In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout

Papers We Love (pwl) meetup Zurich
Animesh Trivedi

Why I have chosen Raft?

Claim to fame: Understandable (not necessarily easy!) [consensus] algorithm

Open-source https://raft.github.io/, multiple implementations

- C, C++, Java, Go, Rust, Scala, Haskell, Python, PHP, JavaScript, Dash, D, Ruby, more?

Taught at various universities

- Stanford, Berkeley, Princeton, UWash, Brown, MIT, Harvard, Duke, IIT and more

Used in production code

- etcd, CockroachDB, InfluxDB, Apache Kudu, Apache Ratis

In Search of an Understandable Consensus Algorithm*

2nd

1st

Diego Ongaro and John Ousterhout

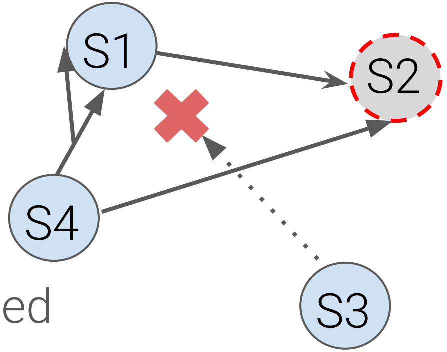
Papers We Love (pwl) meetup Zurich
Animesh Trivedi

*The thesis version

What is a Consensus

Reaching an agreement in a distributed setting when (this is the failure model):

- There are no global clocks
- Machines operate at different speeds
- Machines can fail (fail-stop)
- The network can fail
- Messages can be reordered, dropped, or replicated
- Messages can be take arbitrary long to deliver



A practical solution must ensure that

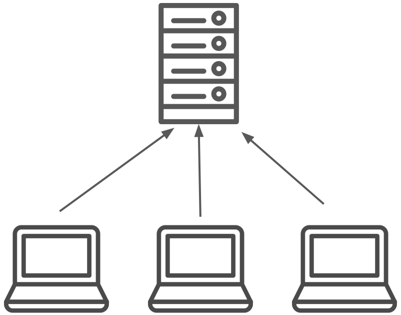
1. Safety: A single proposed value is chosen;
2. Liveness: Some progress is made eventually;

Fischer, Lynch, and Paterson, "*Impossibility of Distributed Consensus with One Faulty Process*", Journal of the ACM (JACM) 1985.
(also known as The FLP Theorem)

Applications of Consensus

building a storage service

key-value store



client machines

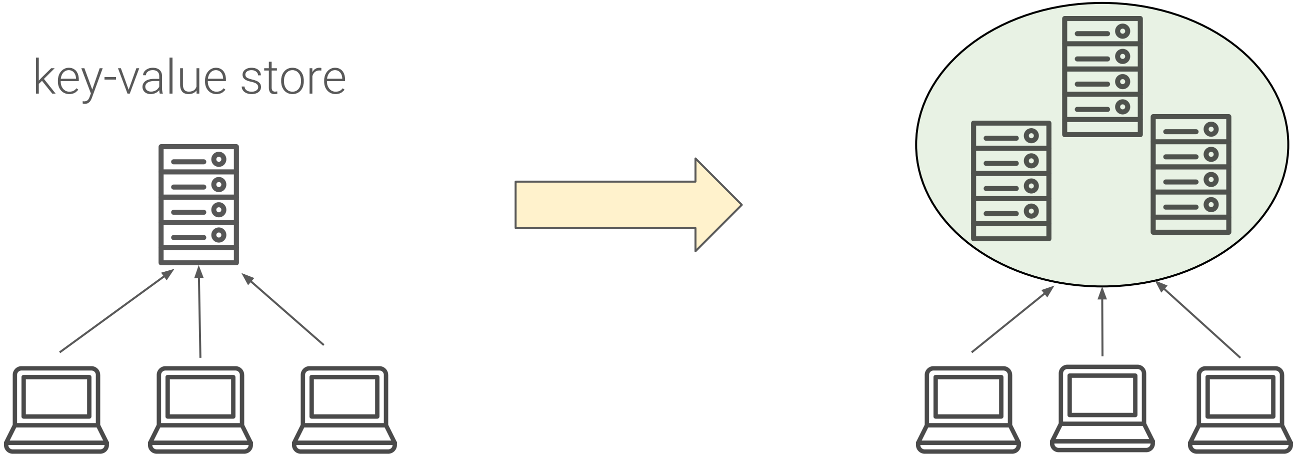
+ Pros: easy, and consistent

- Cons: Failure?

