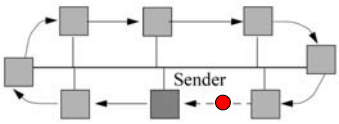
Realisierung von atomarem Broadcast

Verfahren berücksichtigt auch Nachrichtenverluste und gecrashte Prozesse

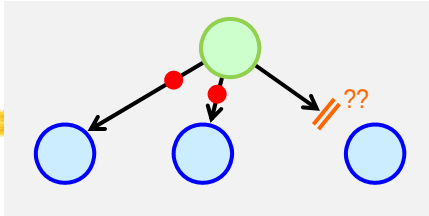
2. Lösung: Token, das auf einem (logischen) Ring kreist



- Token = Senderecht (Token weitergeben!)
- Broadcast selbst z.B. über ein zugrundeliegendes broadcastfähiges Medium

- Token führt eine Sequenznummer (inkrementiert beim Senden); so werden alle Broadcasts global nummeriert
- Empfänger wissen, dass Nachrichten entsprechend der (in den Nachrichten mitgeführten Sequenznummer) ausgeliefert werden müssen
- Bei Lücken in den Nummern: dem Token einen Wiederholungswunsch mitgeben (Sender erhält damit implizit ein NACK bzw. ACK)
- Tokenverlust (z.B. durch Prozessor-Crash) durch Timeouts feststellen (Vorsicht: Gefahr, dass dabei Token unabsichtlich verdoppelt wird!)
- Einen gecrashten Prozess (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

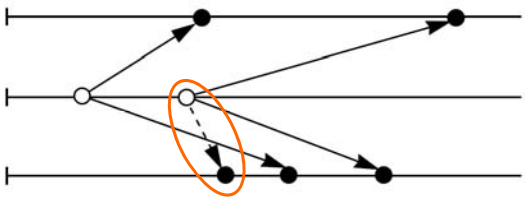
Resümee zu Broadcast



- Übersicht
  - Anwendungen
  - idealisierte Sicht (gleichzeitiger Empfang; kein Verlust von Einzelnachrichten)
  - Fehlerproblematik
  - Vorbeugung gegen Fehler mit ACK, NACK
- Algorithmus für „reliable Broadcast“
  - Redundanz („Fluten“ des Netzes) zur Erhöhung der Fehlertoleranz
  - Effizienz / Kosten (Zahl von Einzelnachrichten)
- FIFO-Broadcasts
  - zwei nacheinander ausgeführte Broadcasts ein und desselben Senders erreichen alle Empfänger in Sendereihenfolge
  - nicht stark genug, um „akausale“ Beobachtungen zu verhindern

FIFO-Ordnung bei Multicast

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



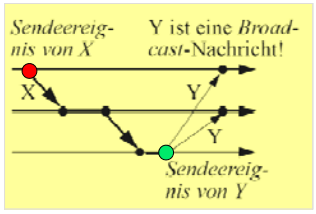
Wdh

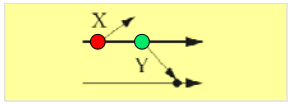
Resümee zu Broadcast (b)

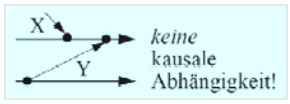
- Kausale Broadcasts
  - kausale Abhängigkeit zweier Nachrichten
  - „causal order“: Nachrichtenempfang „respektiert“ kausale Abhängigkeit von Nachrichten (ist also „kausaltreu“)
  - „globalisiertes“ FIFO
- Atomare Broadcasts
  - logisch gleichzeitiger Empfang der Einzelnachrichten eines Broadcasts
  - Realisierung über zentralen Sequencer bzw. Token auf einem logischen Ring

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 Sendeereignis von X zum Sendeereignis von Y führt.





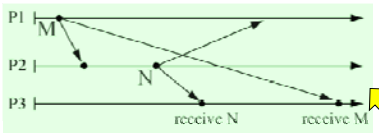


Wdh

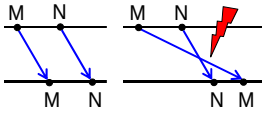
Kausaler Broadcast

Zweck: Wahrung von Kausalität bei der Kommunikation

- *Definition:* Kausale Reihenfolge („causal order“): 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
- „Kausale Reihenfolge“ (bzw. „kausale Abhängigkeit“) lassen sich insbesondere auch auf Broadcasts anwenden
- Kausale Reihenfolge impliziert FIFO-Reihenfolge: kausale Reihenfolge ist eine Art „globales FIFO“



Gegenbeispiel: *Keine* kausalen Broadcasts!



