Chapter 6

Coordination

October, 25th

Clock Synchronization

Clock synchronisation in distributed systems

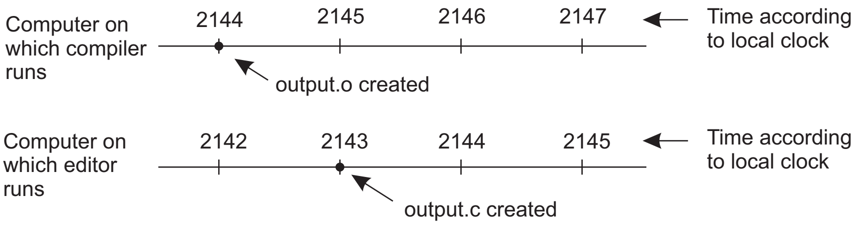
In centralized systems:

- A single clock: time is unambiguous

In distributed systems:

- No global time
- Agreeing on time is not trivial
- Many implications for applications

Example: UNIX `make` program



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

Physical clocks

Why is it so hard to synchronize clocks in a distributed system?

Physical clocks

Why is it so hard to synchronize clocks in a distributed system?

Computer timer (“clock”):

- A quartz crystal that oscillates at a well-defined frequency
- After a number of oscillations, it creates an interrupt (clock tick)
- No two timers are exactly the same
- With time, different clocks tend to diverge (clock skew)

Real-time systems

- How to synchronize clocks with real-world clocks?
- How to synchronize the clocks with each other?
- Universal Coordinated Time (UTC)
  - Computed from International Atomic Time (TAI)
  - Provided as a service by satellites and shortwave radio
  - Precision between $\pm 1 \text{ msec}$ and $\pm 10 \text{ msec}$
- Global Positioning System (GPS)
  - Also give an account of the actual time
  - In principle, tens of nanoseconds

Clock synchronization algorithms

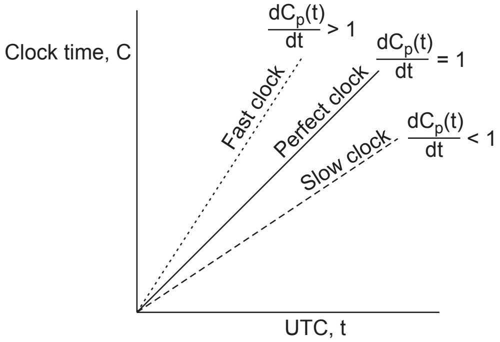
How to keep machines together as well as possible?

System model:

- Each machine has a timer that causes an interrupt H times a sec
- When timer goes off, interrupt handler adds 1 to a software clock
- Clock value C: when UTC time is t, clock on machine p is $C_p(t)$
- Ideally: for all p and t: $C_p(t) = t$ or $C'_p(t) = \frac{dC_p(t)}{dt} = 1$
- $1 - C'_p(t)$ is the drift rate (i.e., difference from a perfect clock)
- $C_p(t) - t$ is the offset relative to a specific time t

Clock synchronization algorithms

Slow, perfect and fast clocks



The relation between clock time and UTC when clocks tick at different rates.

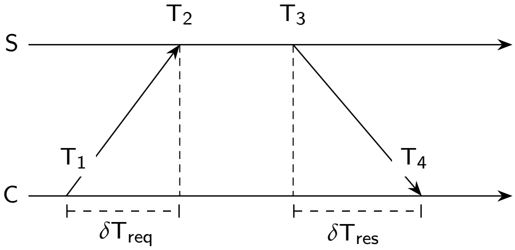
Clock synchronization algorithms

The need for synchronization

- If two clocks are drifting from UTC in the opposite direction, at time δ after last synchronization, they may be up to $2\rho\delta$, where ρ is the maximum drift rate
- In order not to differ by more than δ, clocks must be synchronized (in software) at least every $\delta/2\rho$ seconds
- If goal is **precision**: $\forall t.\ \forall p, q.\ |C_p(t) - C_q(t)| \leq \delta$
- If goal is **accuracy**: $\forall t.\ \forall p.\ |C_p(t) - t| \leq \alpha$

Network Time Protocol (NTP)

- Clients contact a time server, which has an accurate clock
- How to account for message delays?



