

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
- Implementierung:
 - Eingangspuffer wird nach passender Nachricht durchsucht, *oder*:
 - Sender und Empfänger “verhandeln”

- Vorteil bei der Anwendung:

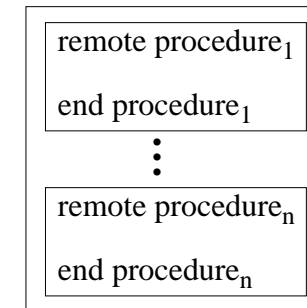
Empfänger muss nicht mehr alles akzeptieren und zwischenspeichern, sondern bekommt nur das, was ihn momentan interessiert

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



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

Gelegentlich ist allerdings ein explizites Empfangen (“receive”) wesentlich bequemer als ein implizites Empfangen!

- *Semantik*:

- Interne Parallelität?

“Routine”

- Mehr als eine gleichzeitig aktive Prozedur, Operation, thread... im Empfänger?
- Vielleicht sogar mehrere Instanzen der gleichen Routine?

- Atomare Routinen?

- Wird eine aktive Routine ggf. unterbrochen, um eine andere aktivierte auszuführen?

Sprachaspekte

- Kommunikation (und Synchronisation) soll in geeigneter Weise sprachlich unterstützt werden
-

1) Aufruf von Primitiven des Betriebs- oder Kommunikationssystems

- bzw. allgemein gebräuchliche Kommunikationsbibliothek

2) Ergänzung einer existierenden Sprache

- zusätzliche Kommunikationsanweisungen (Aufruf von Methoden / Prozeduren)
 - oder gar syntaktische Erweiterungen (--> Änderungen am Compiler)
- } + ggf. neues Laufzeitsystem

3) Eigene Programmiersprache

- “aus einem Guss” (Homogenität der Konzepte; syntaktisch und semantisch harmonische Integration)
 - sichereres Programmieren --> bessere (robustere) Software
 - Beispiele: CSP, Occam, Ada (allerdings eher weniger gebräuchlich)
-

Dazu festlegen:

- Syntax und Semantik der Kommunikation
- Art der kommunizierenden Objekte
- Harmonisierung mit “sequentiellen” Konzepten

Sprachdesign: Modularisierung

- Einheiten der

- Strukturierung

- bezüglich Softwaretechnologieaspekten (information hiding, Schnittstellen, getrennte Übersetzung...)

- Verteilung

- gemeinsamer Namensraum
- Cluster “lokaler” und zusammengehöriger Objekte
- nur als ganze Einheit auf Rechner platzierbar

- Parallelität

- mehrere Module arbeiten nebenläufig (“gleichzeitig”)
 - z.B. thread, Prozess,...
-

- Getrennte oder einheitliche Sprachmittel dafür?

- Gesichtspunkte wie Zweckmässigkeit, Anwendungsbereich, Effizienz, Sicherheit, Einfachheit...

- Hierarchien, Schachtelungen von Modulen?

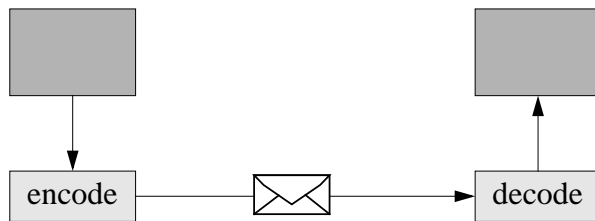
- Granularität; interne Parallelität?

- Dynamik <--> Statik ?

- Ports, Kanäle, Mailboxen... dynamisch erzeugbar?
- Dynamik ist flexibler, aber aufwendiger in der Realisierung und problematischer in der Benutzung (Benennung, “dangling references”)

Kommunizierbare Datentypen?

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



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

- Was ist mit *komplexen Datentypen* wie

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

- Ggf. “Linearisieren” und ggf. Strukturbeschreibung mitschicken (u.U. sprachunabhängig, z.B. mit ASN.1)

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

“first class objects”

Ausflug: ASN.1

ASN.1 ist eine Sprache zur Beschreibung strukturierter Information, die vor allem zur Spezifikation von Kommunikationsprotokollen eingesetzt wird.

Auszug aus: <http://whatis.com/asn1.htm>:

ASN.1 (Abstract Syntax Notation One) is a standard way to describe a message (a unit of application data) that can be sent or received in a network. ASN.1 is divided into two parts:

(1) the rules of syntax for describing the contents of a message in terms of data types and content sequence or structure and

(2) how you actually encode each data item in a message.

ASN.1 is defined in two ISO standards for applications intended for the Open Systems Interconnection (OSI) framework.

