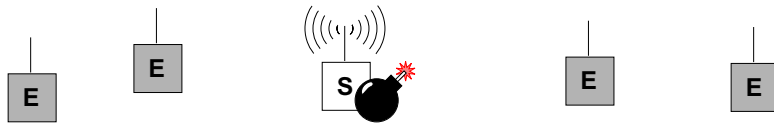


Senderausfall beim Broadcast

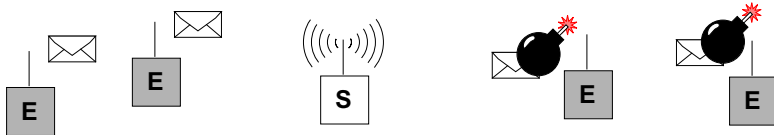
a) Sender fällt ganz aus: *kein* Empfänger erhält Nachricht

- „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 “Empfehlungen“ zur Optimierung

- 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

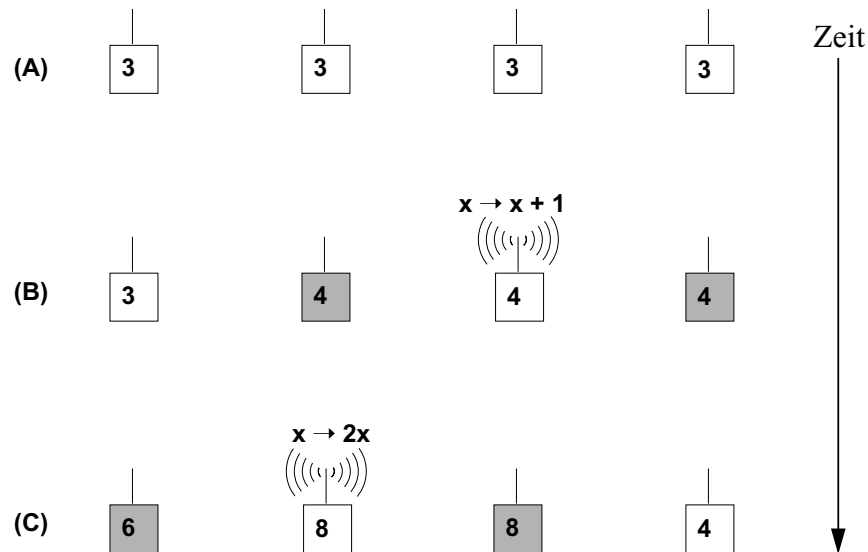
- günstig bei zuverlässigen physische Kommunikationsmedien (wenn Fehlerfall sehr selten → aufwendiges Recovery auf höherer Ebene tolerierbar)

k-Zuverlässigkeit

- Mindestens k Teilnehmer haben die Nachricht empfangen
 - grössere Werte von k sind "teurer"
 - Denkübingen: 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:

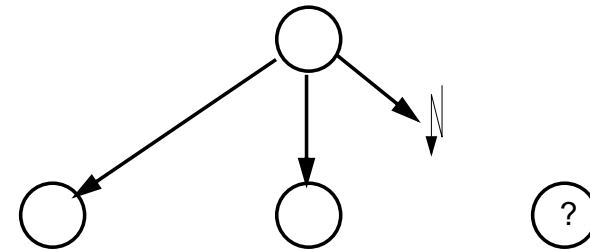
- der Zustand (repräsentiert durch eine Variable x) sei repliziert
- Zustandssynchronisation werde durch "function shipping"



- 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 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: Knoten und Verbindungen mit Fail-Stop-Charakteristik

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

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

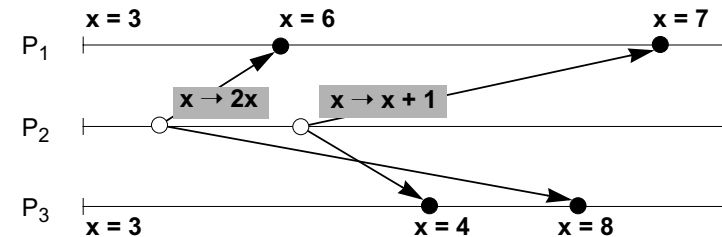
Empfänger r : Realisierung des Nachrichtenempfangs