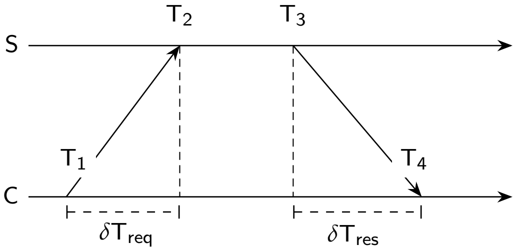
Getting the current time from a time server.

Network Time Protocol (NTP)

- C sends request to S with timestamp T_1; S records T_2 (from its own clock) and returns a response with T_2 and T_3; C records value of T_4
- Assuming *similar propagation delays*, $T_2 - T_1 \approx T_4 - T_3$, C can compute its offset θ as

$$\theta = T_3 + \left[(T_2 - T_1) + (T_4 - T_3)\right]/2 - T_4 = T_3 + (\delta T_{\text{req}} + \delta T_{\text{res}})/2 - T_4$$

- If θ is not zero, C must adjust its clock:
  - $C_{\mathsf{C}}(t) \leftarrow C_{\mathsf{C}}(t) + \theta$

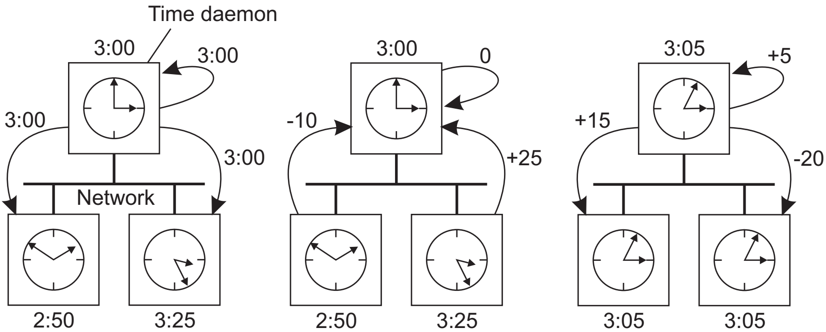


The Berkeley algorithm

- In NTP, time server is passive: other machines ask for time
- In Berkeley UNIX the time server polls every machine periodically
- Based on answers, the server computes average and tells all machines to adjust to the average
- Suitable for systems without access to UTC
- Value of time server (time daemon) is set manually by operator

The Berkeley algorithm

Example



(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.

Logical Clocks

Motivation

- Many applications need to agree on a current time (e.g., make) but this time doesn't need to match real time
- If two processes do not interact, it is not necessary that their clocks be synchronized (lack of synchronization will not be noticed by them!)

Operating Systems

R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

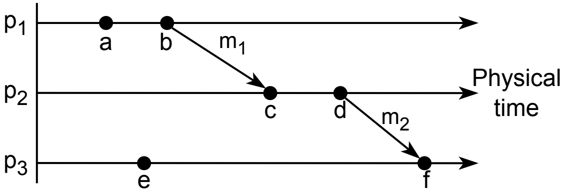
Happens-before relation

- The happens-before relation $\rightarrow$ can be observed directly in two situations:
  - If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$.
  - If a is the event of a message m being sent by one process, and b is the event of m being received by another process, then $a \rightarrow b$
- The happens-before relation is transitive
- If neither $a \rightarrow b$ nor $b \rightarrow a$ then a and b are concurrent

Happens-before relation

Example

1. if $a \to b$ then $b \nrightarrow a$
2. if $a \nrightarrow e$ and $e \nrightarrow a$ then $a||e$
3. there are other sources of dependencies
4. even if $a \to c$, a may or not have caused c



Lamport's logical clock

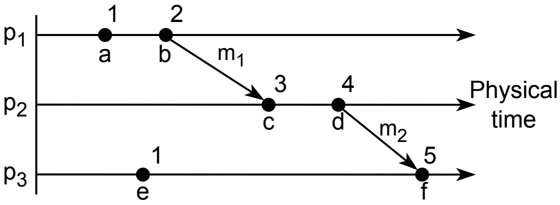
Idea

- Capture happens-before relation numerically
- Monotonically increasing software counter
- Not necessarily related to physical clocks
- Each process keeps a Lamport timestamp L_i
- Timestamp of event e at p_i: $L_i(e)$
- Timestamp of event e at any process: $L(e)$