Applications of Consensus

building a highly reliable storage service

key-value store



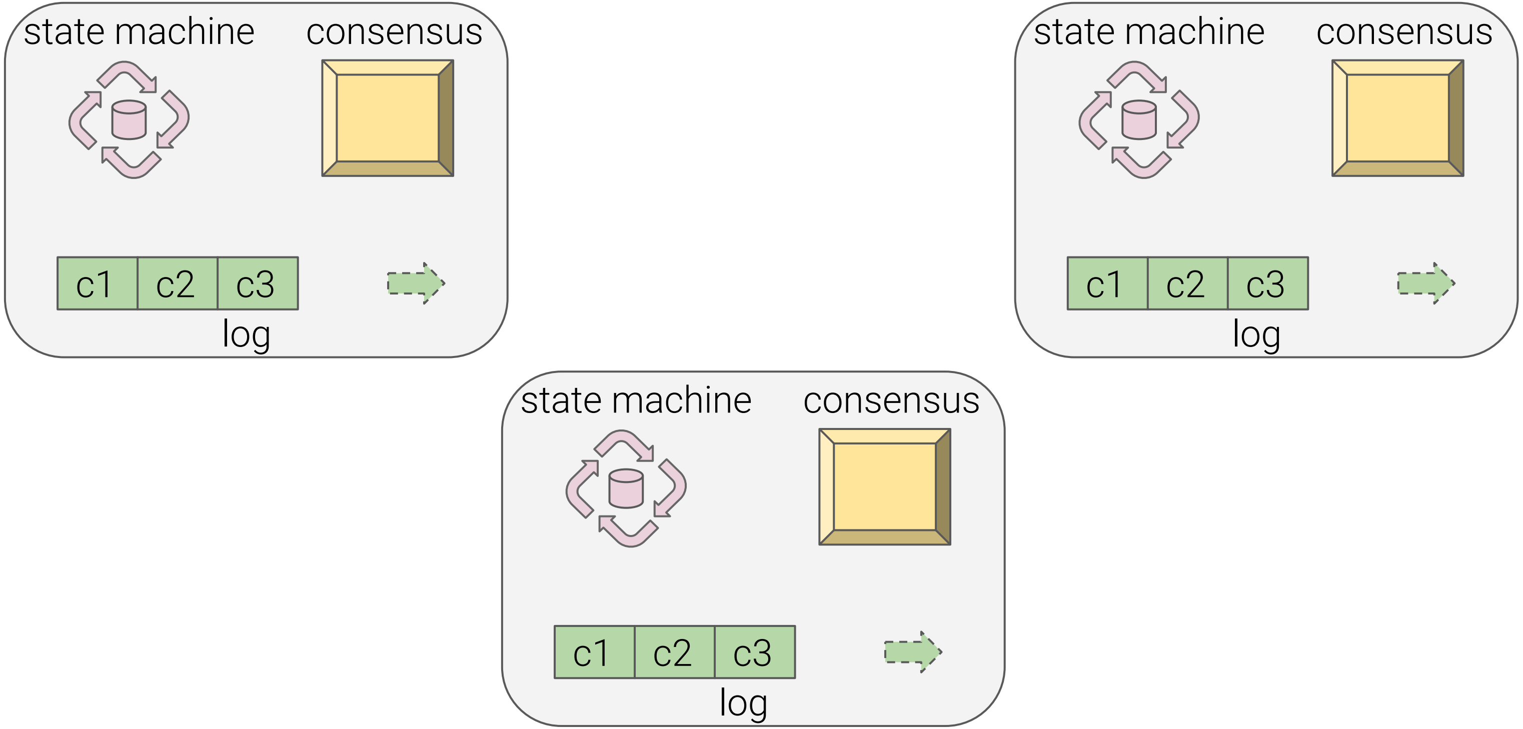
client machines

client machines

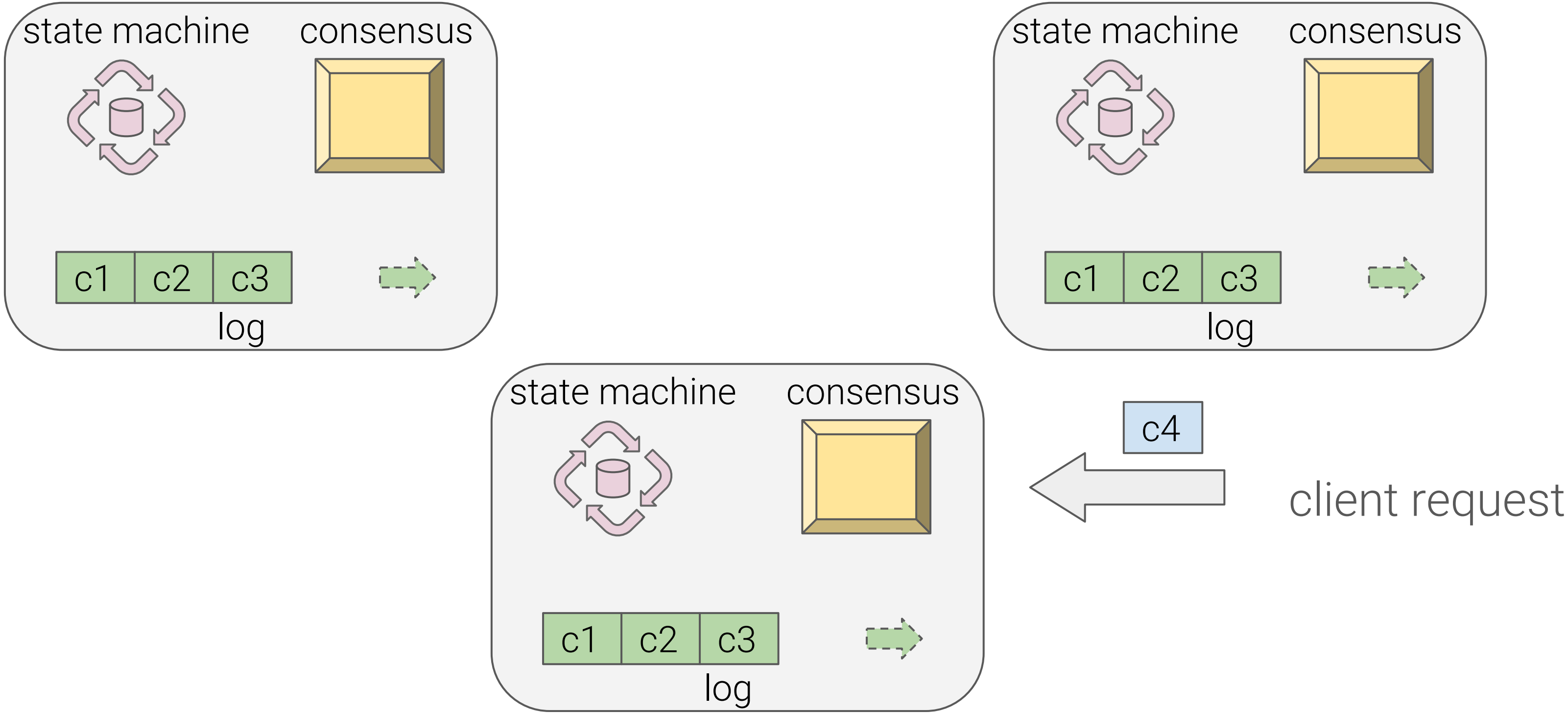
- + Pros: easy, and consistent
- - Cons: Failure?

