

THE UNIVERSITY *of* EDINBURGH

Distributed Systems Fall 2026

Yuvraj Patel

Disclaimer: Slides prepared using multiple sources (UW-Madison – Remzi, Andrea, Mike; Cambridge – Martin Kleepman; Distributed Systems book by MVS/ AST; University of Edinburgh – Yuvraj Patel)

Today's Agenda

Time & Clocks

- Physical, Logical, Vector

Distributed Snapshots

Need for Clocks & Time

Distributed systems need to rely on time for

- Scheduling, Timeouts, failure detectors, etc.
- Performance measurements, statistics, profiling
- Log files & databases (ensure ordering)
- Caching

Physical Clock & Clock Drift

Clocks show time → count number of seconds elapsed

Clocks relies on quartz oscillators and count cycles to measure time

Clock may run little bit fast or slow

Multiple factors such as quality, environmental variables(temperature)

Drift measured in parts per million

- 1 ppm → maximum deviation of 1 ticks per million ticks

Atomic clocks more accurate than mechanical clocks

- 1 in 10^{-14} (1 second in 3 million years)
- Costly

Clock Synchronization

Computers can track physical time with a quartz clock

Clock error increases due to clock drift

Clock Skew – Difference between two clocks at a point in time

Analogy

- Clock drift – Difference in speed of two vehicles on the road
- Clock skew – Distance between two vehicles on the road
- Clock drift causes skew to increase
  - If faster vehicle is ahead, it will drift away
  - If faster vehicle is behind, it will catch up and eventually drift away

Clock Synchronization is necessary

Clock Synchronization (contd…)

Two approaches – External vs. Internal

External Synchronization

- Each process C(i)'s clock is within a bound D of a well-known clock S external to the group
- $|C(i) - S| < D$ always
- S may be connected to UTC or atomic clock
- Example: Cristian's algorithm, NTP

Internal Synchronization

- Every pair of processes in a group have clocks within bound D
- $|C(i) - C(j)| < D$ always and for all the processes I, j
- Example: Berkeley algorithm

Network Time Protocol

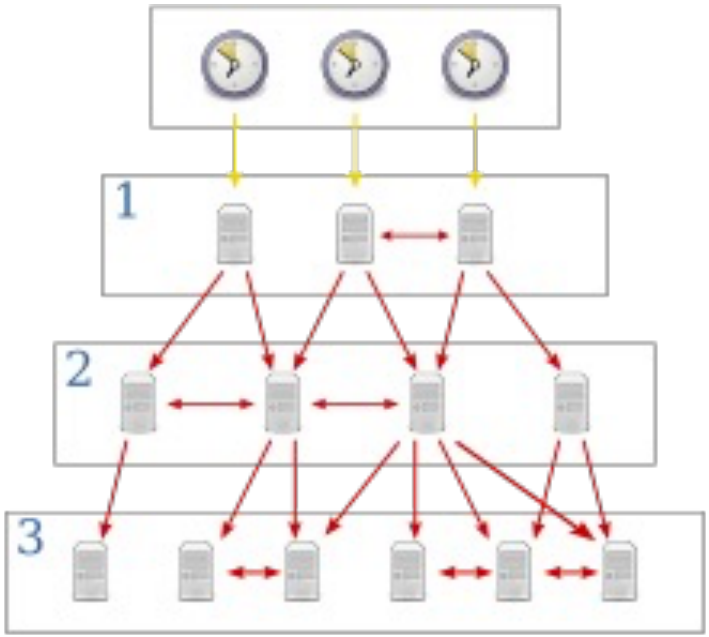
NTP commonly used for clock synchronization

Hierarchy of clock serves arranged in strata

Stratum 0 → Atomic clocks

Stratum 1 → Primary time servers; Sync with stratum 0; May sync with each other

Stratum n + 1 → Sync with stratum n and each other



Source:Wikipedia

Network Time Protocol (contd…)

Yet....

We still have

- A non-zero error
- Cannot get rid of the error

Can we avoid synchronization clocks altogether?

- How do we order events then?

Ordering of Events

Clock synchronization is one approach to order events

Is there another way where we avoid absolute time?

