EECS 262a
Advanced Topics in Computer Systems
Lecture 18

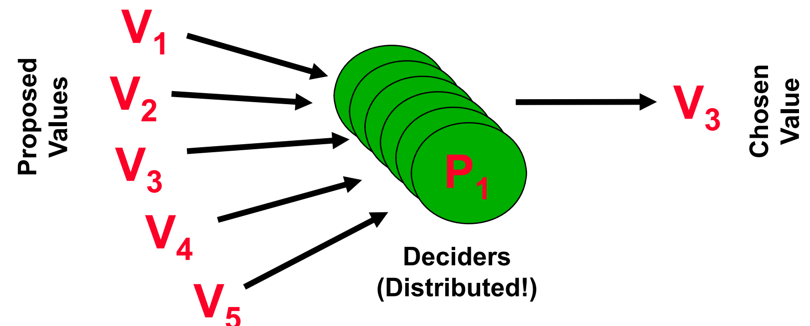
Paxos/RAFT
October 26th, 2021

John Kubiatowicz
(Lots of borrowed slides see attributions inline)
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

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/acted on



Today's Papers

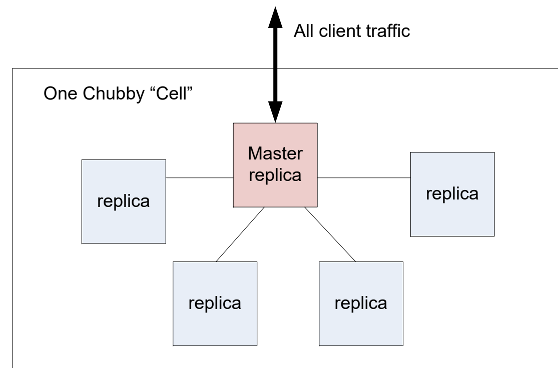
- Paxos Made Live - An Engineering Perspective, Tushar Chandra, Robert Griesemer, and Joshua Redstone. Appears in *Proceedings of the Symposium on Principles of Distributed Computing* (PODC), 2007
- In Search of an Understandable Consensus Algorithm, Diego Ongaro and John Ousterhout, USENIX ATC'14

- Thoughts?

Google Chubby

- A *coarse-grained lock and small file storage service*
  - Other (Google) distributed systems can use this to synchronize access to shared resources
- Intended for use by "loosely-coupled distributed systems"
  - GFS: Elect a master
  - Bigtable: master election, client discovery, table service locking
  - Well-known location for bootstrapping larger systems
  - Partitioning workloads
- Goals:
  - High availability
  - Reliability
- Anti(non)-goals:
  - High performance, Throughput, Storage capacity

Distributed Consensus



- Chubby cell is usually 5 replicas
  - 3 must be alive for cell to be viable
- How do replicas in Chubby agree on their own master, official lock values?
  - Distributed commit algorithm

What about Two Phase Commit?

- Commit request/Voting phase
  - Coordinator sends query to commit
  - Cohorts prepare and reply – single abort vote causes complete abort
- Commit/Completion phase
  - Success: Commit and acknowledge
  - Failure: Rollback and acknowledge
- Disadvantage: Blocking protocol
  - Handles coordinator failures really poorly – blocks
  - Handles cohort failure poorly during voting phase – aborts

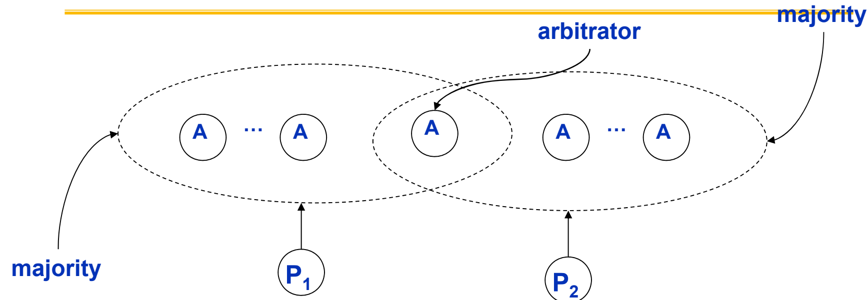
Basic Paxos (Quorum-based Consensus)

- Prepare and Promise
  - Proposer selects proposal number N and sends promise to acceptors
  - Acceptors accept or deny the promise
- Accept! and Accepted
  - Proposer sends out value
  - Acceptors respond to proposer and learners
- Paxos algorithm properties
  - Family of algorithms (by Leslie Lamport) designed to provide *distributed consensus* in a **network** of several **replicas**
  - Enables reaching consensus on a single binding of variable to value ("fact")
  - Tolerates delayed or reordered messages and replicas that fail by stopping
  - Tolerates up to N/2 replica failure (*i.e.*, *F* faults with *2F+1* replicas)

Paxos Assumptions

- Replica assumptions
  - Operate at arbitrary speed
  - Independent, random failures
  - Replicas with stable storage may rejoin protocol after failure
  - Do not lie, collude, or attempt to maliciously subvert the protocol
- Network assumptions
  - All processors can communicate with ("see") one another
  - Messages are sent asynchronously and may take arbitrarily long to deliver
  - Order of messages is not guaranteed: they may be lost, reordered, or duplicated
  - Messages, if delivered, are not corrupted in the process

Basic Paxos – Majority consensus



- Determines the authoritative value for a single variable
- Each proposer makes a proposal to some majority (quorum) of the acceptors; acceptors respond with latest value
- A majority (quorum) of acceptors must accept a proposal for the proposed value to be chosen as the consensus value
- If P_1 and P_2 are making different proposals, then there must be at least one acceptor that they share in common – this common acceptor decides which proposal prevails

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)

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***

Today's Assumptions:
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!)
  - Parliamentarians come and go
  - Somehow the law gets passed anyway and consistently

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 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/acts upon a value unless it was actually chosen
- Eventual liveness: If things go "well" at some point in the future (e.g., no longer losses or 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 $b \geq$ *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 $v_{pst} = p$, $v_{bal} = prevBal[p]$, and $v_{dec} = prevDec[p]$.

Receive *LastVote*(b, v) Message
If b = *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 $Q \subseteq \{v_{pst} : v \in prevVotes[p]\}$, 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 $v_{bal} \neq -\infty$ then $d = v_{dec}$, else d can equal any decree.
– Set $\mathcal{B}$ to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and $B_{bal} = lastTried[p]$.

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 = *nextBal*[p] > *prevBal*[p] then
– Set *prevBal*[p] to b.
– Set *prevDec*[p] to d.
– If there is a ballot B in $\mathcal{B}$ with $B_{bal} = b$ [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 = *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

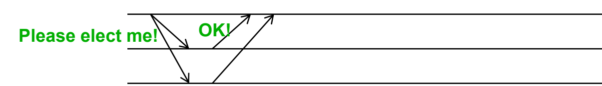
- Typically, every process is **acceptor**, **proposer**, and **learner**
- A **leader** is elected to be the distinguished proposer and learner
  - Distinguishes proposer to guarantee progress
    - » Avoid dueling proposers
  - Distinguishes learners to reduce too many broadcast messages

Recall…

- 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)

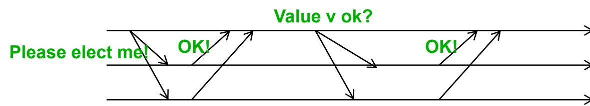
Phase 1 – Election

- Potential leader chooses *unique* ballot ID, *higher* than anything 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 focus on one leader case
  - Processes also log received ballot ID on disk (why?)
- 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?)



