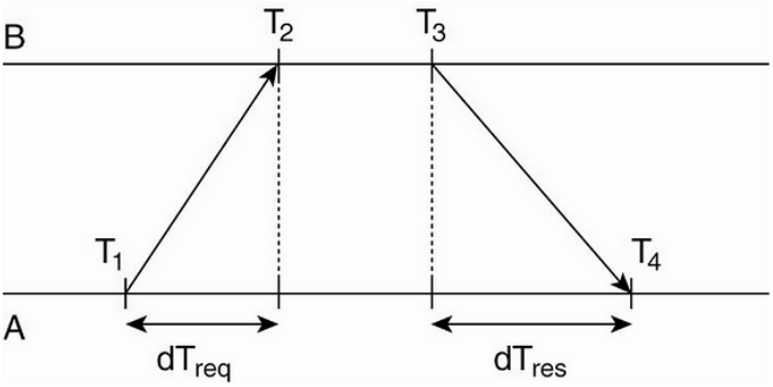
Ein Ansatz um Mobility zu erreichen ist die Verwendung von Forwarding Pointers. Wenn eine Entität seinen Ort ändert, lässt sie eine Referenz zurück, welche auf ihren neuen Ort zeigt.
Vorteil: sehr einfach
Nachteil: Lange Ketten, möglicherweise viele Zwischenschritte notwendig, eine falsche Referenz unterbricht gesamte Kommunikation (broken link), daher müssen die Ketten so kurz wie möglich gehalten werden.

Ein Ansatz um Mobility in großen Netzwerken zu erreichen ist die Verwendung von Homebased Approaches. Der Home-Agent speichert dabei immer den aktuellen Standort einer Entität. Ändert das mobile Gerät (Entität) ihren Standort, registriert sie eine Care-Of-Address beim Home-Agent. Ein Client, der den aktuellen Standort nicht kennt, kann ihn also immer beim Home Agent erfragen und anschließend direkt kommunizieren.

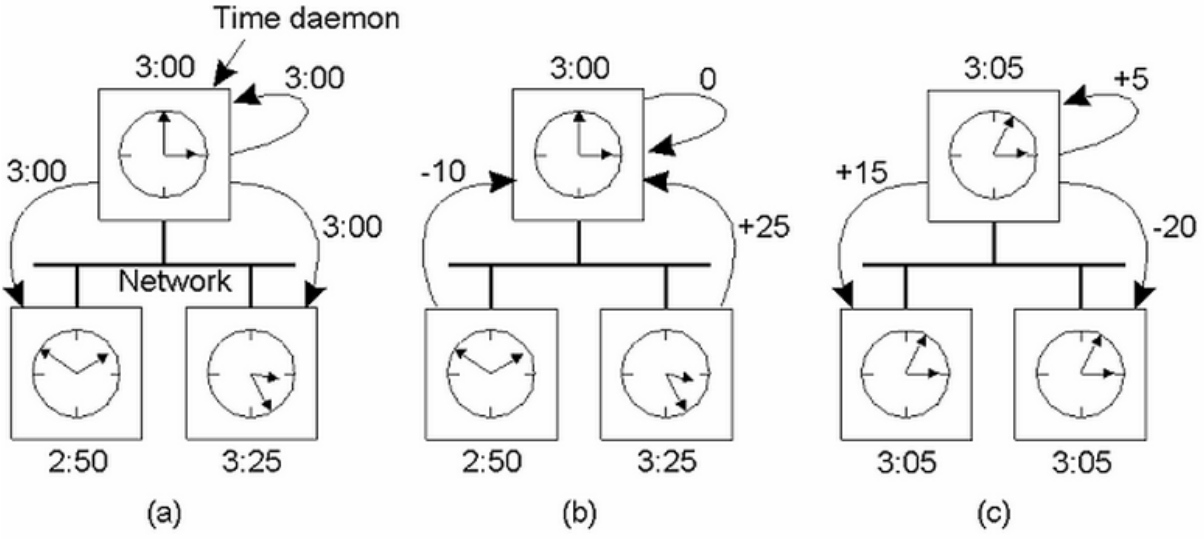
33. Wozu braucht man Uhrensynchronisation? Erläutern Sie das NTP und den Berkeley Algorithmus. Was ist die Problematik bei der Synchronisation von Physical Clocks?

