Lamport Clocks

Tom Anderson + Doug Woos

Logistics notes

Problem Set 1 out Monday

Will be due a week from Monday

Today

- Primary Backup Wrap-up
- Lamport Clocks
  - Motivation
  - Basic idea
  - Mutual exclusion
  - State machine replication
- Vector clocks

Primary Backup: Why its hard

Primary may fail

Backup may fail

Communication may fail partially or temporarily

Participants may lag decisions made at:

- viewserver (has view changed?)
- primary (did it fail? reply to client message?)
- backup (did it fail? has it learned of new view? has state transfer completed?)

Lab 2 Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Lab 2 Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Why Atomic State Transfer?

Until new backup is up to date, there is a window of vulnerability

- if new primary crashes, lose state

This is why we need to do the backup *quickly*

- simpler to implement if no concurrent ops

But why *must* we do it atomically?

Why Atomic State Transfer?

1:A,B

// A is new primary, B is new backup

A starts sending state to B
Client sends op to A // modifies state not yet sent to B
A forwards op to B
B applies op
A sends rest of state to B // overwrites op
A applies op
// A and B are inconsistent

One more corner case

1:A,B

View server stops hearing from A
A and B, and clients, can still communicate

2:B,C

B hasn't heard from view server
Client in view 1 sends a request to A
What should happen?
Client in view 2 sends a request to B
What should happen?

Primary Backup Questions

What state to replicate?

How does the backup get state?

Apply changes to backup, or just log?

When do we cut over to the backup?

Are anomalies visible at the cut over?

How do we repair/re-integrate?

Replicated Virtual Machines

Whole system replication

Completely transparent to applications and clients

High availability for any existing software

Challenge: Need state at backup to exactly mirror primary

Restricted to a uniprocessor VMs

Deterministic Replay

Key idea: state of VM depends only on its input

- Content of all input/output
- Precise instruction of every interrupt
- Only a few exceptions (e.g., timestamp instruction)

Record all hardware events into a log

- Modern processors have instruction counters and can interrupt after (precisely) x instructions
- Trap and emulate any non-deterministic instructions

Replicated Virtual Machines

Replay I/O, interrupts, etc. at the backup

- Backup executes events at primary with a lag
- Backup stalls until it knows timing of next event
- Backup does not perform external events

Primary stalls until it knows backup has copy of every event up to (and incl.) output event

- Then it is safe to perform output

On failure, inputs/outputs will be replayed at backup (idempotent)

Example

Primary receives network interrupt

- hypervisor forwards interrupt plus data to backup
- hypervisor delivers network interrupt to OS kernel
- OS kernel runs, kernel delivers packet to server
- server/kernel write response to network card
- hypervisor gets control and sends response to backup
- hypervisor delays sending response to client until backup asks

Backup receives log entries

- backup delivers network interrupt
- …
- hypervisor does *not* put response on the wire
- hypervisor ignores local clock interrupts

Questions

Why send output events to backup and delay output at primary until backup has acked?

What happens when primary fails after receiving network input but before sending log entry to backup?

Can the same output be produced twice?

Lamport Clocks

Framework for *reasoning* about event ordering

- notion of logical time vs. physical time
- causal ordering and vector clocks (e.g., git)
- state machine replication

A Few Examples

Primary backup

Consistency in distributed make

Update ordering on social media

Merging distributed event logs

Replication w/ Event Ordering

Suppose we had a globally valid way to assign timestamps to events

Clients label ops with timestamp

Send ops directly to *both* primary and backup

Primary and backup apply events in timestamp order

Client safe when get ack from both

Viewserver still needed for failover, split-brain, etc.

- In new view, client asks: did this event happen?

Distributed Make

Distributed file servers hold source and object files

Clients update files (with modification times)

Make uses timestamps to decide what must be rebuilt

- If object O depends on source S

  and O.time < S.time, rebuild O

Depends on correctness of timestamp; what can go wrong?

Update Ordering

Silently block boss on twitter

Tweet: “My boss is the worst, I need a new job!”

Tweets and block/mute lists sharded across many servers

Copies on many replicas, caches, across data centers

How do you guarantee that no read sees the updates in the wrong order?

Example: Merging Event Logs

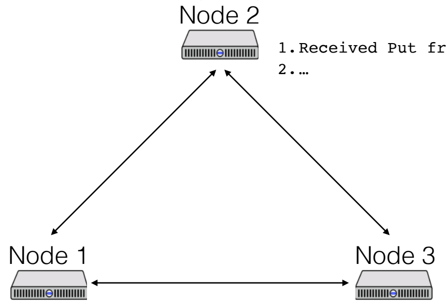
You have a large, complex distributed system

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

You want to be able to debug!

So, each node produces a (partial) event log

Example: Merging Event Logs



Node 2:

```
1.Received Put from 1
2.…
```

Node 1:

```
1. Sent Put to 2
2. Received Get from client
3. Received PutReply from 2
4. Did some stuff
5. Sent GetReply
```

Node 3:

```
1.Sent Get to 2
2.…
```

Centralize the log?

Events will be ordered at the logger

Expensive! More scalable to keep local logs

Might not represent order of events as they happened at each node!

Physical Clocks

Label each event with its physical time

- How closely can we approximate physical time?

Building blocks

- Server clock oscillator skews at 2s/month
- Atomic clock: ns accuracy, expensive
- GPS: 10ns accuracy, requires antenna
- Network packets with variable network latency, scheduling delay

Physical Clocks: Beacon

Designate server with GPS/atomic clock as the master

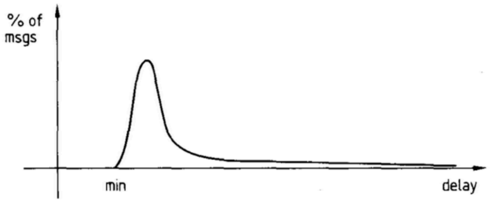
Master periodically broadcasts time

Clients receive broadcast, reset their clock

- Taking care so time never runs backwards

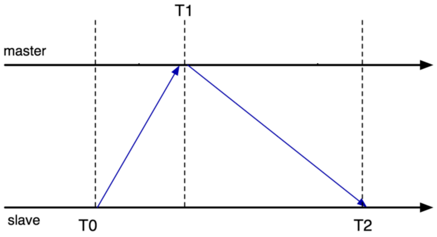
How well does this work?

Network Latency



Network latency is unpredictable with a lower bound

Client Driven Approach: NTP, PTP



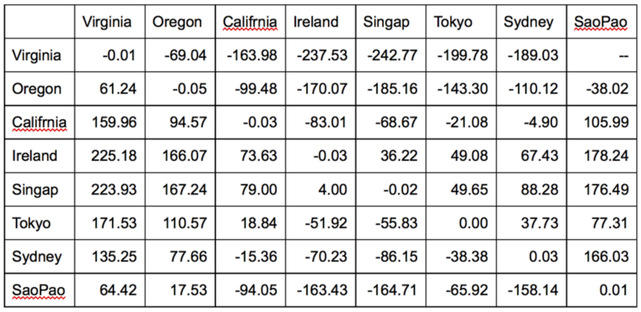
Client queries server

Time = server's clock - 1/2 round trip

Average over several servers; throw out outliers

In between queries, adjust for measured clock skew

Time Accuracy in Practice (ms)



