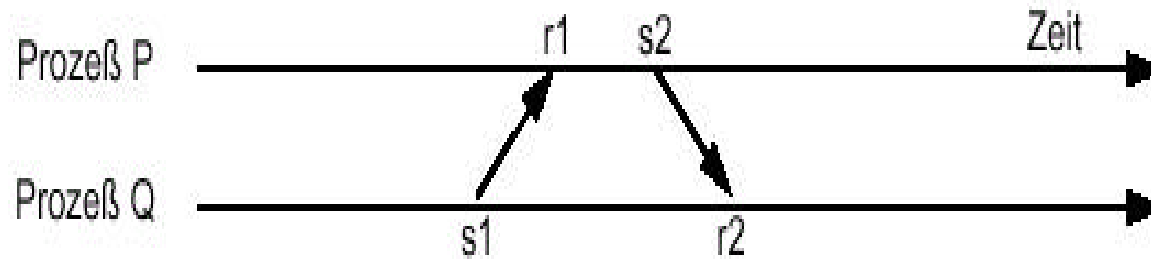


Bild K.21 Kausalitäten zwischen Prozessen

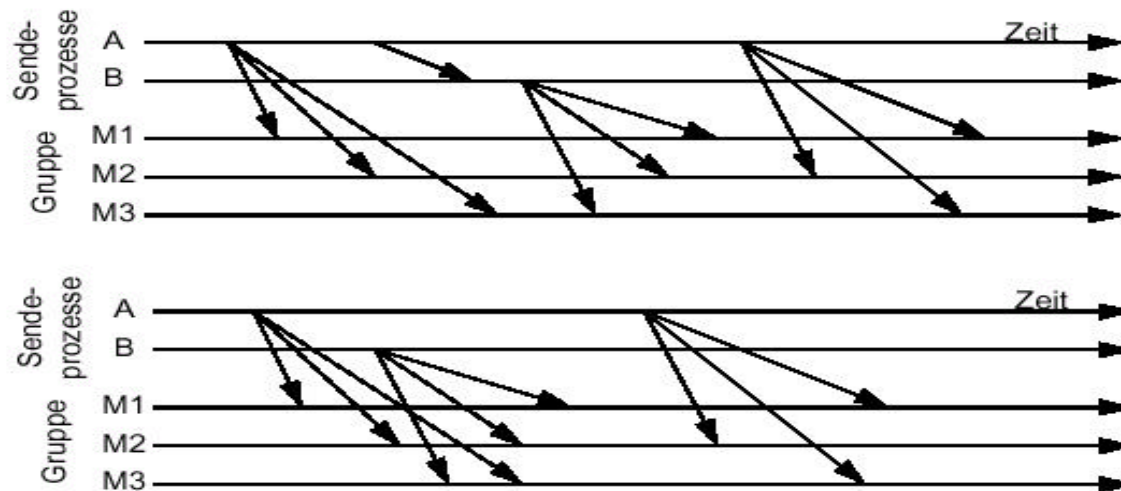


Bild K.22 Kausal geordnete Gruppenkommunikation

Totale Ordnung

Um eine kausale Ordnung zur totalen Ordnung zu erweitern, führt man ein zusätzliches „**willkürliches**“ Ordnungskriterium für kausal unabhängige Ereignisse ein.

Definition: Totale Ordnung

- $e_1 <_t e_2$: $e_1 <_p e_2$ oder im Falle kausal unabhängig ist $\neg(e_1 <_p e_2) \wedge \neg(e_2 <_p e_1)$ dann gilt willkürlich e_1 vor e_2 wenn gilt: $\text{total}(e_1) < \text{total}(e_2)$, $\text{total}(e)$ willkürliche Ordnungsfunktion, die global eindeutige Werte liefert.

$\text{total}(e)$: = $\langle \text{Ereigniszähler}:\text{Rechnerinternetadresse}:\text{Prozeßnr} \rangle$ ($<$ numerisch kleiner)

Da alle Prozesse den gleichen Algorithmus verwenden kann jeder für sich die Ordnung herstellen.

Für Gruppenkommunikation reicht die kausale Ordnung meist aus.