- **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

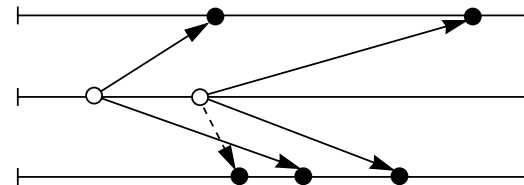
- Wie ist die Empfangsreihenfolge von Nachrichten?
 - problematisch wegen der i.a. ungleichen Übermittlungszeiten
 - Bsp.: Update einer replizierten Variablen mittels "function shipping":



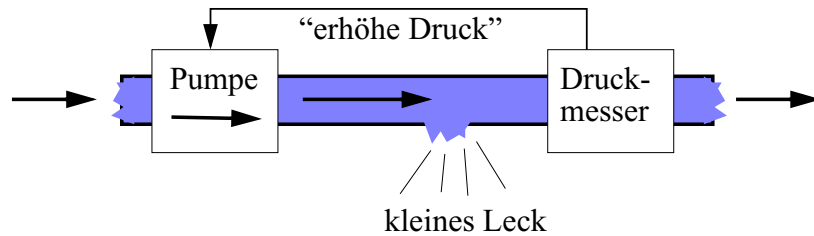
- Es sind verschiedene "Ordnungsgrade" denkbar
 - z.B. ungeordnet, FIFO, kausal geordnet, total geordnet
- **FIFO-Ordnung:**

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

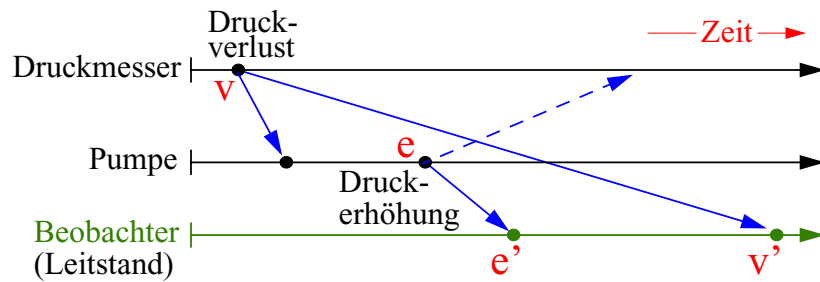
- Denkübung: wie dies in einem Multicast-Protokoll garantieren?



Probleme mit FIFO-Broadcasts



- Annahme: Steuerelemente kommunizieren über FIFO-Broadcasts:



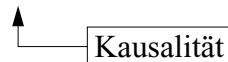
- "Irgendwie" kommt beim Beobachter die Reihenfolge durcheinander!

==> *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

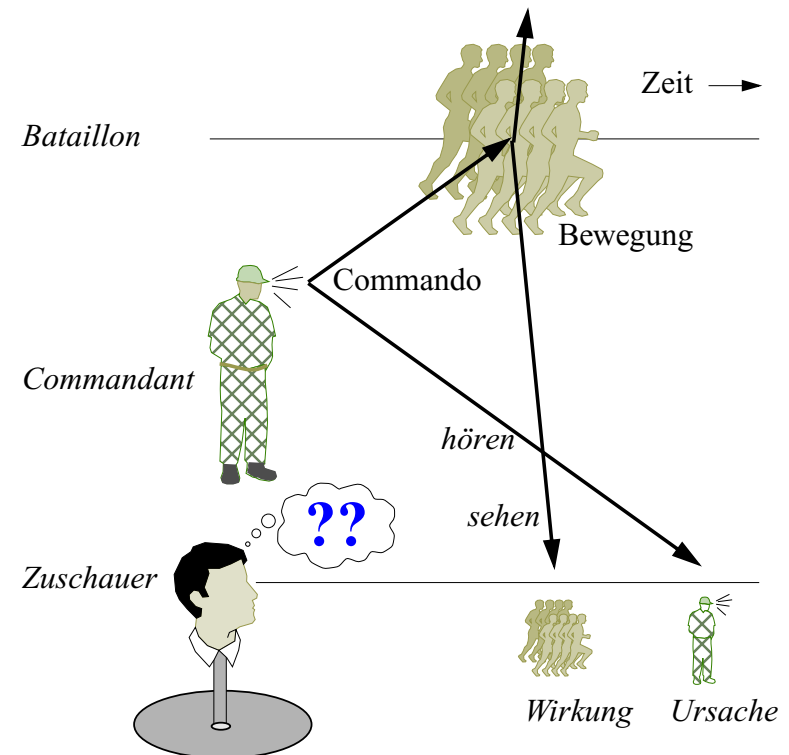


Das "Broadcastproblem" ist nicht neu

- 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, **eh** er die Commandostimme oder das Hornsignal **hört**; aber aus seiner Kenntnis der **Causalzusammenhänge** weiß er, daß die Bewegungen die **Wirkung** des gehörten Commands 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.