	Virginia	Oregon	Califrnia	Ireland	Singap	Tokyo	Sydney	SaoPao
Virginia	-0.01	-69.04	-163.98	-237.53	-242.77	-199.78	-189.03	--
Oregon	61.24	-0.05	-99.48	-170.07	-185.16	-143.30	-110.12	-38.02
Califrnia	159.96	94.57	-0.03	-83.01	-68.67	-21.08	-4.90	105.99
Ireland	225.18	166.07	73.63	-0.03	36.22	49.08	67.43	178.24
Singap	223.93	167.24	79.00	4.00	-0.02	49.65	88.28	176.49
Tokyo	171.53	110.57	18.84	-51.92	-55.83	0.00	37.73	77.31
Sydney	135.25	77.66	-15.36	-70.23	-86.15	-38.38	0.03	166.03
SaoPao	64.42	17.53	-94.05	-163.43	-164.71	-65.92	-158.14	0.01

(measurements from Amazon EC2)

Spanner Time Accuracy

Google put multiple GPS/atomic clocks in every data center, for a system called Spanner

- Prioritize time traffic to reduce network jitter
- Accuracy = Interval between pings * 200usec/sec

Event resolution needed to rely on physical clocks:

5ns = minimum packet on 100Gbps link

100ns = minimum packet latency (intra-rack)

Fine-Grained Physical Clocks

Timestamps taken in hardware on the network interface

Eliminate samples that involve any network queueing

Continually re-estimate clock skew

- Skew is temperature dependent

Connect all servers in data center into a mesh

- average all neighbors (mostly short hops)

Accuracy ~ 100ns in the worst case

Logical Clocks

Way to assign timestamps to events

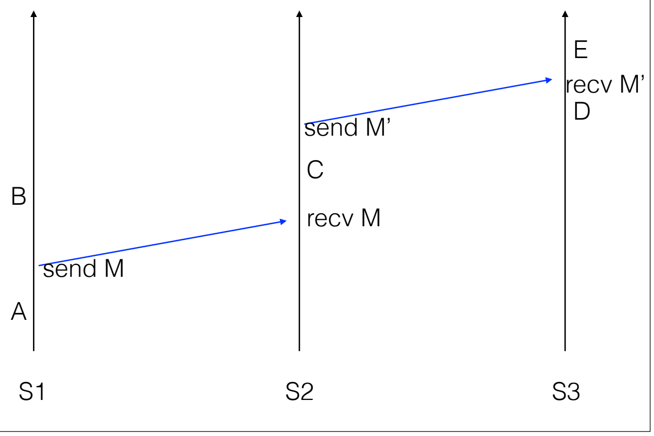
- Globally valid, such that it respects causality
- Using only local information
- No physical clock

What does it mean for *a* to happen before *b*?

Happens-before

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

Example



E

recv M’

D

send M’

C

B

recv M

send M

A

S1

S2

S3

Goal of a logical clock

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

What about the converse?

Logical clock implementation

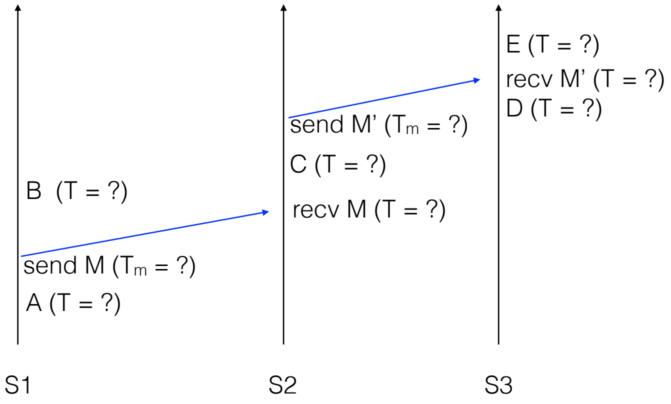
Keep a local clock T

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



E (T = ?)

recv M’ (T = ?)

D (T = ?)

send M’ (T_m = ?)

C (T = ?)

B (T = ?)

recv M (T = ?)

send M (T_m = ?)

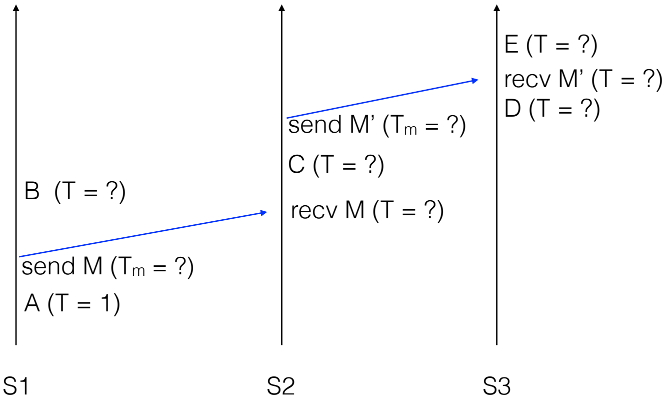
A (T = ?)

S1

S2

S3

Example



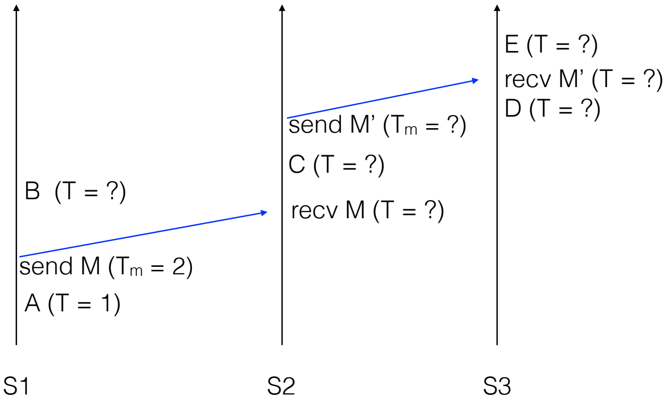
E (T = ?)
recv M’ ($T = ?$)
D (T = ?)

send M’ ($T_m = ?$)
C (T = ?)
recv M (T = ?)

B (T = ?)
send M ($T_m = ?$)
A (T = 1)

S1 S2 S3

Example



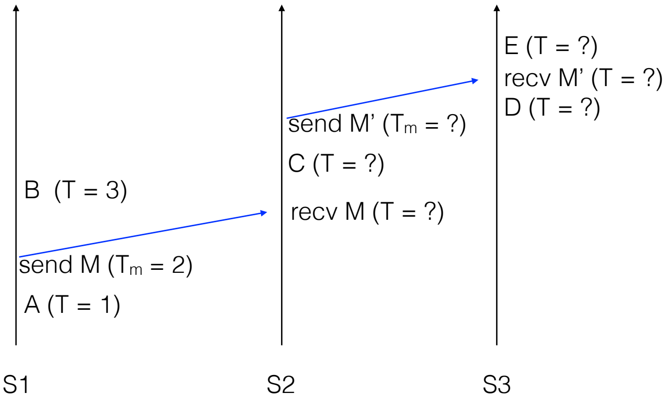
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ ($T_m = ?$)
C (T = ?)
recv M (T = ?)

B (T = ?)
send M ($T_m = 2$)
A (T = 1)

S1 S2 S3

Example



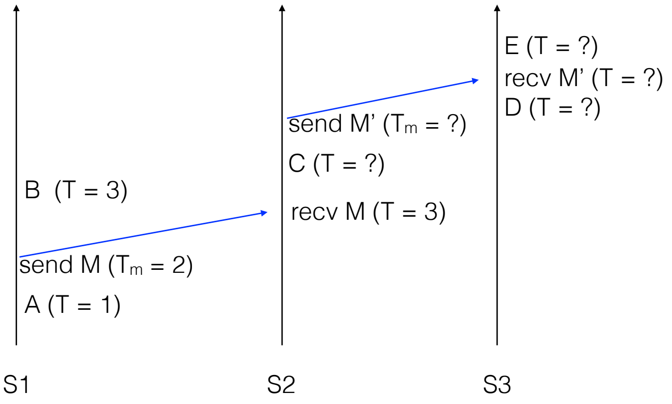
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ ($T_m = ?$)
C (T = ?)
recv M (T = ?)