consensus to reason about
the state of the replicated system

State-Machine Replication using Log



State-Machine Replication using Log



state machine

consensus

c1 c2 c3

log

state machine

consensus

c1 c2 c3

log

state machine

consensus

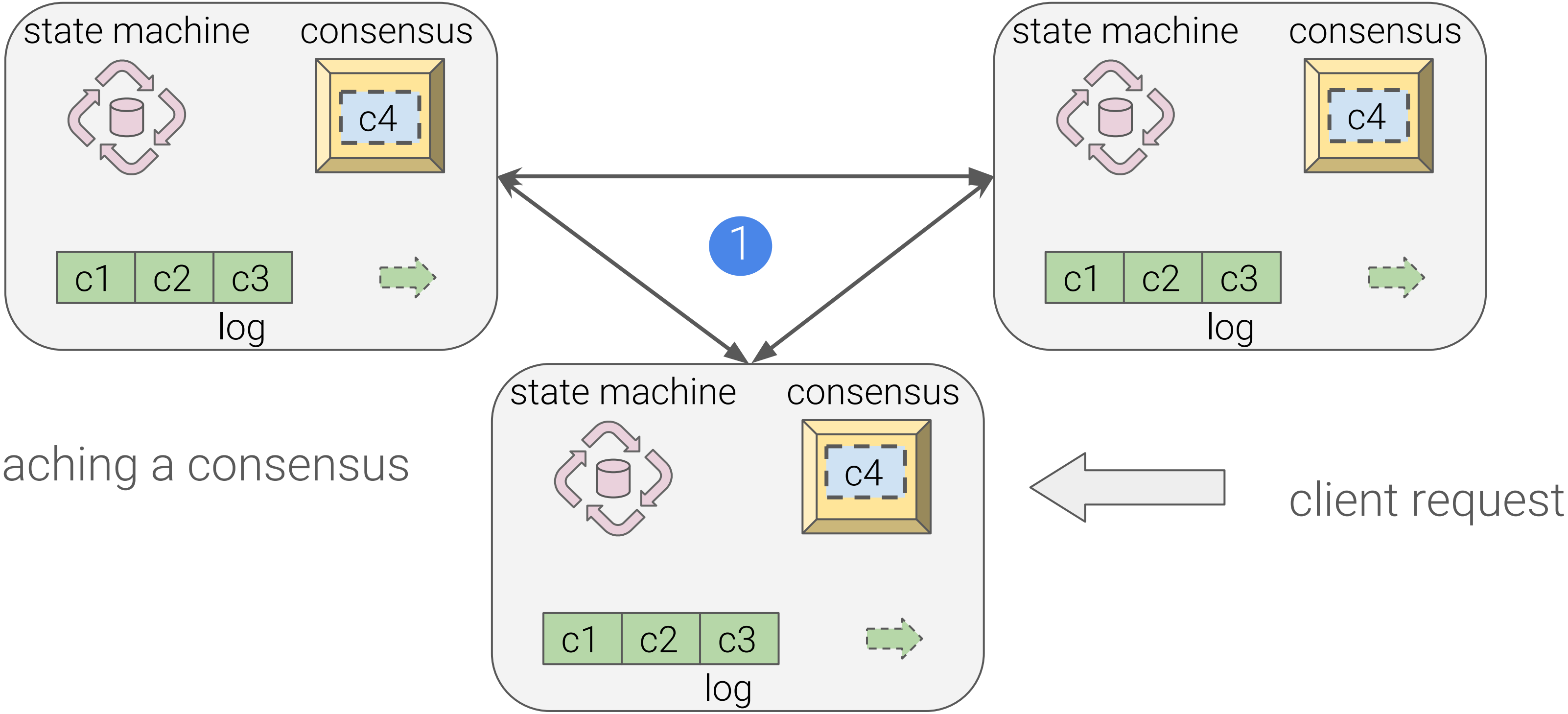
c1 c2 c3

log

c4

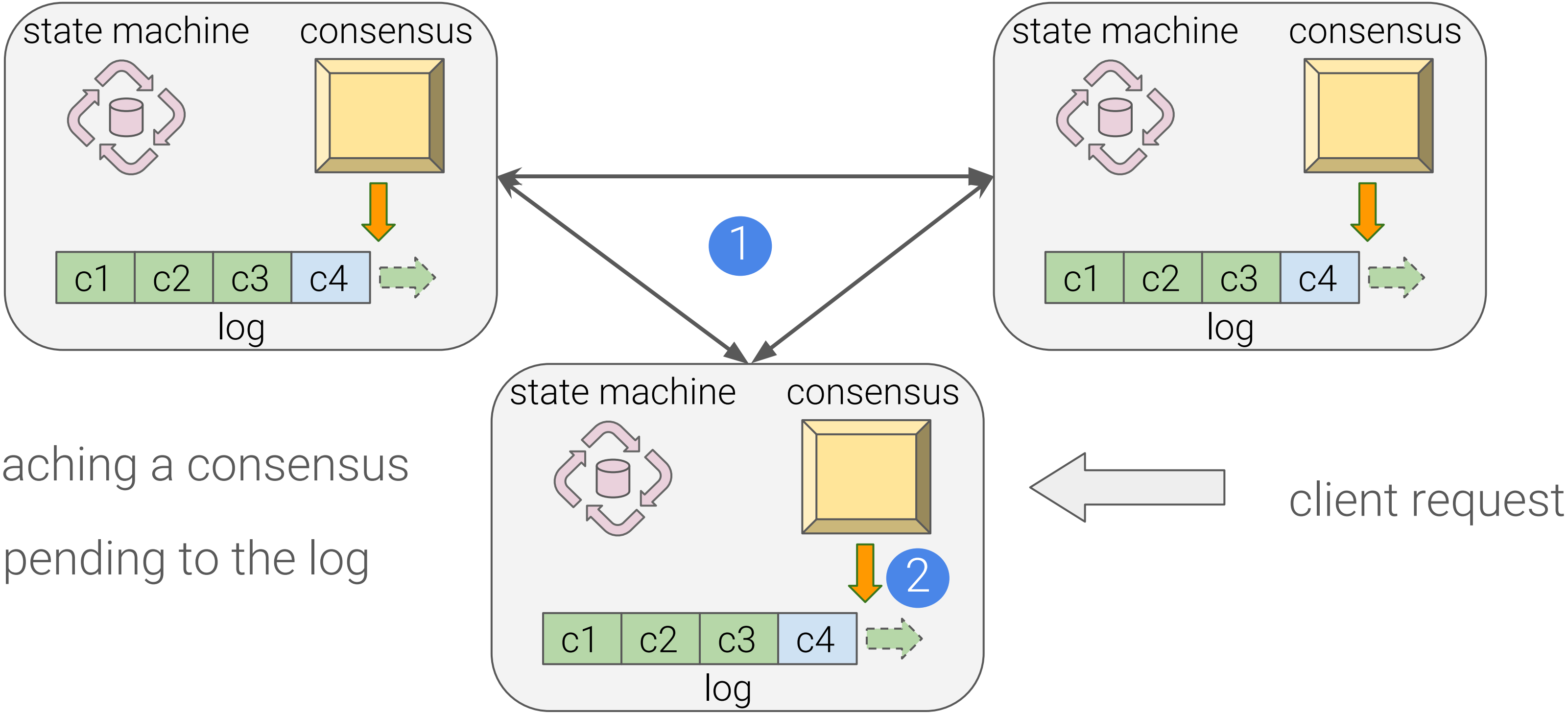
client request

State-Machine Replication using Log



1 Reaching a consensus

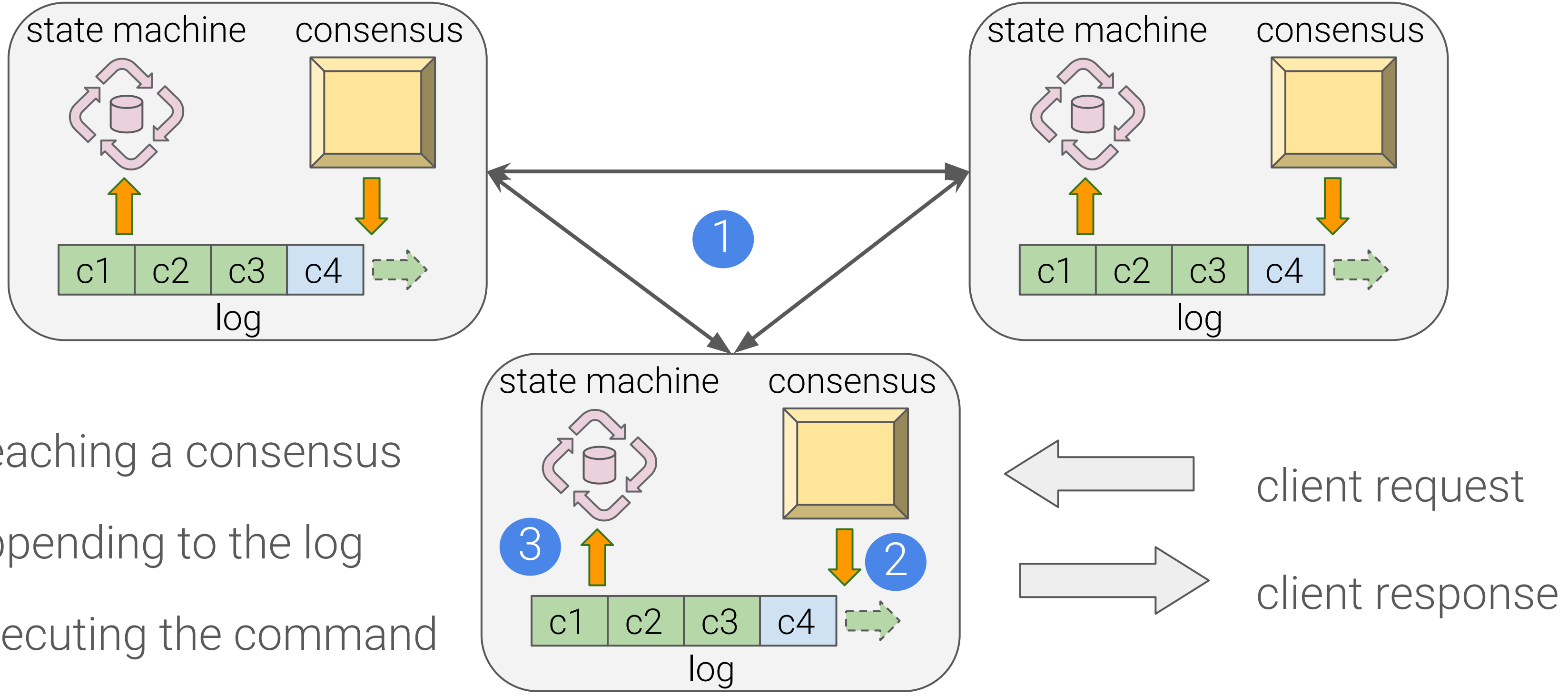
State-Machine Replication using Log



1 Reaching a consensus

2 Appending to the log

State-Machine Replication using Log



1 Reaching a consensus

2 Appending to the log

3 Executing the command

Raft: In a Nutshell