NTP (Network Timing Protocol):
Es handelt sich dabei um eine Synchronisation mit einem **externen Zeitgeber**. Die Genauigkeit von Zeitgebern wird durch das sogenannte Stratum definiert. Je niedriger es ist, desto genauer ist eine Zeitquelle. Beim NTP wird ein höheres Stratum mit einem niedrigeren synchronisiert. Ein Rechner sendet eine Nachricht an den Zeitserver und merkt sich den Sendezeitpunkt t1. Der Server zeichnet sich den Emfangszeitpunkt t2 sowie den Sendezeitpunkt der Antwort (kurz davor) t3 und schickt die Antwort. Der Rechner zeichnet den Empfangszeitpunkt t4 auf. (t1 und t4: lokale Zeit; t2 und t3: Serverzeit) Berechnung der einfachen Übertragung: ((t4 - t1) - (t3 - t2)) / 2. Somit kann durch t4 - t1 einfaches Delay darauf geschlossen werden, um wie viel die Zeit des Senders nach- oder vorgeht. Korrektur: beschleunigen oder bremsen der lokalen Uhr. Dieser Vorgang wird 8-mal wiederholt und das Ergebnis mit dem kürzesten Delay verwendet. Delay über einem Grenzwert --> Zeitquelle nicht vertrauenswürdig.



BERKELEY ALGORITHMUS

Dabei handelt es sich um eine interne Synchronisation. Es gib einen Koordinator (Time daemon), der seine lokale Zeit an alle sendet (auch an sich selbst). Jeder antwortet mit seiner Abweichung von dieser Referenzzeit. Danach berechnet der Koordinator die durchschnittliche Abweichung und sendet sie jedem zurück. Diese beschleunigen oder verlangsamen dann ihre Zeit entsprechend der Korrektur



Was ist die Problematik bei der Synchronisation von Physical Clocks?

- Die Uhr wird dann langsam vorgestellt, jedoch NIEMALS zurückgestellt
  - ➔ zurückstellen ist problematisch wegen Zeitstempel. Zeitstempel müssen immerlinear vorwärts gehen.
- Antwortzeiten über Netzwerk unterschiedlich

34. Was sind die Gründe für die Verwendung von Logical Clocks? Erklären Sie die Unterschiede zu den Physical Clocks. Was ist die "happened-before" Beziehung und wie funktionieren die "Lamport-Timestamps"?

Verwendung von Logical Clocks: Es ist meist ausreichend, wenn alle dieselbe Zeit haben, auch wenn diese nicht mit der realen Zeit zusammenpasst. Es ist oft nur die Reihenfolge der Operationen wichtig.

Unterschiede: Die physikale Uhr ist die "reale" Uhr (Kristall-Oszillator, mittlere Sonnensekunde, Atomuhr (TAI, International Atomic Time), UTC (Universal Coordinated Time), wobei eine Logische Uhr eine Reihung der Ereignisse darstellt.

"happend-before"-Beziehung: a -> b wird gelesen als "a passiert vor b". Dies bedeutet, dass alle Prozesse wissen, dass a vor b eingetreten ist. Zwei Situationen:

- wenn a und b im selben Prozess sind, und a tritt vor b ein dann ist "a->b" = true
- wenn a das Ereignis darstellt, dass eine Nachricht von einem Prozess gesendet wird, und b ist das Ereignis, dass die Nachrricht von einem anderen Prozess empfangen wird. Eine Nachricht kann nicht empfangen werden, bevor sie gesendet wird (und auch nicht gleichzeitig).

