Lamport Clocks

CSE 452

Lamport Clocks

Framework for *reasoning* about event ordering

- notion of logical time vs. physical time
- leads to causal ordering, 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

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

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

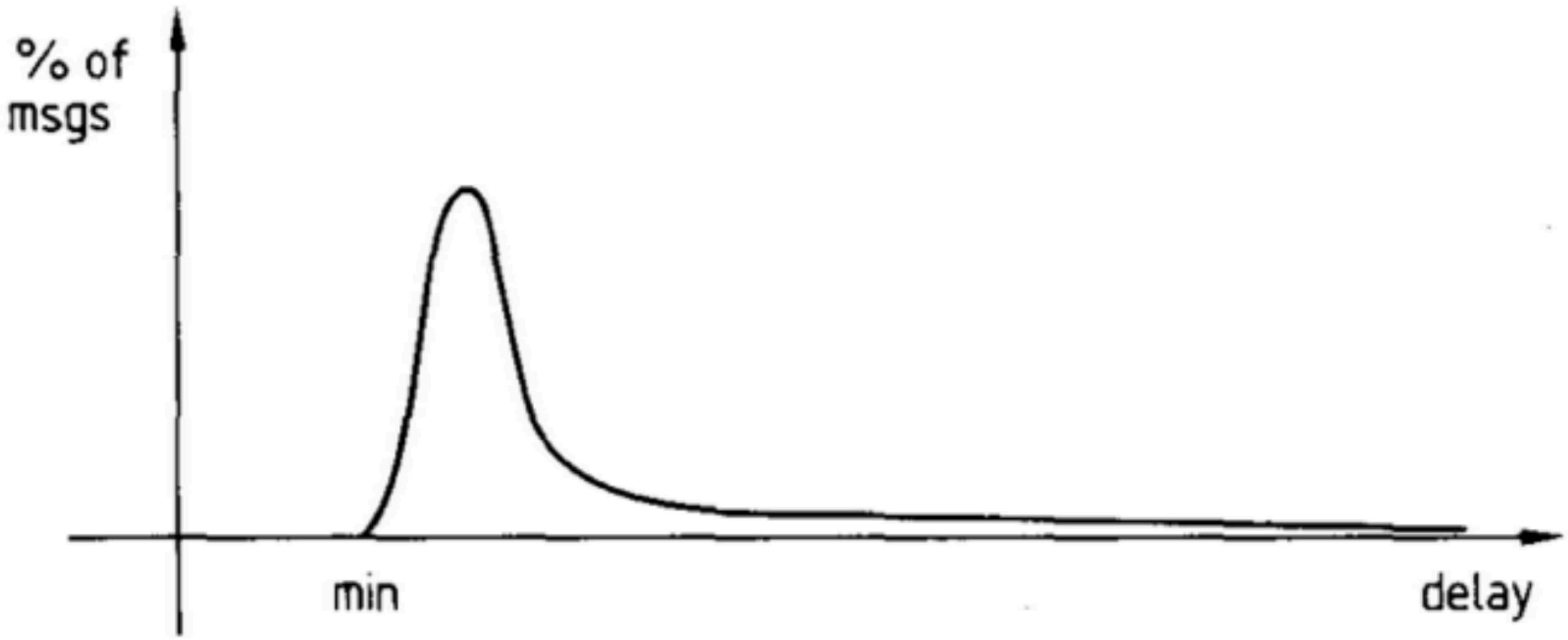
Master periodically broadcasts time

Clients receive broadcast, reset their clock

- Taking care so time never runs backwards

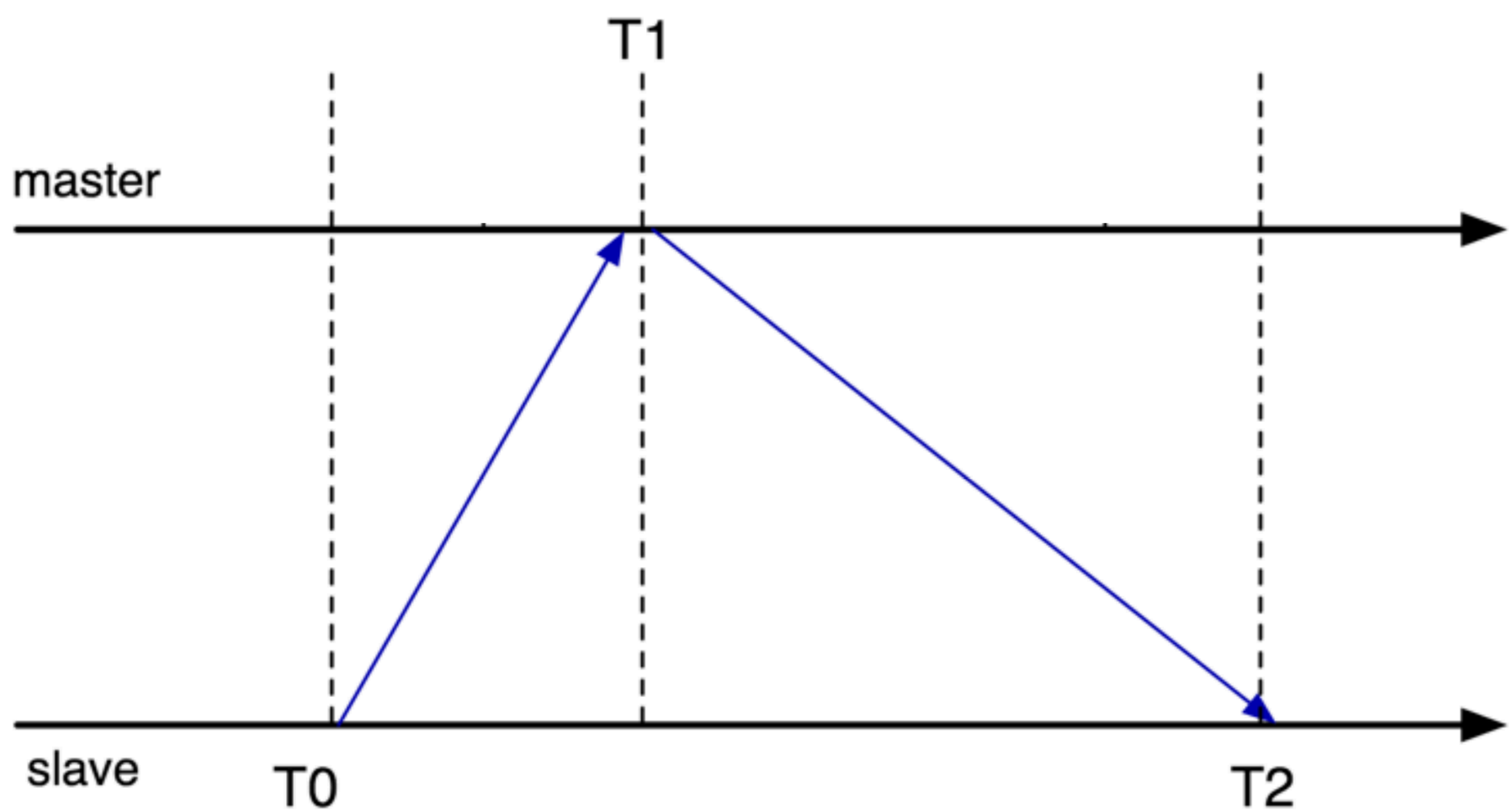
How well does this work?

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

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

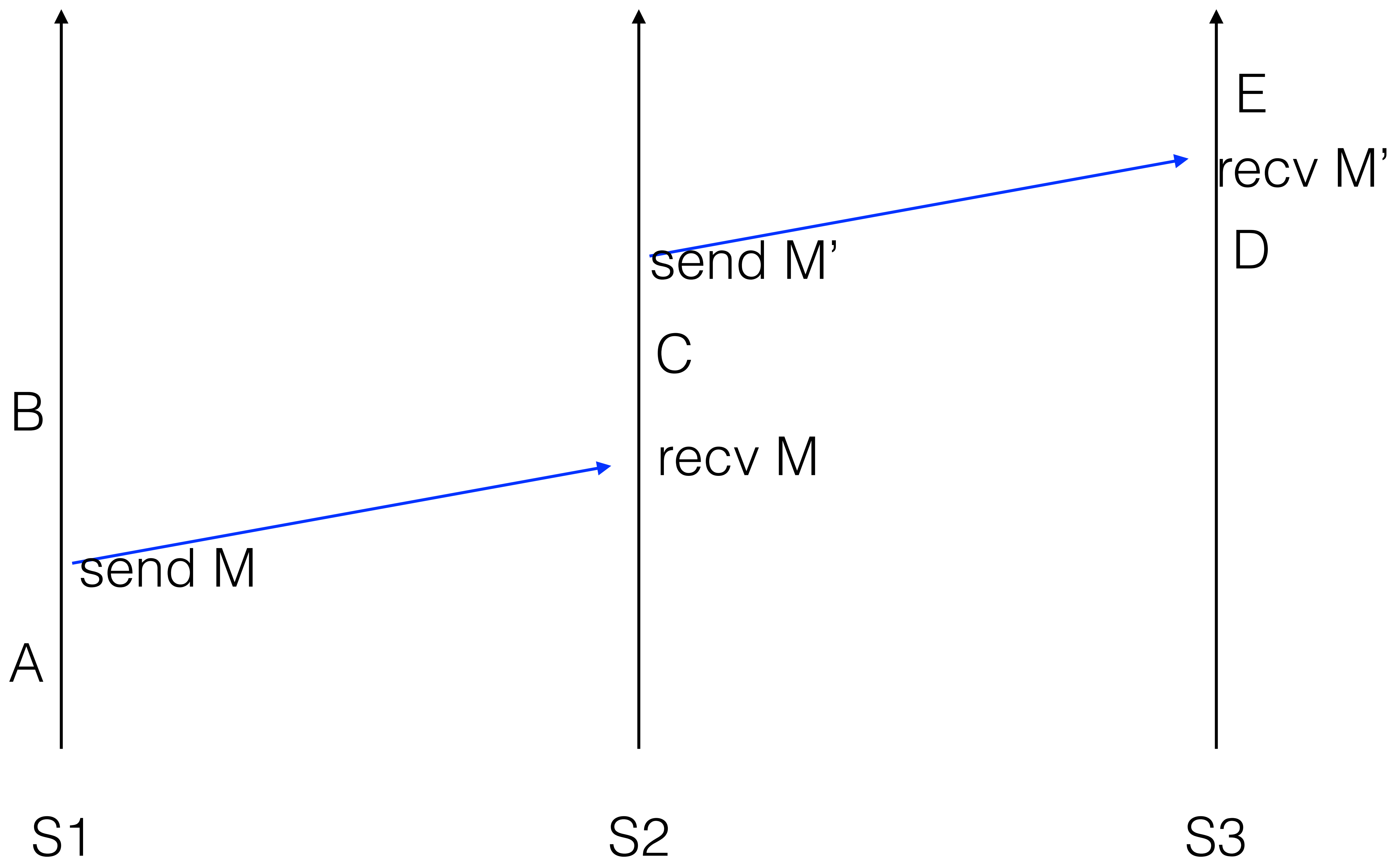
- Globally valid, such that it respects causality
- Using only local information
- No physical clock