What is Raft? An algorithm for managing a replicated log of decisions

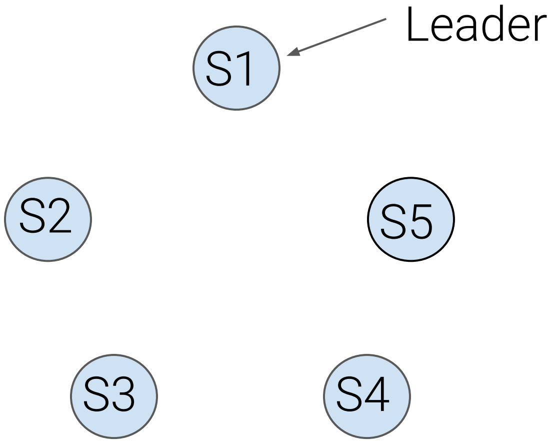
How does it do it? It chooses a leader which makes decisions about the log status and forces other servers to follow its decisions

What is that good for? One can use replicated Raft log to implement the replicated state machine

Is Raft safe? Yes, it is proven to be safe. Its TLA+ specification is available

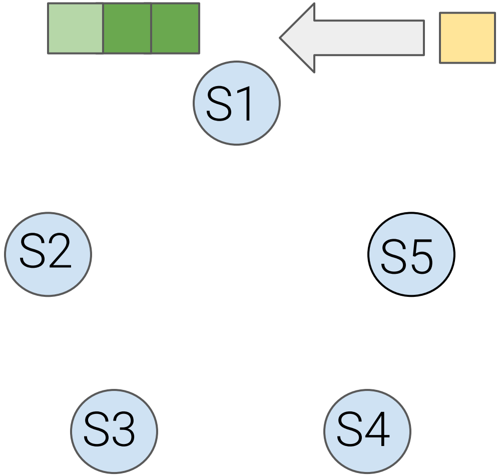
Raft: Basic Commit Operation

(2f +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



Raft: Basic Commit Operation

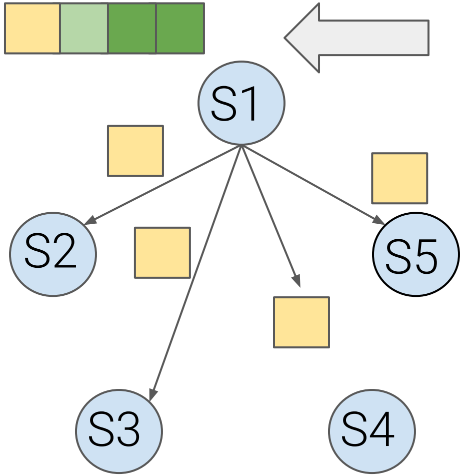
(2f +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