Dies ist eine transitive Relation: a -> b und b -> c, dann a -> c

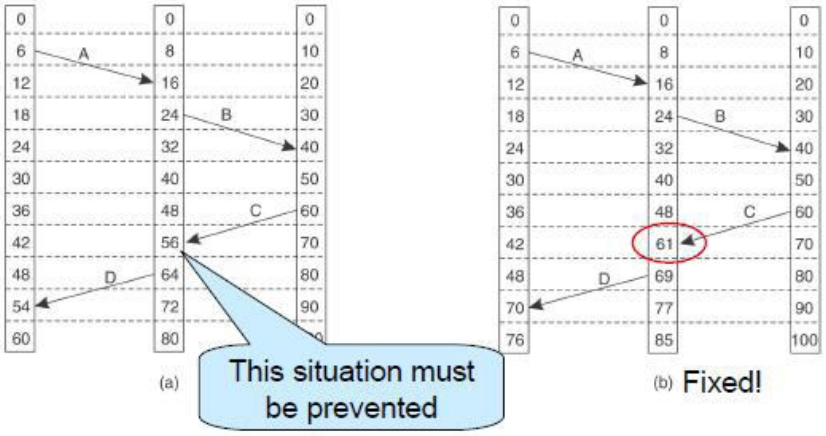
Lamport-Timestamps: Ereignisse werden nur durchnummeriert. Potentiell kausal abhängige Ereignisse (im Sinne der "happend-before"-Beziehung) bekommen dabei Nummern so zugewiesen, dass das abhängige Ereignis eine größere Nummer hat als die Ursache, und dass diese Nummern bei allen Prozessen gleich sind. Jeder Prozess hat seinen eigenen Counter (logische Uhr). Bei jedem Ereignis (Berechnung, Senden, Empfangen) wird er um 1 erhöht und hängt diese Zahl der Nachricht an. Empfänger setzt seinen Counter auf das Maximum aus seinem aktuellen Counter und der mitgeschickten Zahl, anschließend erhöht er seinen Counter um 1. Es muss immer gelten:

- T(b) > T(a), falls a und b in dieser Reihenfolge im gleichen Prozess stattfinden

Eigenschaften:

- 2 aufeinanderfolgende Ereignisse a und b im gleichen Prozess: T(a) < T(b)
- Der Empfang einer Nachricht hat immer höhere Zahl als das Senden
- Die Relation ist transitiv abgeschlossen

Es kann jedoch nicht auf kausale Abhängigkeiten geschlossen werden, d.h. T(a) < T(b) bedeutet nicht unbedingt dass a->b gilt (nur in die andere Richtung).



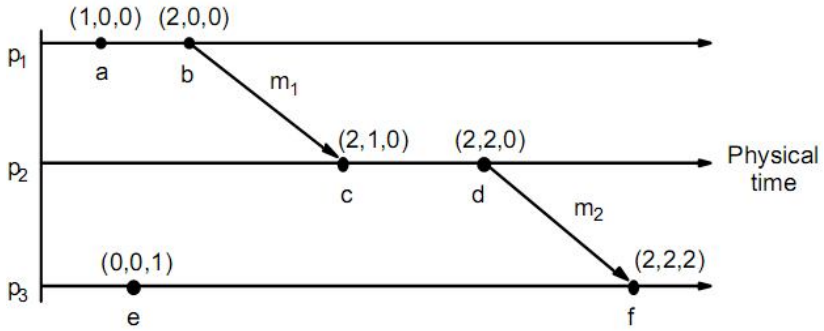
35. Welchen Nachteil haben die Lamport-Timestamps und wie kann dieser durch Vector-Timestamps überwunden werden?

Nachteil: keine kausalen Abhängigkeiten. Nur weil T(a) < T(b) gilt sagt das noch nicht aus, dass a auch vor b stattgefunden hat. Es kann sich ja um Zeiten von 2 oder mehreren verschiedenen unabhängigen Prozessen handeln.