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



Logical clock implementation

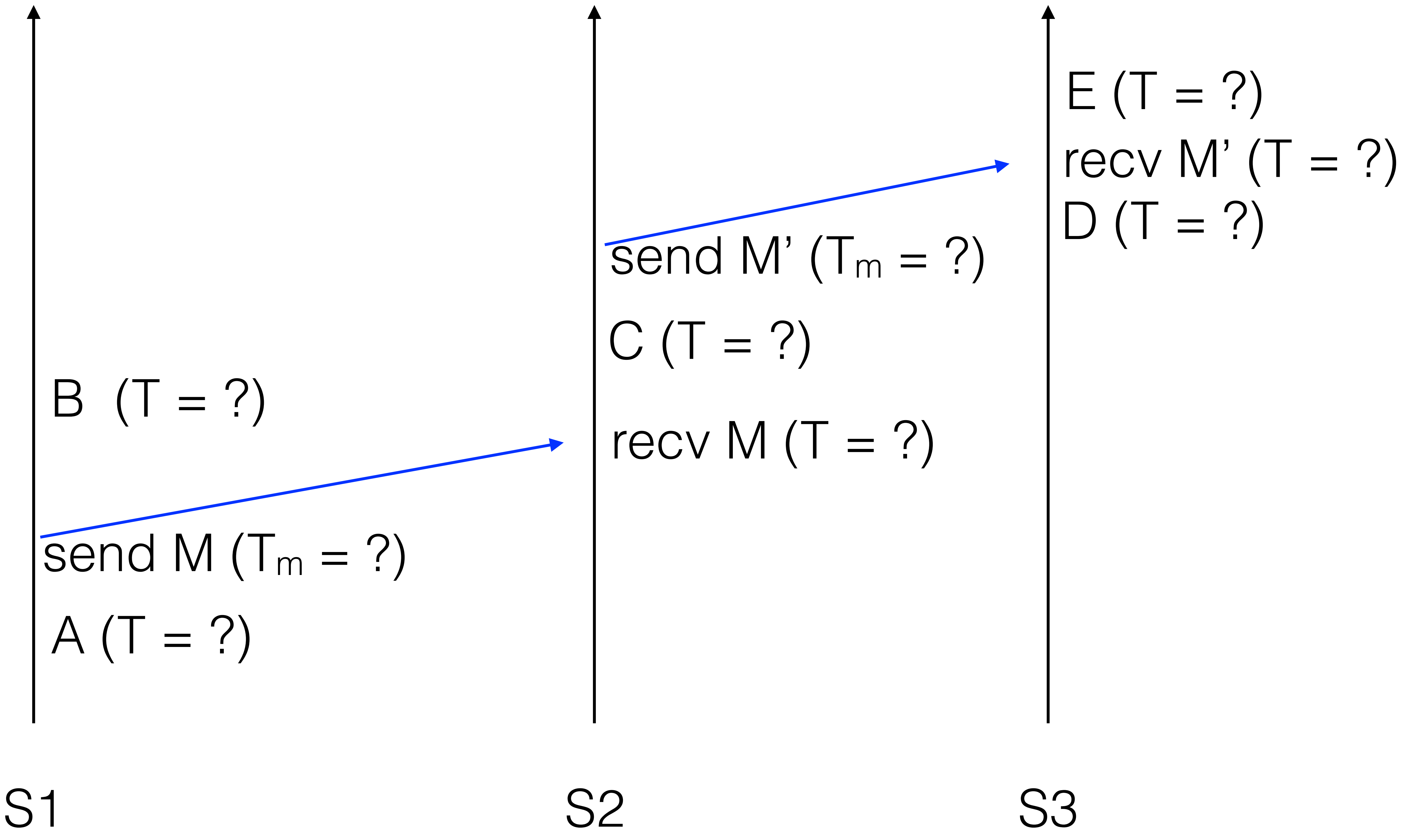
Keep a local clock T

Increment T whenever an event happens

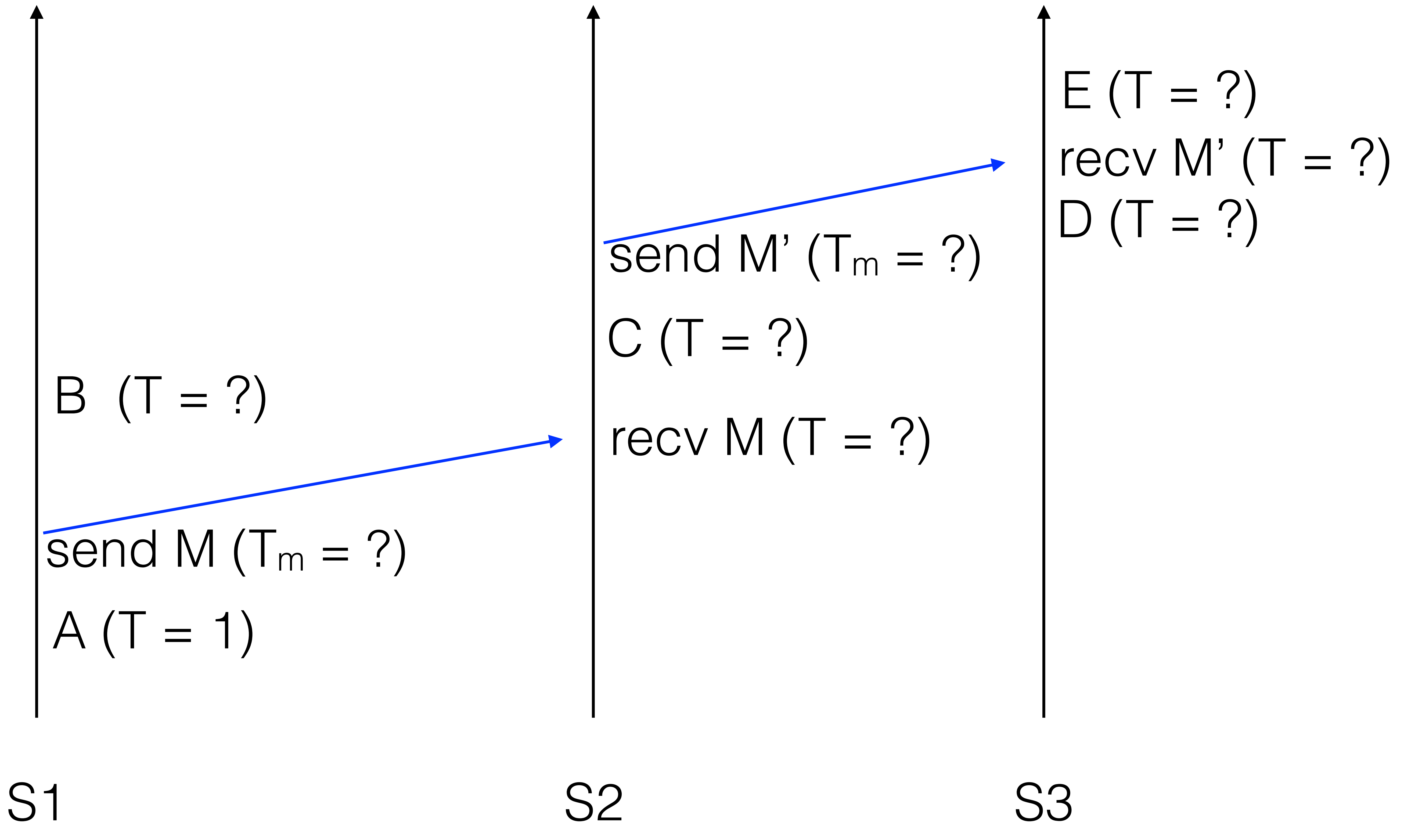
Send clock value on all messages as T_m

On message receipt: $T = max(T, T_m) + 1$

