Verteilte Systeme

Übereinstimmung und Koordination

Wahlen

Wahl-Algorithmen

- In vielen verteilten Algorithmen benötigt man einen Prozeß, der eine irgendwie geartete besondere Rolle spielt, z.B. als **Koordinator**, **Initiator** oder **Monitor**.
- Die **Aufgabe** eines **Wahl-Algorithmus** (Election-Algorithmen) ist es, einen Prozeß unter vielen gleichartigen Prozessen durch eine kooperative, verteilte Wahl eindeutig zu bestimmen, der diese Rolle übernimmt.
- **Anwendungsbeispiele**:
  - wechselseitiger Ausschluß
  - Ausfall eines Koordinators
- Die Wahl (Election) wird dabei meist durch die Bestimmung eines Extremwertes auf einer Ordnung der beteiligten Prozesse **durchgeführt**:
  - Jeder Prozeß hat eine Nummer, die allen anderen Prozessen bekannt ist
  - Kein Prozeß weiß, welcher andere Prozeß gerade funktioniert
  - Alle Algorithmen *wählen* den Prozeß mit der höchsten Nummer aus.
- Nach **Ausfall** gliedert sich ein Prozeß beim Neustart wieder in die Menge der aktiven Prozesse ein

Bully-Algorithmus

- H. Garcia-Molina, 1982
- **Ziel**: Bei Ausfall des bisherigen Koordinators, denjenigen Prozeß zu finden, der noch aktiv ist und den höchsten Identifikator trägt. Dieser wird dann als neuer Koordinator (Bully = Tyrann) eingesetzt und allen bekannt gemacht.
- Der Auswahlalgorithmus wird durch einen beliebigen Prozeß P **ausgelöst** (evtl. sogar nebenläufig!), z.B. durch denjenigen, der den Ausfall des bisherigen Koordinators bemerkt hat.
- Es werden **drei Sorten Nachrichten** verschickt:
  - election: Auslösen der Wahl
  - answer: Bestätigung des Erhalts einer e-Nachricht
  - coordinator: Mitteilung, daß der Sender der neue Koordinator ist

Bully-Algorithmus: Ablauf

- P bemerkt, daß der Koordinator nicht mehr antwortet:

HOLD_ELECTION

```
    ∀Q: (Id(Q) > Id(P))
          send(Q, ELECTION);
    if ∃Q: (receive(Q, ANSWER)) /* timeout T for ∃ */
          then do_something_else;
          else ∀Q: (Id(Q) < Id(P))
                     send(Q, COORDINATOR);
    endif;
```

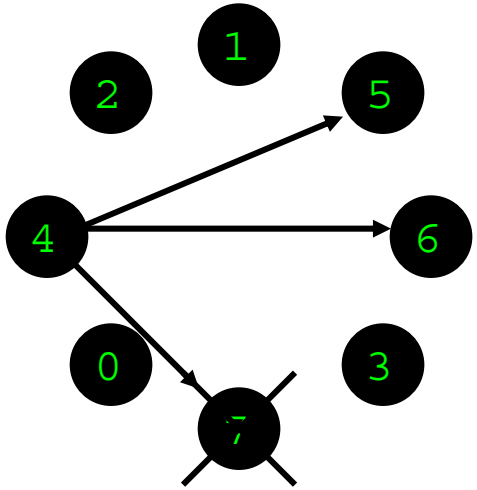
- P empfängt `ELECTION`, gesendet von Prozeß `pred`

CONTINUE_ELECTION

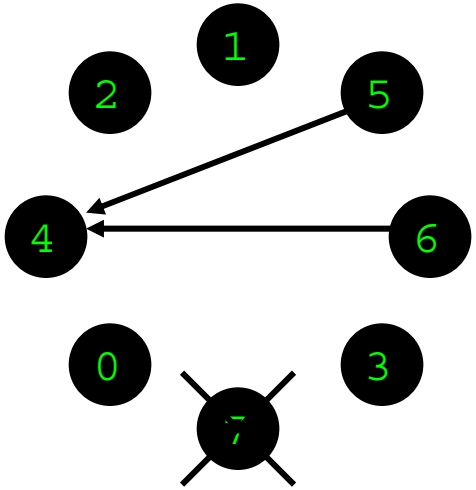
```
    send(pred, ANSWER);
    HOLD_ELECTION;
```

Bully-Algorithmus: Beispiel

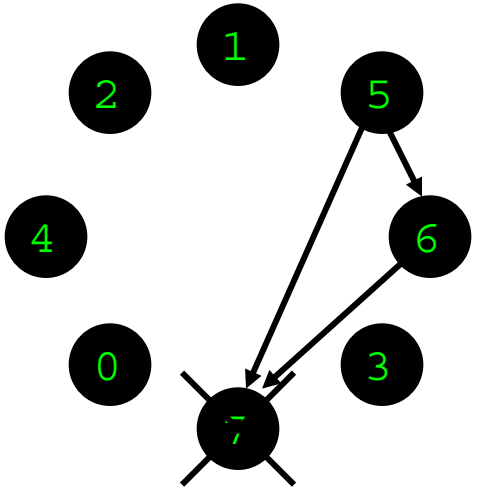
send(Q,ELECTION)



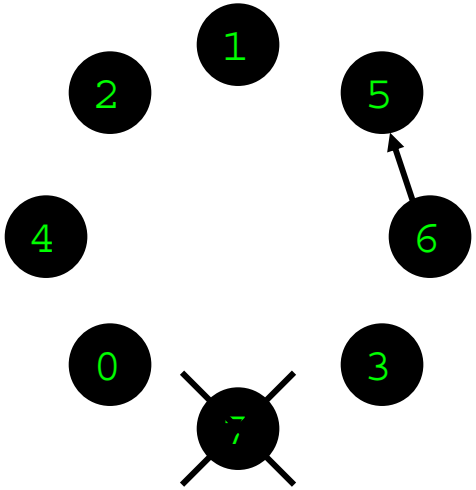
receive(Q,ANSWER)



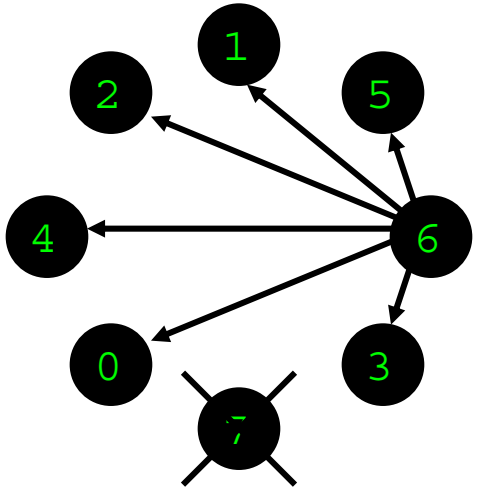
send(Q,ELECTION)



receive(Q,ANSWER)



send(Q,COORDINATOR)