B (T = 3)
send M ($T_m = 2$)
A (T = 1)

S1 S2 S3

Example



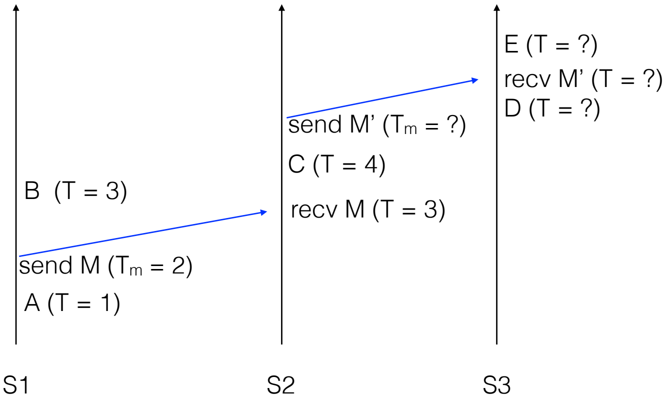
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ ($T_m = ?$)
C (T = ?)
recv M (T = 3)

B (T = 3)
send M ($T_m = 2$)
A (T = 1)

S1 S2 S3

Example



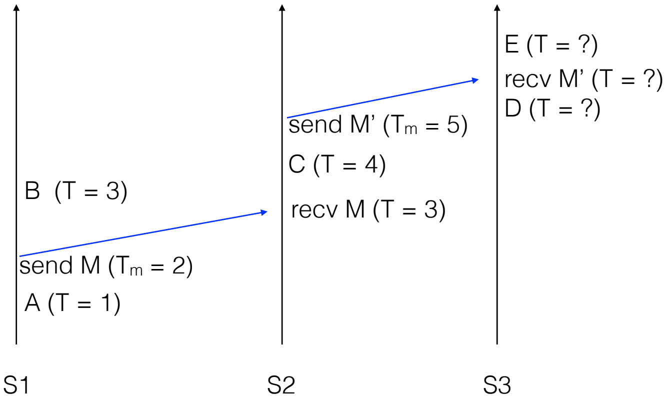
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ ($T_m = ?$)
C (T = 4)
recv M (T = 3)

B (T = 3)
send M ($T_m = 2$)
A (T = 1)

S1 S2 S3

Example



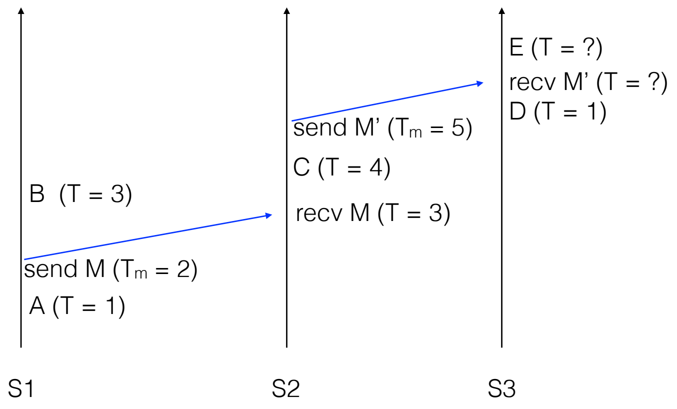
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ ($T_m = 5$)
C (T = 4)
recv M (T = 3)

B (T = 3)
send M ($T_m = 2$)
A (T = 1)

S1 S2 S3

Example



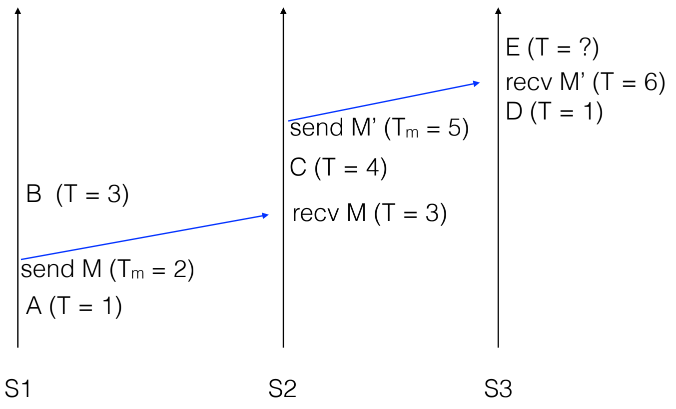
E (T = ?)
recv M' (T = ?)
D (T = 1)

send M' (T_m = 5)
C (T = 4)
recv M (T = 3)

B (T = 3)
send M (T_m = 2)
A (T = 1)

S1
S2
S3

Example



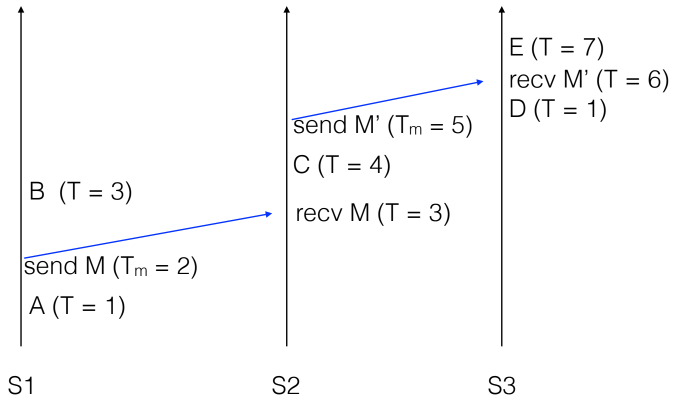
E (T = ?)
recv M' (T = 6)
D (T = 1)

send M' (T_m = 5)
C (T = 4)
recv M (T = 3)

B (T = 3)
send M (T_m = 2)
A (T = 1)

S1
S2
S3

Example



E (T = 7)
recv M' (T = 6)
D (T = 1)

send M' (T_m = 5)
C (T = 4)
recv M (T = 3)

B (T = 3)
send M (T_m = 2)
A (T = 1)

S1
S2
S3

Mutual exclusion

Use clocks to implement a lock

- Using state machine replication

Goals:

- Only one process has the lock at a time
- Requesting processes eventually acquire the lock

Assumptions:

- In-order point-to-point message delivery
- No failures

Mutual exclusion implementation

Each message carries a timestamp T_m (and a seq #)

Three message types:

- *request* (broadcast)
- *release* (broadcast)
- *acknowledge* (on receipt)

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

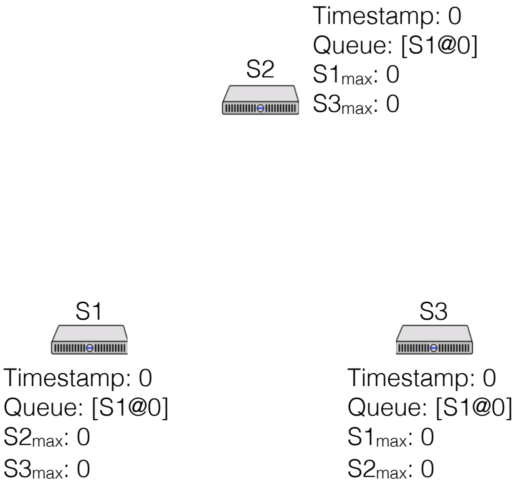
On receiving an *acknowledge*:

- Record message timestamp

Mutual exclusion implementation

To acquire the lock:

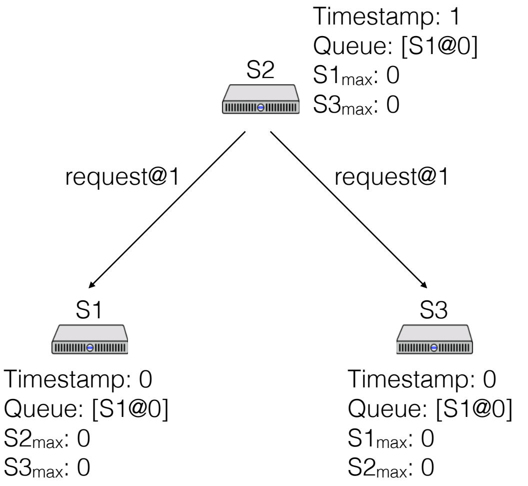
- Send *request* to everyone, including self
- The lock is acquired when:
  - My request is at the head of my queue, and
  - I've received higher-timestamped messages from everyone
  - So my request must be the earliest



Timestamp: 0
Queue: [S1@0]
$S1_{max}$: 0
$S3_{max}$: 0

Timestamp: 0
Queue: [S1@0]
$S2_{max}$: 0
$S3_{max}$: 0

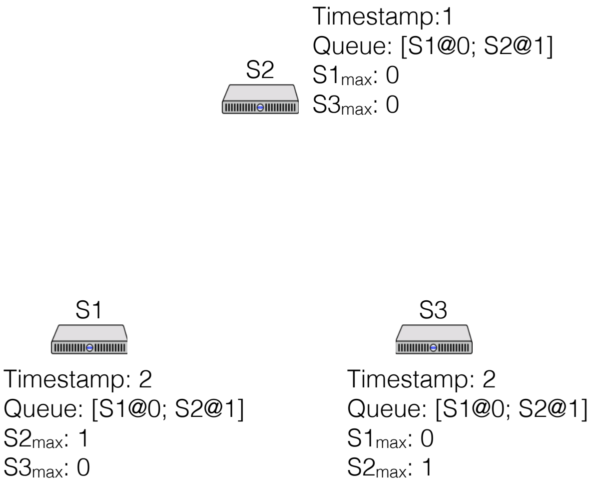
Timestamp: 0
Queue: [S1@0]
$S1_{max}$: 0
$S2_{max}$: 0



Timestamp: 1
Queue: [S1@0]
$S1_{max}$: 0
$S3_{max}$: 0

Timestamp: 0
Queue: [S1@0]
$S2_{max}$: 0
$S3_{max}$: 0

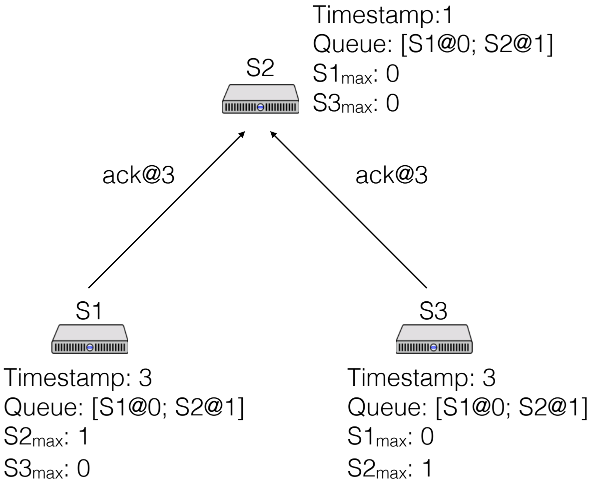
Timestamp: 0
Queue: [S1@0]
$S1_{max}$: 0
$S2_{max}$: 0



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

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

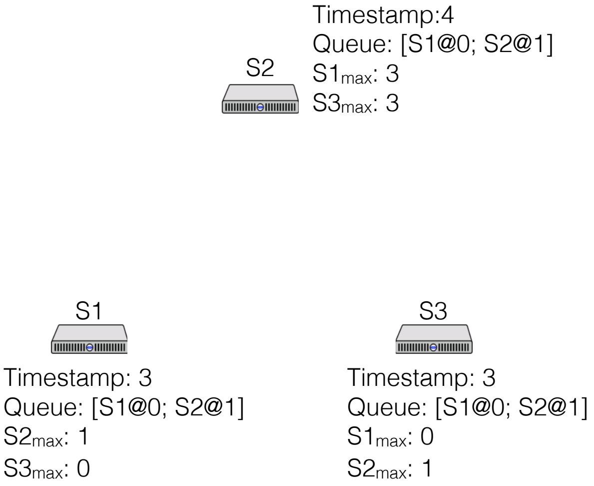
Timestamp: 2
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1



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

Timestamp: 3
Queue: [S1@0; S2@1]
$S2_{max}$: 1
$S3_{max}$: 0

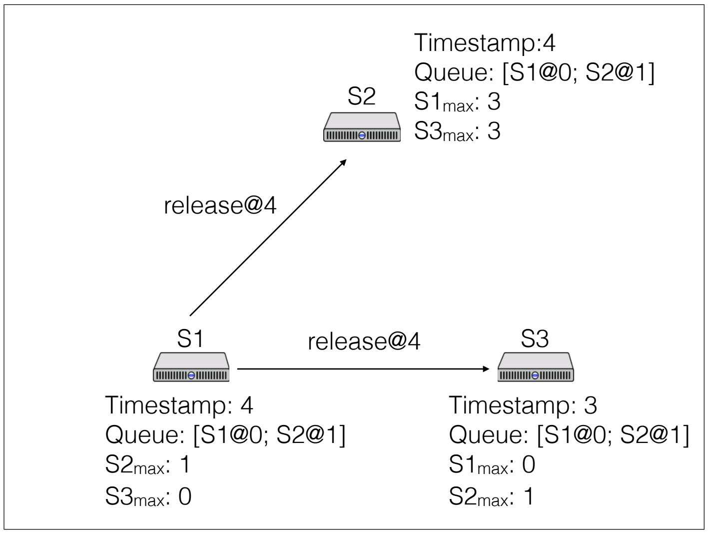
Timestamp: 3
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1



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

Timestamp: 3
Queue: [S1@0; S2@1]
$S2_{max}$: 1
$S3_{max}$: 0

Timestamp: 3
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1



S2

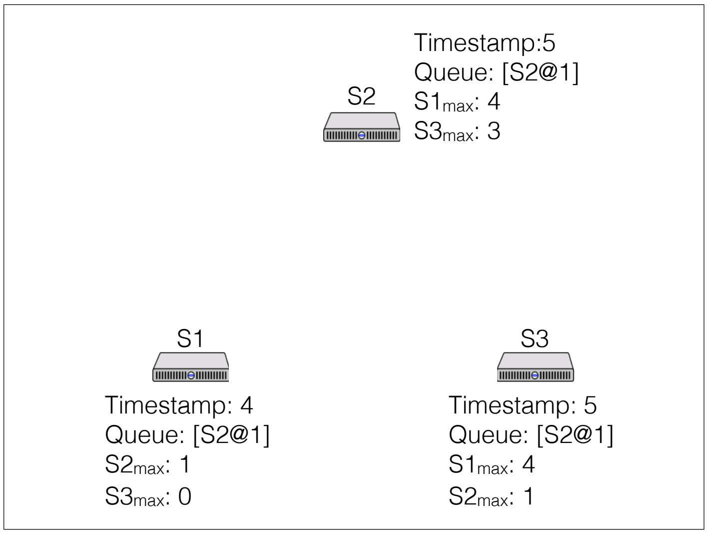
Timestamp:4
Queue: [S1@0; S2@1]
$S1_{max}$: 3
$S3_{max}$: 3

