Logical Clocks

Arvind Krishnamurthy
Fall 2003

Clock Synchronization

- Time is unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has 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, other 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 $=>$ 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 p1 does operation o1 followed by operation o2, then we would like to say o1 occurred before o2
  - If a message is sent/received:
    - Process p1 sends message m (let this be operation o1)
    - Process p2 receives the message m (let this be operation o2)
    - Then o1 occurred before o2
  - Relations are transitive:
    - If o1 occurred before o2 and o2 occurred before o3, then o1 occurred before o3
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

Analysis of logical clocks

- If event $e_1$ happened before $e_2$:
  $\text{LC}(e_1) < \text{LC}(e_2)$

- 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 \(i\)
  - \(V[j]\) : 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: \(V[k] = \max(V[k], V[i])\)
    - Receiver is told about how many events the sender knows occurred at another process \(k\)
    - Also \(V[i] = V[i]+1\)
  - Convince yourself that if \(V(A) < V(B)\), then \(A\) precedes \(B\)

Vector Clocks Example

```
   a b c d e f
p1      m1
p2      m2
p3      
```

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

Physical time
Motivating Example for Reasoning about Global State

- Assume that we have processes interacting in a client-server mode

  - Client makes request to server
    - Waits for response
    - While waiting for response, client simply blocks; does not satisfy requests from other nodes

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

- A cut is inconsistent if:
  - You include an event e2 in p2
  - Event e1 of p1 influences e2
  - But e1 is not included

Other Applications

a. Garbage collection

b. Deadlock

c. Termination
Snapshot

- Develop a simple synchronous protocol
- Refine protocol as we relax assumptions
- Initial assumptions:
  - Real time clock known to all processes
  - Message delays are bounded

Algorithm: (assume that all messages are timestamped)
- Process $P_0$ selects “$t_{ss}$”
- $P_0$ sends “take a snapshot at $t_{ss}$” to all processes
- When clock of $P_i$ reads $t_{ss}$ then it:
  - Records its local state ($\sigma_i$)
  - Sends an empty message along all its outgoing channels
  - Starts recording messages on each of incoming channels
  - Stops recording a channel when it receives first message with timestamp greater than or equal to $t_{ss}$

Snapshot (2nd attempt)

- Operate with logical clocks
- Algorithm:
  - $P_0$ sends “take a snapshot”
  - When $P_i$ receives “take a snapshot” for the first time from $P_j$:
    - Records its local state ($\sigma_i$)
    - Sends “take a snapshot” along all its outgoing channels
    - Sets channel from $P_j$ to be empty
    - Starts recording messages on each of incoming channels
  - When $P_i$ receives “take a snapshot” beyond the first time from $P_k$
    - Stops recording channel from $P_k$
  - When $P_i$ has received “take a snapshot” on all channels, it sends collected state to $P_0$ and stops
Snapshot Algorithm Example

(a) Organization of a process and channels for a distributed snapshot

(b) Process Q receives a marker for the first time and records its local state

(c) Q records all incoming message

(d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel
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

Diagram:
- A
- B
- C
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”
    - Snapshot protocol finds 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
    - \( e_A^1 \) should be reported before \( e_A^2 \)
  - Causal properties are preserved; consistent observations are made
    - \( e_A^1 \) should be reported before \( e_C^4 \)

\[
\begin{align*}
& e_A^1 & e_A^2 & e_A^3 & e_A^4 \\
& e_B^1 & e_B^2 & e_B^3 & e_B^4 \\
& e_C^1 & e_C^2 & e_C^3 & e_C^4 \\
& \text{Physical time} \\
& A & B & C
\end{align*}
\]
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:
    - \( e^A_1 \), \( e^A_2 \), \( e^B_1 \), \( e^A_3 \), \( e^B_2 \)
  - Notification of \( e^A_3 \) has been delayed
  - Gap detection problem: given two events \( e_1 \) and \( e_2 \), detect whether or not there is another event \( e_3 \) 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
  - $\Sigma^i_1$ would refer to process 1 at state $i$ and process 2 at state $j$
- Define a lattice of valid global states

Reachability

We say that $\Sigma^u$ is reachable from $\Sigma^l$ if there is a path from $\Sigma^u$ to $\Sigma^l$ 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 $\Sigma^t$ is termination state for snapshot:

$$\Sigma^i \sim \Sigma^s \sim \Sigma^t$$

- Deadlock in $\Sigma^s$ implies deadlock in $\Sigma^t$
- No deadlock in $\Sigma^s$ implies no deadlock in $\Sigma^t$

Global Predicate Detection

- What if we want to detect non-stable predicates?
- Say we want to evaluate predicate at $\Sigma^i$
  - Cannot use snapshots
  - Example: detect if “$x == y$” or “$x == y - 2$”
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 $\Sigma^{54}$

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

- Scan lattice level after level

- 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 which !\(\Phi\) 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

- When is it safe to “drop” a local state?
- How to build level \(i+1\) of lattice given level \(i\)?
Earliest Consistent Global State

$\sum_{i}^{1} \sigma_{i}^{j}$ represents the global state after $i_1$ operations by processor 1, $i_2$ operations by processor 2, etc.

Level of $\sum_{i}^{1} \sigma_{i}^{j}$ is $i_1 + i_2 + \ldots + i_n$

$\sigma_{i}^{j}$ represents state of processor $i$ after $j$ operations

Earliest consistent global state: $\sum_{\min} (\sigma_{i}^{j})$, latest consistent global state: $\sum_{\max} (\sigma_{i}^{j})$
Earliest & Latest Consistent Global State

- For state $\sigma_2^5$:
  - Earliest consistent global state: $\sigma_1^1, \sigma_2^5, \sigma_3^1$
  - Level of earliest consistent global state $= 1 + 5 + 1 = 7$
  - Latest consistent global state: $\sigma_1^5, \sigma_2^5, \sigma_3^5$
  - Corresponding level: 15

Building the lattice

- Collect states at the monitor
- Store them in separate queues
- Constructing level by level
  - To build level $l$: wait until all the states required for the level are available
  - Earliest level for $\sigma_2^3$: $3 + 4 + 1 = 8$, for $\sigma_2^3$: 5, for $\sigma_2^3$: 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: $\sum_{i_1, i_2, \ldots, i_n}$ is a level L global state (stored in the lattice), then construct $\sum_{i_1+1, i_2, \ldots, i_n}$, $\sum_{i_1, i_2+1, \ldots, i_n}$, $\ldots$, $\sum_{i_1, \ldots, i_n+1}$
- 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 $\sigma_5$: latest consistent global state it belongs to is $\sigma_1^5 \sigma_2^5 \sigma_3^5$
- Corresponding level = 15
- Discard $\sigma_2^5$ 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?