Clock Synchronization

- Physical clocks
- Logical clocks
- Vector clocks

Physical Clocks

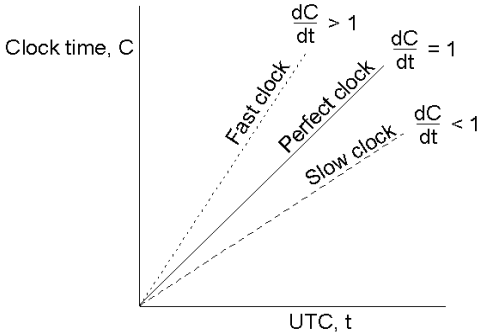
Problem: Suppose we have a distributed system with a UTC-receiver somewhere in it we still have to distribute its time to each machine.

UTC is Universal Coordinated Time, based on some atomic element (Cs)

Basic principle:

- Every machine has a timer that generates an interrupt *H* times per second.
- There is a clock in machine *p* that **ticks** on each timer interrupt. Denote the value of that clock by *Cp{t}*, where *t* is UTC time.
- Ideally, we have that for each machine *p*, *Cp{t}* = *t*, or, in other words, *dC/dt* =1

Clock Synchronization Algorithms



In practice: *1 – ρ <= dC/dt <= 1 + ρ,* for some small constant drift *ρ*

Goal: Never let two clocks in any system differ by more than δ time units ➔ synchronize at least every δ/(2ρ) seconds

Clock Synchronization

Idea 1: Every machine asks a **time server** for the accurate time at least once every d/(2r) seconds.

Good solution, but

- need an accurate measure of round trip delay
- including interrupt handling and processing incoming messages.

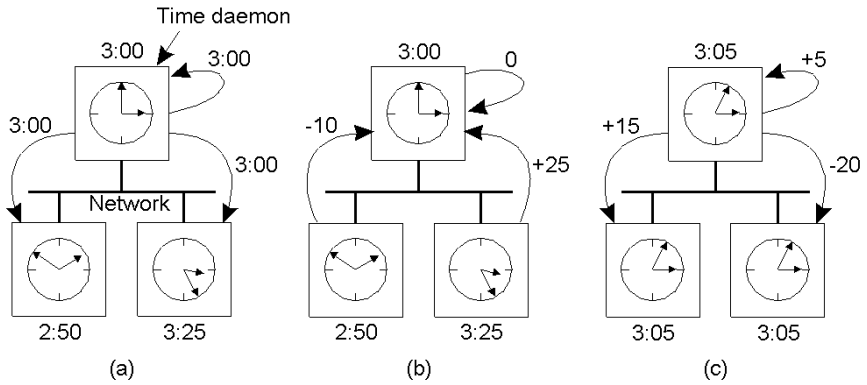
Idea 2: Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Another good solution, you'll probably get every machine in sync.

Fundamental problem: You'll have to take into account that setting the time back is **never** allowed ➔ smooth adjustments.

Note: you don't even need to propagate UTC time. Why not?

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

The Happened-Before Relationship

Problem: We first need to introduce a notion of *ordering* before we can order anything.

The **happened-before** relation on the set of events in a distributed system is the smallest relation satisfying:

- If *a* and *b* are two events in the same process, and *a* comes before *b*, then *a* → *b*.
- If *a* is the sending of a message, and *b* is the receipt of that message, then *a* → *b*.
- *If a* → *b and b* → *c, then a* → *c.*

- Is this a partial or total ordering of events in a system with concurrently operating processes?

Logical Clocks

Problem: How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Solution: attach a timestamp *C*(*e)* to each event *e*, satisfying the following properties:

- **P1:** If *a* and *b* are two events in the same process, and $a \rightarrow b$, then we demand that *C(a) < C(b).*
- **P2:** If *a* corresponds to sending a message *m*, and *b* corresponds to receiving that message, then also *C(a) < C(b).*

Problem: How to attach a timestamp to an event when there's no global clock ➔ maintain a **consistent** set of logical clocks, one per process.

Logical Clocks

Each process P_i maintains a **local** counter C_i and adjusts this counter according to the following rules:

1. For any two successive events that take place within P_i , C_i is incremented by 1.
2. Each time a message *m* is sent by process P_i, the message receives a timestamp $T_m = C_i$
3. Whenever a message *m* is received by a process P_j, P_j, adjusts its local counter C_j:

$$C_j <= max\{C_j + 1, T_m + 1\}.$$

Property **P1** is satisfied by (1); Property **P2** by (2) and (3).

Extension to Multicasting: Vector Timestamps

Observation: Lamport timestamps do not guarantee that if $C(a) < C(b)$ then a indeed happened before b. Why?

We need **vector timestamps** for that.

- Each process Pi has an array $V_i[1..n]$, where $V_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- When P_i sends a message m, it adds 1 to $V_i[i]$, and sends V_i along with m as vector timestamp $vt(m)$.
  Result: upon arrival, each other process knows P_i's timestamp.

Question: What does $V_i[j] = k$ mean in terms of messages sent and received?

