Paxos for Beginners

875350879@qq.com

May 3, 2020

Motivation

- For a process providing services, we want it to be fault-tolerant
- Replicate it with several identical processes into a group
  - An abstraction as a single process to outside clients
- **Goal**: all servers eventually reach the same state

Replicated State Machines

- A server can be viewed as a deterministic finite state machine that performs client commands in some sequence
- For a group of replicas, they will all produce the same sequence of states if they execute the same sequence of commands
- Achieve this by a **replicated log** saving the command sequence
- The group members need to reach consensus on which command to execute for each entry

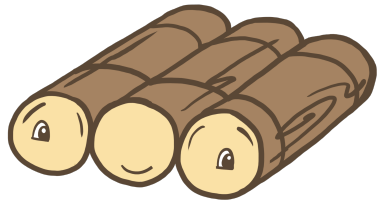
Centralized Approach?

- Why not just let one server determine which command to execute?
- Ensuring there is only one leader all the time could be costy!
- All the members need to know the new leader despite the existence of the old leader
- Better keep a leader at most of the time, but also allow consensus in the cases where multiple nodes think they are leaders

Paxos

- An algorithm achieving consensus on a single value given multiple ones
- For computers communicating via an asynchronous network
- Key idea: majority = consensus achieved
- Foundation of many applications[1] and later algorithms[2]

Apache ZooKeeper™



Raft

[1] https://github.com/Tencent/phxpaxos

[2] Ongaro, Diego and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. USENIX Annual Technical Conference (2014).

Modeling: assumptions

- Each member knows the number of nodes in the group
- Messages passing through the network can be delayed/lost, out of order, duplicated, but not corrupted
- Fail-stop
  - may crash, stop and restart, but must operate correctly when on
  - a process can place a worst-case delay on responses from another

Modeling: objects

- Proposers
  - Handle client requests containing values
  - Propose values to be chosen
  - Each proposal contains a value and a unique natural number called **proposal number** for distinction
- Acceptors
  - Respond to messages from proposers
  - Responses are votes that form consensus
- Learners
  - Want to know which value is chosen
  - Execute the value if it's a command
- In practice a node usually acts all three roles simultaneously

Strawman: single acceptor

- Single acceptor: single point of failure
- Quorum: majority = consensus
  - Multiple acceptors: any odd number larger than 1
  - A value is chosen iff accepted by a majority of acceptors
  - When a minority crash, still work

