Synchronization

- Clock Synchronization, Logical clocks
- Global State, Election, Critical Sections
- Transactions

Original slides from A. Tanenbaum’s Distributed OS Book

Some slides are from Prof. Jalal Y. Kawash at Univ. of Calgary, Prof. Steve Goddard at University Nebraska, Lincoln, Prof. Harandi, Prof. Hou, Prof. Gupta and Prof. Vaidya from University of Illinois, Prof. Kulkarni from Princeton University
Why Synchronize?

• Often important to control access to a single, shared resource.
• Also often important to agree on the ordering of events.
• Synchronization in Distributed Systems is much more difficult than in uniprocessor systems.

We will study:
1. Synchronization based on “Actual Time”.
2. Synchronization based on “Relative Time”.
4. Distributed Transactions.
Event Ordering

Centralized System

• A process makes a kernel call to get the time
• Process which tries to get the time later will always get a higher (or equal) time value

⇒ no ambiguity in the order of events and their time

DS ?
Part I
Clock Synchronization
Logical clocks
Computer clocks and timing events

Each computer in a DS has its own internal clock
  – used by local processes to obtain the value of the current time
  – processes on different computers can timestamp their events
  – but clocks on different computers may give different times
  – computer clocks drift from perfect time and their drift rates differ from one another.
    – *clock drift rate*: the relative amount that a computer clock differs from a perfect clock

Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied
Lack of Global Time in DS

- It is impossible to guarantee that physical clocks run at the same frequency
- Lack of global time, can cause problems
- Example: UNIX `make`
  - Edit `output.c` at a client
  - `output.o` is at a server (compile at server)
  - Client machine clock can be lagging behind the server machine clock
Lack of Global Time – Example

When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
Physical Clock Synchronization (1)

- **External:** synchronize with an external resource, UTC source
  \[ |S(t) - C_i(t)| < D \]
- **Internal:** synchronize without access to an external resource
  \[ |C_i(t) - C_j(t)| < D \]
- **Monotonicity:** time never goes back
  \[ t' > t \Rightarrow C(t') > C(t) \]
Physical Clock Skew

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

A synchronous distributed system is one in which the following bounds are defined:

– the time to execute each step of a process has known lower and upper bounds
– each message transmitted over a channel is received within a known bounded time
– each process has a local clock whose drift rate from real time has a known bound
Cristian’s Algorithm – External Synch

- External source S
- Denote clock value at process X by C(X)

Periodically, a process P:
1. send message to S, requesting time
2. Receive message from S, containing time C(S)
3. Adjust C at P, C(P) = C(S)

- Reply takes time
- Time for different replies varies
- When P adjusts C(P) to C(S), C(S) > C(P)
Cristian's Algorithm

Both $T_0$ and $T_1$ are measured with the same clock

Request

$I$, Interrupt handling time

Getting the current time from a time server.
Uncertainty

- One process $p_1$ sends its local time $t$ to process $p_2$ in a message $m$,
  - $p_2$ set its clock to $t + T_{\text{trans}}$ where $T_{\text{trans}}$ is the time to transmit $m$
  - if $T_{\text{trans}}$ is unknown but $\min \leq T_{\text{trans}} \leq \max$
  - uncertainty $u = \max - \min$.
  - Set clock to $t + (\max - \min)/2 \rightarrow \text{skew} \leq u/2$
Adjusting Client’s Clock

- $T_{propogation} =$ time to send a request or receive reply
- $T_0$ time request is sent
- $T_1$ time reply received
- $T_{propogation} = (T_1 - T_0)/2$ (estimate)
- $C$ at $P =$ time(S) + $T_{propogation}$
- Works if both request and reply are sent on the same network
Improvements to Cristian’s Algorithm

- $I =$ interrupt handling time at $S$
- $T_{\text{propogation}} = (T_1 - T_0 - I)/2$

- Take several measurements for $T_{\text{propogation}}$
- Worst cases, best cases of $(T_1 - T_0)$
- Discard worst cases
- Consider the fastest case of $(T_1 - T_0)$
- Average measurements over time
Berkeley Algorithm – Internal Synch

- Internal: synchronize without access to an external resource

\[ |C_i(t) - C_j(t)| < D \]

Periodically,
S: send C(S) to each client P
P: calculate \( \alpha_P = C(P) - C(S) \)
    send \( \alpha_P \) to S
S: receive all \( \alpha_P \)'s
    compute an average \( \Delta \) of \( \alpha_P \), including \( \alpha_S \)
    send \( \Delta \) to all clients P
P: apply \( \Delta \) to C(P)
The Berkeley Algorithm

- Propagation time?
- Extreme cases? Faulty clocks?
- Time server fails?