1. Client request comes in

Raft: Basic Commit Operation

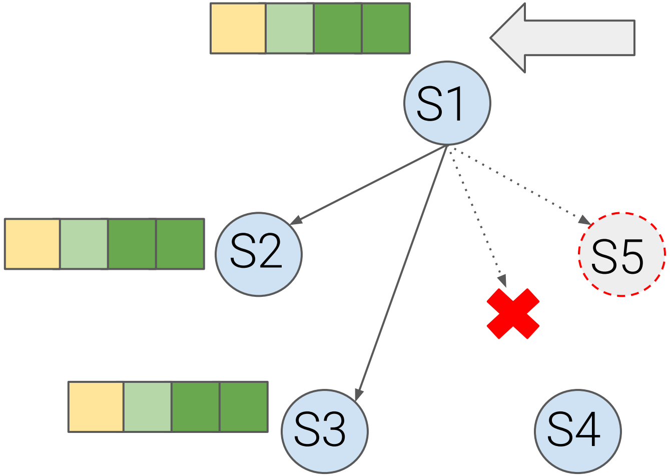
(2f +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



1. Client request comes in
2. The leader applies locally and sends the command out to all servers

Raft: Basic Commit Operation

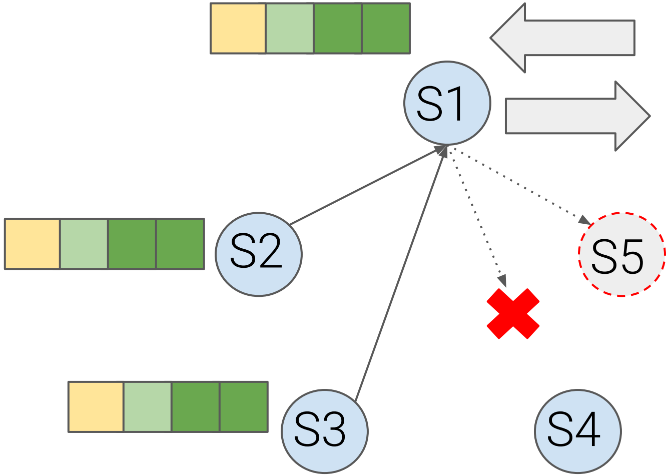
(2f +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



1. Client request comes in
2. The leader applies locally and sends the command out to all servers
3. All servers append the entry in their logs

Raft: Basic Commit Operation

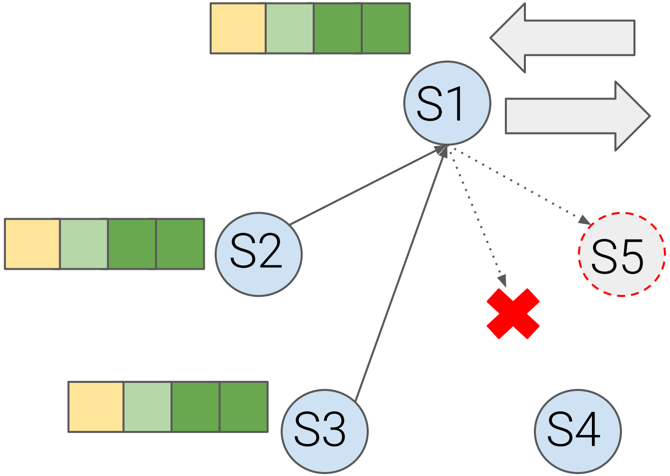
(2f +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



1. Client request comes in
2. The leader applies locally and sends the command out to all servers
3. All servers append the entry in their logs
4. As soon as majority replies the command is committed

Raft: Basic Commit Operation

(2f +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



Issues to solve:

1. What happens when the leader crashes?
   Leader Election

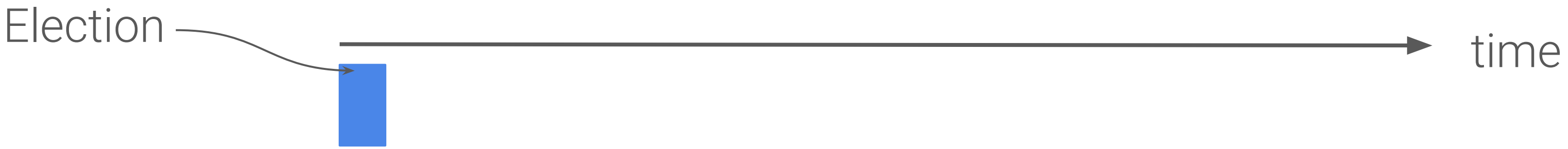
2. How to ensure all servers have the same log entries in sequence?
   Log replication algorithm

