

Synchronization

Chapter 5

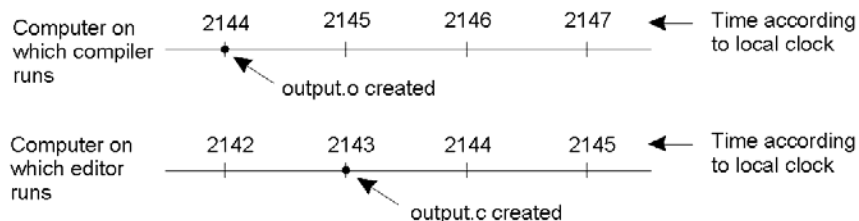
Overview

- We discussed communication between processes in the previous chapter
- Closely related to communication is how processes cooperate and synchronize with one another
- It is essential that multiple processes do not access a shared resource simultaneously
- Also the ordering of events/messages is important
- Synchronization in distributed systems is much harder than uniprocessor or multiprocessor systems
- Mutual exclusion and distributed transactions are two related topics regarding synchronization in distributed systems

6/21/2005

2

Clock Synchronization



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

6/21/2005

3

Universal Coordinated Time (UTC)

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.
- UTC is broadcast through short wave radio and satellite.
- Satellites can give an accuracy of about +/- 0.5 ms.
- Question: Does this solve all our problems? Don't we now have some global timing mechanism?

6/21/2005

4

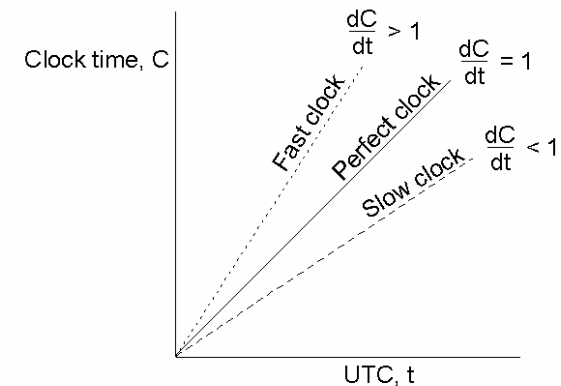
Clock Synchronization Algorithms (1)

- **Problem:** Suppose we have a distributed system with a UTC-receiver somewhere in it → we still have to distribute its time to each machine.
- **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 $C_p(t)$, where t is UTC time.
 - Ideally, we have that for each machine p , $C_p(t) = t$, or, in other words, $dC/dt=1$.

6/21/2005

5

Clock Synchronization Algorithms (2)



- In practice: $1 - \rho \leq dC/dt \leq 1 + \rho$
- Goal: Never let two clocks in any system differ by more than δ time units → synchronize at least every $\delta/2\rho$ seconds.

6/21/2005

6

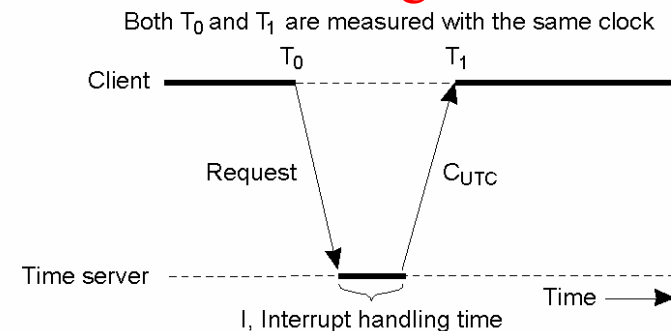
Clock Synchronization Principles

- **Principle I:** Every machine asks a time server for the accurate time at least once every $\delta/2\rho$ seconds.
- Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.
- **Principle II:** 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.
- Okay, you'll probably get every machine in sync. Note: you don't even need to propagate UTC time (why not?)
- **Fundamental problem:** You'll have to take into account that setting the time back is never allowed → smooth adjustments.

6/21/2005

7

Cristian's Algorithm

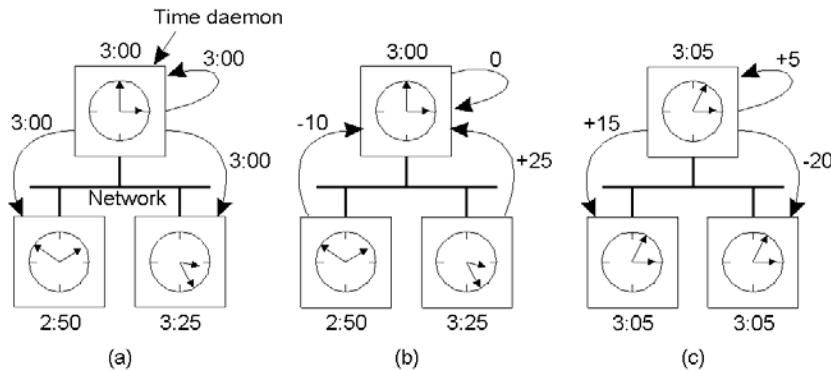


- Suitable if one machine has a UTC receiver (time server) and other machines are synchronized with it
- Every machine asks a time server for the accurate time at least once every $\delta/2\rho$ seconds
- The time server responds as quickly as it can with a message containing the current time, C_{UTC}

