Lamport Clocks

Doug Woos
Logistics notes

Problem Set 1 due Friday

Chandy-Lamport Snapshots thread up
Today

Lamport Clocks
- Motivation
- Basic ideas
- Mutual exclusion
- State machine replication

Vector clocks
Lamport Clocks

Classic paper in distributed systems, but not really implemented in practice

So, why read it?

- Good example of reasoning about systems
- Core ideas are useful—notion of logical time as distinct from physical time
- Causal ordering is important in weak consistency models (eventual consistency!)
Semi-realistic example

You have a large, complex distributed system

Sometimes, things go wrong—bugs, bad client behavior, etc.

You want to be able to debug!

So, each node produces a log
1. Sent Put to 2
2. Received Get from client
3. Received PutReply from 2
4. Did some stuff
5. Sent GetReply

Node 1

1. Received Put from 1
2. ...

Node 2

1. Sent Get to 2
2. ...

Node 3
How do we order these events?

By timestamp, using a physical clock?

- Clock skew is real
- Crystals oscillate at slightly different frequencies
- Typically, ~2s/month skew
- Clock sync relies on communication!

Need a better way of assigning timestamps to events

- Globally valid, s.t. it respects causality
- Using only local information

So: what does it mean for $a$ to happen before $b$?
Happens-before

1. Happens at same location, earlier
2. Transmission before receipt
3. Transitivity
Example

S1

send M

S2

send M'

recv M

S3

recv M'

E

A

B

C

D

E


Goal of a logical clock

happens-before(A, B) -> T(A) < T(B)

What about the converse?
Logical clock implementation

Keep a clock $T$
Increment $T$ whenever an event happens
Send clock value on all messages as $T_m$
On message receipt: $T = \max(T, T_m) + 1$
Example

A (T = 1)

B (T = 3)

send M (T_m = 2)

C (T = 4)

recv M (T = 3)

send M' (T_m = 5)

recv M' (T = 6)

D (T = 1)

E (T = 7)
Mutual exclusion

Use clocks to implement a lock

Goals:

- Only one process has the lock at a time
- Requesting processes eventually acquire the lock, in same order they request it

Assumptions:

- Reliable in-order channels (TCP)
- No failures
Mutual exclusion implementation

Timestamp all messages

Three message types:

- request
- release
- acknowledge

Each node’s state:

- A queue of request messages, ordered by $T_m$
- The latest message it has received from each node
Mutual exclusion implementation

On receiving a request:
  - Record message timestamp
  - Add request to queue

On receiving a release:
  - Record message timestamp
  - Remove corresponding request from queue

On receiving an acknowledge:
  - Record message timestamp
Mutual exclusion implementation

To acquire the lock:

- Send *request* to everyone, including self
- The lock is acquired when:
  - My request is add the head of my queue, and
  - I’ve received higher-timestamped messages from everyone
S1

Timestamp: 2
Queue: [S1@0; S2@1]
S2_{\text{max}}: 1
S3_{\text{max}}: 0

S2

Timestamp: 1
Queue: [S1@0; S2@1]
S1_{\text{max}}: 0
S3_{\text{max}}: 0

S3

Timestamp: 2
Queue: [S1@0; S2@1]
S1_{\text{max}}: 0
S2_{\text{max}}: 1
Timestamp: 1
Queue: [S1@0; S2@1]
S1\text{max}: 0
S3\text{max}: 0

Timestamp: 3
Queue: [S1@0; S2@1]
S2\text{max}: 1
S3\text{max}: 0

Timestamp: 3
Queue: [S1@0; S2@1]
S1\text{max}: 0
S2\text{max}: 1
Timestamp: 4
Queue: [S1@0; S2@1]
S1 max: 3
S3 max: 3

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

Timestamp: 4
Queue: [S1@0; S2@1]
S2 max: 1
S3 max: 0
Timestamp: 5
Queue: [S2@1]
S1_{max}: 4
S3_{max}: 3

Timestamp: 4
Queue: [S2@1]
S2_{max}: 1
S3_{max}: 0

Timestamp: 5
Queue: [S2@1]
S1_{max}: 4
S2_{max}: 1
State machine replication

We’ve seen a SMR implementation: Primary/backup

Key question in SMR: what is the order in which ops are executed?

How does this work in Primary/backup?

How to do SMR with Lamport clocks?
Mutual exclusion as SMR

State: queue of processes who want the lock

Commands: $P_i$ requests, $P_i$ releases

Process a command iff we’ve seen all commands w/ lower timestamp

What are advantages/disadvantages over P/B?
Vector clocks

Another type of logical clock

Sometimes actually used in practice

- Eventual consistency

Better partial order

- Logical time partially ordered, integers totally
- Want $T(A) < T(B) \rightarrow$ happens-before(A, B)

Idea: track a timestamp for each node, *on each node*
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 $C_m$:
  - increment $C[i]$
  - for each $j$:
    - $C[j] = \max(C[j], C_m[j])$
Vector clocks

Ordering is partial: compare vectors pointwise
Why does $T(A) < T(B) \rightarrow \text{happens-before}(A, B)$?
Piazza discussion

What happens when we need to add a process?
Why is coordination necessary for locking?
Events that happened vs. might have happened