Synchronization

Dr. Yingwu Zhu
Topics to Discuss

• Physical Clocks

• Logical Clocks: Lamport Clocks
  – Classic paper: Time, Clocks, and the Ordering of Events in a Distributed System

• Lamport Vector Clocks
Synchronization? What’s for?

- Temporal ordering of events produced by concurrent processes
- Synchronization between senders and receivers of message
  - Msg $m_1$ from process P to Q is sent before or after msg $m_2$ from process Q?
- Coordination of joint activity
- Serialization of concurrent access for shared objects (e.g., access to a shared printer)
An Ideal World

• All machines’ clocks are perfectly synchronized, synchronization is really easy!
Clock Synchronization Example

- In centralized systems, no problem for the above Make program
- In distributed systems, when each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
Logical vs. Physical Clocks

• Logical clock keeps track of event ordering
  – Among related (causal) events
  – Do not care the real time where events occurred

• Physical clock keeps time of day
  – Consistent across systems
Physical Clock (timer) in Computers

• Real-time Clock: CMOS clock (counter) circuit driven by a quartz oscillator
  – battery backup to continue measuring time when power is off

• OS generally programs a timer circuit to generate an interrupt periodically
  – e.g., 60, 100, 250, 1000 interrupts per second (Linux 2.6+ adjustable up to 1000 Hz)
  – Programmable Interval Timer (PIT) – Intel 8253, 8254
  – Interrupt service procedure adds 1 to a counter in memory
Physical Clock Problems

• Getting two systems to agree on time
  – Two clocks hardly ever agree
  – Quartz oscillators oscillate at slightly different frequencies

• Clocks tick at different rates
  – Create ever-widening gap in perceived time
  – Clock Drift: Relative Difference in clock frequencies (rates) of two processes

• Difference between two clocks at one point in time
  – Clock Skew
Clock Drift

Frequencies of perfect, slow and fast clocks
Dealing with Drift

• Assume we set computer to true time

• Not good idea to set clock back
  – Illusion of time moving backwards can confuse message ordering and software development environments
Dealing with Drift

• Go for gradual clock correction
  – If fast:
    • Make clock run slower until it synchronizes
  – If slow:
    • Make clock run faster until it synchronizes

• Clock synchronization, e.g., Linear compensation function
Compensating for a fast clock

\[
\frac{dC}{dt} > 1
\]

Clock synchronized

Linear compensating function applied

UTC time, \( t \)

Computer's time, \( C \)
Getting Accurate Time

• Attach GPS receiver to each computer
  ±1 msec of UTC (Universal Coordinated Time)

• Attach WWV radio receiver
  Obtain time broadcasts from Boulder or DC
  ±3 msec of UTC (depending on distance)

• Attach GOES receiver
  ±0.1 msec of UTC

• Not practical solution for every machine
  – Cost, size, convenience, environment
Practical Clock Synchronization

- NTP (Network Time Protocol)
- Berkeley algorithm
Clock Synchronization: Network Time Protocol (NTP)

• Synchronize from another machine
  – One with a more accurate clock

• Machine/service that provides time information:
  Time server (w/ WWV receiver)
Clock Synchronization: NTP

Assumption: latency A→B and B→A is same, and good estimate!
Offset of A to B: \( \theta = T_3 - \frac{[(T_2 - T_1) + (T_4 - T_3)]}{2} \)
Delay estimate: \( \delta = \frac{[(T_2 - T_1) + (T_4 - T_3)]}{2} \), keeps the minimum one!
Adjust gradually: e.g., to slow down, add a smaller time for each interrupt
The Network Time Protocol (NTP)

- Uses a network of time servers to synchronize all processes on a network.
- Time servers are connected by a synchronization subnet tree. The root is in touch with UTC. Each node synchronizes its children nodes.
Messages Exchanged Between a Pair of NTP Peers (“Connected Servers”)

Each message bears timestamps of recent message events: the local time when the previous NTP message was sent and received, and the local time when the current message was transmitted.
Theoretical Base for NTP

- $t$ and $t'$: actual transmission times for $m$ and $m'$
- $o$: true offset of the clock at $B$ relative to that at $A$
- $o_i$: estimate of the actual offset between the two clocks
- $d_i$: estimate of accuracy of $o_i$; total transmission times for $m$ and $m'$; $d_i = t + t'$

\[ T_{i-2} = T_{i-3} + t + o \]
\[ T_i = T_{i-1} + t' - o \]

This leads to
\[ d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1} \]
\[ o = o_i + \frac{(t' - t)}{2}, \text{ where} \]
\[ o_i = \frac{(T_{i-2} - T_{i-3} + T_{i-1} - T_i)}{2}. \]
It can also be shown that
\[ o_i - d_i / 2 \leq o \leq o_i + d_i / 2. \]
Clock Synchronization: The Berkeley Algorithm

- The time server is active, polling every machine periodically for their time
- Based on responses, it computes an average time and tells every machine to adjust their clocks
- Used scenarios: No machine has a WWV receiver
- All machines agree on the same time, but not necessarily the real time
The Berkeley Algorithm (1)

- The time daemon asks all the other machines for their clock values.
The Berkeley Algorithm (2)

• The machines answer.
The Berkeley Algorithm (3)