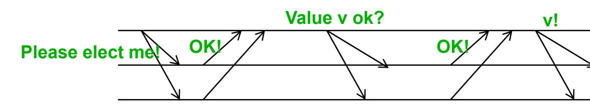
Phase 2 – Proposal (Bill)

- Leader sends proposal value v to all
  - If some process already decided value v' in a previous round sends v = v'
- Recipient log on disk, and responds OK



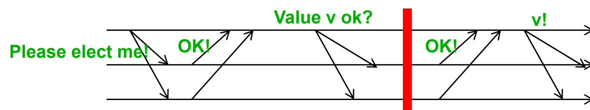
Phase 3 – Decision (Law)

- If leader hears OKs from majority, it lets everyone know of the decision
- Recipients receive decisions, log it on disk



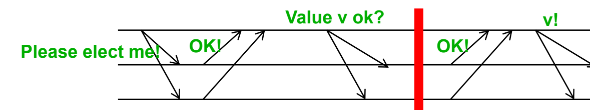
When is Consensus Achieved?

- 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 rounds until some round complete

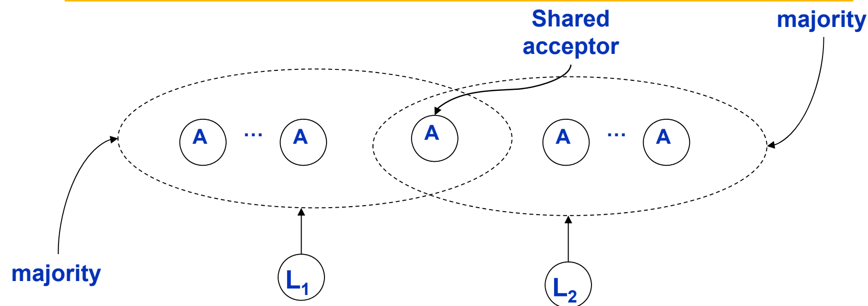


Safety

- 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



Safety (con't): What if L_1 fails before accepted majority noticed???



- Every majority shares one or more elements
- Thus, once value accepted, it will always be accepted

More Paxos in more detail…

Basic Paxos Protocol

Proposer

Phase 1a: "Prepare":
Select proposal number* *N* and send a *prepare(N)* request to a quorum of acceptors.

Acceptor

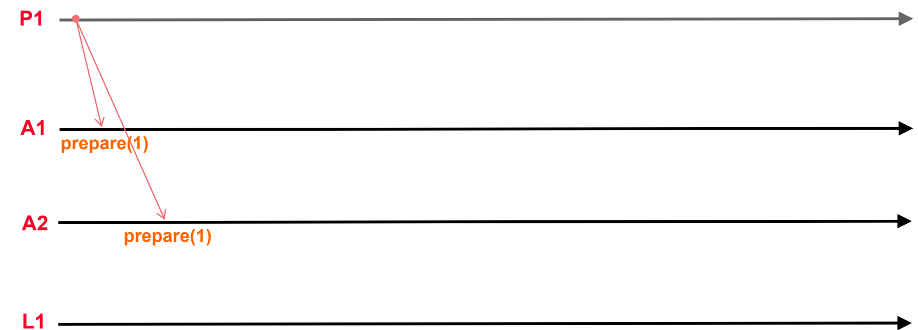
Phase 1b: "Promise":
If N > *number of any previous promises or acceptances*,
 * promise to never accept any future proposal less than *N*,
 - send a *promise(N, U)* response
 (where *U* is the highest-numbered proposal accepted so far (if any))

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)

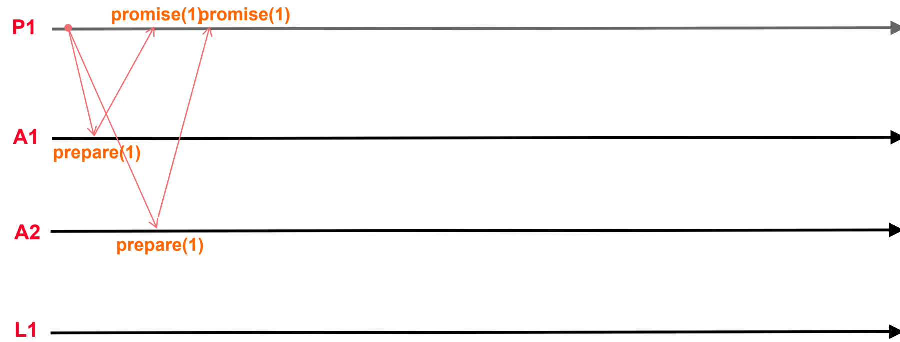
Phase 2b: "Accepted":
If N >= *number of any previous promise*,
 * accept the proposal
 - send an *accepted* notification to the learner

* = record to stable storage

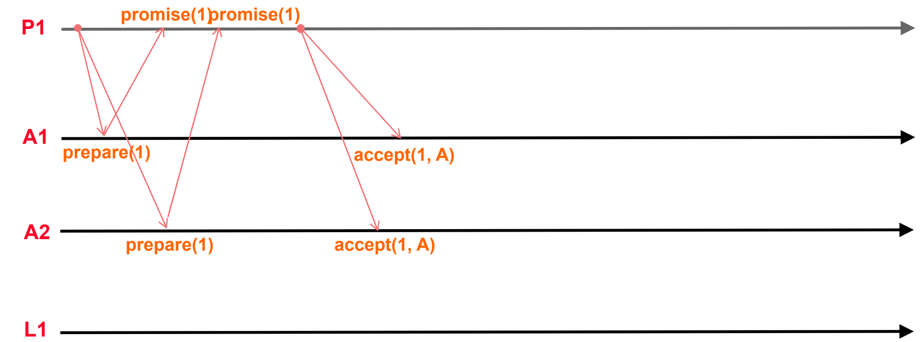
Trivial Example: P1 wants to propose "A"



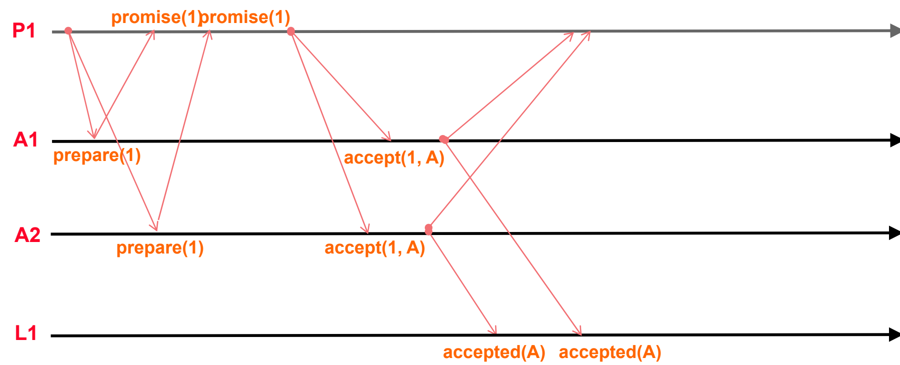
Trivial Example: P1 wants to propose "A"