Lamport's logical clock

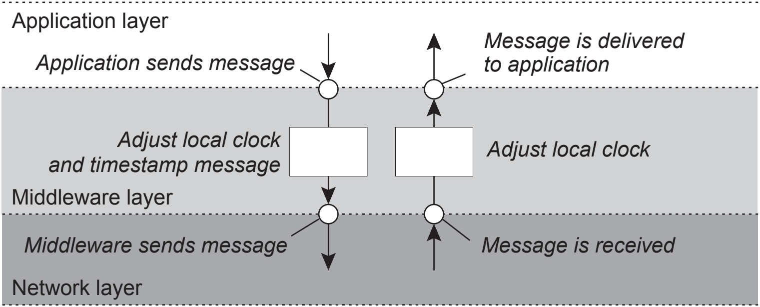
Rules for updating logical clocks

- Processes update their logical clocks and transmit their values in messages following two rules
  - **LC1**: L_i is incremented before each event is issued at process p_i: $L_i \leftarrow L_i + 1$
  - **LC2**:
    (a) when p_i sends m, it piggybacks on m the value $t = L_i$
    (b) on receiving (m, t), p_j sets $L_j \leftarrow \max(L_j, t)$ and then applies LC1 before timestamping `receive(`m`)` event
- **It follows that $\mathbf{e} \rightarrow \mathbf{e}' \Rightarrow \mathbf{L}(\mathbf{e}) < \mathbf{L}(\mathbf{e}')$**



Lamport's logical clock

Implementation of clock adjusts



Totally ordered logical clocks

- Sometimes having a total order of events is useful (e.g., defining total order of messages)
- Lamport timestamps do not give total order

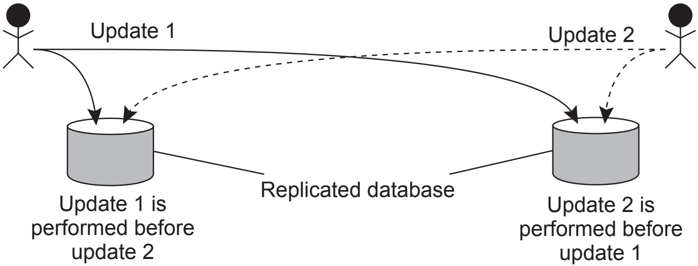
Totally ordered logical clocks

- Sometimes having a total order of events is useful (e.g., defining total order of messages)
- Lamport timestamps do not give total order
- Let e_i be an event at p_i with timestamp T_i and e_j an event at p_j with timestamp T_j
- Define global timestamps as (T_i, i) and (T_j, j) resp.
- $(T_i, i) < (T_j, j)$ iff either (a) $T_i < T_j$ or (b) $T_i = T_j$ and $i < j$

Totally ordered logical clocks

Totally-ordered multicast: Motivation

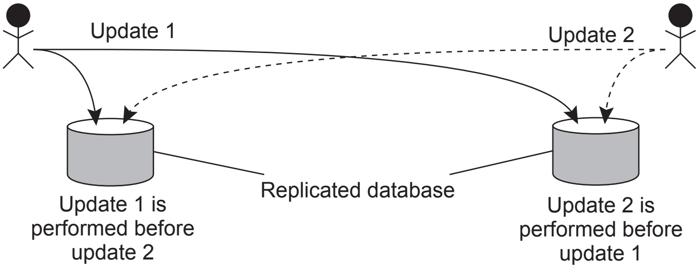
- Concurrent updates on replicated database
- p_1 adds \$100 to an account (initial value \$1000)
- p_2 increments account by 1% (e.g., interests)



Totally ordered logical clocks

Totally-ordered multicast: Motivation

- Concurrent updates on replicated database
- p_1 adds \$100 to an account (initial value \$1000)
- p_2 increments account by 1% (e.g., interests)



Oops! Without synchronization we get \$1111 and \$1110 resp.

