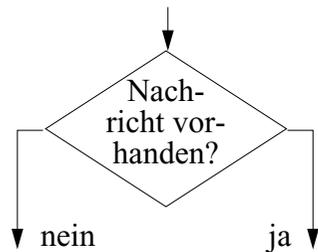


Varianten beim Empfangen von Nachrichten - Nichtblockierung

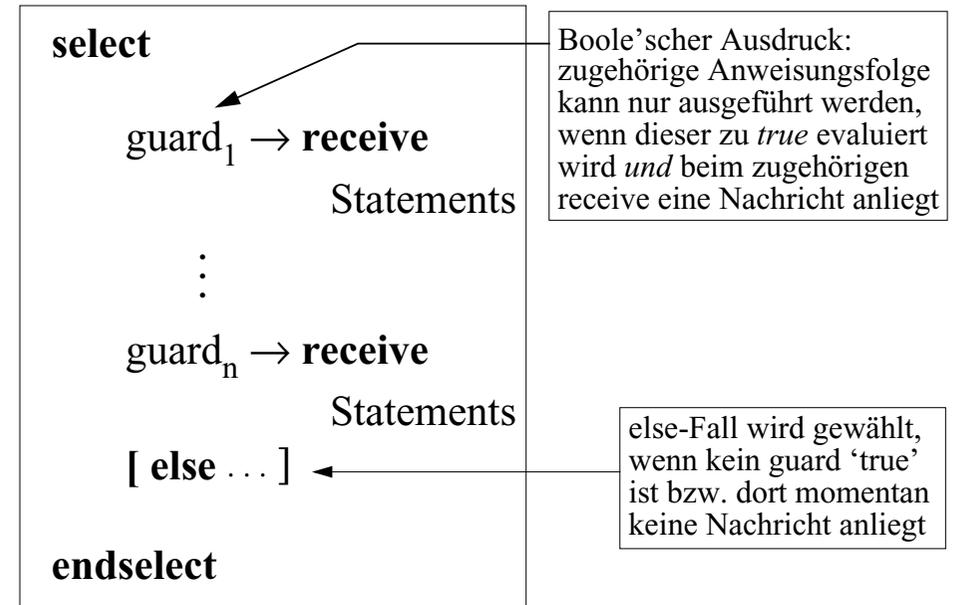
- Typischerweise ist ein "receive" blockierend
- Aber auch *nichtblockierender* Empfang ist denkbar:



- "Non-blocking receive"
- Sprachliche Realisierung z.B. durch "Returncode" eines als Funktionsaufruf benutzten "receive"

Alternatives Empfangen

- Sprachliche Realisierung etwa so:

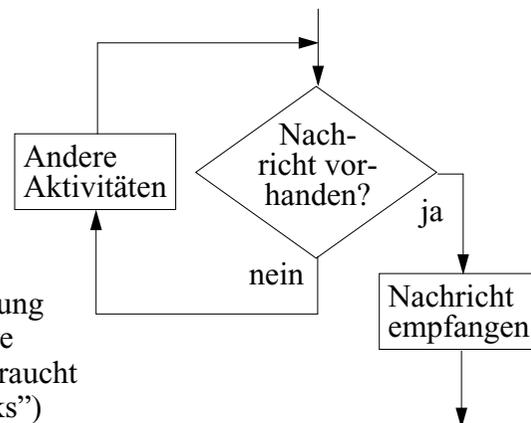


Boole'scher Ausdruck: zugehörige Anweisungsfolge kann nur ausgeführt werden, wenn dieser zu *true* evaluiert wird *und* beim zugehörigen receive eine Nachricht anliegt

else-Fall wird gewählt, wenn kein guard 'true' ist bzw. dort momentan keine Nachricht anliegt

- Aktives Warten: ("busy waiting")

- Nachbildung des blockierenden Wartens wenn "andere Aktivitäten" leer
- Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte ("spin locks")



- Was geschieht, wenn mehrere guards "true" sind?
→ nichtdeterministische Auswahl, Wahl des ersten Falles...
- Typischerweise *nichtblockierend*; aber blockierend, wenn else-Alternative fehlt
- Aktives Warten durch umschliessende while-Schleife
 - bei else könnte im Fall von aktivem Warten die while-Bedingung auf false gesetzt werden, falls das Warten abgebrochen werden soll, oder es könnte dort mittels timer ("wait") eine kurze Zeit gewartet werden
 - else-Fall kann auch einfach das leere Statement enthalten

- Weitere Möglichkeit: unterbrechungsgesteuertes ("asynchrones") Empfangen der Nachricht (→ nicht unproblematisch!)

Selektives Empfangen

≠ alternatives!

- Bedingung an den *Inhalt* (bzw. Typ, Format,...) der zu empfangenden Nachricht
 - Dadurch werden gewisse (“unpassende”) Nachrichten einfach ausgeblendet
 - Bedingung wird oft vom aktuellen Zustand des Empfängers abhängen
-
- Vorteil bei der Anwendung:
Empfänger muss nicht mehr alles akzeptieren und zwischenspeichern, sondern bekommt nur das, was ihm momentan passt

Zeitüberwachte Kommunikation

- Empfangsanweisung soll maximal (?) eine gewisse Zeit lang blockieren (“timeout”)
 - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat
- Sinnvoll bei:
 - Echtzeitprogrammierung
 - Vermeidung von Blockaden im Fehlerfall (etwa: abgestürzter Kommunikationspartner)
 - dann sinnvolle Recovery-Massnahmen treffen (“exception”)
 - timeout-Wert “sinnvoll” setzen!

