

Paxos and Raft

(Lecture 21, cs262a)

Ion Stoica,
UC Berkeley
November 7, 2016

Bezos' mandate for service-oriented-architecture (~2002)

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this will be fired.
7. Thank you; have a nice day!

Today's Papers

Paxos Made Simple,

Leslie Lamport

(research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf)

In Search of an Understandable Consensus Algorithm,

Diego Ongaro and John Ousterhout, USENIX ATC'14

(<https://ramcloud.stanford.edu/raft.pdf>)

Paxos

Based on many slides from Indranil Gupta's presentation:
(https://courses.engr.illinois.edu/cs525/sp2013/L9_paxos.sp13.ppt),
and Gene Pang's presentation
(www.cs.berkeley.edu/~istoica/classes/cs294/11/notes/07-gene-paxos.pptx)

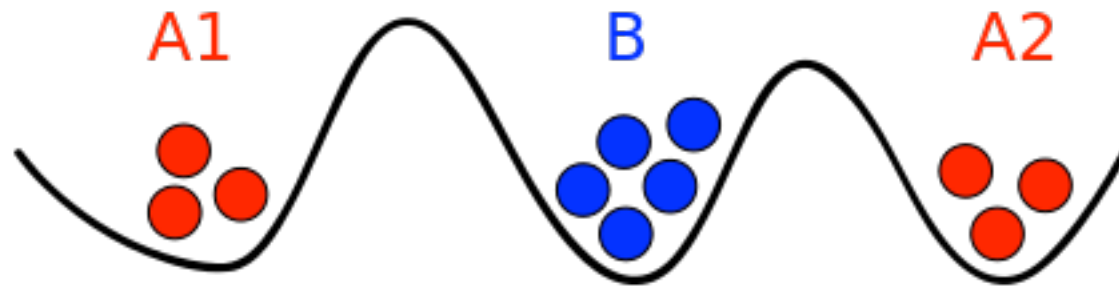
Distributed consensus problem

Group of processes must agree on a single value

Value must be proposed

After value is agreed upon, it can be learned

Consensus Impossibility Result



Byzantine Generals' Problem

Why cannot reach consensus?

Two types of failures

Non-Byzantine

Failed nodes stop communicating with other nodes

- "Clean" failure
- *Fail-stop* behavior

Byzantine

Failed nodes will keep sending messages

- Incorrect and potentially misleading
- Failed node becomes a *traitor*

Assumption: asynchronous, non-byzantine model

Paxos

L. Lamport, The Part-Time Parliament, September 1989

Aegean island of Paxos

A part-time parliament

- Goal: determine the sequence of decrees passed (consensus!)

Political Science Analogy

Paxos has rounds: each round has a unique ballot ID

Rounds are asynchronous

- Time synchronization not required
- If you are in round j and hear a message from round $j+1$, abort everything and go to round $j+1$

Each round

- Phase 1: A leader is elected (**election**)
- Phase 2: Leader proposes a value (**bill**), processes acks
- Phase 3: Leaders multicast final value (**law**)

Does Paxos Solve Consensus?

Provides safety and liveness

Safety:

- Only a value which has been proposed can be chosen
- Only a single value can be chosen
- A process never learns a value unless it was actually chosen

Eventual **liveness**: If things go “well” at some point in the future (e.g., losses, failures), consensus is eventually reached. However, this is not guaranteed.

So Simple, So Obvious

“In fact, it is among the simplest and most obvious of distributed algorithms.”

- Leslie Lamport

Simple Pseudocode

outcome[*p*] The decree written in *p*'s ledger, or BLANK if there is nothing written there yet.
lastTried[*p*] The number of the last ballot that *p* tried to begin, or $-\infty$ if there was none.
prevBal[*p*] The number of the last ballot in which *p* voted, or $-\infty$ if he never voted.
prevDec[*p*] The decree for which *p* last voted, or BLANK if *p* never voted.
nextBal[*p*] The number of the last ballot in which *p* agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

Next come variables representing information that priest *p* could keep on a slip of paper:

status[*p*] One of the following values:
idle Not conducting or trying to begin a ballot
trying Trying to begin ballot number *lastTried*[*p*]
polling Now conducting ballot number *lastTried*[*p*]
If *p* has lost his slip of paper, then *status*[*p*] is assumed to equal *idle* and the values of the following four variables are irrelevant.
prevVotes[*p*] The set of votes received in *LastVote* messages for the current ballot (the one with ballot number *lastTried*[*p*]).
quorum[*p*] If *status*[*p*] = *polling*, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.
voters[*p*] If *status*[*p*] = *polling*, then the set of quorum members from whom *p* has received *Voted* messages in the current ballot; otherwise, meaningless.
decree[*p*] If *status*[*p*] = *polling*, then the decree of the current ballot; otherwise, meaningless.

Try New Ballot

Always enabled.

- Set *lastTried*[*p*] to any ballot number *b*, greater than its previous value, such that *owner*(*b*) = *p*.
- Set *status*[*p*] to *trying*.
- Set *prevVotes*[*p*] to \emptyset .

Send NextBallot Message

Enabled whenever *status*[*p*] = *trying*.

- Send a *NextBallot*(*lastTried*[*p*]) message to any priest.

Receive NextBallot(*b*) Message

If $b \geq \text{nextBal}[p]$ then

- Set *nextBal*[*p*] to *b*.

Send LastVote Message

Enabled whenever *nextBal*[*p*] > *prevBal*[*p*].

- Send a *LastVote*(*nextBal*[*p*], *v*) message to priest *owner*(*nextBal*[*p*]), where $v_{\text{pst}} = p$, $v_{\text{bal}} = \text{prevBal}[p]$, and $v_{\text{dec}} = \text{prevDec}[p]$.

Receive LastVote(*b*, *v*) Message

If $b = \text{lastTried}[p]$ and *status*[*p*] = *trying*, then

- Set *prevVotes*[*p*] to the union of its original value and {*v*}.

Start Polling Majority Set *Q*

Enabled when *status*[*p*] = *trying* and $Q \subseteq \{v_{\text{pst}} : v \in \text{prevVotes}[p]\}$, where *Q* is a majority set.

- Set *status*[*p*] to *polling*.
- Set *quorum*[*p*] to *Q*.
- Set *voters*[*p*] to \emptyset .
- Set *decree*[*p*] to a decree *d* chosen as follows: Let *v* be the maximum element of *prevVotes*[*p*]. If $v_{\text{bal}} \neq -\infty$ then $d = v_{\text{dec}}$; else *d* can equal any decree.
- Set \mathcal{B} to the union of its former value and {*B*}, where $B_{\text{dec}} = d$, $B_{\text{qrm}} = Q$, $B_{\text{vot}} = \emptyset$, and $B_{\text{bal}} = \text{lastTried}[p]$.

Send BeginBallot Message

Enabled when *status*[*p*] = *polling*.

- Send a *BeginBallot*(*lastTried*[*p*], *decree*[*p*]) message to any priest in *quorum*[*p*].

Receive BeginBallot(*b*, *d*) Message

If $b = \text{nextBal}[p] > \text{prevBal}[p]$ then

- Set *prevBal*[*p*] to *b*.
- Set *prevDec*[*p*] to *d*.
- If there is a ballot *B* in \mathcal{B} with $B_{\text{bal}} = b$ [there will be], then choose any such *B* [there will be only one] and let the new value of \mathcal{B} be obtained from its old value by setting B_{vot} equal to the union of its old value and {*p*}.

Send Voted Message

Enabled whenever *prevBal*[*p*] $\neq -\infty$.

- Send a *Voted*(*prevBal*[*p*], *p*) message to *owner*(*prevBal*[*p*]).

Receive Voted(*b*, *q*) Message

If $b = \text{lastTried}[p]$ and *status*[*p*] = *polling*, then

- Set *voters*[*p*] to the union of its old value and {*q*}

Succeed

Enabled whenever *status*[*p*] = *polling*, *quorum*[*p*] \subseteq *voters*[*p*], and *outcome*[*p*] = BLANK.

- Set *outcome*[*p*] to *decree*[*p*].

Send Success Message

Enabled whenever *outcome*[*p*] \neq BLANK.

- Send a *Success*(*outcome*[*p*]) message to any priest.

Receive Success(*d*) Message

If *outcome*[*p*] = BLANK, then

- Set *outcome*[*p*] to *d*.

3 Types of Agents

Proposers

Acceptors

Learners

Simple Implementation

Typically, every process is acceptor, proposer, and learner

A leader is elected to be the distinguished proposer and learner

- Distinguishes proposes to guarantee progress
 - Avoid dueling proposers
- Distinguishes learner to reduce too many broadcast messages

Political Analogy

Paxos has rounds: each round has a unique ballot ID

Rounds are asynchronous

- Time synchronization not required
- If you are in round j and hear a message from round $j+1$, abort everything and move to round $j+1$

Each round consists of three phases

- Phase 1: A leader is elected (**Election**)
- Phase 2: Leader proposes a value, processes acks (**Bill**)
- Phase 3: Leader multicasts final value (**Law**)

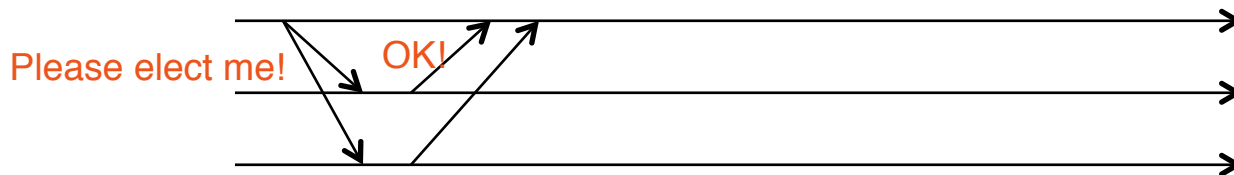
Phase 1 – Election

Potential leader chooses a unique ballot ID, higher than anything it has seen so far

Sends ballot ID to all processes

Processes respond to highest ballot id

- If potential leader sees a higher ballot id, it can't be a leader
- Paxos tolerant to multiple leaders, but we'll mainly discuss only one leader case
- Processes also **log** received ballot ID on disk



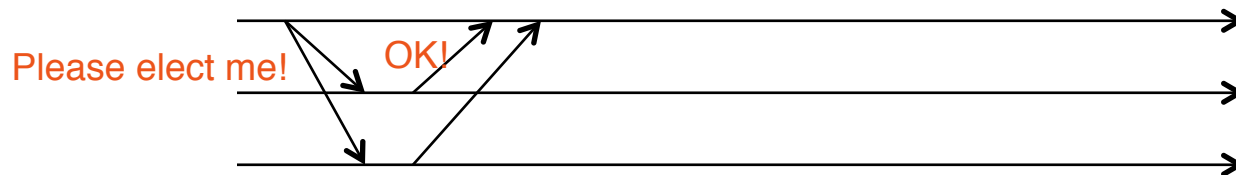
Phase 1 – Election

If a process has in a previous round decided on a value v' , it includes value v' in its response

If **majority (i.e., quorum)** respond OK then you are the leader

- If no one has majority, start new round

A round cannot have two leaders (why?)

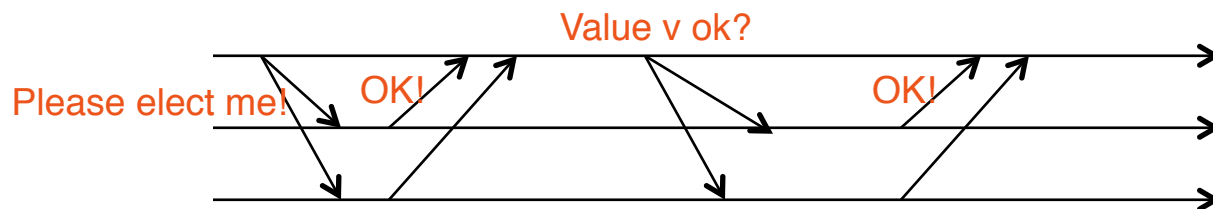


Phase 2 – Proposal (Bill)

Leader sends proposal value v to all

- If some process already decided value v' in a previous round send $v = v'$

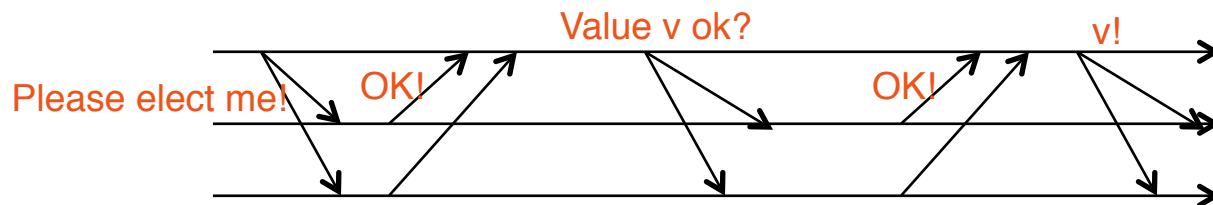
Recipient log on disk, and responds OK



Phase 3 – Decision (**Law**)

If leader hears OKs from majority, it lets everyone know of the decision

Recipients receive decisions, log it on disk



When is Consensus Achieved?

When a majority of processes hear proposed value and accept it:

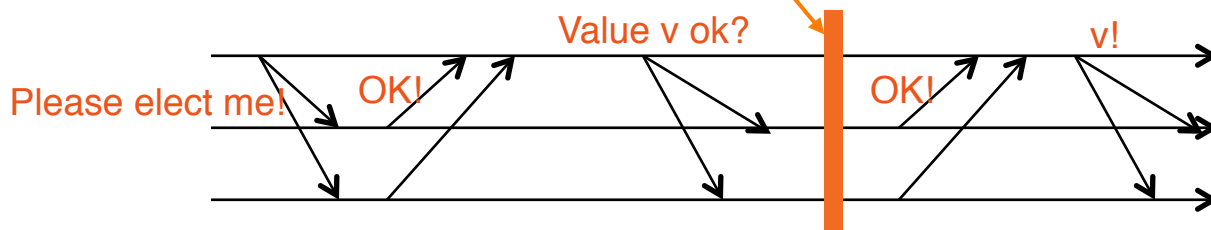
- Are about to respond (or have responded) with OK!

At this point decision has been made even though

- Processes or even leader may not know!

What if leader fails after that?

- Keep having having rounds until some round complete



Safety

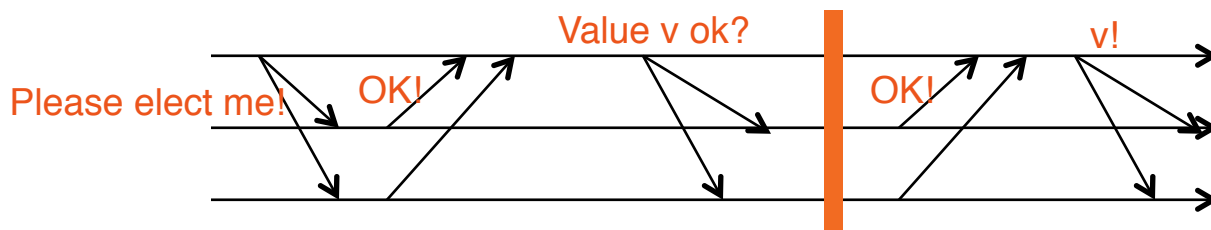
Assume a round with a majority hearing proposed value v' and accepting it (mid of Phase 2). Then subsequently at each round either:

- The round chooses v' as decision
- The round fails

“Proof”:

- Potential leader waits for majority of OKs in Phase 1
- At least one will contain v' (because two majority sets intersect)
- It will choose to send out v' in Phase 2

Success requires a majority, and two majority sets intersects



More Paxos in more detail...

Basic Paxos Protocol

Phase 1a: “Prepare”

Select proposal number* N and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

Phase 1b: “Promise”

If $N >$ number of any previous promises or acceptances,

- * promise to never accept any future proposal less than N ,
- send a ***promise(N, U)*** response

(where U is the highest-numbered proposal accepted so far (if any))

Phase 2a: “Accept!”

