TU Braunschweig
Institut für Betriebssysteme
und Rechnerverbund

Verteilte Systeme

Prof. Dr. Stefan Fischer

Kapitel 8: Zeit, Nebenläufigkeit und Synchronisation

Überblick

- Das Phänomen der Zeit in Computersystemen
- Uhrensynchronisation
  - Christian‘s Algorithm
  - Der Berkeley-Algorithmus
  - Network Time Protocol (NTP)
- Logische Uhren (Lamport)
- Globale Systemzustände in verteilten Systemen
- Synchronisation von Prozessen
  - Auswahlalgorithmen (Election Algorithms)
  - Gegenseitiger Ausschluss (Mutual Exclusion)
  - Übereinstimmung (Consensus)

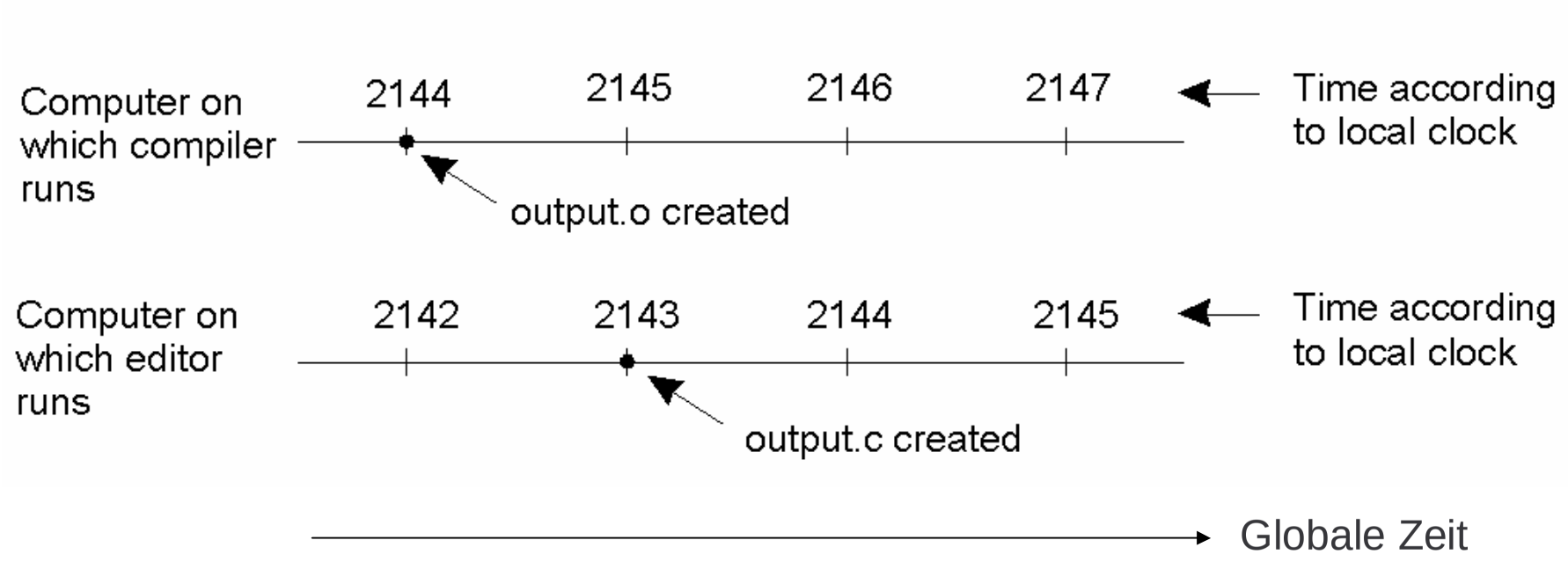
Motivation

- In den vergangenen Kapiteln haben wir uns hauptsächlich mit der *Kommunikation* zwischen Prozessen beschäftigt.
- Sehr wichtiges weiteres Problem: wie koordinieren und synchronisieren Prozesse sich, z.B.
  - Beim Zugriff auf gemeinsam verwendete Ressourcen
  - Bei der Feststellung, welcher Prozess ein bestimmtes Ereignis zuerst ausgelöst hat (verteilte Online-Auktionen)

Motivation

- Für viele dieser Algorithmen ist ein gemeinsames Verständnis der „Zeit“ in allen beteiligten Knoten notwendig.
- Dies ist in einem zentralisierten System kein Problem, da es dort nur eine Zeitquelle gibt.
- In verteilten Systemen hat jedoch jeder Knoten seine eigene Zeitquelle und damit u.U. eine andere Uhrzeit.
- Problem?

Beispiel: Verteilte SW-Entwicklung



- Die Quelldatei ist augenscheinlich älter als die Zieldatei. Ergebnis: Sie wird bei einem erneuten Make *nicht* neu übersetzt.

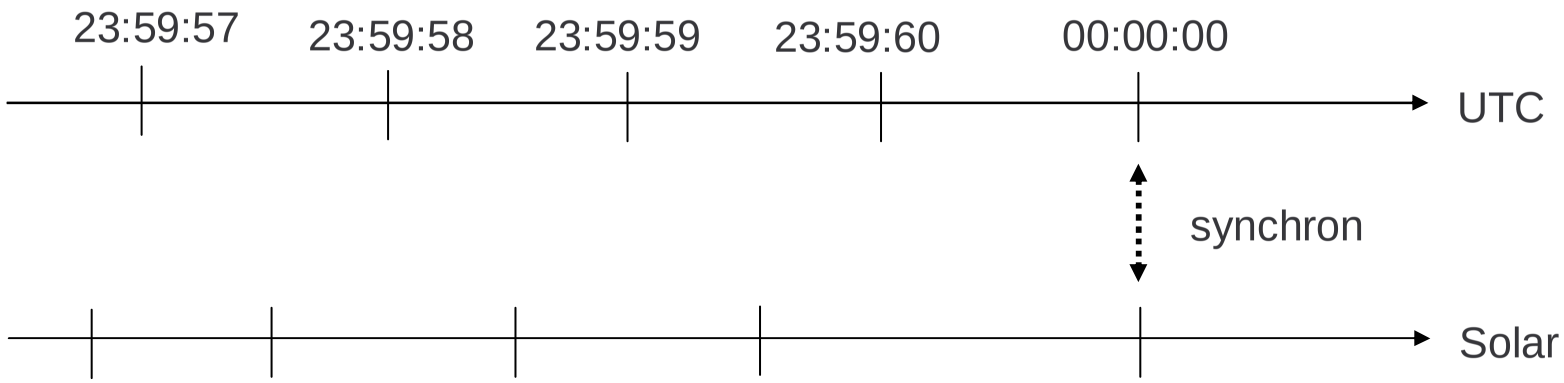
Zeit

- Jeden Tag gegen Mittag erreicht die Sonne ihren höchsten Punkt am Himmel.
- Die Zeitspanne zwischen zwei aufeinanderfolgenden Ereignissen dieses Typs heißt ... Tag (genauer gesagt: ein Sonnentag).
- Eine *Sonnensekunde* ist 1/86400 dieser Spanne.
- Die Zeitmessung findet heute etwas einfacher statt, mit Hilfe von Atomuhren: eine Sekunde ist die Zeit, die ein Cäsium-133-Atom benötigt, um 9.192.631.770 mal zu schwingen.
- Am 1.1.1958 entsprach diese Atomsekunde genau einer Sonnensekunde (wegen der Erdreibung wird die Sonnensekunde immer länger).

TAI und UTC

- Diese Internationale Atomuhrzeit (TAI) stimmt deshalb nicht 100% mit der Sonnenzeit überein, weshalb man einen kleinen Trick anwendet.
- Wenn die beiden Zeiten mehr als 800 ms auseinander liegen, dann wird eine „leap second“ eingeführt bzw. gelöscht.
- Diese neue Zeit heisst UTC – Universal Coordinated Time.
- Es gibt weltweit eine ganze Reihe von UTC-Sendern, für die man inzwischen recht preisgünstige Empfänger erhalten kann.
- Auch GPS sendet UTC-Zeit.

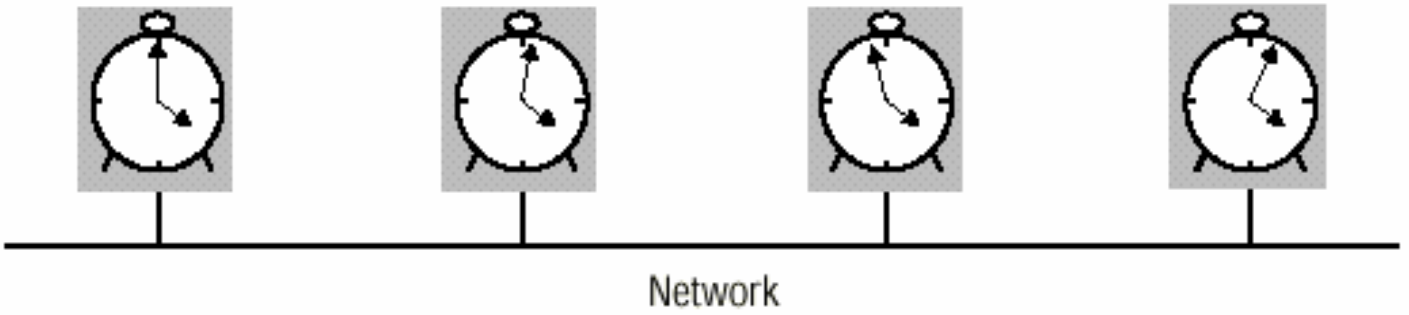
Schaltsekunden



- Wenn die beiden Zeiten nicht mehr „synchron genug“ laufen, werden sie durch eine Schaltsekunde angepasst.
- Meist wird die Zeit Ende oder Mitte des Jahres angepasst.
- Weitere Infos: http://www.ptb.de/de/org/4/43/432/ssec.htm

