Outline

• Clock Synchronization
  ➢ Physical clocks
  ➢ Logical clocks
• Mutual exclusion
• Election algorithms
Clock Synchronization

- Time in a centralized system
- Lack of global time agreement in a distributed system
  - Implications?
- Physical clocks
- Logical clocks

Physical clocks 1/2

- Timer (counter and holding register)
- Clock tick
- Date and time entered after booting converted to the number of ticks after some known starting date and stored in memory
  - Battery-backed up CMOS RAM
  - With every clock tick \(\rightarrow\) ISR adds 1 to time stored in memory
- Multiple CPUs \(\rightarrow\) clock skew
**Physical clocks 2/2**

- Astronomers (Transit of the sun and mean solar second because days are getting longer)
- Physicists
  - Atomic clock in 1948 $\rightarrow$ transitions of the cesium 133 atom
  - Atomic second = mean solar second
    - Time it takes cesium 133 atom to make $9,192,631,770$ transitions
  - International Atomic Time (TAI)
    - Mean number of ticks of cesium 133 clocks since midnight on Jan, 1, 1958 divided by $9,192,631,770$
    - $86,400$ TAI seconds is now about $3$ msec less than a mean solar day
    - Introduce *leap seconds* whenever discrepancy between TAI and solar time grows to $800$ msec
  - Universal coordinated time (UTC)

---

**Getting Time from a Time Server**

\[
\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2}
\]

A’s offset relative to B = $\theta$

NTP buffers eight pairs of $(\theta, \text{delay})$ selecting minimal delay
Lamport’s Logical Clocks 1/5

- To synchronize logical clocks (need to agree on order of events, but not exact time)
- 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 $a \rightarrow b$ is true.
  - 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 $a \rightarrow b$
- If $a \rightarrow b$, then $C(a) < C(b)$ where $C(i)$ is a time value

Lamport’s Logical Clocks 2/5

Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates.
Lamport’s Logical Clocks 3/5

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

Lamport’s Logical Clocks 4/5

Figure 6-10. The positioning of Lamport’s logical clocks in distributed systems.
Lamport’s Logical Clocks 5/5

• Updating counter $C_i$ for process $P_i$
  1. Before executing an event, $P_i$ executes $C_i \leftarrow C_i + 1$.
  2. When process $P_i$ sends a message $m$ to $P_j$, it sets $m$’s timestamp $ts(m)$ equal to $C_i$ after having executed the previous step.
  3. Upon the receipt of a message $m$, process $P_j$ adjusts its own local counter as $C_j \leftarrow \max\{C_j, ts(m)\}$, after which it then executes the first step and delivers the message to the application.

Example: Totally Ordered Multicasting

• Message timestamped with current logical time of sender
• Receiver multicasts an ACK to other processes
• Deliver message when at head of queue and has been ACKed by each other process
Vector Clocks 1/3

• If \( C(a) < C(b) \) does not imply that \( a \) happened before \( b \)?
• \( T_{\text{recv}}(m_1) < T_{\text{snd}}(m_2) \)
• Sending \( m_2 \) has nothing to do with receipt of \( m_1 \)
• Lamport clocks do not capture causality
• Need for vector clocks

\[
\begin{array}{c|c|c|c}
& P_1 & P_2 & P_3 \\
\hline
m_1 & 0 & 8 & 0 \\
6 & 12 & 16 & 10 \\
18 & 24 & 32 & 40 \\
30 & 40 & 60 & 50 \\
42 & 61 & 70 & 70 \\
48 & 69 & 80 & 80 \\
70 & 77 & 90 & 100 \\
\end{array}
\]

Figure 6-12. Concurrent message transmission using logical clocks.

Vector Clocks 2/3

• Vector clocks are constructed by letting each process \( P_i \) maintain a vector \( VC_i \) with the following two properties:
  1. \( VC_i[i] \) is the number of events that have occurred so far at \( P_i \). In other words, \( VC_i[i] \) is the local logical clock at process \( P_i \).
  2. If \( VC_i[j] = k \) then \( P_i \) knows that \( k \) events have occurred at \( P_j \). It is thus \( P_i \)'s knowledge of the local time at \( P_j \).
Vector Clocks 3/3

Steps carried out to accomplish property 2 of previous slide:

1. Before executing an event $P_i$ executes
   \[ VC_i[i] \leftarrow VC_i[i] + 1. \]

2. When process $P_i$ sends a message $m$ to $P_j$, it sets $m$’s (vector) timestamp $ts(m)$ equal to $VC_i$ after having executed the previous step.

3. Upon the receipt of a message $m$, process $P_j$ adjusts its own vector by setting
   \[ VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\} \]
   for each $k$, after which it executes the first step and delivers the message to the application.

Enforcing Causal Communication 1/2

- Causally-ordered communication $\Rightarrow$ weaker than totally-ordered communication
- $P_j$ receives a message $m$ from $P_i$ with vector timestamp $ts(m)$
- Delivery of message to application layer delayed until 2 conditions met
  - $ts(m)[i]=VC_j[i]+1$
    - $m$ is next message $P_j$ expects from $P_i$
  - $ts(m)[k]\leq VC_j[k]$ for all $k \neq i$
    - $P_i$ has seen all the messages that have been seen by $P_i$ when it sent message $m$
Enforcing Causal Communication 2/2

- Figure 6-13. Enforcing causal communication.

Distributed Mutual Exclusion

- Token-based solutions
  - One token
  - Whoever has the token, can access the shared resource
  - When finished, pass token to next resource
  - Avoids starvation an deadlock
  - A problem if token is lost

- Permission-based solutions
  - Ask for permission first
Mutual Exclusion: A Centralized Algorithm 1/3

Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion: A Centralized Algorithm 2/3

Figure 6-14. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.
Mutual Exclusion: A Centralized Algorithm 3/3

Figure 6-14. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

A Distributed Algorithm 1/6

- Assume total ordering of all events in the system
- When access a shared resource
  - Build a message (resource name, process number, and current logical time)
  - Send message to all other processes, including itself
- When process receives a request
  - Action depends on own state with respect to resource in message
  - 3 different cases
A Distributed Algorithm 2/6

Three different cases:
1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

A Distributed Algorithm 3/6

- After sending out asking for permissions
  - Sit back and wait until everyone has given permission
  - After finished
    - send OK to all processes on its queue and delete from queue
Figure 6-15. (a) Two processes want to access a shared resource at the same moment.

(b) Process 0 has the lowest timestamp, so it wins.
A Distributed Algorithm 6/6

Figure 6-15. (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

A Token Ring Algorithm

Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.