If proposer received promise responses from a quorum,

- send an ***accept(N, W)*** request to those acceptors

(where W is the value of the highest-numbered proposal among the ***promise*** responses, or any value if no ***promise*** contained a proposal)

Acceptor

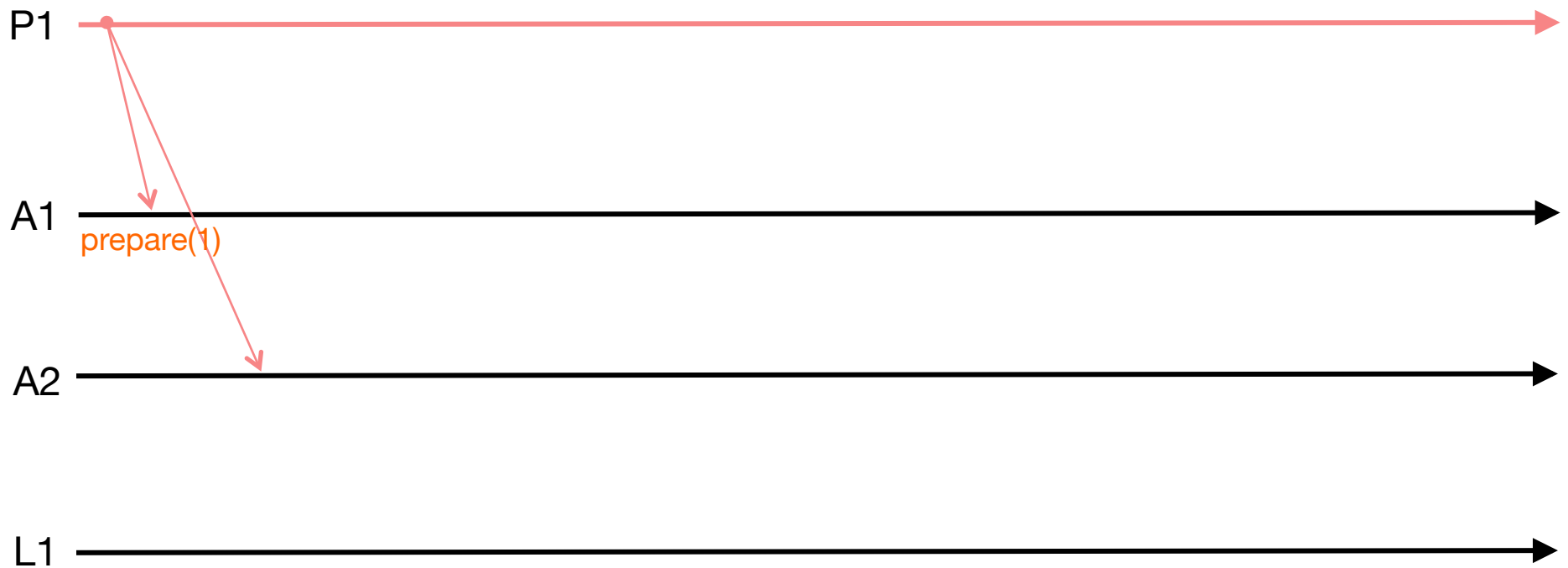
Phase 2b: “Accepted”

If $N \geq$ number of any previous promise,

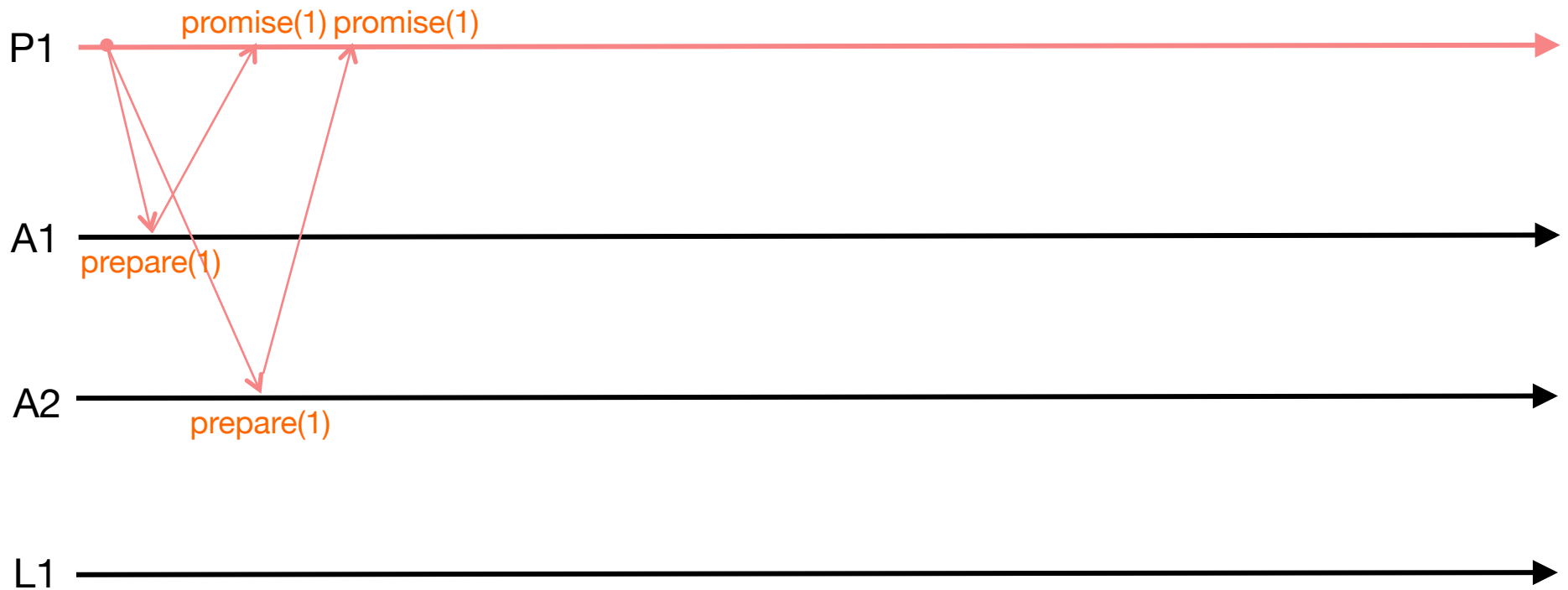
- * accept the proposal
- send an ***accepted*** notification to the learner

* = record to stable storage

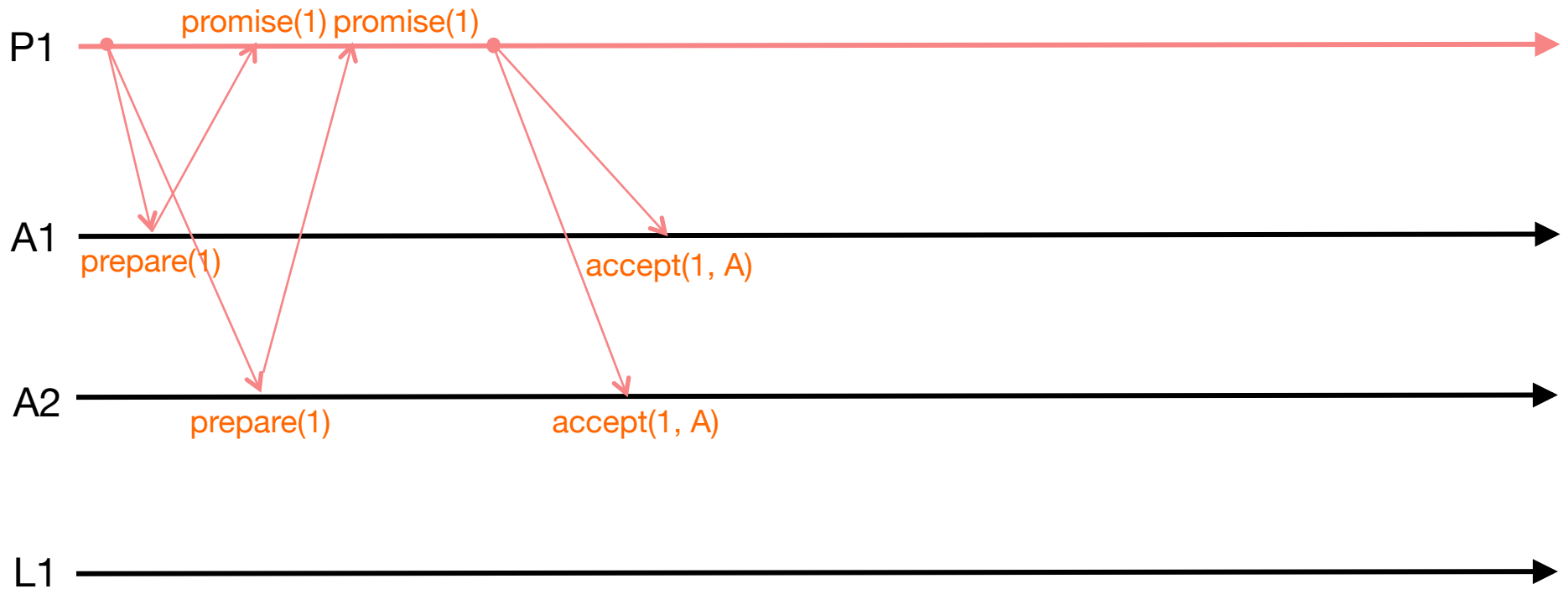
Trivial Example: P1 wants to propose "A"



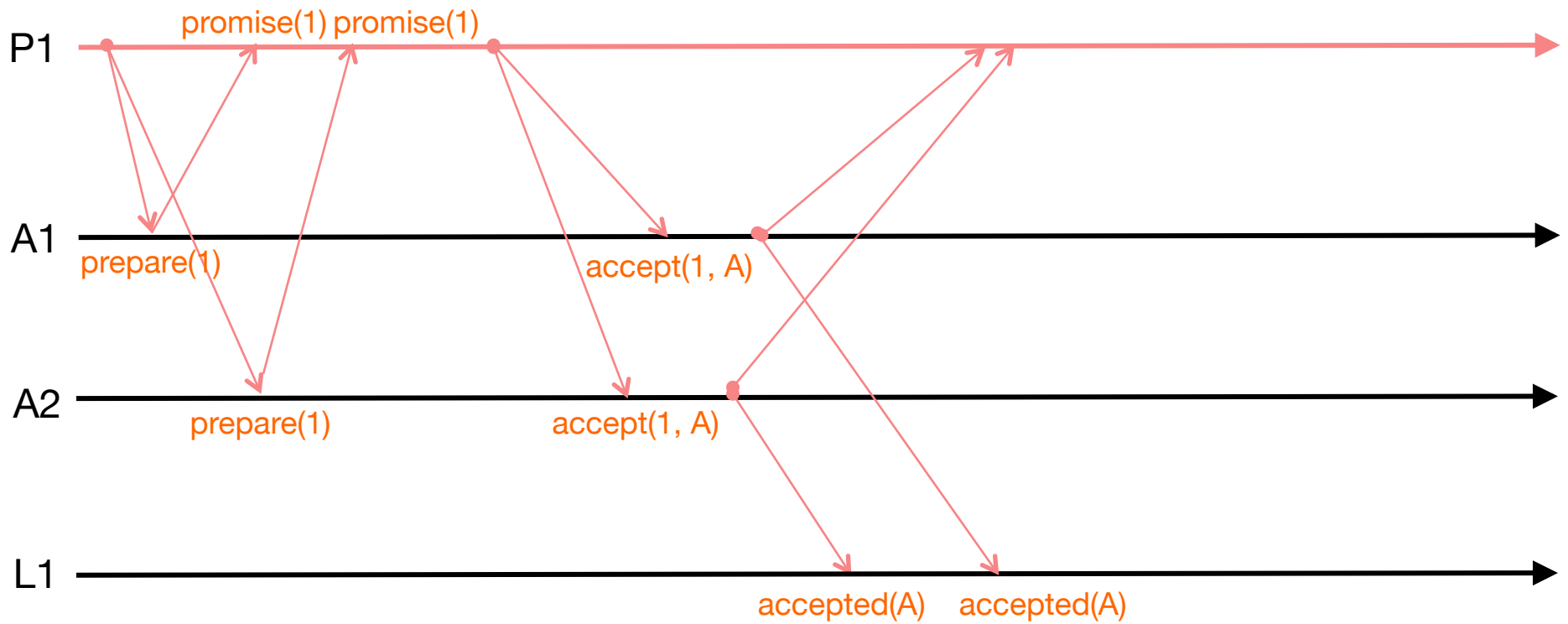
Trivial Example: P1 wants to propose “A”



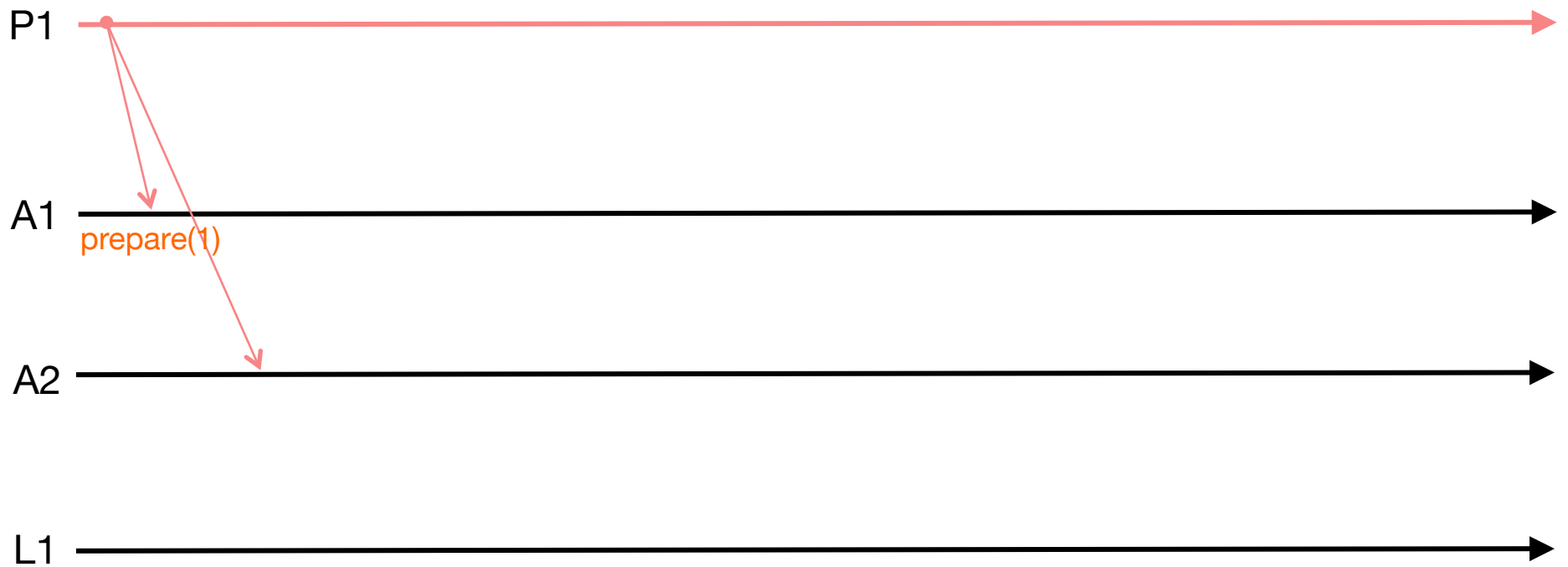
Trivial Example: P1 wants to propose "A"



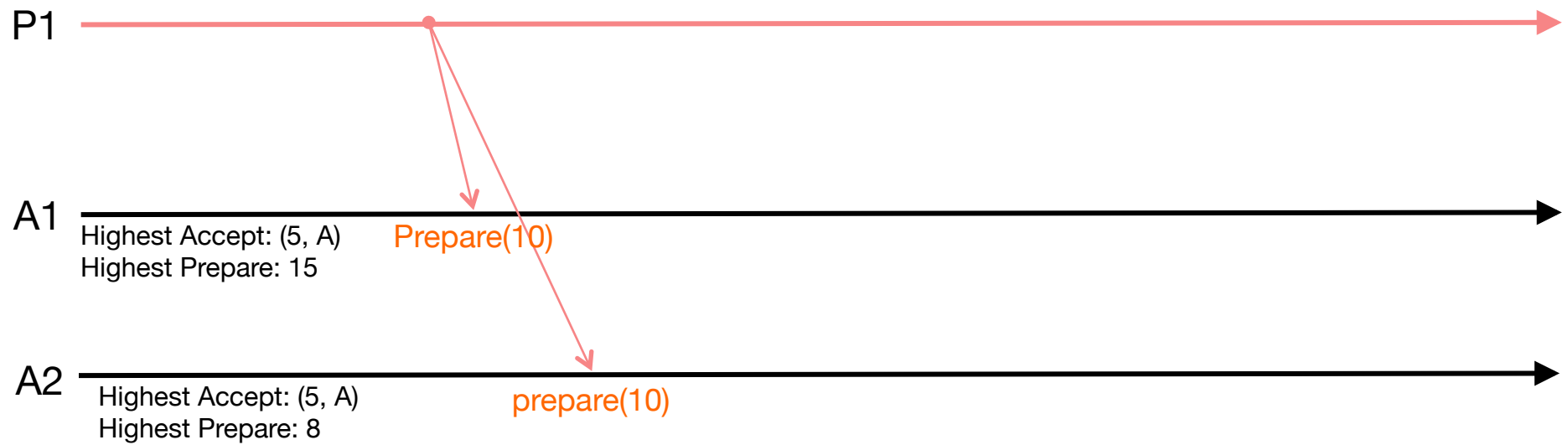
Trivial Example: P1 wants to propose "A"