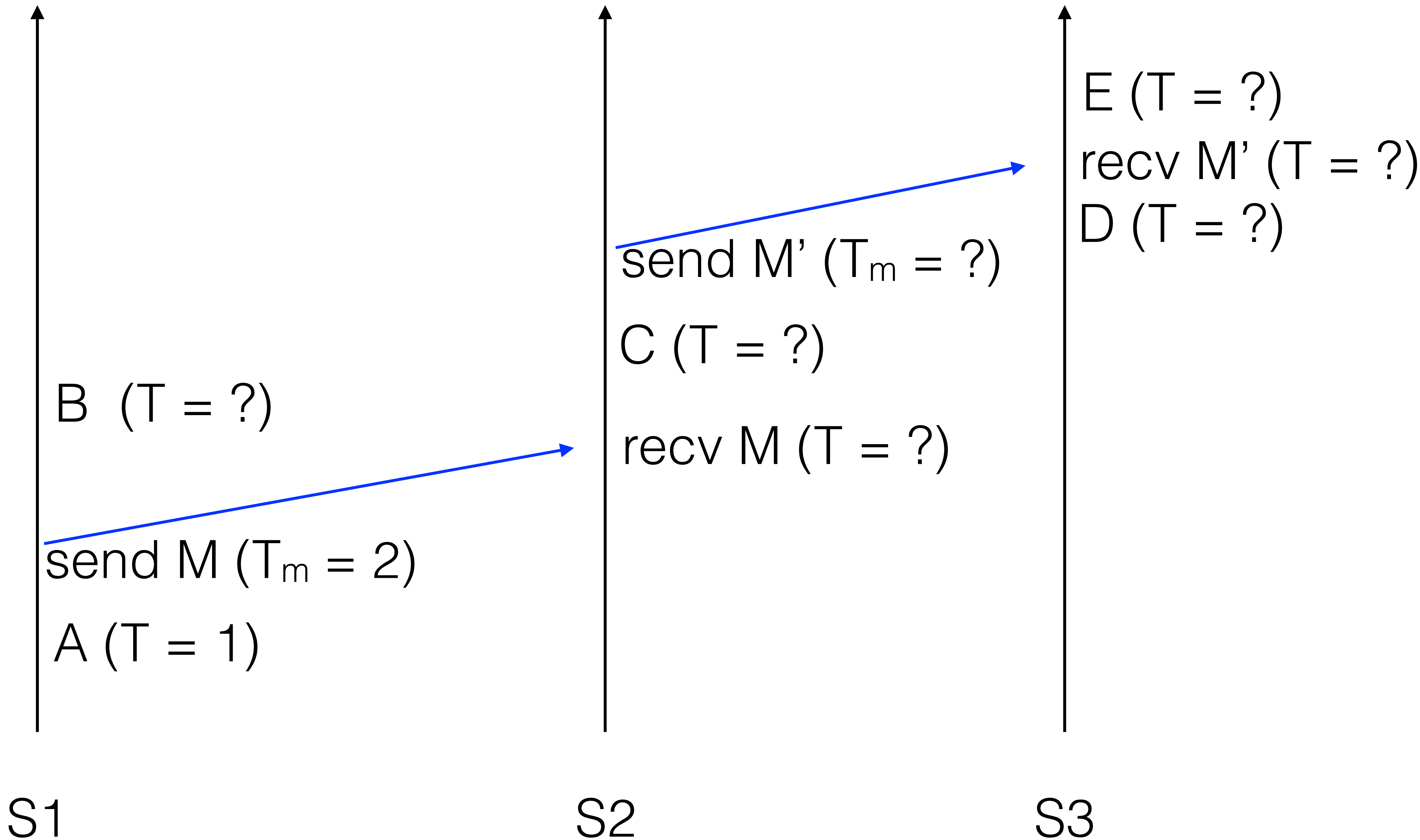
Example



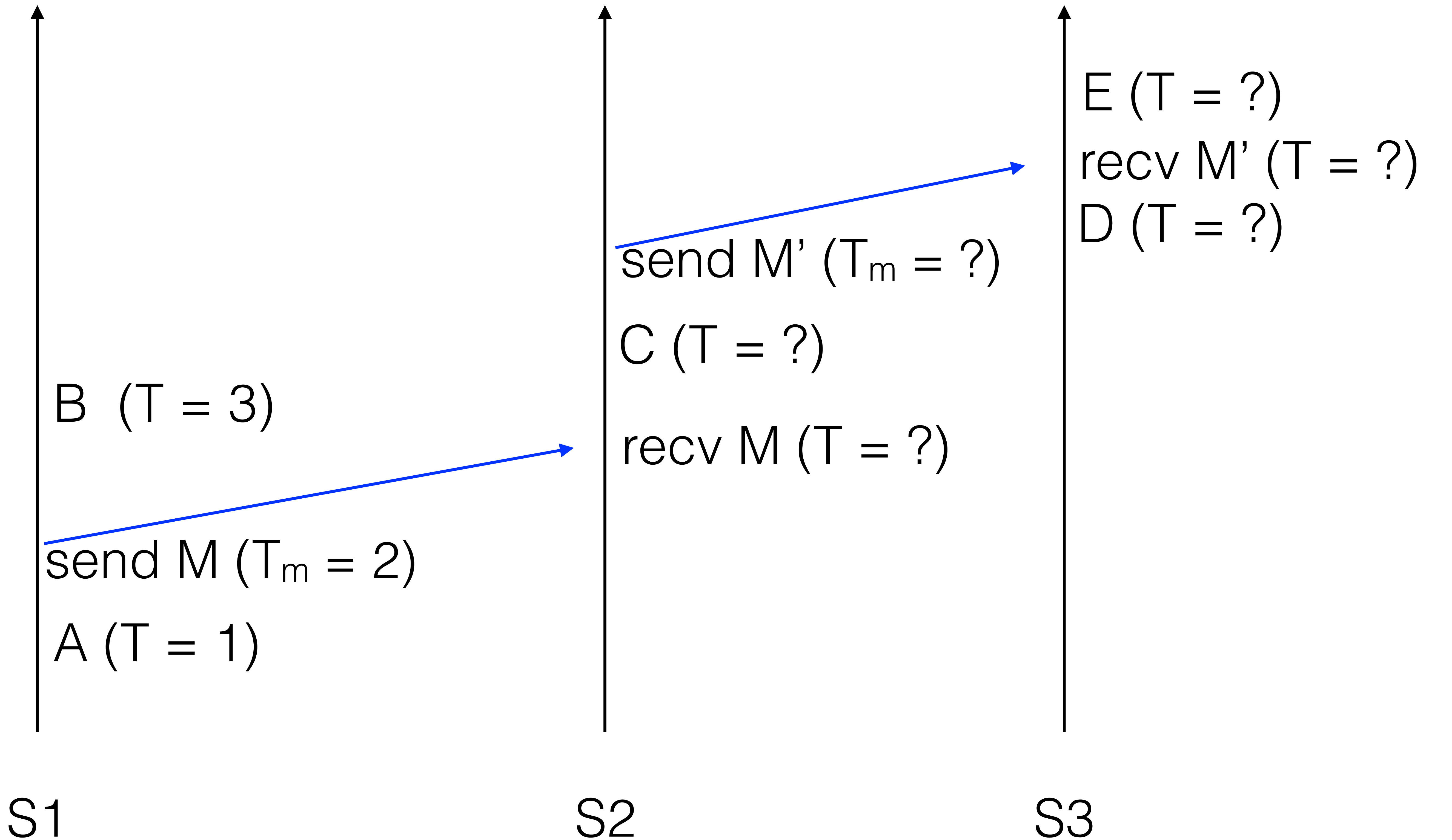
Example



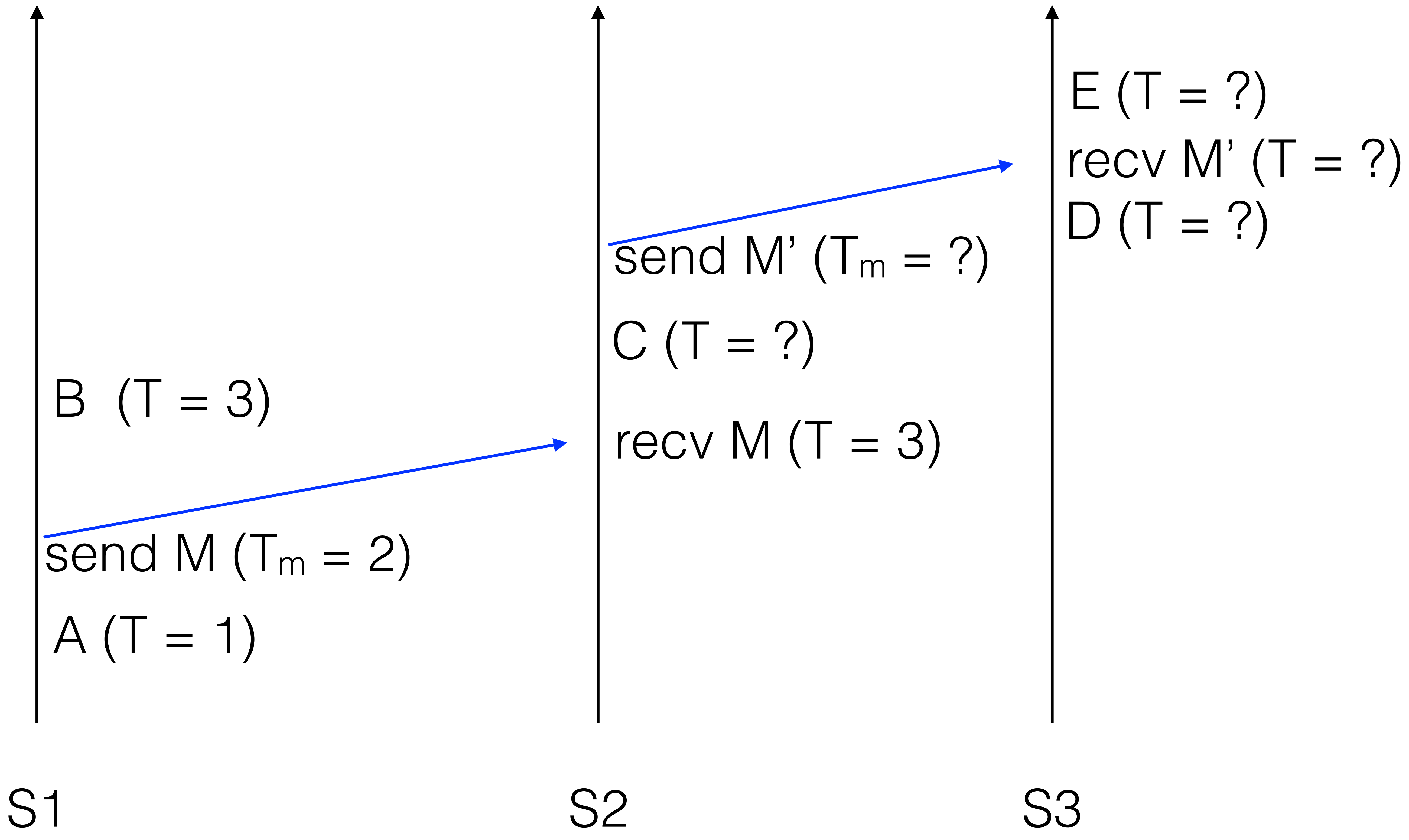
Example



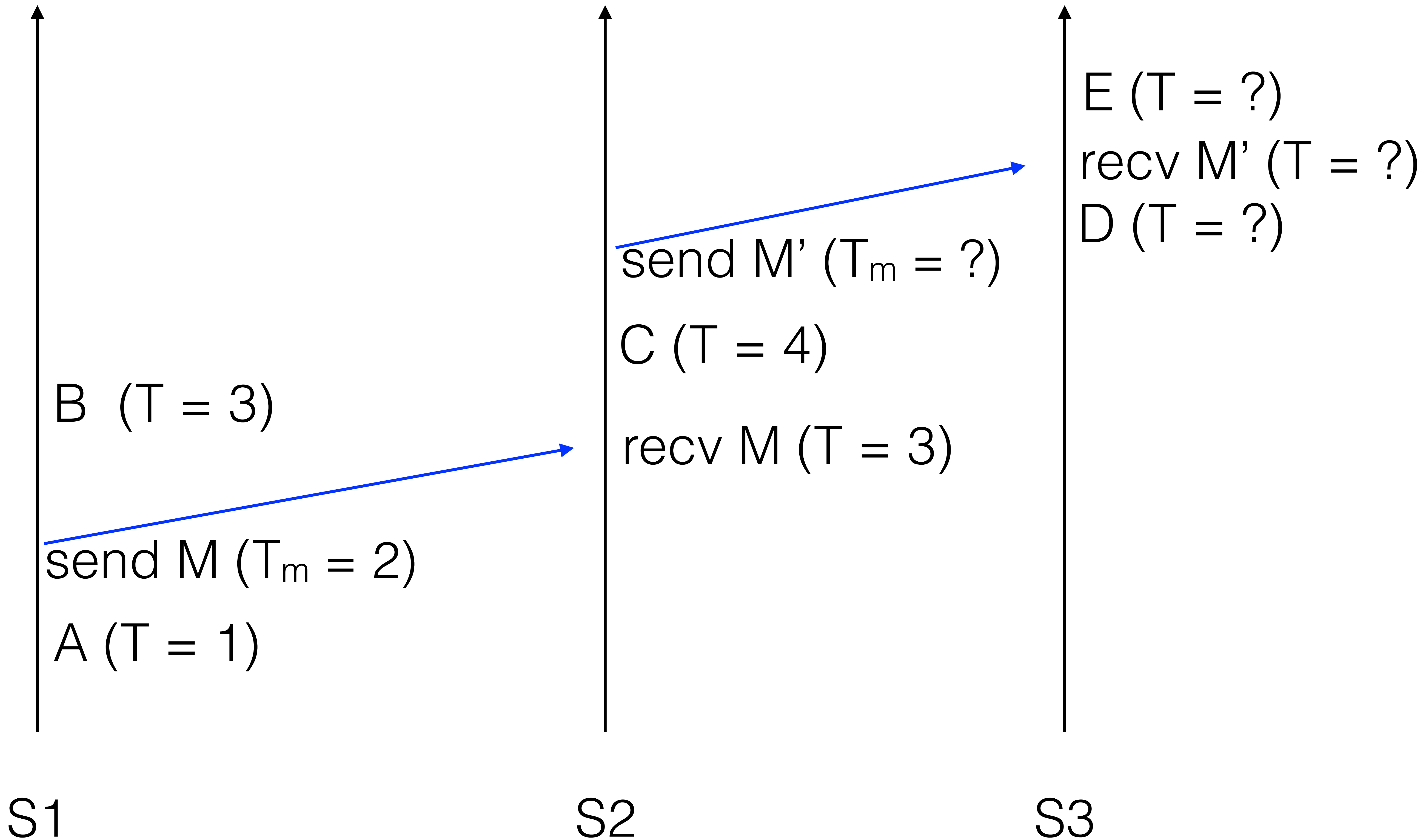
Example



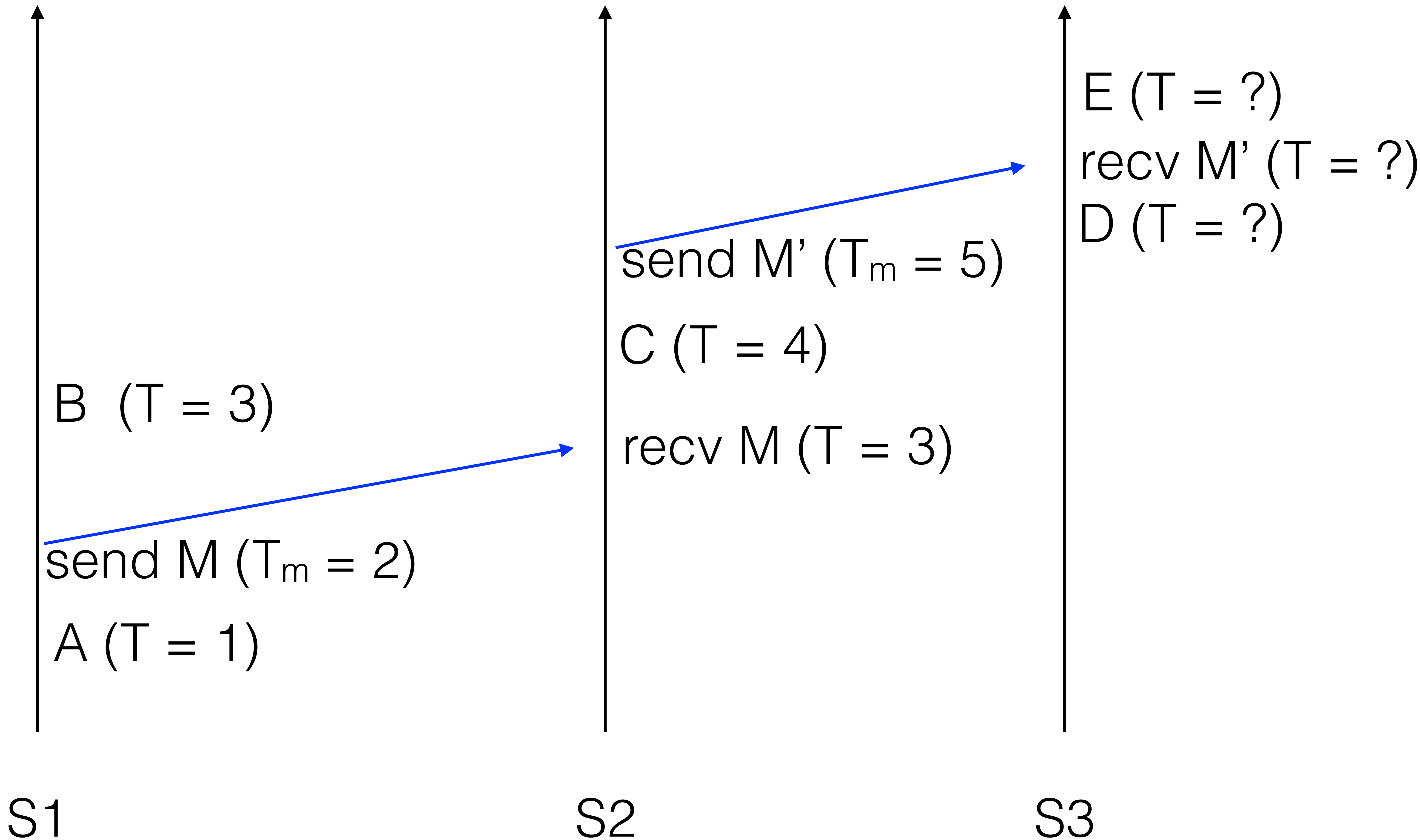
Example



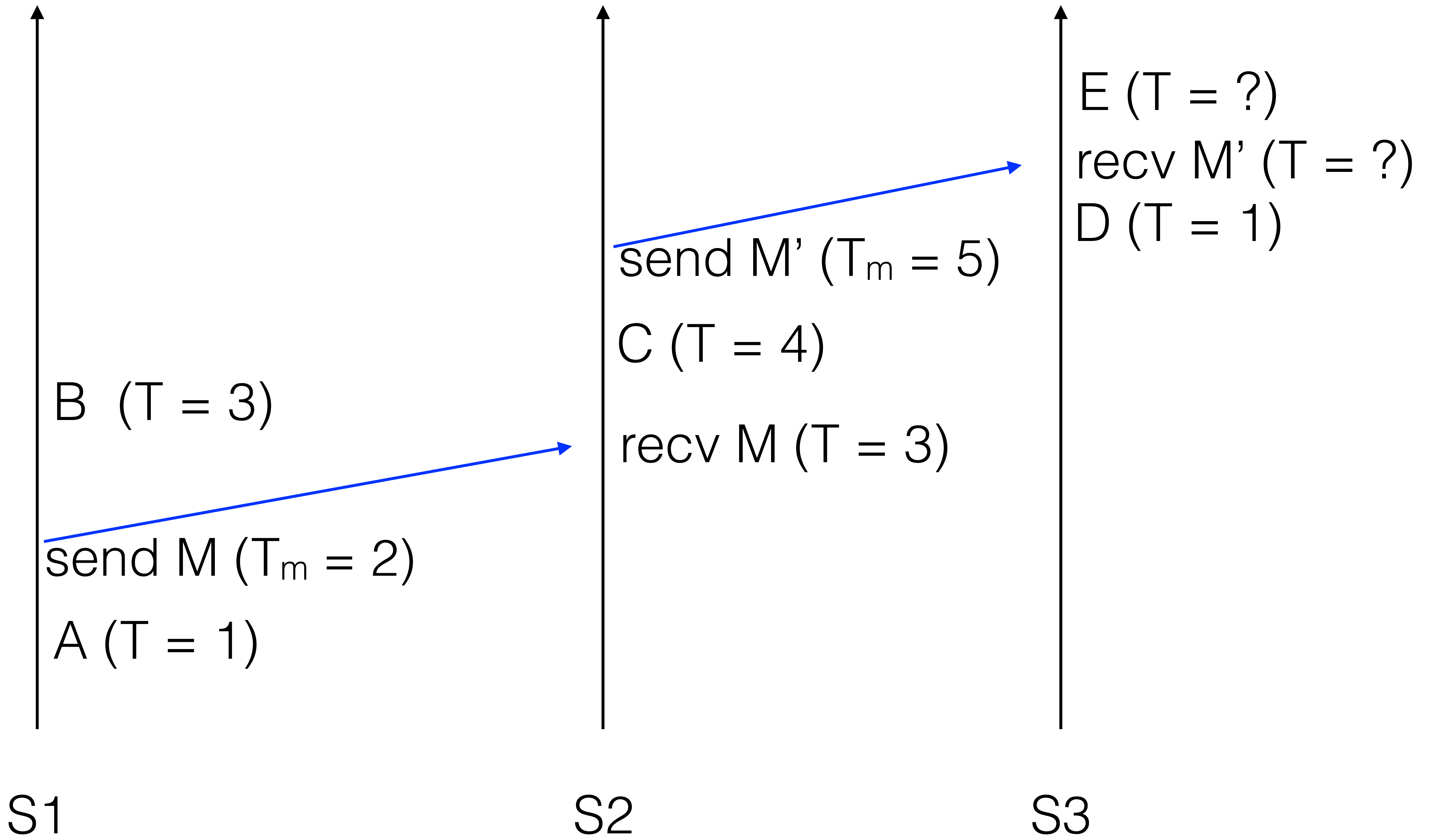