Here's an example of a message definition specified with ASN.1 notation:

```
Report ::= SEQUENCE {
    author    OCTET STRING,
    title     OCTET STRING,
    body      OCTET STRING,
    biblio    Bibliography
}
Bibliography ::= SEQUENCE {
    author    OCTET STRING
    title     OCTET STRING
    publisher OCTET STRING
    year      OCTET STRING
}
```

Auszug aus: <http://www.techapps.co.uk/asn1gloss.html>:

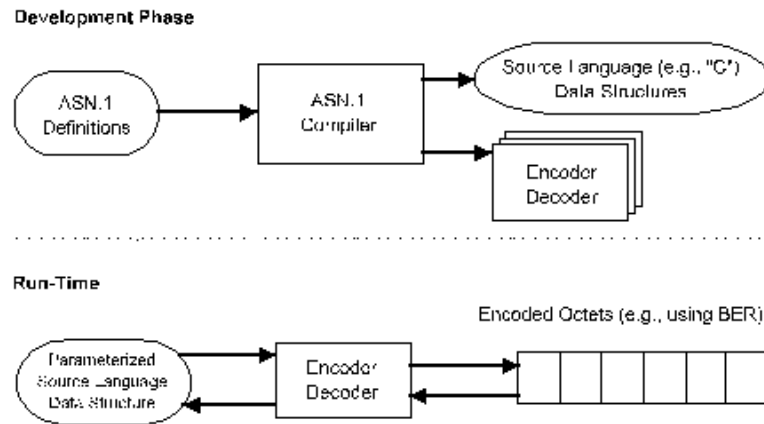
Prior to ASN.1, information to be conveyed in communication protocols was typically specified by ascribing meanings to particular bits and bytes in protocol messages, much as programmers, before the advent of high level languages, had to deal with the bits and bytes of storage layout.

With ASN.1, the protocol designer can view and describe the relevant information and its structure at a high level and need not be unduly concerned with how it is represented while in transit.

ASN.1 comes into its own when the information being described is complex. This is because the language allows arbitrarily complex structures to be built up in a uniform way from simpler components, and ultimately from a few simple information types. ASN.1 is, in effect, a data definition language, allowing a designer to define the parameters in a protocol data unit without concern as to how they are encoded for transmission.

Given any ASN.1 description of a message, a representation can be derived mechanically by applying a set of encoding rules. While many such sets could be imagined, initially only a single set, the Basic Encoding Rules (BER), were standardised as a companion standard to ASN.1.

ASN.1 (2)



The Use of ASN1 Compilers

ASN.1 notation is recognisable as a high level definition language. It is constructed in modules with unique identifiers. There are over 20 built data types such as Simple data types (BOOLEAN, INTEGER, ENUMERATED, REAL, BIT STRING, OCTET STRING, NULL, ...), Character strings (NumericString, PrintableString...)

Arbitrarily complex structures can be built up from these data types using constructors such as:

SET{} - order not significant

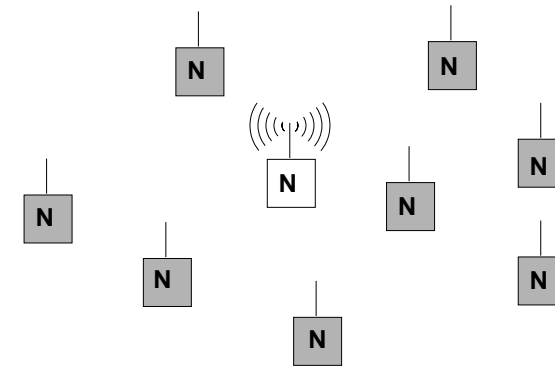
SEQUENCE{} - fixed order

A type may be simple or structured. The simple types are the basic building blocks of ASN.1, and include types like Boolean and integer. A simple type will generally be used to describe a single aspect of something. A structured type, on the other hand, is defined in terms of other types - its components - and its values are made up of values of the component types. Each of these components may itself be simple or structured, and this nesting can proceed to an arbitrary depth, to suite the needs of the application. All structured types are ultimately defined in terms of simple types.

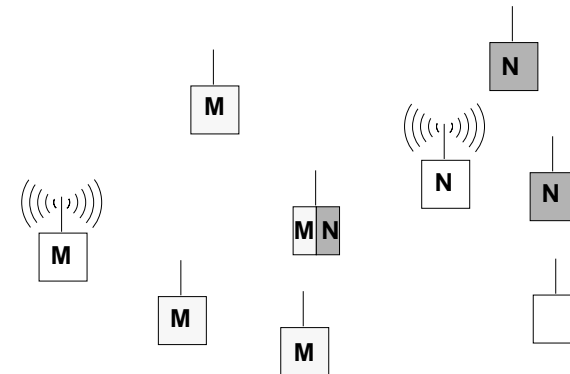
ASN.1 makes available to the abstract syntax designer a number of simple types, as well as techniques for defining structured types and subtypes. The designer employs these types by using the type notation which ASN.1 provides for each such type.

Earlier standards such as ASCII and EBCDIC specified both the abstract syntax (the letter A) and the encoding, or transfer syntax, (hexadecimal 21 or 41). ASN.1 separates these two concepts, such that at connect time you can chose to encode the data. You can chose an encoding which is efficient on the line or reliable or easy to decode. The first defined for ASN.1 was the Basic Encoding Rules (BER). The BER allow the automatic derivation of a transfer syntax for every abstract syntax defined using ASN.1.