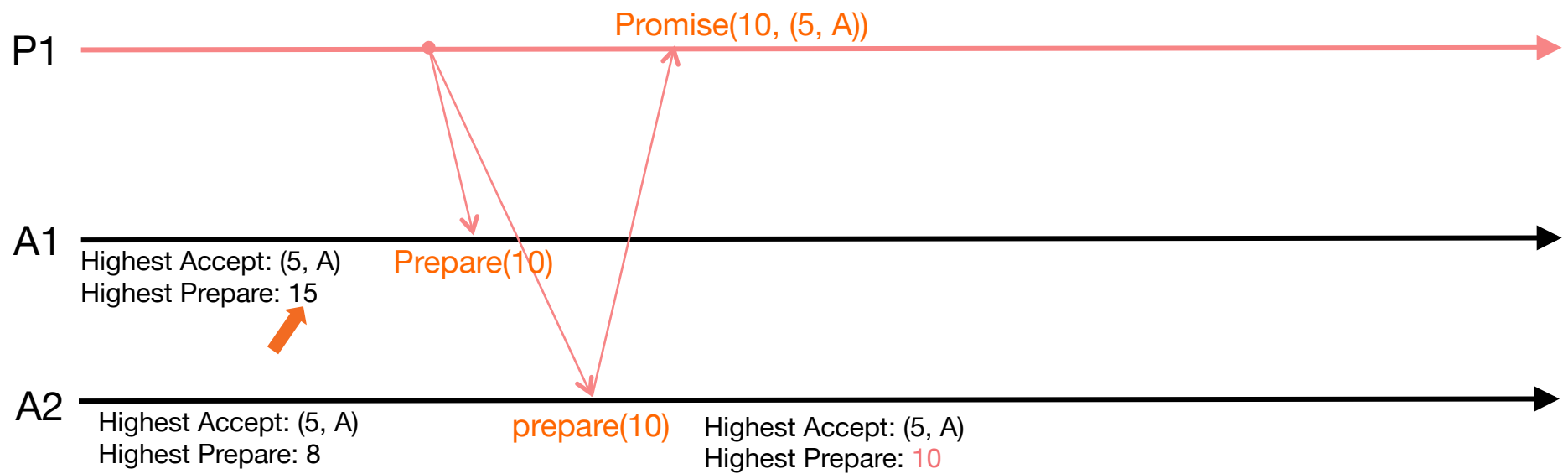
Example



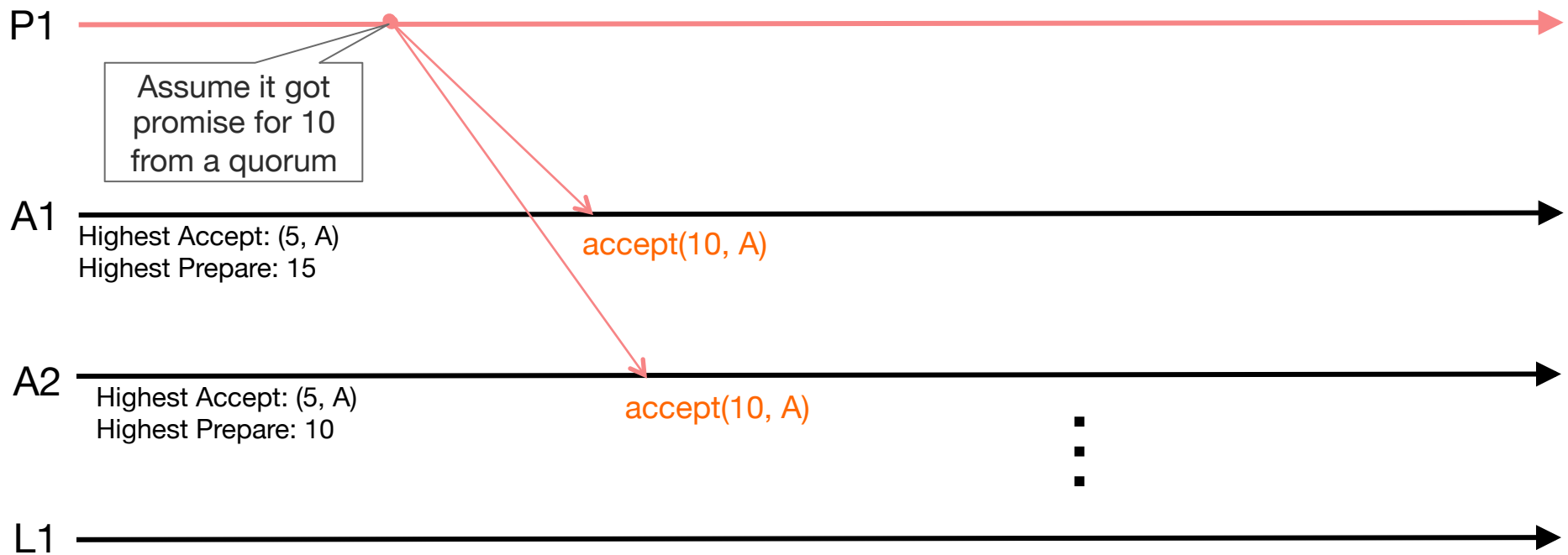
Prepare Example



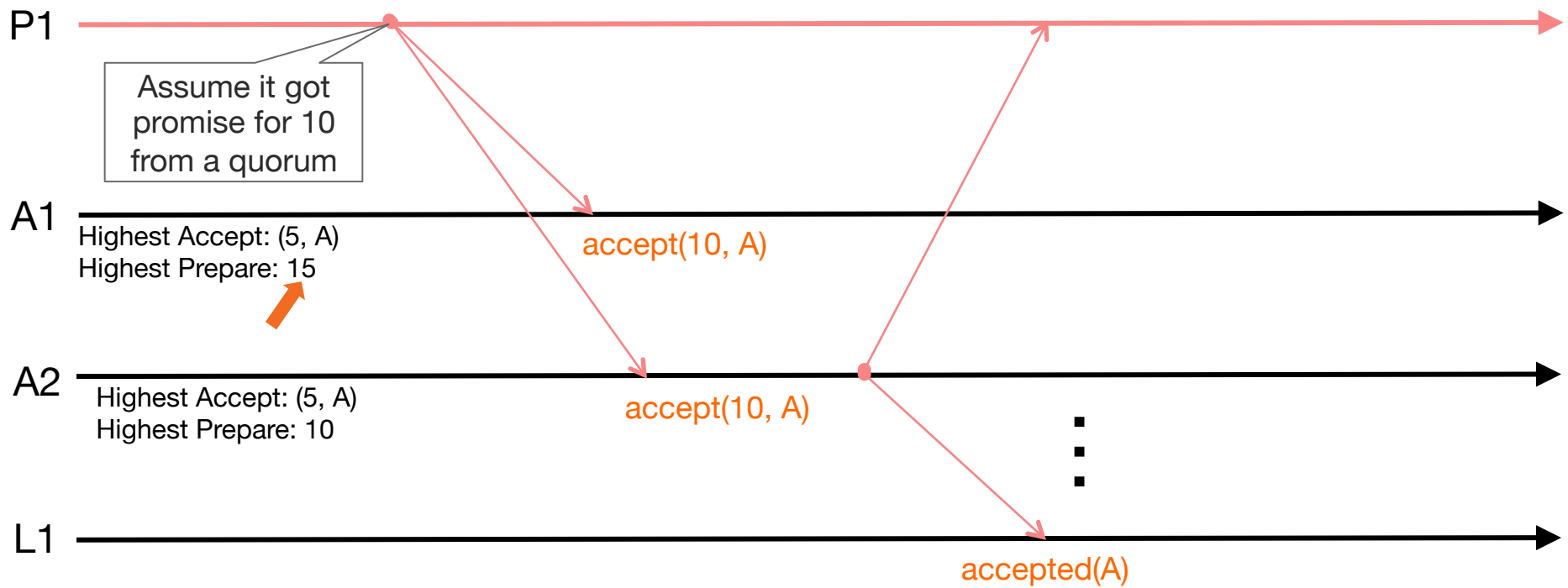
Prepare Example



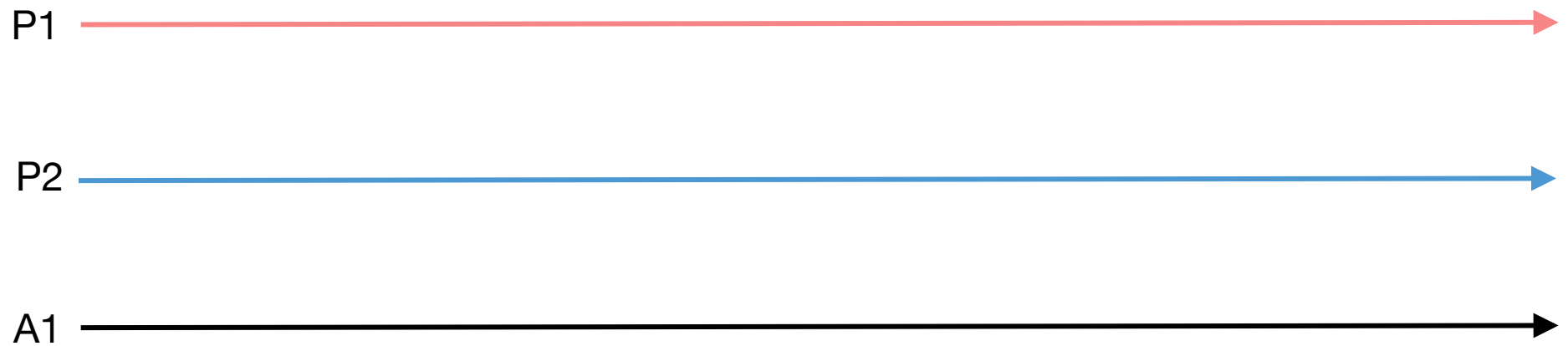
Simple Accept Example



Simple Accept Example

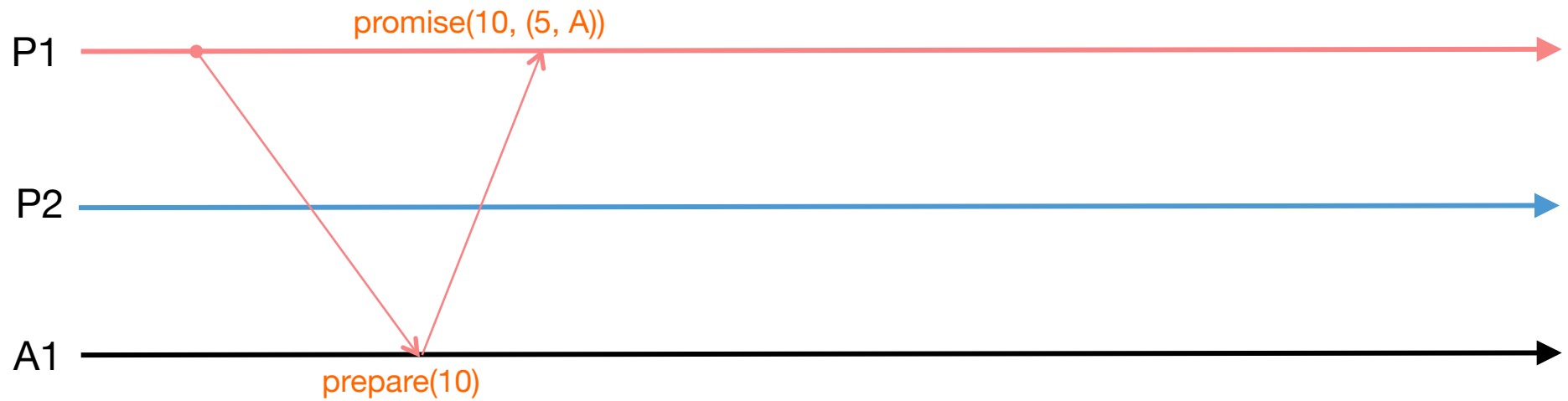


Example: Livelock



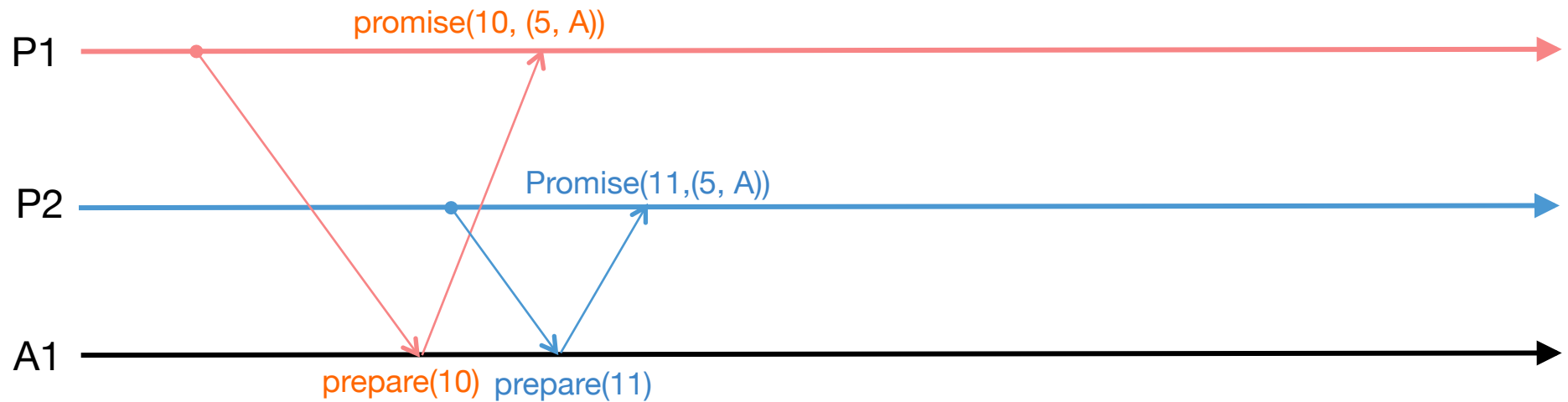
A1: Highest accept; (5, A)
Highest prepare: 8

Example: Livelock



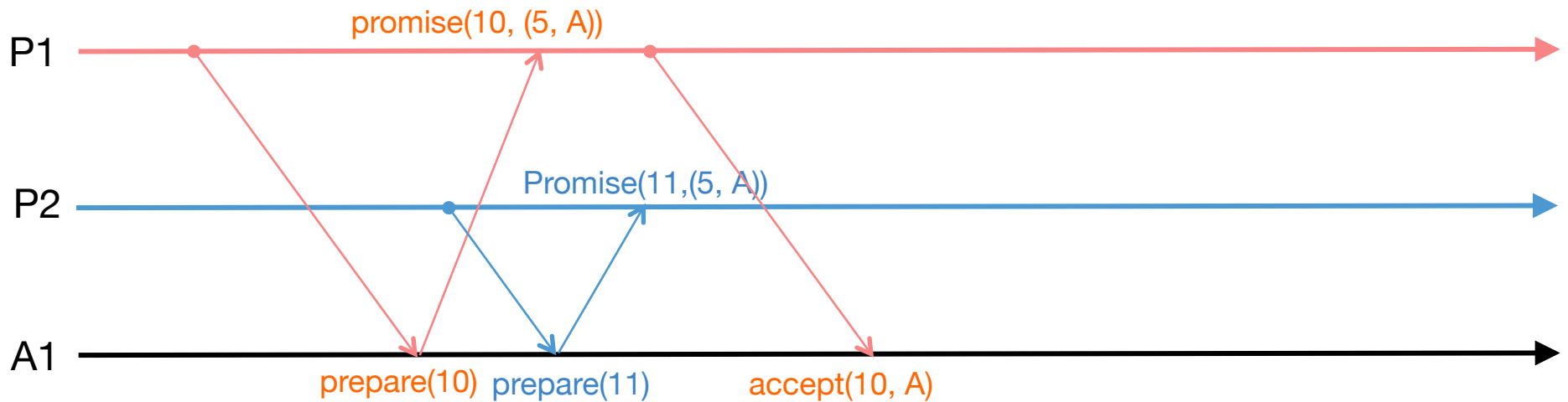
A1: Highest accept; (5, A)
Highest prepare: 10