Quelle vielfältiger Probleme...

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

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

Zeitüberwacher Nachrichtenempfang

- Möglicher Realisierung:
 - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
 - Sprungziel könnte z.B. eine Unterbrechungs-routine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive
- “systemnahe”, unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

- Sprachliche Einbindung besser z.B. so:

receive ... delay t

Vorsicht!

Blockiert maximal t Zeiteinheiten

- bzw. so:

select

guard₁ → **receive ...**

⋮

delay t → Anweisungen ...

endselect

Wird nach *mind.* t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen

- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!
- Frage: sollte “delay 0” äquivalent zu “else” sein?

Implizites Empfangen

- Keine receive, select...-Anweisung, sondern Spezifikation von Routinen, die bei Vorliegen einer Nachricht ausgeführt (“angesprungen”) werden

- z.B. RPC:

bzw. asynchrone Variante oder “Remote Method Invocation” bei objektorientierten Systemen

```

remote procedure1
end procedure1
⋮
remote proceduren
end proceduren
    
```

- Analog auch für den “Empfang” einer Nachricht *ohne Antwortverpflichtung* denkbar

- *Semantik:*

- Interne Parallelität?

“Routine”

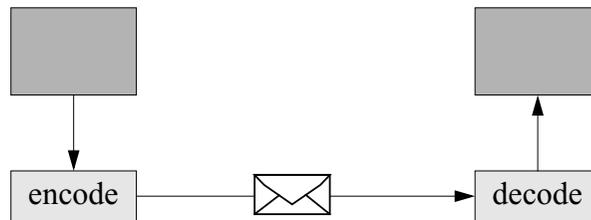
- mehr als eine gleichzeitig aktive Prozedur, Methode,... im Empfänger?
- vielleicht sogar mehrere Instanzen der gleichen Routine?
- Realisierung: z.B. bei Aufruf jeweils Prozess oder Thread gründen

- Atomare Routinen?

- wird eine aktive Routine evtl. unterbrochen, um eine andere aktivierte auszuführen? Oder nicht?

Kommunizierbare Datentypen?

- Werte von “klassischen” einfachen Datentypen
 - int, character, string, floating point,...
- Kompatibilität in heterogenen Systemen
 - Grösse von int?
 - Codierung von Text?
 - höherwertiges Bit links oder rechts?
 - ...



- Vereinbarung einer *Standardrepräsentation* (z.B. XDR)
- marshalling (encode / decode) kostet Zeit

Gruppen- kommunikation

- Was ist mit *komplexen Datentypen* wie

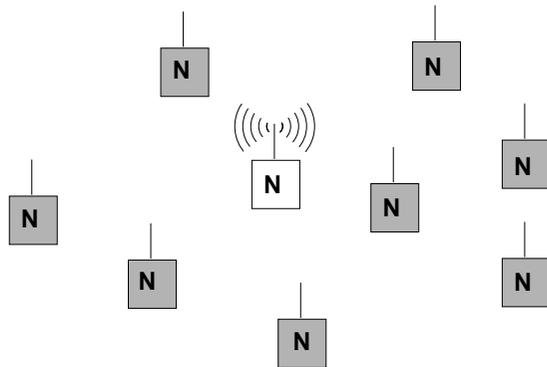
- | | | |
|-----------------------|---|-----------------------------------------------------------------------------------------|
| - Records, Strukturen | } | - sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein? |
| - Objekte | | - sollen Referenzen symbolisch, relativ... interpretiert werden? Ist das stets möglich? |
| - Referenzen, Zeiger | | - wie wird Typkompatibilität sichergestellt? |
| - Zeigergeflechte | | |

- U.U. “Linearisieren” und evtl. Strukturbeschreibung mitschicken

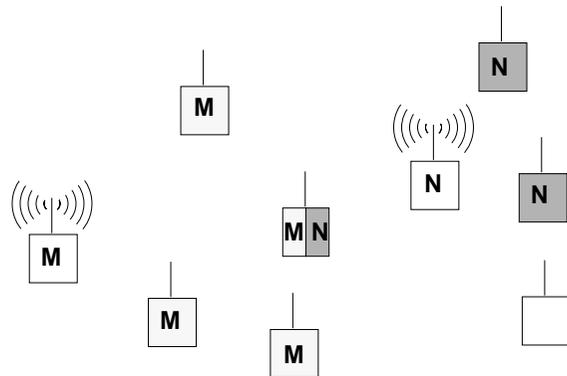
- Sind (Namen von) Ports, Prozessen... eigenständige Datentypen, deren Werte versendet werden können?