Gruppenkommunikation



Broadcast: Senden an die *Gesamtheit* aller Teilnehmer



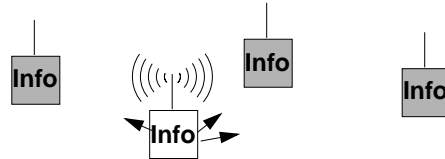
Multicast: Senden an eine *Untergruppe* aller Teilnehmer

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

Anwendungen von Gruppenkommunikation

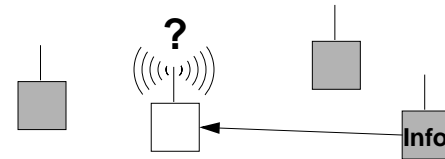
- Informieren

- z.B. Newsdienste, Konferenzsysteme etc.

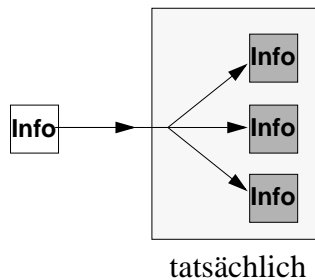
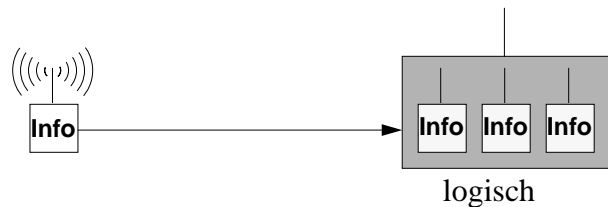


- Suchen

- z.B. Lokation von Objekten und Diensten



- "Logischer Unicast" an replizierte Komponenten

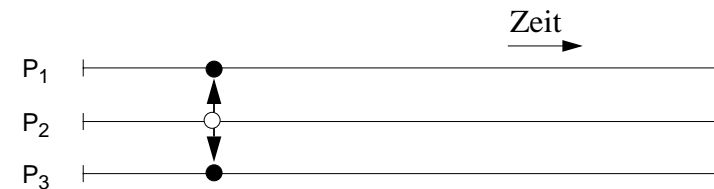


Typische Anwendungs-
klasse von Replikation:
Fehlertoleranz

Gruppenkommunikation - idealisierte Semantik

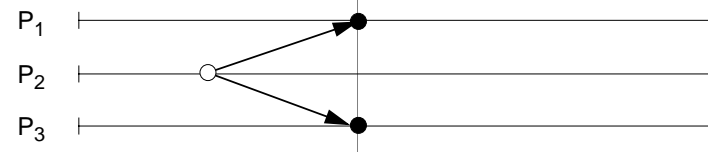
- Modellhaftes Vorbild: Speicherbasierte Kommunikation in zentralistischen Systemen

- augenblicklicher Empfang
- vollständige Zuverlässigkeit



- Nachrichtenbasierte Kommunikation: Idealisierte Sicht

- (verzögerter) gleichzeitiger Empfang
- vollständige Zuverlässigkeit

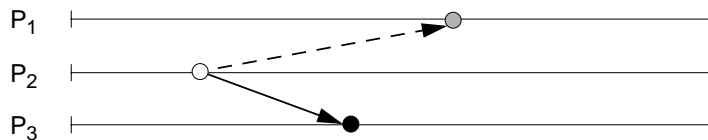


Gruppenkommunikation - tatsächliche Situation

- Medium (Netz) ist oft nicht multicastfähig
 - LANs teilweise (z.B. klassisches Ethernet), jedoch i.a. nur innerhalb eines einzigen Segments (vgl. aber VLANs)
 - multicastfähiges Netz ist effizient (Hardwareunterstützung), typischerweise jedoch 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 einer speicherbasierten Kommunikation
- möglichst hohe Verlässlichkeit und Effizienz

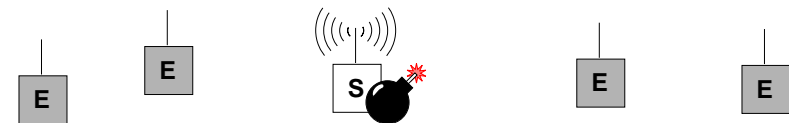
- Beachte: Verlust von Nachrichten und sonstige Fehler sind bei Broadcast ein viel wichtigeres Problem als beim "Unicast"! (Wieso?)

- Hauptproblem bei der Realisierung von Broadcast:
Zuverlässigkeit und garantierte Empfangsreihenfolge

Senderausfall beim Broadcast

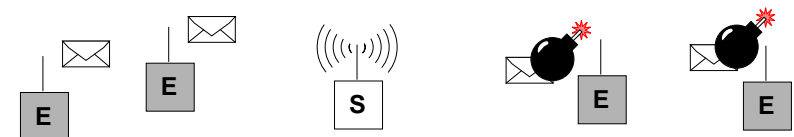
a) Sender fällt aus: *kein* Empfänger erhält Nachricht

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



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

- wenn Broadcast durch Senden vieler Einzelnachrichten realisiert ist
- "ungünstiger" Fall: *Uneinigkeit*



- mögliche Abhilfe: Empfänger leiten die Nachricht untereinander weiter (Denkübung: Müssen die Empfänger dazu wissen, ob ein Sender abgestürzt ist? Wenn ja: wie? Wenn nein: schadet eine falsche Verdächtigung nicht?)