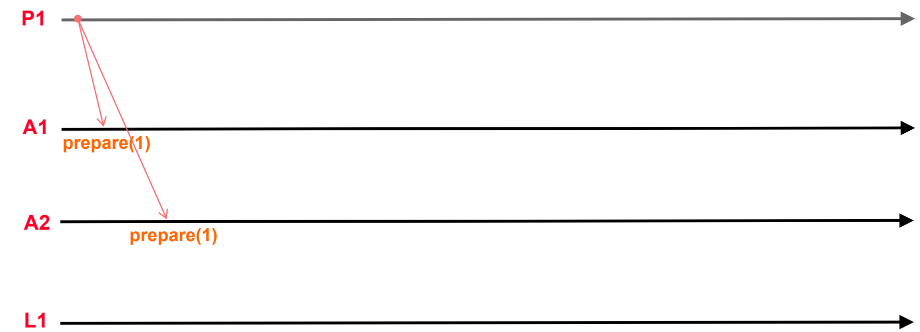
Trivial Example: P1 wants to propose "A"



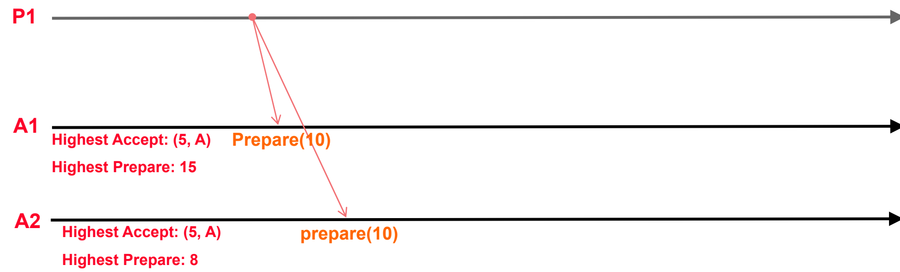
Trivial Example: P1 wants to propose "A"



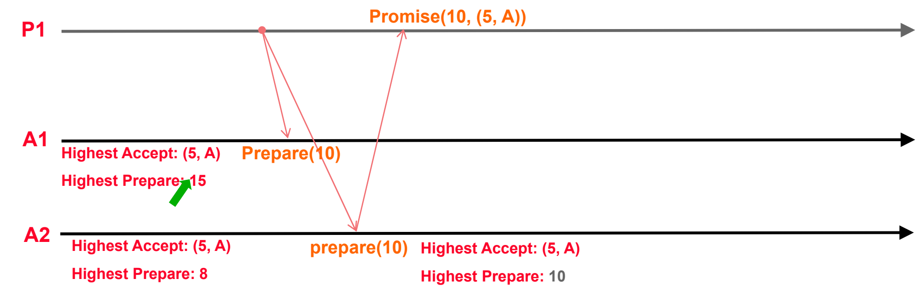
Example



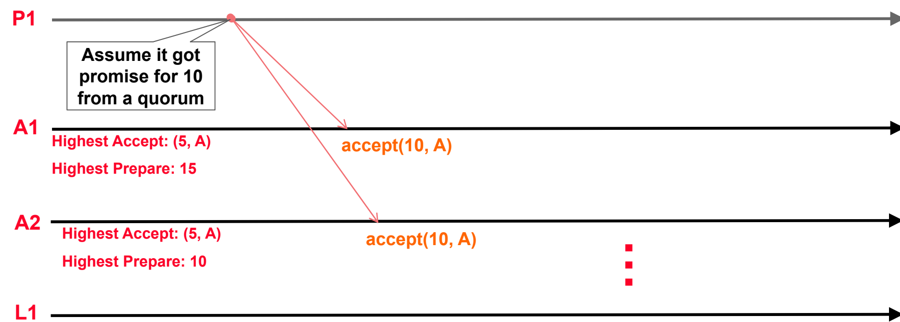
Prepare Example



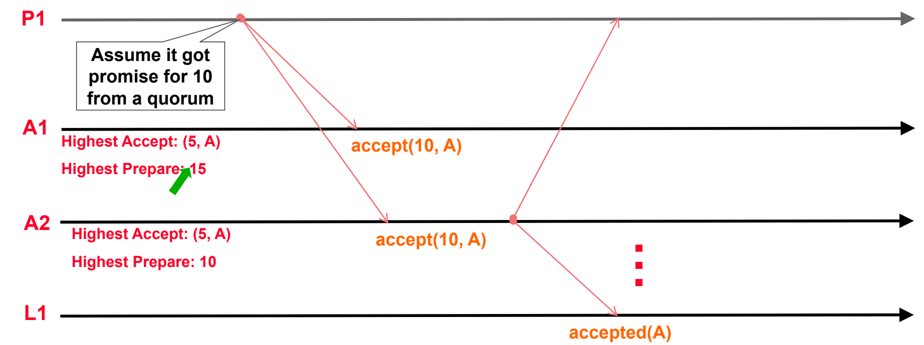
Prepare Example



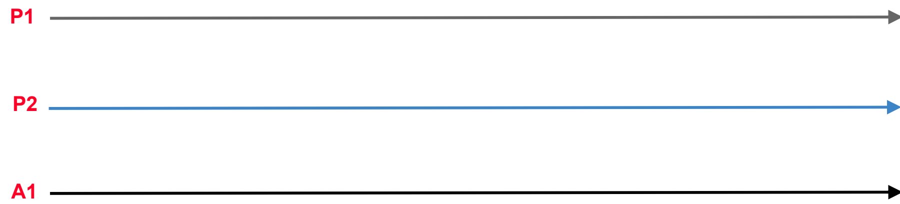
Simple Accept Example



Simple Accept Example



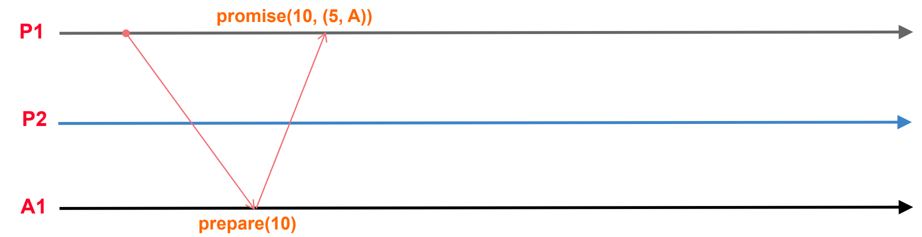
Example: Livelock



A1: Highest accept; (5, A)

Highest prepare: 8

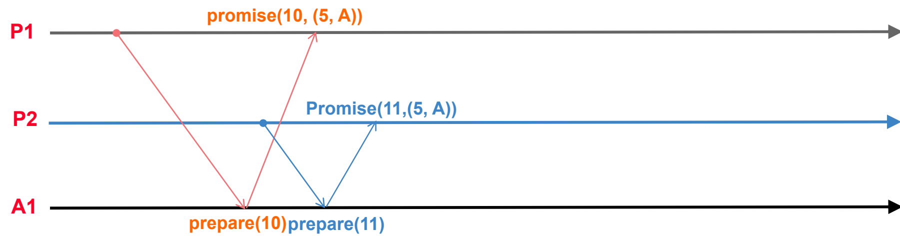
Example: Livelock



A1: Highest accept; (5, A)

