Clocks and Ordering in Distributed Systems

Arvind Krishnamurthy
Why do we need to order events in a distributed system?
Distributed Make

- Distributed file servers holds source and object files
- Clients specify modification time on uploaded files
- Use timestamps to decide what needs to be rebuilt
  - if object O depends on source S, and
  - O.time < S.time, rebuild O
- What can go wrong?
Another Example: Facebook

- Remove boss as friend
- Post: “My boss is the worst, I need a new job!”
- Friendship links, posts, privacy settings stored across a large number of distributed servers
  - lots of copies of data: replicas, caches, cross-datacenter replication, etc.
- Don’t want to get a concurrent read to see the wrong order!
Two Approaches

• Synchronize physical clocks
• Logical clocks
• Design a scheme that synchronizes physical clocks
  • What do you think are the sources of inaccuracy?
  • Why is clock synchronization hard?
Simplest Approach

- Designate one server as the master

- Master periodically broadcasts time

- Clients receive broadcast, set their clock to the value in the message

- Is this a good approach?
Variations in Network Latency

- Latency can be unpredictable and has a lower bound

- Tweak: Clients receive broadcast, set their clock to the value in the message + minimum delay
Interrogation Based Approach

- Client sends a roundtrip message to query server’s time
- Set’s client’s clock to server’s clock + half of RTT

Worst case error (if we know the min latency): \((T2 - T0)/2 - \text{min}\)
Practical Realization

- NTP uses an interrogation-based approach, plus:
  - taking multiple samples to eliminate ones not close to min RTT
  - averaging among multiple masters
  - taking into account clock rate skew

- PTP adds hardware timestamping support to track latency introduced in network
Are physical clocks enough?

(measurements from Amazon EC2)
Clock synchronization measurements

- Within a datacenter: ~20-50 microseconds
- Across datacenters: ~50-250 milliseconds
- RPCs within a datacenter: few microseconds
Logical Clocks

- another way to keep track of time
- based on the idea of causal relationships between events
- doesn’t require any physical clocks
Events and Histories

- Processes execute sequences of events.
- Events can be of 3 types: local, send, and receive.
- The local history of a process is the sequence of events executed by process.
Observation 1:
- Events in a local history are totally ordered.

Observation 2:
- For every message, send precedes receive.
Lamport Clock: Increment Rules

\[ LC(e_{p}^{i+1}) = LC(e_{p}^{i}) + 1 \]

\[ LC(e_{q}^{j}) = \max(LC(e_{q}^{j-1}), LC(e_{p}^{i})) + 1 \]

Timestamp \( m \) with \( TS(m) = LC(send(m)) \)
Discussion

- What are the strengths of Lamport clocks?
- What are the limitations of Lamport clocks?
Example of Global Predicate

- Setting: Locks in distributed system
  - Objects locked by nodes and moved to the node that is currently modifying it
  - Nodes requesting the object/lock, send a message to the current node locking it and blocks for a response
- How do we detect deadlocks in this scenario?
Another example

- Suppose we're running a large ML computation, e.g. PageRank
  - thousands of servers
  - each holds some subset of web pages
  - each page starts out with some reputation
  - each iteration: some of that page's reputation gets transferred to the pages it links to (state on other servers!)

- What if a server crashes?

- If we wanted to take checkpoints, what is a "consistent" snapshot?
Global States & Clocks

- Need to reason about global states of a distributed system
- Global state: processor state + communication channel state
- Consistent global state: causal dependencies are captured
- Use virtual clocks to reason about the timing relationships between events on different nodes
Space-Time diagrams

A graphic representation of a distributed execution

H and \( \rightarrow \) impose a partial order
Cuts

A cut $C$ is a subset of the global history of $H$

The frontier of $C$ is the set of events

\[ e_1^c, e_2^c, \ldots, e_n^c \]
Consistent cuts and consistent global states

- A cut is consistent if
  \[ \forall e_i, e_j : e_j \in C \land e_i \rightarrow e_j \Rightarrow e_i \in C \]

- A **consistent global state** is one corresponding to a consistent cut
Not a consistent global state: the cut contains the event corresponding to the receipt of the last message by $p_3$ but not the corresponding send event
Global Consistent States

Can we use Lamport Clocks as part of a mechanism to get globally consistent states?
Global Snapshot