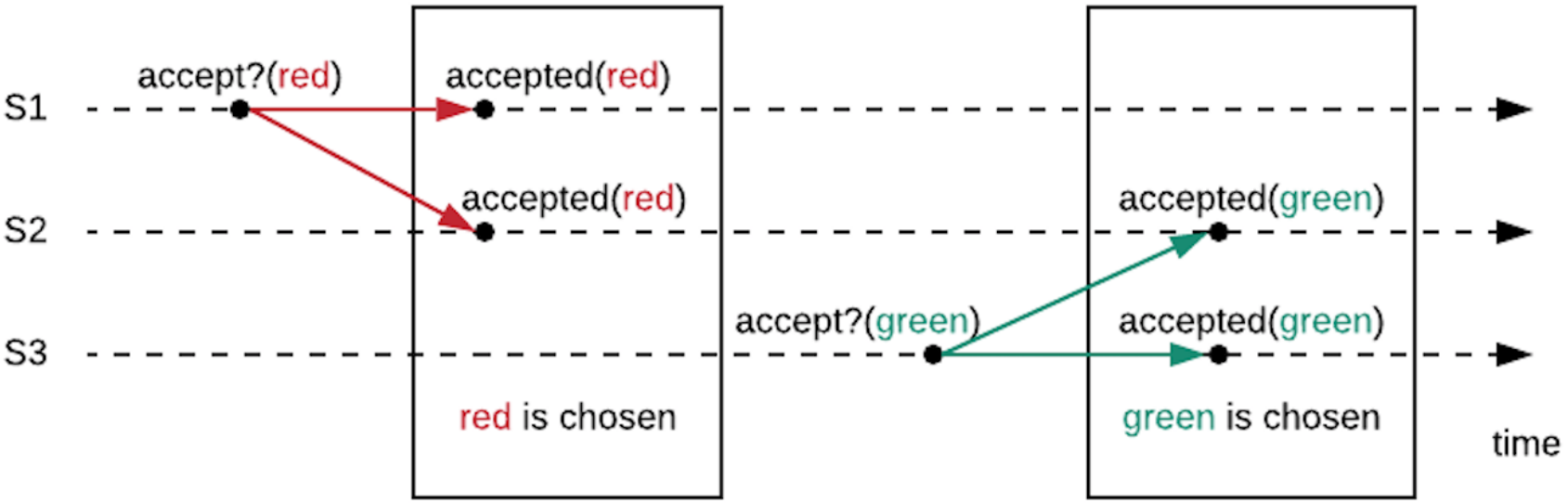
Conflicts

- The acceptor should accept the first value proposed
- May not reach a consensus!

- Solution
  - Either acceptors can accept multiple values (to avoid changing accepted values), or
  - A phase for proposers to change their minds

Accept multiple values?

- If acceptors accept any value received ...



- If the accepted value can't be changed, proposers must negotiate to propose the same value eventually!

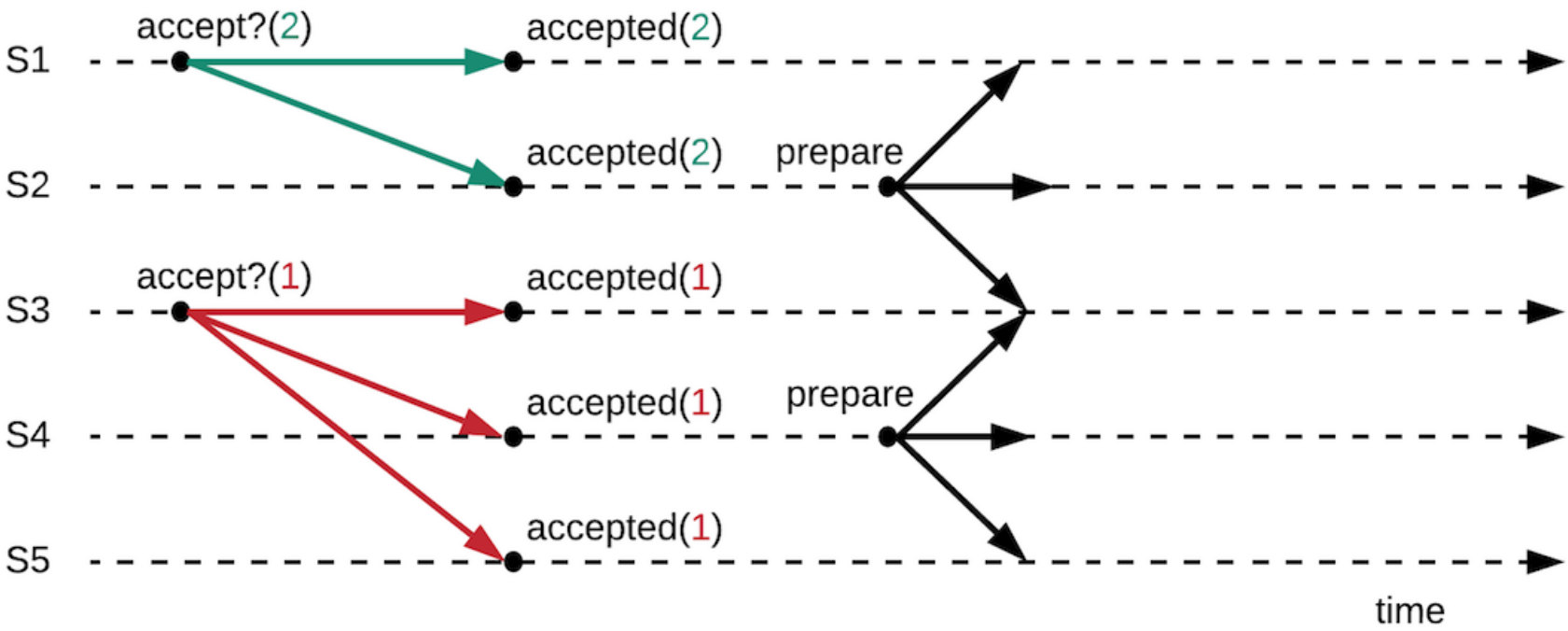
Ask about the accepted value

- The proposer need to contact a majority of acceptors before proposing its own value
  - To see whether there are values already accepted
- Each proposer send a **prepare** to all acceptors
- If an acceptor has already accepted a value, reply a message with the value

