Time

To do ...

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

- A distributed system
  - A collection $P$ of $N$ single-threaded processes ($p_i$, $i = 1, 2, \ldots N$) without shared memory
  - The processes in $P$ communicate via msg exchanges

- Each process $p_i$
  - has a state $s_i$ (that transforms as it executes)
  - executes a series of actions – send, receive, transform state

  - Event – The execution of a single action
Events, process states and clocks

- **Events**
  - 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 how 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 $H_i$.

- OS reads value $H_i$, scales it ($\times \alpha$) and adds an offset ($\beta$) to compute a software clock ($H_i(t)$ is value of $H_i$ 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.
- **Clock drift** – Clocks’ 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, $\sim 10^{-6}$ sec/sec or 1 second every 11.6 days.
- **Skew** – Instantaneous difference between the readings of two clocks.

![Diagram showing clock time (C) vs. real time (t) with different drift rates: fast clock, perfect clock, and slow clock.]
Physical clocks

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

- Universal Coordinated Time (UTC)
  - Currently, real time is avg of ~50 cesium-clocks
  - Since days are getting longer, add leap second to keep it in synch with astronomical time
  - Broadcast through short wave radio (WWV in the US) and satellite (GPS)
Ordering and clock synchronization

- To order distributed events, keep clocks in different machines synchronized

- We want to distribute this to the set 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 $\delta \Rightarrow$ synchronize at least every $\delta/(2r)$ seconds
Clock synchronization

- Two synchronization modes
  - Internal – Clocks must agree within a bound $\delta$
    \[ |C_i(t) - C_j(t)| < \delta \quad \text{for} \quad i, j = 1, 2, \ldots N \]
  - External – Clocks must be accurate respect to a UTC source within a bound $\delta$
    \[ |S(t) - C_j(t)| < \delta \quad \text{for} \quad i = 1, 2, \ldots N \]

- Internally synchronized $\not\equiv$ externally synchronized
  - But if the system is externally synchronized with bound $\delta$, it is internally synchronized with bound $2\delta$
Correctness of clocks

- Correctness of a hardware clock $H$
  - If drift rate $< \text{known bound } p$, error in measuring the time interval between real time $t$ and $t'$ is bounded
    \[(1 - p)(t' - t) \leq H(t') - H(t) \leq (1+p)(t' - t)\]

- For software clocks, a weaker rule may suffice
  - If it satisfies *monotonicity*
    \[t' > t \implies C(t') > C(t)\]
  - We can still adjust clocks changing $\alpha$ and $\beta$ in
    \[C_i(t) = \alpha H_{i(t)} + \beta\]
Correctness of clocks

• A hybrid condition,
  – Monotonicity and
  – Drift rate is bounded between synchronization points, but can jump ahead at those points

• Otherwise, a faulty clock
  – A clock’s crash failure – clock stops ticking altogether
  – Any other failure is an arbitrary failure (e.g., Y2K bug, from 1999 to 1900!)

• Note that clocks don’t have to be accurate to be correct
Clock synchronization – External

- Cristian’s approach (1989)
- A time server gets signal from a UTC source
- Others ask server for accurate time with some periodicity
- While asynchronous, rtts are typically short
  - Must estimate rtt, including interrupt handling, msg processing
  - Cristian describes the algorithm as *probabilistic*
Clock synchronization – External

- Setting the time based on source
  - Set time to $t$ (time when received the reply from server) + $\frac{T_{\text{round}}}{2}$ (time to get back)
  - Accuracy, assuming $\text{min}$ RTT can be estimated
    - Earliest time $t+\text{min}$, latest time $t+T_{\text{round}} - \text{min}$
    - Width $T_{\text{round}} - 2\text{min}$, so accuracy is $\pm(\frac{T_{\text{round}}}{2} - \text{min})$

- Issues with Cristian’s approach
  - 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)
- A logical hierarchy of servers
  - Primary servers directly connected to time sources
  - Secondary servers synchronized with others servers
  - The logical hierarchy – synchronization subnet
- 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
  - All over UDP

![Diagram showing a logical hierarchy of servers with primary servers at the top, secondary servers in the middle, and clients at the bottom.](diagram.png)
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 ➔ Logical clocks
What’s all this for?

- **Causality**
- The causal precedence relation induces a partial order on the events of a distributed computation
- If clocks were synchronized, you could infer (potential) causality
- With that knowledge
  - Ensure liveness and fairness in mutual exclusion
  - Maintain consistency in replicas
  - Build a deadlock-detection algorithm
  - Build a consistent state / snapshot
A collection $P$ of $N$ single-threaded processes ($p_i$)

- Communicating through message exchange

- Each process $p_i$
  - has a state $s_i$ (that transforms as it executes)
  - executes a series of actions – *send, receive, transform state*
  - An event – the execution of a single action
The happened-before (or [potential] causal precedence) relation on the set of events in a distributed system:

- 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 $a \rightarrow c$
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, $x || y$
  - So, for any two events $x$ and $y$, $x \rightarrow y$, $y \rightarrow x$ or $x || y$

- What happen with communication through other channels? e.g., phone

- If $x \rightarrow y$, does it mean $x$ cause $y$?
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 $e$, so that:
  1. If $a$ and $b$ are events in the same process, if $a \rightarrow b$, then $C(a) < C(b)$
  2. If $a$ corresponds to sending a msg $m$, and $b$ to receiving it, then $C(a) < C(b)$

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

*Monotonicity property – clock consistency condition*

- 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: From partial to total ordering, attach process ID
  - Since they are not causally related you can order them using any criteria without impacting $\rightarrow$
Lamport timestamps – an example
With Lamport’s clocks
  - if \( x \rightarrow y \), \( C(x) < C(y) \), but if \( C(x) < C(y) \), we can’t infer \( x \rightarrow y \)
  
  - No strongly consistent
  - Why? Local and global clocks are all squashed into one, loosing all causal dependency info among events at different processes

When \( p_2 \) receives \( m_1 \), it forgets the timestamp of the latest event at \( p_1 \) on which it depends

\[ C(g) < C(c) \text{ but does } g \rightarrow c? \]
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 $m$ 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

Vector clocks are a way to represent the states of a distributed system. Each vector clock is a tuple of non-negative integers. The diagram shows three vector clocks:

- $p_1$: $(1,0,0)$, $(2,0,0)$, $(3,0,0)$
- $p_2$: $(2,1,0)$, $(2,2,0)$
- $p_3$: $(0,0,1)$, $(2,2,2)$

The arrows $m_1$ and $m_2$ represent the updates in the system.
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 $V'$
  - $x \rightarrow y \Leftrightarrow V[i] < V'[i]$
  - Otherwise $x \parallel y$
Vector clocks – an example

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

(2,1,0) \ldots (0,0,1)

\[ a \rightarrow f, \text{ so } V_a \leq V_f \]

(1,0,0) \leq (2,2,2)
More on logical clocks ...

- The topic doesn’t end there
  - Matrix times, everybody knows what everyone else knew when they sent a message
  - Useful, for instance, to discard obsolete info in replicated DB

- How to efficiently implement logical clocks?
  - Example – send only updates, but have to keep an additional vector with what you knew in the last exchange
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