“first class objects”

Gruppenkommunikation



Broadcast: Senden an die *Gesamtheit* aller Teilnehmer



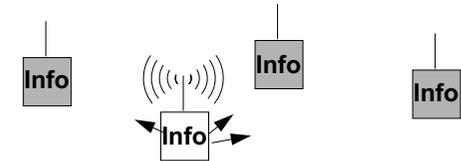
Multicast: Senden an eine *Untergruppe* aller Teilnehmer

- entspricht Broadcast bezogen auf die Gruppe
- verschiedene Gruppen können sich ggf. überlappen
- jede Gruppe hat eine Multicastadresse

Anwendungen von Gruppenkommunikation

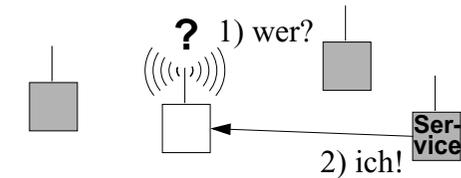
- *Informieren*

- z.B. Newsdienste

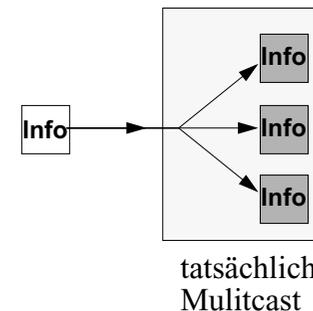
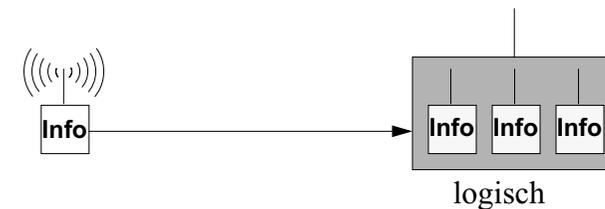


- *Suchen*

- z.B. Finden von Objekten und Diensten



- “*Logischer Unicast*” an replizierte Komponenten

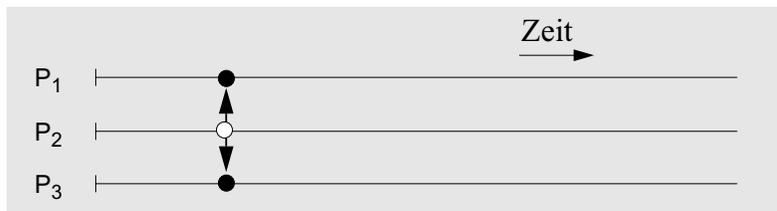


Typische Anwendungs-
klasse von Replikation:
Fehlertoleranz

Gruppenkommunikation - idealisierte Semantik

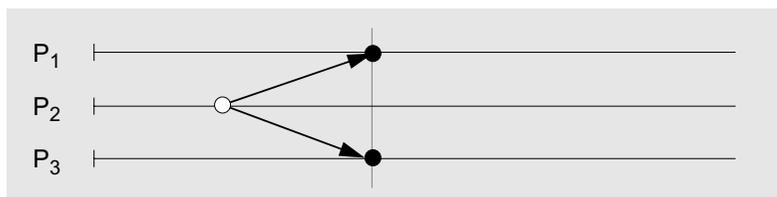
- Modellhaftes Vorbild: Speicherbasierte Kommunikation in zentralistischen Systemen

- augenblicklicher "Empfang"
- vollständige Zuverlässigkeit (kein Nachrichtenverlust etc.)



- Nachrichtenbasierte Kommunikation: Idealisierte Sicht

- (verzögerter) *gleichzeitiger* Empfang
- vollständige Zuverlässigkeit



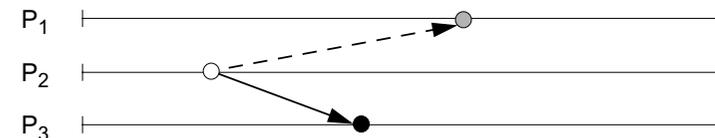
Gruppenkommunikation - tatsächliche Situation

- Medium (Netz) ist oft nicht multicastfähig

- LANs höchstens innerhalb von Teilstrukturen;
WLAN als Funknetz a priori anfällig für Übertragungsstörungen
- multicastfähige Netze sind typischerweise nicht verlässlich
(keine Empfangsgarantie)
- bei Punkt-zu-Punkt-Netzen: "Simulation" von Multicast durch ein Protokoll (z.B. Multicast-Server, der an alle einzeln weiterverteilt)

- Nachrichtenkommunikation ist nicht "ideal"

- indeterministische Zeitverzögerung → Empfang zu *unterschiedlichen* Zeiten
- nur bedingte Zuverlässigkeit der Übermittlung



- Ziel von Broadcast / Multicast-Protokollen:

- möglichst gute Approximation der Idealsituation
- möglichst hohe Zuverlässigkeit und Effizienz

- Beachte: Verlust von Nachrichten und sonstige Fehler sind bei Broadcast ein ernsteres Problem als beim "Unicast"! (Wieso?)