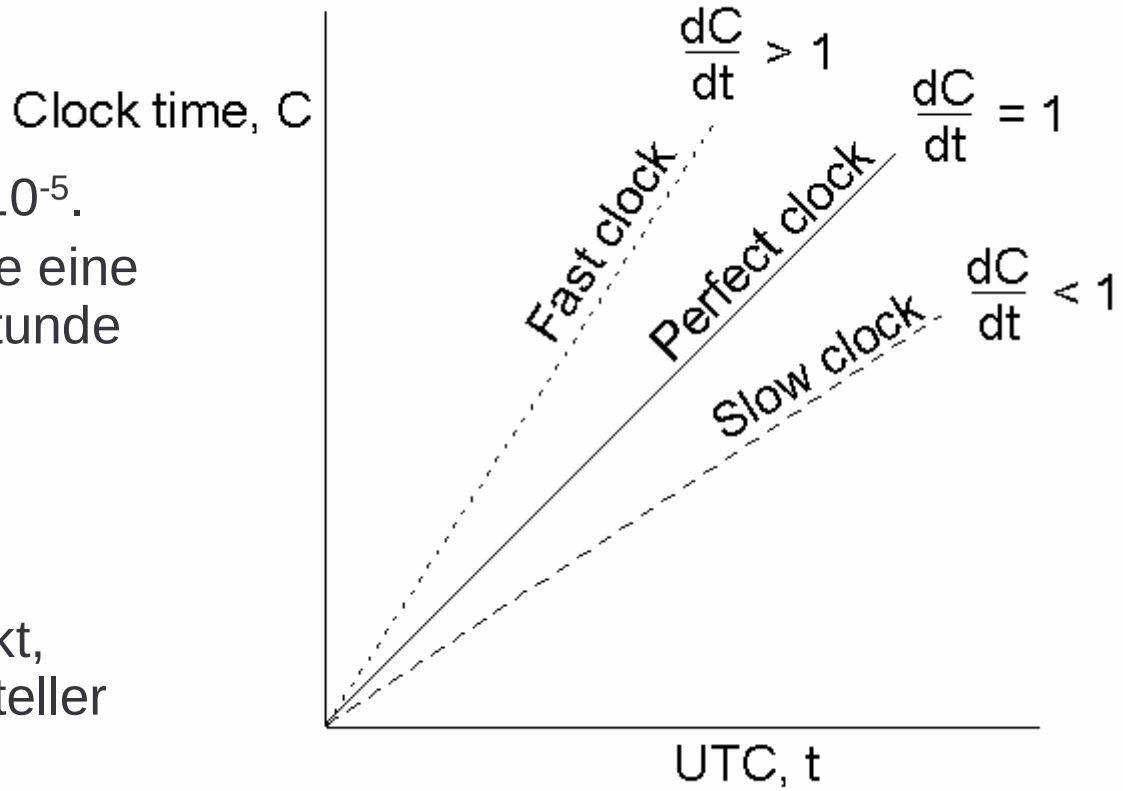
Zeit in verteilten Systemen

- Computer haben eine lokale Uhr, die H mal pro Sekunde einen Interrupt auslöst. Die Interrupts werden gezählt und messen die Zeit.
- Problem: die Uhren unterschiedlicher Computer zeigen unterschiedliche Zeiten an!
- Problem 1: unterschiedliche Startzeiten, kann relativ leicht gelöst werden
- Problem 2: unterschiedliche Laufzeiten!



Genauigkeit von Uhren

- Der Wert der Uhr von Maschine p zum UTC-Zeitpunkt t is $C_p(t)$.
- Chips haben eine Genauigkeit von etwa 10^{-5}.
- Beispiel: bei H=60 sollte eine Uhr 216.000 mal pro Stunde ticken.
- Realistisch ist ein Wert zwischen 215.998 und 216.002.
- Eine Uhr arbeitet korrekt, wenn sie die vom Hersteller angegebene maximale Driftrate ρ einhält, auch wenn sie dann etwas zu langsam oder zu schnell ist.

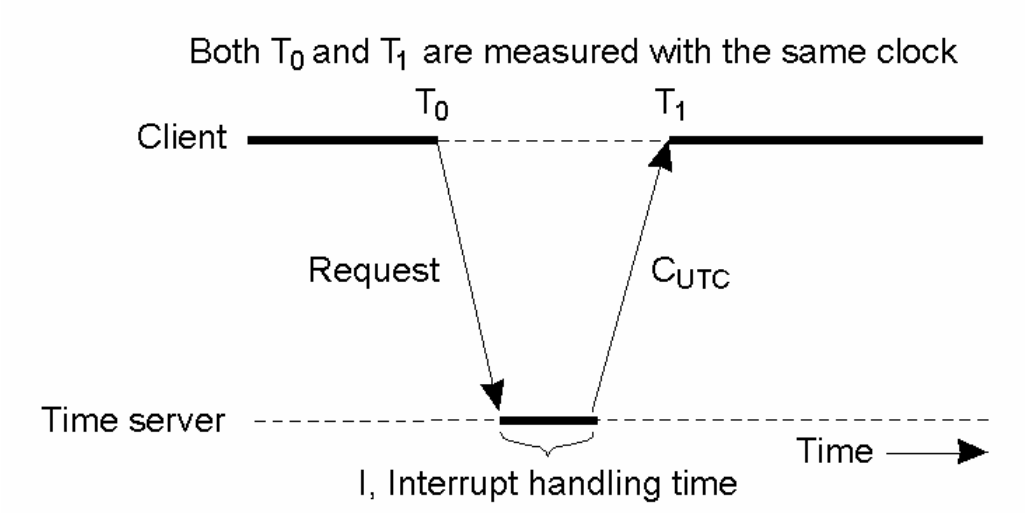


Uhrensynchronisation

- Folge: zu einem Zeitpunkt Δt nach der Synchronisation zweier Uhren können die beiden Uhren maximal 2ρΔt auseinander liegen.
- Will man sicherstellen, dass zwei Uhren niemals mehr als δ auseinander liegen, muss man die Uhren mindestens alle **δ/2ρ** Sekunden synchroniseren.
- Heute hat nicht jeder Rechner einen UTC-Empfänger, so dass man keine vollständige externe Synchronisation durchführen kann.
- Vielmehr gibt es eine Reihe von Algorithmen, die auf der Verwendung weniger Time-Server beruhen.

Der Algorithmus von Christian (1989)

- Es wird die Existenz eines UTC-Empfängers im System angenommen, der dann als *Time-Server* fungiert.
- Jede andere Maschine sendet mind. alle δ/2ρ Sekunden ein Time Request an den Server, der so schnell wie möglich mit der aktuellen UTC antwortet.



Algorithmus von Christian

- Nun könnte man die Uhr auf die erhaltene UTC-Zeit setzen.
- Erstes ABER:
  - Zeit darf niemals rückwärts laufen, so dass ein Zeitpunkt zweimal auftauchen kann! Wenn der Anfragende eine schnelle Uhr hat, dann kann UTC u.U. stark hinterher laufen. Ein Setzen auf den UTC-Wert wäre dann falsch.
- Lösung: die Uhren werden nicht auf einmal angepasst, sondern graduell, so lange, bis die richtige Zeit erreicht ist (ein Clock-Tick wäre nicht mehr z.B. 10ms, sondern nur noch 9ms)

Algorithmus von Christian

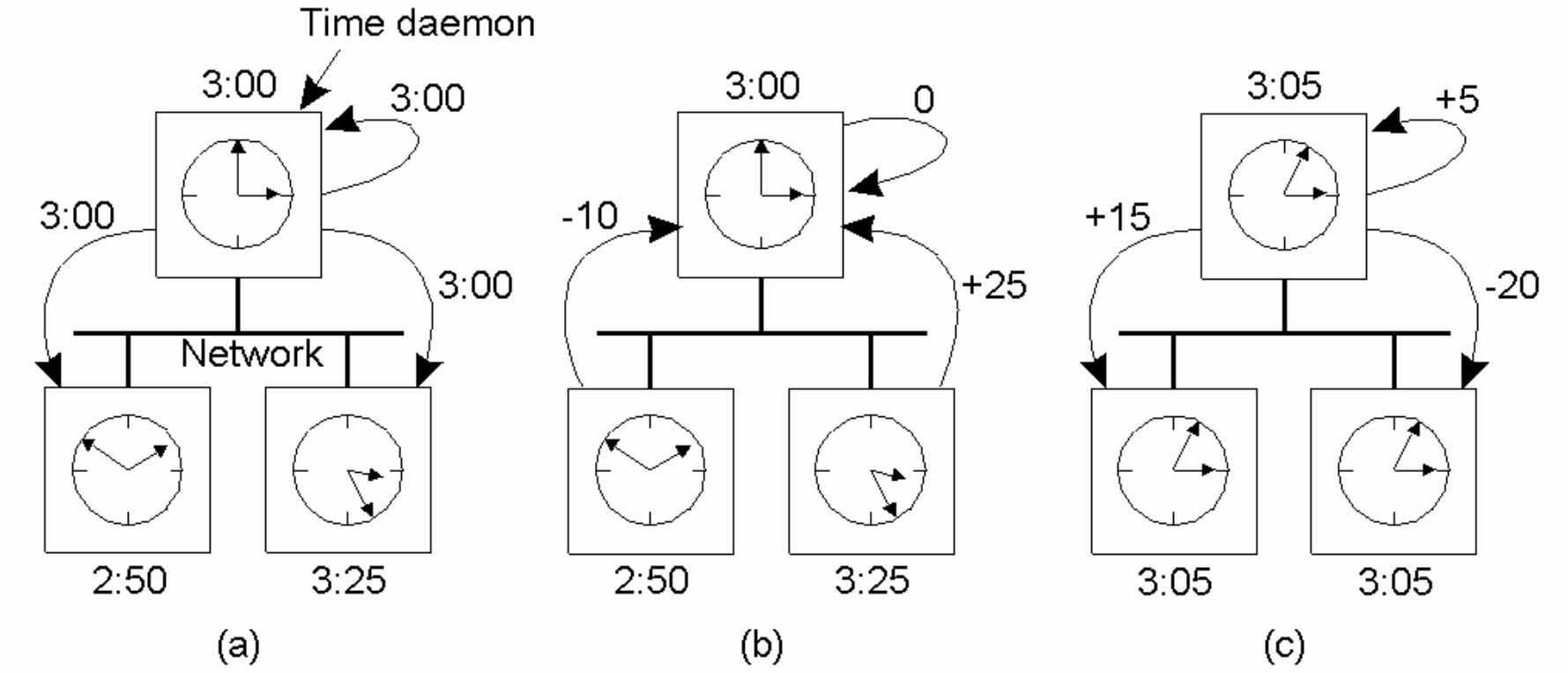
- Zweites ABER:
  - Die Signallaufzeit für die Nachrichten ist größer als 0. Wenn die Antwort kommt, ist sie praktisch schon veraltet.
- Lösung von Christian: versuche die Signallaufzeit zu messen
- Beste Abschätzung, wenn sonstige Informationen fehlen: $(T_1-T_0)/2$
- Wenn die ungefähre Zeit I bekannt ist, die der Time-Server zur Abarbeitung des Requests benötigt: $(T_1-T_0-I)/2$
- Verbesserung des Ergebnisses: messe diesen Wert häufig und nehme den Durchschnitt, aber lasse „Ausreißer“ aus der Wertung

Der Berkeley-Algorithmus

- Entwickelt für eine Gruppe von Berkeley-UNIX-Rechnern ebenfalls 1989
- Sinnvoll, wenn kein UTC-Empänger zur Verfügung steht.
- Ein Rechner ist der Koordinator (=Time-Server)
- Dieser Server ist im Gegensatz zu dem Server von Christian *aktiv*. Er fragt bei allen anderen Rechnern nach der aktuellen Zeit und bildet einen Durchschnittswert.
- Dieser wird dann als neuer aktueller Uhrenwert an alle anderen gegeben.