S1

Timestamp: 4
Queue: [S1@0; S2@1]
$S2_{max}$: 1
$S3_{max}$: 0

S3

Timestamp: 3
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1



S2

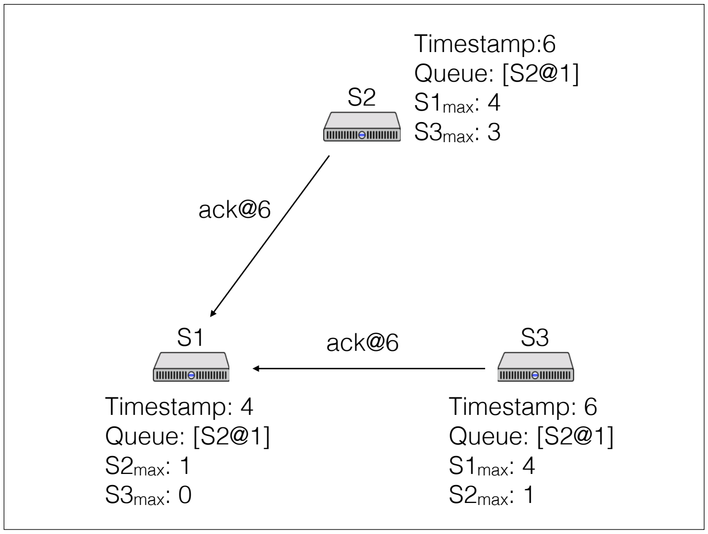
Timestamp:5
Queue: [S2@1]
$S1_{max}$: 4
$S3_{max}$: 3

S1

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

S3

Timestamp: 5
Queue: [S2@1]
$S1_{max}$: 4
$S2_{max}$: 1



S2

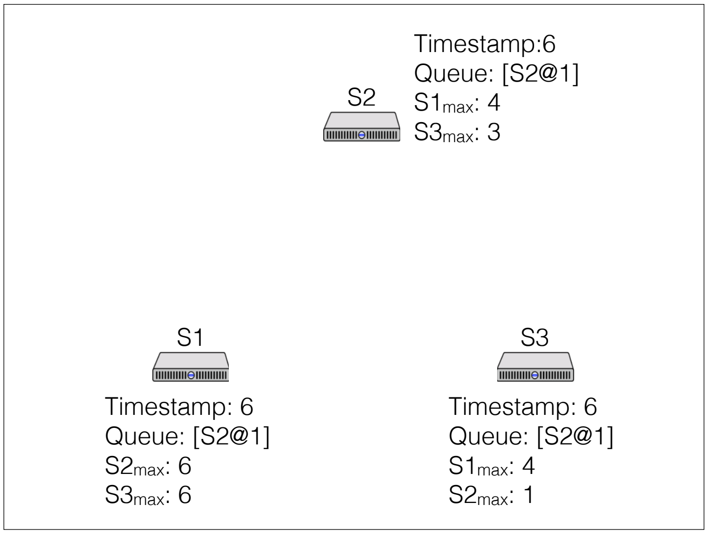
Timestamp:6
Queue: [S2@1]
$S1_{max}$: 4
$S3_{max}$: 3

S1

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

S3

Timestamp: 6
Queue: [S2@1]
$S1_{max}$: 4
$S2_{max}$: 1



S2

Timestamp:6
Queue: [S2@1]
$S1_{max}$: 4
$S3_{max}$: 3

S1

Timestamp: 6
Queue: [S2@1]
$S2_{max}$: 6
$S3_{max}$: 6

S3

Timestamp: 6
Queue: [S2@1]
$S1_{max}$: 4
$S2_{max}$: 1

Mutual exclusion as SMR

State Machine Replication (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?

Lamport paper discussion

What happens when we need to add a process?

Why is coordination necessary for locking?

Events that happened vs. might have happened

Vector clocks

Precisely represent transitive causal relationships

T(A) < *T*(B) <-> *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 != i
  - C[j] = *max*(C[j], C_m[j])

Vector Clocks

Compare vectors element by element

Provided the vectors are not identical,

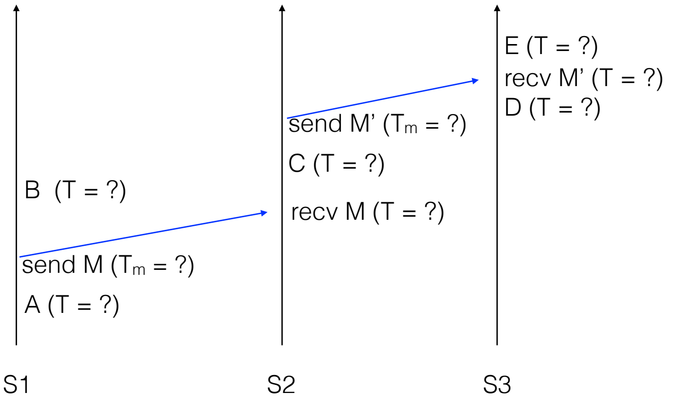
If $C_x[i] < C_y[i]$ and $C_x[j] > C_y[j]$ for some i, j

C_x and C_y are concurrent

if $C_x[i] <= C_y[i]$ for all i

C_x happens before C_y

Example



B (T = ?)

send M (T_m = ?)

A (T = ?)

send M' (T_m = ?)

C (T = ?)

recv M (T = ?)

E (T = ?)

recv M' (T = ?)

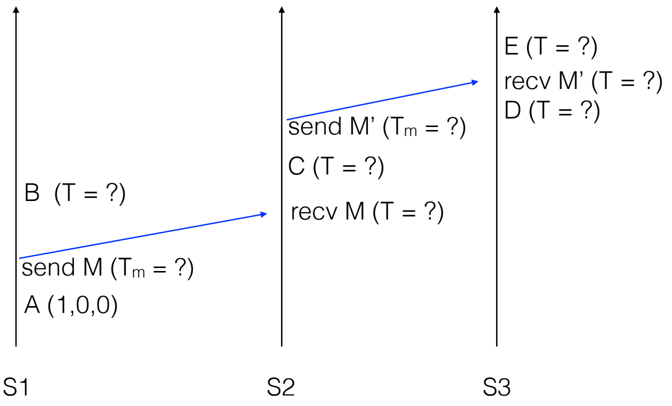
D (T = ?)

S1

S2

S3

Example



B (T = ?)

send M (T_m = ?)

A (1,0,0)

send M' (T_m = ?)

C (T = ?)

recv M (T = ?)

E (T = ?)

recv M' (T = ?)

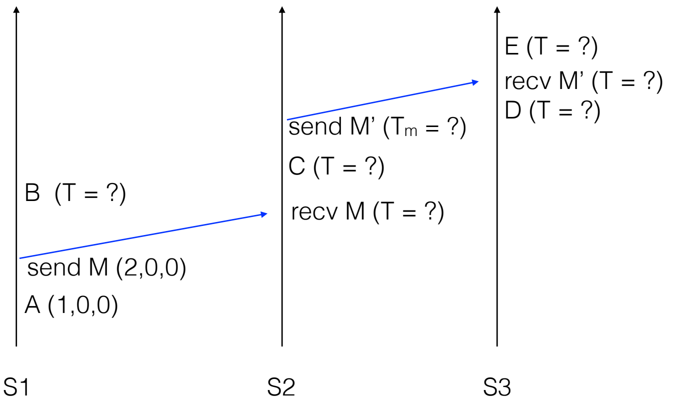
D (T = ?)