- Hauptproblem bei der Realisierung von Broadcast:
(1) Zuverlässigkeit und (2) garantierte Empfangsreihenfolge

Senderausfall beim Broadcast

Wenn Broadcast implementiert ist durch Senden vieler Einzelnachrichten

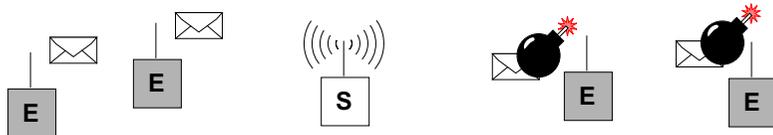
a) Sender fällt ganz aus: *kein* Empfänger erhält Nachricht

- „günstiger“ Fall: *Einigkeit* unter den Überlebenden!



b) Sender fällt *während* des Sendevorgangs aus: nur *einige* Empfänger erhalten u.U. die Nachricht

- “ungünstiger” Fall: *Uneinigkeit*



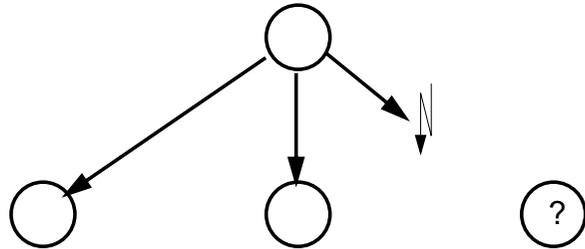
- mögliche Abhilfe: Empfänger leiten zusätzlich die Nachricht auch untereinander weiter
- Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche Folgeprobleme sein! (Da wäre es manchmal besser, *gar kein* Prozess hätte die Nachricht empfangen!)

“Best effort” bei Broadcasts

- Fehlermodell: Verlust von Nachrichten (und evtl. temporärer Crash von Prozessen)
 - Nachrichten können aus unterschiedlichen Gründen verloren gehen (z.B. Netzüberlastung, Empfänger hört gerade nicht zu...)
 - Euphemistische Bezeichnung, da keine extra Anstrengung
 - typischerweise einfache Realisierung ohne Acknowledgements etc.
 - Keinerlei Garantien
 - unbestimmt, wieviele / welche Empfänger eine Broadcastnachricht tatsächlich empfangen
 - unbestimmte Empfangsreihenfolge
- Kann z.B. beim Software-update über Satellit zu einem ziemlichen Chaos führen
- Allerdings effizient (im Erfolgsfall)
 - Geeignet für die Verbreitung unkritischer Informationen
 - z.B. Informationen, die evtl. Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen
 - Evtl. als Grundlage zur Realisierung höherer Protokolle
 - diese basieren oft auf der A-priori-Broadcastfähigkeit von Netzen
 - günstig bei zuverlässigen physischen Kommunikationsmedien (wenn Fehlerfall sehr selten → aufwendiges Recovery auf höherer Ebene tolerierbar)

“Reliable Broadcast”

- Ziel: Unter gewissen Fehlermodellen einen “möglichst zuverlässigen” Broadcast-Dienst realisieren



- Quittung (“positives Acknowledgement”: ACK) für jede Einzelnachricht
 - im Verlustfall einzeln nachliefern oder (falls broadcastfähiges Medium vorhanden) einen zweiten Broadcast-Versuch? (→ Duplikaterkennung!)
 - viele ACKs → teuer / skaliert schlecht
- Skizze einer anderen Idee (“negatives Ack.”: NACK):
 - alle broadcasts werden vom Sender aufsteigend nummeriert
 - Empfänger stellt beim *nächsten* Empfang u.U. eine Lücke fest
 - für fehlende Nachrichten wird ein “negatives ack” (NACK) gesendet
 - Sender muss daher Kopien von Nachrichten (wie lange?) aufbewahren und fehlende nachliefern
 - “Nullnachrichten” sind u.U. sinnvoll (→ schnelles Erkennen von Lücken)
 - Kombination von ACK / NACK mag sinnvoll sein
- Dies hilft aber nicht, wenn der Sender mittendrin crasht!

Reliable-Broadcast-Algorithmus

- Zweck: Jeder nicht gecrashte und zumindest indirekt erreichbare Prozess soll die Broadcast-Nachricht erhalten
 - Voraussetzung: zusammenhängendes “gut vermaschtes” Punkt-zu-Punkt-Netz
 - Fehlermodell: Prozesse und Verbindungen mit Fail-Stop-Charakteristik

Sender s : Realisierung von **broadcast(N)**

- **send($N, s, sequ_num$)** an alle Nachbarn (inklusive an s selber);
- $sequ_num++$

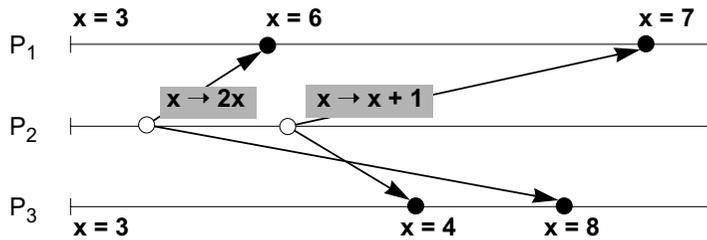
Empfänger r : Realisierung des Nachrichtenempfangs

- **receive($N, s, sequ_num$);**
wenn r noch kein **deliver(N)** für $sequ_num$ ausgeführt hat, dann:
wenn $r \neq s$ dann **send($N, s, sequ_num$)** an alle Nachbarn von r ;
Nachricht an die Anwendungsebene ausliefern (“deliver(N)”);

- Prinzip: “Fluten” des Netzes
 - vgl. dazu “Echo-Algorithmus” und Vorlesung “Verteilte Algorithmen”
- Beachte: **receive \neq deliver**
 - unterscheide Anwendungsebene und Transportebene
- Fragen:
 - müssen die Kommunikationskanäle bidirektional sein?
 - wie effizient ist das Verfahren (Anzahl der Einzelnachrichten)?
 - wie fehlertolerant? (wieviel darf kaputt sein / verloren gehen...?)
 - Optimierungen? Varianten?
 - kann man das gleiche auch ganz anders erreichen?