Raft Basics

Time is split in *terms* with at most one leader in each term

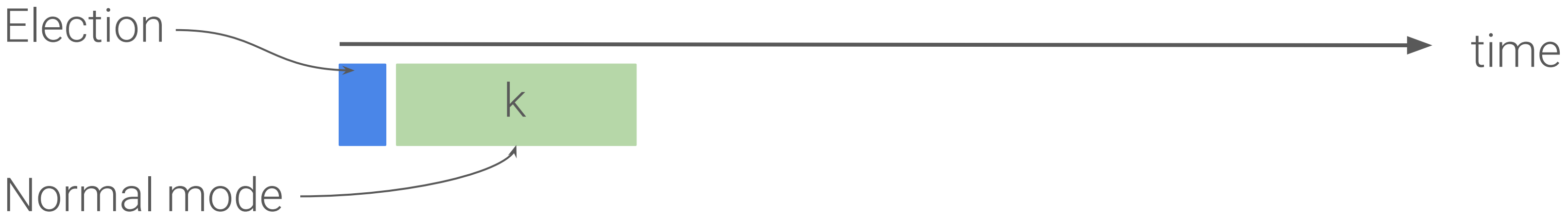
Raft Basics

Time is split in *terms* with at most one leader in each term



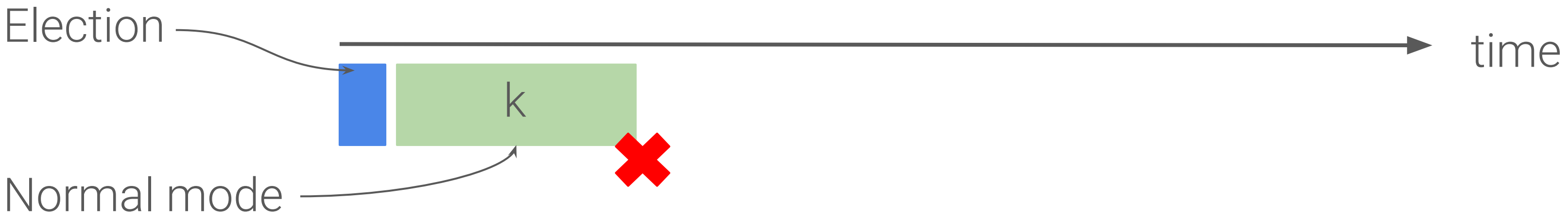
Raft Basics

Time is split in *terms* with at most one leader in each term



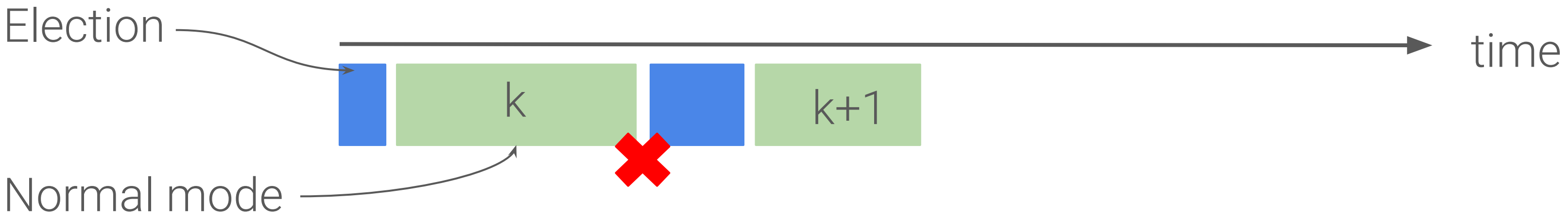
Raft Basics

Time is split in *terms* with at most one leader in each term



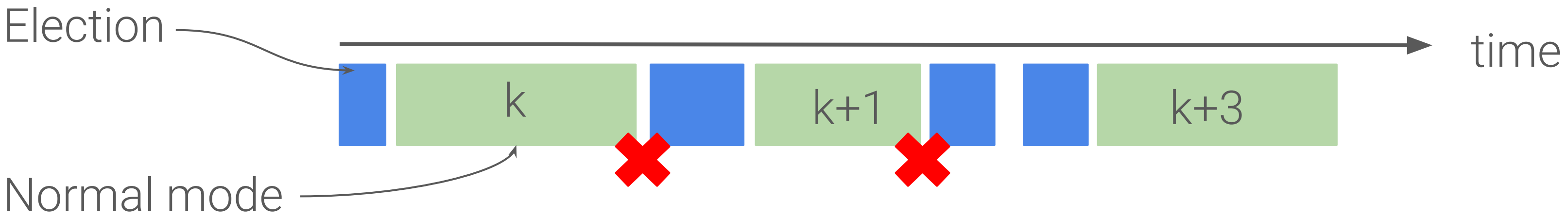
Raft Basics

Time is split in *terms* with at most one leader in each term



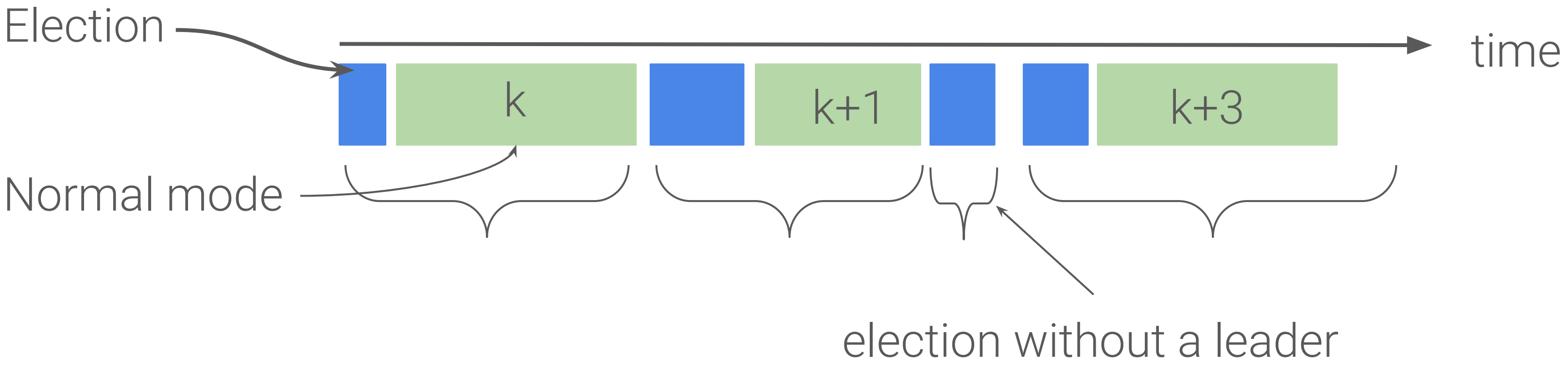
Raft Basics

Time is split in *terms* with at most one leader in each term



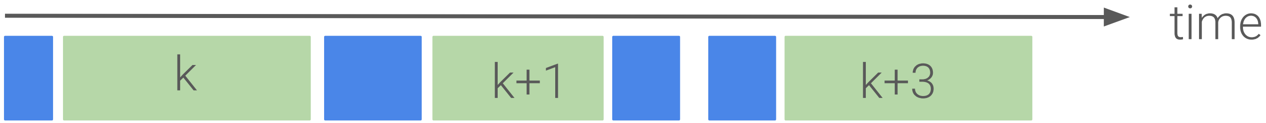
Raft Basics