Totally ordered logical clocks

Totally-ordered multicast: Solution

- Process p_i sends timestamped message m to all others. Message is put in local buffer b_i
- Any incoming message at p_j is buffered in b_j according to timestamp and ack-ed to every other process
- p_j passes message m to the application if
  - no other message in b_j with a timestamp lower than m's
  - for every process p,
    there is a message from p in b_j with a timestamp at least m's
- **Note:** assuming communication is reliable and FIFO

Vector clocks

- Key property $e \to e' \Leftrightarrow L(e) < L(e')$
- Overcomes limitation $L(e) < L(e') \nRightarrow e \to e'$
- Array of N integers, where system has N processes
- Each process p_i keeps its own vector clock V_i:
  - to timestamp local events
  - to piggyback V_i on messages sent to other processes

Vector clocks

Rules for updating vector clocks

- **VC1**: initially, $V_i[j] \leftarrow 0$, for $j = 1, 2, \ldots, N$
- **VC2**: before p_i timestamps an event, it sets $V_i[i] \leftarrow V_i[i] + 1$
- **VC3**: p_i includes $t = V_i$ in every message it sends
- **VC4**: when pi receives a timestamp t in a message:
  - p_i sets $V_i[j] \leftarrow \max(V_i[j], t[j])$ for $j = 1, 2, \ldots, N$ (merge V_i and t)

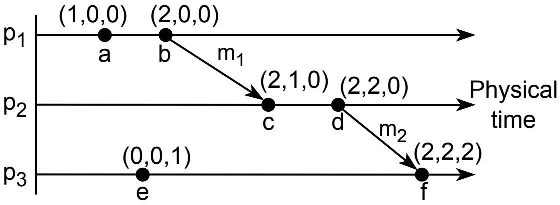
Observations:

- $V_i[i]$ is the number of events p_i has timestamped
- $V_i[j]$, $j \neq i$ is the number of events at p_j that p_i has potentially been affected by

Vector clocks

Vector timestamps are compared as follows

- $V = V'$ iff $V[j] = V'[j]$, for $j = 1, 2, \ldots, N$
- $V \leq V'$ iff $V[j] \leq V'[j]$, for $j = 1, 2, \ldots, N$
- $V < V'$ iff $V \leq V'$ and $V \neq V'$



Mutual exclusion

Introduction

Assumptions:

- N processes, p_i, $i = 1, 2, \ldots, N$
- Processes do not share variables
- Asynchronous system
- No process failures

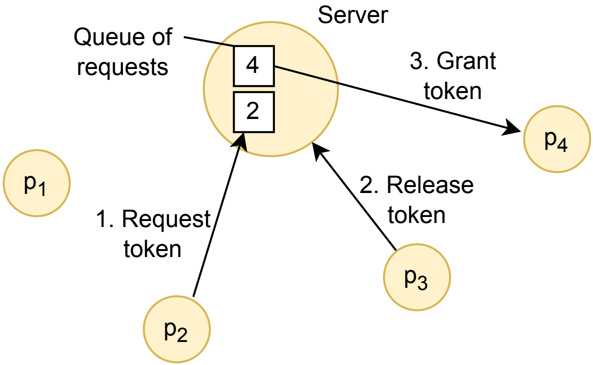
Application-level interface to a critical section:

- `enter()`
- `resourceAccesses()`
- `exit()`

Requirements

- **ME1**: *(safety)* At most one process may execute in the critical section (CS) at a time
- **ME2**: *(liveness)* Requests to enter and exit the critical section eventually succeed
  - *No deadlock*: two or more processes stuck indefinitely while trying to enter or exit CS
  - *No starvation*: indefinite postponement of entry for a process that has requested it
- **ME3**: *(ordering)* If one request to enter the CS happened-before another, then entry to the CS is granted in that order

The central server algorithm



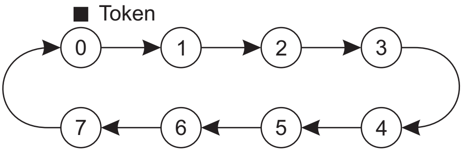
The central server algorithm

Analysis

- In the absence of failures: ME1 and ME2, but...
- ...not ME3!
- Server may become a bottleneck under heavy load
- Performance analysis
    - Entering the critical section
        - 2 messages (request, grant)
        - 2 message delays
    - Exiting the critical section
        - 1 message (release)

Ring-based algorithm

- Arrange processes in a logical ring
- Processes circulate a token around the ring
- To enter CS, a process needs the token
- If a process receives the token and does not want to enter CS, it sends it to its neighbor
- Upon exiting CS, a process sends the token to its neighbor



An overlay network constructed as a logical ring with a token circulating between its members.