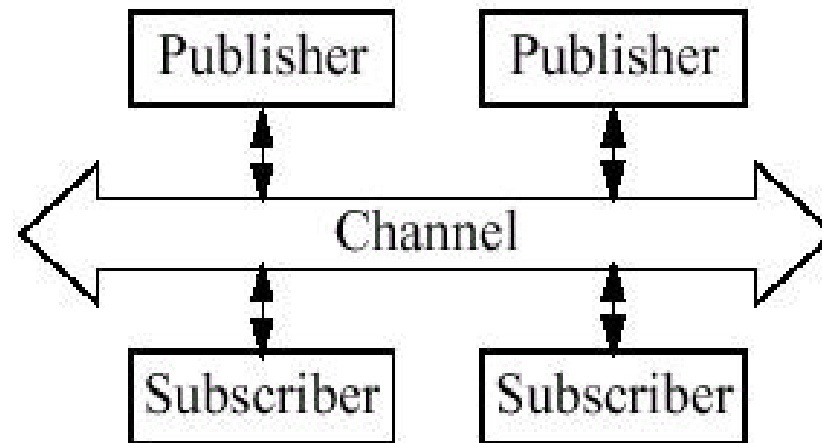
K.7 Messaging Systeme

Charakteristika

- mehrere Klienten sind an der gleichen objekt-initiierten Information interessiert
- Klienten „kommen und gehen“
- Objekte können nicht suspendiert werden, während Kommunikation abläuft
- Peer-to-Peer Kommunikationsmuster
- Messaging Agents unterstützen Erzeugung, Verschicken und Empfang von Nachrichten

Kommunikationsbeziehungen

- Point-to-Point
 - Nachrichten werden an Warteschlangen geschickt und von dort empfangen
- Publish/Subscribe
 - Klientenrollen: Publisher bzw. Subscriber
 - Channel o.ä. als Abstraktion

**Bild K.23** Publish/Subscribe Modell

Kommunikationsmodelle • Push: asynchrones Zustellen bei den Empfängern • Pull: Empfänger holen Nachrichten synchron • beides auch als Multicast/Broadcast	Besonderheiten • Nachrichtentypen • spezifische Kanäle • Quality of Service • Zustellgarantien • Ordnungen • Prioritäten
Beispiele • Java Message Service • Java InfoBus • CORBA Event Service • CORBA Notification Channel	

Übungen zu Kommunikationsmodelle

Kommunikationsmodelle

- a. Ports sind eine Form der indirekten Adressierung. Aus welchen Gründen benötigt ein Prozeß evtl. mehrere Ports? Warum kann es sinnvoll sein, wenn an einem Port mehrere Prozesse empfangen? Nennen Sie je ein Beispiel aus der Internet-Welt.
- b. Wann würden Sie ein blockierendes send verwenden und wann ein nicht-blockierendes?
- c. Bei einem TCP LISTEN Socket werden eingehende Verbindungswünsche von Clienten solange in einem sog. Backlog gepuffert, bis der Serverprozeß diese mit einem `accept()` annimmt. Was ist daran problematisch? Welche alternative Lösung gäbe es?
- d. Wie würden Sie Datagramm-/Rendezvous-orientierte Kommunikation und einen asynchronen bzw. synchronen entfernten Diensteaufruf unter Verwendung der socket Schnittstelle realisieren? Nennen Sie je ein Beispiel für eine typische Anwendung.
- e. Nennen Sie je einen typischen Anwendungsfall für die Fehlersemantiken Maybe, At-Least-Once, At-Most-Once und Exactly-Once.

Gruppenkommunikation

- a. Das MBONE (www.mbone.de) ist ein Multicast Overlay Netzwerk auf dem klassischen Internet. Finden Sie heraus wie es funktioniert.
- b. Welchen Zuverlässigkeitsgrad und welchen Ordnungsgrad würden Sie für die folgenden Anwendungsfälle verwenden:
 - Übertragung einer Radiosendung
 - Videokonferenz
 - Übertragung von Börsenkursen an eine Empfängergruppe
 - Cache Updates in replizierten Filesystemen
- c. Begründen Sie Ihre Entscheidungen.