Propose the same value

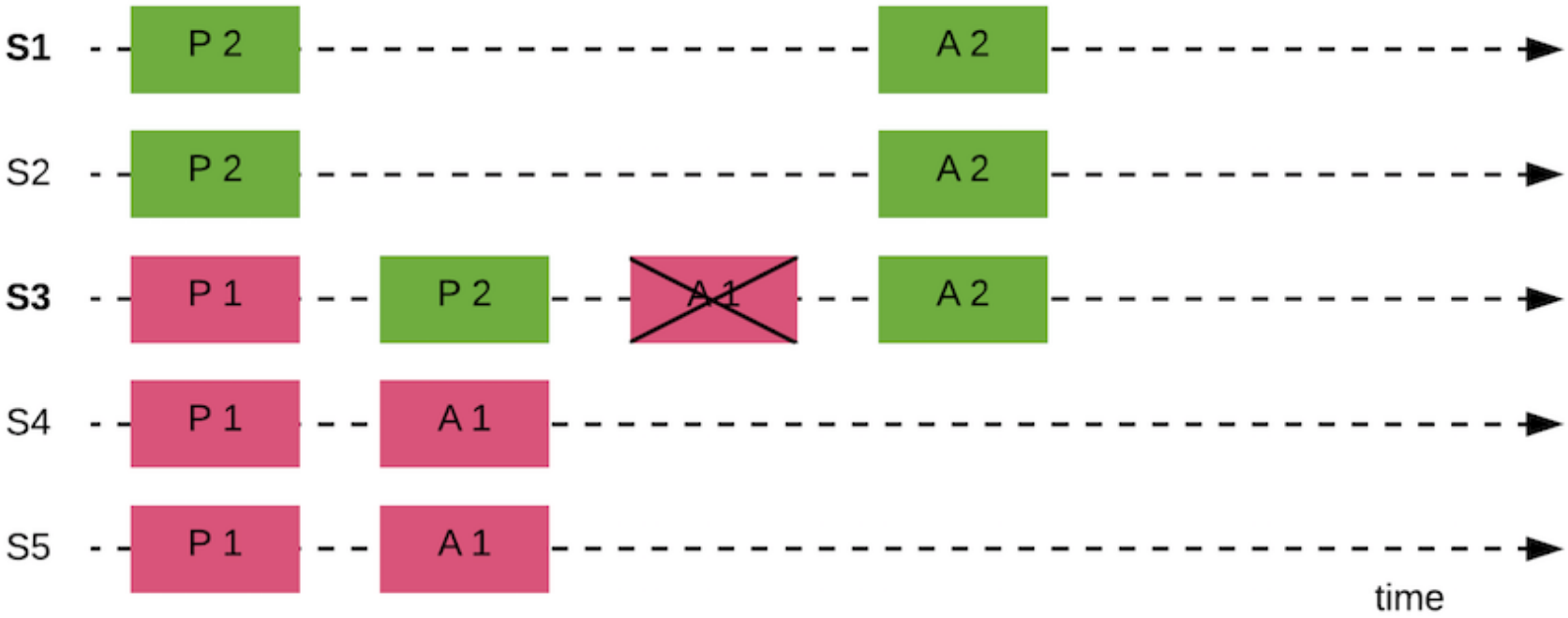
- Now consider all the proposers who receive replies from a majority of acceptors
- Different acceptors may have accepted different values
- Follow the one with the highest accepted proposal number
  - Each proposer attaches the proposal number to its **accept**s
  - Acceptors store **acceptedValue** and **acceptedProposal**
- Still, different proposers may see different values!