Bewertung:

- hohe Nachrichtenkomplexität
  - worst case $O(n^2)$ Nachrichten.
  - best case (n - 2) Nachrichten
- alle Prozesse mit ihren Identifikatoren müssen allen bekannt sein

Ring-Algorithmus

- E.G. Chang und R.Roberts, 1979
- **Annahme**:
  - alle Prozesse haben eine eigene Identifikation
  - Prozesse sind in einem gerichteten logischen Ring angeordnet
  - jeder Prozeß kennt seinen direkten Nachbarn, aber auch die weiteren Knoten des Rings, so daß er bei Nichterreichbarkeit seines Nachfolgers dessen Nachfolger adressieren kann.
  - alle Prozesse sind aktiv.
  - Es treten keine Fehler auf.
- Es werden **zwei Sorten Nachrichten** verschickt:
  - election: Auslösen der Wahl
  - coordinator: Mitteilung, wer der neue Koordinator ist

Ring-Algorithmus: Ablauf

- Starten der Wahl

```
HOLD_ELECTION
 send(next_neighbour, ELECTION, [my_id]);
```

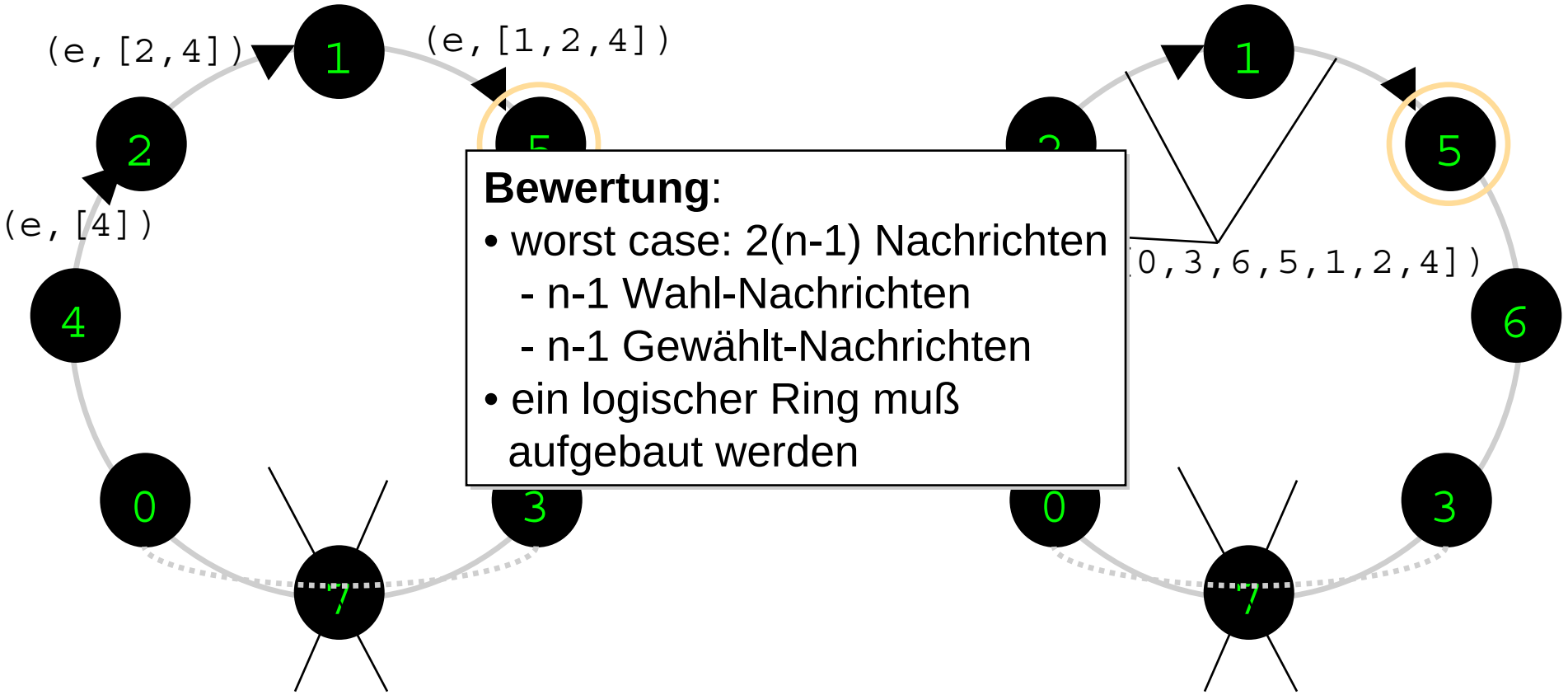
- Empfang einer Wahl-Nachricht

```
CONTINUE_ELECTION
 receive(previous_neighbour, ELECTION, list);
 if member(my_id,list)
    then Coordinator := process[maximum(list)];
         send(next_neighbour, COORDINATOR, list);
    else send (next_neighbour, ELECTION, [my_id|list]);
 endif;
```

- Empfang einer Gewählt-Nachricht

```
FINISH_ELECTION
 receive(previous_neighbour, COORDINATOR, list);
 Coordinator := process[maximum(list)];
```

Ring-Algorithmus: Beispiel



Prozeß 4 hat eine Wahl ausgelöst
Aktueller Prozeß: Nr. 5

Die Wahl ist gelaufen. Prozeß 4 sendet die Gewählt-Nachricht

Wahl auf Bäumen: FireWire

- **Hochgeschwindigkeitsbus** FireWire: IEEE 1394 High Performance Serial Bus
- „**Hot-pluggable**“: Geräte können zu beliebigen Zeitpunkten hinzugefügt oder entfernt werden
- Jedes Gerät kann einen oder mehrere Ports haben (Empfehlung: 3 Ports pro Gerät)
- Hot-Plug-Operationen führen zu einem **Bus-Reset**: Der Bus-Master (Baumwurzel) wird über Wahlalgorithmus bestimmt
- **Wahlbedingungen**
  1. Verbundenes, azyklisches Netzwerk (Baumstruktur!) mit bidirektionalen Kanälen
  2. Jedes Gerät kennt nur die direkt verbundenen Geräte
  3. Dezentrale, nicht-deterministische Wahl