Vector-Timestamp:
Ein Vector Timestamp (VT oder Vector Clock VC) hat die Eigenschaft, dass wenn VC(a) dem Event a zugeordnet ist, dann hat dieser Vektor die Eigenschaft dass wenn VC(a) < VC(b) für ein Event b ist, a kausal vor b ausgeführt worden ist.
Jeder von n Prozessen hat einen Vektor von der Länge n (initialisiert mit dem Null-Vektor). Bei jedem Ereignis in Prozess i zählt dieser Element i um 1 hoch. Der Vektor wird an die gesendete Nachricht angehängt. Beim Empfang durch Prozess j wird das Weitergeben der Nachricht an die Applikation solange verzögert bis gilt (m ist die Message; m[i] beschreibt den Counter des Prozesses i im Vector der Message):

- m[i] = Vj[i] + 1
- m[k] <= Vj[k] für alle k außer i

Potentiell kausal abhängige Nachrichten können durch elementweisen Vergleich der Vektoren gefunden werden. Es handelt sich dabei um eine partielle Ordnung der Ereignisse.



36. Wie funktioniert Distributed Mutual Exclusion. Wie verhalten sich verschiedene Algorithmen (centralized, distributed, token-ring) hinsichtlich Skalierbarkeit und Fehlertoleranz?

Das Ziel von Distributed Mutual Exclusion ist, in einem Verteilten System den exklusiven Zugriff auf Ressourcen zu vergeben, es soll verhindert werden, dass 2 Prozesse gleichzeitig auf dieselbe Ressource zugreifen. Dazu werden mehrere Ansätze verfolgt:

- **token ring**: Ein Token wird zwischen den Prozessen weitergereicht, nur der Prozess, der das Token hält, ist zum Zugriff auf die Ressource berechtigt. Benötigt ein Prozess keinen Zugriff, so reicht er einfach das Token zum nächsten Prozess weiter. (Vorteile): Einfache Implementierung; keine Starvation (jeder Prozess kommt an die Reihe); keine Deadlocks (Nachteile): stürzt der Prozess ab, der das Token hält, geht es verloren, es ist schwer zu erkennen wann ein Token verloren gegangen ist; auch wenn kein Prozess auf die Ressource zugreifen will, muss trotzdem ständig das Token weitergereicht werden.
- **centralized**: ein Prozess agiert als Koordinator, er erhält die Anfragen der anderen Prozesse und gewährt oder verwehrt den Zugriff auf die Ressource, sodass immer nur ein Prozess zur selben Zeit mit der Ressource arbeitet. Ist dieser Prozess mit seiner Arbeit fertig, teilt er dies dem Koordinator mit. Falls ein Prozess eine Anfrage stellt und die Ressource in Verwendung ist, kann der Koordinator die Anfrage in einer Warteschlange zwischenspeichern. (Vorteile):hohe Effizienz; Einfache Implementierung; keine Starvation (jeder Prozess kommt an die Reihe); keine Deadlocks (Nachteile): Der Koordinator ist ein Single-point of failure und ein bottleneck
- **distributed**: Wenn ein Prozess exklusiven Zugriff auf eine Ressource haben will, so schickt er eine Nachricht an alle anderen Prozesse (einschließlich sich selbst). Dieser Nachricht hängt er seine logische Zeit (Lamport clock) an. Der Empfänger dieser Nachricht hat 3 Antwortmöglichkeiten:
    - Greift er selbst auf die Ressource zu, so antwortet er mit denied
    - Greift er nicht darauf zu, und hat es auch nicht vor, so ist die Antwort OK
    - Greift er noch nicht darauf zu, will es aber, so werden die logischen Zeiten der Nachrichten (seine eigene + die Anfrage des anderen Prozesses) verglichen, die niedrigere gewinnt, je nachdem antwortet der Prozess mit denied oder OK

