Models of Distributed Computation and Clocks

Pradipta De
pradipta.de@sunykorea.ac.kr
Limitations of Logical Clock

\[ a \rightarrow b \text{ implies } C(a) < C(b) \]

BUT

\[ C(a) < C(b) \text{ doesn’t imply } a \rightarrow b \]!!

Not a true clock, not strictly consistent !!

Why does this happen ?

- A single integer is used to store the logical time
- Cannot distinguish between advancement of clock due to local events and those due to message exchange between processes.
Solution Intuition

• Find causal relationship between two events happening in different processes

• What do we need?
  – Additional information about the events in other processes
  – Extend logical timestamps (single integer) to maintain a list of timestamps (one for each process)
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 \( a \) is send of message \( m \) from \( i \) to \( j \) with vector timestamp \( t_m \), on receive of \( m \):
    \[
    C_j[k] = \text{max}(C_j[k], t_m[k]) \quad \text{for all } k
    \]
Vector Clock: Space Time Diagram

```
| p1 | | p2 | | p3 |
|-----|-----|-----|-----|
| [0 1 0] | [2 2 2] | [0 0 0] | [3 3 3] | [4 4 4] |
| [0 0 0] | [0 0 0] | [0 0 0] | [4 4 4] | [5 5 5] |
| [0 0 0] | [2 2 2] | [2 2 2] | [5 5 5] | [6 6 6] |
```

CSE 535 – Logical Clocks
Comparing Vector Timestamps

For events a and b with vector timestamps \( t^a \) and \( t^b \),

- **Equal:** \( t^a = t^b \) iff \( \forall i, t^a[i] = t^b[i] \)
- **Not Equal:** \( t^a \neq t^b \) iff \( \exists i, t^a[i] \neq t^b[i] \)
- **Less or equal:** \( t^a \leq t^b \) iff \( \forall i, t^a[i] \leq t^b[i] \)
- **Not less or equal:** \( t^a \not\leq t^b \) iff \( \exists i, t^a[i] > t^b[i] \)
- **Less than:** \( t^a < t^b \) iff \( t^a \leq t^b \) and \( t^a \neq t^b \)
- **Not less than:** \( t^a \not< t^b \) iff \( \neg(t^a \leq t^b \) and \( t^a \neq t^b \) \)
- **Concurrent:** \( t^a \| t^b \) iff \( t^a < t^b \) and \( t^b < t^a \)
Vector Clock Consistency

• Satisfies logical clock consistency
  – If $a \rightarrow b$, then $vt(a) < vt(b)$

• If $vt(a) < vt(b)$, then $a \rightarrow b$

• Satisfies Strong Consistency

• Does VC ensure total order?
Overheads of VC

• M = # of messages sent/rcvd by a process
• N = # of processes

• Space Overhead: n log m
• Time Overhead: O(n)
• Communication Overhead: n log m
Problems 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
• Other efficient implementations of vector clocks have been looked at
• Lamport’s logical clock suffices in many applications
  – For ex., in distributed replication
Application of Vector Clock
Causal Ordering of Messages

M2 is delivered before m1 to P3, although m1 is sent before m2
Example: Causal Order Broadcast

• Delivery in **Causal Order**:
  – If \( \text{send}(m_1) \rightarrow \text{send}(m_2) \), then every recipient of both message \( m_1 \) and \( m_2 \) must **deliver** \( m_1 \) before \( m_2 \)

• “**deliver**” – when the message is actually given to the application for processing
  – Different from when it is **received** from the network
  – Consider FIFO delivery in TCP as an example (receive can be non-FIFO)
CBCAST Protocol

- Count only send events in vector clock
- 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
- When \( m \) is delivered at \( j \), \( C_j \) is updated according to vector clock rule
Matrix Clock
Why do we need MC?

- In VC, the information about C is propagated from B, but it does not capture A’s knowledge about what B knows about C.

- MC is useful in discarding updates
  - How can A know if B knows about an update U?
    - Get B’s VC \( \rightarrow \) tells if B knows about U
  - How can A know if all processes knows about the update?
    - Have to look at all VCs from all processes
    - Can one tell just by looking at VC of one process, say B, if C has received the update from A?
Matrix Clock Definitions

• A process $P_i$ maintains a (nxn) matrix
  – $M_{ti}[1...n, 1...n]$
  – Initialization: $m_{ti}[.,.] = 0$

• $m_{ti}[i,i]$: logical clock of $P_i$, tracks progress of local events

• $m_{ti}[i,j]$: latest knowledge that $P_i$ holds about the local logical clock, $m_{tj}[j,j]$, of $P_j$

• $m_{ti}[i,.]$: vector clock $vc_{i}[.]$

• $m_{ti}[j,k]$: latest knowledge of what $p_i$ knows about the latest knowledge that $p_j$ has about the local logical clock, $m_{tk}[k,k]$, of $p_k$
MC update rules

• Rule 1: update local clock on local event
  – $Mt_i[i,i] = mt_i[i,i] + 1$

• Rule 2: send/recv rules
  – message send:
    • Increment local clock $\Rightarrow$ Rule 1
    • piggyback $mt_i[.,.]$ with message
  – Message receipt ($W$ is piggybacked MC) :
    • For $1 \leq k \leq n$: $mt_i[i,k] = \max(mt_i[i,k], W[j,k])$
    • For $1 \leq k, l \leq n$, $mt_i[k,l] = \max(mt_i[k,l], W[k,l])$
MC and VC

- additional property in MC over VC?
  - $M_{t_i}[i,.]$ satisfies all the properties of VC
  - $\min_k(m_{t_i}[k,\cdot]) \geq t \Rightarrow$ process $p_i$ knows that every other process $p_k$ knows that local time of $p_l$ has progressed till $t$.
  - Any information before time $t$ can be discarded
    - Replicated database scenarios: updates before $t$ can be discarded
Efficient Implementations

• With large number of processes, large messages must be exchanged on each send event

• Techniques to reduce size:
  – Differential approach: only send the values in a vector which has changed
  – Direct dependency clocks: maintain a vector clock, but send only the local clock information