S1

S2

S3

Example



B (T = ?)

send M (2,0,0)

A (1,0,0)

send M' (T_m = ?)

C (T = ?)

recv M (T = ?)

E (T = ?)

recv M' (T = ?)

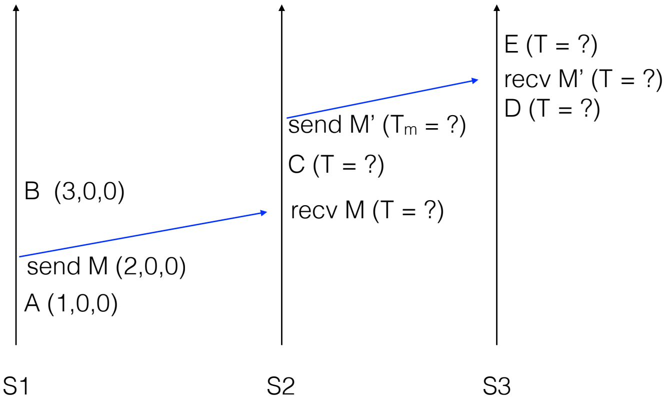
D (T = ?)

S1

S2

S3

Example



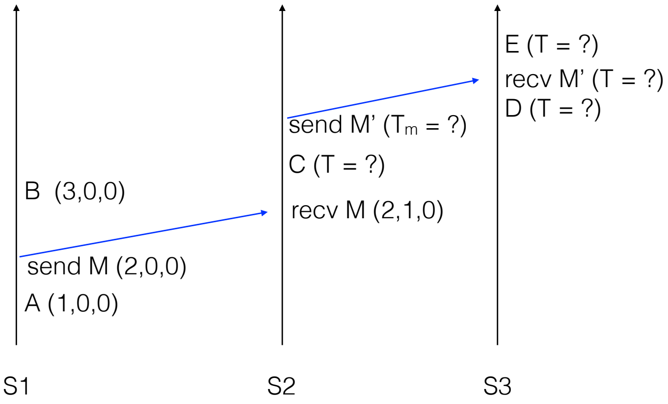
E (T = ?)
recv M’ (T = ?)
D (T = ?)

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

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

S1
S2
S3

Example



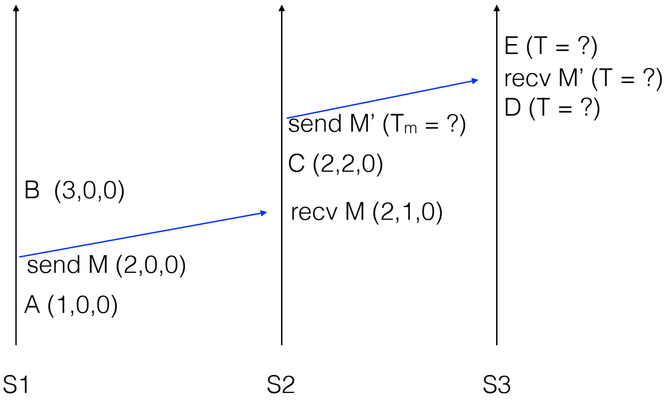
E (T = ?)
recv M’ (T = ?)
D (T = ?)

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

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

S1
S2
S3

Example



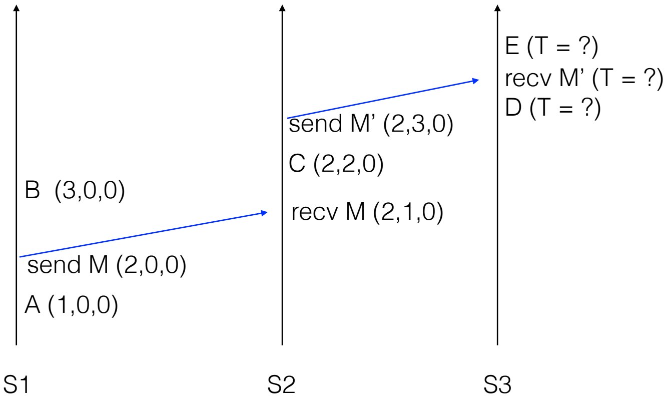
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ (T_m = ?)
C (2,2,0)
recv M (2,1,0)

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

S1
S2
S3

Example



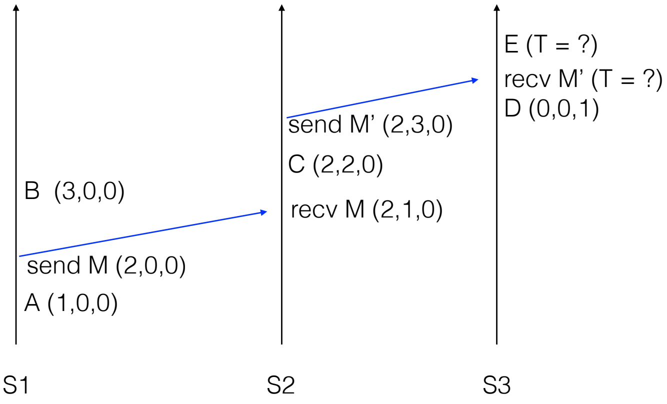
E (T = ?)
recv M’ (T = ?)
D (T = ?)

send M’ (2,3,0)
C (2,2,0)
recv M (2,1,0)

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

S1
S2
S3

Example



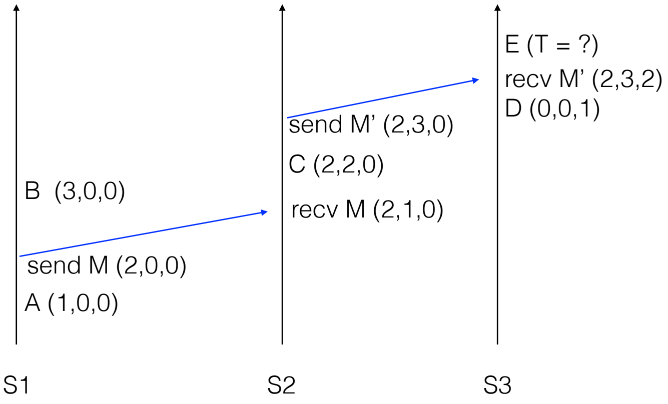
E (T = ?)
recv M’ (T = ?)
D (0,0,1)

send M’ (2,3,0)
C (2,2,0)
recv M (2,1,0)

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

S1
S2
S3

Example



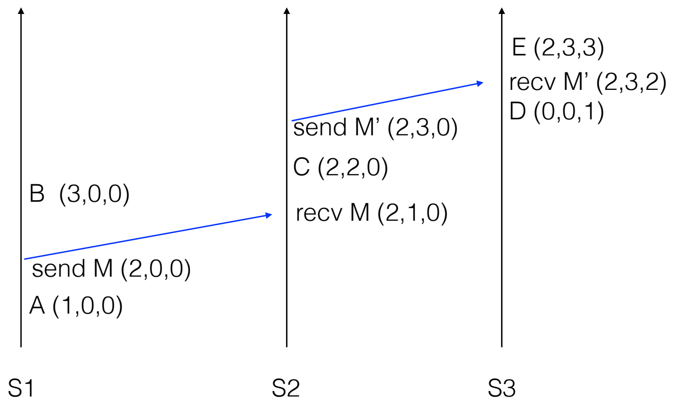
E (T = ?)
recv M’ (2,3,2)
D (0,0,1)

send M’ (2,3,0)
C (2,2,0)
recv M (2,1,0)

