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 = ?)

send M (T_m = ?)

B (T = ?)

recv M (T = ?)

C (T = ?)

send M' (T_m = ?)

recv M (T = ?)

D (T = ?)

E (T = ?)

recv M' (T = ?)
Example

A (1,0,0) → B (T = ?)
send M (2,0,0)

S1

S2

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

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

S3
Example

A (1,0,0)

send M (2,0,0)

B (3,0,0)

recv M (T = ?)

C (T = ?)

send M' (T_m = ?)

D (T = ?)

recv M' (T = ?)

E (T = ?)
Example

S1
A (1,0,0)
send M (2,0,0)

S2
B (3,0,0)

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

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

\[ A (1,0,0) \quad B (3,0,0) \quad C (2,2,0) \quad D (0,0,1) \quad E (T = ?) \]

\[ \text{send } M (2,0,0) \quad \text{recv } M (2,1,0) \quad \text{send } M' (2,3,0) \quad \text{recv } M' (T = ?) \]
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' (2,3,2)

S1

S2

S3
Example

S1

A (1,0,0)

send M (2,0,0)

B (3,0,0)

recv M (2,1,0)

S2

C (2,2,0)

send M' (2,3,0)

recv M (2,1,0)

S3

E (2,3,3)

recv M' (2,3,2)

D (0,0,1)

E (2,3,3)
Example

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

to

c

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

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; S2@0,1,0]

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

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

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

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

Timestamp: 4,4,4
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

Node 1 ➔ token ➔ Node 2

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 ---->

Token

<---- Node 2

haveToken: bool

Invariant: token in at most one place
Stable property: no token
Token conservation system

Node 1  

haveToken: bool

Node 2  

haveToken: bool

token

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

Why do we want snapshots?

- Checkpoint and restart
- 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
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

Node 1

haveToken = true

Node 2

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

haveToken = false

Snapshot:
- haveToken = false

Node 2

Snapshot:
- haveToken = false

Snapshot:
- token@11:59
Physical time algorithm

This seems like it works, right?
What could go wrong?
Physical time algorithm

11:59

Node 1

haveToken = true

Node 2

haveToken = false

11:58
Physical time algorithm

Node 1

haveToken = true

12:00

Node 2

haveToken = false

11:59

Snapshot:
- haveToken = true
Physical time algorithm

12:00 to 11:59

Node 1

Node 2

haveToken = false

haveToken = false

token@12:00

Snapshot:
  - haveToken = true
Physical time algorithm

Node 1

12:00

Node 2

11:59

haveToken = false

haveToken = true

Snapshot:
- haveToken = true
Physical time algorithm

Node 1
haveToken = false
Snapshot:
- haveToken = true

Node 2
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

Node 2

11:58

haveToken = true

haveToken = false
Better algorithm

12:00       11:59
Node 1 ---- snapshot@12:00 ---- Node 2

haveToken = true
Snapshot:
- haveToken = true

haveToken = false
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

Node 2

token@12:00

haveToken = false

Snapshot:
- haveToken = true

11:59

haveToken = false

Snapshot:
- haveToken = false
Better algorithm

Node 1

12:00

Node 2

11:59

haveToken = false

Snapshot:
- haveToken = true

haveToken = true

Snapshot:
- haveToken = false
Better algorithm

Node 1 — haveToken = false

Snapshot:
- haveToken = true

Node 2 — haveToken = true

Snapshot:
- haveToken = false
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

haveToken = true

Node 2

haveToken = false
Chandy-Lamport Snapshots

Node 1

Node 2

haveToken = false

haveToken = false
Chandy-Lamport Snapshots

Node 1

haveToken = false

Node 2

haveToken = false

token

marker

Snapshot:
- haveToken = false
Chandy-Lamport Snapshots

Node 1

- haveToken = false

Snapshot:
- haveToken = false

Node 2

- haveToken = false

Snapshot:
- haveToken = false
Chandy-Lamport Snapshots

Node 1  marker  Node 2

haveToken = false

Snapshot:
- haveToken = false

haveToken = true

Snapshot:
- haveToken = false

In-flight:
- token
Chandy-Lamport Snapshots

Node 1

Snapshot:
- haveToken = false

Snapshot:
- haveToken = false

Node 2

Snapshot:
- token

haveToken = true

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
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.

Consistent Cut
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 *true* in snapshot, we know it *must* still be true in $S_e$

If stable property is *false* in snapshot, we know it *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 *not* know the invariant is true in any other reachable state.