Example: Livelock



A1: Highest accept; (5, A)
Highest prepare: 11

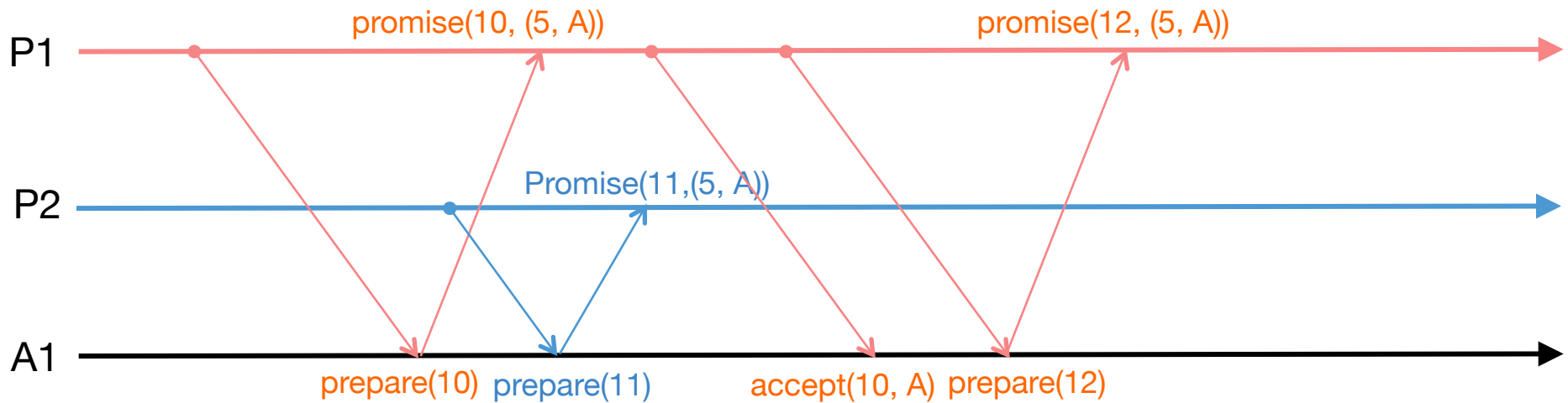
Example: Livelock



A1: Highest accept; (5, A)
Highest prepare: 11

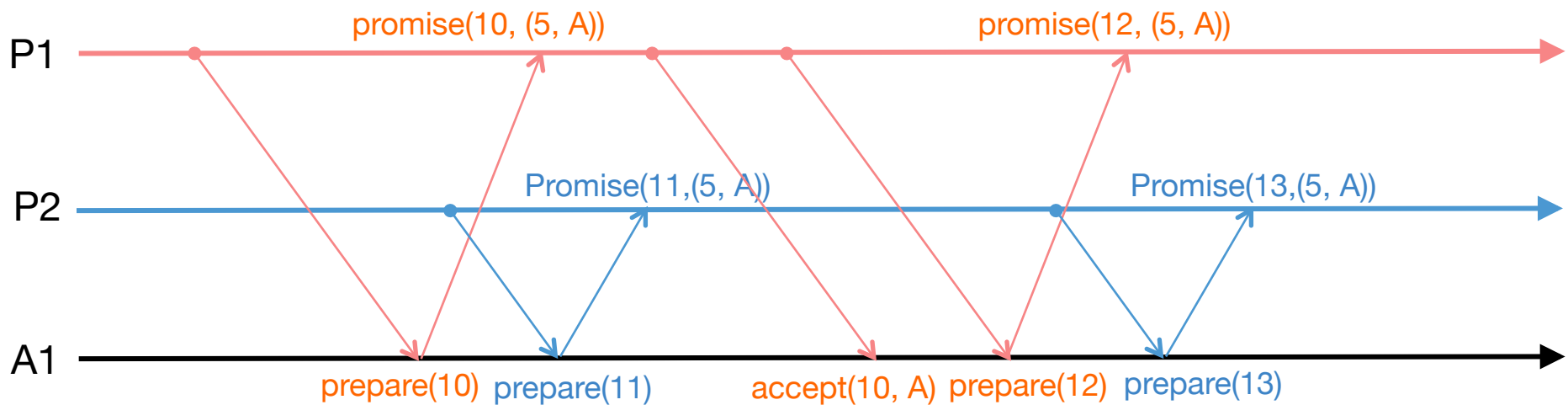


Example: Livelock



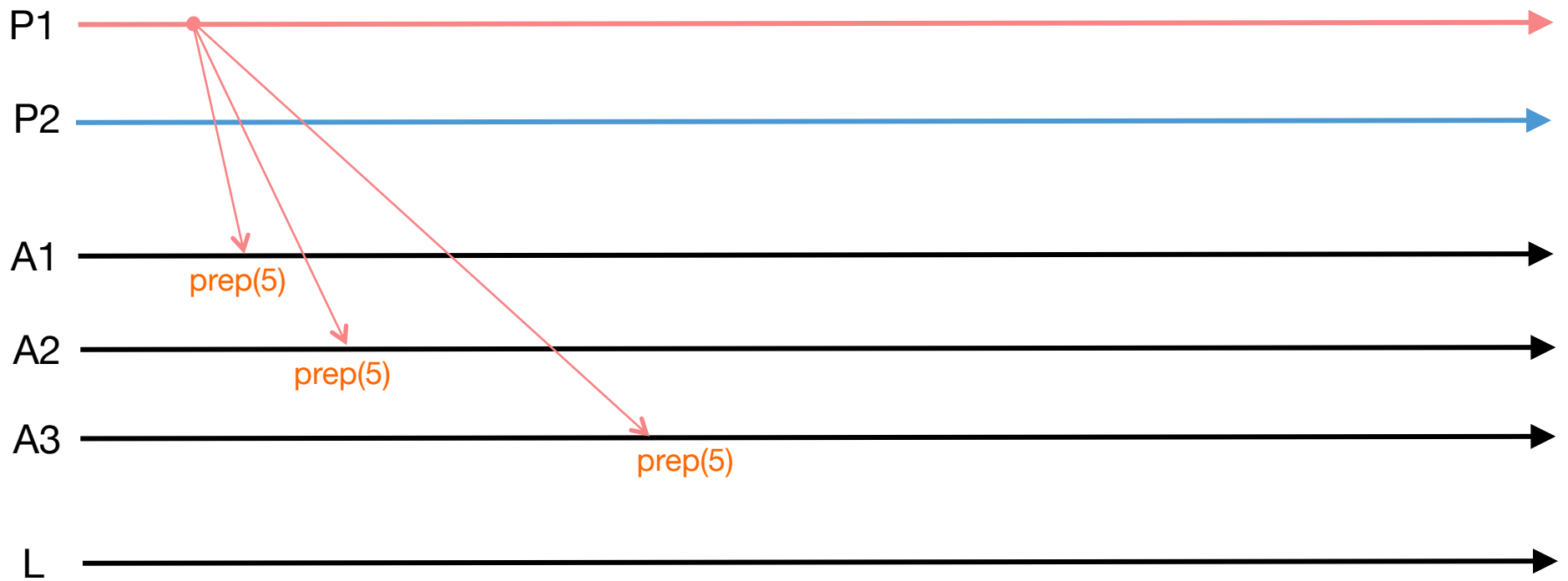
A1: Highest accept; (5, A)
Highest prepare: 12