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
Importance of Synchronized Clocks

• New H/W and S/W for synchronizing clocks is easily available

• Nowadays, it is possible to keep millions of clocks synchronized to within few milliseconds of UTC
Logical Clocks

For many DS algorithms, associating an event to an absolute real time is not essential, we only need to know an unambiguous order of events

- Lamport's timestamps
- Vector timestamps
Logical Clocks (Cont.)

- Synchronization based on “relative time”.
- “relative time” may not relate to the “real time”.
- What’s important is that the processes in the Distributed System agree on the ordering in which certain events occur.
- Such “clocks” are referred to as Logical Clocks.
Example: Why Order Matters?

- Replicated accounts in Ankara (A) and Istanbul (S)
- Two updates occur at the same time
  - Current balance: 10,000YTL
  - Update1: Add 1000YTL at S; Update2: Add interest of 1% at A
  - inconsistent states!
Lamport Algorithm

• Clock synchronization does not have to be exact
  – Synchronization not needed if there is no interaction between machines
  – Synchronization only needed when machines communicate
  – i.e. must only agree on ordering of interacting events
Events and Process States

- A distributed system is defined as a collection $P$ of $N$ processes $p_i$, $i = 1, 2, ..., N$
- Each process $p_i$ has a state $s_i$ consisting of its variables (which it transforms as it executes)
- Processes communicate only by messages (via a network)
- Actions of processes: Send, Receive, change own state
- Event: the occurrence of a single action that a process carries out as it executes e.g. Send, Receive, change state
- Events at a single process $p_i$, can be placed in a total ordering denoted by the relation $\rightarrow_i$ between the events. i.e.
  \[ e \rightarrow_i e' \] if and only if the event $e$ occurs before $e'$ in $p_i$
- A history of process $p_i$: is a series of events ordered by $\rightarrow_i$
  \[ \text{history}(p_i) = h_i = \langle e^0_i, e^1_i, e^2_i, \ldots \rangle \]
Lamport’s “Happens-Before” Partial Order

Given two events e & e’, e \rightarrow e’ if:

- **Same process**: e \rightarrow_i e’, for some process P_i
- **Same message**: e = send(m) and e’ = receive(m) for some message m

**Transitivity**: there is an event e* such that e \rightarrow e* and e* \rightarrow e’
Concurrent Events

• Given two events e & e’:
• If not e → e’ and not e’ → e, then e || e’
Lamport Logical Clocks

Substitute synchronized clocks with a global Ordering of events

- $LC_i$ is a local clock: contains increasing values
  - each process $i$ has own $LC_i$
- Increment $LC_i$ on each event occurrence
- $e_i \rightarrow e_j \Rightarrow LC(e_i) < LC(e_j)$
- within same process $i$, if $e_j$ occurs before $e_k$
  - $LC_i(e_j) < LC_i(e_k)$
- If $e_s$ is a send event and $e_r$ receives that send, then
  - $LC_i(e_s) < LC_j(e_r)$
Lamport Algorithm

- Each process increments local clock between any two successive events
- Message contains a timestamp
- Upon receiving a message, if received timestamp is ahead, receiver fast forward its clock to be one more than sending time
Lamport Algorithm (cont.)

• Timestamp
  – Each event is given a timestamp $t$
  – If $e_s$ is a send message $m$ from $p_i$, then $t = LC_i(e_s)$
  – When $p_j$ receives $m$, set $LC_j$ value as follows
    • If $t < LC_j$, increment $LC_j$ by one
      – Message regarded as next event on $j$
    • If $t \geq LC_j$, set $LC_j$ to $t+1$
Lamport’s Algorithm Analysis (1)

- **Claim:** $e_i \rightarrow e_j \Rightarrow LC(e_i) < LC(e_j)$
- **Proof:** by induction on the length of the sequence of events relating to $e_i$ and $e_j$
Lamport’s Algorithm Analysis (2)

- LC(e_i) < LC(e_j) ⇒ e_i → e_j?
- Claim: if LC(e_i) < LC(e_j), then it is **not** necessarily true that e_i → e_j
Total Ordering of Events

- Happens Before is only a partial order

- Make the timestamp of an event $e$ of process $P_i$ be:
  $$(\text{LC}(e), i)$$

- $(a, b) < (c, d)$ iff $a < c$, or $a = c$ and $b < d$
Application: Totally-Ordered Multicasting

- Message is timestamped with sender’s logical time
- Message is multicast (including sender itself)
- When message is received
  - It is put into local queue
  - Ordered according to timestamp
  - Multicast acknowledgement
- Message is delivered to applications only when
  - It is at head of queue
  - It has been acknowledged by all involved processes
Update 1 is time-stamped and multicast. Added to local queues.
Update 2 is time-stamped and multicast. Added to local queues.