Broadcast: Empfangsreihenfolge

- Wie ist die Empfangsreihenfolge von Nachrichten?
 - problematisch wegen der i.Allg. ungleichen Übermittlungszeiten
 - Bsp.: Update einer replizierten Variablen mittels "function shipping":

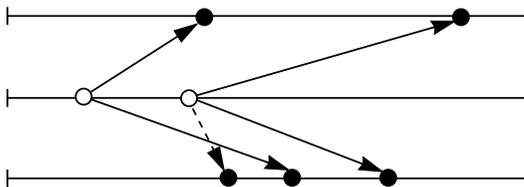


- Es sind verschiedene "Ordnungsgrade" denkbar
 - z.B. ungeordnet, FIFO, kausal geordnet, total geordnet

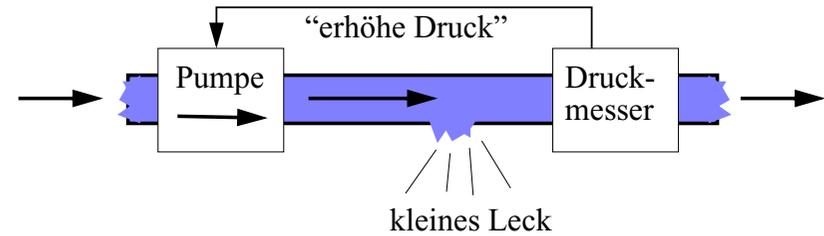
- FIFO-Ordnung:

Alle Multicast-Nachrichten eines (d.h.: *ein und des selben*) Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in FIFO-Reihenfolge an

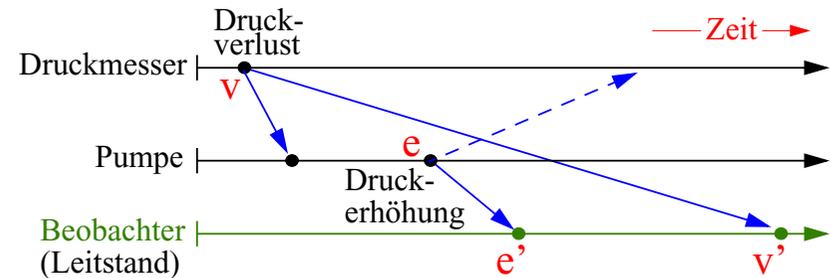
- Denkübung: wie dies in einem Multicast-Protokoll garantieren?



Probleme mit FIFO-Broadcasts



- Annahme: Steuerelemente kommunizieren über FIFO-Broadcasts:



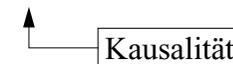
- "Irgendwie" kommt beim Beobachter die Reihenfolge durcheinander!

⇒ *Falsche Schlussfolgerung des Beobachters:*

"Aufgrund einer unbegreiflichen Pumpenaktivität wurde ein Leck erzeugt, wodurch schliesslich der Druck absank."

Man sieht also:

- FIFO-Reihenfolge reicht oft nicht aus, um Semantik zu wahren
- eine Nachricht *verursacht* oft das Senden einer anderen

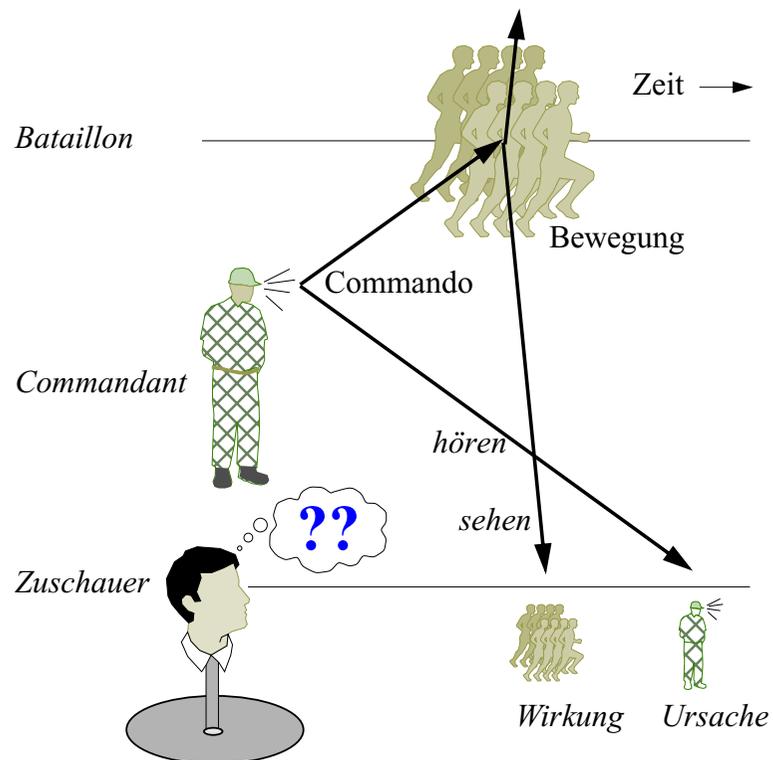


Das “Broadcastproblem” ist nicht neu

- Licht- und Schallwellen werden in natürlicher Weise per Broadcast verteilt
- Wann handelt es sich dabei um FIFO-Broadcasts?
- Wie ist es mit dem Kausalitätserhalt?

