Today

- Physical clocks
- Logical clocks
Events, process states and clocks

- A distributed system – a collection $P$ of $N$ single-threaded processes without shared memory
  - Each process $p_i$ has a state $s_i$
  - Each executes a series of actions – send, receive, transform state

- Events
  - Event – The execution of a single action
  - All events in a process can be placed in a total ordering $\rightarrow_i$
  - $e \rightarrow_i e'$ iff $e$ is an event that occurs after $e'$ in $p_i$
  - History of a process $p_i$ $\text{history}(p_i) = h_i =< e_i^0, e_i^1, e_i^2, ... >$

- How do we order the history of multiple processes?
  - … but first
Events, process states and clocks

- *Within a single process, we can order its events, but can we timestamp them?*

- Computers have their own hardware-based clock, \( C_i \), which can be used to assign timestamps to events.

- Clock is based on a counting of oscillation of a crystal at a given frequency – stored in some register, say \( H_i \).

- The OS reads this value, scales it and adds an offset to compute a software clock (\( H_i \) is value of \( H \) at time \( t \)):
  \[ C_i(t) = \alpha H_i(t) + \beta \]
Events, process states and clocks

- Clocks tend to drift and do so at different rates.
- **Skew** – Instantaneous difference between the readings of two clocks.
- **Clock drift** – The clocks’ underlying oscillators are subject to physical variations that make them drift from each other.
- **Drift rate** – Change in the offset between a clock and a nominal perfect reference clock per unit of time.
  - For common quartz-crystal based clocks, \( \sim 10^{-6} \text{ sec/sec} \) or 1 second every 11.6 days.
Physical clocks

- From solar to atomic time
  - Before, time was measured astronomically (solar time)
  - Since 1967, atomic clocks based on # of transitions/sec of cesium 133 (Cs133) atom
    - Drift rate of one part in $10^{13}$

- Universal Coordinated Time (UTC)
  - Currently, real time is avg of ~50 cesium-clocks
  - Broadcast through short wave radio (WWV in the US) & satellite (GPS)

- We want to distribute this to a bunch of machines
  - Each runs its own timer, keeping a clock $C_p(t)$ ($t$ being UTC)
  - Given a maximum drift rate $r$ ($1 - r \leq dC/dt \leq 1 + r$)
  - To never let two clocks differ by more than $d \Rightarrow$ synchronize at least every $d/(2r)$ seconds
Clock synchronization

- To order distributed events, keep clocks synchronized
- Two synchronization modes
  - Internal – Clocks must agree within a bound $d$
  - External – Clocks must be accurate respect to a UTC source within a bound $d$
- Internally synchronized $\neq$ externally synchronized
  - But if the system is externally synchronized with bound $d$, the it is internally synchronized with bound $2d$
Correctness of clocks

- If drift rate falls below a known bound $p$
  - $(1 - p)(t' - t) \leq H(t') - H(t) \leq (1+p)(t' - t)$
- If it satisfies \textit{monotonicity} – $t' > t \Rightarrow C(t') > C(t)$
  - We can still adjust clocks changing $\alpha$ and $\beta$ in $C_i(t) = \alpha H_i(t) + \beta$
- Both,
  - Monotonicity
  - Drift rate is bound in between synchronization points (but can jump ahead at those points)
- A clock’s crash failure – clock stops ticking altogether
  - Any other failure is an arbitrary failure (e.g., Y2K bug, from 1999 to 1900!)
Clock synchronization – External

- Cristian’s approach (1989)
- A time server that gets a signal from a UTC source
- Others ask server for accurate time at least once every \( d/(2r) \) seconds (\( d \) is the bound)
- While asynchronous, rtts are typically short
  - Must estimate rtt, including interrupt handling, msg processing
  - Cristian describes the algorithm as \textit{probabilistic}
- Faults
  - Single server, single point of failure
    - Cristian’s suggestion: a group of synchronized servers
  - Faulty server, not Cristian’s problem
    - If \( f \) is the number of faulty clocks out of \( N \), you need \( N > 3f \) clocks for the others to achieve agreement
Clock synchronization – Internal

- Berkeley’s algorithm by Gusella and Zatti (1989)
- A coordinator computer, master, periodically polls other machines

- Master calculates a fault-tolerant avg after adjusting for transfer time
  - Average is computed among clocks that don’t differ from the others by more than some given amount
- Tells all how to adjust their clocks (+/-)
Clock synchronization – External

- For Internet synchronization - Network Time Protocol (NTP, Mills 1995)
- Primary servers directly connected to time sources
- Secondary servers synchronized with others servers
- Servers synchronize with others in one of three modes
  - Multicast – For fast LANs
  - Procedure-call – ~Cristian’s, for higher synch or when there’s no multicast
  - Symmetric – Pair of machines exchanged msgs with timing info
Back in 5’
Logical clocks

- We typically assume clock synchronization is related to real time, not necessarily.
- We have seen (Berkeley algorithm) clocks can agree on a current time without this having to be *the* real time.
- Actually
  - In many situations all that matters is that two nodes agree on the order of events.
  - If two nodes do not share events, i.e. they don’t interact, they don’t have to be in synch \(\Rightarrow\) Logical clocks.
Happened-before relationship