Der Berkeley-Algorithmus

- Alle anderen sollten dann ihre Uhren entsprechend langsamer oder schneller laufen lassen, bis alle Uhren in etwa mit dem Durchschnittswert übereinstimmen.

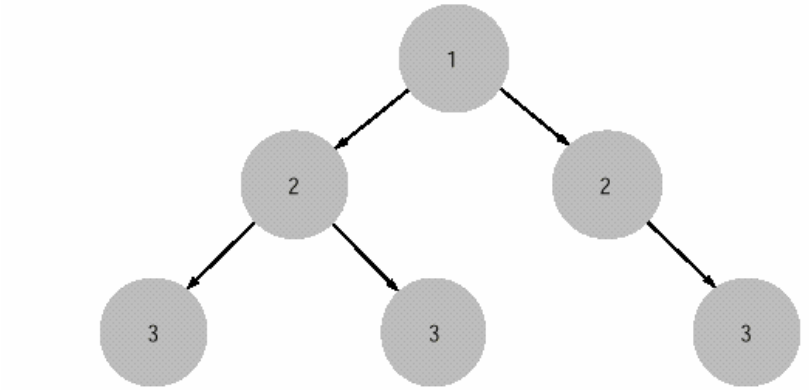


Network Time Protocol NTP

- Entwickelt in den 80er Jahren, inzwischen ein Internet RFC (mit diversen Verbesserungen)
- Ziele:
  - Clients sollen sich möglichst genau mit UTC synchronisieren können, trotz stark schwankender Übertragungsverzögerungen im Netz
  - Bereitstellung eines zuverlässigen Dienstes mittels Redundanz
  - Clients sollen in der Lage sein, sich oft zu synchronisieren, Skalierbarkeit wird damit ein Thema

Funktionsweise von NTP

- Der NTP-Dienst wird von einem Netzwerk von Servern erbracht.
- Die „primary server“ sind direkt mit der UTC-Quelle verbunden.
- Die „secondary server“ synchronisieren sich mit den „primary servers“ etc.



Note: Arrows denote synchronization control, numbers denote strata.

Funktionsweise von NTP

- Das Server-Netzwerk ist rekonfigurierbar, um auf Fehler entsprechend reagieren zu können.
- Die Server tauschen häufig Nachrichten aus, um Netzwerkverzögerungen und Uhrungenauigkeiten zu messen.
- Basierend auf diesen Messungen lässt sich die Qualität eines Servers abschätzen und Clients können die beste Quelle wählen.

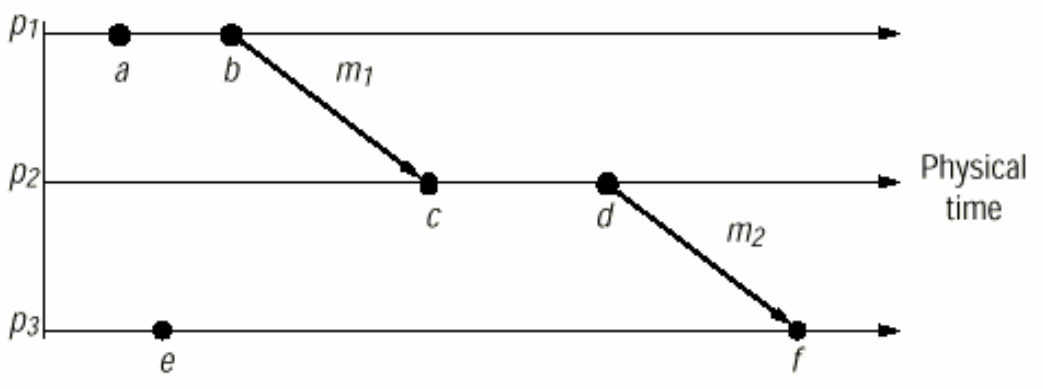
Logical Time

- Es ist generell unmöglich, physikalische Uhren in einem verteilten System absolut zu synchronisieren.
- Damit ist es auch nicht möglich, basierend auf diesem System die Reihenfolge zweier beliebiger Ereignisse zu bestimmen.
- Für einige Anwendungen benötigt man jedoch genau diese Information, dafür aber keinen Bezug zur realen Zeit.
- Die Lösung dafür: logische Zeit (logical time)
  - Wenn zwei Ereignisse im selben Prozess stattfinden, dann fanden sie in der Reihenfolge ihrer Beobachtung statt.
  - Wenn eine Nachricht zwischen zwei Prozessen ausgetauscht wird, dann ist das Sendereignis immer vor dem Empfangsereignis.

Die *Happened-Before*-Beziehung

- Aus diesen Beobachtungen machte Lamport die Happened-Before-Relation „ $\rightarrow$“oder auch die „relation of causal ordering“:
  - Wenn in einem Prozess p_i gilt: $a \rightarrow_i b$, dann gilt allgemein für das System: $a \rightarrow b$
  - Für jede Nachricht m gilt: send(m) $\rightarrow$ receive(m), wobei send(m) das Sendeereignis im sendenden Prozess und receive(m) das Empfangsereignis im empfangenden Prozess darstellt
  - Wenn $a \rightarrow b$ und $b \rightarrow c$ gilt, dann gilt auch $a \rightarrow c$.
- Ereignisse, die nicht in dieser Beziehung stehen, werden als *nebenläufig* bezeichnet.

Beispiel



- a $\rightarrow$ b, b $\rightarrow$ c, c $\rightarrow$ d, d $\rightarrow$ f, und also a $\rightarrow$ f, aber nicht a $\rightarrow$ e
- Wenn a $\rightarrow$ b gilt, wird eine kausale oder möglicherweise kausale Beziehung angenommen.

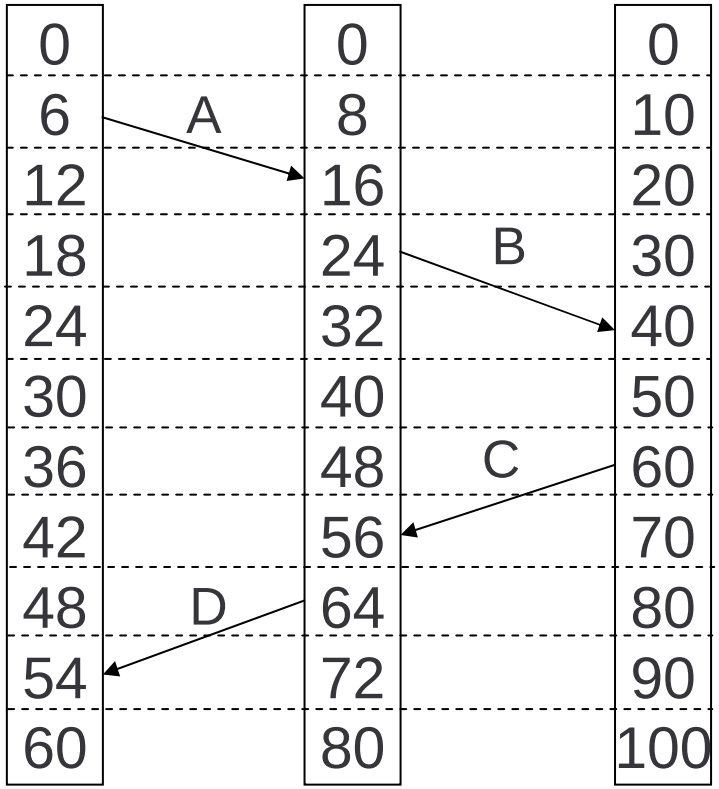
Praktische Umsetzung

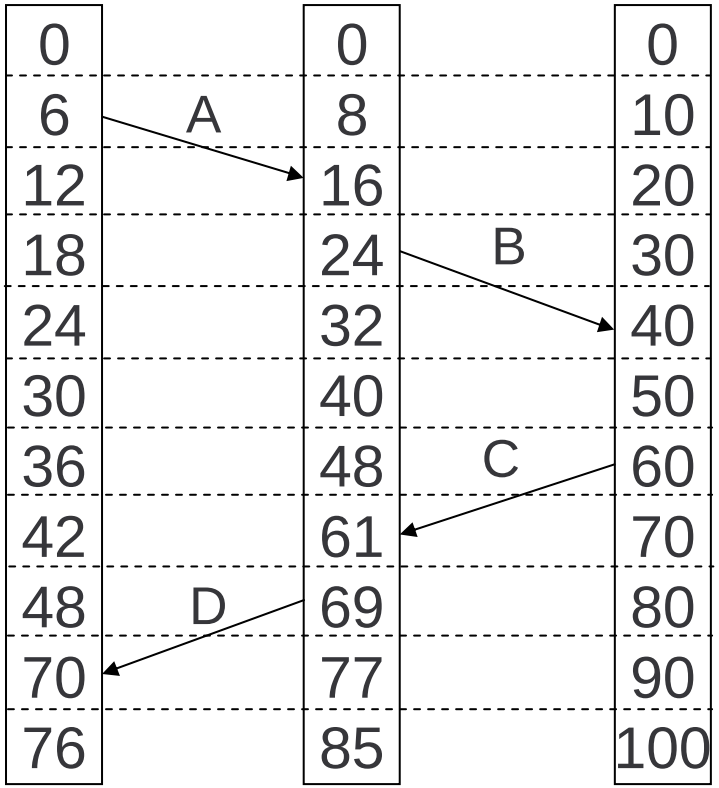
- Jeder Prozess P_i hat nun eine logische Uhr, die beim Auftreten eines Ereignisses a abgelesen wird und den Wert $C_i(a)$ liefert.
- Dieser Wert muss so angepasst werden, dass er als C(a) eindeutig im ganzen verteilten System ist.
- Ein Algorithmus, der die logischen Uhren entsprechend richtig stellt, muss folgendes umsetzen:
  Wenn $a \rightarrow b$, dann $C(a) < C(b)$.