Extension to Multicasting: Vector Timestamps

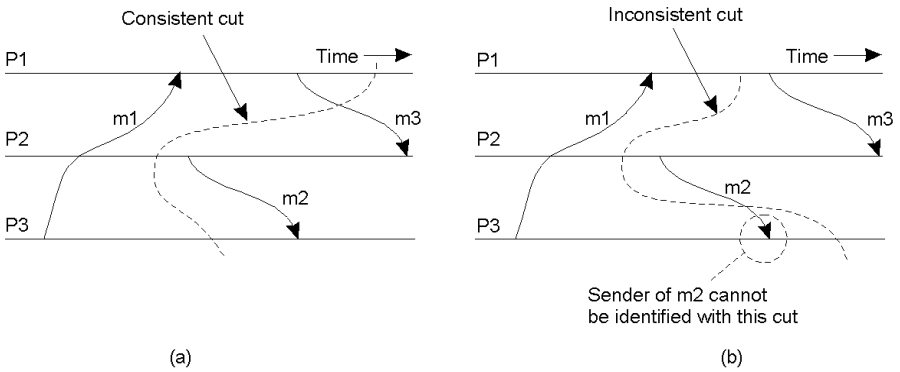
- When a process P_j receives a message m from P_i with vector timestamp $vt(m)$, it (1) updates each $V_j[k]$ to max $\{V_j[k], V(m)[k]\}$, and (2) increments $V_j[j]$ by 1
- Is the book correct?
- To support causal delivery of messages, assume you increment your own component only when sending a message. Then, P_j postpones delivery of m until:
  - $V_t(m)[i] = V_j[i] + 1$
  - $V_t(m)[k] <= V_j[k]$ for $k != i$

  Example: Take $V3 = [0,2,2]$, $v_t(m) = [1,3,0]$. What information does P_3 have, and what will it do when receiving m (from P_1)?

Global State (1)

Basic Idea: Sometimes you want to collect the current state of a distributed computation, called a **distributed snapshot**. It consists of all local states and messages in transit.

Important: A distributed snapshot should reflect a **consistent** state:

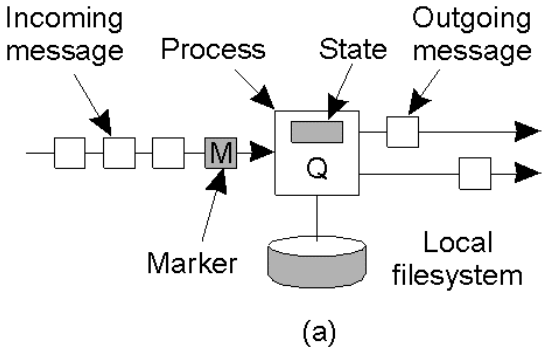


Global State

Note: any process P can initiate taking a distributed snapshot

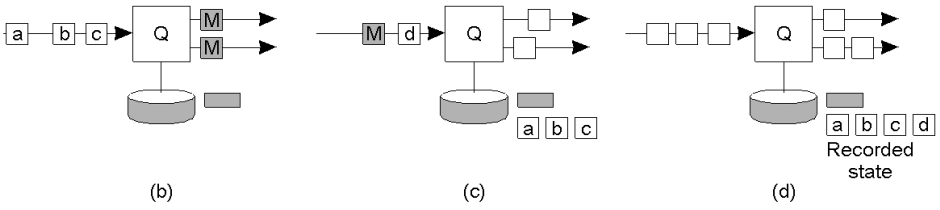
- P starts by recording its own local state
- P subsequently sends a marker along each of its outgoing channels
- When Q recieves a marker through channel C, its action depends on whether it has already recorded its local state:
  - Not yet recorded: it records its local state, and sends the marker along each of its outgoing channels
  - Already recorded: the marker on C indicates that the channel's state should be recorded: all messages received before this marker and the time Q recorded its own state.
- Q is finished when it has received a marker along each of its incoming channels.

Global State (2)



a) Organization of a process and channels for a distributed snapshot

Global State (3)



b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel

Election Algorithms

Principle: An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.

Note: In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solution ➔ single point of failure.

Question: If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

Question: Is a fully distributed solution, i.e., one without a coordinator, always more robust than any centralized/coordinated solution?

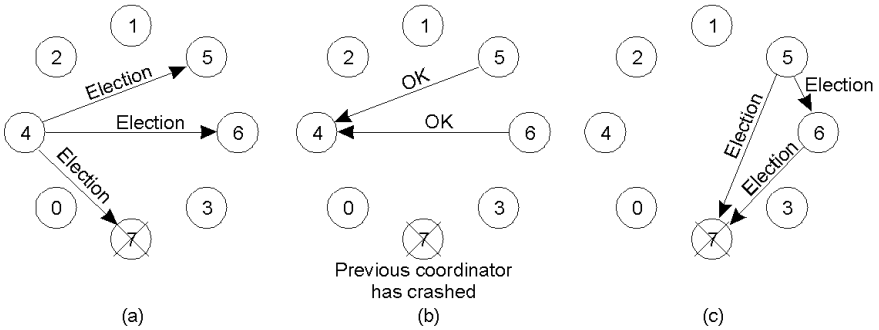
Election by Bullying

Principle: Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator.

Issue: How do we find the heaviest process?

- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).
- If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.
- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

The Bully Algorithm (1)



The bully election algorithm

- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Algorithm (2)