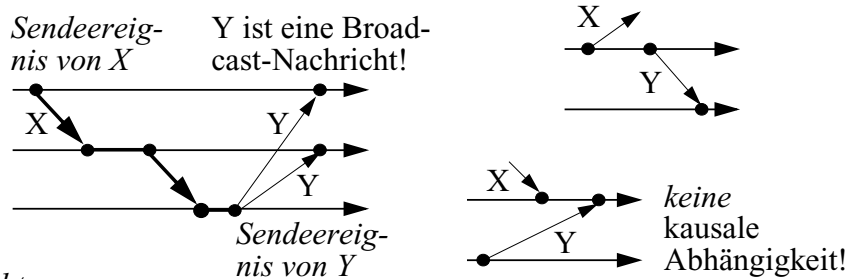
Christoph von Sigwart (1830-1904) *Logik* (1889)



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. logische Zeit später in der Vorlesung und auch Vorlesung “Verteilte Algorithmen”)

Wahrung von Kausalität bei der Kommunikation:

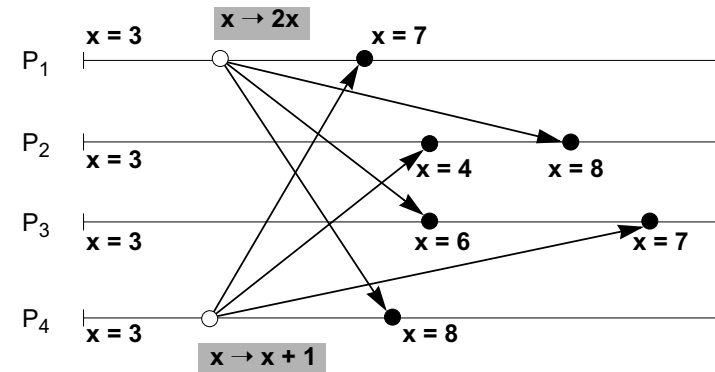
- *Kausale Reihenfolge (Def.)*: Wenn eine Nachricht N kausal von einer Nachricht M abhängt, und ein Prozess P die Nachrichten N und M empfängt, dann muss er M vor N empfangen haben

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”
- 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:

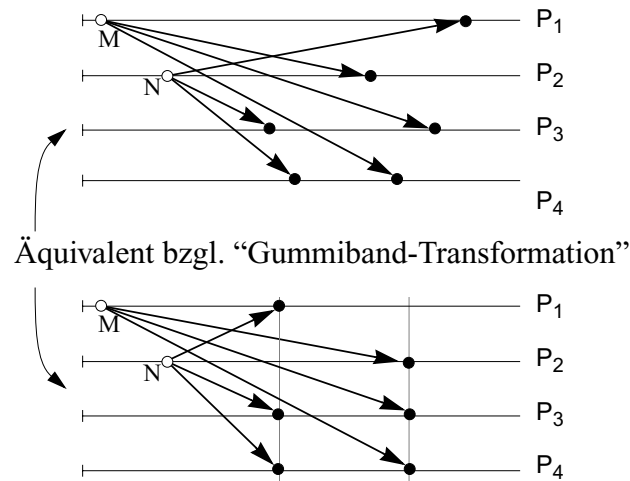
- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

- auch kausale Broadcasts haben keine “perfekte” Semantik (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 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!)