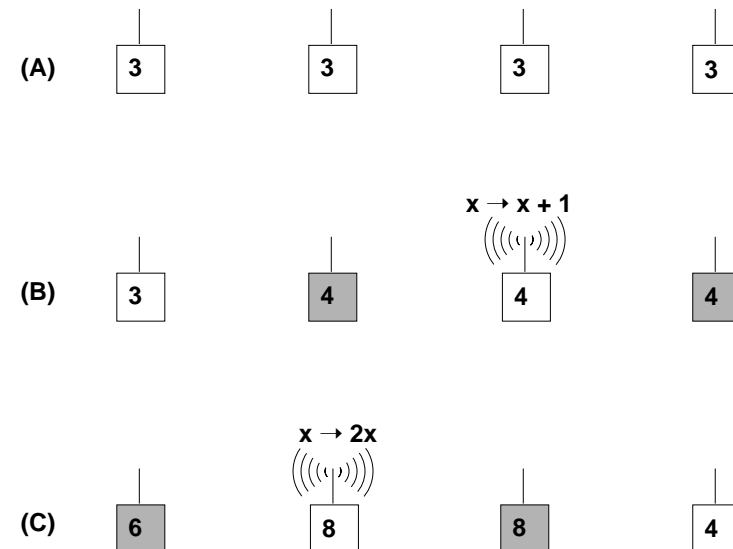
- Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche Folgeprobleme sein! (Da wäre es manchmal besser, *kein* Prozess hätte die Nachricht empfangen!)

Zuverlässigkeitsstrategie “Best effort” bei Broadcasts

- Fehlermodell: Verlust von Nachrichten (und ggf. 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 im Fehlerfall 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. Lastdaten oder unverbindliche “Tips“ und “Empfehlungen“
 - Information, die ggf. Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen
 - Ggf. als Grundlage zur Realisierung höherer Protokolle
 - oft basierend auf multicastfähigen Netzen
 - günstig bei zuverlässigen physische Kommunikationsmedien (wenn Fehlerfall sehr selten --> aufwendiges Recovery auf höherer Ebene tolerierbar)

k-Zuverlässigkeit

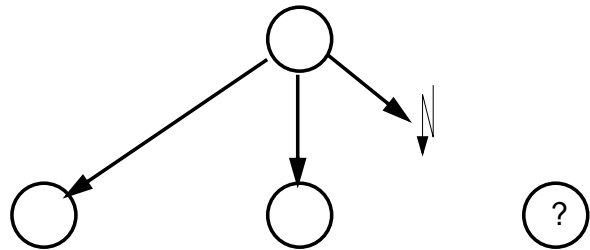
- Mindestens k Teilnehmer haben die Nachricht empfangen
 - grössere Werte von k sind “teurer”
 - Denkübungen: Wie realisiert man einen k-zuverlässigen Multicast? Ist ein “100%ig zuverlässiger” Broadcast überhaupt möglich? Wo lässt sich dies (für welches k?) sinnvoll verwenden?
- Problem der Fehlerakkumulation:
 - der Zustand (repräsentiert durch eine Variable x) sei repliziert
 - Zustandssynchronisation werde durch “function shipping” mittels 2-zuverlässigem Multicast realisiert



- Ergebnis nach einiger Zeit: Alle Replikate sind verschieden!
- in einem solchen Fall hilft also k-Zuverlässigkeit nicht viel

“Reliable Broadcast”

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



- Quittung (“positives Acknowledgement”: ACK) für jede Einzelnachricht ist teuer
 - im Verlustfall einzeln nachliefern oder (falls broadcastfähiges Medium vorhanden) einen zweiten Broadcast-Versuch? (--> Duplikaterkennung!)
- 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
 - “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: Knoten und Verbindungen mit Fail-Stop-Charakteristik

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

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

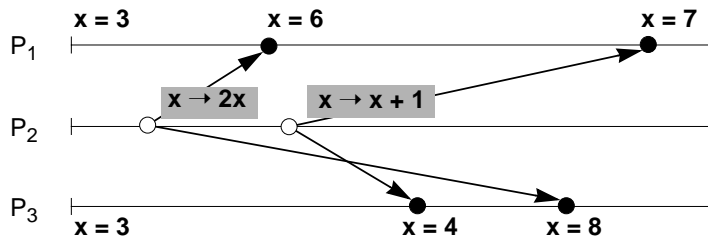
Empfänger r : Realisierung des Nachrichtenempfangs

- **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 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)?
 - Optimierungen? Varianten?
 - wie fehlertolerant? (wieviel darf kaputt sein / verloren gehen...?)
 - kann man das gleiche auch ganz anders erreichen?

Empfangsreihenfolge

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

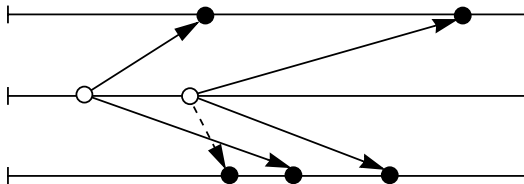


- Es sind verschiedene "Ordnungsgrade" denkbar
 - z.B. ungeordnet, FIFO, kausal geordnet, total geordnet

