TDTS04 Distributed Systems Part C

Slides derived from "Distributed Systems: Principles and Paradigms", by Andrew S. Tanenbaum and Maarten Van Steen, Pearson Int. Ed., as well as material from other instructors including Niklas Carlsson and Juha Takkinen.

MIKAEL ASPLUND
REAL-TIME SYSTEMS LABORATORY
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

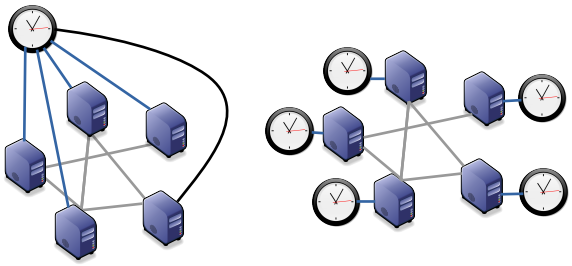
Synchronization

- Agreement over global state among distributed servers/processes
- Communication:
  - Different processes must see messages and events in a consistent order
- Sharing:
  - Shared resources/data should be consistent
- Master/slave relation:
  - Many distributed algorithms require a master server

Notion of time in distributed systems

- Real time
  - Physical clocks
  - Requires clock synchronisation
- Event ordering
  - Logical clocks
  - Requires mechanisms for ordering messages

Global vs. local clocks



Two fundamental models

- Synchronous system model:
  - There is a constant C such that in any time interval in which some processor takes C+1 steps, every nonfaulty processor must take at least one step in that interval.
  - There is a bound on message delays
- Asynchronous system model:
  - No such bounds exist

Clock synchronization

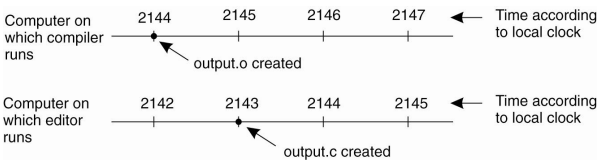
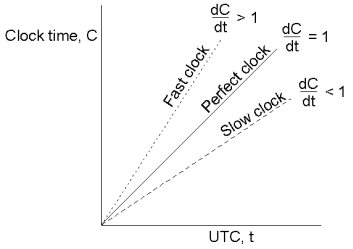


Figure 6-1. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Clocks and clock drifts

- Clocks are oscillators
- Drift caused by differences in oscillator frequencies
- Coordinated universal time (UTC)
  - International standard based on atomic time
  - Broadcast via radio, satellites

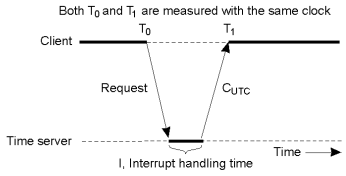
Clock synchronization



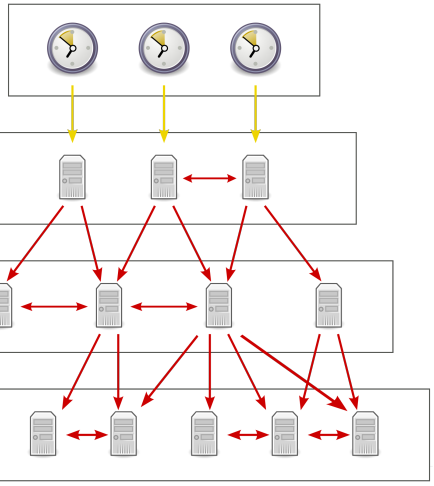
- Each clock has a maximum drift rate ρ
  - $1-\rho \leq dC/dt \leq 1+\rho$
  - Two clocks may drift by $2\rho\Delta$ in time Δ
  - To limit drift to δ, we must therefore resynchronize every $\delta/2\rho$ seconds

Physical Clock Synchronization

- Cristian's Algorithm and NTP – periodically get information from a time server (assumed to be accurate).