Highest prepare: 10

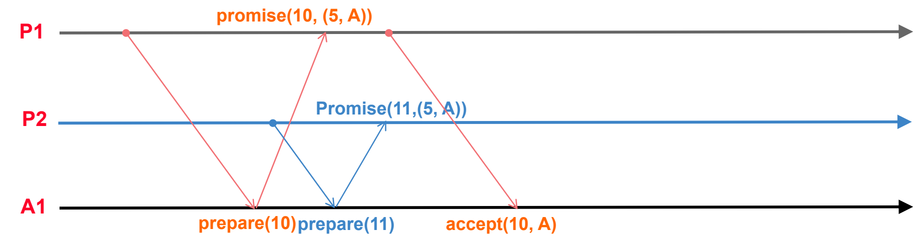
Example: Livelock



A1: Highest accept; (5, A)

Highest prepare: 11

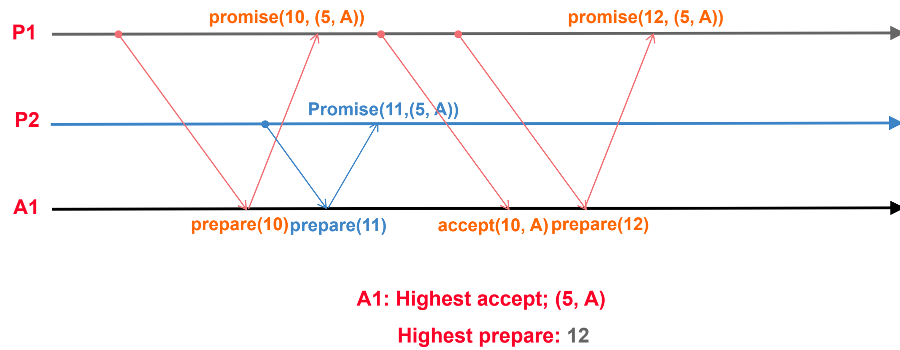
Example: Livelock



A1: Highest accept; (5, A)

Highest prepare: 11

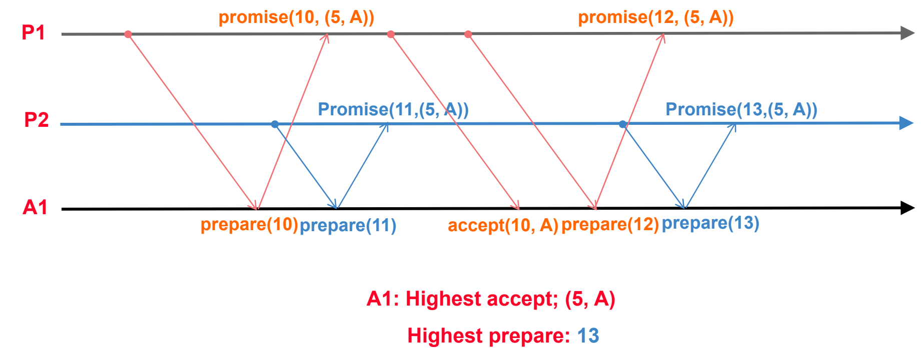
Example: Livelock



A1: Highest accept; (5, A)

Highest prepare: 12

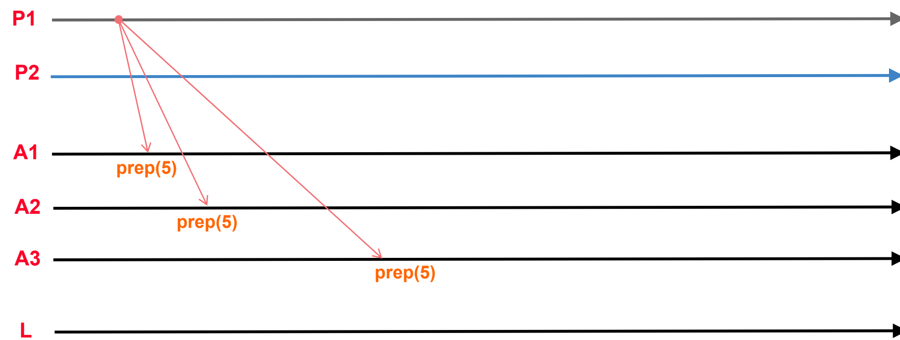
Example: Livelock



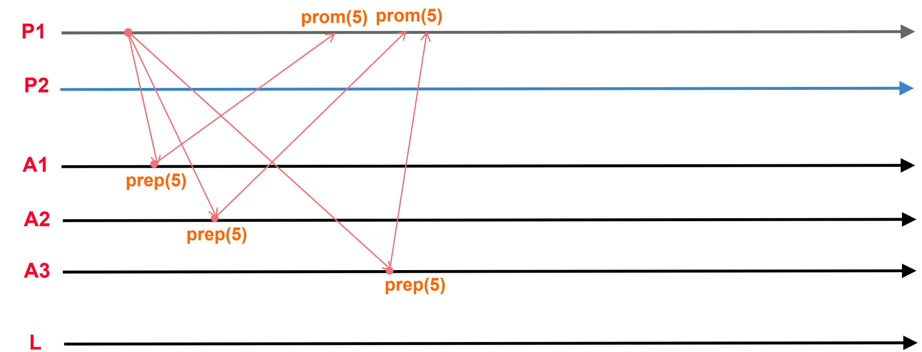
A1: Highest accept; (5, A)