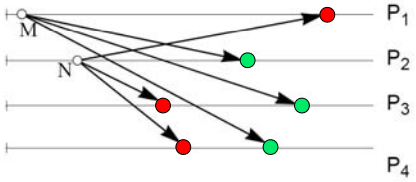
Wdh

Atomarer Broadcast

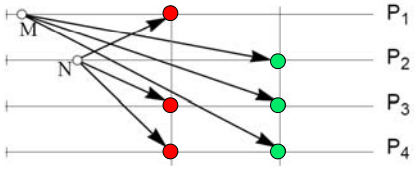
- *Definition:* Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 die Nachricht M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt
- Anschaulich: Nachrichten eines Broadcasts werden „überall quasi gleichzeitig“ empfangen

Wdh

Atomarer Broadcast: Beispiel



Äquivalent bzgl. „Gummiband-Transformation“

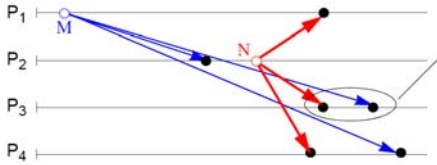


Beachte: das Senden wird nicht als Empfang der Nachricht beim Sender selbst gewertet

Wdh

Wie „gut“ ist atomarer Broadcast?

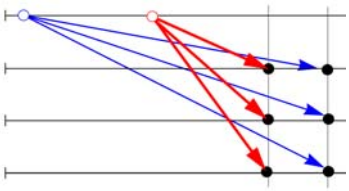
1) Ist atomar auch kausal?



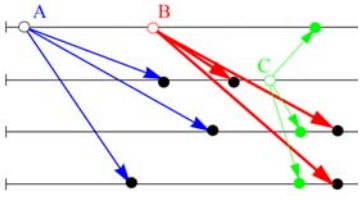
Nicht kausal!

Atomar: P3 und P4 empfangen beide M, N und zwar in gleicher Reihenfolge

2) Ist atomar wenigstens FIFO?



3) Ist atomar + FIFO evtl. kausal?



Fazit: Semantik von Broadcast

- Atomare Übermittlung $\not\Rightarrow$ kausale Reihenfolge
- Atomare Übermittlung $\not\Rightarrow$ FIFO-Reihenfolge
- Atomare Übermittlung + FIFO $\not\Rightarrow$ kausale Reihenfolge
  - Bemerkung zu vorheriger Seite: nicht nur 3), sondern auch 1) ist ein (Gegen)beispiel, da M, N FIFO-Broadcast ist
- Vergleich mit („idealer“) speicherbasierter Kommunikation:
  - Kommunikation über gemeinsamen Speicher ist atomar (alle „sehen“ das Geschriebene gleichzeitig – so sie hinschauen)
  - Kommunikation über gemeinsamen Speicher wahrt Kausalität (Wirkung tritt unmittelbar mit dem Schreibereignis als Ursache ein)

Kausaler atomarer Broadcast

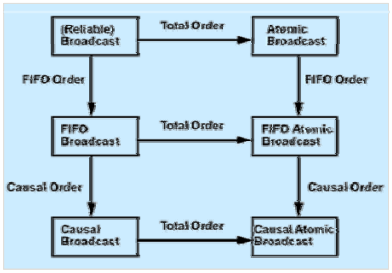
- Der speicherbasierten Kommunikation vergleichbares Kommunikationsmodell per Nachrichten: kausaler atomarer Broadcast
  - = kausaler Broadcast + atomarer Broadcast
- Man nennt daher kausale, atomare Übermittlung auch virtuell synchrone Kommunikation
- Denkübung: realisieren die beiden Implementierungen „zentraler Sequencer“ bzw. „Token auf Ring“ die virtuell syn. 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 Kausalbeziehungen
  - identische totale Ordnung aller Ereignisse
- Konsequenz: 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 einziger Broadcast, der alles kann? „Stärkere Semantik“ hat auch Nachteile:
  - Leistungseinbussen
  - weniger potentielle Parallelität
  - aufwändiger zu implementieren