Example



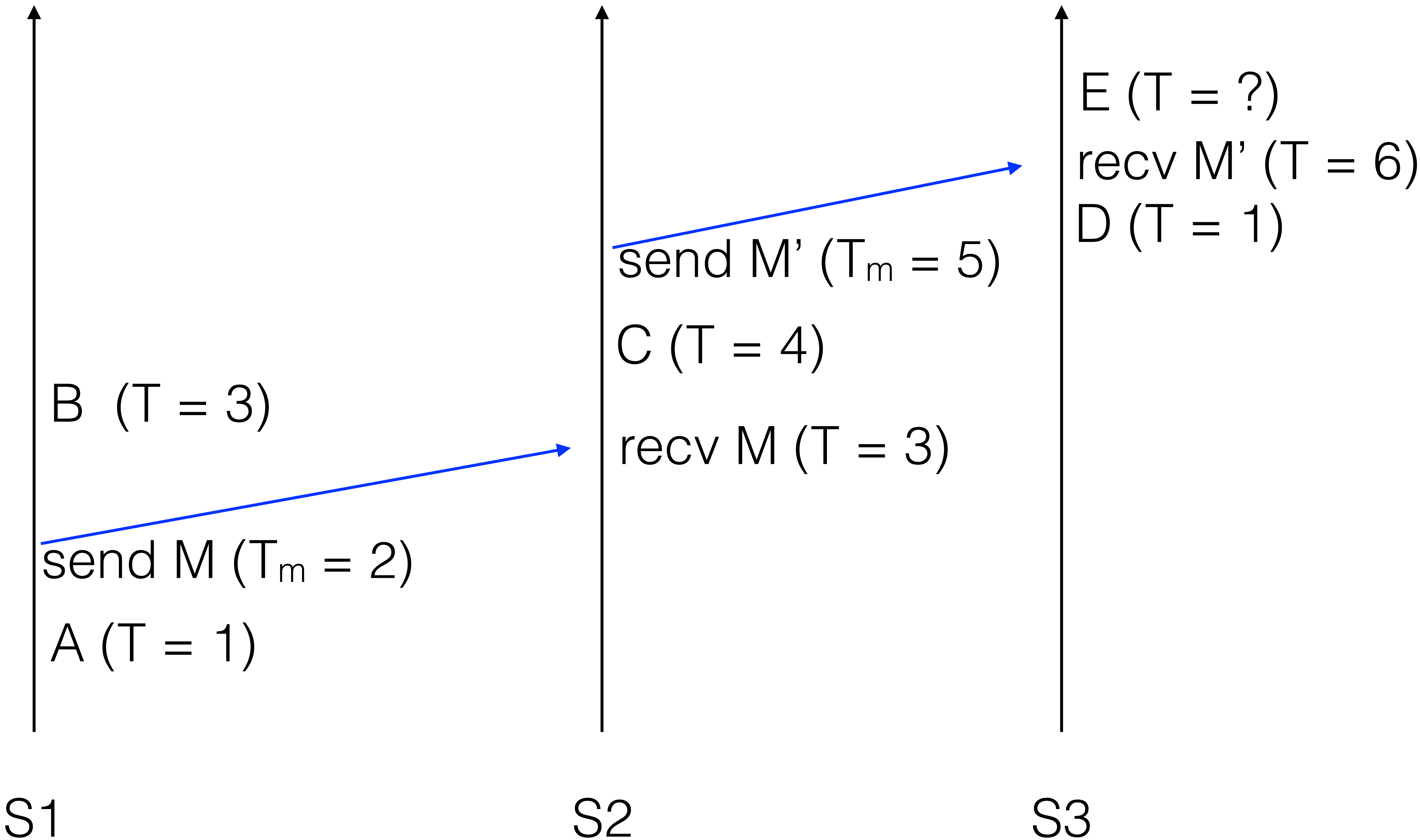
Example



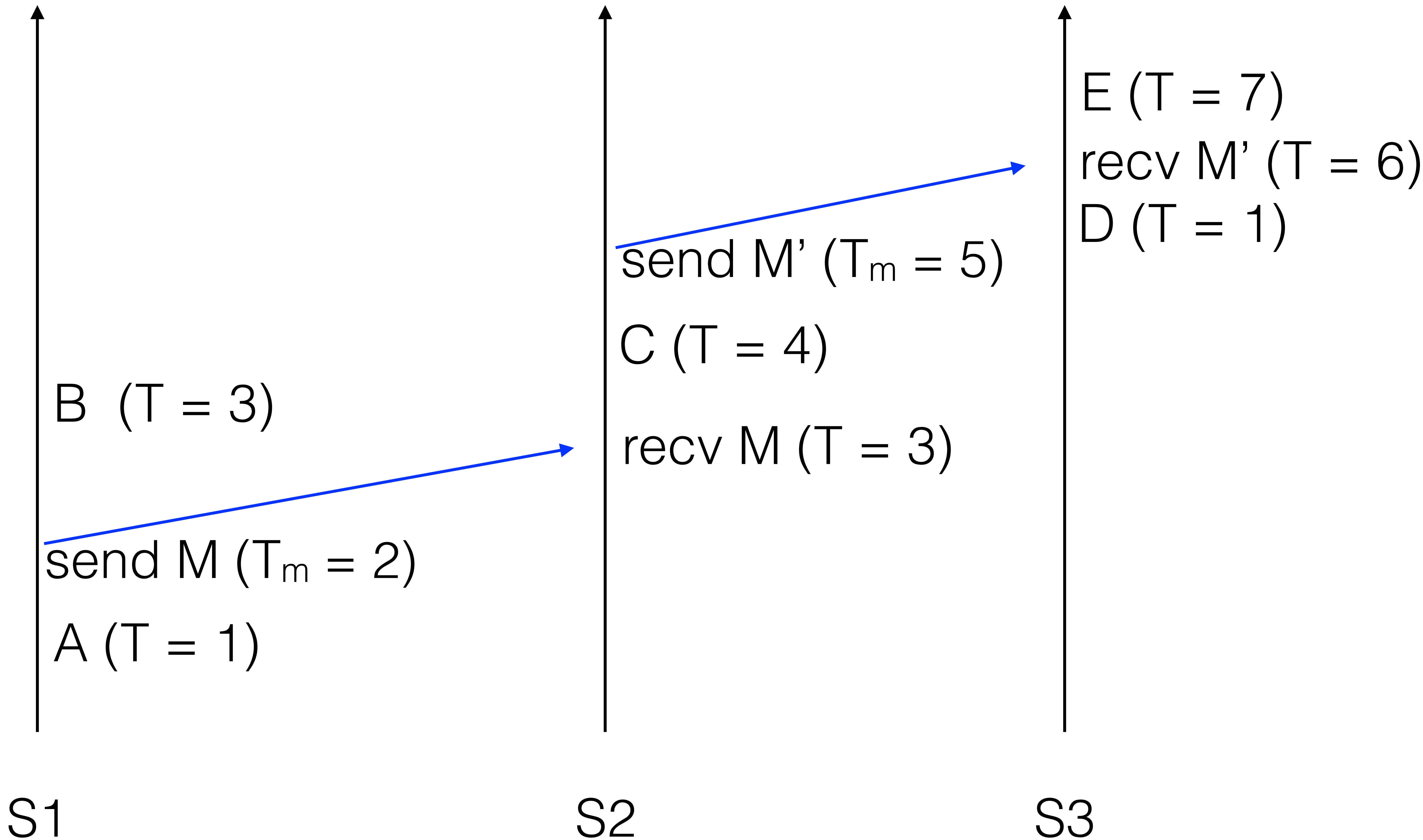
Example



Example


E (T = ?)
recv M’ (T = 6)
D (T = 1)
send M’ (T_m = 5)
C (T = 4)
B (T = 3)
recv M (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’ (Tm = 5)
C (T = 4)
B (T = 3)
recv M (T = 3)
