



Clocks in Distributed System



Types of Clocks

- Physical Clocks
 - Tied to the notion of real time
 - Can be used to order events, find time difference between two events,..
- Logical Clocks
 - Derived from the notion of potential cause-effect between events
 - Not tied to the notion of real time
 - Can be used to order events
 - Different types
 - Lamports Logical Clock
 - Vector Clocks
 - ...



Physical Clocks

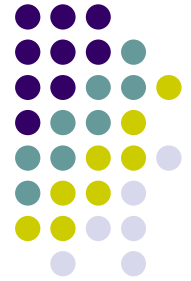
- Each node has a local clock used by it to timestamp events at the node
- Local clocks of different nodes may vary
- Need to keep them synchronized (**Clock Synchronization Problem**)
- Perfect synchronization not possible because of inability to estimate network delays exactly
- But still useful, synchronization requirements vary
 - Kerberos: requires synchronization of the order of minutes
 - GPS: requires synchronization of the order of milliseconds

Clock Synchronization



- Internal Synchronization
 - Requires the clocks of the nodes to be synchronized to within a pre-specified bound
 - However, the clock times may not be synchronized to any external time reference, and can vary arbitrarily from any such reference
- External Synchronization
 - Requires the clocks to be synchronized to within a pre-specified bound of an external reference clock

How Computer Clocks Work



- Computer clocks are crystals that oscillate at a certain frequency
- Every H oscillations, the timer chip interrupts once (clock tick).
 - Resolution: time between two interrupts
- The interrupt handler increments a counter that keeps track of no. of ticks from a reference in the past (epoch)
- Knowing no. of ticks per second, we can calculate year, month, day, time of day etc.

Why Clocks Differ: Clock Drift



- Unfortunately, period of crystal oscillation varies slightly
- If it oscillates faster, more ticks per real second, so clock runs faster; similar for slower clocks
- For machine p , when correct reference time is t , let machine clock show time as $C = C_p(t)$
- Ideally, $C_p(t) = t$ for all p , t
- In practice,
$$1 - \rho \leq dC/dt \leq 1 + \rho$$
- $\rho = \text{max. clock drift rate}$, usually around 10^{-5} for cheap oscillators
- Drift \Rightarrow Skew between clocks (difference in clock values of two machines)

Resynchronization



- Periodic resynchronization needed to offset skew
- If two clocks are drifting in opposite directions, max. skew after time t is $2\rho t$
- If application requires that clock skew $< \delta$, then resynchronization period
$$r < \delta / (2 \rho)$$
- Usually ρ and δ are known

Cristian's Algorithm



- One m/c acts as the time server
- Each m/c sends a message periodically (within resync. period r) asking for current time
- Time server replies with its time
- Sender sets its clock to the reply
- Problems:
 - message delay
 - time server time is less than sender's current time



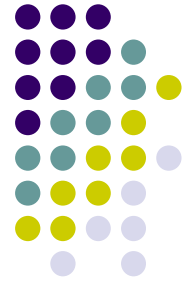
- Handling message delay: try to estimate the time the message with the timer server's time took to reach the sender
 - Measure round trip time and halve it
 - Make multiple measurements of round trip time, discard too high values, take average of rest
 - Make multiple measurements and take minimum
 - Use knowledge of processing time at server if known to eliminate it from delay estimation (How?)
- Handling fast clocks
 - Do not set clock backwards; slow it down over a period of time to bring in tune with server's clock

Berkeley Algorithm



- Centralized as in Cristian's, but the time server is active
- Time server asks for time of other m/cs at periodic intervals
- Other machines reply with their time
- Time server averages the times and sends the adjustments (difference from local clock) needed to each machine
 - Adjustments may be different for different machines
 - Why do we send adjustments, and not the new absolute clock value?
- M/cs sets their time (advances immediately or slows down slowly) to the new time

Some Points to Note



- Cristian's algorithm
 - Can also give external synchronization if the time server is sync'ed with external clock reference
 - Requires a special node with a time source
 - Prone to failure of the central server
- Berkeley's algorithm
 - Can be used for internal synchronization only
 - No separate time source needed, one of the nodes can be elected as leader and then act as the time server
 - Note that the actual time of the central server does not matter, enough for it to tick at around the same rate as other clocks to compute average correctly (why?)
 - Failures are handled by electing a new leader from the remaining machines
- What is the max. difference between two clocks after the synchronization?