Different Views



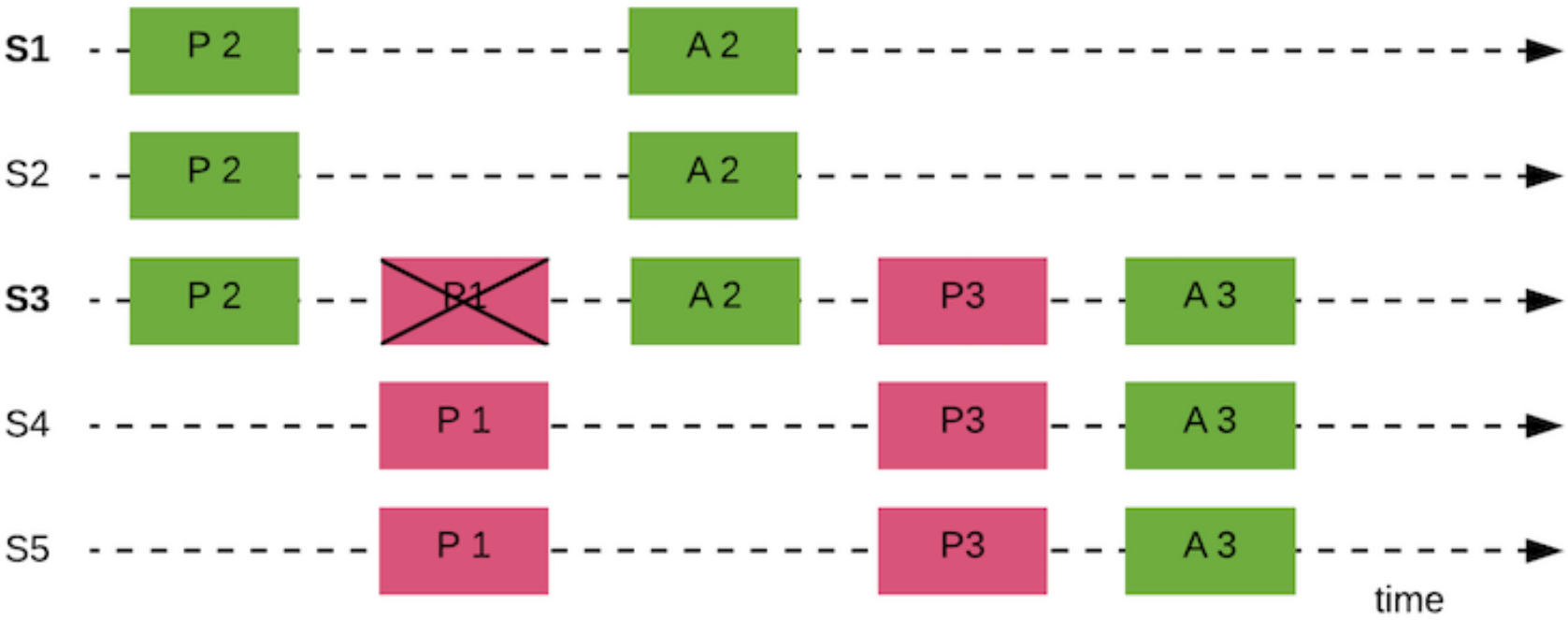
- Problem: the acceptedValue with the highest acceptedProposal might not be accepted by majority!
- Solution: each proposer must gain a majority of **promise**s from acceptors before proposing values

Propose the same value, example 1



- The Color indicates value, bold indicates proposers, P=sends a **prepare** & receives a **promise** , A=sends an **accept** & receives an **accepted**
- When an acceptor **promise** to a **prepare** with proposal number n, it would reject **accept** with lower proposal number