Umsetzung von Happens-Before

- Prozesse wenden deshalb den folgenden Algorithmus an, um ihre Uhren richtig zu stellen:
  - C_i wird vor jedem neuen Ereignis in P um eins erhöht: $C_i := C_i + 1$. Der neue Wert ist der „Timestamp“ des Ereignisses.
  - Wenn ein Prozess P_i eine Nachricht m sendet, dann sendet er den Wert $t= C_i$ mit.
  - Bei Erhalt von (m,t) berechnet P_j den neuen Wert $C_j := \max(C_j, t)$ und wendet dann die erste Regel an, um den Timestamp für receive(m) festzulegen.

Lamports Timestamps



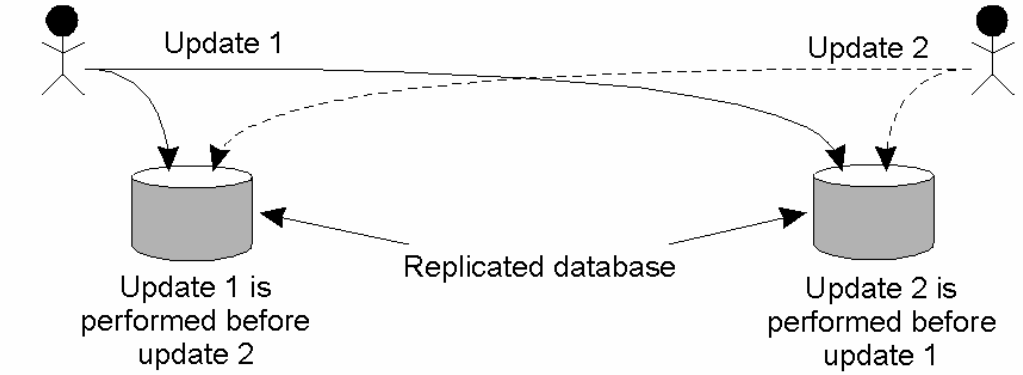


a) Drei Prozesse mit unterschiedlichen Clock Rates.

b) Lamports Algorithmus löst das Problem.

Beispiel: Replizierte Datenbank

- Zwei Kopien der selben Datenbank
- Kunde überweist $100 auf sein $1000-Konto.
- Gleichzeitig überweist ein Bankangestellter 1% Zinsen auf den aktuellen Kontostand.
- Die Operationen werden per Multicast an beide DBS geschickt und kommen dort in unterschiedlicher Reihenfolge an, Ergebnis: Inkonsistenz ($1111 vs $1110)

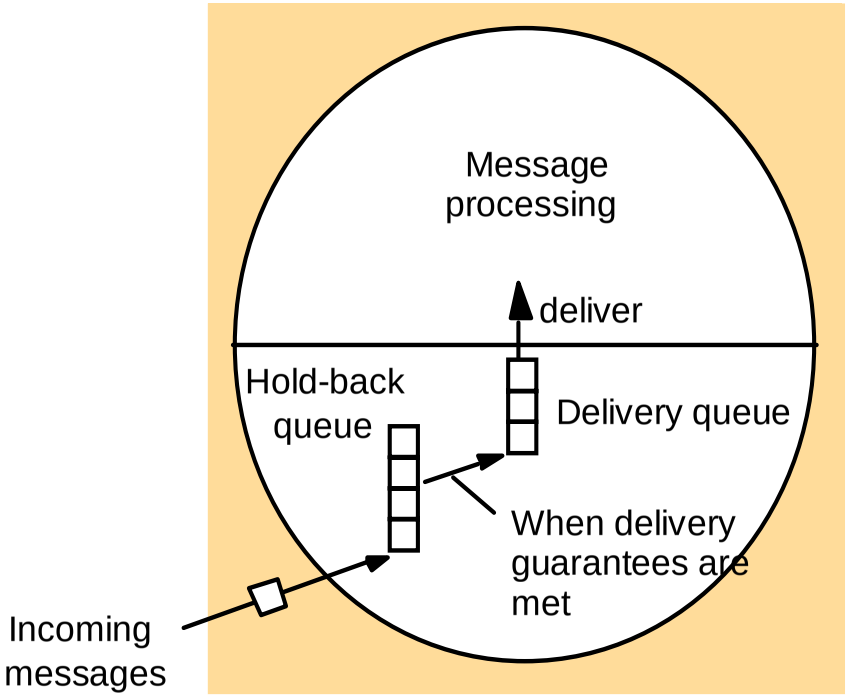


Lösung: Totally-Ordered Multicast

- Anforderung bei solchen Problemen: alle Kopien müssen immer im gleichen Zustand sein
- Lösung dafür: totally-ordered multicast, d.h., alle Nachrichten werden bei allen Empfängern in derselben Reihenfolge ausgeliefert
- Annahmen:
  - Reihenfolgeerhaltung
  - Nachrichten gehen nicht verloren
- Implementierung??

Totally-Ordered Multicast

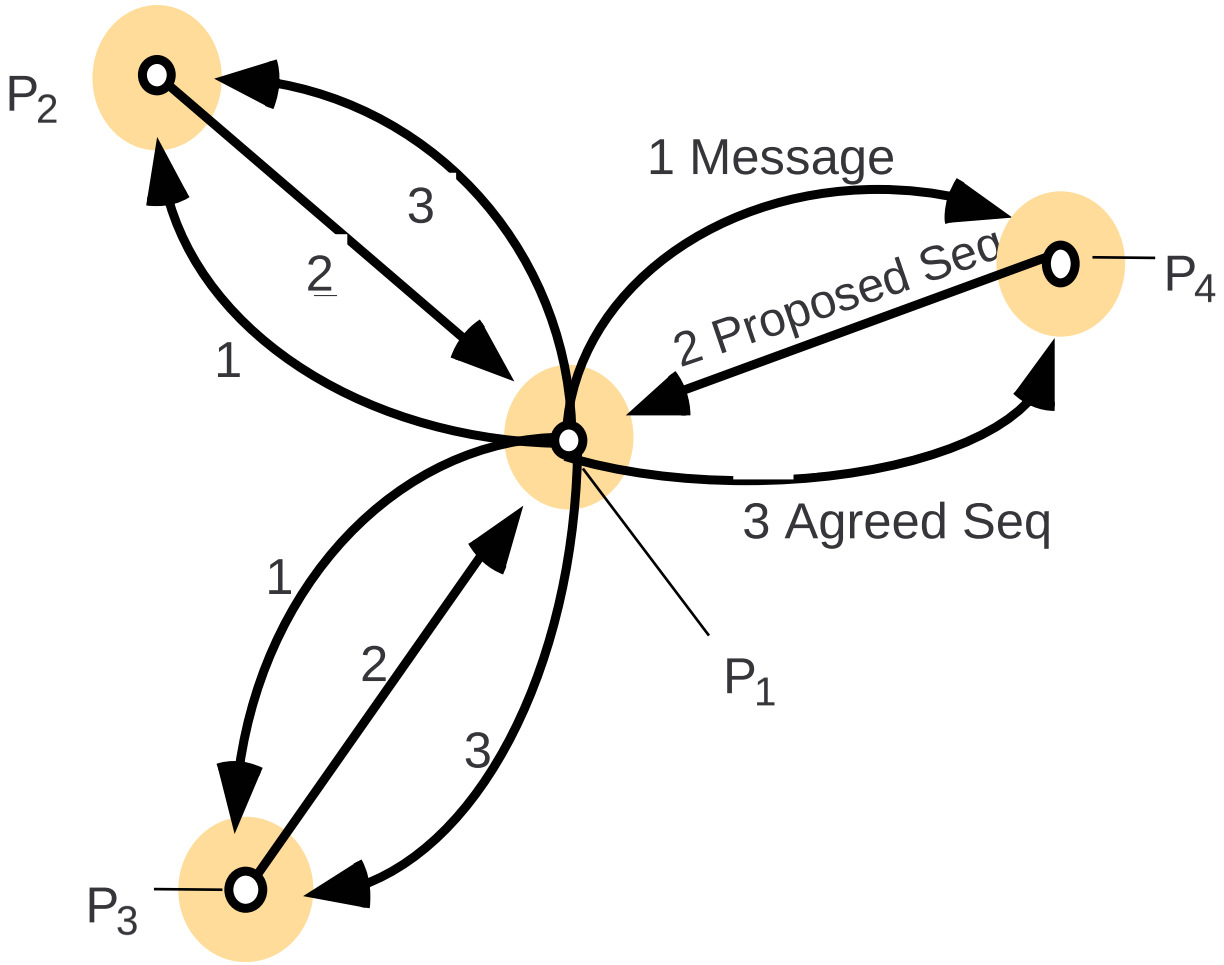
- Jeder Prozess hat eine lokale Warteschlange für eingehende Nachrichten, geordnet nach den Timestamps, die Hold-Back Queue. Elemente hierin dürfen nicht ausgeliefert werden.
- Außerdem hat jeder Prozess eine Delivery-Queue. Elemente hierin werden entsprechend ihrer Reihefolge an die Anwendung ausgeliefert.



Umsetzung: der ISIS-Algorithmus

- Der Algorithmus verwendet im Prinzip Lamports Timestamps.
- Vorgehen:
  - Prozess P sendet per Multicast eine Nachricht <m,i>, wobei i ein eindeutiger Identifier ist (Prozess-ID + Timestamp).
  - Alle Prozesse fügen die Nachricht in die Hold-Back-Queue ein und antworten mit einem Vorschlag für eine akzeptierte Sequenznummer a.
  - P sammelt diese Vorschläge, wählt den höchsten Wert aus (der ist dann sicher von den anderen Prozessen noch nicht vergeben) und schickt die Nachricht <i,a> an alle Prozesse zurück.
  - Die Prozesse verwenden a als die neue Sequenznummer für die Nachricht mit Nummer i.
  - Wenn die Nachricht am Anfang der Hold-Back-Queue ist, wird sie zur Delivery-Queue transferiert.