- None of them are scalable to large systems
 - Load on the central server
 - Variance in message delay in large networks
- Works well in LANs with small number of machines

External Synchronization with Real Time

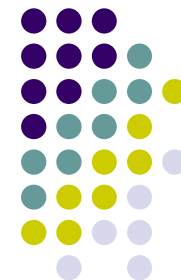


- Clocks must be synchronized with real time
- But what is “real time” anyway?

Measurement of time



- Astronomical
 - traditionally used
 - based on earth's rotation around its axis and around the sun
 - solar day : interval between two consecutive transits of the sun
 - solar second : $1/86,400$ of a solar day
 - period of earth's rotation varies, so solar second is not stable
 - mean solar second : average length of large no of solar days, then divide by 86,400



- Atomic
 - Based on the transitions of Cesium 133 atom
 - 1 sec. = time for 9,192,631,770 transitions
 - about 50+ labs maintain Cesium clock
 - International Atomic Time (TAI) : mean no. of ticks of the clocks since Jan 1, 1958
 - Highly stable
 - But slightly off-sync with mean solar day (since solar day is getting longer)
 - A leap second inserted occasionally to bring it in sync.
 - Resulting clock is called UTC – Universal Coordinated Time



- UTC time is broadcast from different sources around the world, ex.
 - National Institute of Standards & Technology (NIST) – runs WWV radio station, anyone with a proper receiver can tune in
 - United States Naval Observatory (USNO) – supplies time to all defense sources
 - National Physical Laboratory in UK
 - Satellites
 - Many others
 - Accuracies can vary (< 1 milliseconds to a few milliseconds)

Synchronizing with UTC Time



- Can use a Cristian-like algorithm with the time server sync'ed to a UTC source
- Not scalable for internet-scale synchronization
- Solution: Use a hierarchical approach

NTP : Network Time Protocol



- Protocol for time sync. in the internet
- Hierarchical architecture
 - Primary time servers (stratum 1) synchronize to national time standards via radio, satellite etc.
 - Most accurate
 - Secondary servers and clients (stratum 2, 3,...) synchronize to primary servers in a hierarchical manner (stratum 2 servers sync. with stratum 1, stratum 3 with stratum 2 etc.)
 - Lower stratum means more accurate



- Reliability ensured by synchronizing with redundant servers
- Communication by multicast (usually within LAN servers), symmetric (usually within multiple geographically close servers), or client server (to higher stratum servers)
- Complex algorithms to combine and filter times
- Sync. possible to within tens of milliseconds for most machines
- But just a best-effort service, no guarantees
- <http://www.ntp.org> for more details



Ordering Events

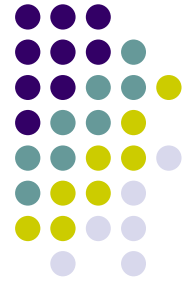
- Given two events in a distributed system (at same or different nodes), can we say if one happened **before** another or not?
 - Common requirement, for example, in applying updates to replicas in a replicated system
- Physical clocks can be used with synchronization in many cases
- Fails to order when events happen too fast (faster than the maximum possible skew between two clocks)
- Are physical clocks needed at all for ordering events?



Causality and Ordering

- Can what happened in one event at one node affect what happens in another event in the same or another node?
 - Because if not, ordering them is not important
- Can we capture this notion of **causality** between events and build a local clock around it?
 - Use the causality to synchronize the local clocks
 - No relation to time synchronization as we have seen so far, no real notion of time

Lamport's Ordering



Lamport's *Happened Before* relationship:

- For two events x and y , $x \rightarrow y$ (x *happened before* y) if
 - x and y are events in the same process and x occurred before y
 - x is a send event of a message m and y is the corresponding receive event at the destination process
 - $x \rightarrow z$ and $z \rightarrow y$ for some event z



- $x \rightarrow y$ implies x is a *potential* cause of y
 - x can affect y
 - Does not mean that x must affect y , just that it can
 - But y cannot affect x (i.e. y cannot be a potential cause of x)
- Causal ordering : *potential* dependencies
- “Happened Before” relationship causally orders events
 - If $x \rightarrow y$, then x causally affects y
 - If $x \nrightarrow y$ and $y \nrightarrow x$, then x and y are concurrent
($x \parallel y$)

Lamport's Logical Clock



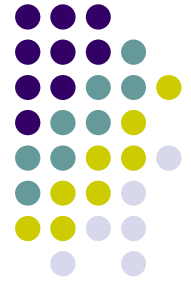
- Each process i keeps a clock C_i
- Each event x in i is timestamped $C(x)$, the value of C_i when x occurred
- C_i is incremented by 1 for each event in i
- In addition, if x is a send of message m from process i to j , then on receive of m ,
$$C_j = \max(C_j + 1, C(x)+1)$$
- Increment amount can be any positive number not necessarily 1