Example: Livelock

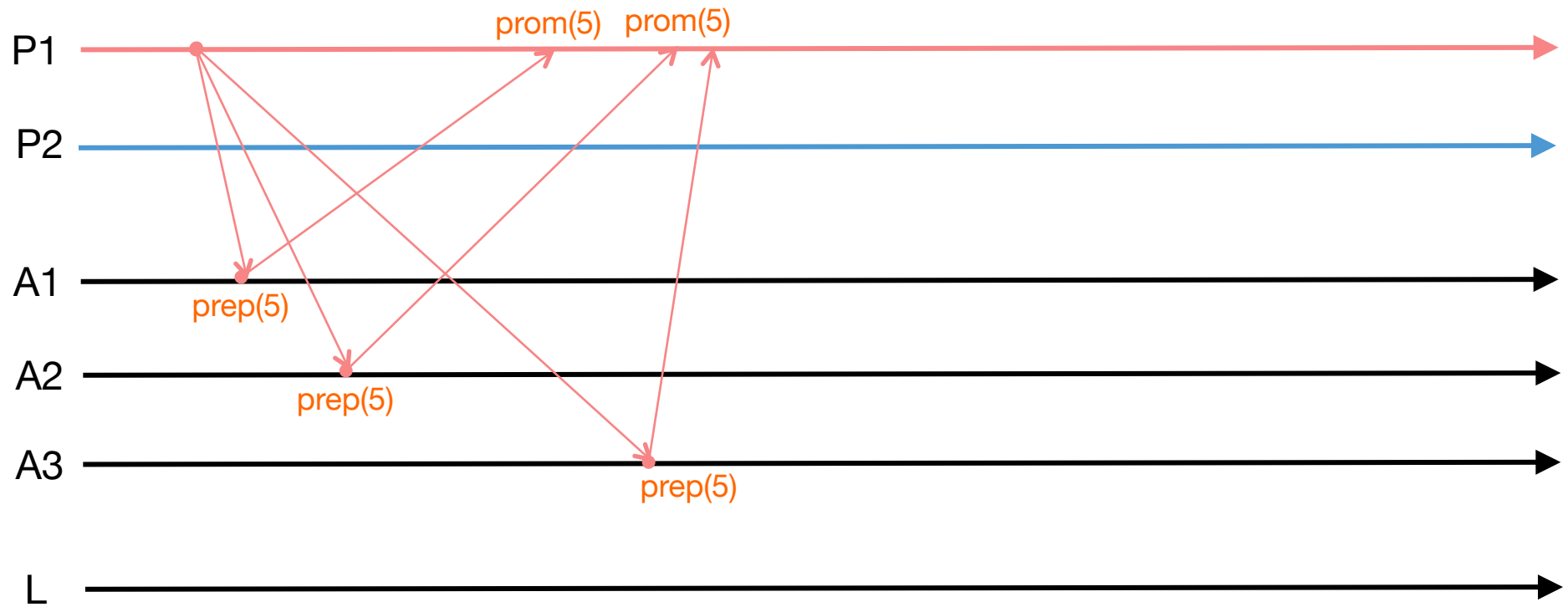


A1: Highest accept; (5, A)
Highest prepare: 13

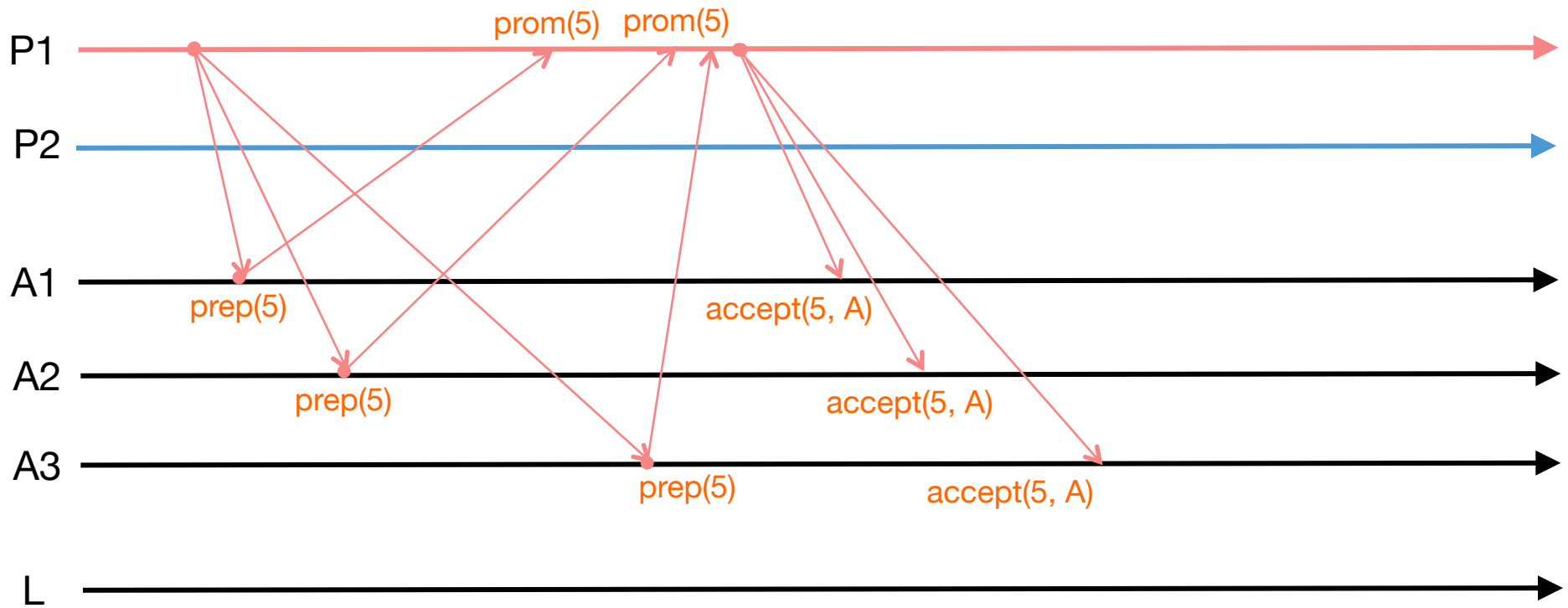
Example: P1 want to propose value A



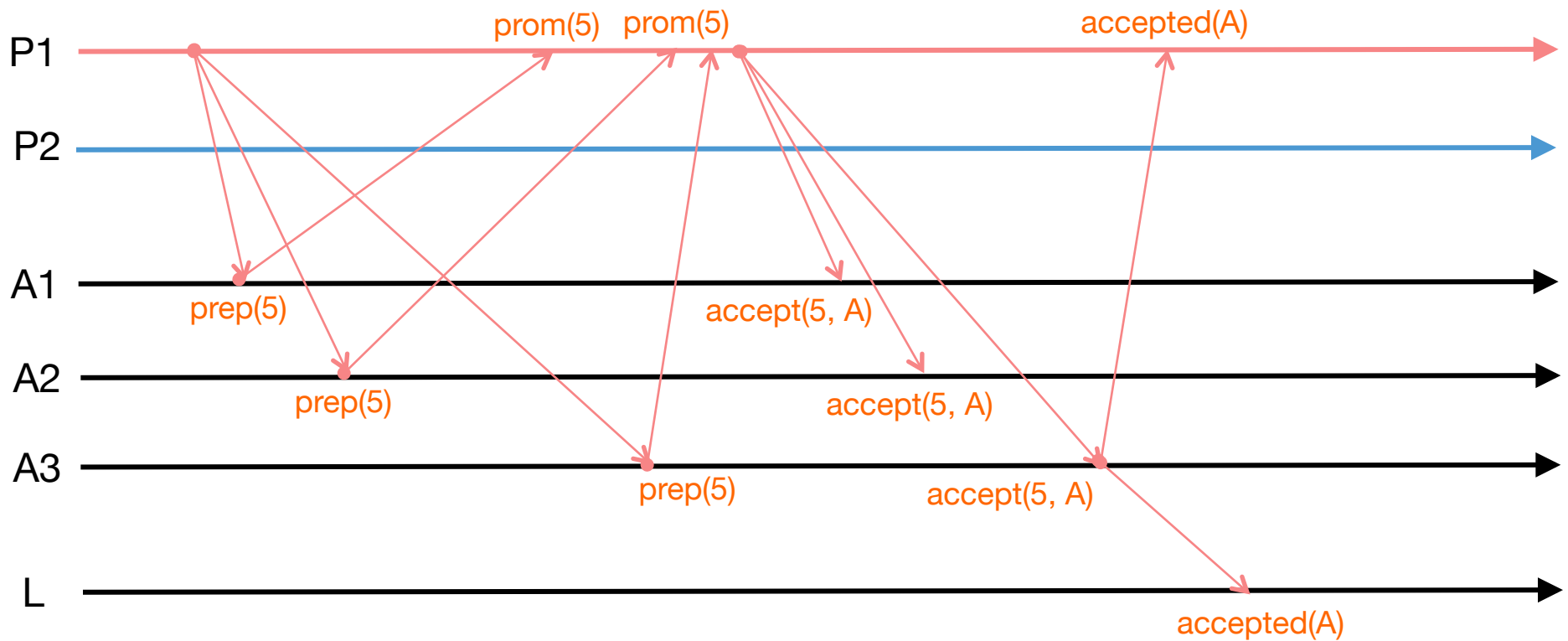
Example: P1 want to propose value A



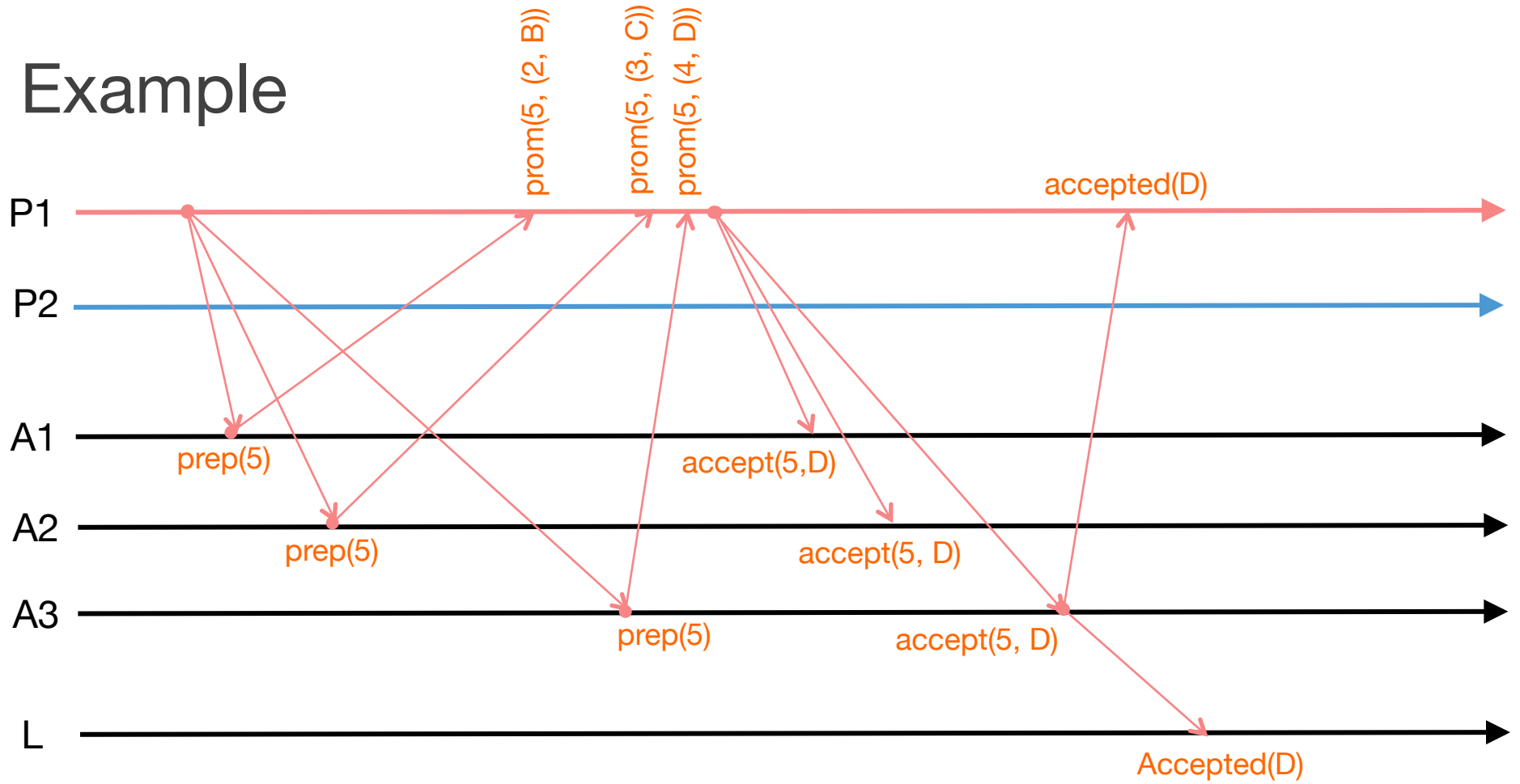
Example: P1 want to propose value A



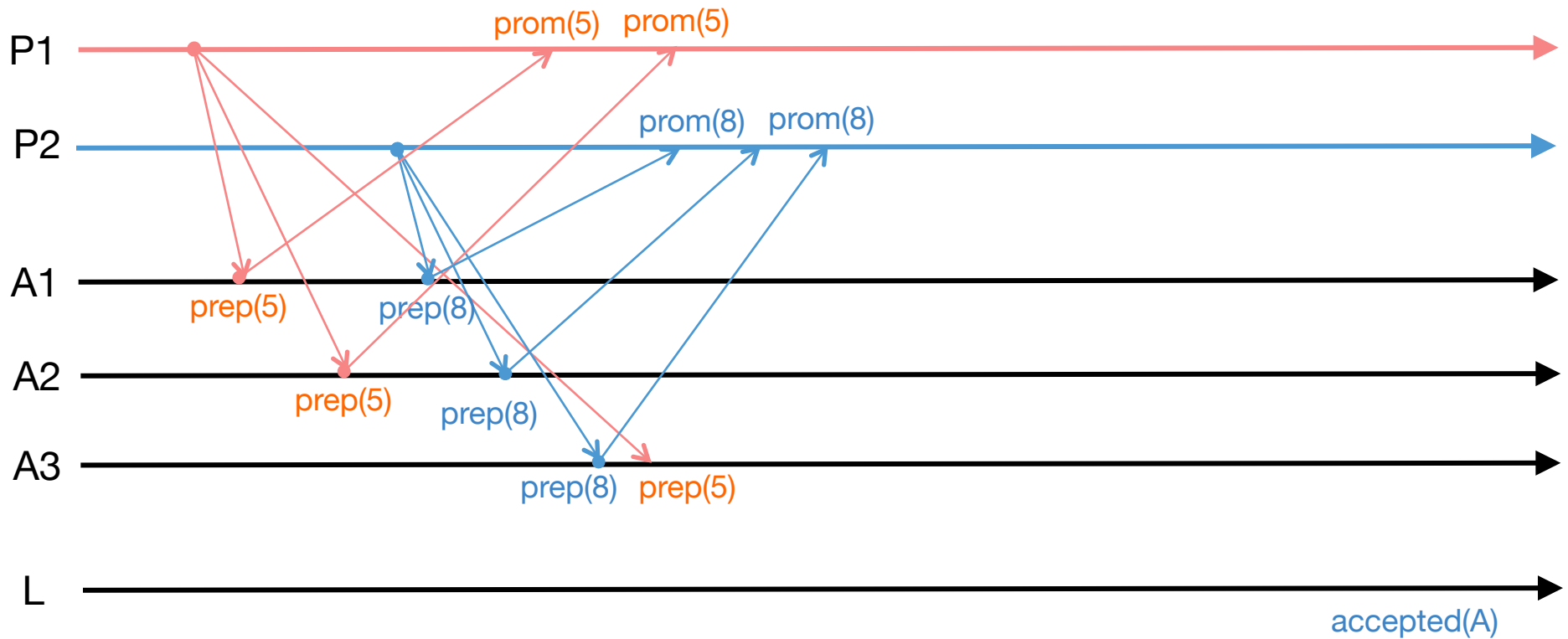
Example: P1 want to propose value A



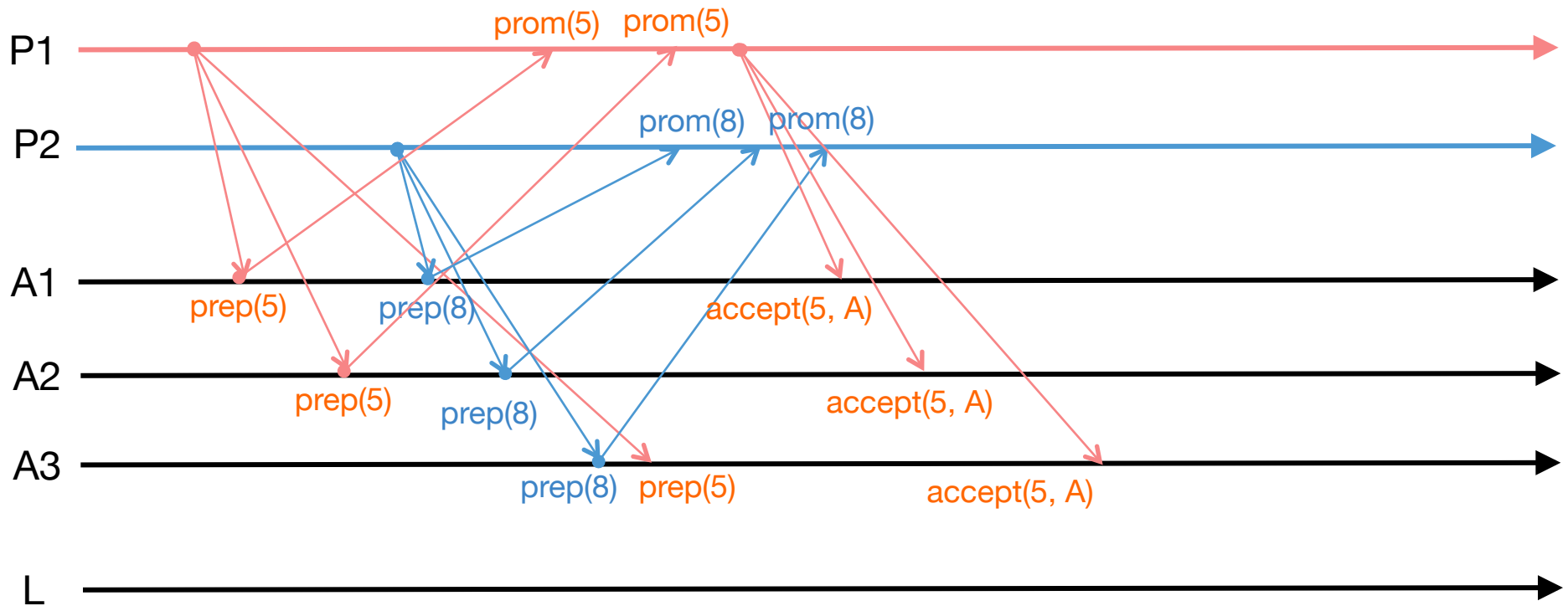
Example



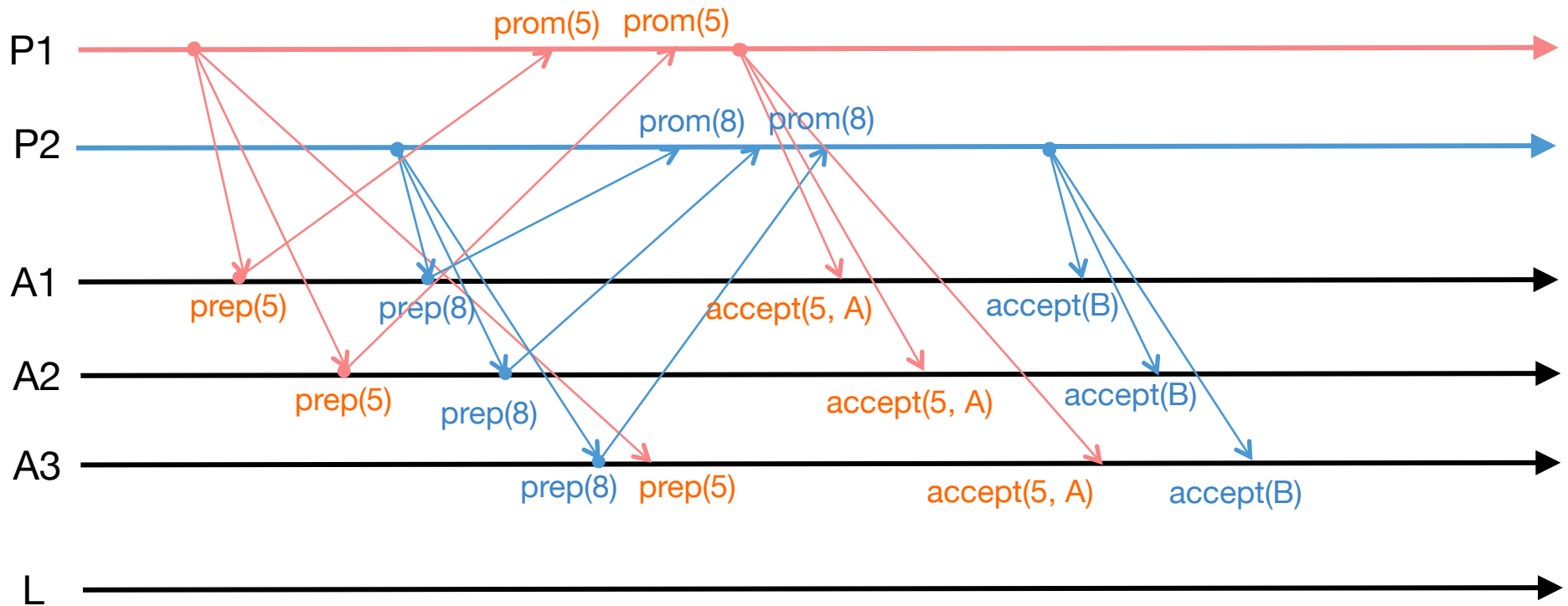
Example: P1 wants A, and P2 wants B



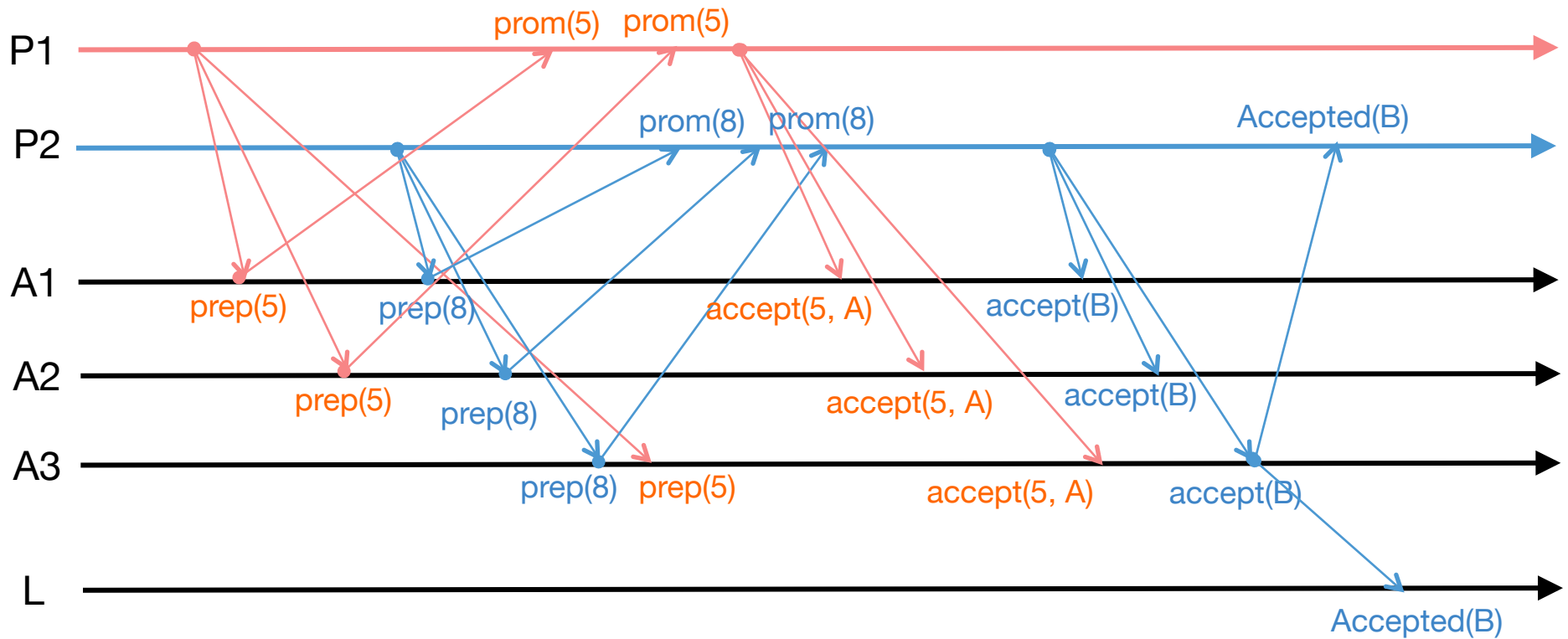
Example: P1 wants A, and P2 wants B



Example: P1 wants A, and P2 wants B



Example: P1 wants A, and P2 wants B



Others

In practice send NACKs if not accepting a promise

Promise IDs should increase slowly

- Otherwise too much too converge
- Solution: different ID spaces for proposers

Raft

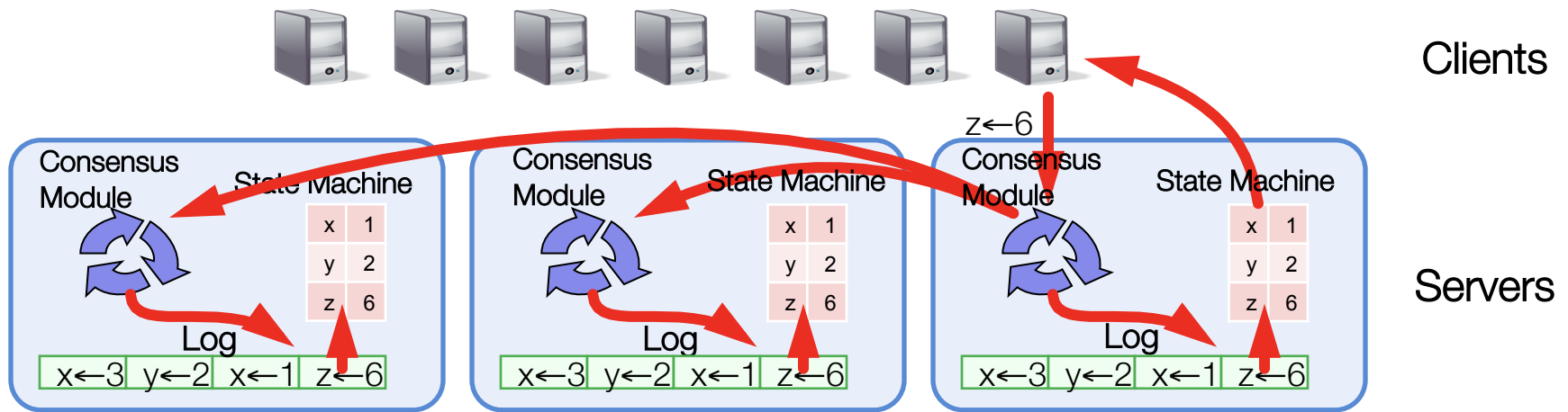
Many slides from Diego Ongaro & John Ousterhout presentation:
(<http://www2.cs.uh.edu/~paris/6360/PowerPoint/Raft.ppt>)

Paxos Limitations

“The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-).” – NSDI reviewer

“There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system...the final system will be based on an unproven protocol.” – Chubby authors

Replicated State Machines



Replicated log \Rightarrow replicated state machine

- All servers execute same commands in same order

Consensus module ensures proper log replication

System makes progress as long as any majority of servers are up

Failure model: fail-stop (not Byzantine), delayed/lost messages

Designing for understandability

Main objective of RAFT

- Whenever possible, select the alternative that is the easiest to understand

Techniques that were used include

- Dividing problems into smaller problems
- Reducing the number of system states to consider

Raft Overview

1. Leader election
 - Select one of the servers to act as cluster leader
 - Detect crashes, choose new leader
2. Log replication (normal operation)
 - Leader takes commands from clients, appends them to its log
 - Leader replicates its log to other servers (overwriting inconsistencies)
3. Safety
 - Only a server with an up-to-date log can become leader

Raft basics: the servers

A RAFT cluster consists of several servers

- Typically five

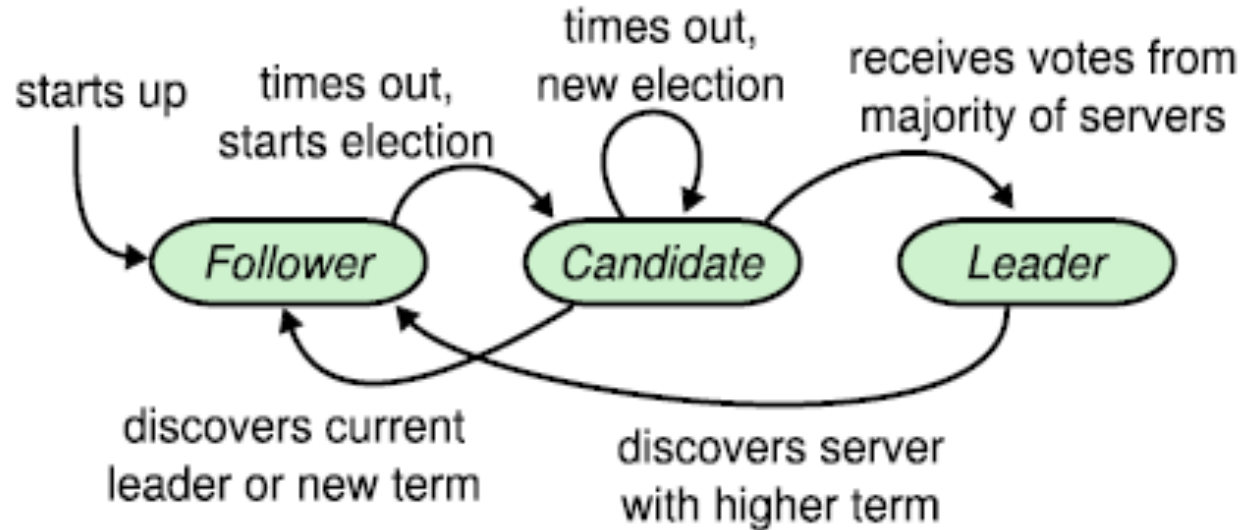
Each server can be in one of three states

- *Leader*
- *Follower*
- *Candidate* (to be the new leader)

Followers are passive:

- Simply reply to requests coming from their leader

Server states



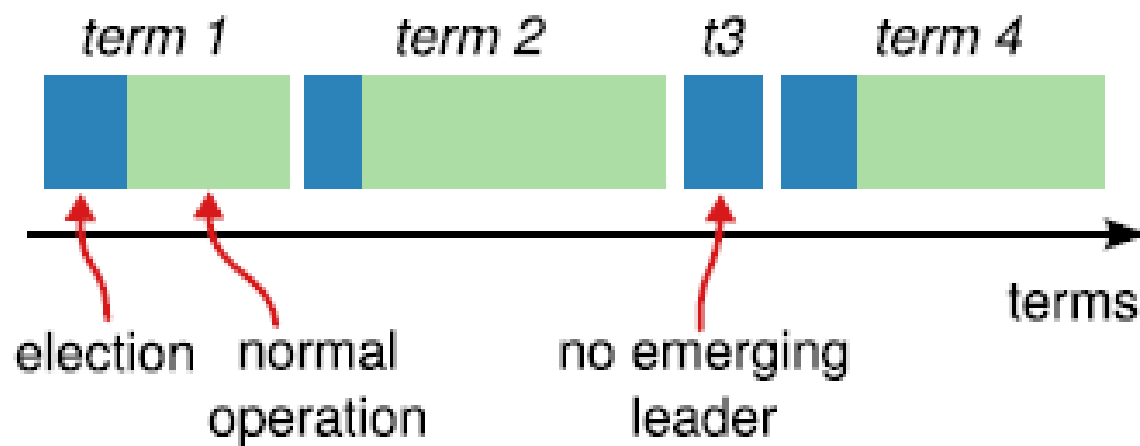
Raft basics: terms (I)

Epochs of arbitrary length

- Start with the election of a leader
- End when
 - Leader becomes unavailable
 - No leader can be selected (split vote)

Different servers may observe transitions between terms at different times or even miss them

Raft basics: terms (II)



Raft basics: RPC

Servers communicate through idempotent RPCs

RequestVote

- Initiated by candidates during elections

AppendEntry: Initiated by leaders to

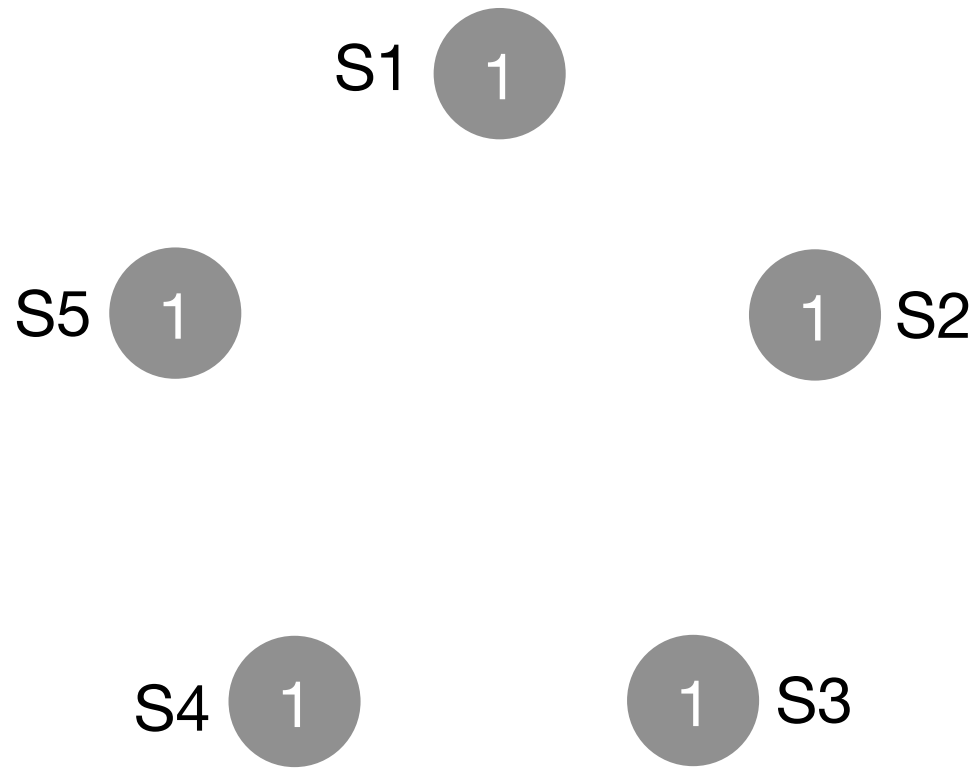
- Replicate log entries
- Provide a form of heartbeat
 - Empty AppendEntry() calls

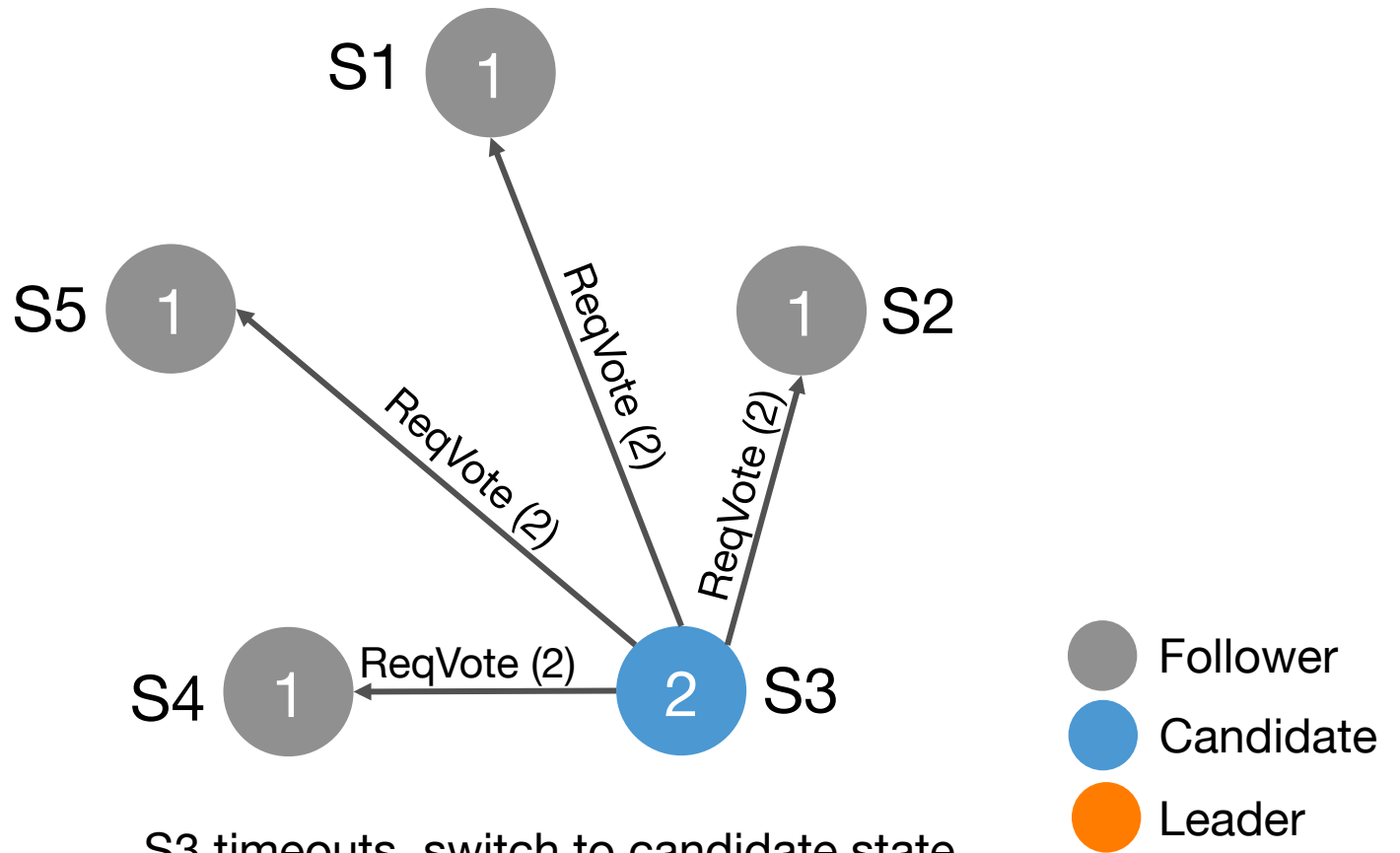
Leader elections

Servers start being *followers*

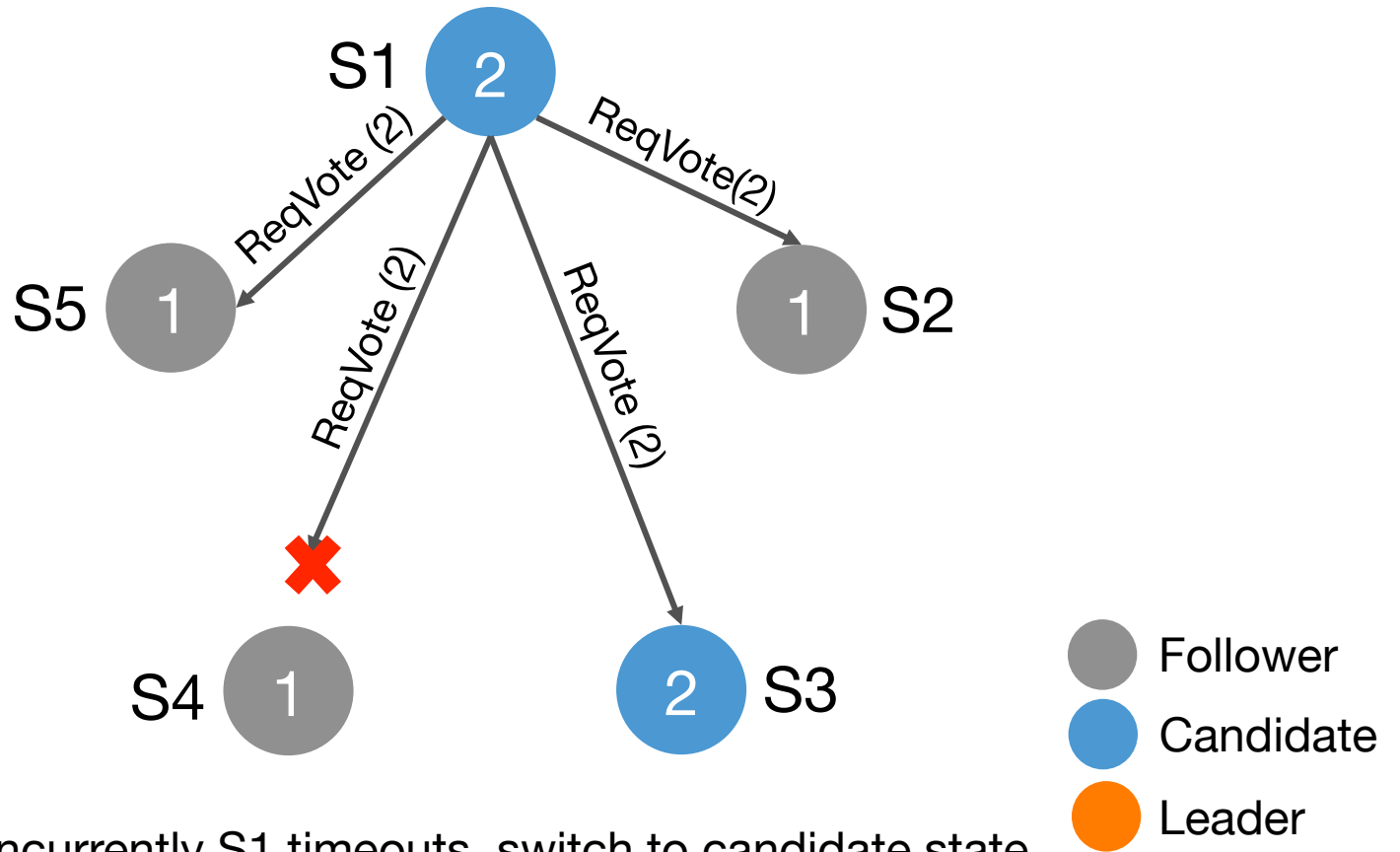
Remain followers as long as they receive valid RPCs from a leader or candidate

When a follower receives no communication over a period of time (the *election timeout*), it starts an election to pick a *new leader*

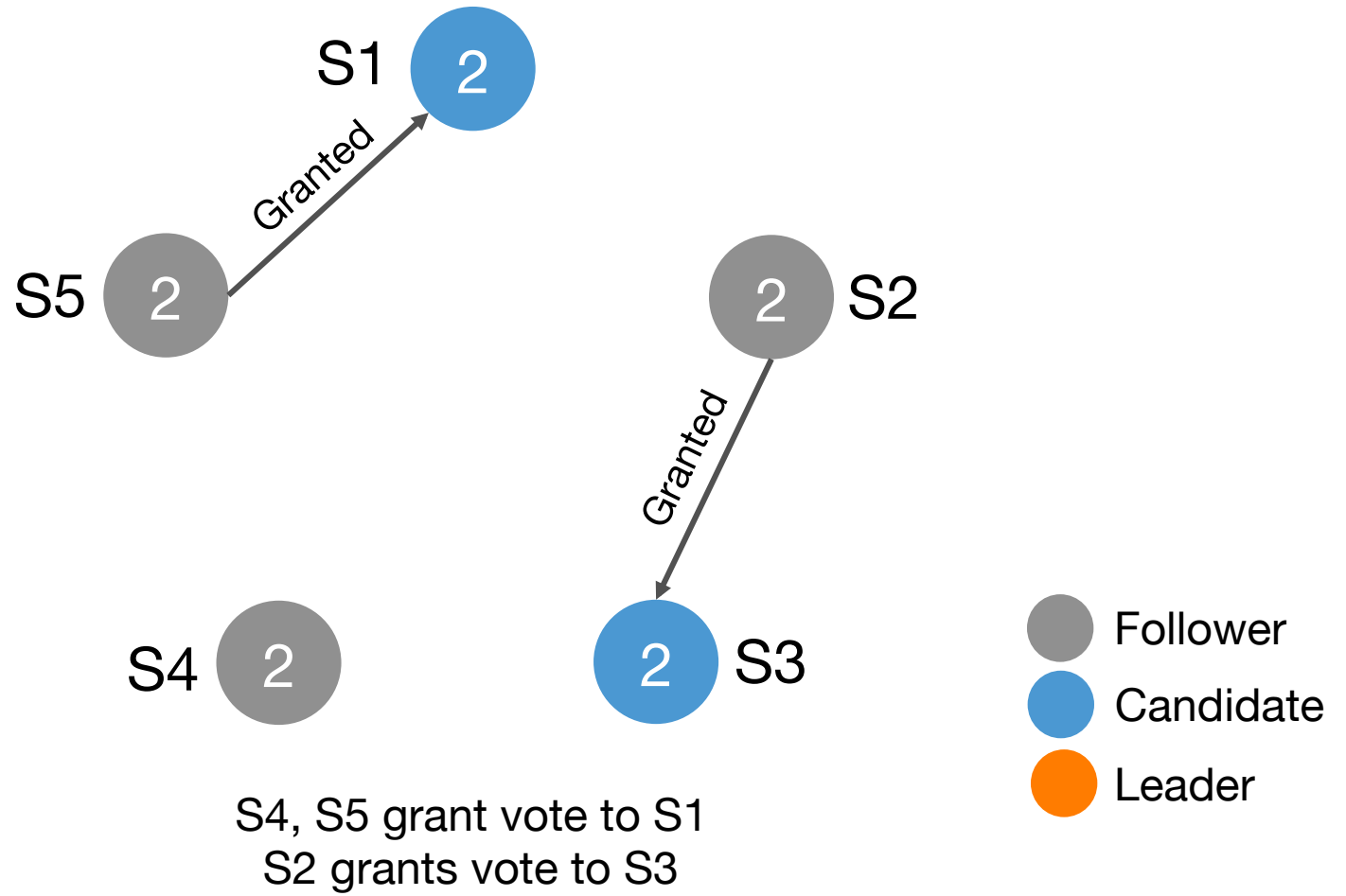


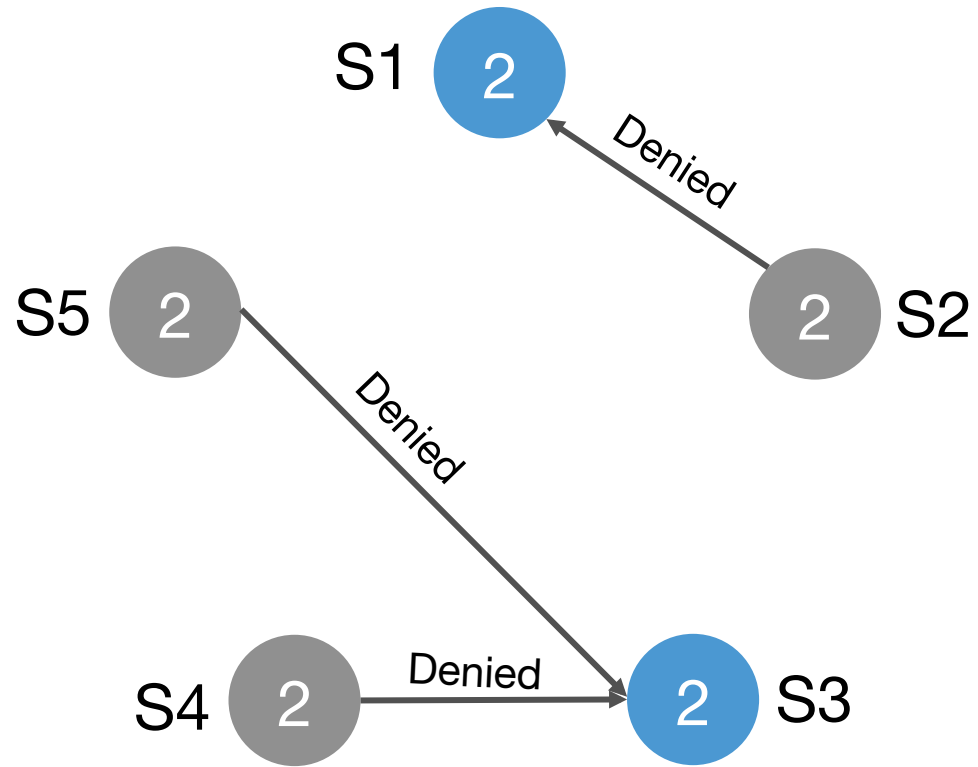





S3 timeouts, switch to candidate state, increment term, vote itself as a leader and ask everyone else to confirm