Propose the same value, example 2

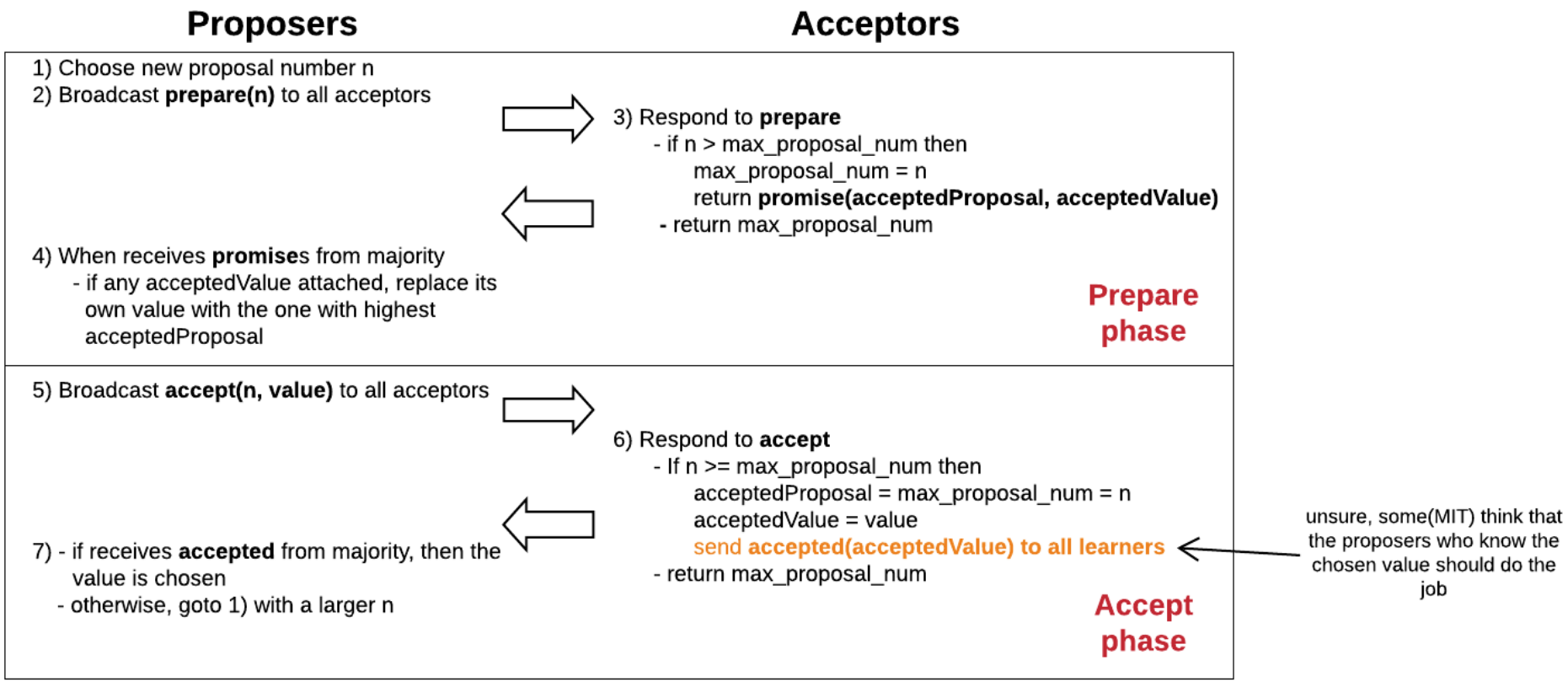


- When an acceptor **promise** to a **prepare** with proposal number n, it would also reject **prepare** with lower proposal number

Propose the same value, continue

- Each proposer must gain a majority of **promise**s from acceptors before proposing values
  - Attach the proposal number to **prepare**
  - The proposer with a larger proposal number gets the vote
- For each acceptor,
  - Maintain a variable `max_proposal_num`
  - Ignore any **prepare** with a proposal number n <=`max_proposal_num`
  - Otherwise replies a **promise** indicating it won't listen to any proposer with lower proposal numbers
  - Attach the accepted value to the **promise** if any