Points to Note

- if $x \rightarrow y$, then $C(x) < C(y)$
- Total ordering possible by arbitrarily ordering concurrent events by process numbers (assuming process numbers are unique)
- Frequent communication between nodes brings their logical clocks closer (sync'ed)
- Infrequent communication between nodes may make their logical clocks very different
 - Not a problem, as less communication means less chance of events at one node affecting events at another node

Using the Clock



- Given two events x and y at processes i and j :
 - Order x before y if
 - $C(x) < C(y)$, or
 - $C(x) = C(y)$ and $i < j$
 - This may order two concurrent events also, but that's fine as then the order does not matter for causality anyway
 - If $x \rightarrow y$, then y will never be ordered before x

Limitation of Lamport's Clock



- $x \rightarrow y$ implies $C(x) < C(y)$ but $C(x) < C(y)$ doesn't imply $x \rightarrow y$!!

So not a true clock !!

Though not a big limitation in many applications

Solution: Vector Clocks



- C_i is a vector of size n (no. of processes)
- $C(a)$ is similarly a vector of size n
- Update rules:
 - $C_i[i]++$ for every event at process i
 - if x is send of message m from i to j with vector timestamp t_m , on receive of m :
$$C_j[k] = \max(C_j[k], t_m[k]) \text{ for all } k$$



- For events x and y with vector timestamps t_x and t_y ,
 - $t_x = t_y$ iff for all i , $t_x[i] = t_y[i]$
 - $t_x \neq t_y$ iff for some i , $t_x[i] \neq t_y[i]$
 - $t_x \leq t_y$ iff for all i , $t_x[i] \leq t_y[i]$
 - $t_x < t_y$ iff ($t_x \leq t_y$ and $t_x \neq t_y$)
 - $t_x \parallel t_y$ iff ($t_x \not\leq t_y$ and $t_y \not\leq t_x$)



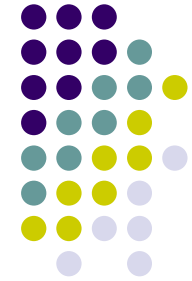
- $x \rightarrow y$ if and only if $t_x < t_y$
- Events x and y are causally related if and only if $t_x < t_y$ or $t_y < t_x$, else they are concurrent

Application of Vector Clocks: Causal Ordering of Messages

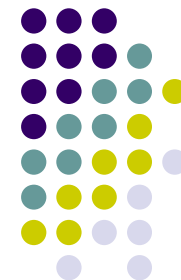


- Different message delivery orderings
 - Atomic: all message are delivered by all recipient nodes in the same order (any order possible, but same)
 - Causal: For any two messages m_1 and m_2 , if $\text{send}(m_1) \rightarrow \text{send}(m_2)$, then every recipient of m_1 and m_2 must deliver m_1 before m_2 (but messages not causally related can be delivered by different nodes in different order)
 - FIFO Order: For any two messages m_1 and m_2 **from the same node**, if m_1 is sent before m_2 , then every recipient of m_1 and m_2 must deliver m_1 before m_2 (but messages from different nodes can be delivered by different nodes in different order)
 - Atomic Causal (Atomic and Causal), Atomic FIFO (Atomic and FIFO)
- “deliver” – when the message is actually given to the application for processing, not when received by the network

Birman-Schiper-Stephenson Protocol for Causal Order Broadcast (CBCAST)



- To broadcast m from process i , increment $C_i[i]$, and timestamp m with $VT_m = C_i$
- When $j \neq i$ receives m , j delays delivery of m until
 - $C_j[i] = VT_m[i] - 1$ and
 - $C_j[k] \geq VT_m[k]$ for all $k \neq i$
 - Delayed messages are queued in j sorted by vector time. Concurrent messages are sorted by receive time.
- When m is delivered at j , C_j is updated according to vector clock rule



- First condition says that j has delivered all previous broadcasts sent by i before delivering m
 - This is the set of all messages at i that can causally precede m
- Second condition says j has delivered at least as many (may be more) broadcasts sent by k as delivered by i ($k \neq i, j$) when i sent m
 - This is the set of all messages at nodes $\neq i$ that can causally precede m
- So both conditions true means j has delivered all messages that causally precedes m



Problem of Vector Clock

- Message size increases since each message needs to be tagged with the vector
- Size can be reduced in some cases by only sending values that have changed (
- Can also send only a scaler to keep track of direct dependencies only, with indirect dependencies computed when needed
 - Tradeoff between message size and time