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

S1
S2
S3

Example



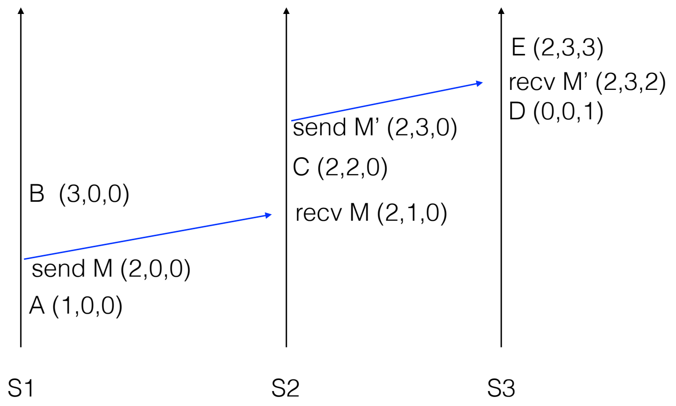
E (2,3,3)
recv M’ (2,3,2)
D (0,0,1)

send M’ (2,3,0)
C (2,2,0)
recv M (2,1,0)

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

S1 S2 S3

Example

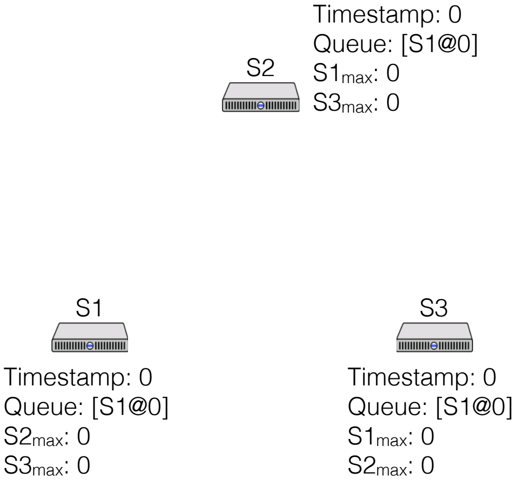


E (2,3,3)
recv M’ (2,3,2)
D (0,0,1)

send M’ (2,3,0)
C (2,2,0)
recv M (2,1,0)

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

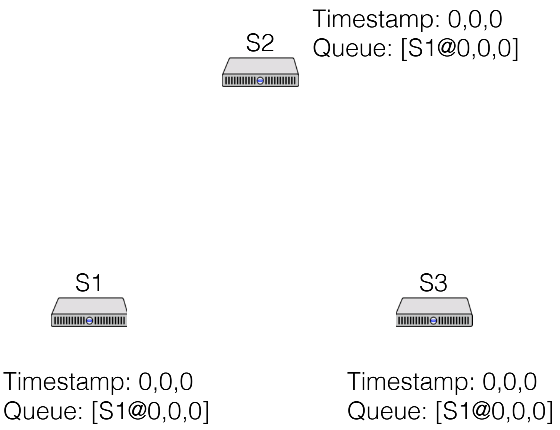
S1 S2 S3



S2
Timestamp: 0
Queue: [S1@0]
$S1_{max}$: 0
$S3_{max}$: 0

S1
Timestamp: 0
Queue: [S1@0]
$S2_{max}$: 0
$S3_{max}$: 0

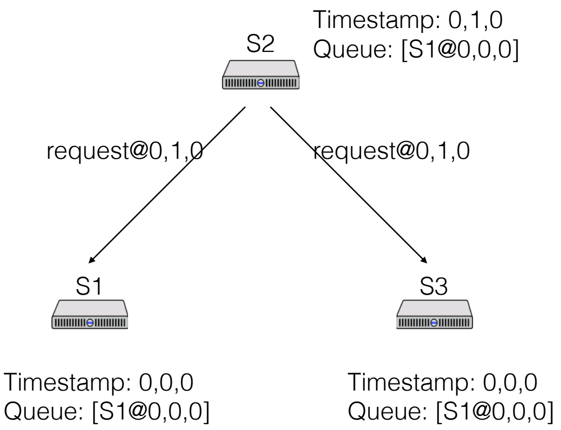
S3
Timestamp: 0
Queue: [S1@0]
$S1_{max}$: 0
$S2_{max}$: 0



S2
Timestamp: 0,0,0
Queue: [S1@0,0,0]

S1
Timestamp: 0,0,0
Queue: [S1@0,0,0]

S3
Timestamp: 0,0,0
Queue: [S1@0,0,0]

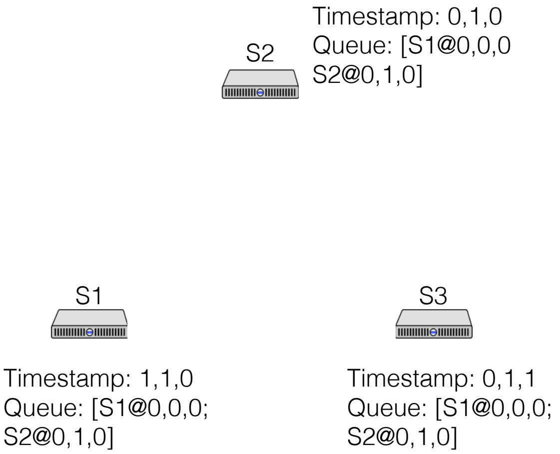


S2
Timestamp: 0,1,0
Queue: [S1@0,0,0]

request@0,1,0
request@0,1,0

S1
Timestamp: 0,0,0
Queue: [S1@0,0,0]

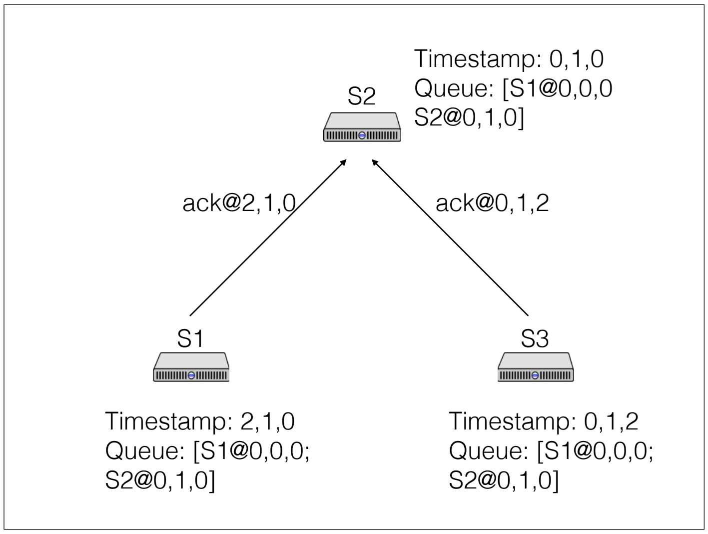
S3
Timestamp: 0,0,0
Queue: [S1@0,0,0]