Der ISIS-Algorithmus

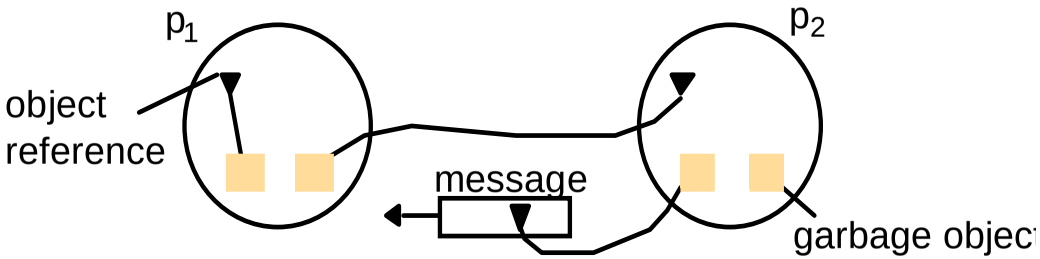


Globale Systemzustände

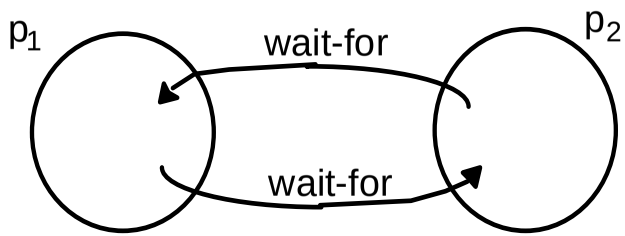
- Es gibt eine Reihe von Gelegenheiten, bei denen man gern über den Gesamtzustand des verteilten Systems Bescheid wüsste
- Der Gesamtzustand des Systems besteht aus
  - Den lokalen Zuständen der Einzelkomponenten (der Prozesse) und
  - Allen Nachrichten, die sich zur Zeit in der Übertragung befinden.
- Diesen Zustand exakt zur selben Zeit bestimmen zu können ist so unmöglich wie die exakte Synchronisation von Uhren – es lässt sich kein globaler Zeitpunkt festlegen, an dem alle Prozesse ihre Zustände festhalten sollen.

Anwendungen des globalen Zustands

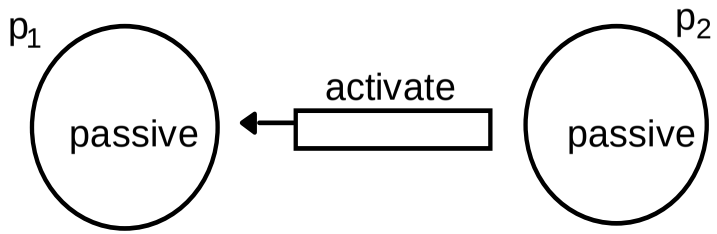
a. Garbage collection



b. Deadlock



c. Termination

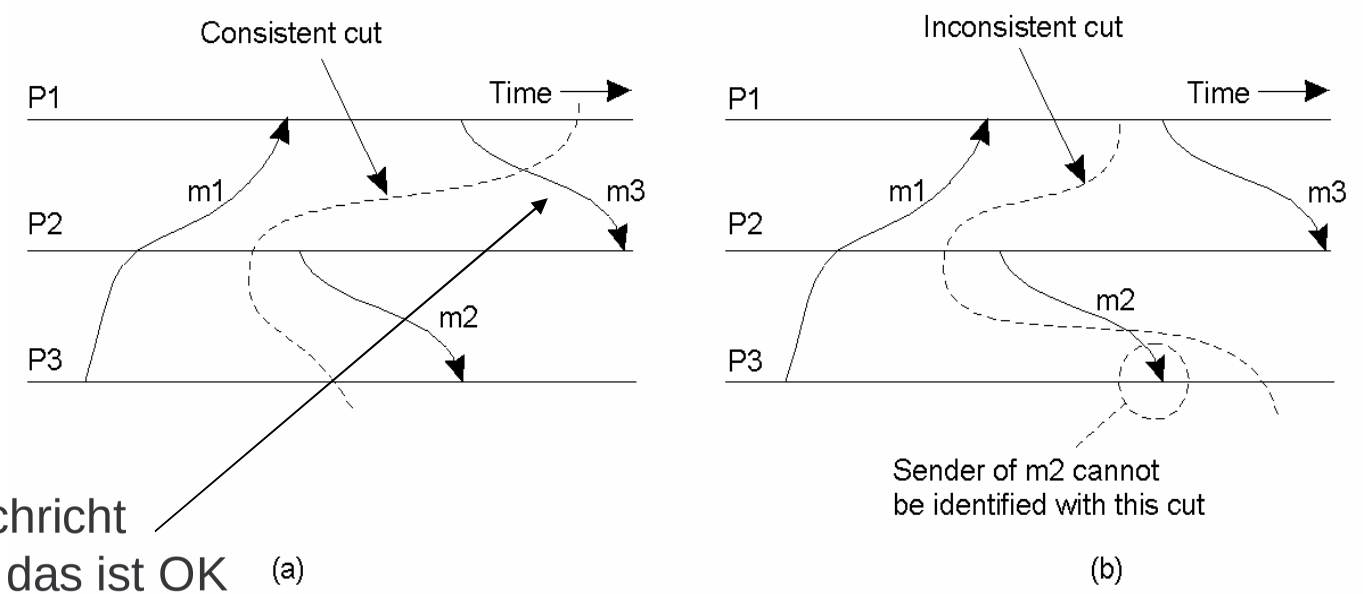


Distributed Snapshot

- Wie kann man nun den globalen Zustand eines verteilten Systems ermitteln?
- Lösung von Chandy und Lamport (1985): Distributed Snapshot:
  - ermittle einen Zustand, in dem das System möglicherweise war,
  - der aber auf jeden Fall konsistent ist
- Konsistenz bedeutet insbesondere: wenn festgehalten wurde, dass Prozess P eine Nachricht m von einem Prozess Q empfangen hat, dann muss auch festgehalten sein, dass Q diese Nachricht geschickt hat. Sonst kann das System nicht in diesem Zustand gewesen sein.

Consistent und Inconsistent Cut

- Definition der Konsistenz über den sog. „cut“, der für jeden Prozess das letzte aufgezeichnete Ereignis angibt.



m3 ist Nachricht
in Transit; das ist OK

Formale Definition des Cut

- Gegeben sei ein System $\wp$ von N Prozessen p_i (i=1,...,N).
- Betrachtet man nun den globalen Zustand $S=(s_1,...,s_N)$ des Systems, dann ist die Frage, welche globalen Zustände möglich sind.
- In jedem Prozess findet eine Serie von Ereignissen statt, womit jeder Prozess mittels der Geschichte seiner Ereignisse charakterisiert werden kann: $history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, ...>$
- Jeder endliche Präfix der Geschichte eines Prozesses wird bezeichnet mit $h_i^k = <e_i^0, e_i^1, e_i^2, ..., e_i^k>$

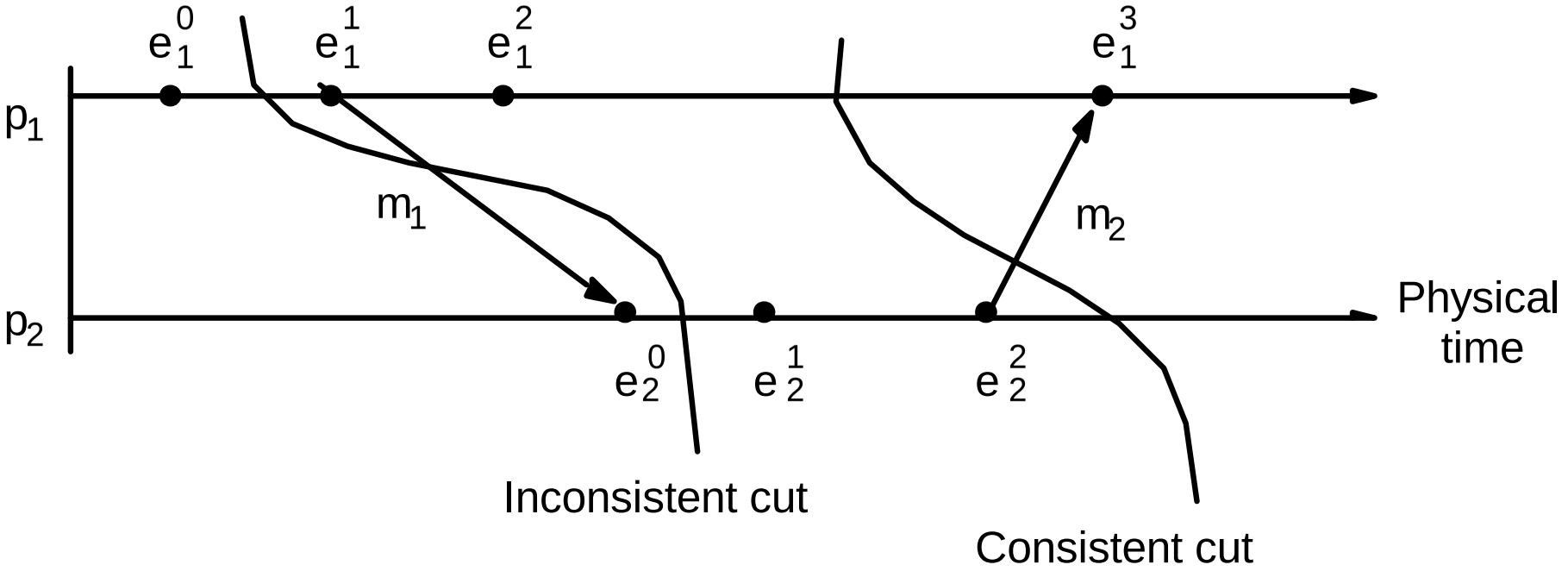
Formale Definition des Cut

- Ein Cut ist damit definiert wie folgt

$$C = h_1^{c_1} \cup h_2^{c_2} \cup ... \cup h_N^{c_N}$$

- Der Zustand s_i aus dem globalen Zustand ist dann genau derjenige von p_i, der durch das Ausführen des letzten Ereignisses im Cut erreicht wird, also von $e_i^{c_i}$
- Die Menge $\{e_i^{c_i} : i = 1,2,...N\}$ wird als Frontier des Cuts bezeichnet.