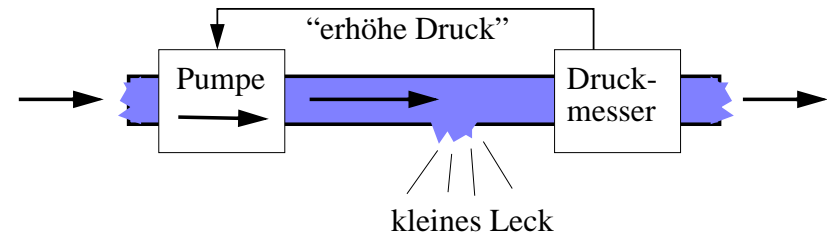
- FIFO-Ordnung:

Alle Multicast-Nachrichten *eines* Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in FIFO-Reihenfolge an

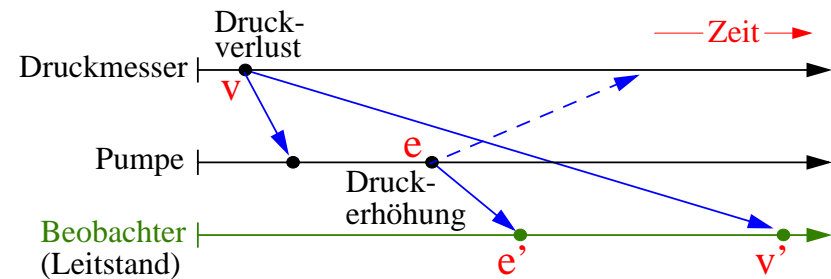
- Denkübung: wie dies in einem Multicast-Protokoll garantieren?



Probleme mit FIFO-Broadcasts



- Annahme: Kontrollinstanzen kommunizieren über FIFO-Broadcasts:



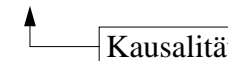
- "Irgendwie" kommt beim Beobachter die Reihenfolge durcheinander!

==> *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

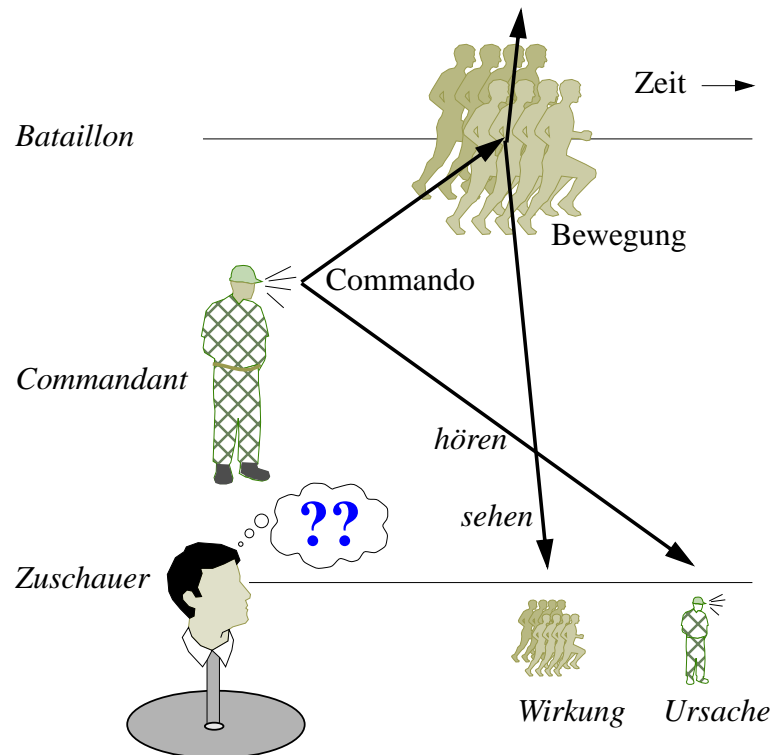


Das “Broadcastproblem” ist nicht neu

- 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 Exerzieren 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.

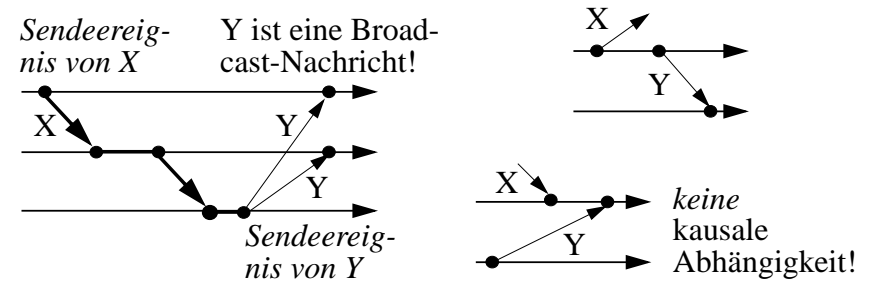
Christoph von Sigwart (1830-1904) *Logik* (1889)



Kausale Broadcasts

Kausale Abhängigkeit zwischen Nachrichten (Def.):

- *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 (--> vgl. L. Lamport und auch Vorlesung “Verteilte Algorithmen”)

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

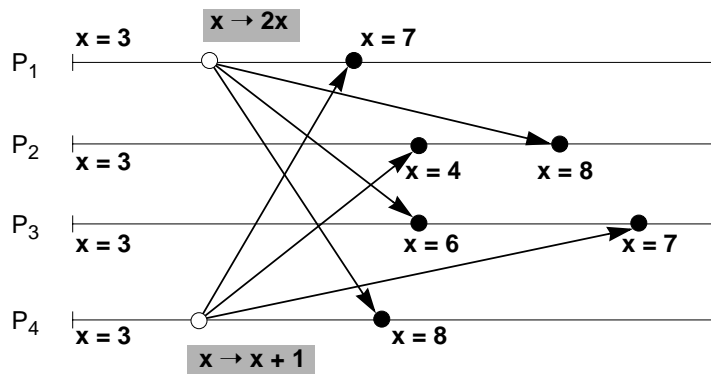
Beachte:

“causal order”

- “Kausale Reihenfolge” und “kausale Abhängigkeit” lassen sich insbesondere auch auf *Broadcasts* anwenden
- Kausale Reihenfolge *impliziert FIFO-Reihenfolge*: kausale Reihenfolge ist eine Art “globales FIFO” (“Dreiecksungleichung”)
- 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: Statt *überall 7 oder 8* als Ergebnis: Hier “beides”!

Konkrete Ursache des Problems:

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

Abstrakte Ursache:

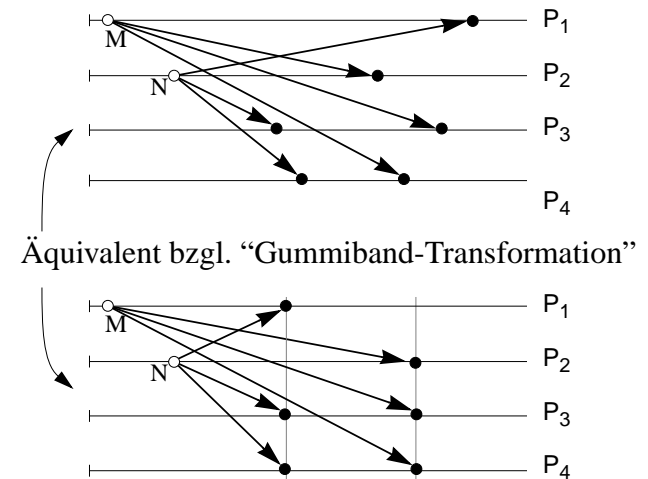
- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

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

Atomarer bzw. “totaler” Broadcast

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



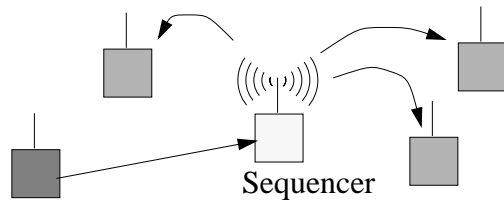
Anschaulich:

- Nachrichten eines Broadcasts werden “überall gleichzeitig” empfangen

Realisierung von atomarem Broadcast

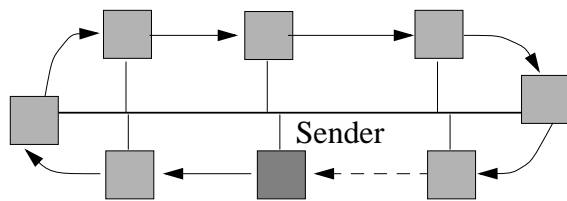
- Zentraler „Sequencer“, der Reihenfolge festlegt

- ist allerdings ein potentieller Engpass!



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

- Token, das auf einem (logischen) Ring kreist



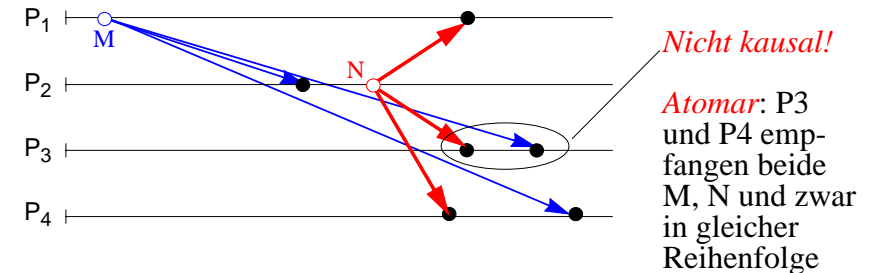
- Token = Senderecht (Token weitergeben!)
- Broadcast selbst könnte z.B. über ein zugrundeliegendes broadcast-fähiges Medium erfolgen

- 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 zu vielen Teilnehmern): Token auf Anforderung direkt zusenden (broadcast: “Token bitte zu mir”), dabei aber Fairness beachten (vgl. analoge Prinzipien bei Algorithmen für den wechselseitigen Ausschluss in Netzen --> Vorlesung “Verteilte Algorithmen”)

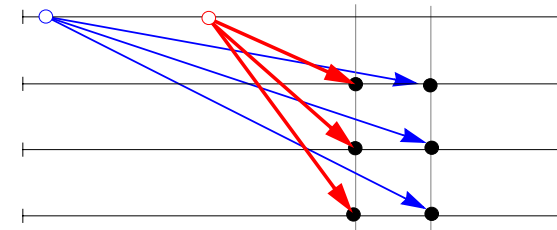
- Geht es auch ohne zentrale Elemente (Sequencer, Token)?

Wie “gut” ist atomarer Broadcast?