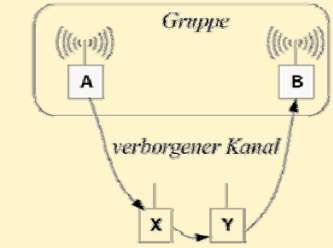


- Bekannte „Strategie“:
  - man begnügt sich daher, falls es der Anwendungskontext gestattet, oft mit einer billigeren, aber weniger perfekten Lösung
  - Motto: so billig wie möglich, so „perfekt“ wie nötig
  - man sollte aber die Schwächen einer Billiglösung kennen!
- ⇒ grössere Vielfalt ⇒ komplexer bzgl. Verständnis und Anwendung

Multicast

- Multicast = Broadcast an eine Teilmenge von Prozessen
  - diese Teilmenge wird „Multicast-Gruppe“ genannt
- Zweck von Multicast-Gruppen
  - „selektiver Broadcast“
  - Vereinfachung der Adressierung (z.B. statt Liste von Einzeladressen)
  - Verbergen der Gruppenzusammensetzung nach aussen
  - „logischer Unicast“: Gruppen ersetzen Individuen (z.B. für transparente Replikation)
- Alles, was zur Broadcastsemantik gesagt wurde, gilt (innerhalb der Gruppe) auch bzgl. Multicastsemantik:
  - zuverlässiger Multicast, FIFO-Multicast, kausaler Multicast, atomarer Multicast, kausaler atomarer Multicast

Problem der „Hidden Channels“ beim Multicast



Kausalitätsbezüge verlassen (z.B. durch Gruppenüberlappung) die Multicastgruppe 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

Dynamische Multicastgruppen

- Bei dynamischen Gruppen können Prozesse jederzeit der Gruppe beitreten oder aus der Gruppe austreten
  - Crash kann als eine besondere Austrittsform modelliert werden
- Und wenn dies während des Ablaufs einer Multicast-Operation geschieht?
  - haben verschiedene Sender an die Gruppe die gleiche Sicht bzgl. der Gruppenzusammensetzung?
- Man wünscht sich (Realisierung besprechen wir hier nicht!):
  - die Gruppe soll bei allen (potentiellen) Sendern an die Gruppe hinsichtlich der (logischen) Ein- und Austrittszeitpunkte jedes Gruppenmitglieds übereinstimmen
  - Eintritt und Austritt sollen atomar erfolgen

Weitere Kommunikationspardigmen

Wir besprechen nachfolgend 3 weitere Aspekte:

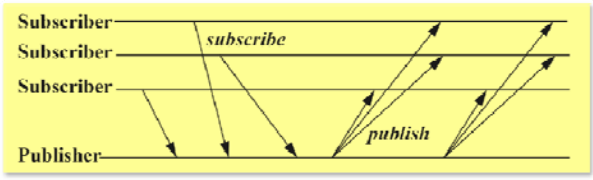
1. Push-Paradigma
2. Publish & Subscribe
3. Tupelräume (und JavaSpaces)

Push-Paradigma

- „Push“ im Unterschied zum klassischen „Pull-“ (bzw. „Request / Reply“)-Paradigma, bei dem
  - Clients die gewünschte Information aktiv anfordern müssen,
  - sie aber oft nicht wissen, ob bzw. wann sich eine Information geändert hat,
  - dadurch periodisches Nachfragen („polling“) beim Server notwendig ist
- Unangefragt (vom Client) schickt der Server etwas
- Push: „event driven“ ↔ pull: „demand driven“

Publish & Subscribe

- Subscriber (= Client) meldet sich für den Empfang des gewünschten Typs von Information („channel“) an
- Subscriber erhält automatisch (aktualisierte) Information vom Publisher (= Server), sobald diese zur Verfügung steht
  - push vom Publisher bzw. „callback“ des Subscribers durch Publisher