send M (Tm = 2)
A (T = 1)
S1
S2
S3

Goal of a logical clock

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

What about the converse?

I.e., if T(A) < T(B) then what?

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

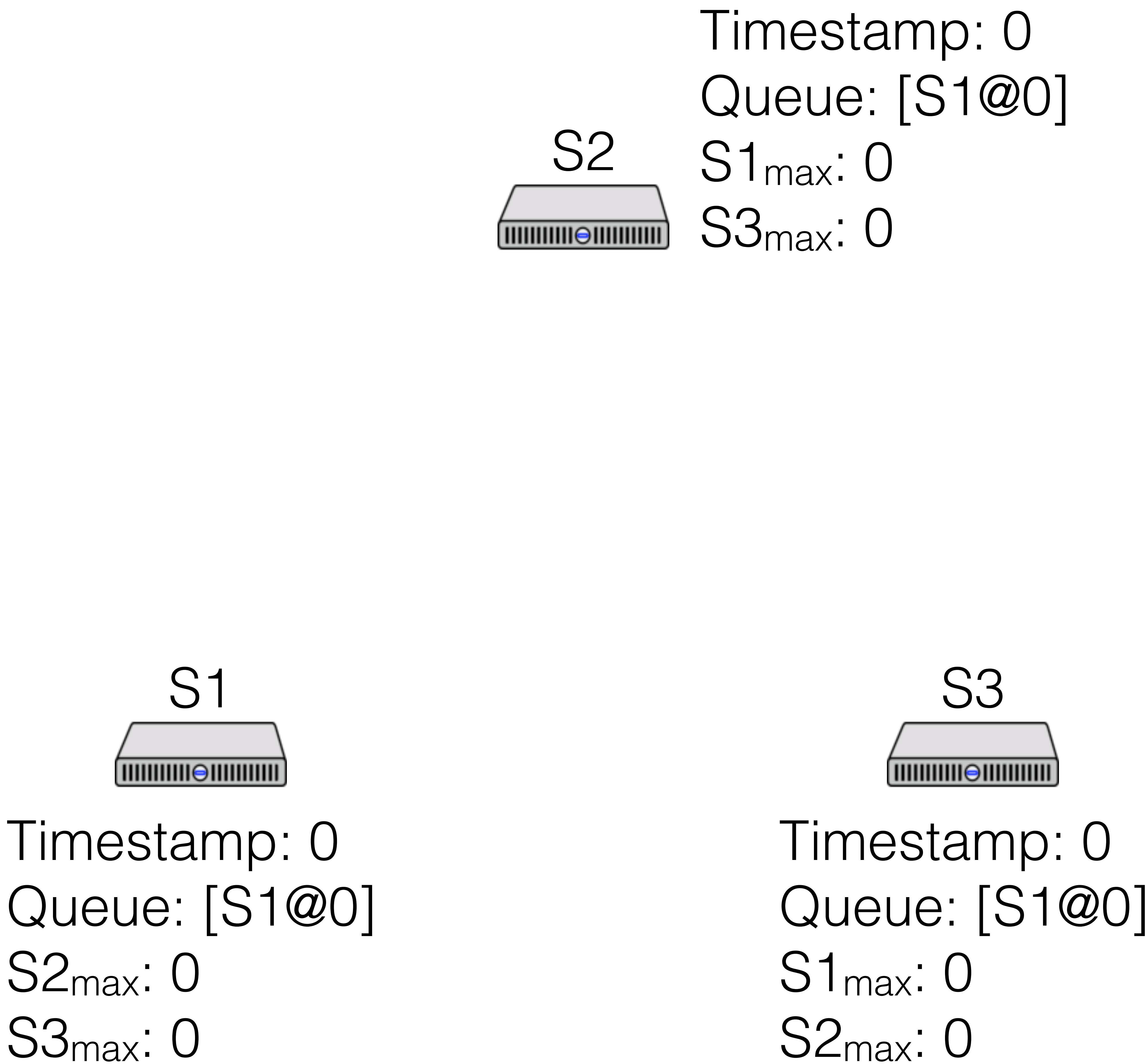
On receiving an *acknowledge*:

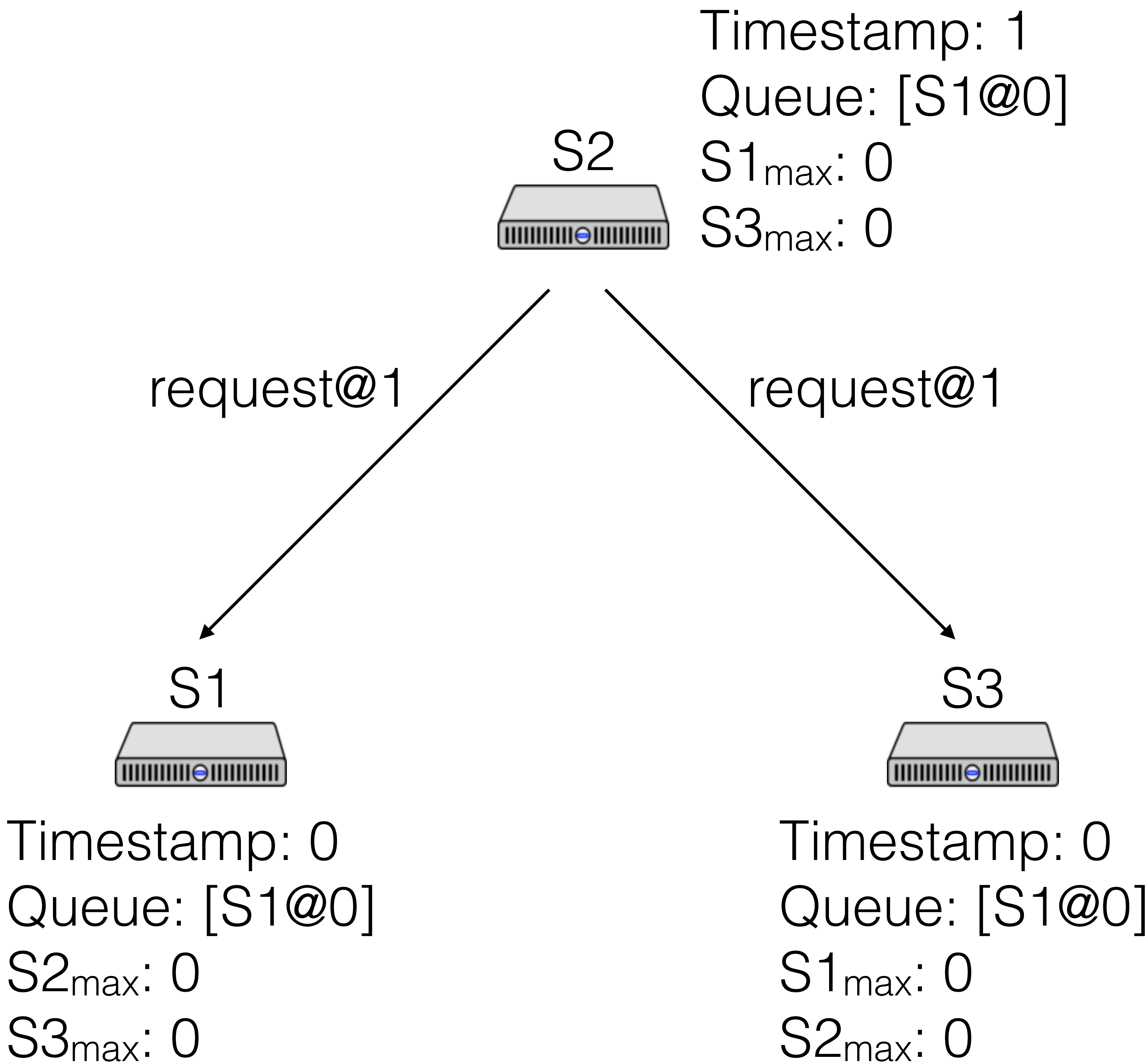
- Record message timestamp

Mutual exclusion implementation

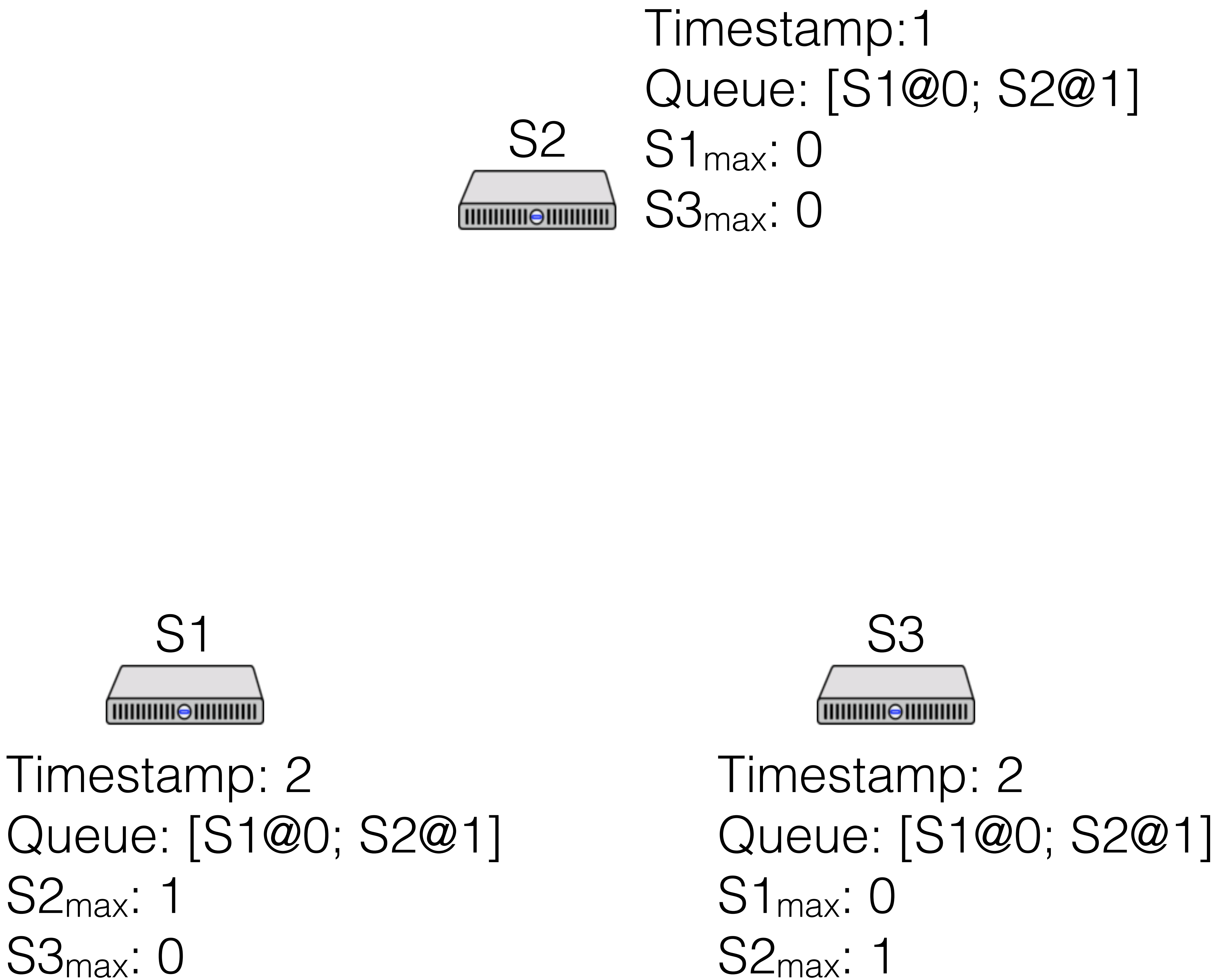
To acquire the lock:

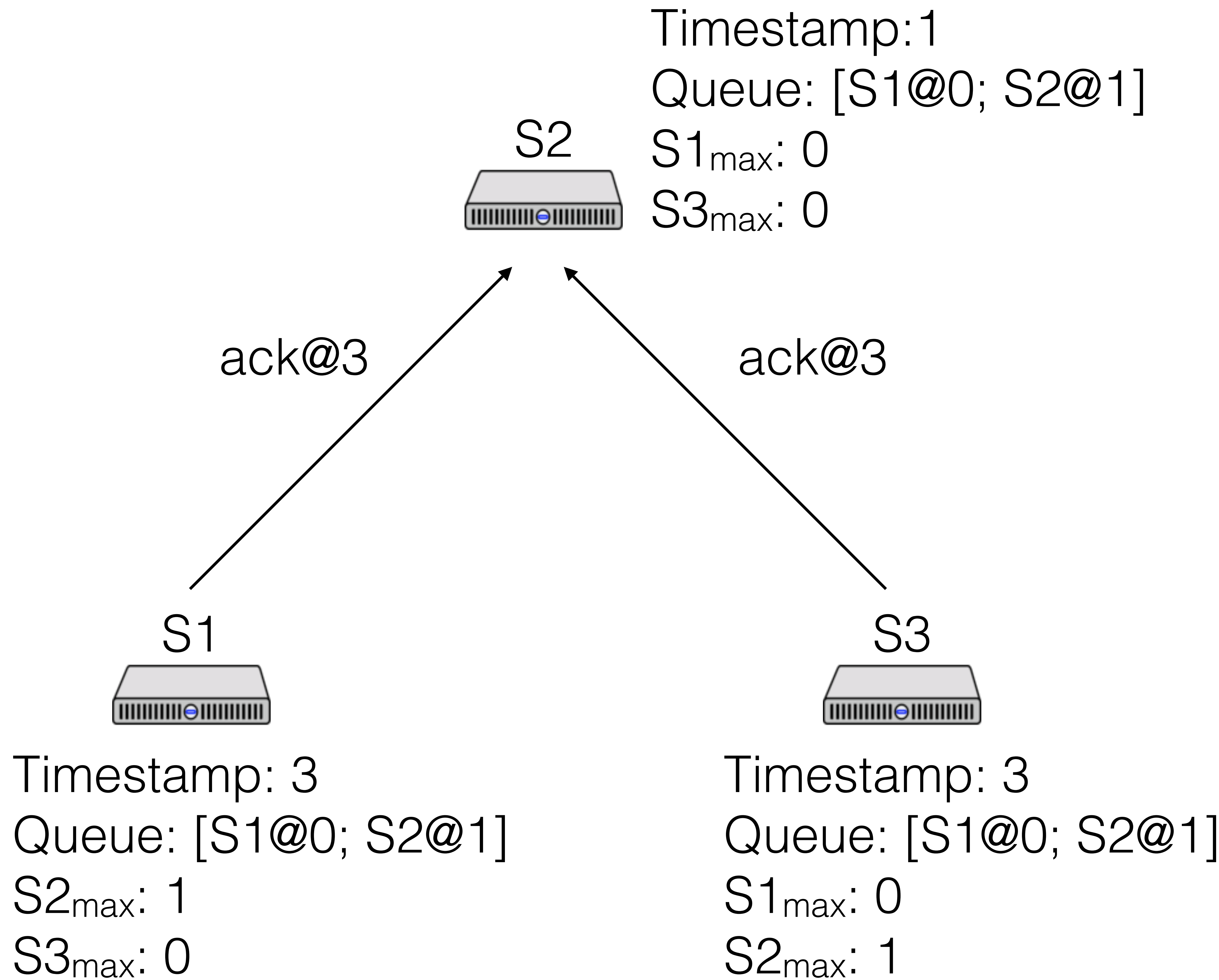
- Send *request* to everyone, including self
- The lock is acquired when:
  - My request is at the head of my queue, and
  - I've received same or higher-timestamped messages from everyone
  - So my request must be the earliest