Highest prepare: 13

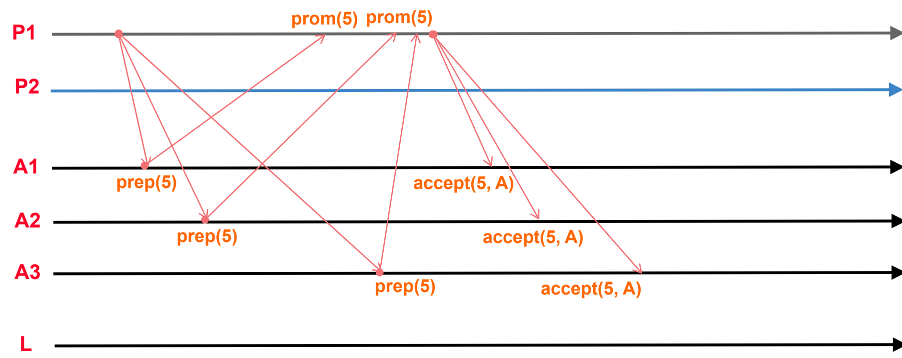
Example: P1 want to propose value A



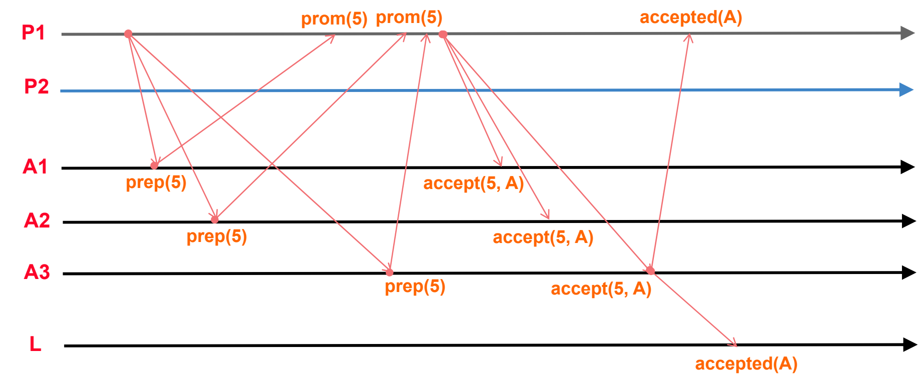
Example: P1 want to propose value A



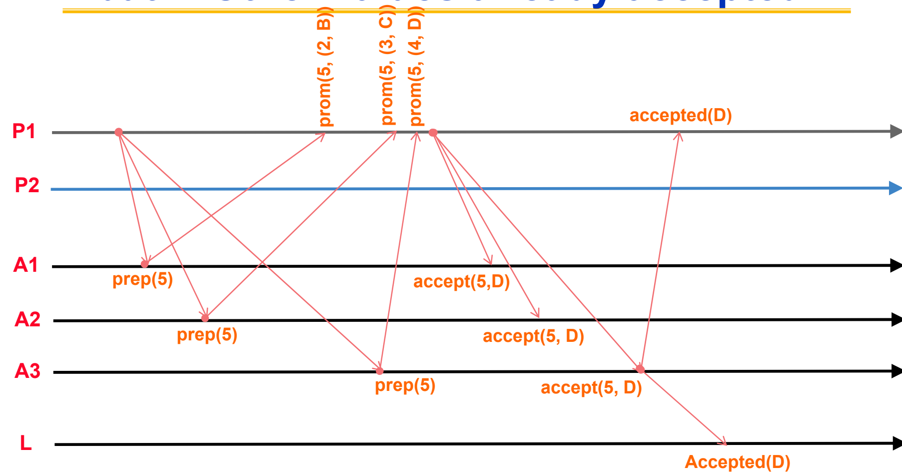
Example: P1 want to propose value A



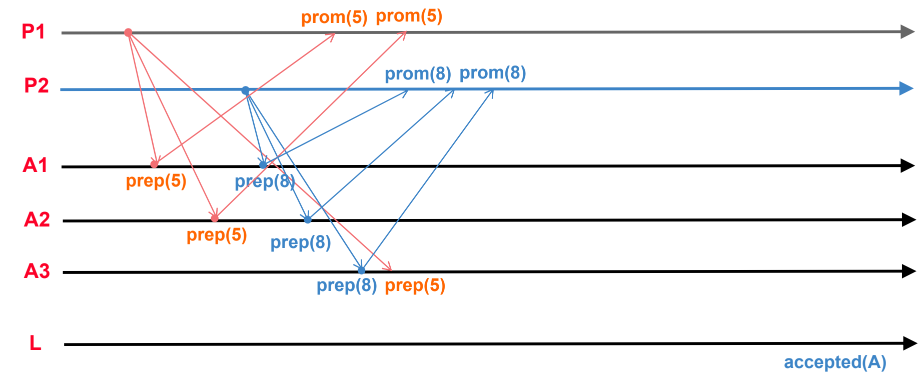
Example: P1 want to propose value A



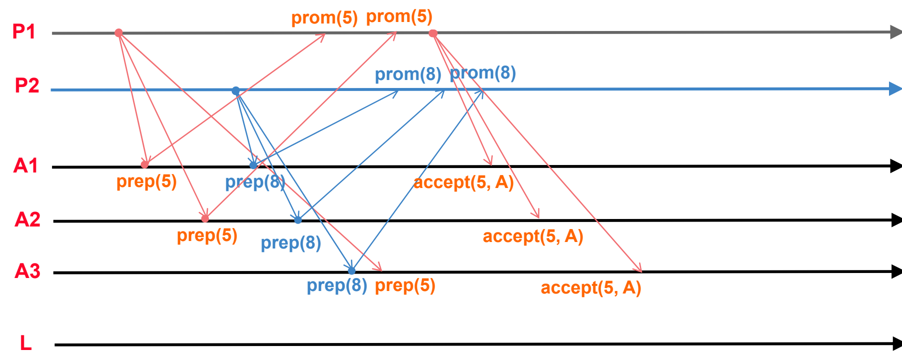
Example: P1 want to propose value A, but… Other values already accepted