- „Publish“ entspricht Multicast
  - „subscribe“ entspricht so gesehen dem Beitritt einer Multicast-Gruppe

Tupelräume

- Gemeinsam genutzter („virtuell globaler“) Speicher
- Blackboard- oder Marktplatz-Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkoppelung der Teilnehmer
- Tupel = geordnete Menge typisierter Datenwerte
- Entworfen 1985 von D. Gelernter (für die Sprache Linda)
- Operationen:
  - out (t): Einfügen eines Tupels t in den Tupelraum
  - in (t): Lesen und Löschen von t aus dem Tupelraum
  - read (t): Lesen von t im Tupelraum

Tupelräume (2)

- Inhaltsadressiert („Assoziativspeicher“)
  - Vorgabe eines Zugriffsmusters (bzw. „Suchmaske“) beim Lesen, damit Ermittlung der restlichen Datenwerte eines Tupels („wild cards“)
  - Beispiel: int i,j; in(„Buchung“, ?i, ?j) liefert ein „passendes“ Tupel
  - analog zu einigen relationalen Datenbankabfragesprachen
- Synchrone und asynchrone Leseoperationen
  - ‘in’ und ‘read’ blockieren, bis ein passendes Tupel vorhanden ist
  - ‘inp’ und ‘readp’ blockieren nicht, sondern liefern als Prädikat („passendes Tupel vorhanden?“) ‘wahr’ oder ‘falsch’ zurück

Tupelräume (3)

- Mit Tupelräumen sind auch die üblichen Kommunikationsmuster realisierbar, z.B. Client-Server:

```
/* Client */
...
out("Anfrage" client_Id, Parameterliste);
in("Antwort", client_Id, ?Ergebnisliste);
...
/* Server*/
...
while (true)
{  in("Anfrage", ?client_Id, ?Parameterliste);
   ...
   out("Antwort", client_Id, Ergebnisliste);
}
```

Zuordnung des „richtigen" Clients über die client_Id

Tupelräume (4)

- Kanonische Erweiterungen des Modells hinsichtlich
  - Persistenz (Tupel bleiben nach Programmende erhalten)
  - Transaktionseigenschaft (wichtig, wenn mehrere Prozesse parallel auf den Tupelraum bzw. gleiche Tupel zugreifen)
- Problem: effiziente, skalierbare Implementierung?
  1) zentrale Lösung: Engpass
  2) replizierter Tupelraum (jeder Teilnehmer hat vollständige Kopie des Tupelraums; schnelle Zugriffe, jedoch hoher Synchronisationsaufwand)
  3) aufgeteilter Tupelraum (jeder Teilnehmer hat einen Teil des Tupelraums; 'out'- Operationen können z.B. lokal ausgeführt werden, 'in' evtl. mit Broadcast)
- Kritik: globaler Speicher ist der strukturierten Programmierung und der Verifikation abträglich
  - unüberschaubare potentielle Seiteneffekte

JavaSpaces

- „Tupelraum“ für Java
  - gespeichert werden Objekte → neben Daten auch „Verhalten“
  - Tupel entspricht Gruppen von Objekten
- Operationen
  - write: mehrfache Anwendung erzeugt verschiedene Kopien
  - read
  - readifexists: blockiert (im Gegensatz zu read) nicht; liefert u.U. ‘null’
  - takeifexists
  - notify: Benachrichtigung (mittels eines Ereignisses), wenn ein passendes Objekt in den JavaSpace geschrieben wird

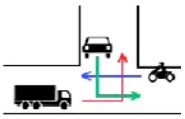
JavaSpaces (2)

- Teil von Jini (Infrastrukturplattform und Middleware für Java)
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode vom Sender zum Empfänger
  - gemeinsame Nutzung von Objekten
  - persistente Speicherung von Objekten (aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt)
- Semantik: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn ein write vor einem read beendet wird, muss read nicht notwendigerweise das lesen, was write geschrieben hat

Wechselseitiger Ausschluss

Wechselseitiger Ausschluss
(Mutual Exclusion, „Mutex“)