- Absolute time may not be accurate
- Absolute time may be less important

Can logical clocks work where we do not need synchronization?

For ordering, we need causality to be determined

Event A happens causally before another event B, then

- timestamp(A) < timestamp(B)

Ordering of Events (contd…)

Ordering of Events

Clock synchronization is one approach to order events

Is there another way where we avoid absolute time?

- Absolute time may not be accurate
- Absolute time may be less important

Can logical clocks work where we do not need synchronization?

For ordering, we need causality to be determined

Event A happens causally before another event B, then

- timestamp(A) < timestamp(B)

Happens-Before Relation

An event is something happening at one node

- Executing an instruction
- Sending a message
- Receiving a message

Event A happens before Event B (written as A→B) iff:

- A and B occurred on the same node, and A occurred before B in that node's local execution order
- Event A is the sending of some message M, and event B is the receipt of that same message M
- There exists an event C such that A→C and C→B (Transitive relationship)

Happens-before relation is a partial order

- A→B or B→A is not possible
- In that case, A and B are concurrent (written A || B)

Happens-Before Relation Example

Causality

Relativity in Physics

- Information is not possible to travel faster than speed of light

When a→b, then a might have caused b

When a ||b, we know that a cannot have caused b

Happens-before relation encodes potential causality

Let $\prec$ (precede) be a strict total order on events

If (a→ b) ➔($a \prec b$) then $\prec$ is a causal order

Logical Clocks

Key idea

- Physical clocks inconsistent with causality
- If two machines do not interact, no need for synchronization
- Processes need to agree on the ordering of events instead of the time at which they occurred

Logical clocks

- (a → b) ➔ Timestamp(a) < Timestamp(b)

Lamport Clocks

Assign logical timestamp to each event

Timestamp obeys causality

Rules/Algorithm

- Each process maintains a counter value
- On init, counter value is 0
- Each process increments its counter when a send or an instruction is executed.
- A send message event carries the counter (timestamp)
- For a receive message event, the recipient process will update its local counter max(local counter, message counter (timestamp)) + 1

Properties

- If a → b then value-of-counter(a) < value-of-counter(b)
- However, if value-of-counter(a) < value-of-counter(b), does not imply a→b
- Possible value-of-counter(a) - value-of-counter(b) for a ≠ b

Lamport Clock Example

Concurrent Events

A pair of concurrent events doesn't have a causal path from one event to another

Lamport clock not guaranteed to be ordered or unequal for concurrent events

Concurrent events are not causality related

- value-of-counter(a) < value-of-counter(b)
- Maybe a→b or a || b

Can we have logical timestamps from which we can tell if two events are concurrent or causally related?

Vector Clocks

Assume n nodes in the system, N = <N1, N2… Nn>

Vector timestamp of an event a is V(a) = <t1, t2…tn>

- ti is the number of events observed on node Ni

Each node has a current vector timestamp T

On event at node Ni, increment vector element T[i]

Attach current vector timestamp to each send message

Recipient merges message vector into its local vector

Vector Clocks (contd…)

Rules/Algorithm

- On initialization at node Ni,
  - do T = <0, 0, …0>
- On any event occurring at node Ni,
  - do T[i] = T[i] + 1
- On Send event while sending message M at node Ni,
  - do T[i] = T[i] + 1;
  - Send (T, M)
- On Receiving event message M at node Nj, do
  - T[j] = T[j] + 1
  - T[k] = max(T[k], T'[k]), for every $k \neq j$ and $k \in \{1,\ldots n\}$

Vector Clocks Example

Causality-Relation@Vector Clocks

VTa = VTb, iff VTa[i] = VTb[i], for all i = 1, …, N

VTa ≤ VTb, iff VTa[i] ≤ VTb[i], for all i = 1, …, N

Two events are causally related, iff

- VTa < VTb, i.e.
- VTa ≤ VTb AND there exists j such that 1 ≤ j ≤ N AND VTa[j] < VTb[j]

Two events VTa and VTb are concurrent, iff

- NOT (VTa ≤ VTb) AND NOT (VTb ≤ VTa)

Detecting Global Properties

Sometimes it is necessary to have a global view of the system

- Checkpointing to support restarting the system post failure
- Garbage collection of objects
- Deadlock detection
- Termination of computation
- Debugging in general

Global Snapshot

Global Snapshot = Global State

Global state comprises

- Individual state of each process in the system
- Individual state of each communication channel in the system

Capture instantaneous state of each process and the communication channels

Obvious Choice

Synchronize clocks of all processes

Ask all processes to record their states at known time t

Issues

- Time synchronization is error prone
- Cannot record the state of the messages in the communication channels

Do we need synchronization or causality is enough?