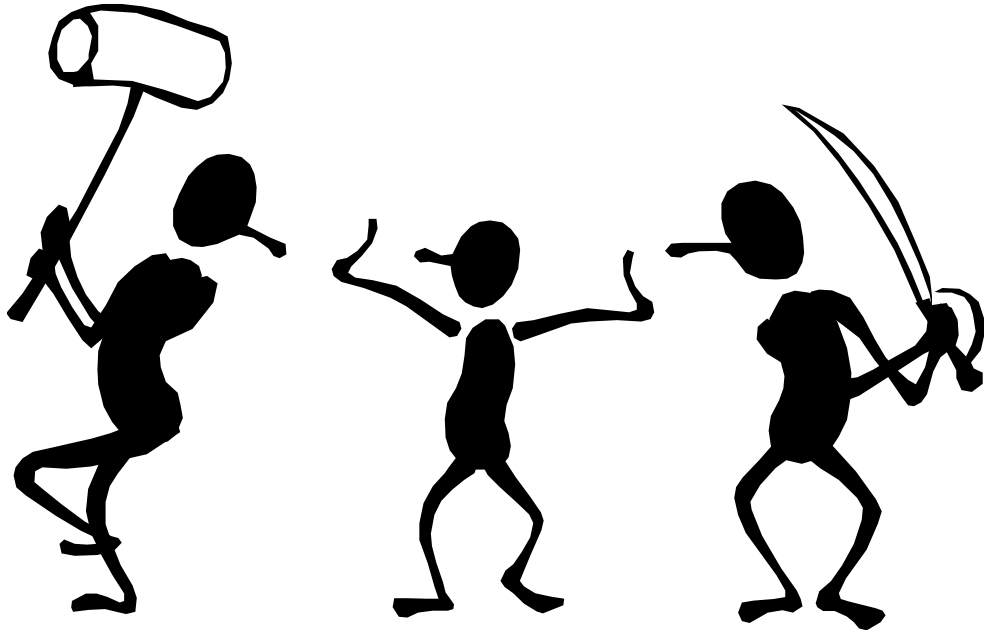
Wahl auf Bäumen: Ablauf

- **Hold** Election: (Wahlauslösungs Welle): Bus-Reset
- **Continue** Election 1: Blattknoten (nur **1 Port** „belegt“) senden „*Be_my_parent*“-Nachrichten an ihre (direkten) Nachbarn
- **Continue** Election 2: Sobald ein innerer Knoten (**k>1 Ports** belegt) über **k-1 Kannäle** „*Be_my_parent*“-Nachrichten empfangen hat, bestätigt er diese durch „*You_are_my_child*“-Nachrichten und sendet über den verbliebenen Kanal eine „*Be_my_parent*“-Nachricht
- **Finish** Election: Sobald ein innerer Knoten (**k>1 Ports** belegt) über **k Kanäle** „*Be_my_parent*“-Nachrichten empfangen hat und selbst noch keine solche Nachricht ausgesendet hat, bestätigt er diese durch „*You_are_my_child*“-Nachrichten und wird damit automatisch Bus-Master

Wahl auf Bäumen: Konfliktsituation

- Auf **einem Kanal** wurde in **beiden Richtungen** „*Be_my_parent*“-Nachrichten versendet. (Es kann nur maximal einen Kanal geben, auf dem das passieren kann!)
- Beide beteiligten Knoten **warten zufällig** eine kurze Zeit (240…260ns) oder eine lange Zeit (570…600ns)
- Nach dieser Zeit wird **erneut** eine „*Be_my_parent*“-Nachricht gesendet, falls noch keine solche Nachricht von dem anderen Knoten empfangen wurde
- Dies wird **solange wiederholt**, bis der Konflikt gelöst wurde
- **Alternativ**: sofern eine ID vorhanden ist, gewinnt standardmäßig der Knoten mit der größeren ID

Wechselseitiger Ausschluß



Wechselseitiger Ausschluß

- Beim **Zugriff auf gemeinsame Daten** oder der Nutzung von Betriebsmitteln muß oft die **Exklusivität** des Zugriffs oder Nutzung sichergestellt werden.
- Das Programmstück, das den Zugriff realisiert, heißt **kritischer Abschnitt** oder auch **kritische Region** (critical section)
- In **Einprozessor-Systemen**:
  - Semaphore,
  - Monitore, ...
- Im **verteilten System** nicht möglich (z.B. spontane Netzwerke):
  - kein gemeinsamer Speicher,
  - keine gemeinsamen Variablen
- **Ziel**: Verfahren, das verteilt kritische Regionen verwalten kann.

Wechselseitiger Ausschluß: Anforderungen

1. **Sicherheit** (engl.: safety): Höchstens ein Prozeß darf sich in der kritischen Region befinden.
2. **Lebendigkeit** (engl.: liveness): Ein Prozeß, der den Eintritt in die kritische Region beantragt, darf auf jeden Fall irgendwann eintreten. Dies impliziert Verklemmungs-Freiheit (engl.: deadlock free) und kein Verhungern (engl.: starvation) der Prozesse.

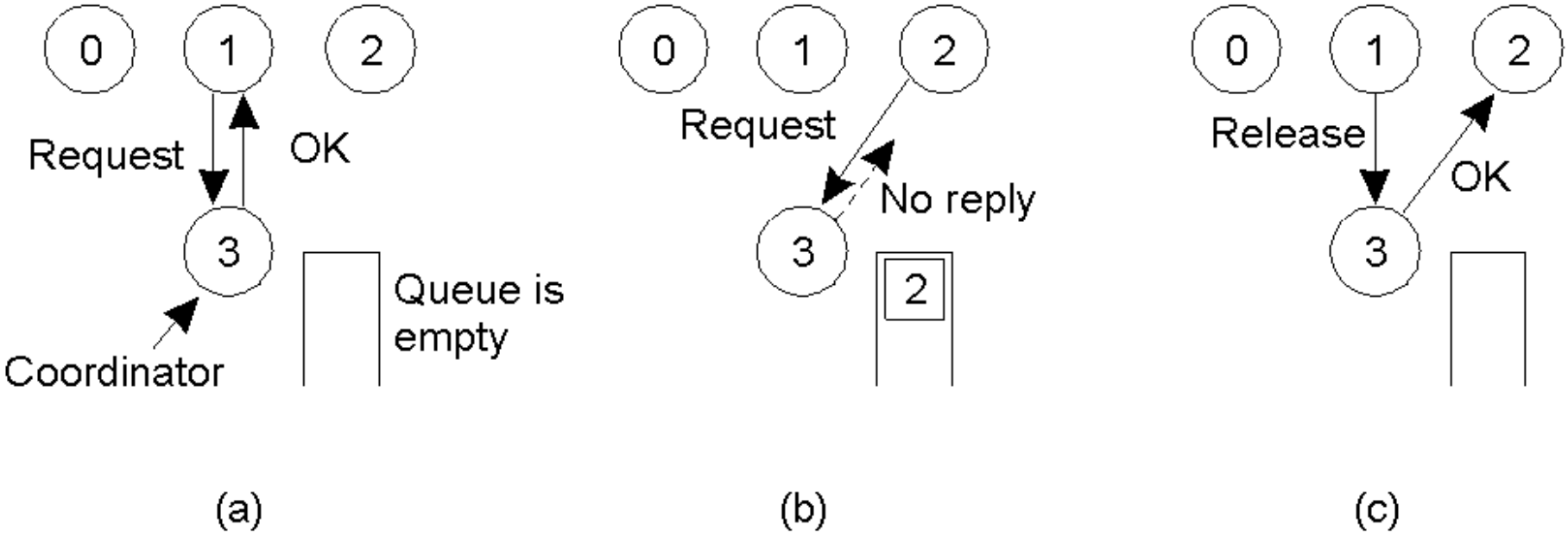
Zusätzlich oft folgende Forderungen:

3. **Ordnung** (engl.: ordering) Der Eintritt in die kritische Region erfolgt nach der Happened-Before Relation, d.h. falls ein Prozeß, der auf den Eintritt wartet, eine Nachricht an einen anderen Prozeß schickt, der daraufhin ebenfalls in den kritischen Abschnitt will, dann darf der Sendeprozeß vor dem Empfangsprozeß in den kritischen Abschnitt.
4. **Fehlertoleranz:** Forderungen 1 und 2 werden auch bei Ausfällen erfüllt.

Algorithmus 1: Zentrale Lösung

- Rückführung auf lokalen Mechanismus: Einer der Prozesse wird zum **Koordinator** für eine kritische Region bestimmt, d.h. er verwaltet den Zugriff auf die Ressourcen und damit den wechselseitigen Ausschluß.
- Alle anderen müssen sich nun zuerst **an den Koordinator wenden**, bevor sie die entsprechende Region betreten.
- Wenn die **kritische Region frei** ist, erhält der Prozess das **OK** vom Server. Nach Abarbeitung der Aufgaben gibt der Prozess dieses Token zurück.
- Ist die **Region nicht frei**, wird der anfragende Prozess in eine **Warteschlange** aufgenommen. Er erhält erst das Token, wenn alle Prozesse vor ihm bedient wurden.

Zentrale Lösung: Beispiel



Zentrale Lösung: Bewertung

- **Wechselseitiger Ausschluß** wird **erreicht**: es ist immer nur ein Prozess im kritischen Bereich, da der Server immer nur ein Token vergibt
- Die Basisbedingungen **Sicherheit** und **Lebendigkeit** sind **erfüllt**
- Fairness in Bezug auf die Zugriffsreihenfolge: Tokens werden in der Reihenfolge der Anfrage vergeben
- **Ordnung** nach Happened-before Relation ist **nicht gewährleistet**
- **Fehlertoleranz** z.B. bei Ausfall von Client oder Server **ist nicht gewährleistet**
- **Vorteile**:
  - einfach realisierbar
  - Nachrichtenkomplexität: maximal drei Nachrichten für den Eintritt
- **Nachteile**:
  - **single-point-of-failure**, d.h. Client kann zwischen langer Warteschlange und abgestürztem Server nicht unterscheiden.
  - **Ausfall eines Clients** innerhalb der kritischen Region oder
  - der **Verlust von Nachrichten**

  kann zu einem **Deadlock** führen

Algorithmus 2: Verteilte Lösung

- nach G. Ricart und A.K. Agrawala, 1981
- Abstimmungsverfahren als verteilter, symmetrischer Algorithmus basierend auf **Lamport‘s logischer Zeit**, d.h. jeder Prozess besitzt eine logische Uhr.
- **Prinzip**: alle beteiligen Prozesse entscheiden gemeinsam über den Eintritt eines einzelnen Prozesses in den kritischen Abschnitt
- Alle Prozesse verständigen sich über **Multicast-Nachrichten**.
- **Annahme**: jeder Prozeß kennt drei Zustände
  - Released: Der Prozeß befindet sich nicht im kritischen Abschnitt.
  - Wanted: Der Prozeß verlangt den Eintritt in den kritischen Abschnitt.
  - Held: Der Prozeß befindet sich im kritischen Abschnitt.
- **Nachrichten**: zwei Arten von Nachrichten:
  - request, beinhaltet eignen (logischen) Zeitstempel und ID
  - reply, gibt kritische Region für Anforderer frei.

Verteilte Lösung: Korrektheitsbedingung

- Alle Orte verhalten sich identisch. Deshalb genügt Betrachtung eines Ortes x
- Nach Ende des Abstimmungsverfahrens sind drei **Gruppen von Anforderungen** unterscheidbar
  1. Solche, die am Ort x bekannt sind und deren Zeitmarke kleiner ist als Cx.
  2. Solche, die am Ort x bekannt sind und deren Zeitmarke größer ist als Cx.
  3. Solche, die am Ort x noch unbekannt sind.
- Im Verlaufe der Abstimmung können sich die **Marken ändern**. Daher die folgenden **Korrektheitsbedingungen** :
  - **Bedingung 1**: Die Anforderungen der Gruppe 1 müssen entweder erledigt werden oder eine Marke größer Cx annehmen.
  - **Bedingung 2**: Die Anforderungen der Gruppe 2 dürfen im Zuge der Abstimmung keine Marken kleiner Cx annehmen.
  - **Bedingung 3**: Die Anforderungen der Gruppe 3 müssen Marken größer Cx erhalten.

Verteilte Lösung: Algorithmus

- Initialisierung:

```
state := RELEASED;
```

- Eintrittswunsch von P_i

```
state := WANTED;
send_to_all(request,time_i,P_i);   /*Multicast*/
time := time_i;                    /*P_i's timestamp*/
wait_until(number_of_replies = (n-1));
state := HELD;
```

- Erhalt einer Nachricht $<time_i, P_i>$ bei P_j ($i \neq j$)

```
if (state = HELD or
    (state = WANTED and (time,P_j)<(time_i,P_i)))
      then   queue_request;
      else   send_reply(P_i);
      endif;
```

- Austritt aus kritischem Abschnitt

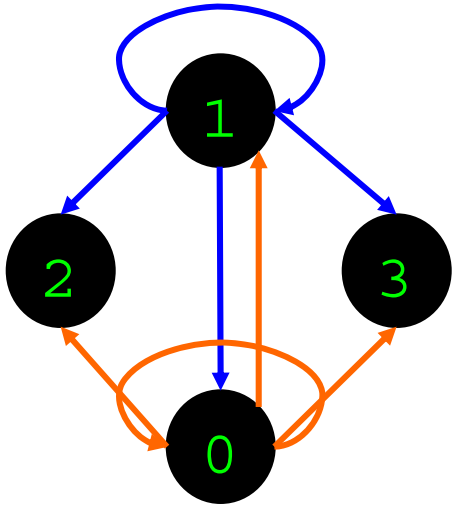
```
state := RELEASED;
send_reply (all queued P_i);
delete_queue;
```