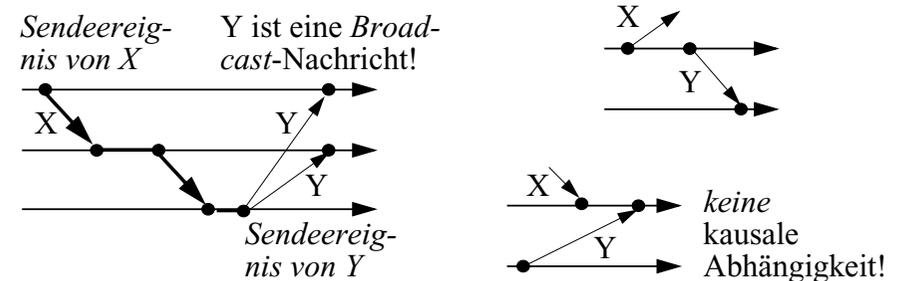
Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so **sieht** er übereinstimmende Bewegungen desselben plötzlich eintreten, **ehe** er die Commandostimme oder das Hornsignal **hört**; aber aus seiner Kenntnis der **Causalzusammenhänge** weiß er, daß die Bewegungen die **Wirkung** des gehörten Commandos sind, dieses also jenen **objectiv** vorangehen muß, und er wird sich sofort der Täuschung bewußt, die in der **Umkehrung der Zeitfolge in seinen Perceptionen** liegt.

Christoph von Sigwart (1830-1904) *Logik* (1889)



Kausale Nachrichtenabhängigkeit

- Definition:
Nachricht Y hängt kausal von Nachricht X ab, wenn es im Raum-Zeit-Diagramm einen von links nach rechts verlaufenden Pfad gibt, der vom Sendereignis von X zum Sendereignis von Y führt



Beachte:

- Dies lässt sich bei geeigneter Modellierung auch abstrakter fassen (→ logische Zeit später in der Vorlesung und auch Vorlesung “Verteilte Algorithmen”)

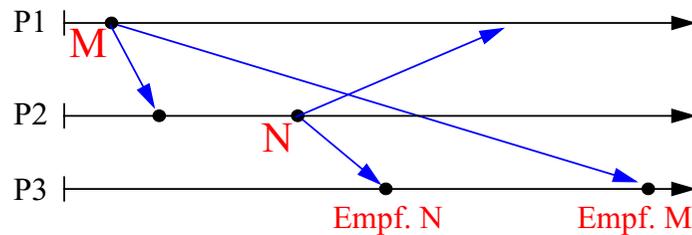
Kausaler Broadcast

Wahrung von Kausalität bei der Kommunikation:

- *Kausale Reihenfolge (Def.):* Wenn eine Nachricht N kausal von einer Nachricht M abhängt, und ein Prozess P die Nachrichten N und M empfängt, dann muss er M vor N empfangen haben

“causal order”

- “Kausale Reihenfolge” (und “kausale Abhängigkeit”) lassen sich insbesondere auch auf *Broadcasts* anwenden:

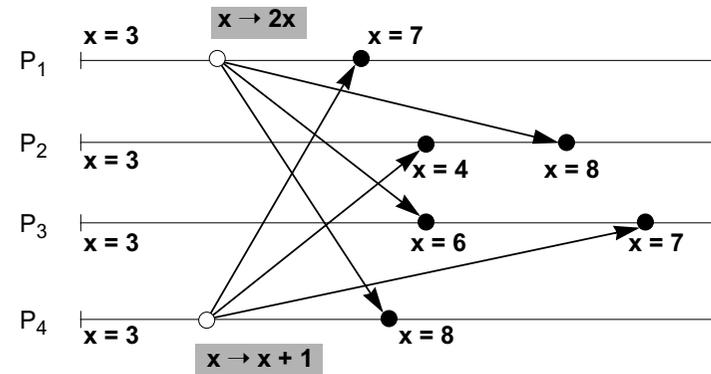


Gegenbeispiel: *Keine* kausalen Broadcasts

- Kausale Reihenfolge *impliziert FIFO-Reihenfolge*: kausale Reihenfolge ist eine Art “globales FIFO”
- Das *Erzwingen* der kausalen Reihenfolge ist mittels geeigneter Algorithmen möglich (→ Vorlesung “Verteilte Algorithmen”, z.B. Verallgemeinerung der Sequenzzählermethode für FIFO)

Probleme mit kausalen Broadcasts ?