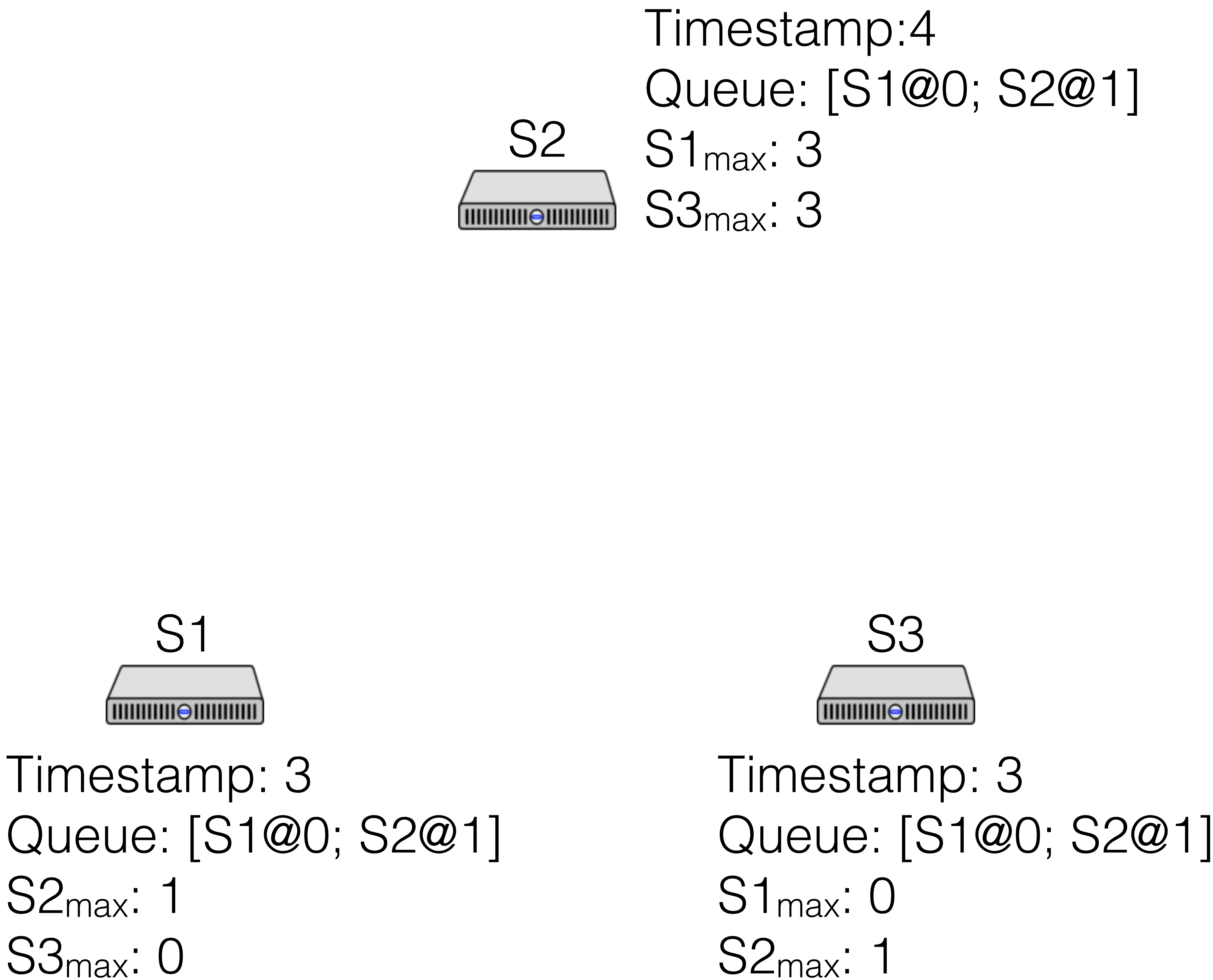

S2
Timestamp: 0
Queue: [S1@0]
S1max: 0
S3max: 0
S1
Timestamp: 0
Queue: [S1@0]
S2max: 0
S3max: 0
S3
Timestamp: 0
Queue: [S1@0]
S1max: 0
S2max: 0



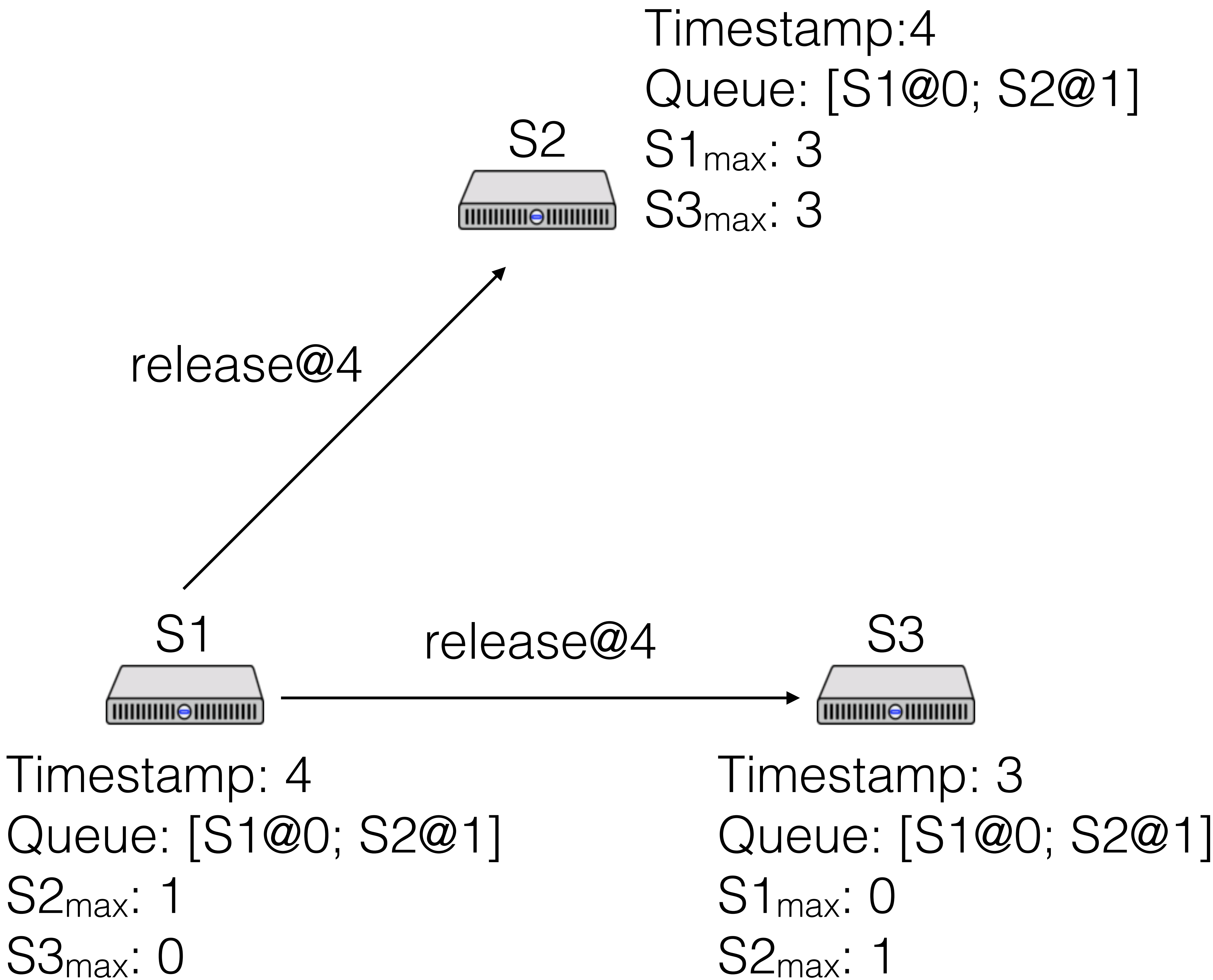
Timestamp: 1
Queue: [S1@0]
S2
$S1_{max}$: 0
$S3_{max}$: 0
request@1
request@1
S1
S3
Timestamp: 0
Queue: [S1@0]
$S2_{max}$: 0
$S3_{max}$: 0
Timestamp: 0
Queue: [S1@0]
$S1_{max}$: 0
$S2_{max}$: 0


S2
Timestamp:1
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S3_{max}$: 0
S1
Timestamp: 2
Queue: [S1@0; S2@1]
$S2_{max}$: 1
$S3_{max}$: 0
S3
Timestamp: 2
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1

