Clock Synchronization
- Time is unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has its own system clock
  - Two clocks could differ at a given point in time (skew)
  - Clocks that agree at time $t$ might disagree later (drift)
- Makes it harder to reason about events on different systems
- Some examples:
  - Makefile: edit on one system, compile on another system
  - Kerberos leases: valid only for a certain period of time
  - Using timestamps to serialize transactions

Pair-wise synchronization: Cristian’s Algorithm
- Synchronize machines to a time server with a UTC receiver (some trusted physical clock)
- Machine $P$ requests time from server (every once in a while)
  - Receives time $t$ from server, $P$ sets clock to $t + t_{\text{reply}}$ where $t_{\text{reply}}$ is the time to send reply to $P$
  - Use $(t_{\text{req}} + t_{\text{reply}})/2$ as an estimate of $t_{\text{reply}}$
  - Improve accuracy by making a series of measurements

Berkeley Algorithm
- Used in systems without UTC receiver
  - Keep clocks synchronized with one another
  - One computer is master, others are slaves
  - Master periodically polls slaves for their times
    - Average times and return differences to slaves
  - Communication delays compensated as in Cristian’s algo
  - Failure of master $\Rightarrow$ election of a new master

Logical Clocks
- For many problems, internal consistency of clocks is important
  - Absolute time is less important
  - Use logical clocks
- Key idea:
  - Clock synchronization need not be absolute
  - If two machines do not interact, no need to synchronize them
  - More importantly, processes need to agree on the order in which events occur rather than the time at which they occurred

Logical Ordering of Events
- Two kinds of ordering:
  - If a process $p_1$ does operation $o_1$ followed by operation $o_2$, then we would like to say $o_1$ occurred before $o_2$
  - If a message is sent/received:
    - Process $p_1$ sends message $m$ (let this be operation $o_1$)
      - Process $p_2$ receives the message $m$ (let this be operation $o_2$)
    - Then $o_1$ occurred before $o_2$
  - Relations are transitive:
    - If $o_1$ occurred before $o_2$ and $o_2$ occurred before $o_3$, then $o_1$ occurred before $o_3$
Logical clocks
- Each process maintains a local counter
  - Counter is incremented for every local event (including send events)
  - Counter value is sent along with every message
- When message is received:
  - Take max of local counter and message's counter → new local counter
  - Increment local counter by one

![Logical clocks diagram](image)

Analysis of logical clocks
- If event e1 happened before e2: LC(e1) < LC(e2)
- Are we done? Are logical clocks sufficient to reason about distributed systems?

Vector Clocks
- Each process i maintains a vector V_i
  - V[i] : number of events that have occurred at i
  - V[i][j] : number of events i knows have occurred at process j
- Update vector clocks as follows
  - Local event: increment V[i]
  - Send a message: piggyback entire vector V
  - Receipt of a message: V[i][k] = max(V[i][k], V[j][k])
    - Receiver is told about how many events the sender knows occurred at another process k
    - Also V[i][j] = V[j][i]+1
- Convince yourself that if V(A)<V(B), then A precedes B

![Vector Clocks example](image)

Motivating Example for Reasoning about Global State
- Assume that we have processes interacting in a client-server mode
  - A → B
- Client makes request to server
  - Waits for response
  - While waiting for response, client simply blocks; does not satisfy requests from other nodes
  - C

Deadlock Detection
- Assume that you have a centralized server
  - It queries each node
    - Each node responds with a list of requests that are pending (requests for which a response has not been sent)
- Centralized server can then build a “waits-for” graph:
  - Cycle in graph implies deadlock
Possible Execution

Different Observation

Consistent & Inconsistent Cuts

Other Applications

Snapshot

Snapshot (2nd attempt)
**Snapshot Algorithm Example**

- **Organization of a process and channels for a distributed snapshot**

**Distributed Snapshot**

- A process finishes when
  - It receives a marker on each incoming channel and processes them all
  - State: local state plus state of all channels
  - Send state to initiator, initiator analyzes state

- Any process can initiate snapshot
  - Multiple snapshots may be in progress
    - Each is separate, and each is distinguished by tagging the marker with the initiator ID (and sequence number)

**A Different Approach**

- Monitor process does not query explicitly
- It just passively collects information
- Uses it to build an “observation”

**Delivery of messages to monitor**

- What properties do we need to satisfy in delivering messages to the monitor?

**Causal Delivery**

- A message cannot be delayed to appear after a later message
Summary so far…

- Interested in “global predicate detection”
  - Whether the state of a distributed application matches some predicated (deadlocks, termination, distributed garbage collection, etc.)
- Two approaches:
  - A centralized process sends messages to capture the current state of all processes
  - Centralized process needs to observe a “consistent cut”
  - Intuition: rely on FIFO property of channels; propagate markers along channels and save state as marker messages reach processes
  - Each process continually sends messages to centralized process when “interesting” events happen
  - Centralized process builds global state - can compute all possible global states that may or may not occur in the system

Delivery of events to centralized process

- Requirements:
  - FIFO: messages from same processor is delivered in order
  - An event e1 should be reported before e2
  - Causal properties are preserved; consistent observations are made
  - An event e1 should be reported before e2

How to deliver messages?