- HB1: If $a$ and $b$ are two events in the same process, and $a$ comes before $b$, then $a \rightarrow b$
- HB2: If $a$ is the sending of a message, and $b$ is the event of receiving that message, then $a \rightarrow b$
- HB3: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

The happened-before (or [potential] causal precedence) relation on the set of events in a distributed system:
Happened-before relationship – notes

- This introduces a partial ordering of events in a system with concurrently operating processes
  - If $x$ and $y$ happen in two processes that do not exchange messages, then neither $x \rightarrow y$ nor $y \rightarrow x$
  - $x$ and $y$ are concurrent
- *What happen with communication through other channels?* e.g., phone
- *If $x \rightarrow y$, does it mean $x$ cause $y$?*

![Diagram showing happened-before relationship between processes and messages](image)
Lamport clock

- How to maintain a global view on system’s behavior that is consistent with the happened before relation?
- Attach a timestamp $C(e)$ to each event $e$, satisfying the following properties:
  - P1: If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then $C(a) < C(b)$
  - P2: If $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) < C(b)$

- How to attach a timestamp to an event when there’s no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process
Lamport clock

- Each process $p_i$ maintains a local counter $C_i$ and adjusts this counter according to the following rules:
  1. For any two successive events that take place within $p_i$, $C_i$ is incremented by $d$ (let’s say $d = 1$)
  2. When $p_i$ sends a message $m_i$, it includes a timestamp $ts(m) = C_i$
  3. Whenever $p_j$ receives $m$, $p_j$ adjusts its local counter $C_j$ to $max(C_j, ts(m))$; then executes step 1 before passing $m$ to the application

- Property 1 is satisfied by (1)
- Property 2 by (2) and (3)
- Note: To impose total ordering (instead of partial), attach process ID
Lamport timestamps – an example
From Lamport to vector clocks

- With Lamport’s clocks – if \( x \rightarrow y \), \( C(x) < C(y) \), but if \( C(x) < C(y) \), we can’t infer \( x \) causally preceded \( y \)
  - Why? Local and global logical clock are all squashed into one, loosing all causal dependency info among events at different processes

Is the sending of \( m_2 \) (\( C(b) = 20 \)) causally related to receiving of \( m_1 \) (\( C(c) = 16 \))?
Vector clocks

- Vector clock for a system with $N$ processes
  - An array of $N$ integers
  - Processes piggyback vector timestamps on each message

- Rules for updating clocks
  - Just before $p_i$ sends a message $m$,
    1. It adds 1 to $V_i[i]$, and
    2. Sends $V_i$ along with $m$ as vector timestamp $vt(m)$
  - When a $p_j$ receives a message $m$ that it received from $p_i$ with vector timestamp $ts(m)$, it
    1. updates each $V_j[k]$ to $\max\{V_j[k], ts(m)[k]\}$ for $k = 1 \ldots N$
    2. increments $V_j[j]$ by 1
Vector clocks – an example

Physical time

$\mathbf{p}_1$

(1,0,0)  (2,0,0)

$a$  $b$

$m_1$

$\mathbf{p}_2$

(2,1,0)  (2,2,0)

c  d

$m_2$

$\mathbf{p}_3$

(0,0,1)  (2,2,2)

e  f
Vector clocks

- For process $p_i$ with vector $V_i[1..n]$,  
  - $V_i[i]$ number of events that have taken place at process $p_i$  
  - $V_i[j]$ number of events that $p_i$ knows have taken place at process $p_j$ (i.e., that have potentially affected $p_i$)

- Comparing vector timestamps  
  - $V = V'$ iff $V[j] = V'[j]$ for $j = 1 .. N$  
  - $V \leq V'$ iff $V[j] \leq V'[j]$ for $j = 1 .. N$  
  - If not ($V < V'$) and not ($V > V'$) (i.e., sometimes $V[j] > V'[j]$ and sometimes smaller) – then $V \parallel V'$

- If events $x$ and $y$ occurred at $p_i$ and $p_j$ with vectors $V$ and $V'$  
  - $x \rightarrow y \iff V[i] < V'[i]$  
  - Otherwise $x \parallel y$
Vector clocks – an example

\[p_1\]
- (1,0,0) \(a\)
- (2,0,0) \(b\)
- (2,1,0) \(m_1\)
- (2,2,0) \(c\)
- (2,2,2) \(f\)

\[p_2\]
- (0,0,1) \(e\)
- (2,1,0) \(c\)
- (2,2,0) \(d\)

\[p_3\]
- (0,0,1) \(e\)
- (2,2,2) \(f\)

Physical time

\(c \parallel e\), since \(V_c \leq V_e\) nor \(V_e \leq V_c\)

\[a \rightarrow f, \; so \; V_a \leq V_f\]
Summary

- Synchronization is about doing the right thing at the right time …
- What’s the right time?
  - An issue when you don’t share clocks