Ring-based algorithm

Analysis

- Properties ME1 and ME2 are guaranteed
- What about ME3?
- Performance analysis
  - Processes consume network bandwidth even if no process requires access to cs
  - Entering the critical section
    - Delay between 0 messages and N messages
  - Exiting the critical section
    - No delay and one message exchanged

Algorithm using multicast and logical clocks

- Proposed by Ricart and Agrawala, 1981
- Basic idea
  - To request entry to CS multicast request message
  - Access granted after receiving reply from other processes
- Guarantees properties ME1, ME2, ME3
- Each process p_i has a Lamport's clock
- Messages requesting entry $\langle T, p_i \rangle$, where T is the sender's timestamp and p_i the sender's id

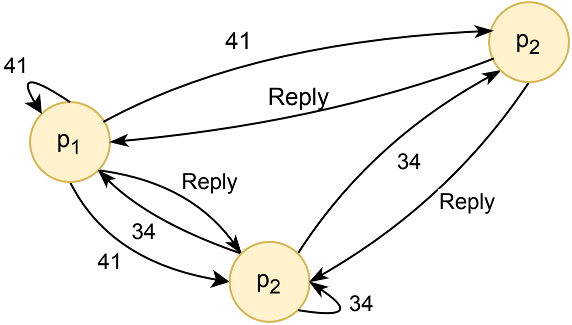
Algorithm using multicast and logical clocks

init
state $\leftarrow$ RELEASED
end
to *enter the critical section*
state $\leftarrow$ WANTED
Multicast `request` to all processes
Wait until number of replies received is $N - 1$
state $\leftarrow$ HELD
end
upon *receipt of a* `request` $\langle T_i, p_i \rangle$ *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
end
to *exit the critical section*
state $\leftarrow$ RELEASED
reply to any queued requests
end

Algorithm using multicast and logical clocks

Example

p_1 and p_2 request access, p_3 is not



Algorithm using multicast and logical clocks

Performance analysis

- Entering the critical section
  - $2(N-1)$ messages ($N-1$ for multicast and $N-1$ for replies)
- Existing the critical section
  - Up to $N-1$ messages (worst case)

Maekawa's voting algorithm

Key observation: access to CS does not need permission from *all* processes, but from a subset of them, *if the subsets overlap!*

Basic idea

- Processes vote for each others' access to CS
- A process must collect a sufficient number of votes
- Processes in the intersection of two voter sets ensure ME1
- Each process p_i is associated to a voting set V_i

Maekawa's voting algorithm

Selecting a voting set $\mathrm{V_i}$

- p_i belongs to V_i (p_i is in its voting set)
- There is at least one common member in the intersection of any two voting sets: $V_i \cap V_j \neq \emptyset$
- Each process has a voting set of the same size K: $|V_i| = K$
- Each process is in K of the voting sets

Optimal value for K is such that $N = K \times (K - 1) + 1$

Maekawa's voting algorithm

```
init
    state ← RELEASED
    voted ← FALSE
end
to enter the critical section at p_i
    state ← WANTED
    Multicast request to all processes in V_i
    Wait until number of replies received is K
    state ← HELD
end
upon receipt of a request from p_i at p_j
    if state = HELD or voted = TRUE then
        queue request from p_i without replying
    else
        send reply immediately to p_i
        voted ← TRUE
    end
end
to exit the critical section at p_i
    state ← RELEASED
    Multicast release to all processes in V_i
end
upon receipt of a release from p_i at p_j
    if queue of requests is non-empty then
        remove head of queue — from p_k, say
        send reply to p_k
        voter ← TRUE
    else
        send reply immediately to p_i
        voted ← TRUE
    end
end
```

Maekawa's voting algorithm

Correctness analysis

- The algorithm ensures ME1
- But it is deadlock prone!
  - Processes p_1, p_2 and p_3 request CS concurrently
  - Let $V1 = \{p1, p2\}$, $V2 = \{p2, p3\}$, $V3 = \{p3, p1\}$
  - p_1 replies to itself and waits for p_2
  - p_2 replies to itself and waits for p_3
  - p_3 replies to itself and waits for p_1!!!
- The fixed protocol ensures ME2 and ME3
  - Processes queue outstanding requests in happened-before order (similar to Ricart and Agrawala's algorithm)

Maekawa's voting algorithm

Performance analysis

- Entering the critical section
  - $\approx 2\sqrt{N}$ messages
- Exiting the critical section
  - $\approx \sqrt{N}$ messages