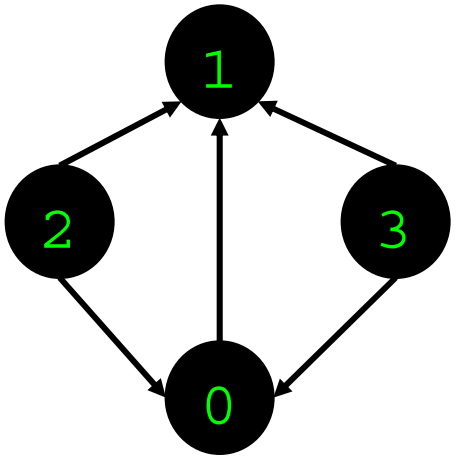
Verteilte Lösung: Beispiel

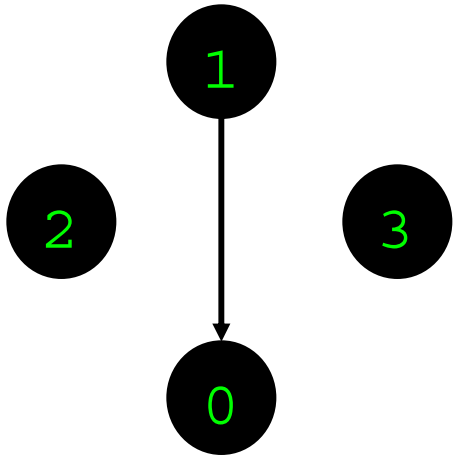
```
send_to_all(request,8,P1);
send_to_all(request,12,P0);
```

```
send_reply(P1);
```

```
send_reply(P0);
```







- Zwei Prozesse (P_0, P_1) wollen *gleichzeitig* die kritische Region betreten.
- P_1 hat den niedrigeren Zeitstempel 8 und gewinnt die verteilte Wahl. P_0‘s Anfrage wird in der Warteschlange von P_1.gespeichert.
- Wenn P_1 fertig ist, gibt er die kritische Region frei und sendet ein reply an P_0.

Verteilte Lösung: Bewertung

- Alle drei **Basisbedingungen** werden **erfüllt**. Aber:

1. Der single-point-of-failure wurde ersetzt durch **n points-of-failure**. Wenn ein Prozess (mit Warteschlange) nicht mehr arbeitet, funktioniert das System nicht mehr.

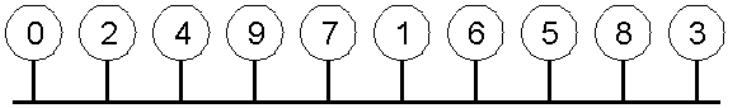
⇒ **Verbesserung**: Dieses Problem könnte durch explizite Verwendung eines Request-Reply-Protokolls ersetzt werden (jede Nachricht wird sofort bestätigt). Wenn keine Bestätigung kommt, ist der Prozess nicht mehr aktiv.

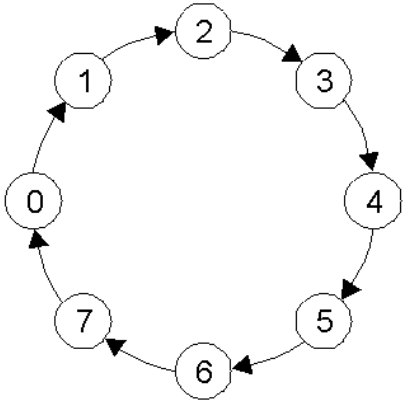
2. **Jeder Prozess muss** immer bei der Entscheidung **mitwirken**, obwohl er evtl. gar kein Interesse an der kritischen Region hat.

⇒ **Verbesserung**: eine einfache Mehrheit genügt.

- Der Algorithmus ist insgesamt **langsamer, komplizierter, teurer und weniger robust**, aber, wie A.S. Tanenbaum sagt: „*Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way*.” Manchmal ist es z.B. gut, einen Algorithmus zu betrachten, nur um das Problem besser zu verstehen.

Algorithmus 3: Token-Ring





Idee:

- die am wechselseitigen Ausschluß beteiligten Prozesse werden in einem **logischen Ring** entsprechend der Prozeßnummern angeordnet.
- es gibt **genau eine Zugriffsberechtigung (Token)**, die gerichtet weitergereicht wird.
- wer das Token besitzt, **darf in die kritische Region**, allerdings nur einmal.
- tritt ein Prozeß in eine kritische Region ein, so behält er das Token und **gibt** es erst **beim Verlassen** wieder **weiter**.