Beispiel: Aktualisierung einer replizierten Variablen x:



Problem: Ergebnis statt *überall entweder 7 oder 8* hier nun “beides”!

Konkrete Ursache des Problems:

- Broadcasts werden nicht überall “gleichzeitig” empfangen
- dies führt lokal zu verschiedenen Empfangsreihenfolgen

Abstrakte Ursache:

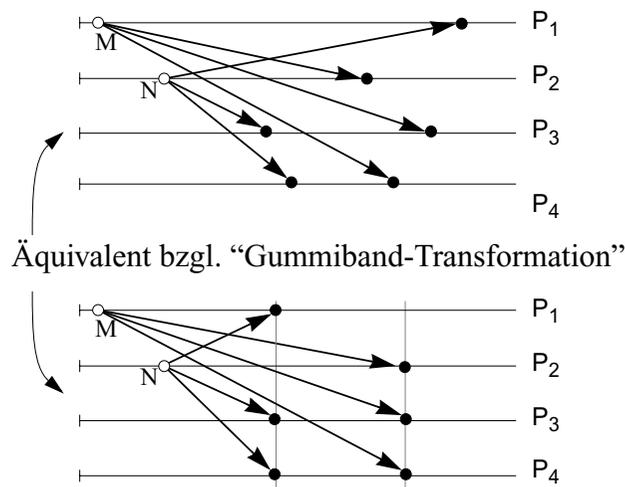
- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

- auch kausale Broadcasts haben keine “perfekte” Semantik (d.h. Illusion einer speicherbasierten Kommunikation)

Atomarer bzw. "totaler" Broadcast

- *Totale Ordnung*: Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 die Nachricht M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt
 - das Senden wird dabei *nicht* als Empfang der Nachricht beim Sender selbst gewertet!
- Beachte: "Atomar" heisst hier *nicht* "alles oder nichts" (wie etwa beim Transaktionsbegriff von Datenbanken!)



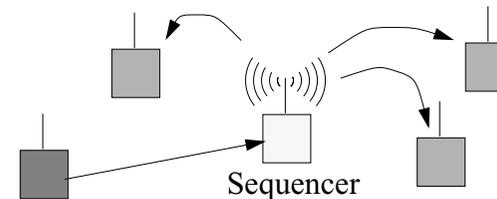
Anschaulich:

- Nachrichten eines Broadcasts werden "überall gleichzeitig" empfangen

Realisierung von atomarem Broadcast

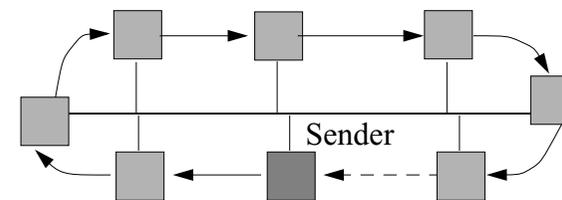
1) Zentraler „Sequencer“, der Reihenfolge festlegt

- ist allerdings ein potentieller Engpass!



- "Unicast" vom Sender zum Sequencer
- Multicast vom Sequencer an alle
- Sequencer *wartet* jew. auf alle Acknowledgements (oder genügt hierfür FIFO-Broadcast?)

2) Token, das auf einem (logischen) Ring kreist



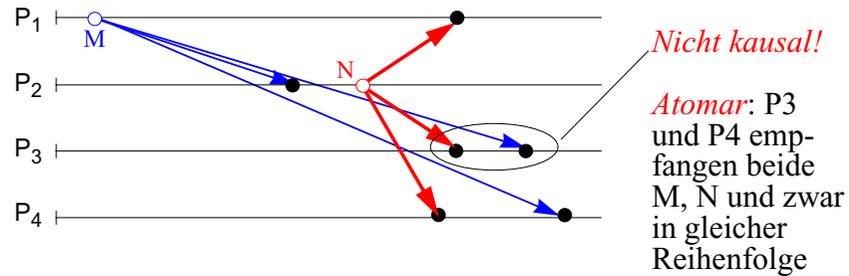
- Token = Senderecht (Token weitergeben!)
- Broadcast selbst könnte z.B. über ein zugrundeliegendes broadcast-fähiges Medium erfolgen

- Token führt eine Sequenznummer (inkrementiert beim Senden), dadurch sind alle Broadcasts *global nummeriert*
- Empfänger wissen, dass Nachrichten entsprechend der (in den Nachrichten mitgeführten Nummer) ausgeliefert werden müssen
- bei Lücken in den Nummern: dem Token einen Wiederholungswunsch mitgeben (Sender erhält damit implizit ein Acknowledgement)
- Tokenverlust (z.B. durch Prozessor-Crash) durch Timeouts feststellen (Vorsicht: Token dabei nicht versehentlich verdoppeln!)
- einen gecrashten Prozessor (der z.B. das Token nicht entgegennimmt) aus dem logischen Ring entfernen
- Variante (z.B. bei vielen Teilnehmern): Token auf Anforderung direkt zusenden (broadcast: "Token bitte zu mir"), dabei aber Fairness beachten

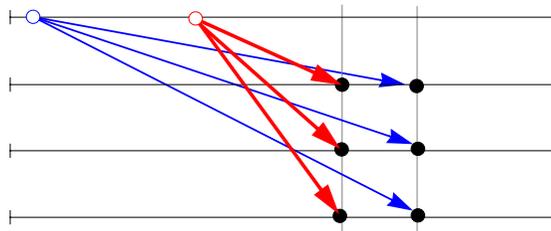
- Geht es auch ohne zentrale Elemente (Sequencer, Token)?

