Distributed Systems

Clocks, Ordering of events

Rik Sarkar
Edinburgh Spring 2018
Logical clocks

• Why do we need clocks?
  – To know when something happened
  – To determine when one thing happened before another

• Can we determine that without using a “clock” at all?
  – Then we don’t need to worry about synchronization, millisecond errors etc.
Happened before

• $a \rightarrow b : a$ happened before $b$
  – If $a$ and $b$ are successive events in same process then $a \rightarrow b$
  – Send before receive
    • If $a$ : “send” event of message $m$
    • And $b$ : “receive” event of message $m$
    • Then $a \rightarrow b$
  – Transitive: $a \rightarrow b$ and $b \rightarrow c \implies a \rightarrow c$
Events

• “Happened before” can be defined between events

• Examples of event:
  – Sent message
  – Received message
  – Started/finished computation
  – Received input
  – .. Etc..
  – We can decide which events are important to us

• Events in a process are ordered by time/causality
States

• “Happened before” can be defined between states
• A “state” can be seen as the values in all memory and registers of a computer
  – Changes all the time
• More useful: State changes when something important to us has happened
• E.g. an event we care about.
• States in a process are also ordered by time/causality
Ordering among events in different processes

• There is a directed path:
• Events without a happened before relation are “concurrent”
• $e_1 \rightarrow e_2$, $e_3 \rightarrow e_4$, $e_1 \rightarrow e_5$, $e_5 \parallel e_2$
• Events without a happened before relation are “concurrent”
• Happened before is a partial ordering
Happened before & causal order

- Happened before == could have caused/influenced
- Preserves causal relations
- Implies a partial order
  - Implies time ordering between certain pairs of events
  - Does not imply anything about ordering between concurrent events
Logical clocks

• Idea: Use a counter at each process
• Increment after each event
• Can also increment when there are no events
  – Eg. A clock
• An actual clock can be thought of as such an event counter
• It counts the states/events of the process
• Each event has an associated time: The count of the state when the event happened
Lamport clocks

• Keep a logical clock (counter)
• Send it with every message
• On receiving a message, set own clock to max({own counter, message counter}) + 1
• For any event e, write c(e) for the logical time
• Property:
  – If a→b, then c(a) < c(b)
  – If a || b, then no guarantees
Lamport clocks: example
Concurrency and lamport clocks

• If $e_1 \rightarrow e_2$
  – Then no Lamport clock $C$ exists with $C(e_1) = C(e_2)$
Concurrency and lamport clocks

• If $e_1 \rightarrow e_2$
  – Then no Lamport clock $C$ exists with $C(e_1) = C(e_2)$

• If $e_1 || e_2$, then there exists a Lamport clock $C$ such that $C(e_1) = C(e_2)$
The purpose of Lamport clocks
The purpose of Lamport clocks

- If $a \rightarrow b$, then $c(a) < c(b)$
- If we order all events by their Lamport clock times
  - We get a partial order, since some events have the same time
  - The partial order satisfies "causal relations"
The purpose of Lamport clocks

• Suppose there are events in different machines
  – Transactions, money in/out, file read, write, copy

• An ordering of events that guarantees preserving causality
Total order from lamport clocks

• If event e occurs in process j at time C(e)
  – Give it a time (C(e), j)
  – Order events by (C, process id)
  – For events e1 in process i, e2 in process j:
    • If C(e1)<C(e2), then e1<e2
    • Else if C(e1)==C(e2) and i<j, then e1<e2

• Leslie Lamport. Time, clocks and ordering of events in a distributed system.
Logical clocks

• Formally, a map:
  • C:S -> \( \mathbb{N} \)
    – That satisfy happened before relation
    – Where \( S \) is set of states, \( \mathbb{N} \) is natural numbers
    – Essentially, Assign a “number” to each state
    – Other sets (like Integers \( \mathbb{Z} \)) work equally well. As long as the set has a total order.

• Problem:
  – Ordering preserves happened-before
  – But does not imply happened-before
  – There is no way to tell from logical clock if there can be causality.
    • The relation is not an if and only if
Vector clocks

• We want a clock such that:
  – If $a \rightarrow b$, then $c(a) < c(b)$
  – AND
  – If $c(a) < c(b)$, then $a \rightarrow b$

– Ref: Coulouris et al., V. Garg
Vector clocks

• \( V: S \rightarrow \mathbb{N}^n \)
  – Where \( n \) is the number of processes

• And:
  – For states \( x \) and \( y \)
  – \( V_x \leq V_y \) iff for each \( i \) in \( \{1,2,...,n\} \), \( V_x[i] \leq V_y[i] \)
  – The strict inequality is defined as:
    • \( V_x < V_y \) iff for each \( i \) in \( \{1,2,...,n\} \), \( V_x[i] \leq V_y[i] \)
    • And there is a \( j \) such that \( V_x[j] < V_y[j] \)

• That satisfy that
  – \( V_x < V_y \) iff \( x \rightarrow y \)
Vector clock algorithm

• Each process $i$ maintains a vector $V_i$
• $V_i$ has $n$ elements
  – keeps clock $V_i[j]$ for every other process $j$
  – On every local event: $V_i[i] = V_i[i] + 1$
  – On sending a message, $i$ sends entire $V_i$
  – On receiving a message at process $j$:
    • Takes max element by element
    • $V_j[k] = \max(V_j[k], V_i[k])$, for $k = 1, 2, \ldots, n$
    • And adds 1 to $V_j[j]$
Comparing timestamps

- $V = V'$ iff $V[i] == V'[i]$ for $i=1,2,\ldots,n$
- $V \leq V'$ iff $V[i] \leq V'[i]$ for $i=1,2,\ldots,n$
- $V < V'$ iff $V[i] \leq V'[i]$ for $i=1,2,\ldots,n$
  - And there is an $i$ such that $V[i] < V'[i]$
Comparing timestamps

• $V = V'$ iff $V[i] == V'[i]$ for $i=1,2,...,n$
• $V < V'$ iff $V[i] < V'[i]$ for $i=1,2,...,n$

• For events $a$, $b$ and vector clock $V$
  – $a \rightarrow b$ iff $V_a < V_b$

• Is this a total order?
Comparing timestamps

• $V = V'$ iff $V[i] == V'[i]$ for $i=1,2,...,n$

• $V \leq V'$ iff $V[i] \leq V'[i]$ for $i=1,2,...,n$

• For events $a$, $b$ and vector clock $V$
  – $a \rightarrow b$ iff $V_a < V_b$

• Two events are concurrent if
  – Neither $V_a < V_b$ nor $V_b < V_a$
Vector clock examples

• $(1,2,1) \leq (3,2,1)$ but $(1,2,1) \not\leq (3,1,2)$

• Also $(3,1,2) \not< (1,2,1)$

• No ordering exists
Vector clocks

• What are the drawbacks?

• What is the communication complexity?
Vector clocks

• What are the drawbacks?
  – Entire vector is sent with message
  – All vector elements (n) have to be checked on every message

• What is the communication complexity?
  – $\Omega(n)$ per message
Logical and vector clocks

• There is no way to have perfect knowledge on ordering of events
  – A “true” ordering may not exist..

  – Logical and vector clocks give us a way to have ordering consistent with causality