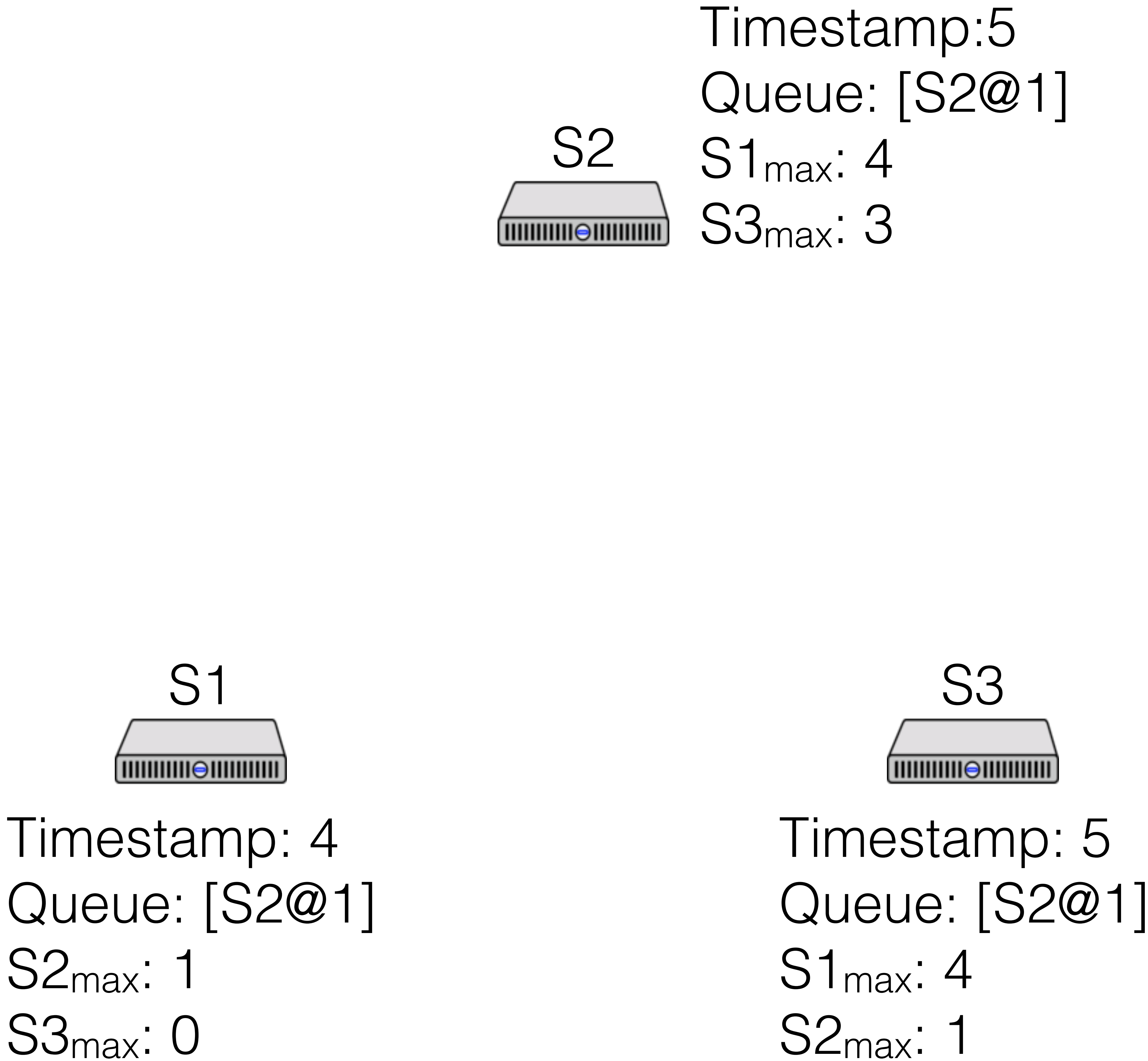

S2
Timestamp:1
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S3_{max}$: 0
ack@3
ack@3
S1
Timestamp: 3
Queue: [S1@0; S2@1]
$S2_{max}$: 1
$S3_{max}$: 0
S3
Timestamp: 3
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1

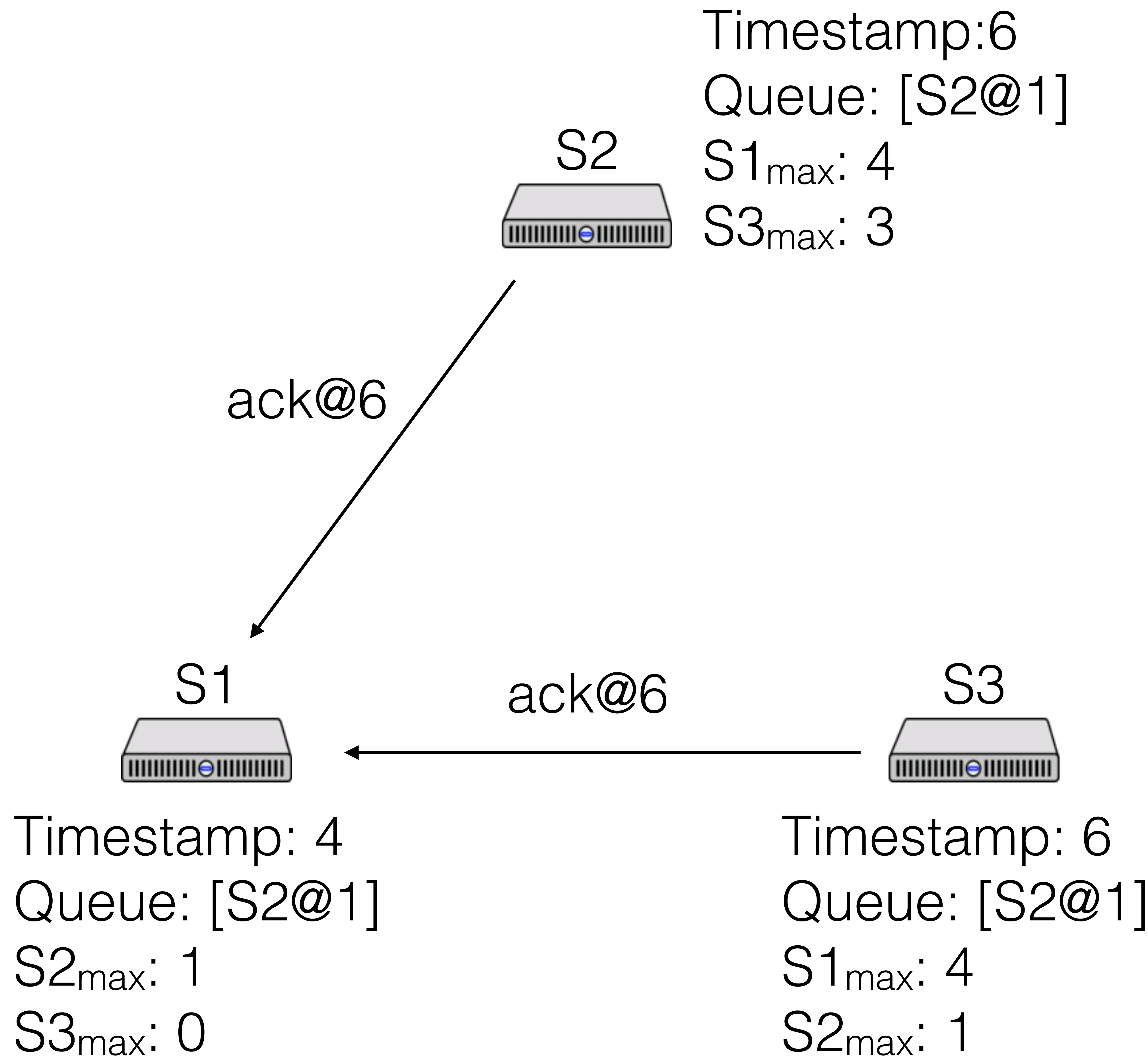


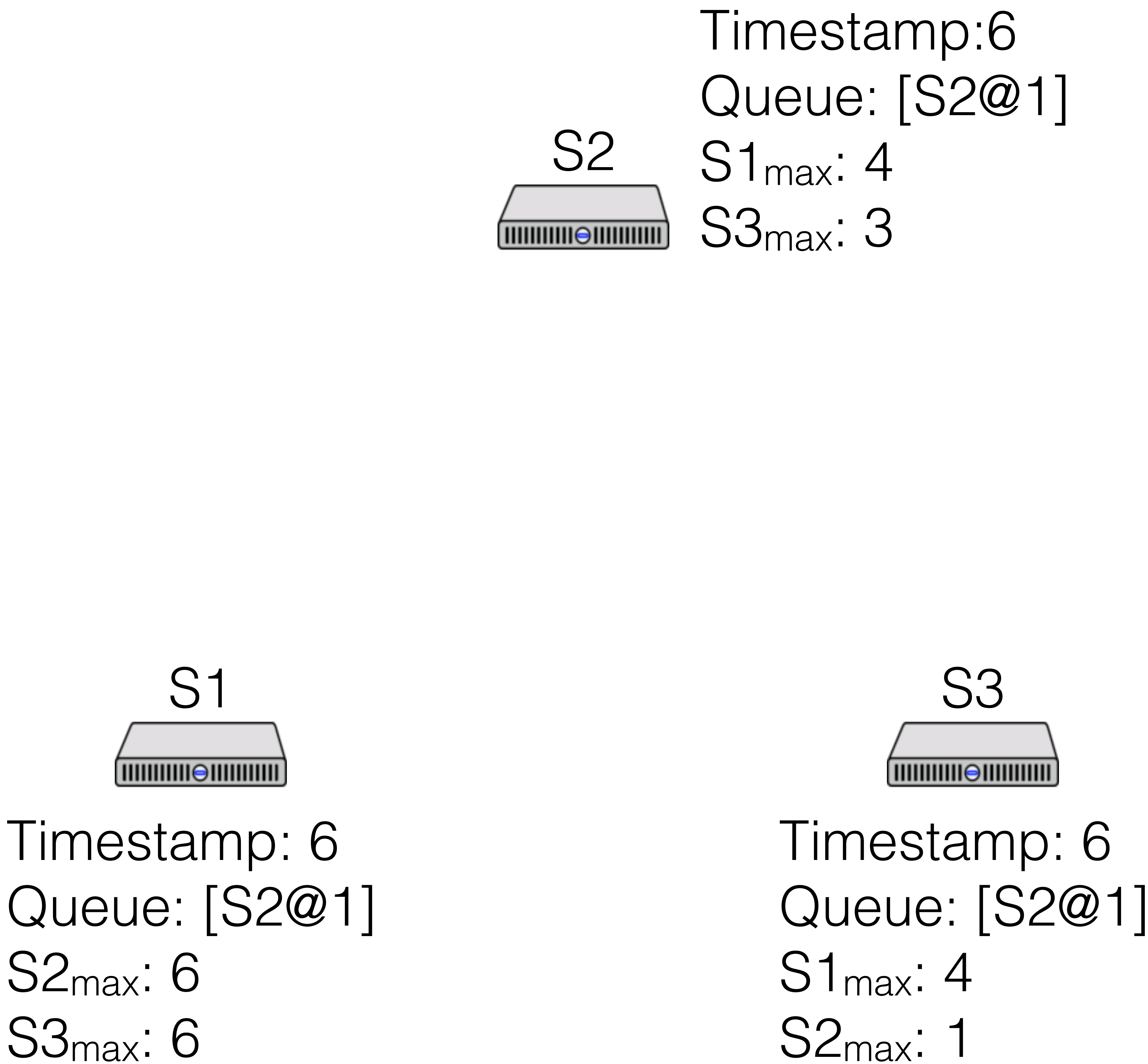
S2
Timestamp:4
Queue: [S1@0; S2@1]
$S1_{max}$: 3
$S3_{max}$: 3
S1
Timestamp: 3
Queue: [S1@0; S2@1]
$S2_{max}$: 1
$S3_{max}$: 0
S3
Timestamp: 3
Queue: [S1@0; S2@1]
$S1_{max}$: 0
$S2_{max}$: 1



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


Timestamp:6
Queue: [S2@1]
S1max: 4
S3max: 3
S2
ack@6
S1
ack@6
S3
Timestamp: 4
Queue: [S2@1]
S2max: 1
S3max: 0
Timestamp: 6
Queue: [S2@1]
S1max: 4
S2max: 1


S2
Timestamp:6
Queue: [S2@1]
S1max: 4
S3max: 3
S1
Timestamp: 6
Queue: [S2@1]
S2max: 6
S3max: 6
S3
Timestamp: 6
Queue: [S2@1]
S1max: 4
S2max: 1

Questions

- What happens if you don’t have in-order delivery?
- What happens if you eliminate the ack for the request?
- What happens when nodes fail?

Generic State Machine Replication (SMR)

In mutual exclusion:

- State: queue of processes who want the lock
- Commands: P_i *requests*, P_i *releases*

Approach generalizes to other “state machines"

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

Lamport paper discussion

What happens when we need to add a process?

How can we separate out concurrent events that just happened to have a certain ordering for their times?