The whole process

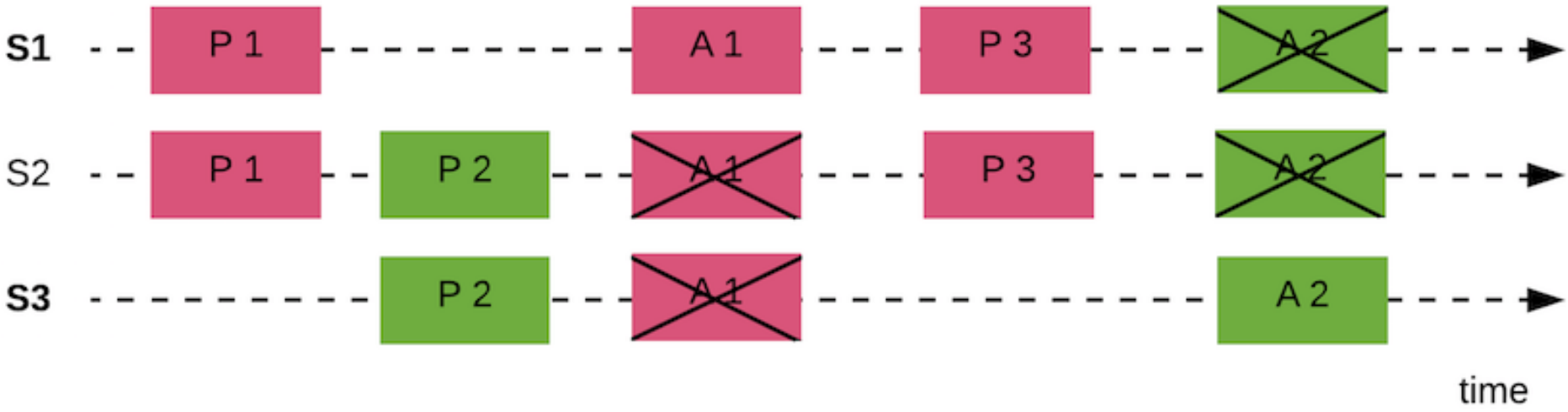


- acceptedProposal, max_proposal_num, acceptedValue must be stored on disks

Learners

- Receive notifications from acceptors
- When a majority of acceptors notify the same value, that is the chosen one
- To eventually learn,
  - Either acceptors keep trying sending notifications until they know that learners have received,
  - Or some distinguished learners take the job

Livelock



- May loop forever! (trying to gain promises from majority)
- Solution
  - A randomize/exponential delay before restarting for one to proceed
  - Set a leader