6/21/2005

8

The Berkeley Algorithm



- The time daemon asks all the other machines for their clock values
- The machines answer
- The time daemon tells everyone how to adjust their clock

6/21/2005

9

Happened-Before Relationship

- 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 $a \rightarrow b$
 - If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
 - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
- This introduces a partial ordering of events in a system with concurrently operating processes

6/21/2005

10

Logical Clocks (1)

- 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 by one process, and b is the reception of that message by another process, then $C(a) < C(b)$
 - P3: C always goes forward (increasing)
- Problem: How to attach a timestamp to an event when there's no global clock \rightarrow maintain a consistent set of logical clocks, one per process.

6/21/2005

11

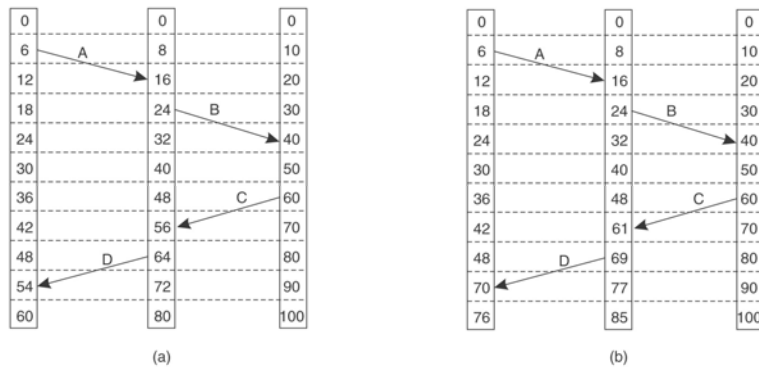
Logical Clocks (2)

- Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
 - For any two successive events that take place within P_i , C_i is incremented by 1
 - Each time a message m is sent by process P_i , the message receives a timestamp $T_m = C_i$.
 - Whenever a message m is received by a process P_j , P_j adjusts its local counter C_j : $C_j \leftarrow \max\{C_j+1, T_m+1\}$
- Property P1 is satisfied by (1); Property P2 by (2) and (3)

6/21/2005

12

Lamport Timestamps



- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clocks.

6/21/2005

13

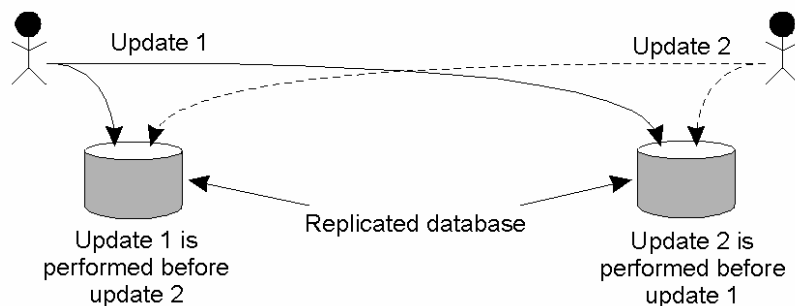
Total Ordering with Logical Clocks

- Problem: It can still occur that two events happen at the same time.
- Solution: Avoid this by attaching a process number to an event
- P_i timestamps event e with $C_i(e).i$, then
 - $C_i(a).i \rightarrow C_j(b).j$ if and only if
 - $C_i(a) < C_j(a)$; or
 - $C_i(a) = C_j(b)$ and $i < j$

6/21/2005

14

Example: Totally-Ordered Multicast(1)



6/21/2005

15

Example: Totally-Ordered Multicast(2)

- Process P_i sends time-stamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
 - Any incoming message at P_j is queued in $queue_j$, according to its timestamp.
 - P_j passes a message msg_i to its application if:
 - msg_i is at the head of $queue_j$
 - for each process P_k , there is a message msg_k in $queue_j$ with a larger timestamp.
- Note: We are assuming that communication is reliable and FIFO ordered.