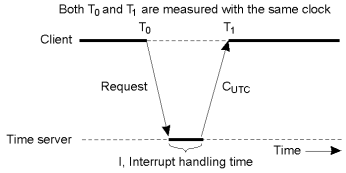
Network Time Protocol



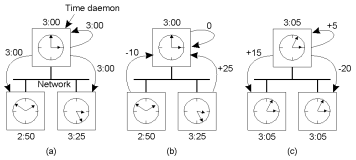
Source: Wikipedia

Physical Clock Synchronization

- Cristian's Algorithm and NTP – periodically get information from a time server (assumed to be accurate).



- Berkeley – active time server uses polling to compute average time Note that the goal is the have correct "relative" time



Event ordering

What is the problem?

- Multiple communicating processes running on different machines
- Events taking place on each process
  - Computation
  - Data read/write
  - Sending/receiving of messages
- In what order are these events happening?
- Can we use clock times of machines?

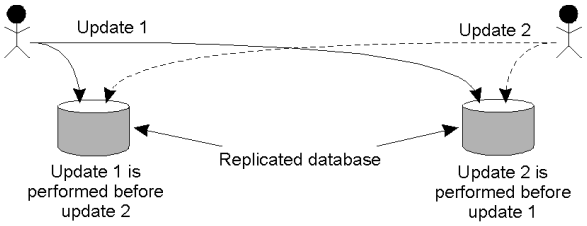
Event ordering

Observation: It may be sufficient that every node agrees on a current time – that time need not be ‘real’ time.

Observation: It may be sufficient that every node agrees on a current time – that time need not be ‘real’ time.

Taking this one step further, in some cases, it is adequate that two systems simply agree on the order in which system events occurred.

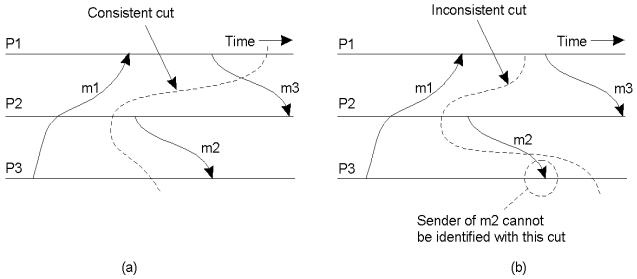
When order matters



Principles of event ordering

- Maintain ordering of distributed events in a consistent manner
- Main Ideas:
  - Idea 1: Non-communicating processes do not need to be synchronized
  - Idea 2: Agreement on ordering is more important than actual time
  - Idea 3: Ordering can be determined by sending and receiving of messages

Example: global state



a) A consistent cut
b) An inconsistent cut

Causal ordering

The "happens-before" relation → can be observed directly in two situations:

- **Rule 1:** If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ is true.
- **Rule 2:** If *a* is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then $a \rightarrow b$

Transitivity: A → B and B → C => A → C

Properties of causal ordering

- "Happens-before" operator creates a partial ordering of all events
- If events A and B are connected through other events
  - Always a well-defined ordering
- If no connection between A and B
  - A and B are considered concurrent

Alternatives to causal ordering

- Total ordering
  - High cost in overhead, all messages must receive a unique order identifier
- Order based on timestamps
  - Clock synchronisation is nontrivial and imperfect
  - Even systems with high precision synchrony can suffer from inconsistency in message ordering
    - Requires periods of inactivity as in the Time-Triggered Architectur (TTA)

Lamport timestamps

- Timestamps should follow the partial event ordering
  - A → B => C(A) < C(B)
- Timestamps always increase
- Lamport's Algorithm:
  - Each processor i maintains a logical clock Ci
  - Whenever an event occurs locally, Ci = Ci+1
  - When i sends message to j, piggyback Ci
  - When j receives message from i
    - Cj = max(Ci, Cj)+1

Lamport's logical clocks (without)