Wie “gut” ist atomarer Broadcast?

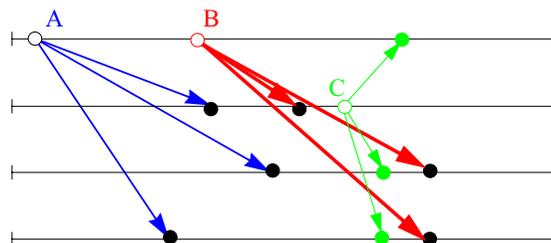
1) Ist **atomar** auch **kausal**?



2) Ist **atomar** wenigstens **FIFO**?



3) Ist **atomar + FIFO** vielleicht **kausal**?



Bem.: 1) ist ebenfalls ein Gegenbeispiel, da M, N FIFO-Broadcast ist!

Kausaler atomarer Broadcast

- Fazit:

- atomare Übermittlung $\not\Rightarrow$ kausale Reihenfolge
- atomare Übermittlung $\not\Rightarrow$ FIFO-Reihenfolge
- atomare Übermittlung + FIFO $\not\Rightarrow$ kausale Reihenfolge

- Vergleich mit speicherbasierter Kommunikation:

- Kommunikation über gemeinsamen Speicher ist *atomar* (alle „sehen“ das Geschriebene gleichzeitig)
- Kommunikation über gemeinsamen Speicher *wahrt Kausalität* (die Wirkung tritt unmittelbar mit der Ursache, dem Schreibereignis, ein)

- Vergleichbares Kommunikationsmodell per Nachrichten:
Kausaler atomarer Broadcast

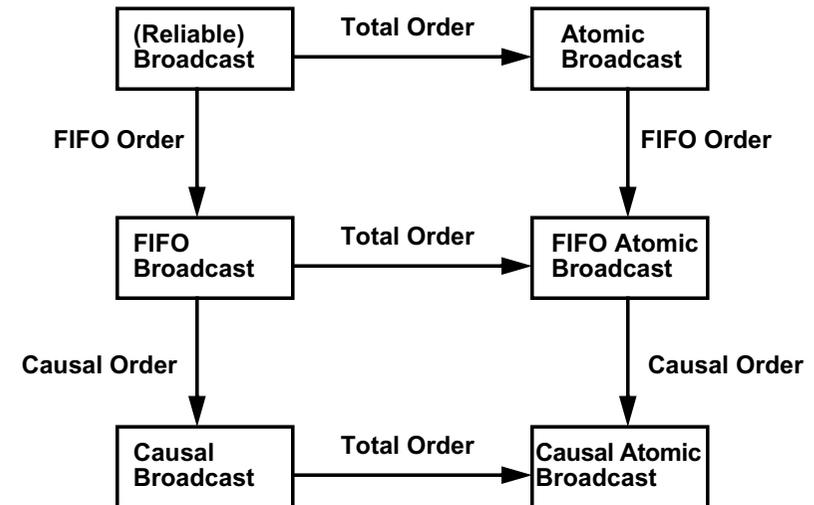
- kausaler Broadcast + totale Ordnung
- man nennt daher kausale, atomare Übermittlung auch *virtuell synchrone Kommunikation*
- Denkübung: realisieren die beiden Implementierungen “zentraler Sequencer” bzw. “Token auf Ring” die virtuell synchrone Kommunikation?

Stichwort: Virtuelle Synchronität

- Idee: Ereignisse finden zu verschiedenen Realzeitpunkten statt, aber zur *gleichen logischen Zeit*
 - *logische Zeit* berücksichtigt nur die Kausalstruktur der Nachrichten und Ereignisse; die exakte Lage der Ereignisse auf dem “Zeitstrahl” ist verschiebbar (Dehnen / Stauchen wie auf einem Gummiband)
- *Innerhalb* des Systems ist synchron (im Sinne von “gleichzeitig”) und virtuell synchron *nicht unterscheidbar*
 - identische totale Ordnung aller Ereignisse
 - identische Kausalbeziehungen
- Folge: Nur mit Hilfe Realzeit / echter Uhr könnte ein externer Beobachter den Unterschied feststellen

Den Begriff “logische Zeit” werden wir später noch genauer fassen (mehr dazu dann wieder in der Vorlesung “Verteilte Algorithmen”)

Broadcast - schematische Übersicht



- Warum nicht ein einziger Broadcast, der alles kann? “Stärkere Semantik“ hat auch Nachteile:
 - Performance-Einbussen
 - Verringerung der potentiellen Parallelität
 - aufwendiger zu implementieren
 - Bekannte “Strategie”:
 - man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Lösung
 - Motto: so billig wie möglich, so „perfekt“ wie nötig
 - man sollte aber die Schwächen einer Billiglösung kennen!
- ⇒ grössere Vielfalt ⇒ komplexer bzgl. Verständnis und Anwendung