6/21/2005

16

Vector Timestamps (1)

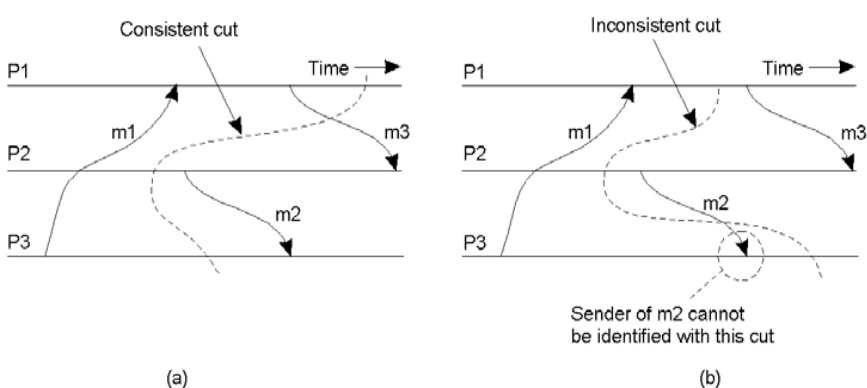
- Observation:
 - Lamport timestamps do not guarantee that if $C(a) < C(b)$ that an event a indeed happened before b .
 - Lamport timestamps do not capture causality (relationship between cause and effect).
- Solution: Vector timestamps
 - Event a is known to causally precede event b if $VT(a) < VT(b)$ where $VT(a)$ is the vector timestamp assigned to an event a
 - Each process P_i maintains a vector $V_i[1..n]$ where $V_i[j]$ denotes the number of events that have occurred so far at process P_j .
 - If $V_i[j] = k$ then P_i knows that k events have occurred at P_j

Vector Timestamps (2)

- At each occurrence of an event at process P_i , $v_i[i]$ is incremented
- When P_i sends a message m , it sends vector v_i along with m as vector timestamp $vt(m)$. Result: upon arrival, each process knows how many events have occurred at P_i (# of events preceded by m , on which m may causally depend).
- When a process P_j received a message m from P_i with vector timestamp $vt(m)$, it (1) updates each $v_j[k]$ to $\max\{v_j[k], vt(m)[k]\}$, and (2) increments $v_j[j]$ by 1.
- To support casual delivery of messages, assume you increment your own component only when sending a message. Then, message m is delivered only if:
 - $vt(m)[j] = v_k[j] + 1$ (m is the next message expected at P_k from P_j)
 - $vt(m)[i] \leq v_k[i]$ for $i \neq j$ (P_k has not seen any messages that were not seen by P_j when it sent message m)

Global State (1)

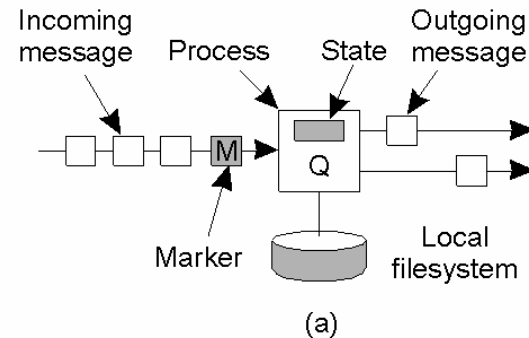
- Current state of a distributed system, called a distributed snapshot.
- Consists of all local states and messages in transit.



(a) A consistent cut

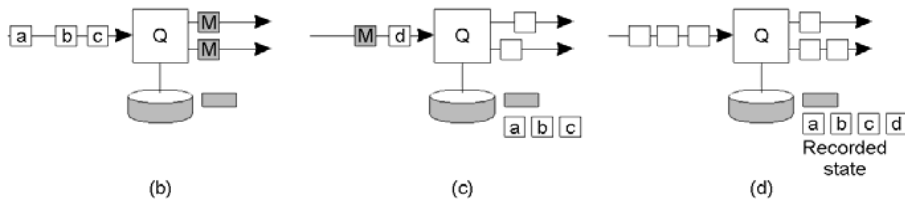
(b) An inconsistent cut

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

6/21/2005

21

Election Algorithms

- An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.
- In many systems the coordinator is chosen by hand (e.g., file servers). This leads to centralized solutions which implies single point of failure.

6/21/2005

22

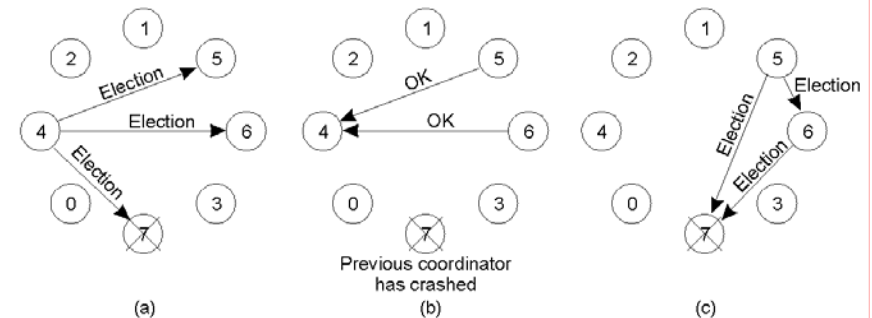
The Bully Algorithm (1)

- Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator.
- To find the heaviest process following algorithm is used:
 - 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.