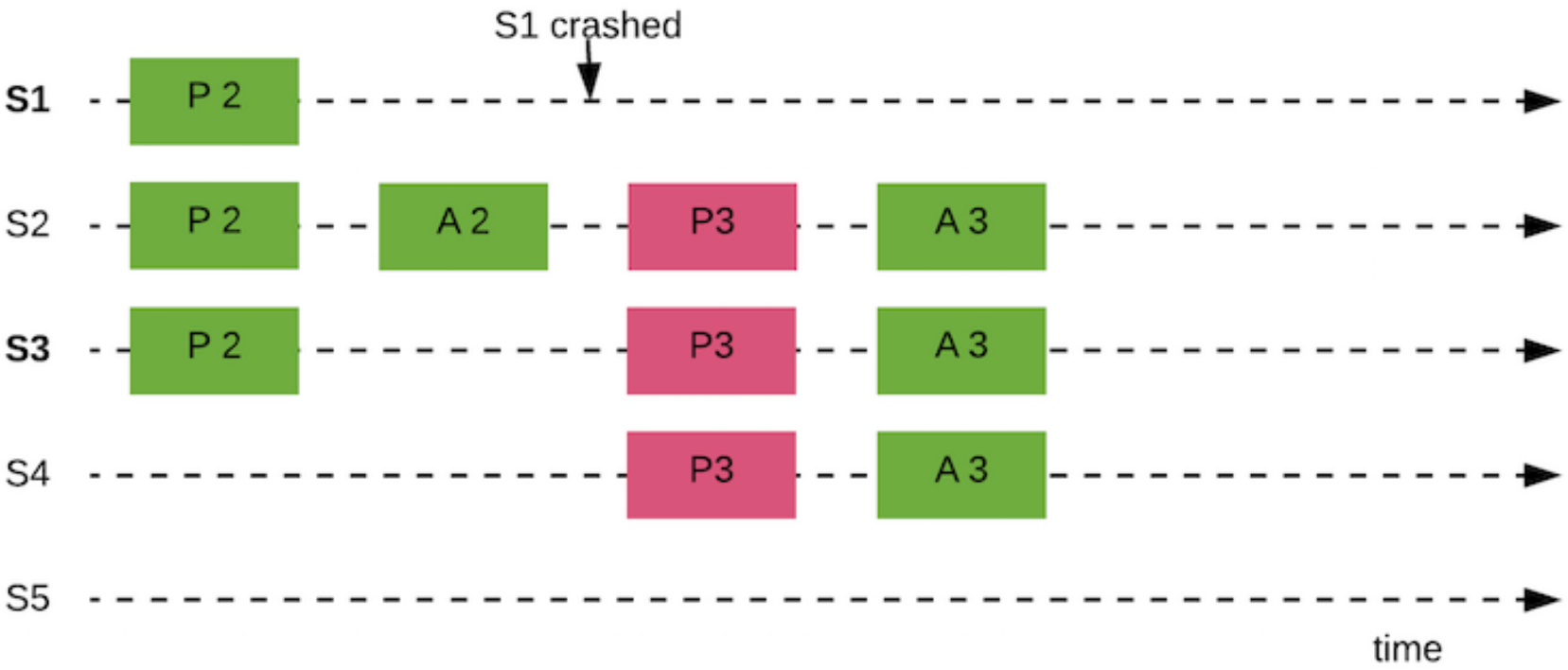
Proposer failures: prepare phase

- If the proposer fails to make any acceptor accept its value before failure, then it's the same as if nothing happens
- The other proposers will eventually gain the **promises** from a majority of acceptors by a higher proposal number

Proposer failures: accept phase

- A proposer that takes over does not know it is taking over a pending consensus
- It simply proposes a value
- If it doesn't get any responses from a majority of acceptors that contains acceptedProposal and acceptedValue
  - No majority agreement yet
  - The proposer then just executes normally and finishes its propose
- Else it just executes using the previous value with the highest acceptedProposal and finishes the protocol

Proposer failures: accept phase



Acceptor failures

- When a majority of them fail, proposers wait until a majority is available
- When fewer than majority fail, proceed as usual

Conclusion

- An algorithm that selects one value among multiple proposals
- **Safety**
  - Only a single value may be chosen
  - Any role never learns that a value has been chosen unless it really has been
- **Liveness**
  - Some proposed value is eventually chosen
  - If a value is chosen, learners eventually learn about it