  Wird von jedem der n Prozesse auf ein OK gewartet, so gibt es nicht mehr einen single-point of failure, sondern n (sobald einer der n Prozesse gecrasht ist funktioniert dieses Prinzip nicht mehr); Bei dieser Variante kommt es zu einem sehr hohen Kommunikationsaufwand; Dieser Algorithmus beseitigt keine Bottlenecks, da jeder Prozess befragt wird und jeder dieselbe Arbeit erledigen muss -> keine Aufteilung der Arbeit.

37. Sie sollen einen Dateiserver implementieren der mehrere Clients gleichzeitig bedienen kann. Nennen und beschreiben Sie einen (konkreten) Mechanismus der garantiert dass immer nur einer der Clients gleichzeitig eine Datei schreiben darf. Die anderen Clients sollen solange blockiert werden (egal ob schreibend oder lesend) bis der schreibende Client seine Arbeit beendet hat. Warum ist das sinnvoll?

CENTRALIZED Algorithmus

- Ein zentraler Koordinator nimmt die READ und WRITE Anforderungen der Clients entgegen.
- READ dürfen parallel von mehreren Clients durchgeführt werden.
- WRITE darf nur ein einzelner CLIENT durchführen.
  -> sobald WRITE freigegeben wird, darf kein READ mehr durchgeführt werden. Alle READ-Requests kommen in eine Warteschlange. Sobald WRITE-Request fertig gestellt ist, werden weitere WRITE-Anfragen in der Warteschlange bearbeitet, ansonsten werden die anderen READ-Requests bearbeitet.
- WRITE –Requests kommen ebenso in eine Warteschlange.

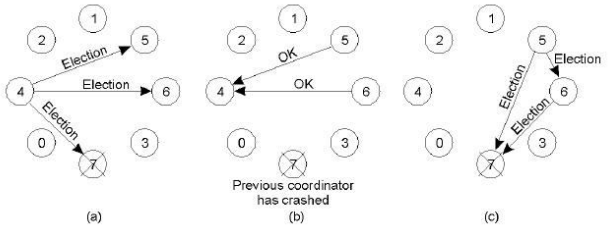
Sinnvoll:

- Wenn ein Schreibvorgang noch nicht beendet ist und ein READ ausgeführt wird, würde der Client eine ungültige Datei erhalten
- Ein Schreibvorgang darf nicht in einen anderen Schreibvorgang hineinschreiben → würden sich gegenseitig die Daten zerstören

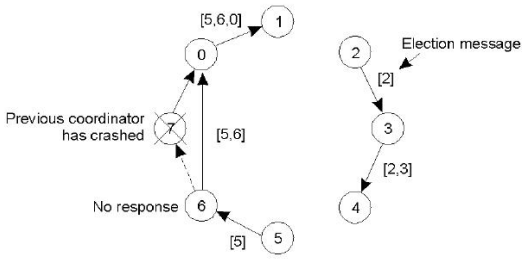
38. Erläutern Sie den "Bully" und den "Ring"-Algorithmus für Election und vergleichen sie die beiden hinsichtlich Fehlertoleranz. Warum sind diese Algorithmen für ad-hoc oder large-scale Systeme weniger geeignet und welche grundsätzlichen Lösungsansätze verfolgt man daher dort?

Sowohl bully, als auch der ring election Algorithmus basieren auf der Idee, dass jeder Prozess eine eindeutige ID besitzt (z.B.: ProzessID), und der Prozess mit der höchsten ID als Koordinator fungiert.