Time is split in *terms* with at most one leader in each term



Raft Basics

Time is split in *terms* with at most one leader in each term



A server can be: the Leader, a Follower, a Candidate

Follower

only accepts commands from the leader

Candidate

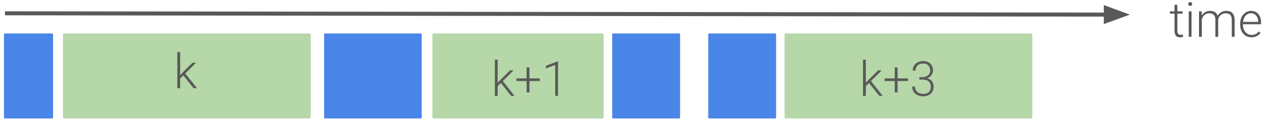
tries to become the leader

Leader

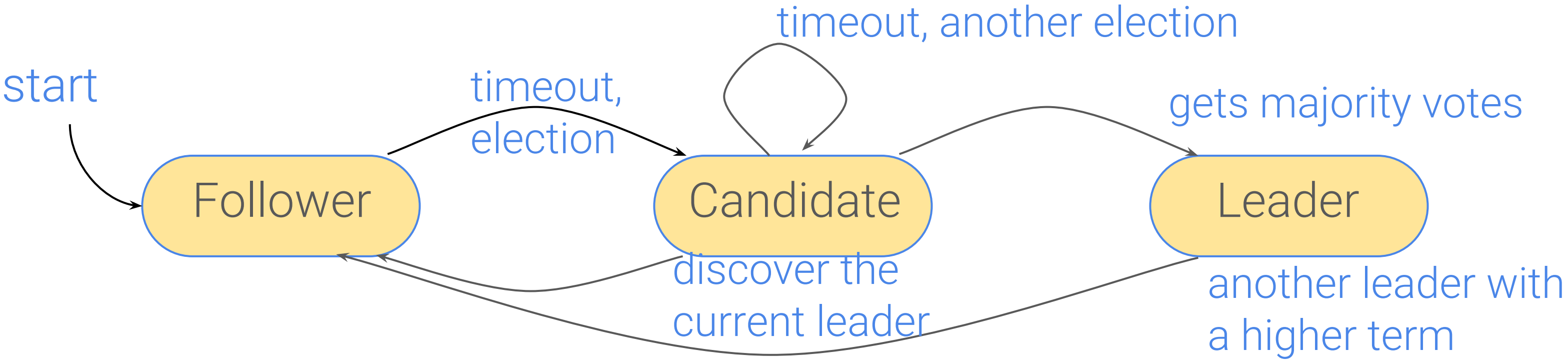
manages client requests and followers

Raft Basics

Time is split in *terms* with at most one leader in each term

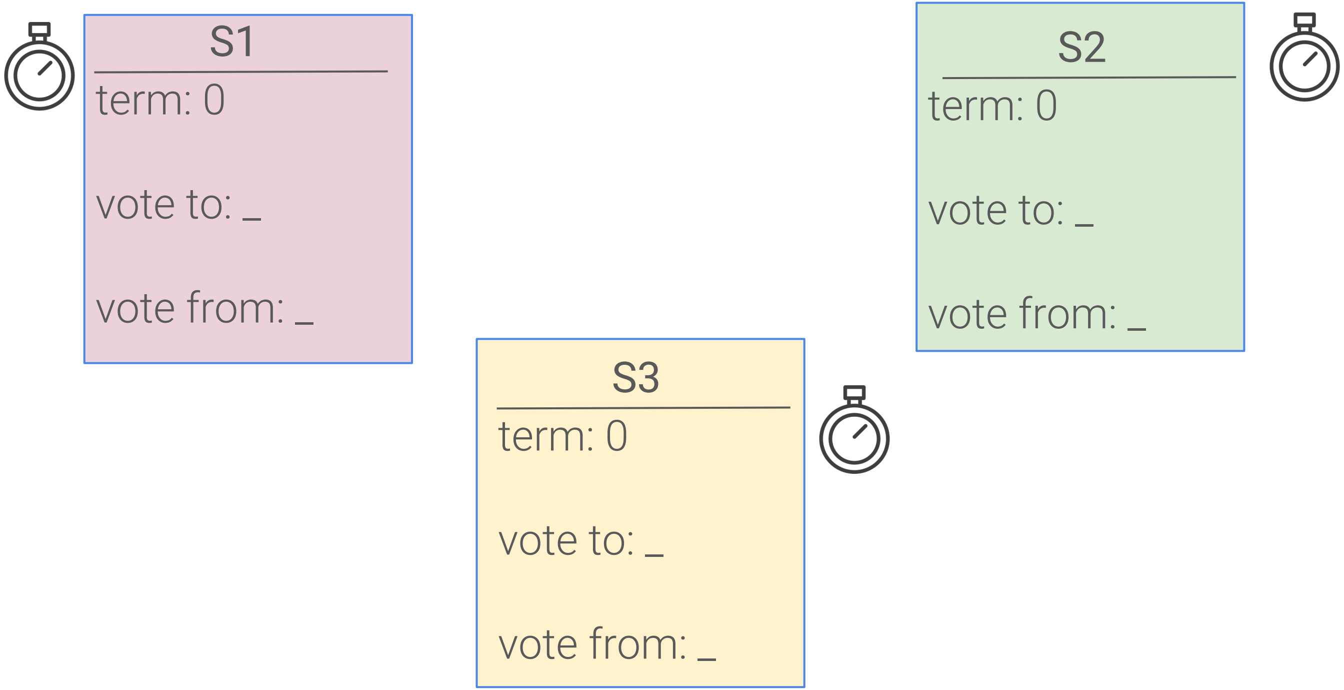


A server can be: the Leader, a Follower, a Candidate



start
Follower
timeout, election
Candidate
timeout, another election
gets majority votes
Leader
discover the current leader
another leader with a higher term