Multi-Paxos

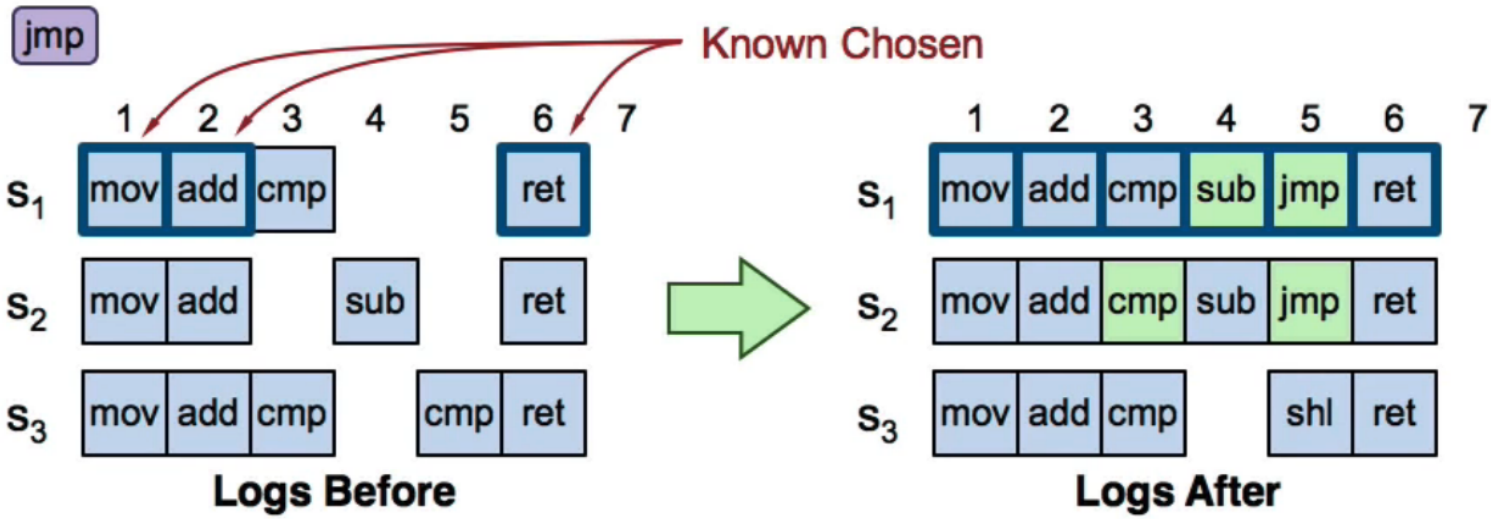
- (Basic) Paxos is for single value consensus
  - Our goal is to reach consensus on a sequence of values
- Multi-Paxos: for each log entry, run an instance of Paxos
- All three roles need to be aware of the index of the log entry
  - Attach the log entry index into **prepare**, **promise**, **accept**, **accepted**
- Multiple entries of values can be determined concurrently
  - Though executed by learners sequentially

Hole

- Which index for the new client command?
- For different entries, it's usually the case that different majority achieves the consensus
- **Hole**s may appear
  - For the same entry, some see accepted values while others not
  - E.g. entries 1-3 requested simultaneously, while the value for entry 2 fails to be chosen
  - It could be restarted or replaced by a new value (for entry 2)
    - since the new value will be from a different client
    - for the same client, it doesn't violate FIFO (if the client side handles properly)

Hole[3], continue

- When a request arrives from client:
  - Find first log entry not known to be chosen
  - Run Paxos to propose client's command for this index
  - prepare returns acceptedValue?
    - Yes, finish choosing acceptedValue and start again
    - No, choose client's command



[3]Credit: Diego Ongaro, Implementing Replicated Logs with Paxos

Full Replication

- For a log entry, only proposers who propose it surely know it's chosen
- Solution
  - Each server marks entries i known to be chosen: acceptedProposal[i] = ∞; maintains **firstUnchosenIndex**
- Proposers tell (other) acceptors about chosen entries
  - Proposers keep retrying sending **accept**s until all acceptors respond
  - Proposers attach its firstUnchosenIndex in **accept**s
  - Acceptors mark all entries i chosen if
    - i $<$ **accept** .firstUnchosenIndex, and
    - acceptedProposal[i] $==$ **accept** .proposal_num
  - Acceptors return their firstUnchosenIndexes in **accept** replies
  - If proposer's firstUnchosenIndex $<$ acceptor's firstUnchosenIndex, keep informing it until equal

More Efficient

- Each proposer broadcasts at least twice! (**prepare** , **accept**)
- Elect a leader (e.g. Bully Algorithm)
- Eliminating unncessary **prepare**s
  - Make **prepare** &**promise** refer to the whole log instead of an entry
  - Each acceptor replies **prepare** with **noMoreAccepted** if there are no more entries accepted later than the index associated with the **prepare**
- If the leader receives noMoreAccepted from an acceptor, skip future **prepare**s with it
- Once the leader receives noMoreAccepted from majority acceptors, no need for **prepare**s

Reference

1. Lamport, Leslie. Paxos Made Simple. (2001).
2. Lamport, Leslie. The part-time parliament. ACM Trans. Comput. Syst. 16 (1998): 133-169.
3. Diego Ongaro, "Implementing Replicated Logs with Paxos", Stanford Unversity lecture video, 2013.
4. Rutgers CSC417 Course Notes
   `https://www.cs.rutgers.edu/~pxk/417/notes/paxos.html`