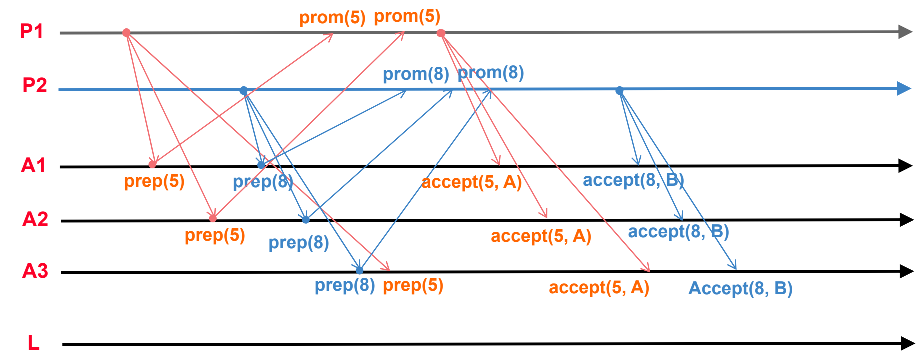
Example: P1 wants A, and P2 wants B



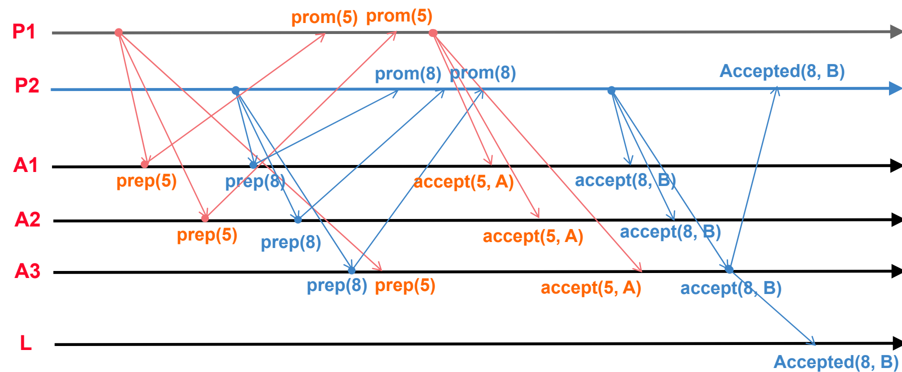
Example: P1 wants A, and P2 wants B



Example: P1 wants A, and P2 wants B



Example: P1 wants A, and P2 wants B



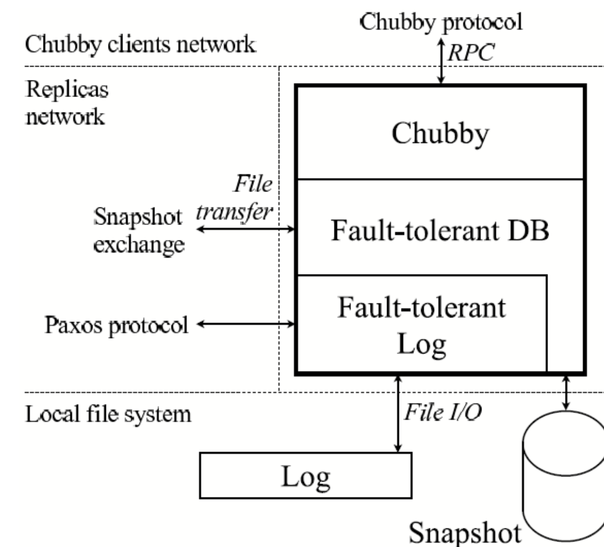
Others

- 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

Paxos in Chubby

- MultiPaxos:
  - Steps 1 (*prepare*) and 2 (*promise*) done once
  - Steps 3 (*accept!*) and 4 (*accepted*) repeated multiple times by same leader
- Replicas in a cell initially use Paxos to establish the leader
  - Majority of replicas must agree
- Optimization: Master Lease
  - Replicas promise not to try to elect new master for at least a few seconds
  - Master lease is periodically renewed
- Master failure
  - If replicas lose contact with master, they wait for grace period (4-6 secs)
  - On timeout, hold new election

Architecture



From Theory to Practice: Fault-tolerant LOG implement with Paxos

- Disk Corruption
  - Need to recognize and handle subtle corruption in stable state
- Use of Master Leases
  - Grant leadership for fixed period of time
  - Allows clients to read latest value from Master
  - Prevents inefficient oscillation in algorithm
- Use of Epochs
  - Recognize when leader changes
  - Chubby semantics requires abort in these circumstances
- Group membership
  - Use of Paxos protocol to select servers that are members of Paxos group
- Snapshot integration with Paxos
- MultiOp
  - Allows implementation of atomic operations on log
  - If (guard[database]) then {t_op} else {f_op}

Building a Correct System

- Simple one-page pseudocode for Paxos algorithm == thousands of lines of C++ code
  - Created simple state machine specification language and compiler
  - Resulting code is "Correct by construction"
- Aggressive testing strategy
  - Tests for *safety* (consistent) and *liveness* (consistent and making progress)
  - Added entry points for test harnesses
  - Reproducible simulation environment
    - » Injection of random errors in network and hardware
    - » Use of pseudo-random seed provided reproducibility
- Data structure and database corruption
  - Aggressive, liberal usage of `assert` statements (makes Chubby fail-stop)
  - Added lots of checksum checks
- Upgrades and rollbacks are hard
  - Fix buggy scripts!
  - Recognize differences between developers and operators
- Forced to "add concurrency" as project progressed

Reliability

- Started out using replicated Berkeley DB (“3DB”)
  - Ill-defined, unproven, buggy replication protocol
- Replaced with custom write-thru logging DB
- Entire database periodically sent to GFS
  - In a different data center
- Chubby replicas span multiple racks

Summary

- Simple protocols win again (?!)
- Reuse of functionality
  - Chubby uses GFS
  - GFS uses Chubby
- Many challenges going from theoretical algorithm to practical implementation
  - No tools for implementing fault-tolerant protocols
  - Test, test, and test again (critical component!)
  - Everything can be corrupted so checksum everything
  - People are fallible (so are scripts!)

Is this a good paper?

- What were the authors’ goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the “Test of Time” challenge?
- How would you review this paper today?

BREAK

Raft

Many slides from Diego Ongaro & John Ousterhout presentation:

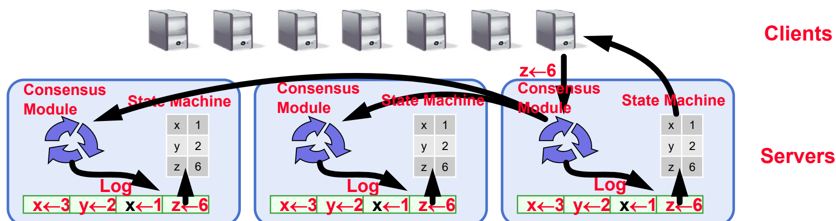
(http://www2.cs.uh.edu/~paris/6360/PowerPoint/Raft.ppt)

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 ⇒ 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

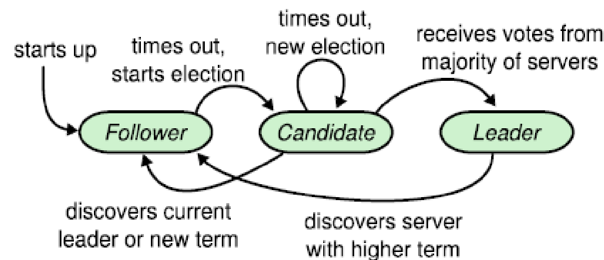
2. Log replication (normal operation)
   – Leader takes commands from clients, appends them to its log
   – Leader replicates its log to other servers (overwriting inconsistencies)

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

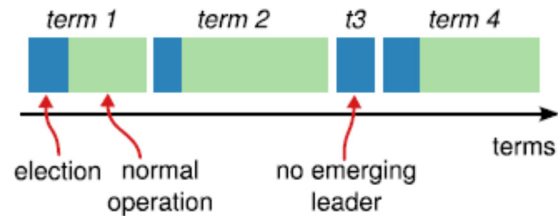
Server states



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)

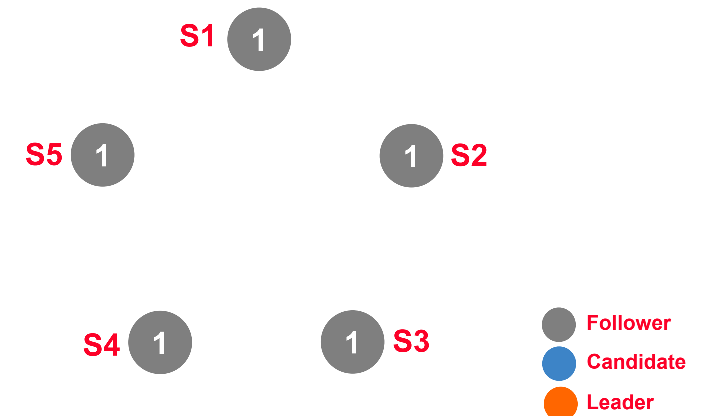


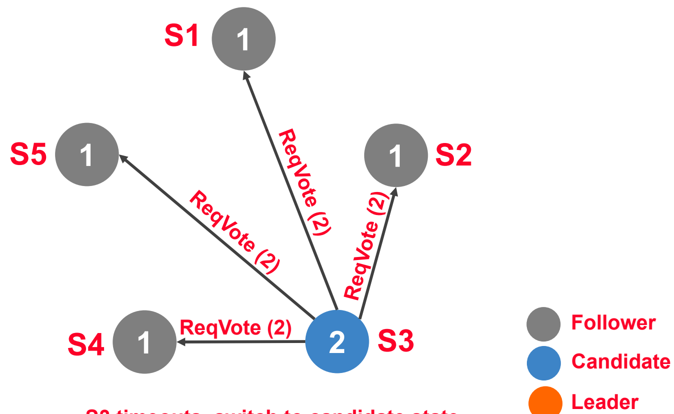
Raft basics: RPC

- Servers communicate though idempotent RPCs
- **RequestVote**
  - Initiated by candidates during elections
- **AppendEntry:** Initiated by leaders to
  - Replicate log entries
  - Provide a form of heartbeat
    - » Empty AppendEntry( ) calls