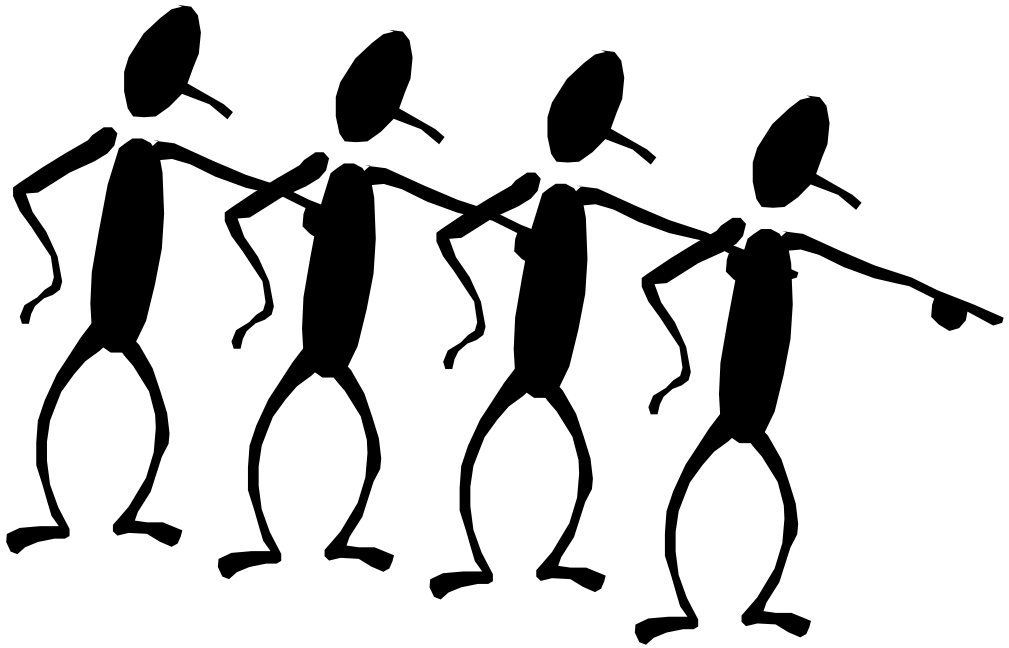
Token-Ring: Bewertung

- Sicherheit und Lebendigkeit **sind erfüllt:**
  - **Sicherheit** ist leicht zu sehen: Nur ein Prozess hat das Token zur selben Zeit.
  - **Kein** Prozess wird **ausgehungert**, da die Reihenfolge durch den Ring bestimmt ist.
- Die **Happened-before** Relation ist nicht erfüllt: **Maximal** muss ein Prozess **warten**, bis alle anderen Prozesse einmal im kritischen Bereich waren.
- **Verlorene Token** erfordern Neugenerierung durch Koordinator.
- Verlust eines Tokens ist **schwer zu erkennen**, da es sich auch um einen sehr langen Aufenthalt in einem kritischen Bereich handeln kann.

Wechselseitiger Ausschluß: Fehlertoleranz

- **Wichtigsten Aspekte** hier sind:
  - Was passiert, wenn Nachrichten verloren gehen ?
  - Was passiert wenn ein Prozeß abstürzt ?
- **Nachrichtenverlust**: Keiner der hier beschriebenen Algorithmen toleriert einen Nachrichtenverlust.
- **Prozeßabsturz**:
  - *Zentrale Lösung*: Absturz eines Clients, der das Token weder besitzt noch angefordert hat, kann toleriert werden.
  - *Verteilte Lösung*: Kann Absturz tolerieren, wenn Request-Reply-Protokoll verwendet wird.
  - *Token-Ring*: kein Prozeß darf abstürzen

Zuverlässige Gruppenkommunikation



Koordination durch Gruppenkommunikation

- Viele der bisher diskutierten Probleme können einfacher und **skalierbarer** mit Hilfe von Gruppenkommunikation gelöst werden – **wie?**
- Zusätzlich erlaubt Multicast die Koordination einander **unbekannter Teilnehmer**
- **Problem**:
  - Standard Multicast ist ein unzuverlässiger Dienst
  - Wie können wir dennoch zuverlässig koordinieren?
- **Lösungsidee**:
  - Ergänze Multicast um eine Sicherungsschicht

Zuverlässiger Multicast

Können wir sicherstellen, dass eine Gruppennachricht alle Empfänger erreicht?

Aspekte:

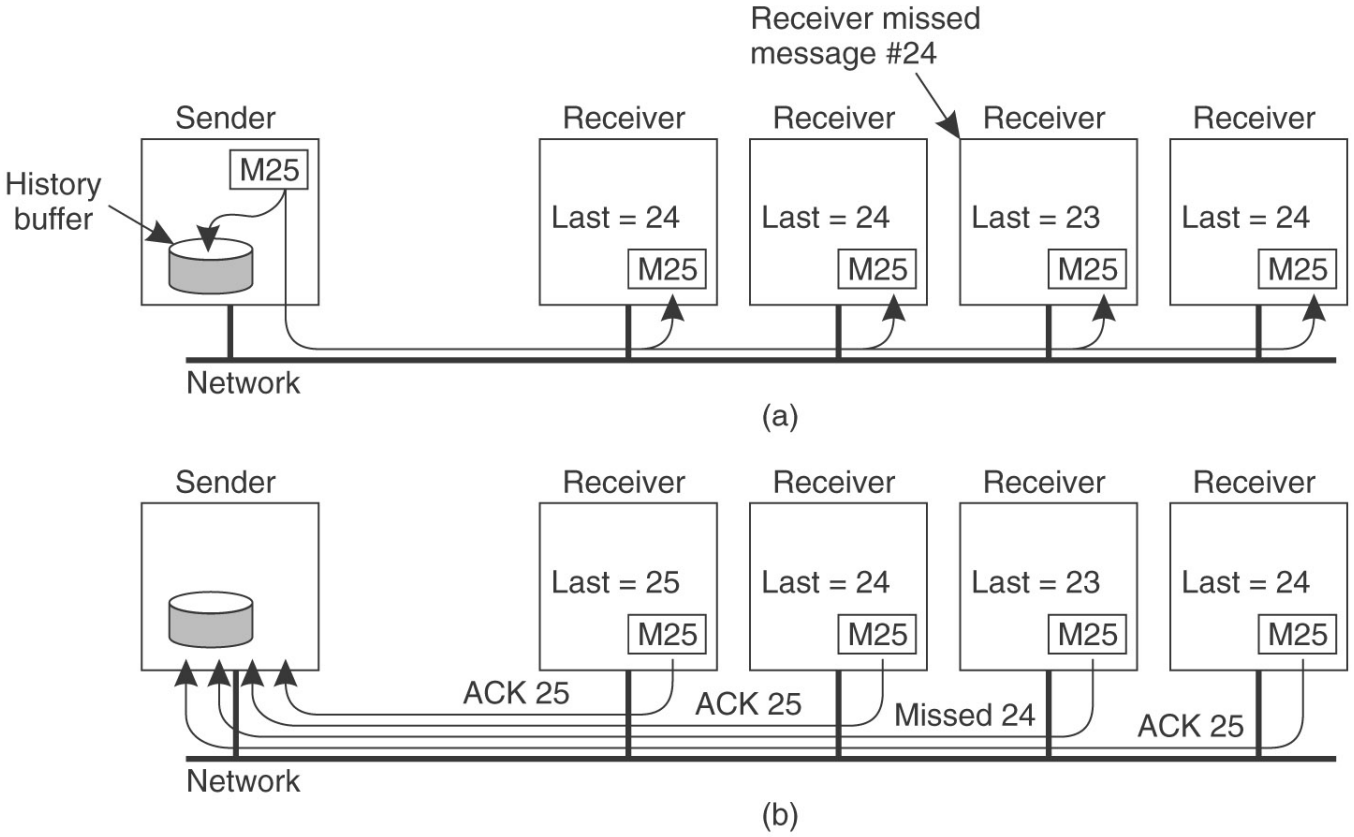
- Entdecken von Nachrichtenverlusten?
- Behandlung von Teilnehmerabstürzen?
- Reaktion auf JOINs und LEAVEs?

Problem:

Wie werden Teilnehmer bekannt?

Wie wird die (aktuelle) Gruppenzusammensetzung bestimmt?