- Develop a simple global snapshot protocol
- Refine protocol as we relax assumptions
- Record:
  - processor states
  - channel states
- Assumptions:
  - FIFO channels
  - Each $m$ timestamped with $T(\text{send}(m))$
Snapshot I

i. $p_0$ selects $t_{ss}$

ii. $p_0$ sends “take a snapshot at $t_{ss}$" to all processes

iii. when clock of $p_i$ reads $t_{ss}$ then $p$
   - records its local state $\sigma_i$
   - sends an empty message along its outgoing channels
   - starts recording messages received on each of incoming channels
   - stops recording a channel when it receives first message with timestamp greater than or equal to $t_{ss}$
Snapshot II

processor $p_0$ selects $\Omega$

$p_0$ sends “take a snapshot at $\Omega$” to all processes; it waits for all of them to reply and then sets its logical clock to $\Omega$

when clock of $p_i$ reads $\Omega$ then $p_i$

- records its local state $\sigma_i$
- sends an empty message along its outgoing channels
- starts recording messages received on each incoming channel
- stops recording a channel when receives first message with timestamp greater than or equal to $\Omega$
Relaxing synchrony

Process does nothing for the protocol during this time!

$p_i$

take a snapshot at $\Omega$

empty message: $TS(m) \geq \Omega$

records local state $\sigma_i$

sends empty message: $TS(m) \geq \Omega$

monitors channels
processor $p_0$ sends itself “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 its outgoing channels
- sets channel from $p_j$ to empty
- starts recording messages received over each of its other 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 and stops.
Same problem, different approach

- Monitor process does not query explicitly
- Instead, it passively collects information and uses it to build an observation.
  
  *(reactive architectures, Harel and Pnueli [1985])*

An **observation** is an ordering of events of the distributed computation based on the order in which the receiver is notified of the events.
Update rules

\[ VC(e_i)[i] := VC[i] + 1 \]

Message \( m \) is timestamped with \( TS(m) = VC(send(m)) \)

\[ VC(e_i) := \max(VC, TS(m)) \]
\[ VC(e_i)[i] := VC[i] + 1 \]
Example

The diagram shows three paths, labeled $p_1$, $p_2$, and $p_3$, each with specific endpoints and intermediate points. The coordinates of the endpoints and intermediate points are as follows:

- For $p_1$: $[0,1,0]$ to $[1,0,0]$ to $[2,1,0]$ to $[3,1,2]$ to $[4,1,2]$ to $[5,1,2]$
- For $p_2$: $[0,1,0]$ to $[1,0,1]$ to $[1,0,2]$ to $[1,2,3]$ to $[4,3,3]$
- For $p_3$: $[1,0,1]$ to $[1,0,2]$ to $[1,0,3]$ to $[5,1,4]$
Operational interpretation

\[ VC(e_i)[i] = \text{no. of events executed by } p_i \text{ up to and including } e_i \]

\[ VC(e_i)[j] = \text{no. of events executed by } p_j \text{ that happen before } e_i \text{ of } p_i \]
VC properties: event ordering

Given two vectors $V$ and $V'$, less than is defined as:

$$V < V' \equiv (V \neq V') \land (\forall k: 1 \leq k \leq n : V[k] \leq V'[k])$$

- **Strong Clock Condition:** $e \rightarrow e' \equiv VC(e) < VC(e')$

- **Simple Strong Clock Condition:**
  Given $e_i$ of $p_i$ and $e_j$ of $p_j$, where $i \neq j$
  $$e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$$

- **Concurrency**
  Given $e_i$ of $p_i$ and $e_j$ of $p_j$, where $i \neq j$
  $$e_i \parallel e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \land (VC(e_j)[j] > VC(e_i)[j])$$
The protocol

- $p_0$ maintains an array $D[1, \ldots, n]$ of counters

- $D[i] = TS(m_i)[i]$ where $m_i$ is the last message delivered from $p_i$

**Rule:** Deliver $m$ from $p_j$ as soon as both of the following conditions are satisfied:

\[
D[j] = TS(m)[j] - 1
\]

\[
D[k] \geq TS(m)[k], \forall k \neq j
\]
Summary

- Lamport clocks and vector clocks provide us with good tools to reason about timing of events in a distributed system.
- Global snapshot algorithm provides us with an efficient mechanism for obtaining consistent global states.