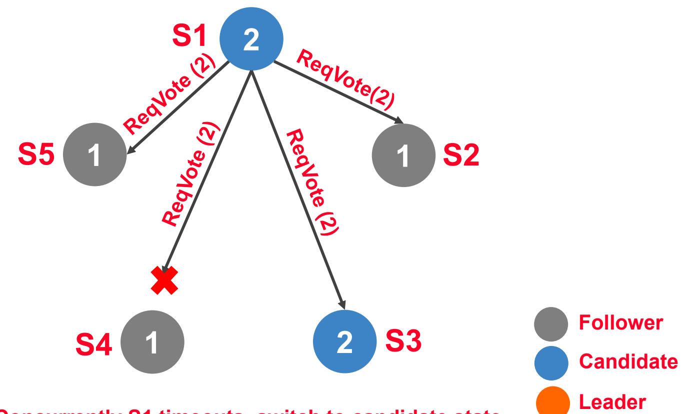
Leader elections

- 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***

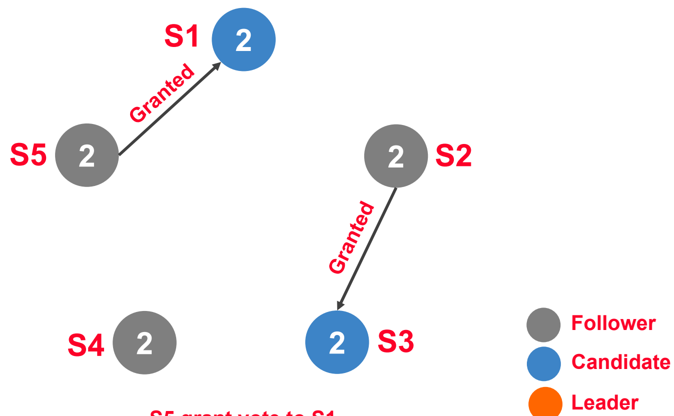




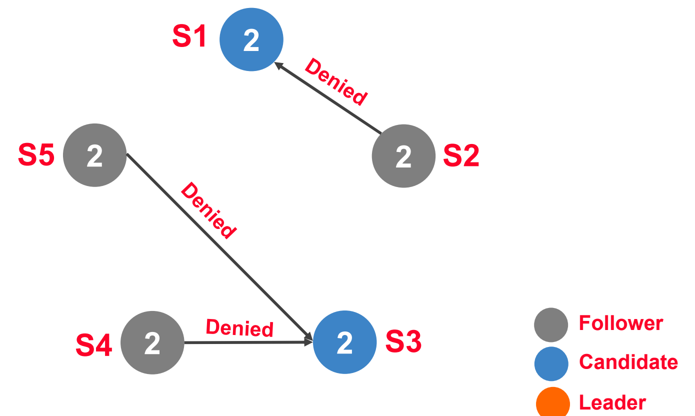
S3 timeouts, switch to candidate state,
increment term, vote itself as a leader and ask everyone else to confirm

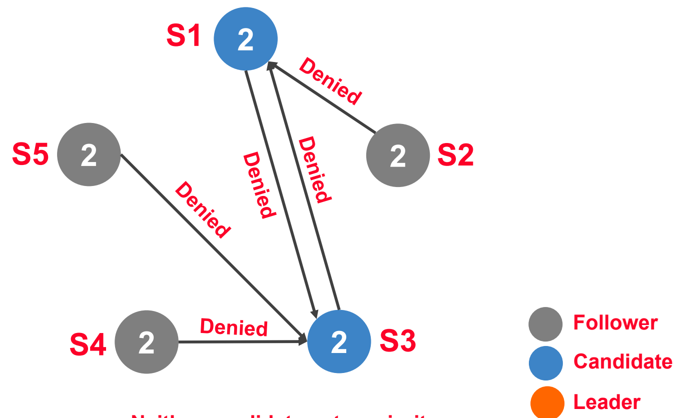


Concurrently S1 timeouts, switch to candidate state,
increment term, vote itself as a leader and ask everyone else to confirm



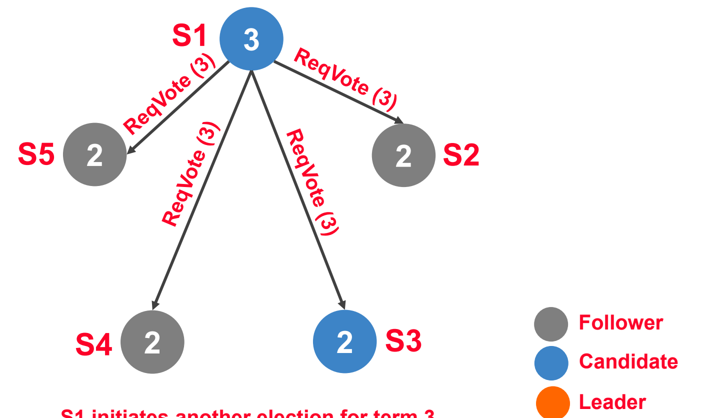
S5 grant vote to S1
S2 grants vote to S3



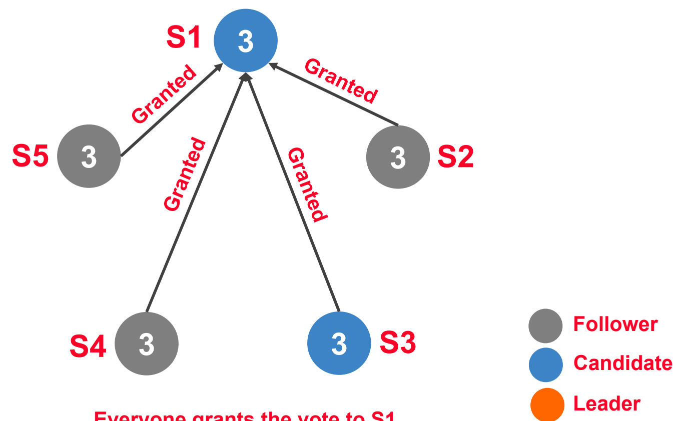


Neither candidate gets majority.

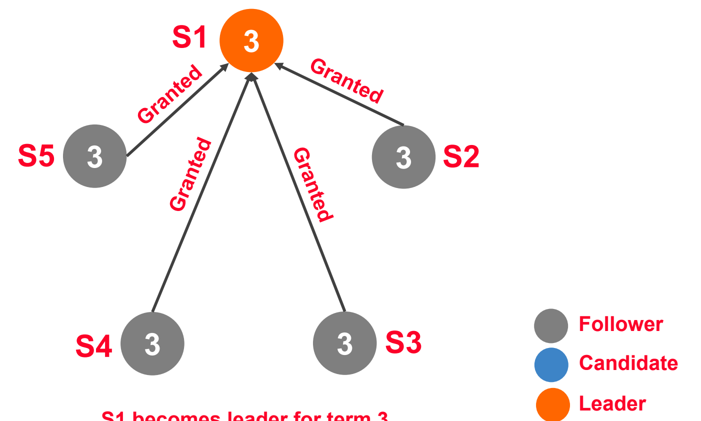
After a random delay between 150-300ms try again.