- **bully algorithmus**: Sobald ein Prozess bemerkt, dass der Koordinator ausgefallen ist, sendet er eine ELECTION Nachricht an alle Prozesse mit einer ID höher als seiner eigenen ID. Wenn er keine Antwort erhält, so ist er der neue Koordinator (schickt COORDINATOR Nachricht an die anderen Prozesse), wenn jedoch einer der Prozesse antwortet, so übernimmt dieser die Kontrolle, wer neuer Koordinator wird (in diesem Fall schickt der Prozess wieder eine ELECTION Nachricht, die geschieht so lange, bis der Prozess mit der höchsten ID Koordinator geworden ist und eine COORDINATOR Nachricht an die anderen Prozesse sendet).



- **ring algorithmus**: Die Prozesse sind logisch in Form eines Ringes angeordnet. Der Prozess, der den Ausfall des Koordinators bemerkt, sendet eine ELECTION Nachricht an seinen Nachfolger (wenn dieser nicht antwortet an dessen Nachfolger, so lange bis einer antwortet). Jeder Prozess hängt seine ID an die Nachricht an. Wenn die Nachricht den Ring einmal durchlaufen hat, und beim Initiator angelangt ist, sendet dieser eine COORDINATOR Nachricht aus, da ihm nun der Prozess mit der höchsten ID bekannt ist. Der neue Koordinator kann seine Arbeit aufnehmen, nachdem die COORDINATOR Nachricht wieder beim Initiator angelangt ist, wird sie entfernt. Falls mehrere Prozesse gleichzeitig den Ausfall des Koordinators bemerken und eine ELECTION Nachricht aussenden, ändert es nichts am Ergebnis, es wird lediglich ein wenig zusätzlicher Traffic erzeugt.



Beide Algorithmen sind sehr Fehlertolerant, da immer wieder ein neuer Koordinator gewählt werden kann. Es besteht KEIN Single-Point-Failure Problem.

Bei **ad-hoc-Netzwerken** kann sich die Topologie schnell ändern und man kann nicht davon ausgehen dass die Kommunikation zuverlässig (reliable) ist. Der Bully- und Ring-Algorithmus sind daher für derartige Netzwerke nicht geeignet. Außerdem ist es durch die heterogene Struktur wichtig dass nicht irgendein Koordinator gewählt wird, sondern der beste (anhand eines Kriteriums, z.B. Bandbreite).
Lösung: Bei der Auswahl sendet ein Knoten die ELECTION Nachricht an all seine Nachbarn. Empfängt ein Nachbar diese Nachricht setzt er den Sender als Parent und leitet die ELECTION Nachricht an alle Nachbarn außer den Parent weiter. Dadurch entsteht eine Baumstruktur. Jeder Parent wartet nun nach dem Weiterleiten auf die Response aller seiner Kinder. Die Response enthält die ID und das Score (Bewertung) des besten Prozesses im Subtree. Hat ein Parent von allen Kindern eine Response erhalten, kann er wiederum eine Response an seinen (mit dem besten Prozess) Parent senden. Der Root-Prozess weiß daher welcher Prozess der geeignetste Leader ist.

Bei **large-scale-Netzwerken** werden Superpeers ausgewählt, da ein Koordinator zu wenig sein wird. Auswahl mehrerer ausgezeichneter Knoten, die sich jeweils um eine Anzahl anderer Knoten kümmern. Dabei müssen Superpeers folgende Kriterien erfüllen:

1. Superpeers sollten geringe Latenzzeiten gegenüber normalen Peers haben
2. Superpeers sollten im Netzwerk gleichmäßig sein.

3. Die Anzahl der Superpeers sollte eine definierte Anzahl relativ zur Anzahl normaler Knoten nicht unterschreiten
4. Jeder Superpeer sollte nur eine gewisse Anzahl an Knoten bedienen