Beispiel



Frontier: $< e_1^0, e_2^0 >$ $< e_1^2, e_2^2 >$

Definition des konsistenten Cut

- Ein Cut ist dann konsistent, wenn er für jedes Ereignis, das er enthält, auch alle Ereignisse enthält, die zu diesem Ereignis in der Happened-Before-Relation stehen:

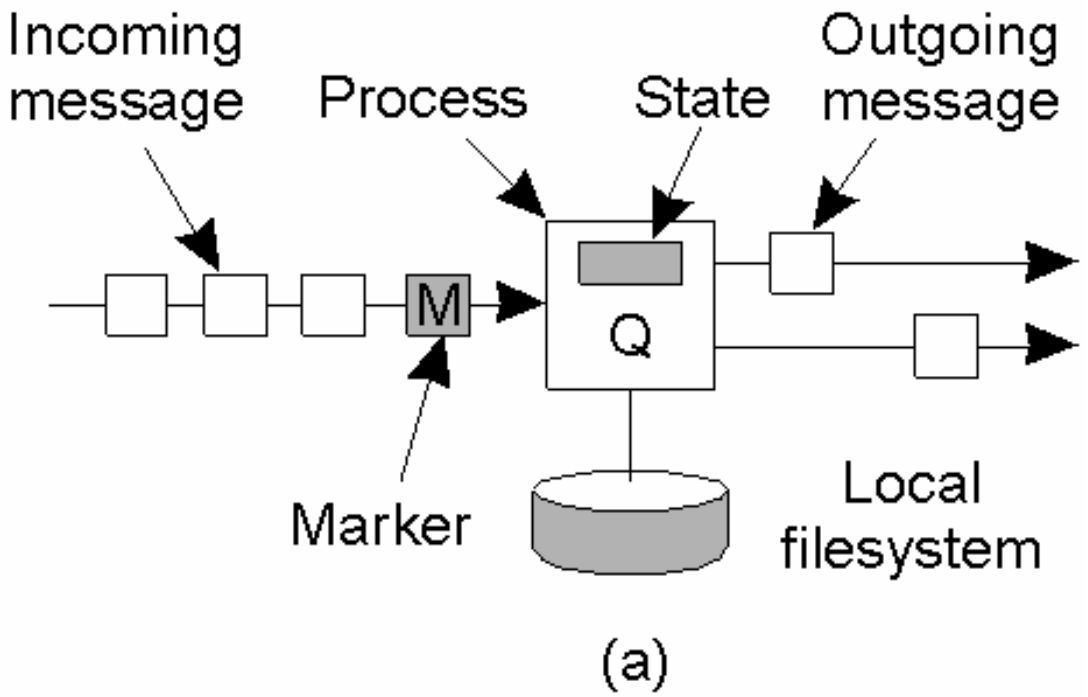
$$\forall e \in C, f \rightarrow e \Rightarrow f \in C$$

- Ein globaler Zustand ist konsistent, wenn er mit einem konsistenten Cut korrespondiert.

Zurück zum Chandy-Algorithmus

- Prozesse sind mittels Punkt-zu-Punkt-Kanälen verbunden.
- Ein oder mehrere Prozesse starten den Algorithmus zur Feststellung eines Systemzustands, so dass gleichzeitig immer mehrere Snapshots erstellt werden können.
- Das System läuft unterdessen ungehindert weiter.
- Die Prozesse verständigen sich über Markierungsnachrichten über die Notwendigkeit der Speicherung eines Systemzustands.

Prozessmodell für den Algorithmus



Der Algorithmus

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel *c*:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of *c* as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of *c* as the set of messages it has received over *c* since it saved its state.

end if

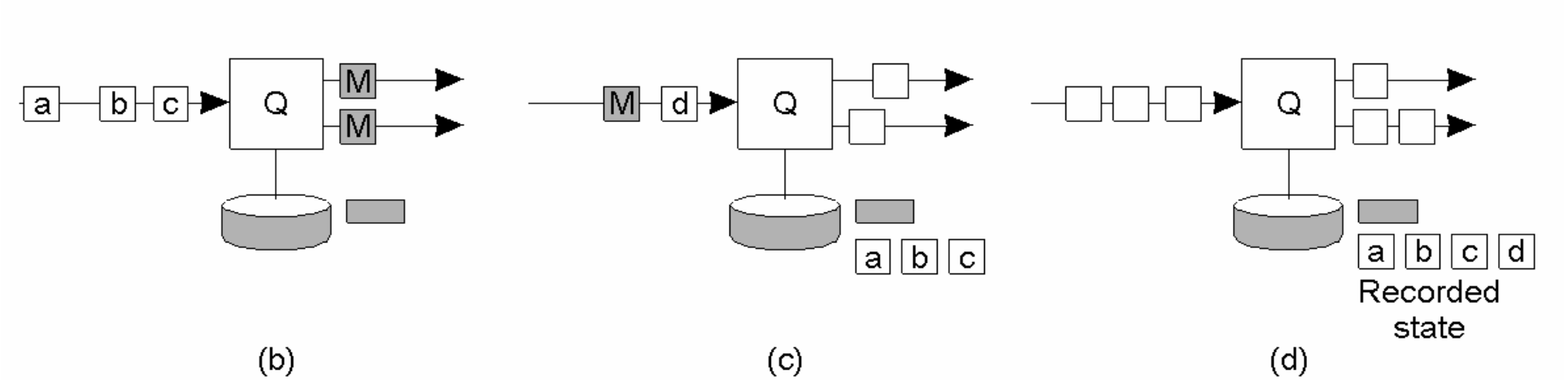
Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel *c*:

p_i sends one marker message over *c*

(before it sends any other message over *c*).

Ablauf des Algorithmus



b) Prozess Q erhält zum ersten Mal einen Marker und hält seinen lokalen Zustand fest

c) Q hält alle ankommenden Nachrichten fest

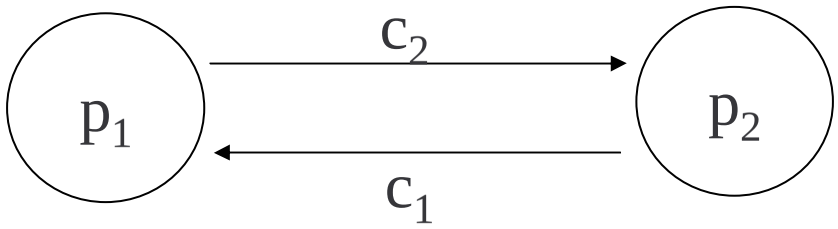
d) Q erhält einen Marker auf seinem Eingangskanal und stoppt die Aufzeichnung für diesen Kanal

Ende des Algorithmus

- Wenn Q einen Marker auf allen Eingangskanälen erhalten und verarbeitet hat, ist für diesen Prozess der Algorithmus beendet.
- Q sendet dann seinen lokalen Zustand sowie die aufgezeichneten Nachrichten für alle Eingangskanäle an den initiierenden Prozess.
- Dieser wertet schließlich das Ergebnis entsprechend aus, analysiert also z.B. bestimmte Zustandsprädikate.
- Man kann beweisen, dass dieser Algorithmus immer einen konsistenten Cut erzeugt.

Beispiel

- 2 Prozesse p_1 und p_2 kommunzieren.
- p_2 verkauft Schrauben zum Preis von 10 DM das Stück.
- p_1 kauft Schrauben.
- Der Zustand der beiden Prozesse wird durch die Zahl der Schrauben und den Kontostand bestimmt.
- Zwischen den Prozessen gibt es zwei Kommunikationskanäle c_1 und c_2.

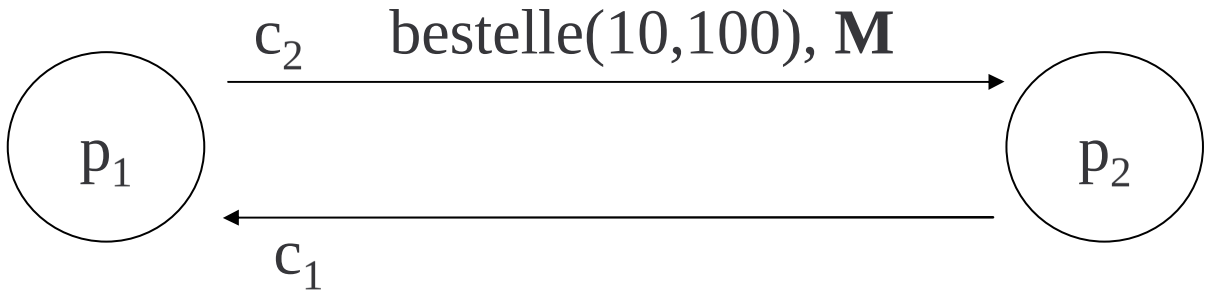


<Konto: DM 1000, Schrauben: 0>

<Konto: DM 50, Schrauben: 2000>

Ablauf des Beispiels - 1

- Sei der so gegebene Zustand s_0, in dem p_1 nun das Recording startet.
- p_1 sendet eine Marker-Nachricht an p_2 und dann eine Kaufanfrage für 10 Schrauben.
- Das System ist dann im Zustand s_1 wie folgt:

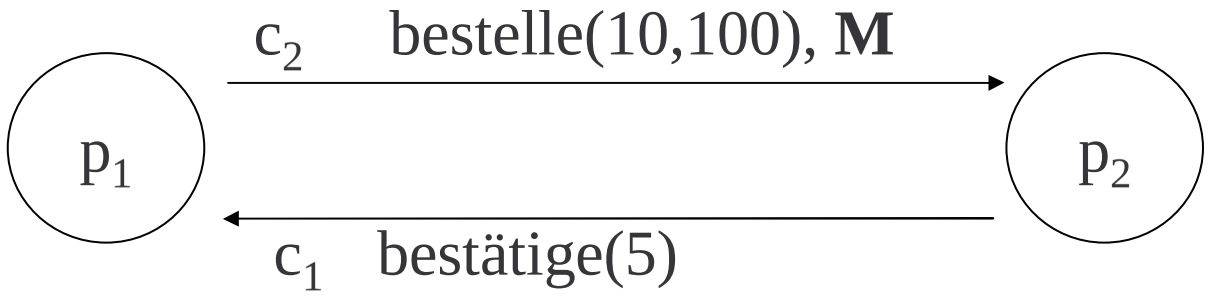


<Konto: DM 900, Schrauben: 0>

<Konto: DM 50, Schrauben: 2000>

Ablauf des Beispiels - 2

- Bevor p_2 den Marker erhält, sendet er die Bestätigung über einen früheren Schraubenkauf an p_1. Dadurch ergibt sich Zustand s_2 wir folgt.



<Konto: DM 900, Schrauben: 0>

<Konto: DM 50, Schrauben: 1995>

Ablauf des Beispiels - 3

- Nun empfängt p1 die Nachricht von p2 und p2 empfängt den Marker.
- p2 zeichnet den Zustand <50,1995> auf und den von c2 als leeren Kanal. Schließlich schickt er selbst einen Marker an p1.
- Nach Erhalt des Markers zeichnet p1 den Zustand von c1 auf (eine Nachricht). Resultierender Zustand:

<Konto: DM 900, Schrauben: 5>

<Konto: DM 50, Schrauben: 1995>

Aufgezeichneter Zustand

- Der aufgezeichnete Zustand ist dann:
  - p_1: <1000,0>
  - p_2: <50,1995>
  - c_1: <bestätige(5)>
  - c_2: <>
- Dies entspricht keinem der wirklichen globalen Zustände, ist aber ein möglicher Zustand des Systems.