6/21/2005

23

The Bully Algorithm (2)



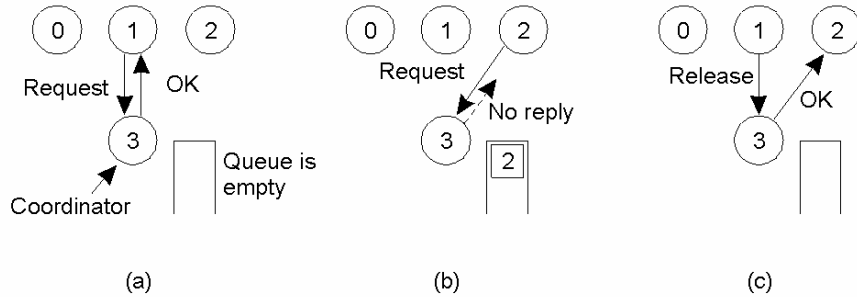
The bully election algorithm

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

6/21/2005

24

A Centralized Algorithm



- a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
- b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
- c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2

6/21/2005

29

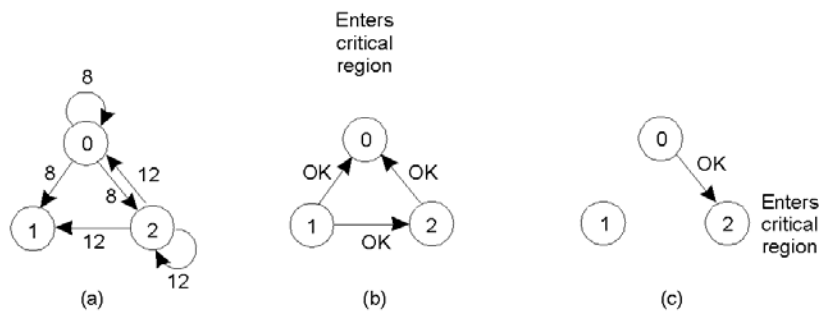
A Distributed Algorithm (1)

- First algorithm based on Lamport's clock synchronization algorithm
- Ricart and Agrawala provided a more efficient version
- Similar to Lamport's algorithm except that acknowledgments aren't sent. Instead, replies (*i.e.*, grants) are sent only when:
 - The receiving process has no interest in the shared resource; or
 - The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps)
- In all other cases, reply is deferred, implying some more local administration
- What are the disadvantages of the distributed algorithm?

6/21/2005

30

A Distributed Algorithm (2)



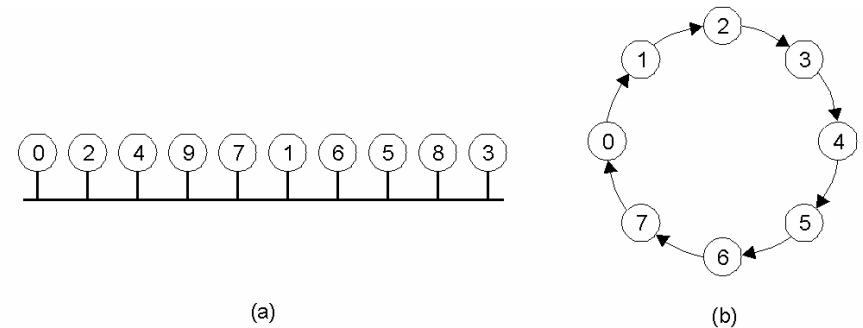
- a) Two processes want to enter the same critical region at the same moment.
- b) Process 0 has the lowest timestamp, so it wins.
- c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

6/21/2005

31

A Token Ring Algorithm

Principle: Organize processes in a logical ring, and let a token be passed between them. The process that holds the token is allowed to enter the critical region (if it likes).



- a) An unordered group of processes on a network.
- b) A logical ring constructed in software.

6/21/2005

32

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3 (request,grant,release)	2	Coordinator crash
Distributed	$2(n-1)$ (n-1) requests & grants	$2(n-1)$	Crash of any process
Token ring	1 to ∞	0 to $n-1$	Lost token, process crash

A comparison of three mutual exclusion algorithms.