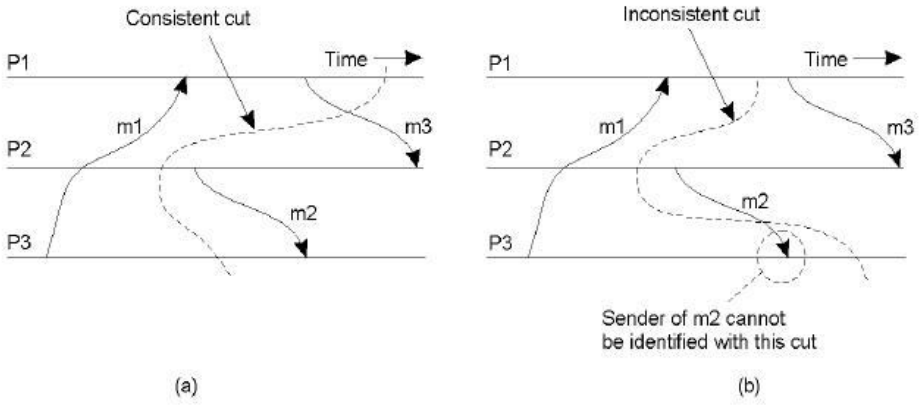
Lösung: Es werden für n Superpeers n Token zufällig im Netz verteilt. Kein Knoten kann mehr als einen Token gleichzeitig haben. Außerdem wirken auf die Token so genannte Kräfte die sicherstellen sollen, dass sich Token voneinander abstoßen (vgl. Teilchen mit gleicher Ladung). Erfährt ein Knoten von ein oder mehreren Token in der Umgebung kann aufgrund der Richtung, in der die anderen Token liegen jene Richtung bestimmt werden in die sein Token weitergegeben wird. Hält ein Knoten einen Token für eine gewisse Zeit (Token haben sich stabilisiert) wird dieser zu einem Superpeer. Im englischen Buch S. 270 wird darauf hingewiesen, dass diese mysteriöse Kraft mit einem Gossip-Protokoll realisiert werden kann.

39. Welche Probleme gibt es bei der Ermittlung des "Global State" und wie können diese überwunden werden? Geben Sie zumindest einen Algorithmus an.

Global State

Der Globale State eines verteilten Systems besteht aus
- dem lokalen Zustand eines jeden Prozesses
- zusammen mit den Nachrichten die gerade unterwegs sind (gesendet aber noch nicht empfangen)

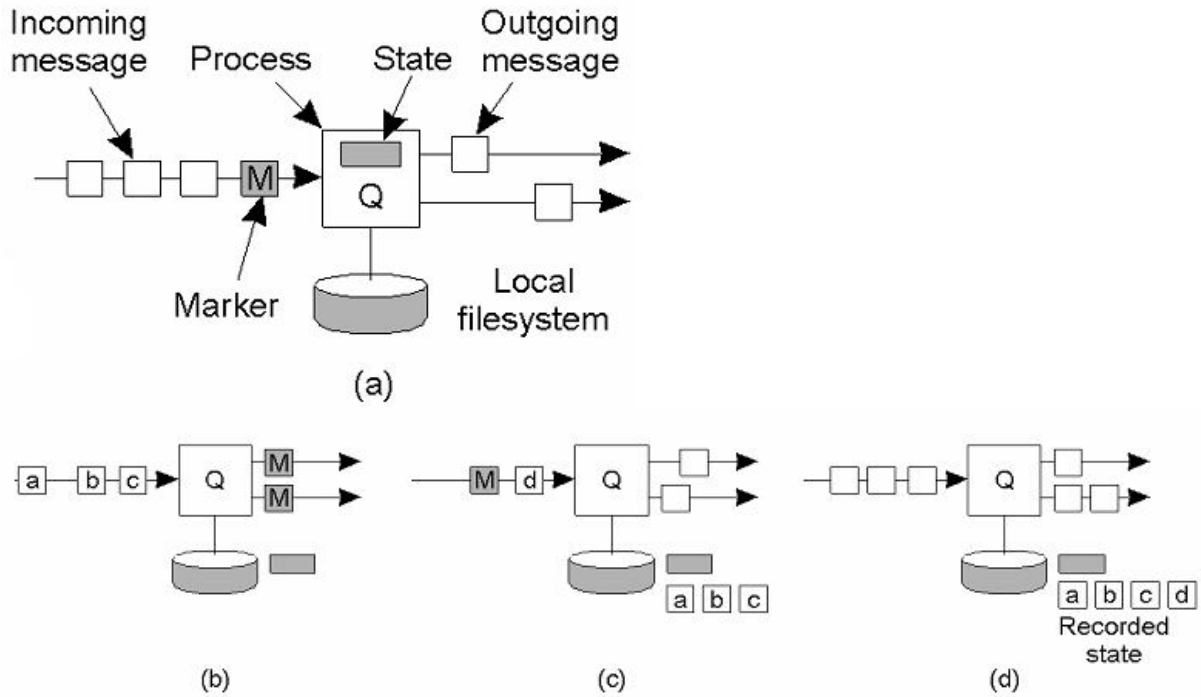
Probleme:
- Es könnte ein inkonsistener "Schnitt" durch das System gemacht werden. (Ergebnisse ohne Ursache. z.B.: Empfangen einer Nachricht ohne Senden dieser Nachricht)