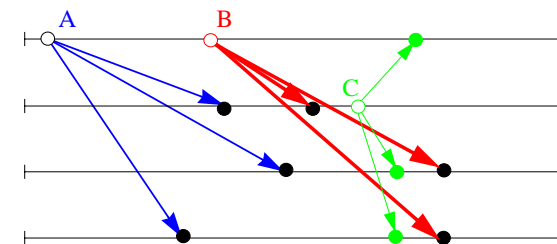
1) Ist **atomar** auch **kausal**?



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 \Rightarrow kausale Reihenfolge
 - atomare Übermittlung \Rightarrow FIFO-Reihenfolge
 - atomare Übermittlung + FIFO \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*
 - Denkbildung: realisieren die beiden Implementierungen „zentraler Sequencer“ bzw. „Token auf Ring“ die virtuell synchrone Kommunikation?

Stichwort: Virtuelle Synchronität

- Idee: Ereignisse finden zu verschiedenen Real-Zeitpunkten statt, aber zur *gleichen logischen Zeit*
 - in Bezug auf die bis dahin empfangenen Nachrichten
 - vorläufig: „logische Zeit“ = Menge aller vergangenen Ereignisse

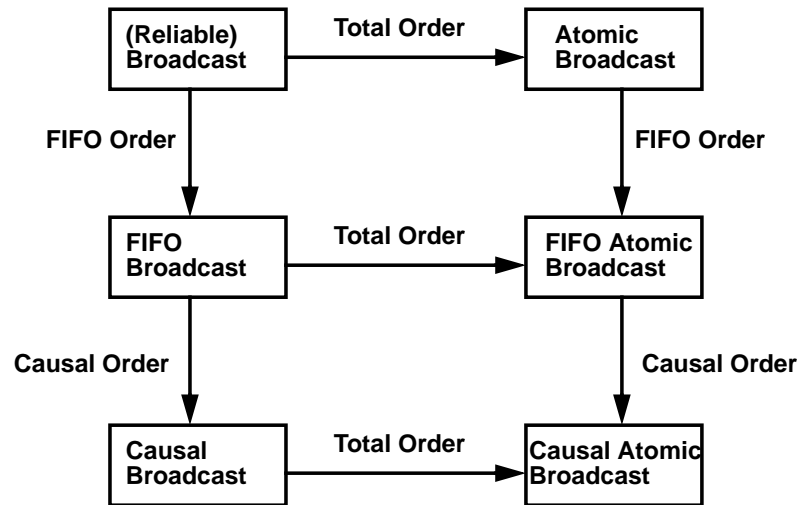
aber in welchem Sinne?

- *Innerhalb* des Systems ist synchron und virtuell synchron *nicht unterscheidbar*
 - identische totale Ordnung aller Ereignisse
 - identische Kausalbeziehungen
- Folge: Nur mit Hilfe einer globalen Uhr könnte ein externer Beobachter den Unterschied feststellen

Den Begriff „logische Zeit“ müssen wir aber irgendwann noch genauer fassen!
(Mehr dazu dann wieder in der Vorlesung „Verteilte Algorithmen“)

Broadcast - schematische Übersicht

Multicast



- Definition von Multicast (grob): “*Multicast ist ein Broadcast an eine Teilmenge von Prozessen*”
- Somit: Alles, was bisher über Broadcast gesagt wurde, gilt (innerhalb der Teilmenge) auch weiterhin:
 - zuverlässiger Multicast
 - FIFO-Multicast
 - kausaler Multicast
 - atomarer Multicast
 - kausaler atomarer Multicast
- Unterschied: Wo bisher “alle Prozesse” gesagt wurde, gilt nun “alle Prozesse innerhalb der Teilmenge”
- Wesentlich: Man kann *verschiedene* Teilmengen bilden
- Neu: Begriff der *Multicast-Gruppe* = Teilmenge von Prozessen

- 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

Multicast-Gruppen

- Funktion einer Multicast-Gruppe

- “Selektiver Broadcast”
- Vereinfachung der Adressierung (z.B. statt Liste von Einzeladressen)
- Verbergen der Gruppenzusammensetzung (vgl. Mailbox/Port-Konzept)
- “Logischer Unicast”: Gruppen ersetzen Individuen (z.B. für transparente Replikation)

- Gruppenadressierung

- *Explizite Benennung*: Sender nennt den Namen der Gruppe (“...grüsse den Kuckuckszuchtverein in Gimbelhausen”)
- *Aufzählung der Mitglieder*: u.U. Multicast über Broadcast-Medium; gestattet dynamische Gruppen (“...grüsse Susi, Hugo & Erni”)
- *Prädikatadressierung*: Ein potentieller Empfänger akzeptiert die Nachricht nur, wenn ein mitgesendetes Prädikat im lokalen Zustand des Empfängers ‘wahr’ ergibt (“...grüsse alle, die mich lieben”)

- Offene / geschlossene Gruppen

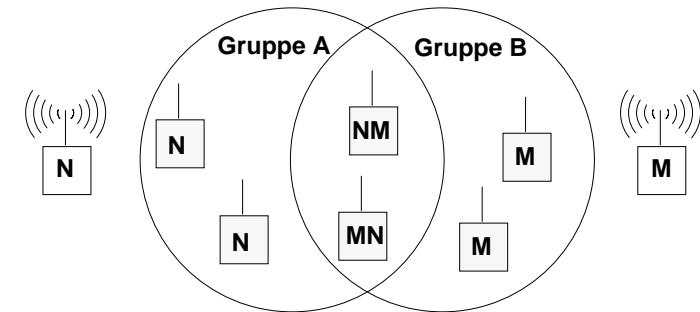
- *Offen*: Nicht-Gruppenmitglieder dürfen Multicast-Nachrichten an die Gruppe senden
- *Geschlossen*: Nur Gruppenmitglieder dürfen...