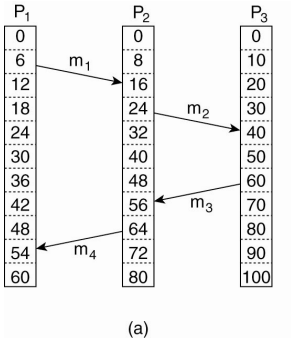


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's logical clocks (with)

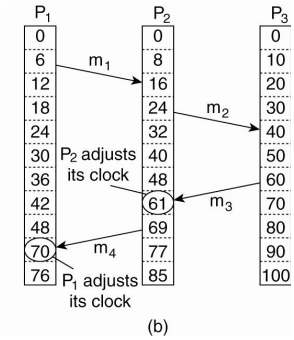


Figure 6-9. (b) Lamport's algorithm corrects the clocks.

Lamport's logical clocks

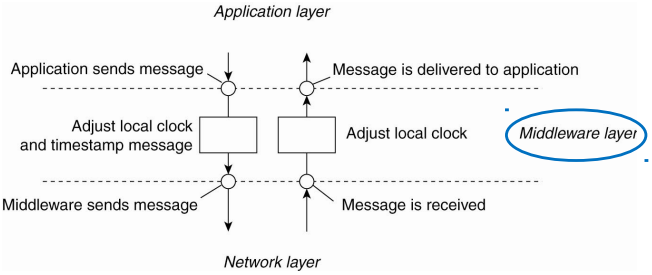


Figure 6-10. The positioning of Lamport's logical clocks in distributed systems.

Lamport clocks and causal ordering

- Lamport clocks fulfill causal order
  A → B => C(A) < C(B)
- Lamport clocks do not exactly match causal order
  C(A) < C(B) ≠> A → B
- Alternative: vector clocks
  - N machines, N logical clocks
  - A vector with N elements is sent with each message
  - Captures exactly causal order
  - Less flexible, more expensive

Some basic distributed algorithms

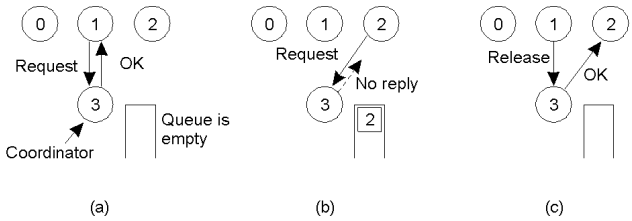
Distributed mutual exclusion

- Multiple processes on different machines may need to access a critical section
- Shared-memory systems:
  - Typically implemented in shared memory
  - Processes share same blocking queues
- How to implement mutual exclusion in distributed systems?

Centralized algorithm

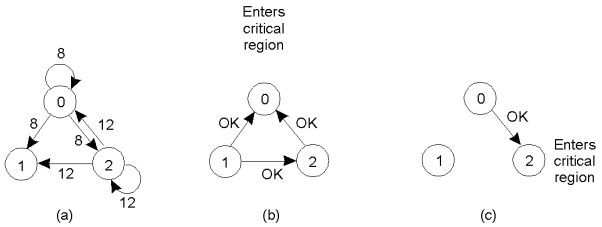
- A coordinator grants access to critical section
  - Maintains a local queue
  - Coordinator can be elected using an election algorithm
- A process sends request to coordinator
  - If nobody in critical section, grant access
  - Otherwise, put process in queue
- When process done:
  - Send release to coordinator
  - Coordinator grants access to next process in queue

Mutual Exclusion: A Centralized Algorithm



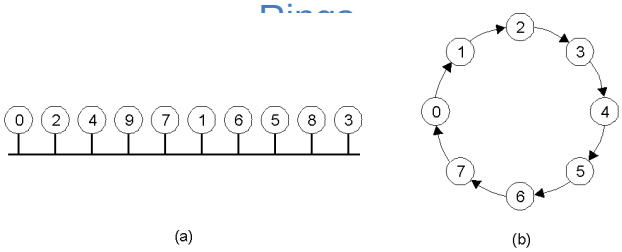
a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

Distributed Mutual Exclusion: Ricart/Agrawala



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.

