Vector Clocks & Distributed snapshots

CS 452
Vector clocks

Precisely represent transitive causal relationships

\[ T(A) < T(B) \iff \text{happens-before}(A, B) \]

Idea: track events known to each node, *on each node*

Used in practice for eventual and causal consistency

- git, Amazon Dynamo, …
Vector clocks

Clock is a vector $C$, length = # of nodes
On node $i$, increment $C[i]$ on each event
On receipt of message with clock $C_m$ on node $i$:
  - increment $C[i]$
  - for each $j \neq i$
    - $C[j] = \max(C[j], C_m[j])$
Example

A (T = ?)
B (T = ?)
send M (T_m = ?)  
recv M (T = ?)
C (T = ?)
S1
S2
S3
send M' (T_m = ?)
recv M' (T = ?)
D (T = ?)
E (T = ?)
Example

A (1,0,0)

send $M$ ($T_m = ?$)

recv $M$ ($T = ?$)

B ($T = ?$)

send $M$ ($T_m = ?$)

C ($T = ?$)

recv $M'$ ($T = ?$)

D ($T = ?$)

E ($T = ?$)

recv $M'$ ($T = ?$)

S1

S2

S3
Example

S1
A (1,0,0)
B (3,0,0)
send M (2,0,0)
recv M (T = ?)

S2
send M' (T_m = ?)
C (T = ?)
recv M (T = ?)

S3
E (T = ?)
recv M' (T = ?)
D (T = ?)
Example

A (1,0,0)
B (3,0,0)
send M (2,0,0)
recv M (2,1,0)
C (T = ?)
send M' (Tm = ?)
recv M (2,1,0)
D (T = ?)
E (T = ?)
recv M' (T = ?)
D (T = ?)
Example

S1

A (1,0,0)

B (3,0,0)

send M (2,0,0)

recv M (2,1,0)

S2

send M' (T_m = ?)

C (2,2,0)

recv M (2,1,0)

S3

E (T = ?)

recv M' (T = ?)

D (T = ?)
Example

A (1,0,0)
B (3,0,0)
C (2,2,0)
D (T = ?)
E (T = ?)

send M (2,0,0)
recv M (2,1,0)
send M' (2,3,0)
recv M' (T = ?)
Example

A (1,0,0)  B (3,0,0)  C (2,2,0)  D (0,0,1)  E (T = ?)

send M (2,0,0)  recv M (2,1,0)  send M' (2,3,0)  recv M' (T = ?)
Example

A (1,0,0)
B (3,0,0)
send M (2,0,0)
recv M (2,1,0)
C (2,2,0)
send M' (2,3,0)
recv M' (2,3,2)
D (0,0,1)
E (T = ?)

S1
S2
S3
Example

A (1,0,0)
B (3,0,0)
C (2,2,0)
D (0,0,1)
E (2,3,3)

send M (2,0,0)
recv M (2,1,0)
send M' (2,3,0)
recv M' (2,3,2)
Example

S1
A (1,0,0)
B (3,0,0)
send M (2,0,0)
recv M (2,1,0)

S2
send M' (2,3,0)
C (2,2,0)
recv M (2,1,0)

S3
E (2,3,3)
recv M' (2,3,2)
D (0,0,1)
Vector Clocks

Compare vectors element by element

Provided the vectors are not identical,

If \( C_x[i] < C_y[i] \) and \( C_x[j] > C_y[j] \) for some \( i, j \)

\( C_x \) and \( C_y \) are concurrent

if \( C_x[i] \leq C_y[i] \) for all \( i \)

\( C_x \) happens before \( C_y \)
Timestamp: 2,2,2
Queue: [S1@0,0,0,0; S2@0,1,0,0]

Timestamp: 3,1,0
Queue: [S1@0,0,0,0; S2@0,1,0,0]

Timestamp: 0,1,2
Queue: [S1@0,0,0,0; S2@0,1,0,0]
Timestamp: 3,4,2
Queue: [S2@0,1,0]

Timestamp: 3,1,0
Queue: [S2@0,1,0]