Global Snapshot – What is needed?

Need to tell each process what to record and when

Need to record contents of communication channels properly

Cannot pause the entire system or interference with the normal functioning

Collect the global state in a distributed manner

Possibly check for other useful properties on the global state

Consistent Cut

A cut is a set of cut events, one per node, each of which captures the state of the node on which it occurs

A cut C = {c1, c2, … cn} is consistent if for all the nodes there are not events ei and ej such that

- (ei → ej) AND (ej → cj) AND (ei -/-> ci)

Simply, cut C is consistent cut, iff

- for (each pair of events e, f in the system)
- event e is in the cut C, and if f --> e, then event f is also in the cut C

The cut events in a consistent cut are not causally related

- The cut is a set of concurrent events, and a set of concurrent events is a cut

Consistent States

A global state S is consistent iff it corresponds to a consistent cut

Chandy-Lamport Snapshot Algorithm Model

System Modelling

- Collection of processes each having a local state
- Collection of communication channels, either empty or have messages in-flight, between each pair of processes
- Communication channels are unidirectional and FIFO-ordered
- No failure, all messages arrive intact, exactly once
- Snapshot does not interfere with normal execution
- Each process can record its own state and the state of the incoming channels
- Only one snapshot initiator

Chandy-Lamport Snapshot Algorithm

Initiator Pi records it own state

Initiator process creates special messages called “Marker” messages

For j = 1 to N except i

- Pi sends out a Marker message on outgoing channel Cij
- Starts recording the incoming messages on each of the incoming channels at Pi: Cji (for j = 1 to N except i)

Chandy-Lamport Snapshot Algorithm

Whenever a process Pi receives a Marker message on an incoming channel Cki

- If (this is the first Marker Pi is seeing)
  - Pi records its own state first
  - Marks the state of the channel Cki as “empty”
  - for j = 1 to N except I
    - Pi sends out a Marker message on outgoing channel Cij
  - Starts recording the incoming messages on each of the incoming channels at Pi: Cji (for j = 1 to N except i and k)
- Else (Marker already seen)
  - Mark the state of the channel Cki as all the messages that have arrived on it since recording was turned on for Cki

Chandy-Lamport Snapshot Algorithm

Algorithm terminates when

- All the processes have received a Marker to record their own state
- All the process have received a Marker on all the (N – 1) incoming channels to record the state of all the channels

A central authority(server/entity) may collect all the local state of the processes and the communication channels to obtain the full global snapshot

Any run of the algorithm creates a consistent cut

Liveness & Safety Property

Correctness in a system can be seen two ways – Liveness and Safety

Liveness – guarantee that something good will happen eventually

- Not time bound, if in long run, things will work out fine
- Example: Criminal will be jailed sooner or later; Computation will complete; Consensus will arrive

Safety – guarantee that something bad will never happen

- Example: Innocent will never be jailed; Deadlock never happens; Consensus on different values for the same proposal

Difficult to guarantee both in asynchronous system

In The Language of Global States

A system moves from one global state to another, via causal steps

Liveness w.r.t a property Pr in a given state S means

- S satisfies Pr, or there is some causal path of global states from S to S' where S' satisfies Pr

Safety w.r.t a property Pr in a given state S means

- S satisfies Pr, and all global states S' reachable from S also satisfy Pr

Using Global Snapshot Algorithm

Chandy-Lamport algorithm can be used to detect global properties that are stable (stable means once true, always true)

Stable Liveness – Computation has terminated

Stable Non-Safety – There is a deadlock, an object is not referenced anymore

All stable global properties can be detected using the algorithm due to its causal correctness

Chandy-Lamport Snapshot Algorithm Example