Distributed Mutual Exclusion: Token Ring



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

Algorithm works by passing a token around the ring. When a process holds the token, it decides if it needs to access the resource at this time. If so, it holds the token while it does so, passing the token on once done.

Problems if the token is ever 'lost' – token loss may also be difficult to detect.

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed (Ricart/Agrawala)	2 (n – 1)	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.

Election Algorithms

Some algorithms require some participating process to act as coordinator. Assuming

- all processes are the same except for a unique number
- the highest numbered process gets to be coordinator
- processes can fail and restart

Election algorithms are a method of finding this highest numbered process and making it known to all processes as the coordinator.

The Bully Algorithm (1)

When process P notices that the current coordinator is no longer responding to requests, it initiates an election:

1. *P* sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, *P* wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

The Bully Algorithm (2)

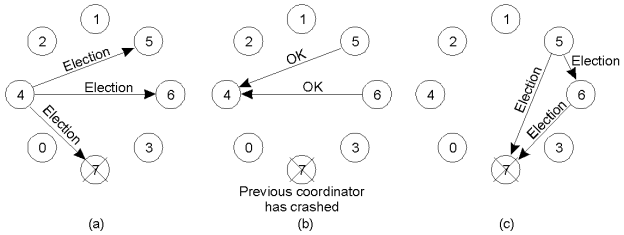
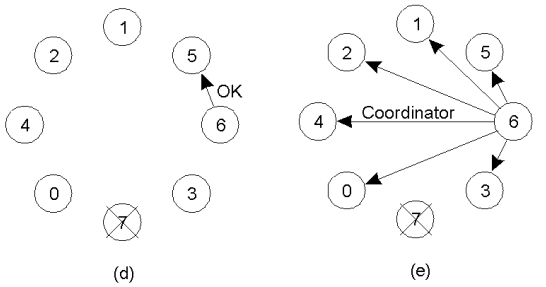


Fig 6-20. The bully election algorithm

- Process 4 notices that 7 is no longer available and holds an election by sending messages to 5 and 6
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

Bully Algorithm (3)



d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone
f) if process 7 ever restarts, it will notify everyone that it is the coordinator

Some notes on replication

Reasons for Replication

- Data are replicated to increase the reliability and performance of a system.
- Replication for performance
  - Scaling in numbers
  - Scaling in geographical area
- Caveat
  - Gain in performance
  - Cost of increased bandwidth for maintaining replication

Content Replication and Placement

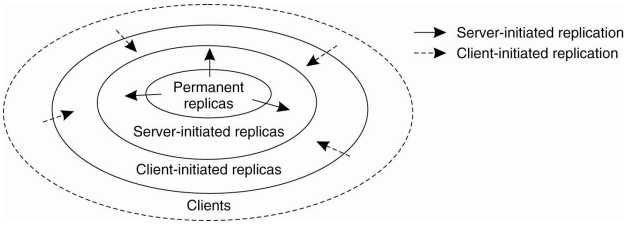
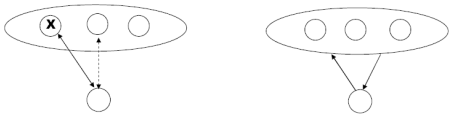


Figure 7-17. The logical organization of different kinds of copies of a data store into three concentric rings.

Replication mechanisms



- Passive replication
  - Primary – backup
- Active replication
  - Group membership

Underlying mechanisms
- Message ordering
- Agreement among replicas

The consensus problem

- **Processes $p_1,\ldots, p_n$ take part in a decision**
  - Each p_i proposes a value v_i
  - All correct processes decide on a common value v that is equal to one of the proposed values
- Desired properties
  - **Termination:** Every correct process eventually decides
  - **Agreement:** No two correct processes decide differently
  - **Validity:** If a process decides v then the value v was proposed by some process

Basic impossibility result

[Fischer, Lynch and Paterson 1985]

- There is no deterministic algorithm solving the consensus problem in an asynchronous distributed system with a single crash failure.