Auswahlalgorithmen

- 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 Auswahlalgorithmus ist es, einen Prozess unter vielen gleichartigen zu bestimmen, der diese Rolle übernimmt.
- Wichtigstes Ziel: am Ende der Wahl sind sich alle darüber einig, wer der neue Koordinator ist.

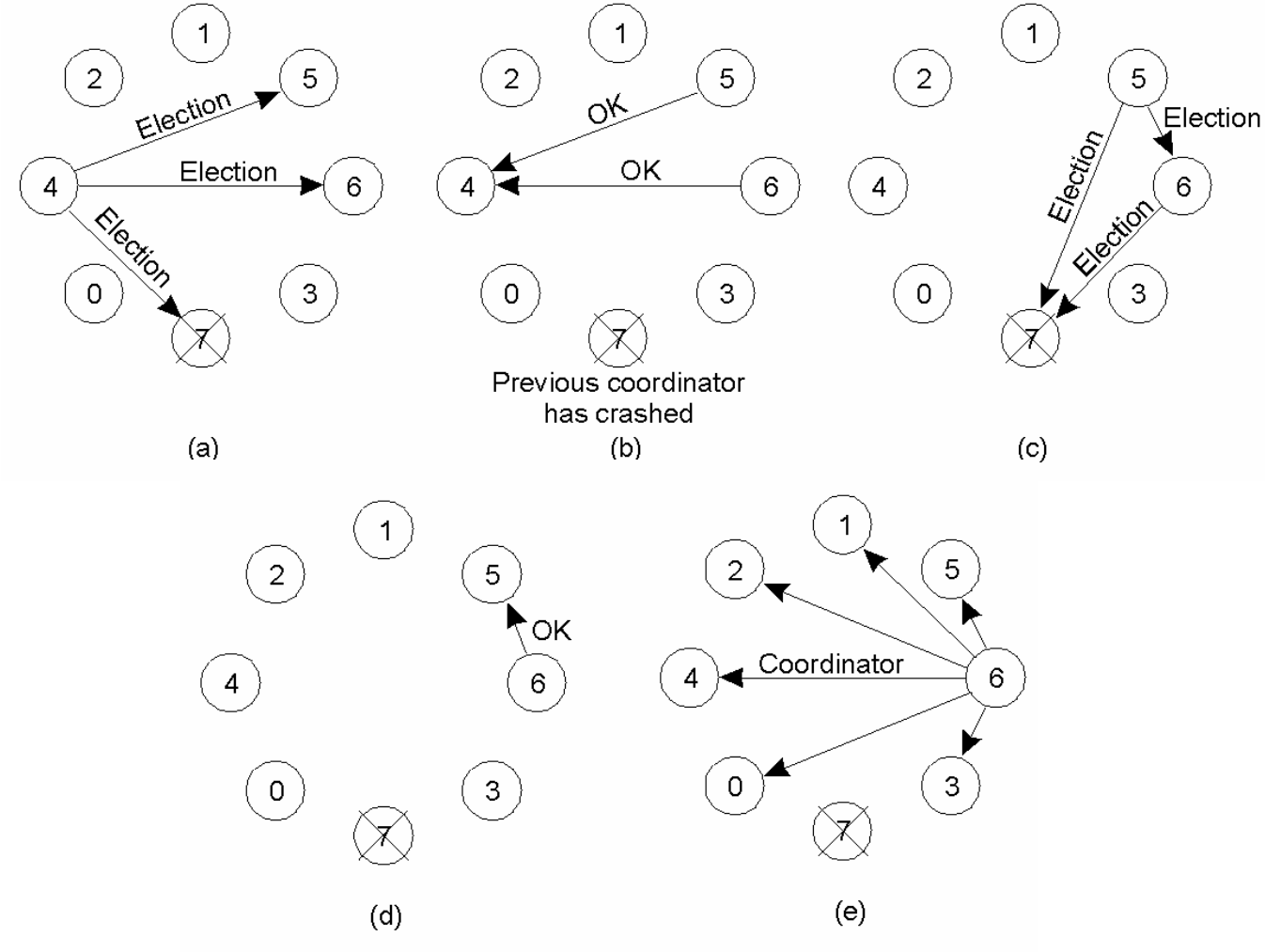
Auswahlalgorithmen

- Vorgehen:
  - Jeder Prozess hat eine Nummer, die allen anderen Prozessen bekannt ist.
  - Kein Prozess weiß, welcher andere Prozess gerade funktioniert oder nicht läuft.
  - Alle Algorithmen wählen den Prozess mit der höchsten Nummer aus. Der Weg kann sehr unterschiedlich sein.
- Bekannte Algorithmen:
  - Bully-Algorithmus
  - Ring-Algorithmus

Der Bully-Algorithmus

- Wenn ein Prozess feststellt, dass der augenblickliche Koordinator nicht mehr reagiert, startet er den Auswahlprozess:
  - P schickt eine ELECTION-Nachricht an alle Prozesse mit höherer Nummer
  - Bekommt er keine Antwort, ist er der neue Koordinator.
  - Bekommt er eine Antwort, ist seine Aufgabe erledigt. Der Antwortende übernimmt seine Arbeit.

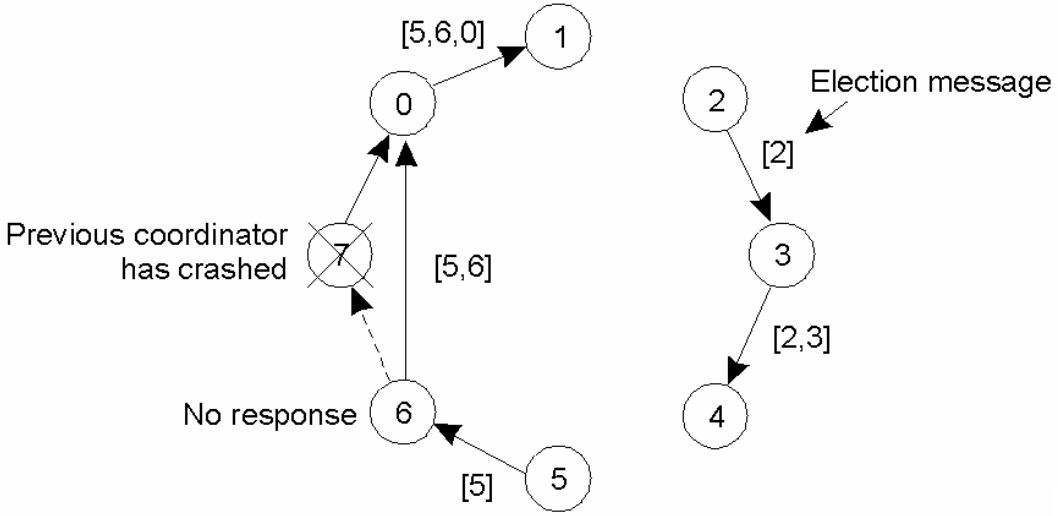
Beispiel: Bully-Algorithmus

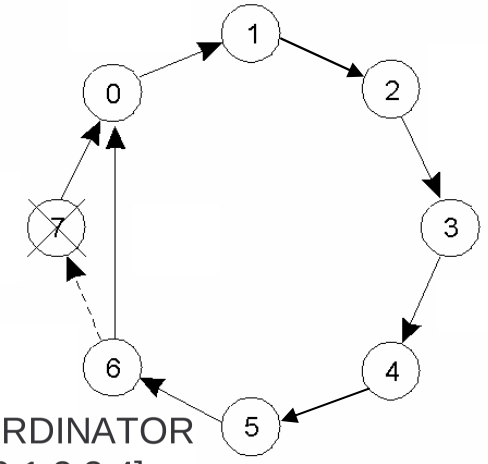


Ein Ring-Algorithmus

- Für diesen Algorithmus sind die Prozesse logisch in Form eines Rings organisiert, d.h., jeder Prozess besitzt einen Vorgänger und einen Nachfolger entsprechend aufsteigender Prozessnummern.
- Wenn ein Prozess feststellt, dass der Koordinator nicht mehr funktioniert, sendet er eine ELECTION-Nachricht auf den Ring, in die er sich als ersten einträgt.
- Jeder weitere aktive Prozess fügt sich selbst in die Liste ein.
- Wenn die Nachricht wieder beim Initiator eintrifft, wandelt er diese in eine KOORDINATOR-Nachricht um, die den neuen Koordinator und die aktiven Mitglieder enthält.

Beispiel: Ring-Algorithmus





KOORDINATOR
[5,6,0,1,2,3,4]

Gegenseitiger Ausschluss

- Wenn sich zwei oder mehrere Prozesse beim Zugriff auf gemeinsame Daten koordinieren müssen, um die Konsistenz der Daten zu erhalten, geschieht dies am einfachsten über das Konzept der *kritischen Region*.
- Jeweils nur ein Prozess darf in einer kritischen Region aktiv sein, d.h., es wird gegenseitiger Ausschluss (mutual exclusion) erreicht.
- In Ein-Prozessor-Systemen werden Semaphore oder Monitore verwendet, um gegenseitigen Ausschluss zu implementieren (s. Vorlesung Betriebssysteme und Netze).