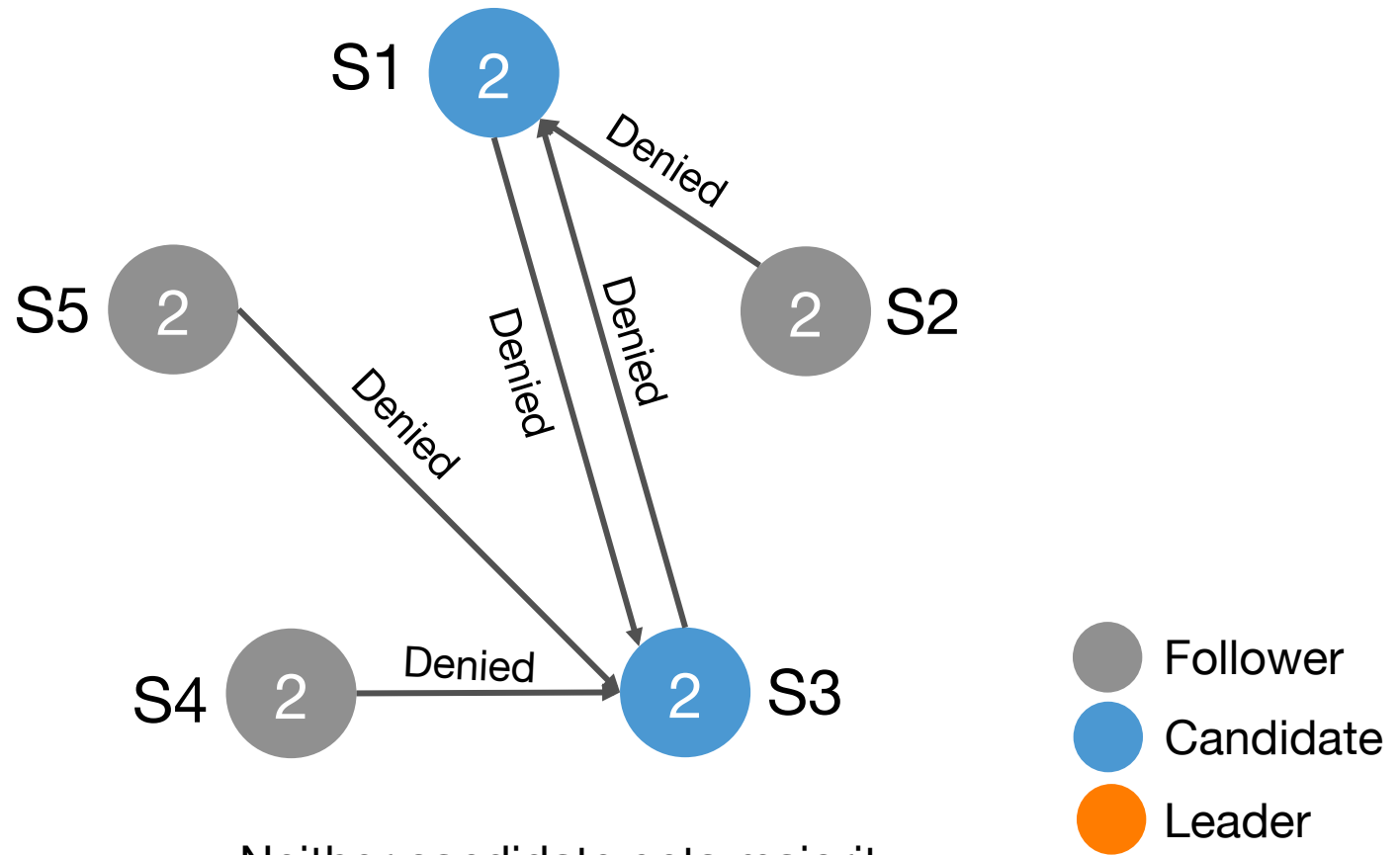


Concurrently S1 timeouts, switch to candidate state, increment term, vote itself as a leader and ask everyone else to confirm

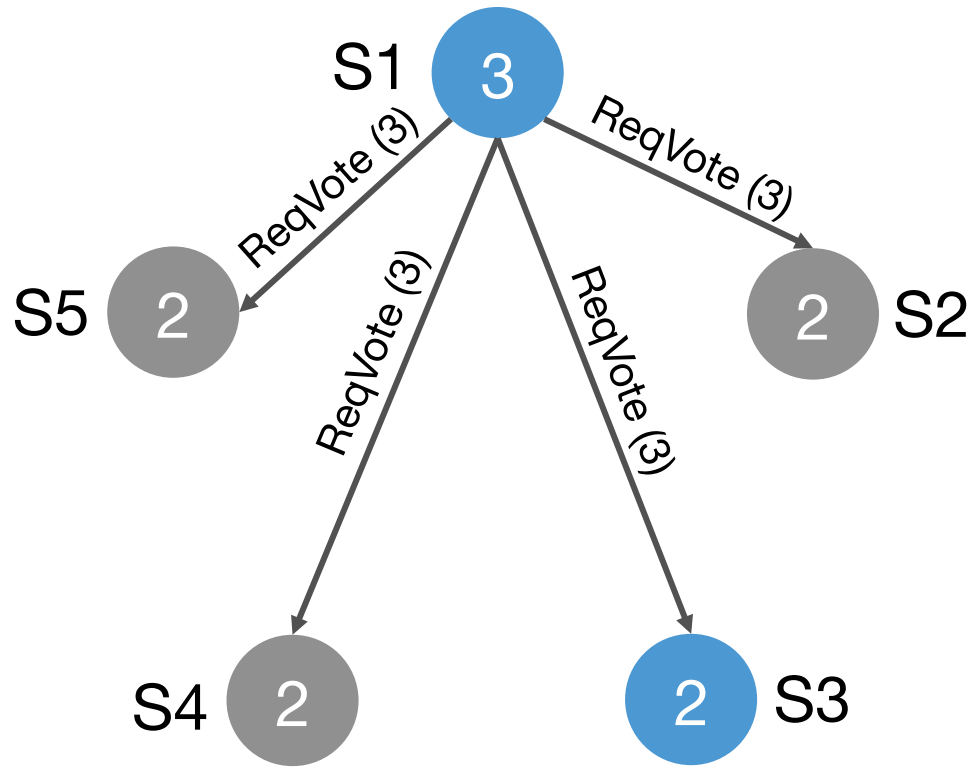




-  Follower
-  Candidate
-  Leader

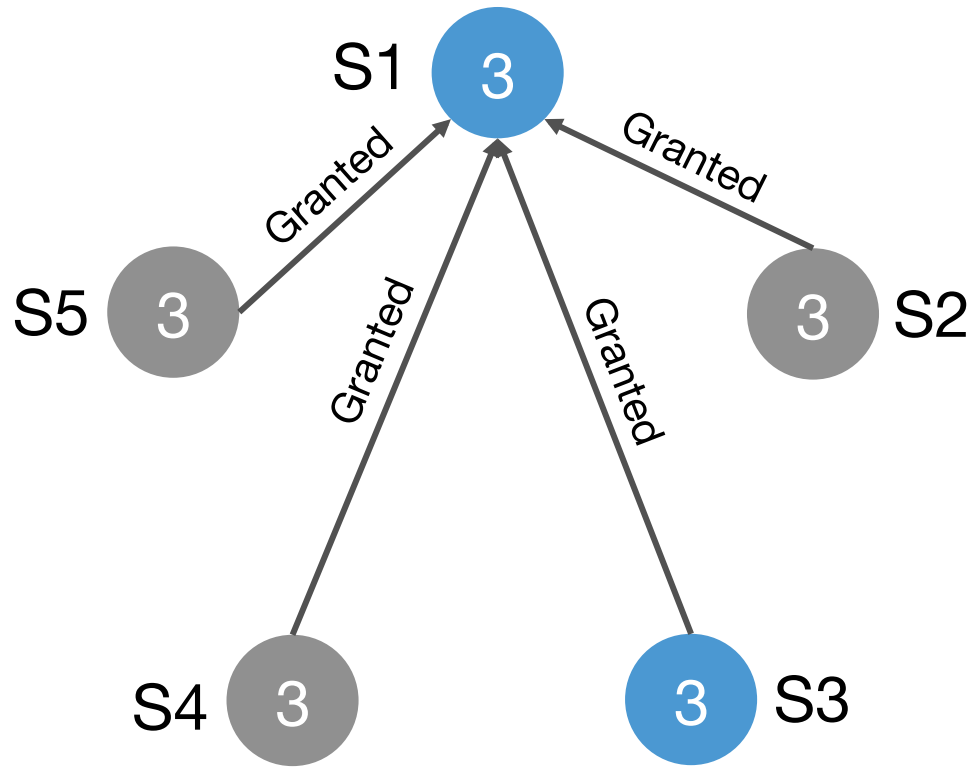


Neither candidate gets majority.
After a random delay between 150-300ms try again.



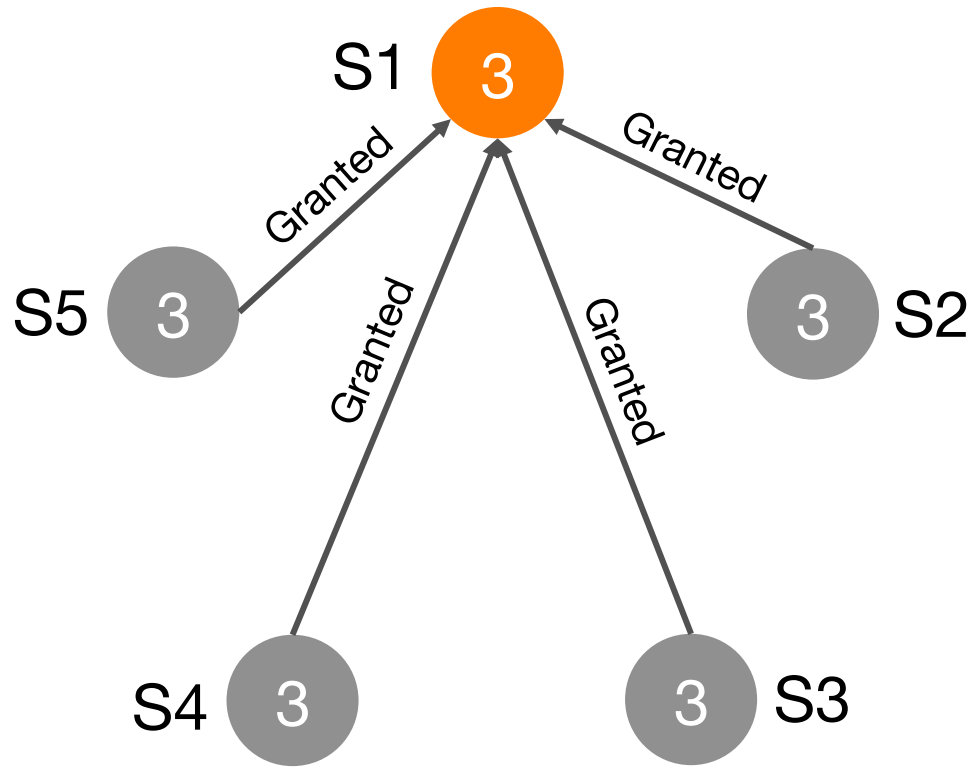
S1 initiates another election for term 3.

- Follower
- Candidate
- Leader



Everyone grants the vote to S1

- Follower
- Candidate
- Leader



- Follower
- Candidate
- Leader

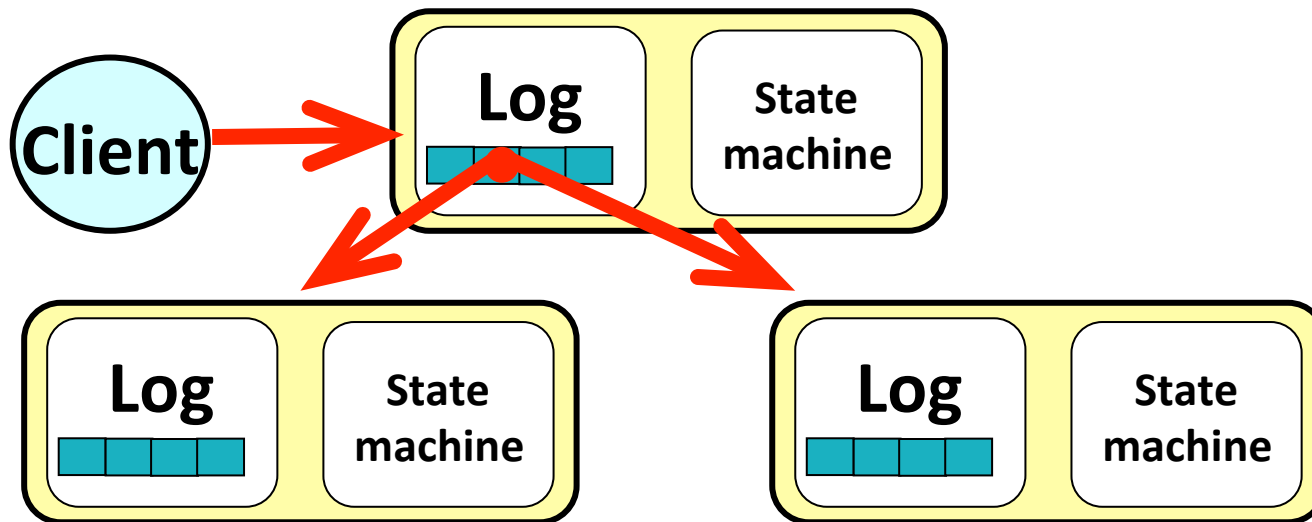
S1 becomes leader for term 3,
and the others become followers.

Log replication

Leaders

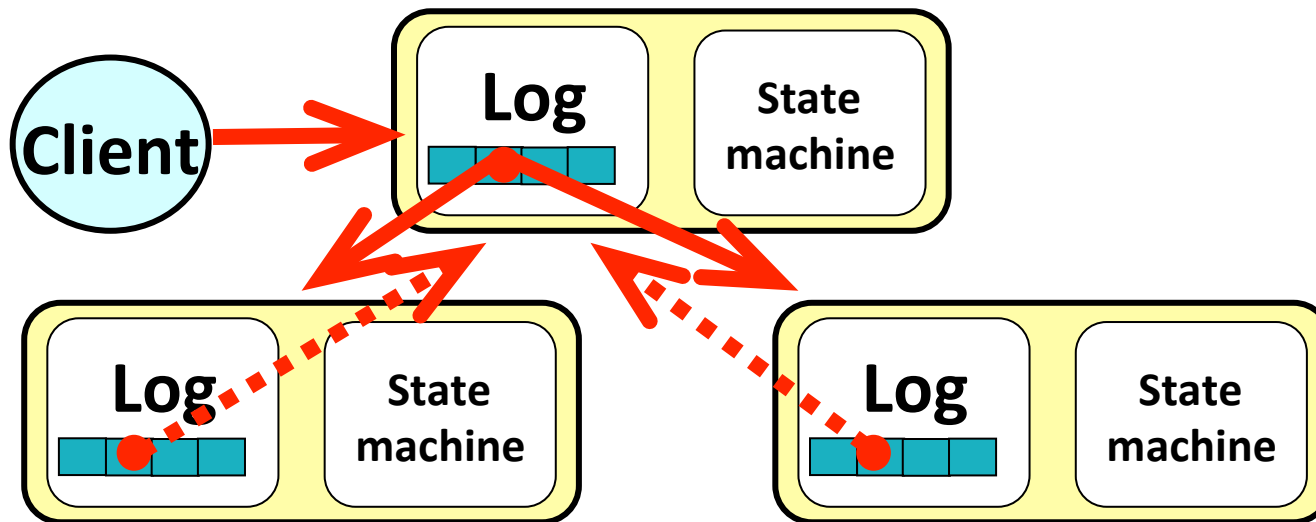
- Accept client commands
- Append them to their log (new entry)
- Issue **AppendEntry** RPCs in parallel to all followers
- Apply the entry to their state machine once it has been safely replicated
 - Entry is then *committed*

A client sends a request



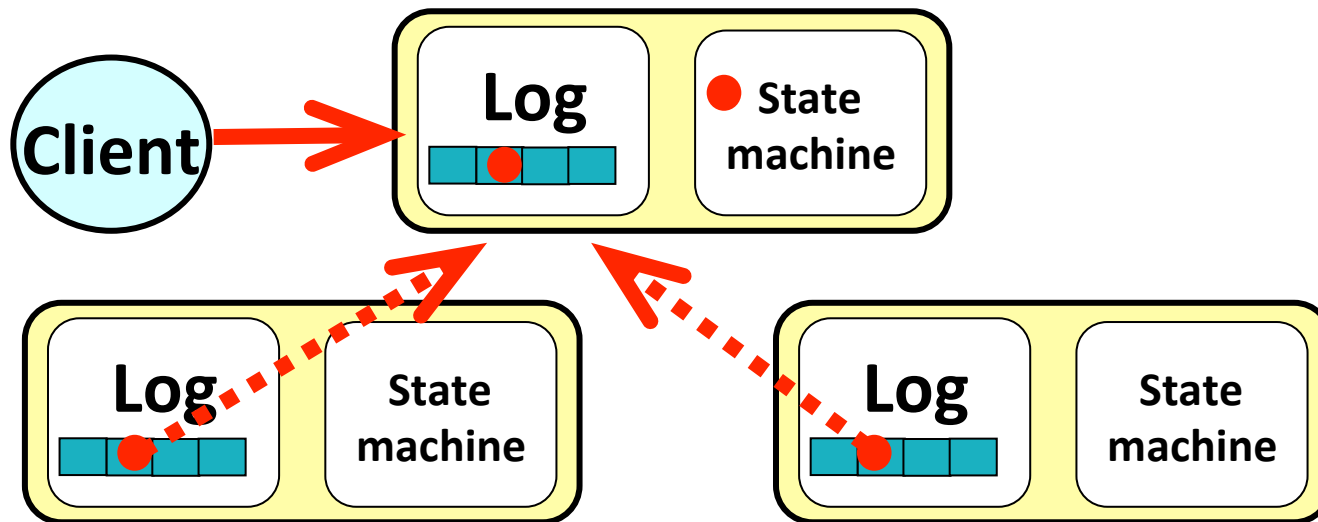
Leader stores request on its log and forwards it to its followers