S1 initiates another election for term 3.



Everyone grants the vote to S1



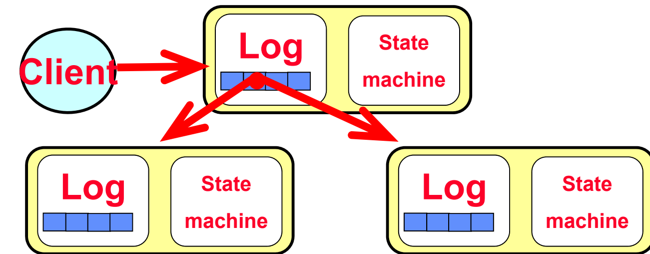
S1 becomes leader for term 3,

and the others become followers.

Log replication

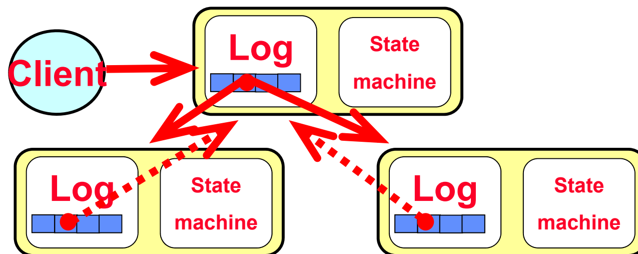
- Leaders
  - Accept client commands
  - Append them to their log (new entry)
  - Issue **AppendEntry** RPCs in parallel to all followers
  - Apply the entry to their state machine once it has been safely replicated
    - » Entry is then ***committed***

A client sends a request



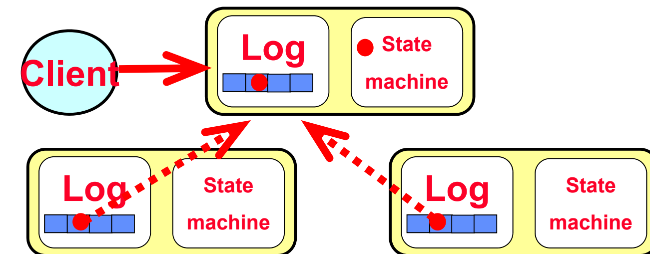
- Leader stores request on its log and forwards it to its followers

The followers receive the request



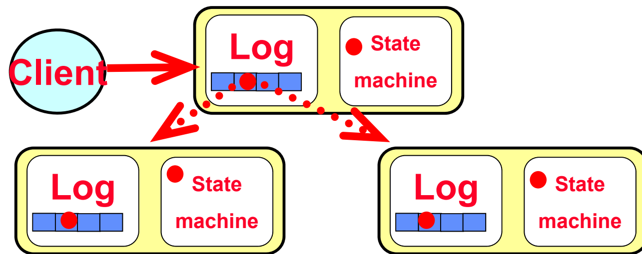
- Followers store the request on their logs and acknowledge its receipt

The leader tallies followers' ACKs



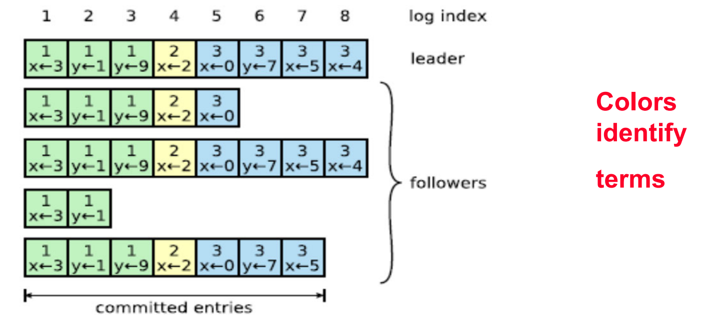
- Once it ascertains the request has been processed by a majority of the servers, it updates its state machine

The leader tallies followers' ACKs



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

Log organization



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

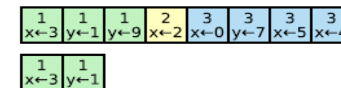
Why?

- 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***



Safety

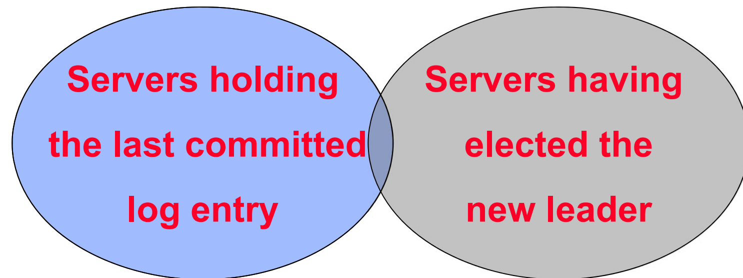
- Two main questions

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)

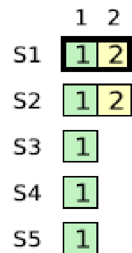


Two majorities of the same cluster *must* intersect

Committing entries from previous term

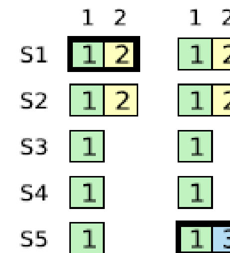
- A leader cannot conclude that an entry from a previous term is committed even if stored on a majority of servers.
- Leader should never commits log entries from previous terms by counting replicas
- Should only do it for entries from the current term
- Once it has been able to do that for one entry, all prior entries are committed indirectly

Committing entries from previous term



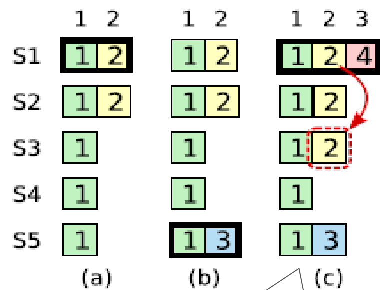
S1 is leader and partially replicates the log entry at index 2.

Committing entries from previous term



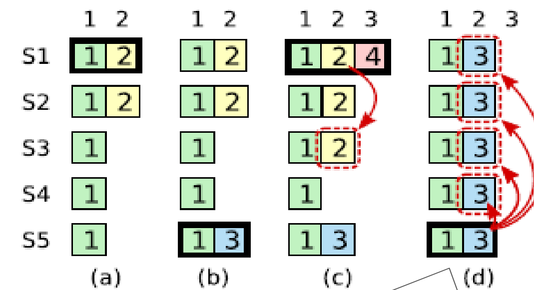
S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2.

Committing entries from previous term



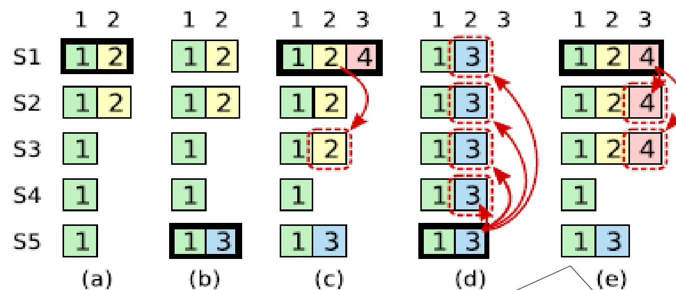
S5 crashes; S1 restarts, is elected leader, and continues replication

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

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?