Timestamp: 3,1,4
Queue: [S2@0,1,0]
Some terms

Often useful: states, executions, reachability

- A state is a global state $S$ of the system: states at all nodes + channels

- An execution is a series of states $S_i$ s.t. the system is allowed to transition from $S_i$ to $S_{i+1}$

- A state $S_j$ is reachable from $S_i$ if, starting in $S_i$, it’s possible for the system to end up at $S_j$

Types of properties: stable properties, invariants

- A property $P$ is stable if

  $$P(S_i) \Rightarrow P(S_{i+1})$$

- A property $P$ is an invariant if it holds on all reachable states
Token conservation system

haveToken: bool

In $S_o$

- No messages
- Node 1 has haveToken = true
- Node 2 has haveToken = false

Nodes can send each other the token or discard the token
Token conservation system

Node 1  \[\text{token}\]  Node 2

haveToken: bool

Invariant: token in at most one place

Stable property: no token
Token conservation system

How can we check the invariant at runtime?

How can we check the stable property at runtime?
Distributed snapshots

Why do we want snapshots?

- Detect stable properties (e.g., deadlock)
- Distributed garbage collection
- Diagnostics (is invariant still true?)
Distributed snapshots

Record global state of the system

- Global state: state of every node, every channel

Challenges:

- Physical clocks have skew
- State can’t be an instantaneous global snapshot
- State must be consistent
Consistent snapshots

- Consistent global state: causal dependencies are captured
  - If a snapshot of a node includes some events
    - All causally earlier events should be part of snapshots of other nodes
Space Time Diagrams
Cuts

A cut C is a subset of the global history of H
Consistent Cuts

• A cut is consistent if
  • e2 is in the cut and if e1 happens before e2
    • then e1 should also be in the cut
• A consistent global state is one corresponding to a consistent cut
Inconsistent Cut (or global state)
Physical time algorithm

What if we could trust clocks?

Idea:

- Node: “hey, let’s take a snapshot @ noon”
- At noon, everyone records state
- How to handle channels?
Physical time algorithm

Channels:

- Timestamp all messages
- Receiver records channel state
- Channel state = messages received after noon but sent before noon

Example: is there \( \leq 1 \) token in the system?
Physical time algorithm

11:59

haveToken = true

haveToken = false
Physical time algorithm

11:59

token@11:59

Node 1  -->  Node 2

haveToken = false  haveToken = false
Physical time algorithm

12:00

Node 1

Snapshot:
- haveToken = false
- token@11:59

Node 2

Snapshot:
- haveToken = false

haveToken = false

Snapshot:
- haveToken = false
Physical time algorithm

This seems like it works, right?

What could go wrong?
Physical time algorithm

Node 1

haveToken = true

Node 2

haveToken = false

11:59

11:58
Physical time algorithm

Node 1

haveToken = true

Node 2

haveToken = false

Snapshot:
- haveToken = true

12:00

11:59
Physical time algorithm

12:00
Node 1

11:59
Node 2

haveToken = false

token@12:00

Snapshot:
- haveToken = true

haveToken = false
Physical time algorithm

12:00

Node 1

haveToken = false

Snapshot:
- haveToken = true

Node 2

11:59

haveToken = true
Physical time algorithm

Node 1

12:01

haveToken = false

Snapshot:
- haveToken = true

Node 2

12:00

haveToken = true

Snapshot:
- haveToken = true
Avoiding inconsistencies

As we’ve seen, physical clocks aren’t accurate enough
Need to use messages to coordinate snapshot
=> make sure Node 2 takes snapshot before receiving any messages sent after Node 1 takes snapshot
Better algorithm

11:59

Node 1

haveToken = true

11:58

Node 2

haveToken = false
Better algorithm

Node 1

12:00

snapshot@12:00

Node 2

11:59

haveToken = true

haveToken = false

Snapshot:
- haveToken = true
Better algorithm

12:00

Node 1

11:59

Node 2

token@12:00

snapshot@12:00

haveToken = false

Snapshot:
- haveToken = true

haveToken = false
Better algorithm

12:00

Node 1

haveToken = false

Snapshot:
- haveToken = true

Node 2

haveToken = false

Snapshot:
- haveToken = false

token@12:00
Better algorithm

Node 1

12:00

haveToken = false

Snapshot:
- haveToken = true

Node 2

11:59

haveToken = true

Snapshot:
- haveToken = false
Better algorithm

Node 1
haveToken = false
Snapshot:
- haveToken = true

Node 2
haveToken = true
Snapshot:
- haveToken = false
Distributed Snapshots

As we’ve seen, physical clocks aren’t accurate enough. Need to use messages to coordinate snapshot.