- Statische / dynamische Gruppen

- *Dynamisch*: Gruppenzusammensetzung ändert sich ggf. im Laufe der Zeit: Gruppeneintritt, Gruppenaustritt, Ausfall eines Gruppenmitglieds; sind schwieriger zu verwalten als statische Gruppen

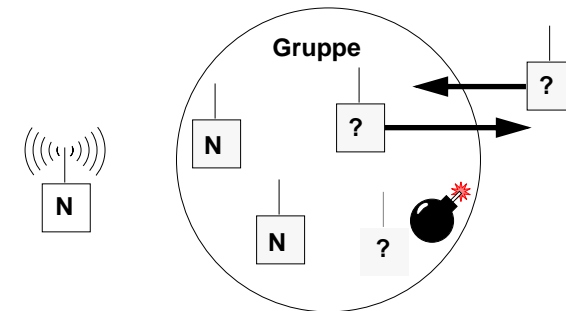
Grundprobleme bei Gruppen

• Gruppenüberlappung



- was genau geschieht im Überlappungsbereich?

• Gruppen-Management und Membership-Problem



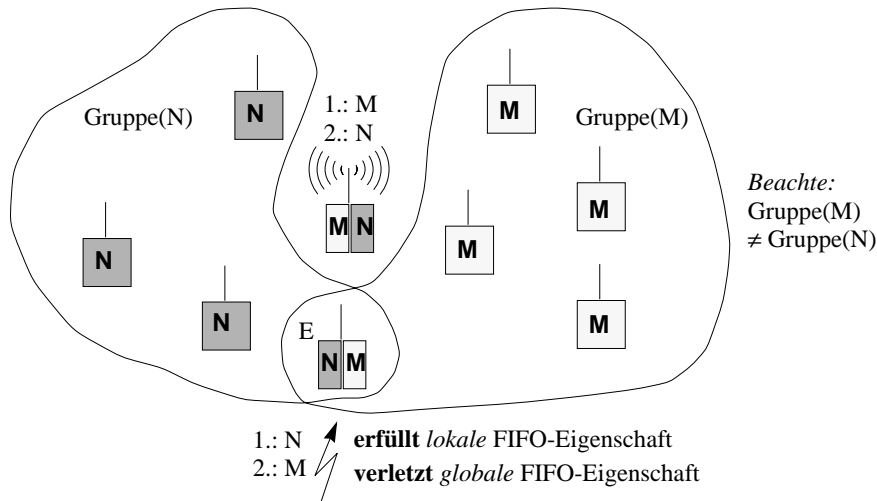
- dynamische Gruppe: wie sieht die Gruppe “momentan” aus?
- haben alle Mitglieder (gleichzeitig?) die gleiche Sicht?
- was tun mit Crashes?

Probleme der Gruppenüberlappung am Beispiel von FIFO-Multicast

was ist sinnvoll?

- Auf was genau soll sich die FIFO-Eigenschaft beziehen?
- Bezeichne $Gruppe(X)$ die Multicast-Gruppe, an die die Nachricht X gesendet wird

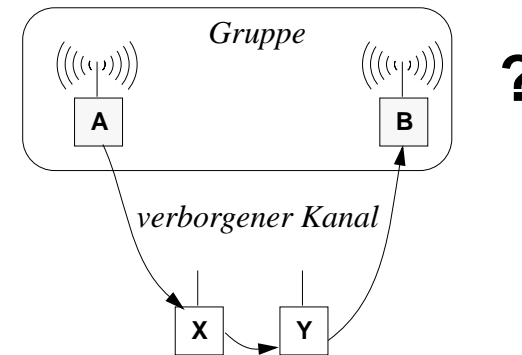
- *Globale FIFO-Reihenfolge*: Wenn ein Prozess erst M und dann N sendet und ein Empfänger in $Gruppe(M)$ die Nachricht N empfängt, dann muss er zuvor auch M empfangen haben
- *Lokale FIFO-Reihenfolge*: Wenn ein Prozess erst M und dann N sendet mit $Gruppe(N) = Gruppe(M)$ und ein Empfänger die Nachricht N empfängt, dann muss er zuvor auch M empfangen haben



- Analoge Unterscheidungen bzgl. lokaler / globaler Gültigkeit auch bei *kausalen* und *atomaren* Multicasts

Problem der "Hidden Channels"

- Kausalitätsbezüge verlassen (z.B. durch Gruppenüberlappung) die Multicast-Gruppe und kehren später wieder



- Soll nun das Senden von B als kausal abhängig vom Senden von A gelten?
- *Global* gesehen ist das der Fall, *innerhalb* der Gruppe ist eine solche Abhängigkeit jedoch nicht erkennbar
- Wie lautet die *sinnvolle* Definition von kausalem Multicast?