Acknowledgements for Update 2 sent/received. Update 2 can now be processed.
Acknowledgements for Update 1 sent/received. Update 1 can now be processed.

(Note: all queues are the same, as the timestamps have been used to ensure the “happens-before” relation holds.)
Limitation of Lamport’s Algorithm

- $e_i \rightarrow e_j \Rightarrow \text{LC}(e_i) < \text{LC}(e_j)$
- However, $\text{LC}(e_i) < \text{LC}(e_j)$ does not imply $e_i < e_j$
  - for instance, $(1,1) < (1,3)$, but events a and e are concurrent

![Diagram showing the events and their timestamps on different processes](image-url)
Vector Timestamps

- $P_i$’s clock is a vector $VT_i[]$

- $VT_i[i] = \text{number of events } P_i \text{ has stamped}$

- $VT_i[j] = \text{what } P_i \text{ thinks number of events } P_j \text{ has stamped } (i \neq j)$
Vector Timestamps (cont.)

Initialization

– the vector timestamp for each process is initialized to (0,0,…,0)

Local event

– when an event occurs on process \( P_i \),

\[
VT_i[i] \leftarrow VT_i[i] + 1
\]

• e.g., at processor 3, \( (1,2,1,3) \rightarrow (1,2,2,3) \)
Vector Timestamps (cont.)

Message passing

– when $P_i$ sends a message to $P_j$, the message has timestamp $t[] = VT_i[]$

– when $P_j$ receives the message, it sets $VT_j[k]$ to max ($VT_j[k], t[k]$), for $k = 1, 2, \ldots, N$

• e.g., $P_2$ receives a message with timestamp (3,2,4) and $P_2$’s timestamp is (3,4,3), then $P_2$ adjust its timestamp to (3,4,4)
Comparing Vectors

- $\mathbf{VT}_1 = \mathbf{VT}_2$ iff $\mathbf{VT}_1[i] = \mathbf{VT}_2[i]$ for all $i$

- $\mathbf{VT}_1 \leq \mathbf{VT}_2$ iff $\mathbf{VT}_1[i] \leq \mathbf{VT}_2[i]$ for all $i$

- $\mathbf{VT}_1 < \mathbf{VT}_2$ iff $\mathbf{VT}_1 \leq \mathbf{VT}_2$ & $\mathbf{VT}_1 \neq \mathbf{VT}_2$
  - for instance, $(1, 2, 2) < (1, 3, 2)$
Vector Timestamp Analysis

- Claim: $e \rightarrow e'$ iff $e.VT < e'.VT$

The diagram illustrates the relationships and timestamps among three processes: $P_1$, $P_2$, and $P_3$. The vectors $m_1$ and $m_2$ represent messages exchanged between these processes. The timestamps are shown as vectors in the diagram:

- $P_1$: $[1,0,0] \rightarrow [2,0,0]$
- $P_2$: $[2,1,0] \rightarrow [2,2,0]$
- $P_3$: $[0,0,1] \rightarrow [0,0,2] \rightarrow [2,2,3]$
Application: Causally-Ordered Multicasting

• For ordered delivery, we also need...
  – Multicast msgs (reliable but may be out-of-order)
  – $V_i[i]$ is only incremented when sending
  – When $k$ gets a msg from $j$, with timestamp $ts$, the msg is buffered until:
    • 1: $ts[j] = V_k[j] + 1$
    – (this is the next timestamp that $k$ is expecting from $j$)
    • 2: $ts[i] \leq V_k[i]$ for all $i \neq j$
    – ($k$ has seen all msgs that were seen by $j$ when $j$ sent the msg)
Causally-Ordered Multicasting

Message $a$ arrives at P2 before the reply $r$ from P3 does.
The message \( a \) arrives at P2 after the reply from P3; The reply is not delivered right away.
Ordered Communication

• Totally ordered multicast
  – Use Lamport timestamps

• Causally ordered multicast
  – Use vector timestamps