- Each event notification is tagged with logical clock value
- Messages are “delivered” to observing process in a manner that satisfies above properties
- Delivery manager delivers messages in increasing order of logical clock values
  - Ties are broken based on processor ids

Gap Detection

- Consider the following state:
  - Observing processor has received the following event notifications: e1, e2, e3, e4
  - Notification of e3 has been delayed
  - Gap detection problem: given two events e1 and e2, detect whether or not there is another event e3 that occurs in the middle

Gap detection using logical clocks

- Wait for a while until there is at least one undelivered observation from each process
- Deliver the event with the lowest logical clock value
- Has liveness issues:
  - Requires processors to continually send observations to observing processor
- Is there a better solution? Is there some way of deciding whether or not to delay delivery as soon as a message is received?

Global Predicate Evaluation

- Two methods:
  - Distributed snapshot initiated at arbitrary times
  - Centralized observations made using reports of all events
- Global predicates that can be evaluated using either method:
  - Deadlock detection
  - Termination detection
  - Garbage collection
- When would you use distributed snapshots and when would you use centralized observations?
Formalisms
- Denote global states by \( \Sigma \)
- For example, assume two processes
  - \( \Sigma^1 \) would refer to process 1 at state 1 and process 2 at state j
- Define a lattice of valid global states

Reachability
- We say that \( \Sigma^i \) is reachable (from \( \Sigma^j \)) if there is a path from \( \Sigma^j \) to \( \Sigma^i \) in the lattice.

Why do we care about \( \Sigma^? \)?
- Deadlock is a stable property
  - Deadlock now implies deadlock in the future
- If \( \Sigma^i \) is initial state and \( \Sigma^f \) is termination state for snapshot:
  - \( \Sigma^i \sim \Sigma^s \sim \Sigma^f \)
  - Deadlock in \( \Sigma^s \) implies deadlock in \( \Sigma^f \)
  - No deadlock in \( \Sigma^s \) implies no deadlock in \( \Sigma^i \)

The Lattice
- In \( \Sigma^{31} \) or \( \Sigma^{41} \), the predicate \( x = y - 2 \) is detected
  (Notice that it might be detected, but might never have occurred.)
- We know that \( x = y \) has occurred, but it may not be detected if tested before \( \Sigma^{32} \) or after \( \Sigma^{41} \)
- Not enough to look at one state: look at all observations instead

Possibly and Definitely
- Possibly: There exists a consistent observation \( O \) of the computation such that the predicate holds in a global state of \( O \)
- Definitely: For every consistent observation \( O \) of the computation, there exists a global state of \( O \) in which the predicate holds
Computing Possibly and Definitely

- To compute Possibly(\( \Phi \)):
  - If \( \Phi \) holds in one global state, declare Possibly(\( \Phi \)) to be true.
- To compute Definitely(\( \Phi \)):
  - Given a level, only expand those nodes that correspond to states in which \( \neg \Phi \) holds.
  - If no such state, announce Definitely(\( \Phi \)).

Building the lattice

- \( P_0 \) collects local state from each process.
- For each process, keep a sequence \( Q \) of local states in FIFO order.
- Construct global states as combination of all possible local states.

Earliest Consistent Global State

- earliest consistent global state: \( \sum_{i_1 i_2 i_3 \ldots i_n} \), latest consistent global state: \( \sum_{i_1 i_2 i_3 \ldots i_n} \).
- \( \sigma_{ij} \) represents state of processor \( i \) after \( j \) operations.

Earliest & Latest Consistent Global State

- For state \( \sigma_{13} \):
  - Earliest consistent global state: \( \sigma_{1} \), \( \sigma_{2} \), \( \sigma_{3} \).
  - Level of earliest consistent global state: \( 1 + 5 + 1 = 7 \).
  - Latest consistent global state: \( \sigma_{1} \), \( \sigma_{2} \), \( \sigma_{3} \).
  - Corresponding level: 15.

Notation and Terminology

- \( \sum_{1 2 3 \ldots i_n} \) represents the global state after \( i_1 \) operations by processor 1, \( i_2 \) operations by processor 2, etc.
- Level of \( \sum_{1 2 3 \ldots i_n} \) is \( i_1 + i_2 + \ldots + i_n \).
- \( \sigma_{ij} \) represents state of processor \( i \) after \( j \) operations.

Building the lattice

- Collect states at the monitor.
  - Store them in separate queues.
  - Constructing level by level:
    - To build level \( i \): wait until all the states required for the level are available.
    - Earliest level for \( \sigma_{13} \): 3 + 4 + 1 = 8.
    - For \( \sigma_{23} \): 5, for \( \sigma_{33} \): 9.
    - Can construct levels 1 through 5.
Building the lattice (contd.)

- Once monitor decides to build level L+1:
  - It takes all the consistent global states of level L.
  - Extends them by one extra step for some processor.
  - For example: $\sum_{i1,i2,i3,\ldots,in}^1$ is a level L global state (stored in the lattice), then construct $\sum_{i1+1,i2,i3,\ldots,in}^1$, $\sum_{i1,i2+1,i3,\ldots,in}^1$, $\ldots$, $\sum_{i1,i2,\ldots,in+1}^1$.
  - Some of these are inconsistent states: can be detected by looking at the vector clock values of local states.
  - Discard these spurious global states.

Building the lattice (contd.)

- Once the monitor has finished building level L, it can discard some of the local states from its queue.
- Consider $a^2$: latest consistent global state it belongs to is $a_1^2\ a_2^2\ a_3^2$.
- Corresponding level = 15.
- Discard $a_2^2$ after computing level 15.

What about shared memory programs?

- So far we discussed message passing programs.
- For shared memory programs:
  - How do we order events?
  - And make consistent observations?