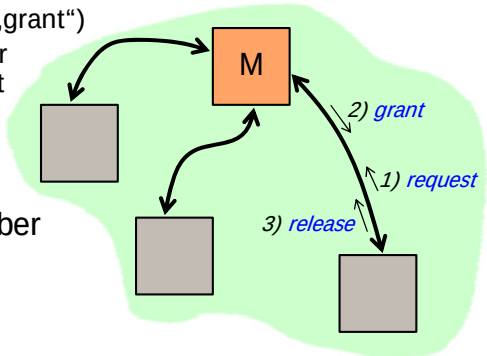
- Koordination, wenn viele wollen, aber nur einer darf
- „Streit“ um exklusives Betriebsmittel, z.B.:
  - konkrete Ressource (z.B. gemeinsamer Datenbus)
  - abstrakte Ressource (z.B. ein „Termin“ in einem verteilten Terminkalendersystem)
  - „kritischer Abschnitt“ in einem nebenläufigen Programm



- Es gibt klassische Lösungen bei shared memory
  - z.B. Semaphore und Monitore (→ Betriebssystemtheorie)
  - sind in unserem Kontext aber nicht interessant

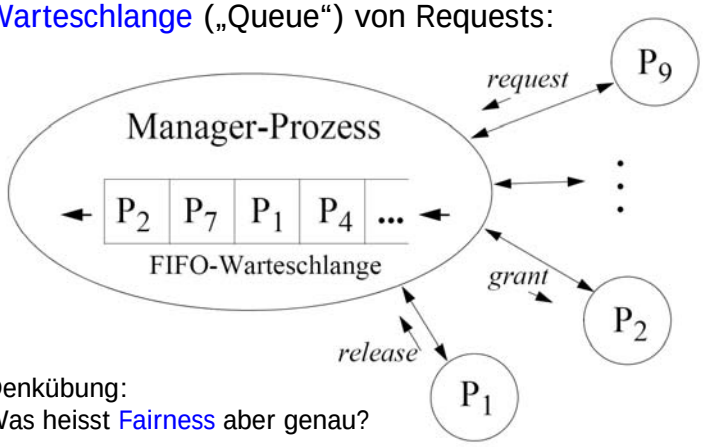
Zentraler Manager?

- Hier: Nachrichtenbasiertes System konkurrierender Prozesse
- Idee: Manager, der die Ressource (in fairer Weise!) zuordnet
  - ein Prozess bewirbt sich um die Ressource mit „request“
  - wartet dann auf Erlaubnis („grant“)
  - teilt schliesslich Freigabe der Ressource dem Manager mit „release“ mit
- Vergleichsweise einfach und wenige Nachrichten, aber
  - potentieller Engpass
  - single point of failure



Globale Warteschlange garantiert Fairness

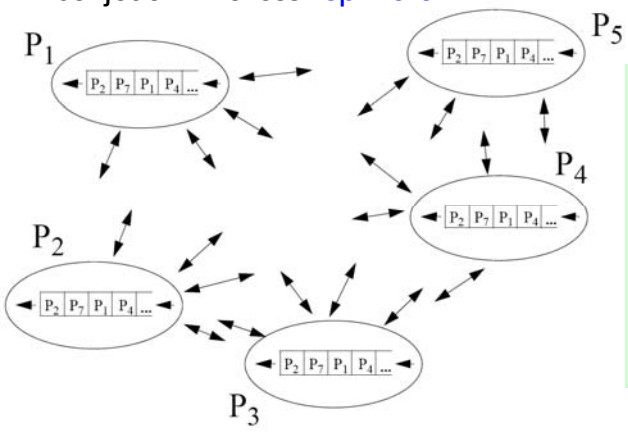
- Der Manager-Prozess hält eine (zeitlich geordnete) Warteschlange („Queue“) von Requests:



- Denkübung:
  Was heisst Fairness aber genau?

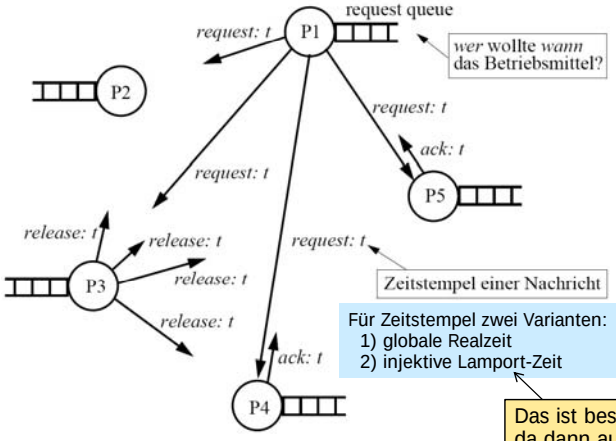
Replizierte Warteschlange?