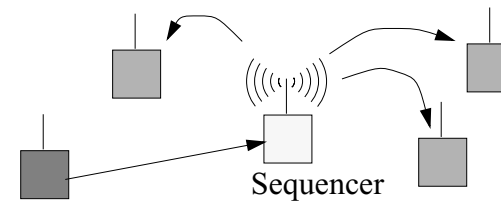
Anschaulich:

- Nachrichten eines Broadcasts werden "überall gleichzeitig" empfangen

Realisierung von atomarem Broadcast

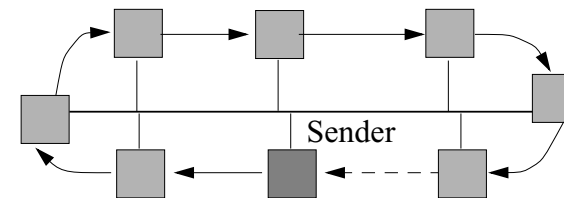
- Zentraler „Sequencer“, der Reihenfolge festlegt

- 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



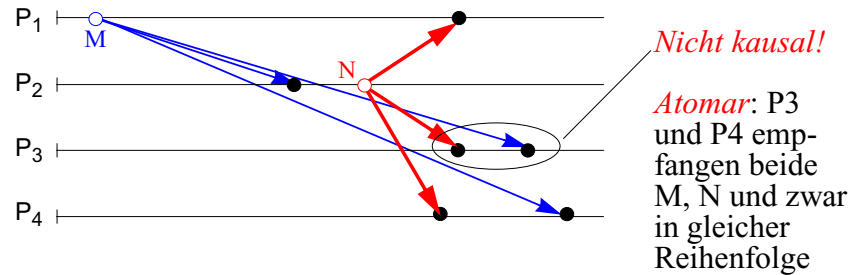
- Token = Senderecht (Token weitergeben!)
- Broadcast selbst könnte z.B. über ein zugrundeliegendes broadcast-fähiges Medium erfolgen

- 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")

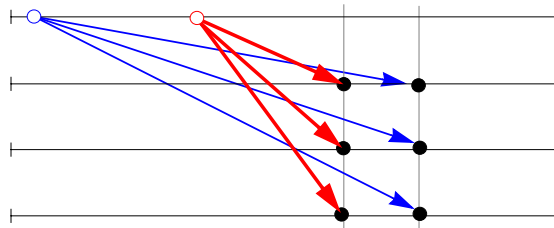
- Geht es auch ohne zentrale Elemente (Sequencer, Token)?

Wie “gut” ist atomarer Broadcast?

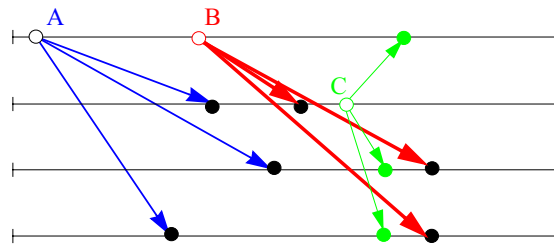
1) Ist **atomar** auch **kausal**?



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 $\neq \Rightarrow$ kausale Reihenfolge
- atomare Übermittlung $\neq \Rightarrow$ FIFO-Reihenfolge
- atomare Übermittlung + FIFO $\neq \Rightarrow$ 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*

- in Bezug auf die bis dahin empfangenen Nachrichten
- vorläufig: „logische Zeit“ = Menge aller vergangenen Ereignisse

aber in welchem Sinne?

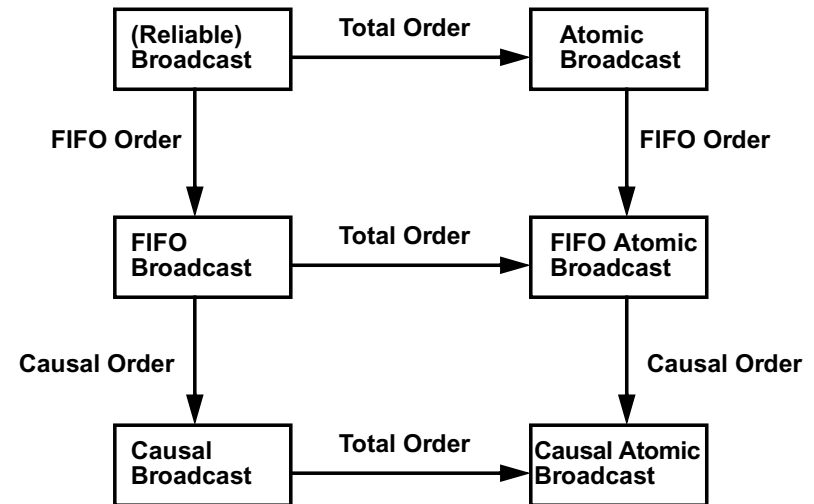
- *Innerhalb* des Systems ist synchron und virtuell synchron *nicht unterscheidbar*

- 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“ werden wir demnächst 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