=> make sure Node 2 takes snapshot before receiving any messages sent after Node 1 takes snapshot.
Chandy-Lamport Snapshots

At any time, a node can decide to snapshot
  - Actually, multiple nodes can
That node:
  - Records its current state
  - Sends a “marker” message on all channels
When a node receives a marker, snapshot
  - Record current state
  - Send marker message on all channels
How to record channel state?
Chandy-Lamport Snapshots

Channel state recorded by the receiver

Recorded when marker received on that channel

- Why do we know we’ll receive a marker on every channel?

When marker received on channel, record:

- Empty, if this is the first marker

- Messages received on channel since we snapshotted, otherwise
Chandy-Lamport Snapshots
Chandy-Lamport Snapshots

Node 1 → Node 2

haveToken = true

haveToken = false
Chandy-Lamport Snapshots

Node 1

haveToken = false

Node 2

haveToken = false

token
Chandy-Lamport Snapshots

Node 1

haveToken = false

token

Node 2

marker

haveToken = false

Snapshot:
- haveToken = false
Chandy-Lamport Snapshots

Node 1

haveToken = false

Snapshot:
- haveToken = false

Node 2

haveToken = false

Snapshot:
- haveToken = false

marker
token

Node 1 → Node 2

Node 2 → Node 1
Chandy-Lamport Snapshots

Snapshot:
- haveToken = false

In-flight:
- token

Node 1

marker

Node 2

haveToken = true

Snapshot:
- haveToken = false
Chandy-Lamport Snapshots

Node 1

Snapshot: haveToken = false

Node 2

Snapshot: haveToken = true

haveToken = false

Snapshot: haveToken = false

Snapshot: haveToken = false
Chandy-Lamport Snapshots

What if multiple nodes initiate the snapshot?

- Follow same rules: send markers on all channels

Intuition:

- All initiators are concurrent
- Concurrent snapshots are ok, as long as we account for messages in flight
- If receive marker before initiating, must snapshot to be consistent with other nodes
Chandy-Lamport Snapshots
Consistent Cut

A cut is the set of events on each node in the system that are included in the snapshot.

A consistent cut is a cut that respects causality.

If an event is included by any node, all events that “happen before” the event are also included.
Which state is snapshotted?

Let’s say we have an execution $S_0, S_1, \ldots$

Some node starts the snapshot in $S_b$

The snapshot finishes in $S_e$

Which state did we snapshot?
Which state is snapshotted?

Node 1

counter = 4

Node 2

counter = 2
Which state is snapshotted?

Node 1

counter = 4

marker

Node 2

counter = 2

Snapshot:
- counter = 4
Which state is snapshotted?

Node 1: counter = 5
Node 2: counter = 2

Snapshot:
- counter = 4
Which state is snapshotted?

Node 1: counter = 5
Node 2: counter = 3

Marker

Snapshot:
- counter = 4
Which state is snapshotted?

Node 1
counter = 5
Snapshot:
- counter = 4

Node 2
counter = 3
Snapshot:
- counter = 3
Which state is snapshotted?

What *can* we say about this snapshotted state?

Two things:

- Reachable from $S_b$
- Can reach $S_e$

Proof is in the paper

- Intuition: state is “consistent” with what actually happened
Stable Properties and Invariants

Recall: a stable property is one that, once true, stays true

An invariant is true of all states

Snapshot represents a reachable state, but it may not represent any actual global state from $S_b$ to $S_e$
Stable Properties and Invariants

If stable property is \textit{true} in snapshot, we know it \textit{must} still be true in $S_e$.

If stable property is \textit{false} in snapshot, we know it \textit{must} have been false in $S_b$.

If invariant is false in snapshot, we know the invariant is violated in at least one reachable state.

If invariant is true in snapshot, we do \textit{not} know the invariant is true in any other reachable state.