The followers receive the request



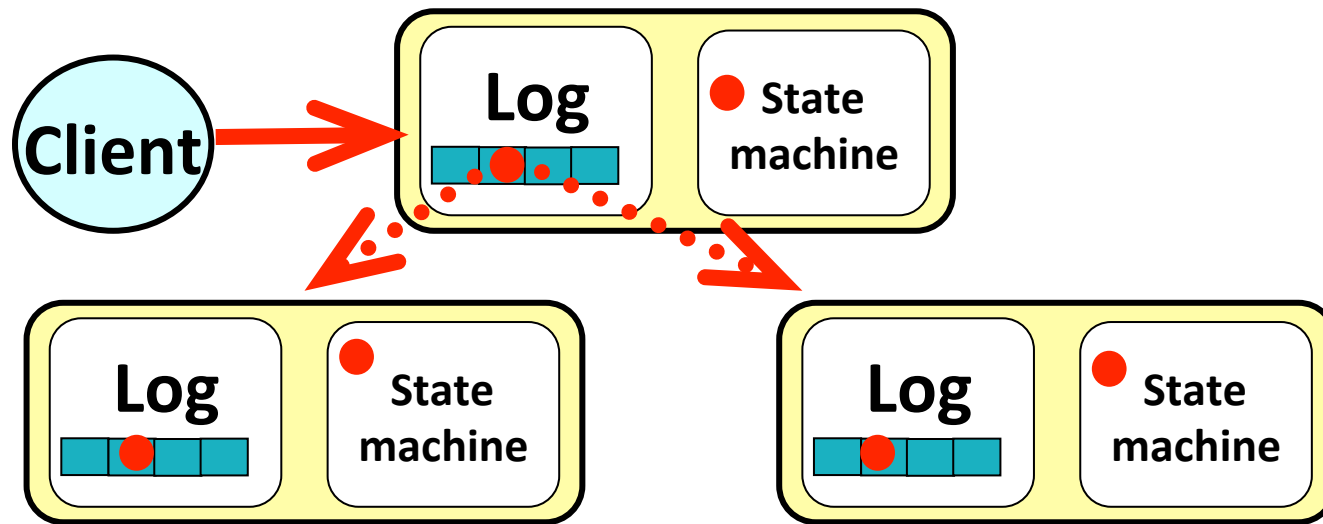
Followers store the request on their logs and acknowledge its receipt

The leader tallies followers' ACKs



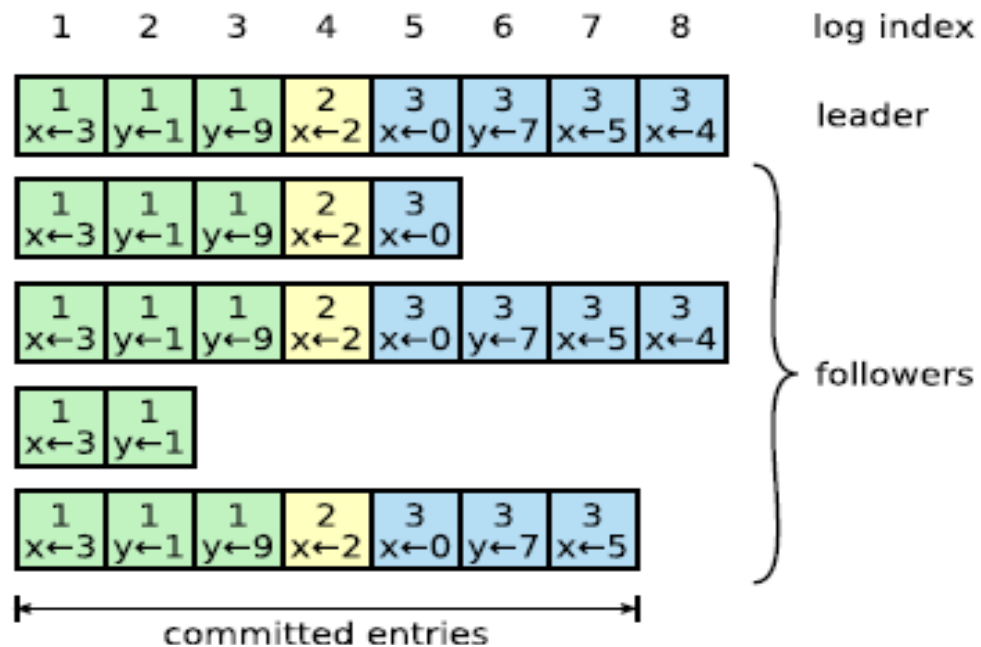
Once it ascertains the request has been processed by a majority of the servers, it updates its state machine

The leader tallies followers' ACKs



Leader's heartbeats convey the news to its followers: they update their state machines

Log organization



Colors identify terms

Handling slow followers ,...

Leader reissues the AppendEntry RPC

- They are idempotent

Committed entries

Guaranteed to be both

- Durable
- Eventually executed by all the available state machine

Committing an entry also commits all previous entries

- All AppendEntry RPCs—including heartbeats—include the index of its most recently committed entry

Why?

Raft commits entries in *strictly sequential order*

- Requires followers to accept log entry appends in the same sequential order
 - *Cannot "skip" entries*

Greatly simplifies the protocol

Raft log matching property

If two entries in different logs have the same index and term

- These entries store the same command
- *All previous entries* in the two logs are *identical*

1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
1 x←3	1 y←1						

Safety

Two main questions

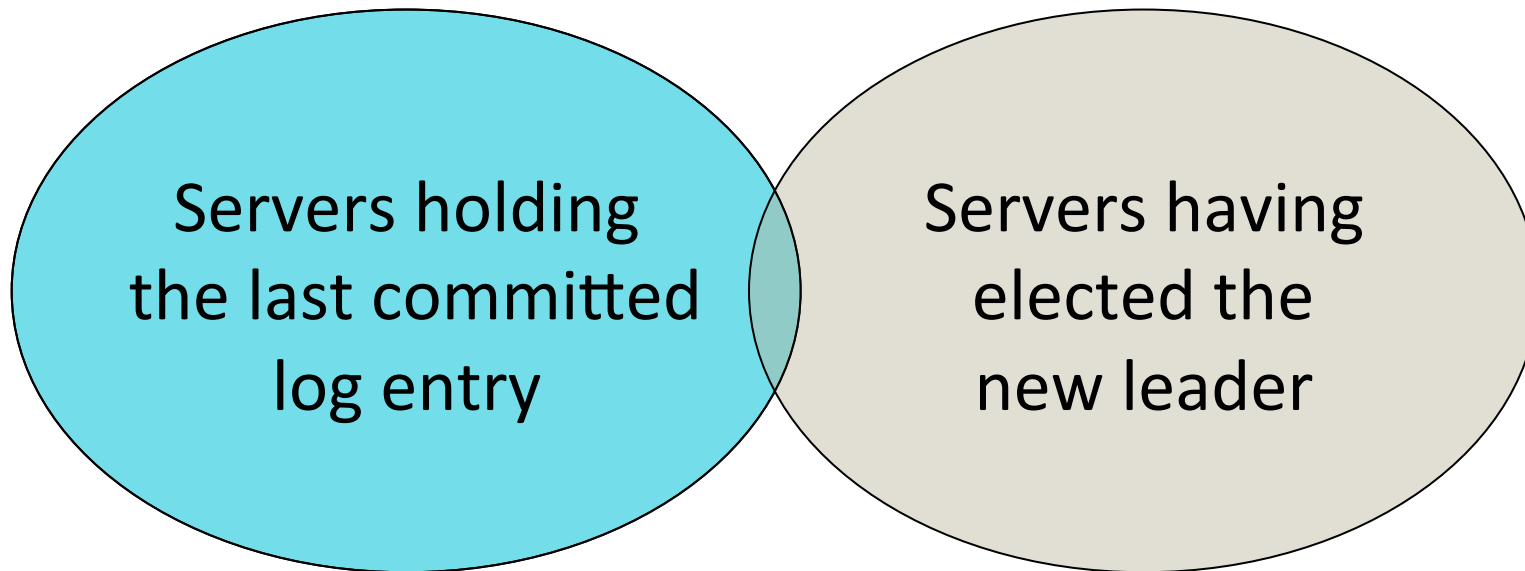
1. What if the log of a new leader did not contain all previously committed entries?
 - Must impose conditions on new leaders
2. How to commit entries from a previous term?
 - Must tune the commit mechanism

Election restriction (I)

The log of any new leader *must* contain all previously committed entries

- Candidates include in their *RequestVote* RPCs information about the state of their log
- Before voting for a candidate, servers check that the log of the candidate is at least as up to date as their own log.
 - Majority rule does the rest

Election restriction (II)



Two majorities of the same cluster ***must*** intersect

Committing entries from previous term

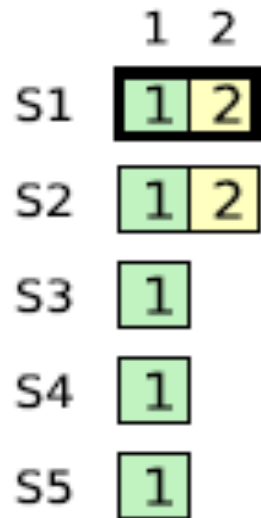
A leader cannot conclude that an entry from a previous term is committed even if stored on a majority of servers.

Leader should never commits log entries from previous terms by counting replicas

Should only do it for entries from the current term

Once it has been able to do that for one entry, all prior entries are committed indirectly

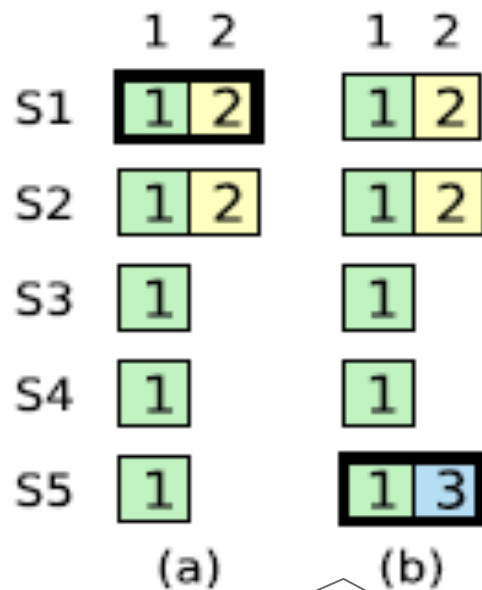
Committing entries from previous term



(a)

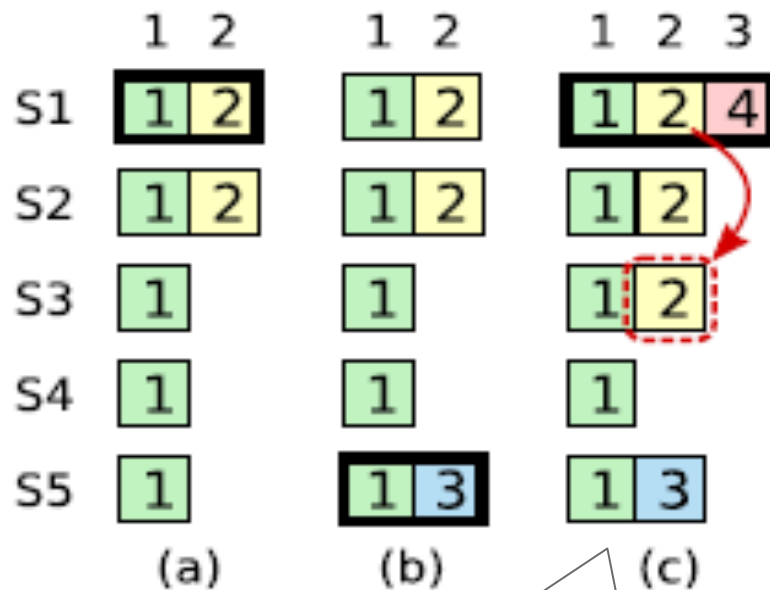
S1 is leader and partially replicates the log entry at index 2.

Committing entries from previous term



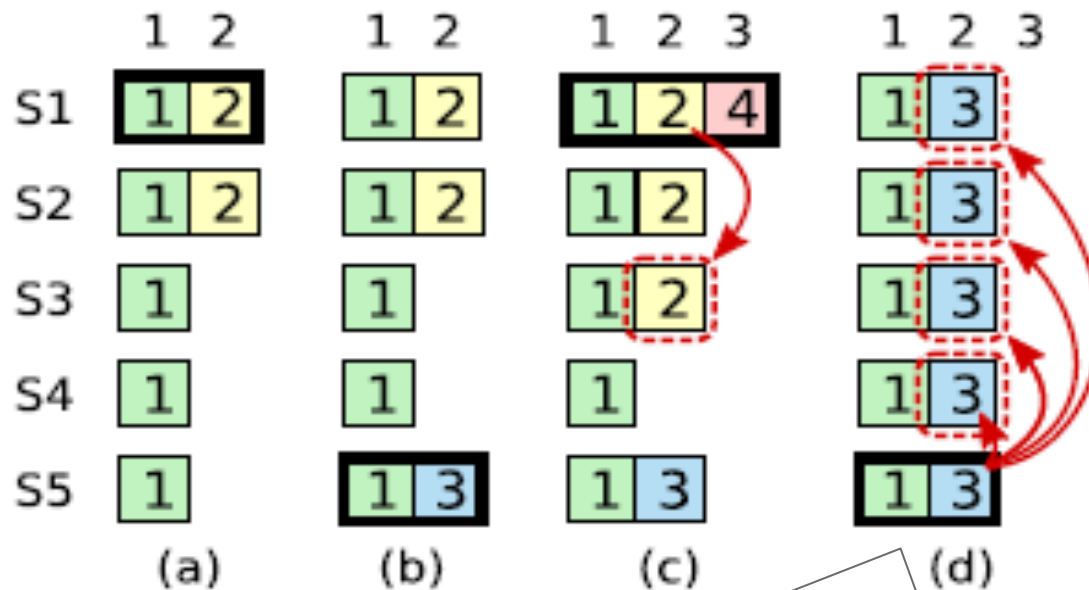
S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2.

Committing entries from previous term



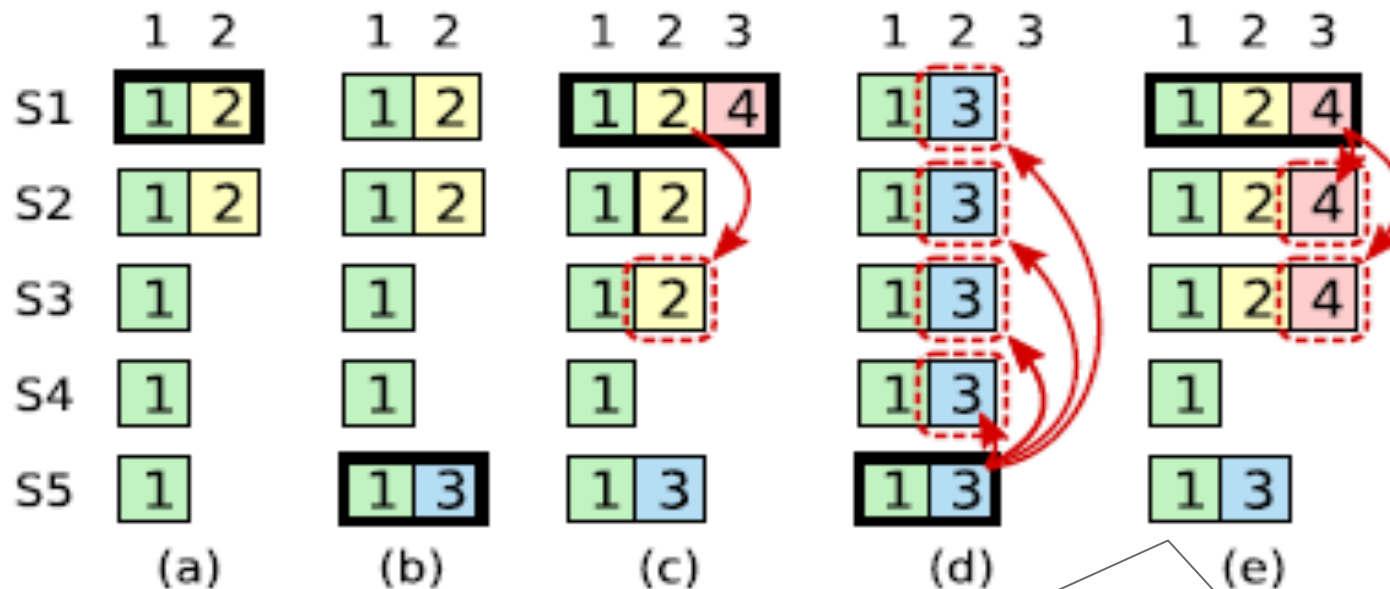
S5 crashes; S1 restarts, is elected leader, and continues replication

Committing entries from previous term



S1 crashes, S5 is elected leader (with votes from S2, S3, and S4) and overwrites the entry with its own entry from term 3.

Committing entries from previous term



However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as this entry is committed (S5 cannot win an election).

Summary

Consensus key building block in distributed systems

Raft similar to Paxos

Raft arguably easier to understand than Paxos

- It separates stages which reduces the algorithm state space
- Provides a more detailed implementation