- Idee für eine dezentrale Lösung: globale Warteschlange bei jedem Prozess replizieren



- Alle Prozesse sollen die gleiche Sicht der „virtuell globalen“ Warte-schlange haben
- Konsistenz wird mit (vielen) Nachrichten und logischer Zeit erreicht (→ nächste slides)

Synchronisation der Warte-schlangen mit Zeitstempeln



Für Zeitstempel zwei Varianten:
1) globale Realzeit
2) injektive Lamport-Zeit

Das ist besonders interessant, da dann auf synchronisierte Uhren verzichtet werden kann

- Voraussetzung: FIFO-Kommunikation
- Alle Nachrichten tragen (eindeutige!) Zeitstempel
- Request- und Release-Nachrichten immer an alle senden (FIFO-Broadcast)
- Requests werden bestätigt („ack“)

Der Algorithmus (Lamport 1978)

1) Bewerbung um Betriebsmittel: Request mit Zeitstempel und Absender an alle senden und in eigene Queue einfügen
2) Bei Empfang eines Request: Request in eigene Queue einfügen, ack versenden
3) Bei Freigabe des Betriebsmittels: Aus eigener Queue entfernen und Release an alle versenden
4) Bei Empfang eines Release: Zugehörigen Request aus eigener Queue entfernen
5) Ein Prozess darf das Betriebsmittel nutzen, wenn:
   a) der eigene Request der früheste in seiner Queue ist
   b) und er bereits von jedem anderen Prozess (irgendeine) spätere Nachricht bekommen hat

(Frühester Request ist global eindeutig ⇒ die beiden Bedingungen garantieren, dass kein früherer Request mehr kommt (wieso?))

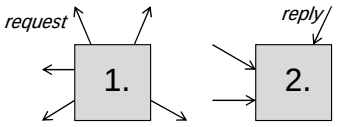
Denkübungen:
- Wieso ist FIFO notwendig?
- Wo geht (bei Lamport-Zeit) die Uhrenbedingung ein?

Wieso sind garantiert:
1) Safety (zu jedem Zeitpunkt höchstens einer),
2) Fairness (jeder Request wird „schliesslich“ erfüllt)?

3(n-1) Nachrichten pro Bewerbung (n = Zahl der Prozesse)

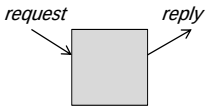
Ein anderer verteilter Mutex-Algorithmus (Ricart / Agrawala, 1981)

- Nur 2(n-1) Nachrichten (Reply übernimmt Rolle von Release und ack)



1. Sende Request (mit log. Zeitstempel!) an alle n-1 anderen
2. Dann auf n-1 Replies warten, danach Betriebsmittel nutzen

- Bei Eintreffen einer Request-Nachricht:
  - wenn nicht selbst beworben oder der Sender „ältere Rechte“ (bzgl. logischer Zeit!) hat, dann Reply sofort schicken
  - ansonsten Reply erst später (im Sinne von Release) schicken, nach Erfüllen des eigenen Requests (d.h. exklusivem Zugriff)



Nur älteste Bewerbung setzt sich überall durch!

Denkübungen: Safety? Fairness? Deadlockfreiheit? FIFO-Kanäle notwendig?

Resümee

- Kausal atomare Broadcasts
  - virtuelle Synchronität
- Multicast
  - dynamische Gruppen, „hidden channels“
- Push-Prinzip und Publish & Subscribe
- Tupelräume
  - Linda-Modell, JavaSpaces
- Wechselseitiger Ausschluss (mit logischer Zeit)
  - replizierte Warteschlangen von Lamport (request, reply, ack)
  - anderes Verfahren: verzögerte Replies (mit 2(n-1) Nachrichten)
  - Korrektheitsargumente? (Safety, Fairness,...)