Unmittelbare Rückmeldungen



- Alle Teilnehmer müssen bekannt sein
- Gruppengröße beschränkt

Negative Ackknowledgement

Empfänger sendet NACK nur bei fehlenden Nachrichten

- Empfänger müssen nicht explizit beim Sender bekannt sein
- Verbesserte Skalierbarkeit

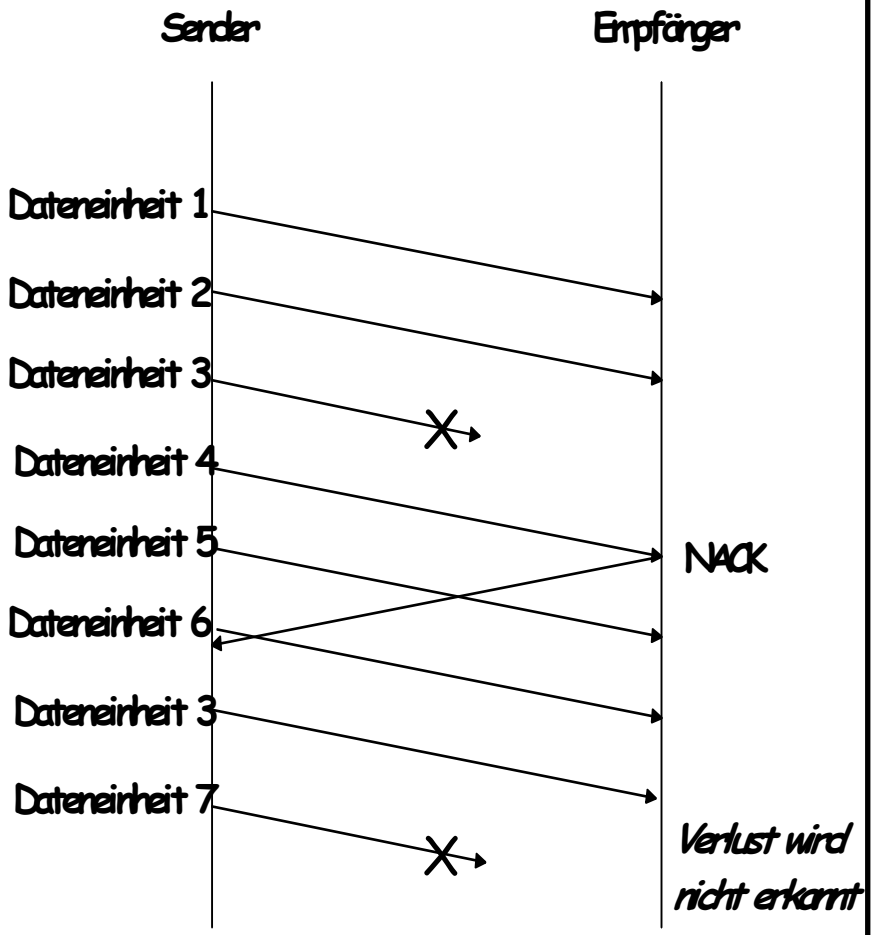
Problem:

Nachrichtenverlust am Sendeende unerkannt

Abhilfe:

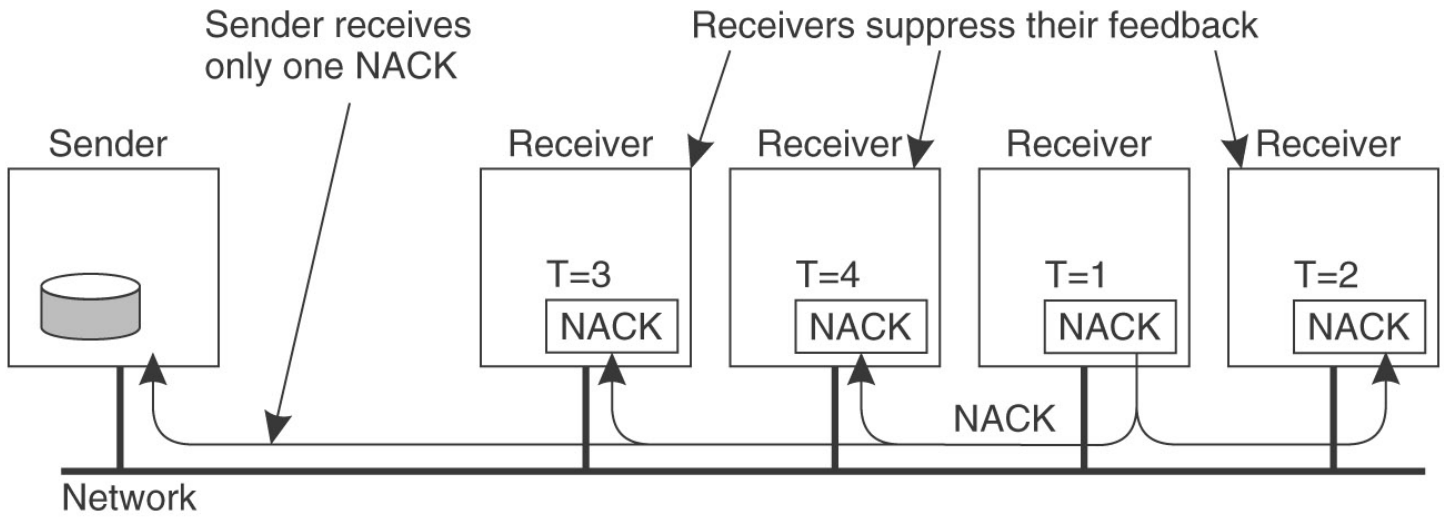
Anwendungslogik

Redundantes Stop Signal



Scalable Reliable Multicast (SRM – Floyd 97)

- Negative Acknowledgements werden per Multicast an alle Teilnehmer verteilt (wettbewerbsgesteuert)
- Sendewiederholungen gehen an die Empfängergruppe



Problem: Unerwünschte Nachrichten/Pakete

Virtual Synchrony

Können wir Multicast Gruppenkommunikation verwenden, um verlässlich Teilnehmer zu koordinieren?