The time daemon tells everyone how to adjust their clock.
Logical Clocks

• In a classic paper (1978), Lamport showed
  – Although clock synchronization is possible, it need not be absolute
  – If two processes do not interact, it is not necessary that their clocks be synchronized!
  – More importantly, the processes should agree on the order in which events occur! This matters!
Logical Clocks

• Assign sequence numbers to messages
  – All cooperating processes can agree on order of events
  – vs. physical clocks: time of day

• Assume NO central time source
  – Each system maintains its own local clock
  – No total ordering of events
  – No concept of happened-when
Lamport’s Logical Clocks (1)

- 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 \( a \rightarrow b \)
  - \textit{Happens-before} is transitive: if \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \)
Logical clocks & concurrency

• Assign “clock” value to each event
  – if $a \rightarrow b$ then $\text{clock}(a) < \text{clock}(b)$
  – since time cannot run backwards

• If $a$ and $b$ occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true
  – These events are concurrent
Lamport’s Logical Clocks (1)

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

Lamport clocks: Counters or Sequence numbers
Lamport’s Logical Clocks (2)

(b) Lamport’s algorithm corrects the clocks (by adding 1).
Lamport’s Logical Clocks (3)

Application layer

Application sends message
Adjust local clock and timestamp message
Middleware sends message

Network layer

Middleware layer

Message is delivered to application
Adjust local clock
Message is received
Lamport’s Logical Clocks (4)

• Updating the local counter $C_i$ for process $P_i$
• Each process maintains a local counter

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
   \[ C_j \leftarrow \max\{C_j, ts(m)\} + 1, \] delivers the message to the application.
Lamport’s algorithm

• Each message carries a timestamp of the sender’s clock

• When a message arrives:
  – if receiver’s clock < message timestamp, set system clock to (message timestamp + 1)
  – else do nothing

• Clock must be advanced between any two events in the same process
Lamport’s algorithm

• Algorithm allows us to maintain time ordering among related events
  – Partial ordering
Summary

• Algorithm needs monotonically increasing software counter
• Incremented at least when events that need to be timestamped occur
• Each event has a Lamport timestamp attached to it
• For any two events, where $a \rightarrow b$: $C(a) < C(b)$
Problems

• Identical timestamps: two events could be concurrent

• Detect causal relations
  – If \( C(e) < C(e') \), cannot conclude that \( e \rightarrow e' \)
  – Looking at Lamport timestamps, cannot conclude which events are causally related

• Solution: use a vector clock
Lamport Logical Time Problem

3 and 7 are logically concurrent events.
Vector Clocks (1)

- Concurrent message transmission using logical clocks.
- $T_{\text{recv}}(m_1) < T_{\text{snd}}(m_2)$, but $m_1$ and $m_2$ are concurrent
- Lamport clocks do not capture causality!
Vector Clocks (2)

- 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 $P_j$.

If $VC[a] < VC[b]$ then event $a \rightarrow$ event $b$
Vector Clocks (3)

- With Lamport Logical Timestamp
  
  $e \text{ “happens-before” } f \Rightarrow \text{timestamp}(e) < \text{timestamp}(f)$, but
  
  $\text{timestamp}(e) < \text{timestamp}(f) \Rightarrow e \text{ “happens-before” } f$
  
- Vector clock addresses this issue:
  
  - All processes use a vector of counters (logical clocks), $i^{th}$ element is the clock value for process $i$, initially all zero.
  
  - Each process $i$ increments the $i^{th}$ element of its vector upon an instruction or send event. Vector value is timestamp of the event.
  
  - A $\text{send(message)}$ event carries its vector timestamp (counter vector)
  
  - For a $\text{receive(message)}$ event,

\[
V_{\text{receiver}[j]} = \begin{cases} 
\text{Max}(V_{\text{receiver}[j]} , V_{\text{message}[j]}), & \text{if } j \text{ is not self} \\
V_{\text{receiver}[j]} + 1, & \text{otherwise}
\end{cases}
\]
Vector Clocks (4)

Physical time

Vector Clocks (4)
Vector Clocks (5)

Physical Time

p 1 0,0,0

p 2 0,0,0

p 3 0,0,0

p 4 0,0,0

(1,0,0,0) (2,0,0,0) (1,2,0,0) (4,0,2,2) (2,0,2,2) (2,0,2,3)

(2,0,1,0) (2,2,3,0) (4,2,4,1) (4,2,5,3)

(Vector logical clock) (vector timestamp) Message
Comparing Vector Clocks

- $\text{VT}_1 = \text{VT}_2$,
  
  $iff$ $\text{VT}_1[i] = \text{VT}_2[i]$, for all $i = 1, \ldots, n$

- $\text{VT}_1 \preceq \text{VT}_2$,
  
  $iff$ $\text{VT}_1[i] \preceq \text{VT}_2[i]$, for all $i = 1, \ldots, n$

- $\text{VT}_1 < \text{VT}_2$,
  
  $iff$ $\text{VT}_1 < \text{VT}_2$ &
  
  $\exists j \ (1 \leq j \leq n \land \text{VT}_1[j] < \text{VT}_2[j])$

- $\text{VT}_1$ is concurrent with $\text{VT}_2$
  
  $iff$ (not $\text{VT}_1 \preceq \text{VT}_2$ AND not $\text{VT}_2 \preceq \text{VT}_1$)