a) A consistent cut
b) An inconsistent cut (effect without cause)

- Keiner hat eine globale Sicht auf das System.
- Es gibt keine gemeinsame Zeit für die Aufzeichnung ("Stichtag")

Lösung der Probleme durch Ermittlung des Globale State durch Chandy und Lamport-Algorithmus (Snapshot vom Global State).



Ein Initiator beginnt seinen eigenen Status aufzuzeichnen und schickt einen Marker aus. Beim Empfang des 1. Markers zeichnet jeder Rechner seinen lokalen Status auf und schickt den Marker weiter (b). Bis zum Empfang des Markers zum zweiten Mal zeichnet jeder Prozess alle eingehenden Nachrichten auf (c). Beim Empfang des Markers zum zweiten Mal ist die Aufzeichnung abgeschlossen (d): der lokale Status und die aufgezeichneten Nachrichten können an den ursprünglichen Initiator gesendet werden, wo dann die Auswertung passiert. Der aufgezeichnete Status ist garantiert konsistent, kann aber Kombinationen von lokalen Zuständen enthalten, die so nie aufgetreten sind.

40. Was sind die Hauptgründe für den Einsatz von Replikation in verteilten Systemen? In welcher Beziehung stehen Replikation und Skalierbarkeit zueinander? Erläutern Sie in diesem Zusammenhang verschiedene Varianten der Content Replication und des Content Placement.
Bei Replikation führt man mehrere Kopien einer Entität auf verschiedenen Knoten.

- **Zuverlässigkeit**
  - Umschalten im Fehlerfall → **Fehlertoleranz**
  - Schutz gegen beschädigten Daten (corrupted data)
- **Leistung durch Skalierbarkeit**
  - **Größe**: Die Skalierung nach der Größe tritt beispielsweise auf, wenn immer mehr Prozesse auf Daten zugreifen müssen, die von einem einzigen Server verwaltet werden. In diesem Fall kann die Leistung verbessert werden, indem man diesen Server repliziert und damit die Arbeit aufteilt.
  - **geographischem/topologischem Kontext**: Die Skalierung in Hinblick auf die Größe eines geografischen Bereichs kann ebenfalls eine Replikation erforderlich machen. Dabei wird eine Kopie der Daten in die Nähe des Prozesses platziert, der sie nutzt, und somit die Datenzugriffszeit reduziert.

Skalierbarkeit und Replikation stehen insofern in der Beziehung, dass die Replikation einen deutlichen Mehraufwand bedeutet, da die Daten konsistent gehalten werden müssen. Man muss also abschätzen ob das Einsetzen von Replikaten den zusätzlichen Aufwand (Traffic, Ressourcen) rechtfertigt. (Medizin schlimmer als Krankheit → Kompromiss)