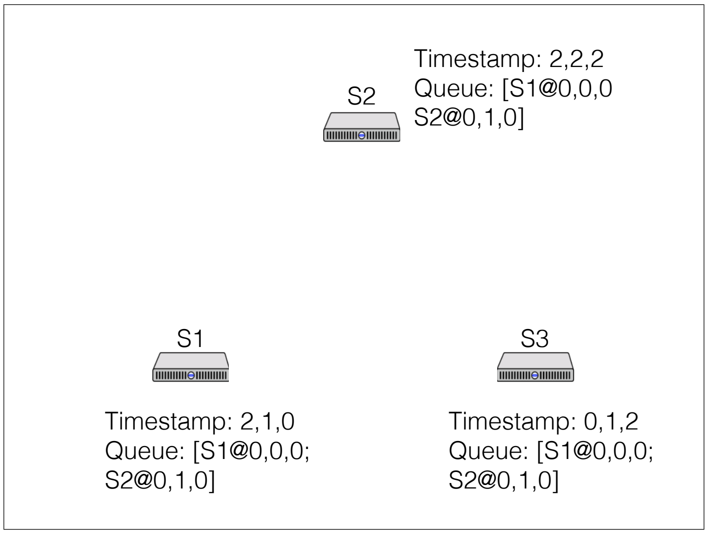


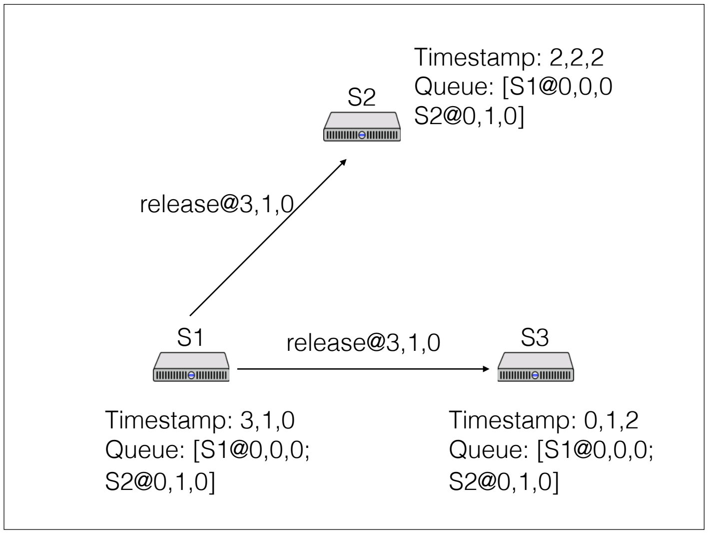
S2
Timestamp: 0,1,0
Queue: [S1@0,0,0
S2@0,1,0]

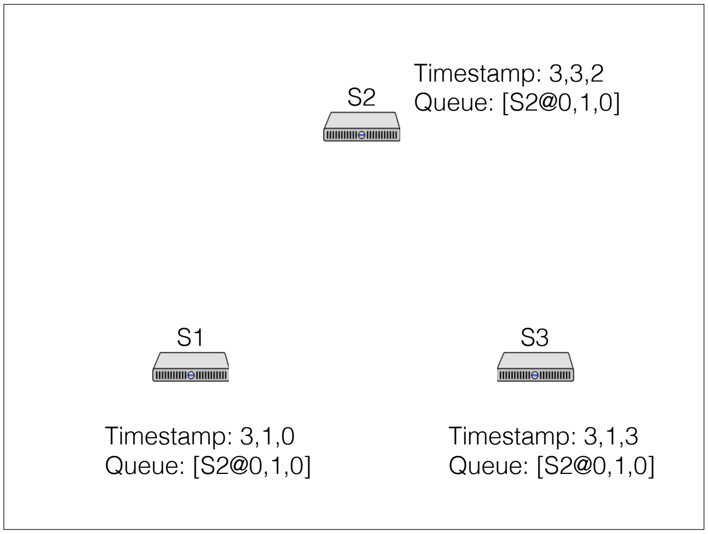
S1
Timestamp: 1,1,0
Queue: [S1@0,0,0;
S2@0,1,0]

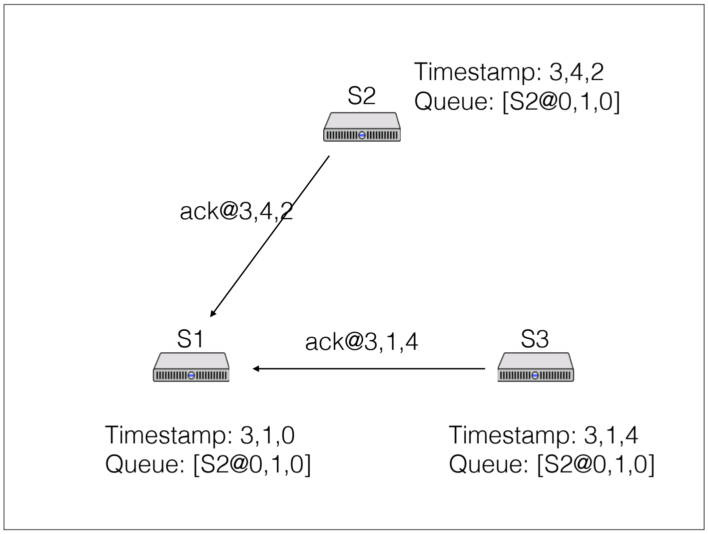
S3
Timestamp: 0,1,1
Queue: [S1@0,0,0;
S2@0,1,0]

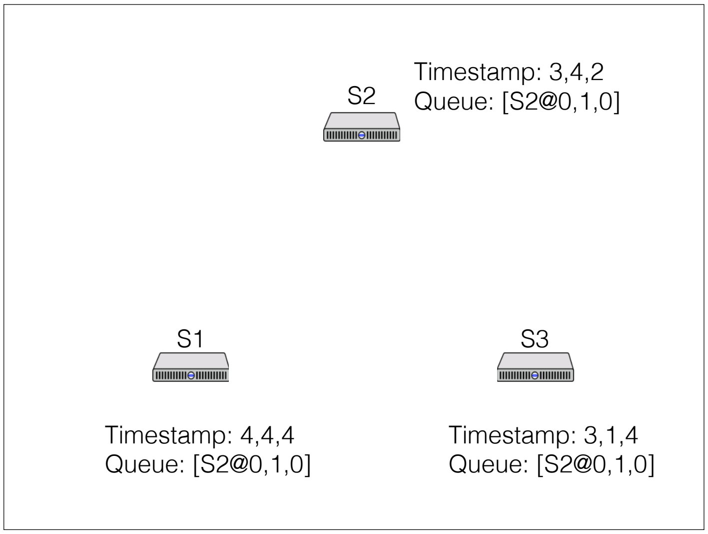

S2
Timestamp: 0,1,0
Queue: [S1@0,0,0
S2@0,1,0]
ack@2,1,0
ack@0,1,2
S1
S3
Timestamp: 2,1,0
Queue: [S1@0,0,0;
S2@0,1,0]
Timestamp: 0,1,2
Queue: [S1@0,0,0;
S2@0,1,0]


S2
Timestamp: 2,2,2
Queue: [S1@0,0,0
S2@0,1,0]
S1
S3
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]


S2
Timestamp: 2,2,2
Queue: [S1@0,0,0
S2@0,1,0]
release@3,1,0
S1
release@3,1,0
S3
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]


S2
Timestamp: 3,3,2
Queue: [S2@0,1,0]
S1
S3
Timestamp: 3,1,0
Queue: [S2@0,1,0]
Timestamp: 3,1,3
Queue: [S2@0,1,0]


S2
Timestamp: 3,4,2
Queue: [S2@0,1,0]
ack@3,4,2
S1
ack@3,1,4
S3
Timestamp: 3,1,0
Queue: [S2@0,1,0]
Timestamp: 3,1,4
Queue: [S2@0,1,0]


S2
Timestamp: 3,4,2
Queue: [S2@0,1,0]
S1
S3
Timestamp: 4,4,4
Queue: [S2@0,1,0]
Timestamp: 3,1,4
Queue: [S2@0,1,0]