Anforderungen:

Atomar - Nachrichten sind entweder bei allen Teilnehmern oder bei keinem zugestellt

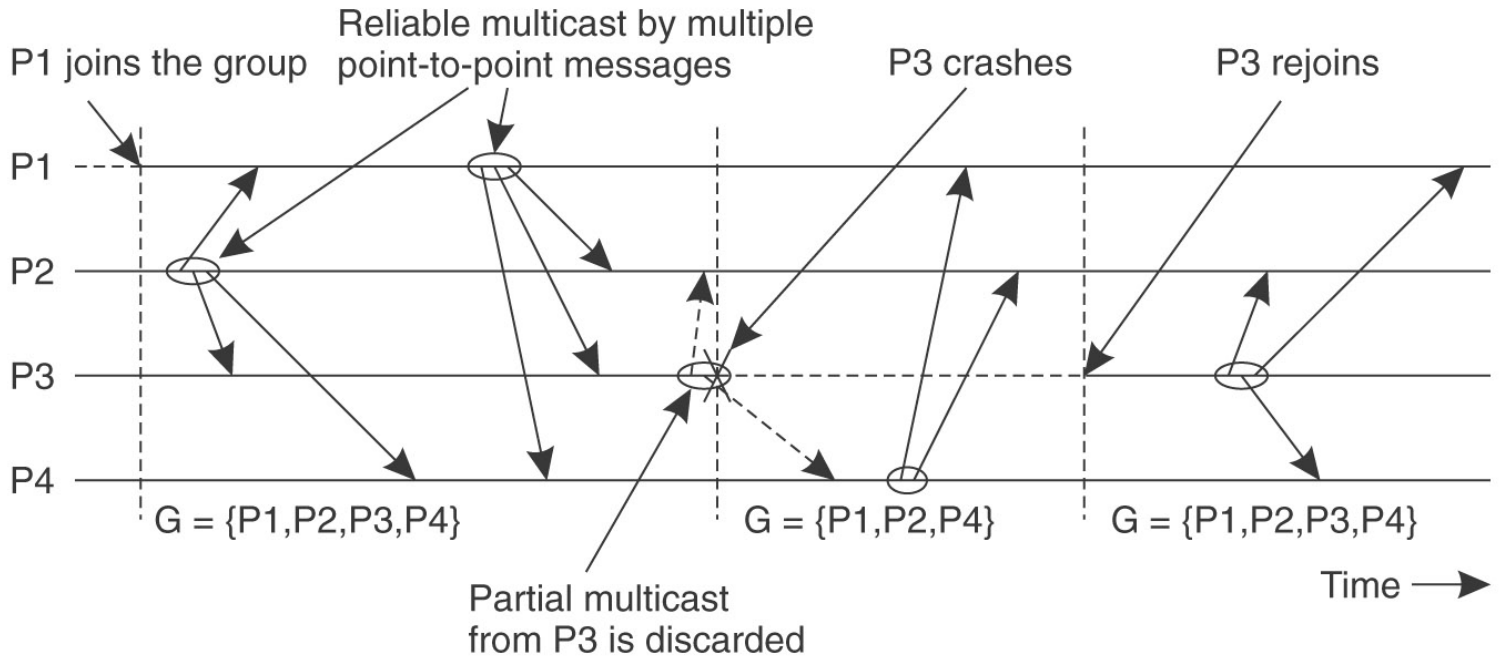
Total geordnet – Nachrichten werden bei allen Teilnehmern in der gleichen Reihenfolge verarbeitet

Gleichmäßige Gruppensicht – Alle funktionierenden Teilnehmer haben die gleichen Teilnehmerinformationen

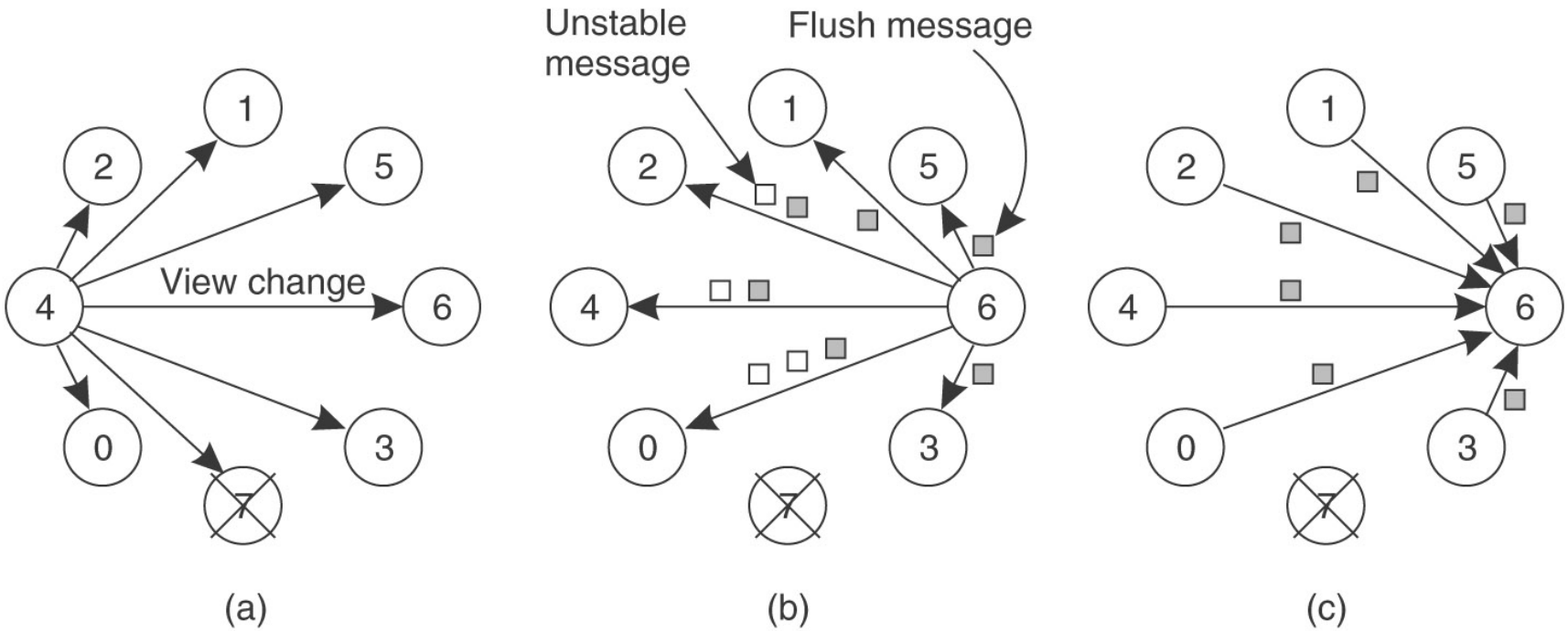
Zuverlässiger Multicast, der diese Anforderungen erfüllt, wird **virtuell synchron** (virtually synchronous) genannt.

Das Prinzip der synchronen Gruppensicht

- Gruppenkommunikation findet immer nur zwischen ‘Sichtveränderungen’ statt
  - Sichtveränderungen wirken wie eine Nachrichtenbarriere



Virtual Synchrony View Updates



- Eine Gruppensicht ist immer unmittelbare Grundlage für Koordinierungsentscheidungen