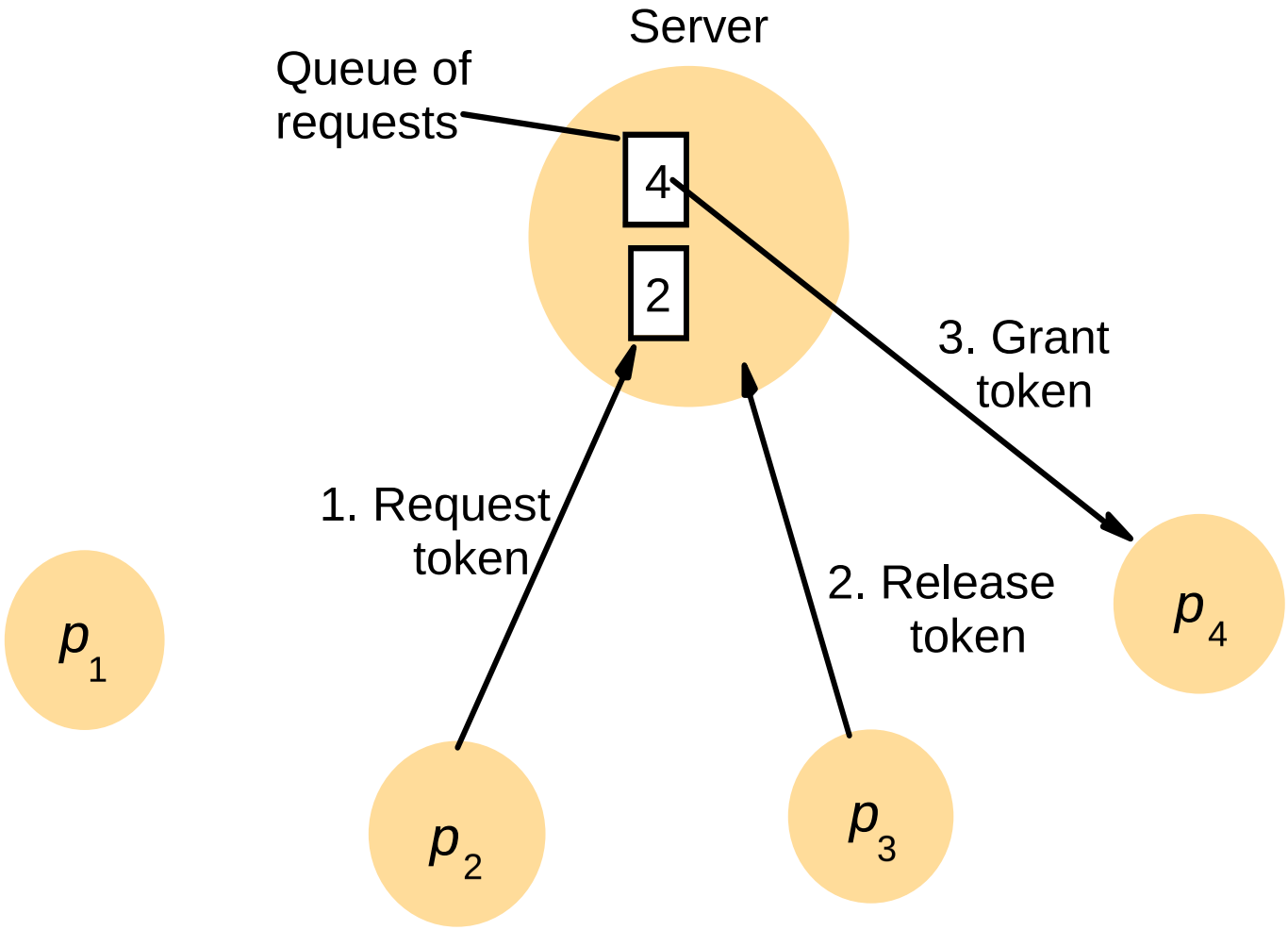
Mutual Exclusion in verteilten Systemen

- Wie kann man Mutual Exclusion in verteilten Systemen umsetzen?
- Wir betrachten drei Algorithmen:
  - Ein zentralisierter Algorithmus
  - Ein verteilter Algorithmus
  - Ein Token-Ring-Algorithmus

Ein zentralisierter Algorithmus

- Einer der Prozesse wird zum Koordinator für eine kritische Region bestimmt.
- 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.

Beispiel: Mutual-Exclusion-Server



Eigenschaften des Algorithmus

- Mutual Exclusion wird erreicht: es ist immer nur ein Prozess im kritischen Bereich, da der Server immer nur ein Token vergibt
- Fair: Tokens werden in der Reihenfolge der Anfrage vergeben
- Einfach zu implementieren
- Nur 3 Nachrichten pro Zugang zur kritischen Region
- Koordinator ist single point of failure, d.h., Problem, wenn der Koordinator zusammenbricht
- Prozesse, die nach einem Request blockieren, können nicht zwischen einem „toten“ Koordinator und einer langen Warteschlange unterscheiden.
- Performance Bottleneck in großen Systemen

Ein verteilter Algorithmus

- Der folgende Algorithmus besitzt keinen ausgewiesenen Koordinator.
- Alle Prozesse verständigen sich über Multicast-Nachrichten.
- Jeder Prozess besitzt eine logische Uhr.
- Wenn ein Prozess eine kritische Region betreten will, sendet er ein Request an alle anderen Prozesse.
- Erst wenn alle Prozesse ihr OK gegeben haben, kann der Prozess die kritische Region betreten.

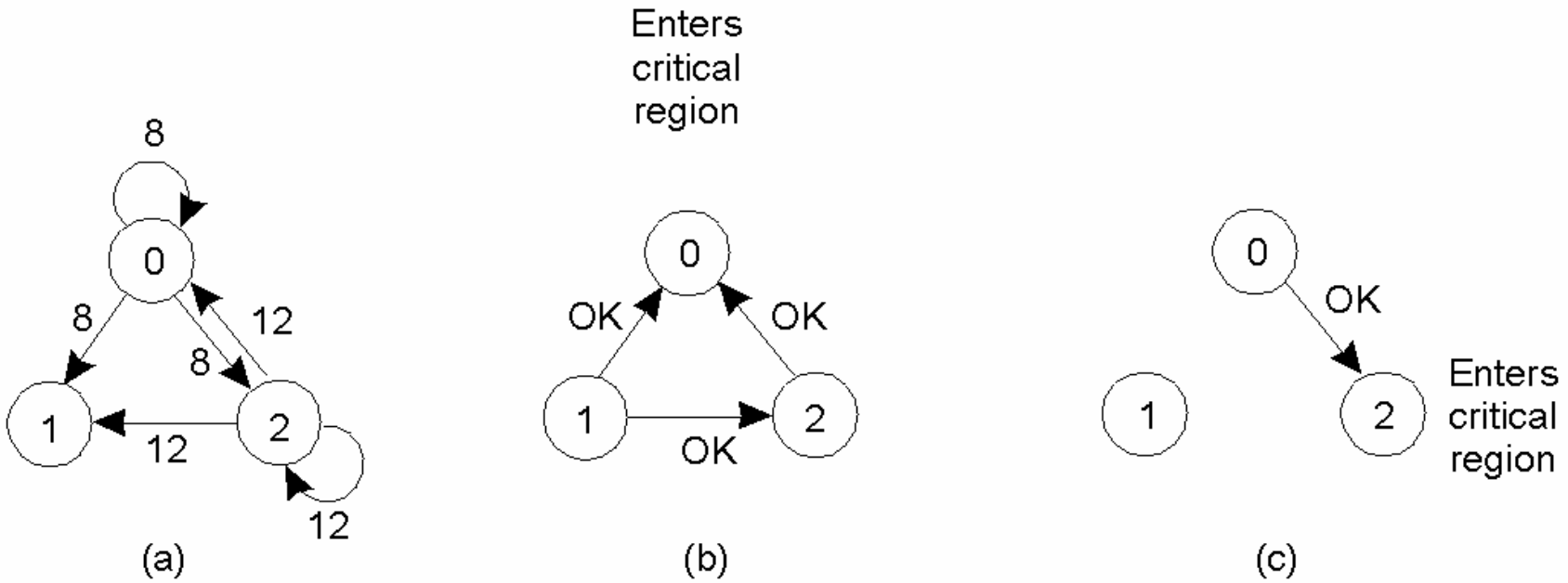
Der Algorithmus (Ricart and Agrawala, 1981)

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Beispiel



a) Zwei Prozesse wollen gleichzeitig die kritische Region betreten.

b) Prozess 0 hat den niedrigeren Timestamp und gewinnt.

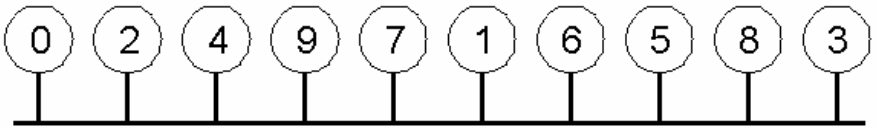
c) Wenn Prozess 0 fertig ist, gibt er die kritische Region frei und sendet ebenfalls ein OK an 2.

Eigenschaften des Algorithmus

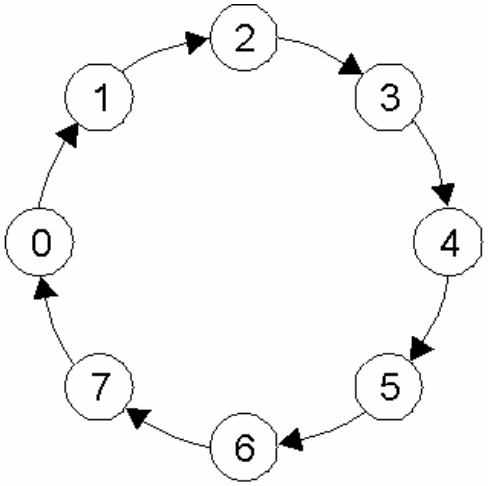
- Der single-point-of-failure wurde ersetzt durch n points-of-failure. Wenn ein Prozess nicht mehr arbeitet, funktioniert das System nicht mehr.
- Dieses Problem könnte durch explizite Verwendung eines Request-Reply-Protokolls ersetzt werden (jede Nachricht wird sofort bestätigt). Wenn keine Bestätigung kommt, ist der Prozess nicht mehr aktiv.
- Jeder Prozess muss immer bei der Entscheidung mitwirken, obwohl er evtl. gar kein Interesse an der kritischen Region hat.
- Verbesserung: eine einfache Mehrheit genügt
- Insgesamt ist der Algorithmus langsamer, komplizierter, teurer und weniger robust, aber, wie 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.

Ein Token-Ring-Algorithmus

- Die Prozesse in einem lokalen Netz werden in einer logischen Ringstruktur entsprechend der Prozessnummern organisiert.
- Ein Token kreist; wer das Token besitzt, darf in den kritischen Bereich, allerdings nur einmal.



(a)



(b)

Eigenschaften des Algorithmus

- Korrektheit ist ebenfalls leicht zu sehen. Nur ein Prozess hat das Token zur selben Zeit.
- Kein Prozess wird ausgehungert, da die Reihenfolge durch den Ring bestimmt ist.
- 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.
- Tote Prozesse müssen erkannt werden.

Vergleich

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Literatur

- Leslie Lamport: *Time, Clocks, and the Ordering of Events in a Distributed System',* Communications of the ACM. Juli 1978.
- Jeff McGee und Jeff Kramer: *Concurrency: State Models and Java Programs*, Wiley, 1999.