Raft Leader Election



S1

term: 0

vote to: _

vote from: _

S2

term: 0

vote to: _

vote from: _

S3

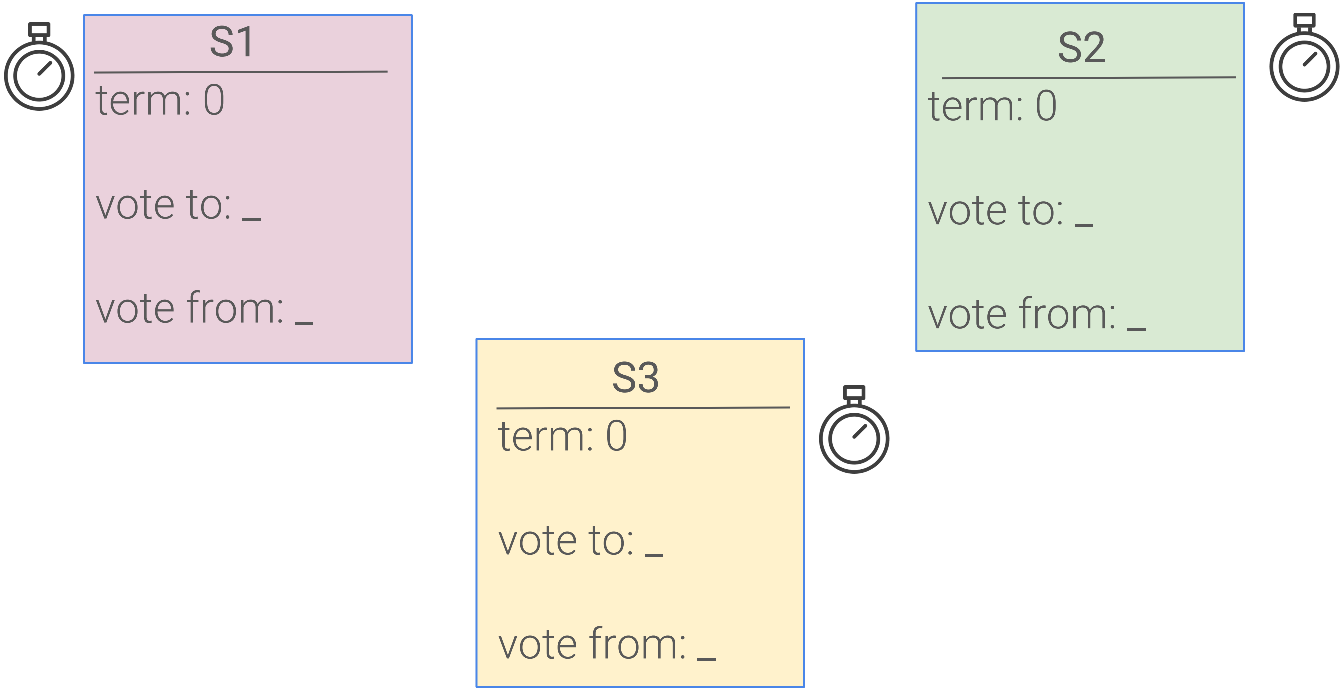
term: 0

vote to: _

vote from: _

Raft Leader Election

Who calls the election? Use randomized timeouts in a range



S1

term: 0

vote to: _

vote from: _

S2

term: 0

vote to: _

vote from: _

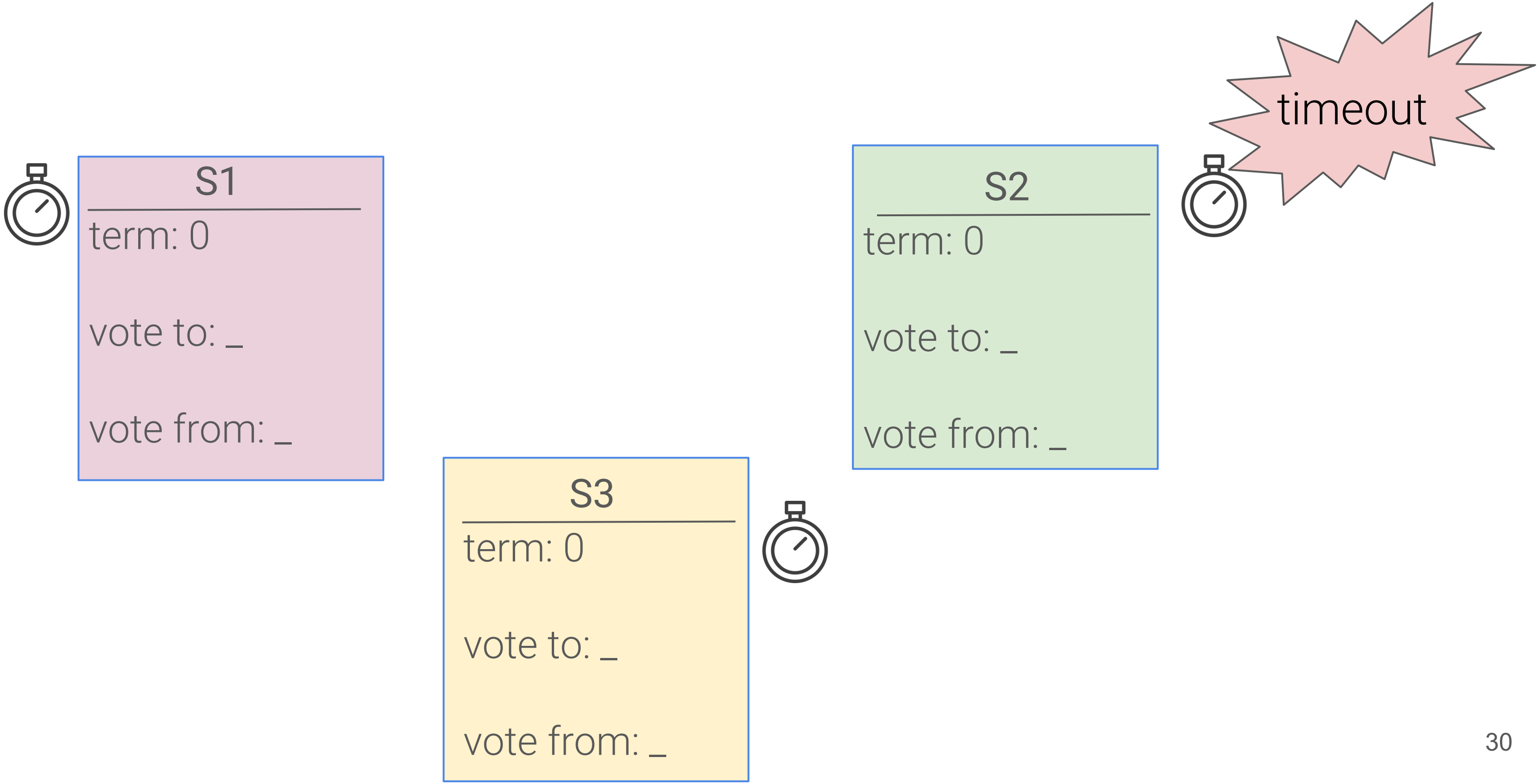
S3

term: 0

vote to: _

vote from: _

Raft Leader Election



timeout

S1

term: 0

vote to: _

vote from: _

S2

term: 0

vote to: _

vote from: _

S3

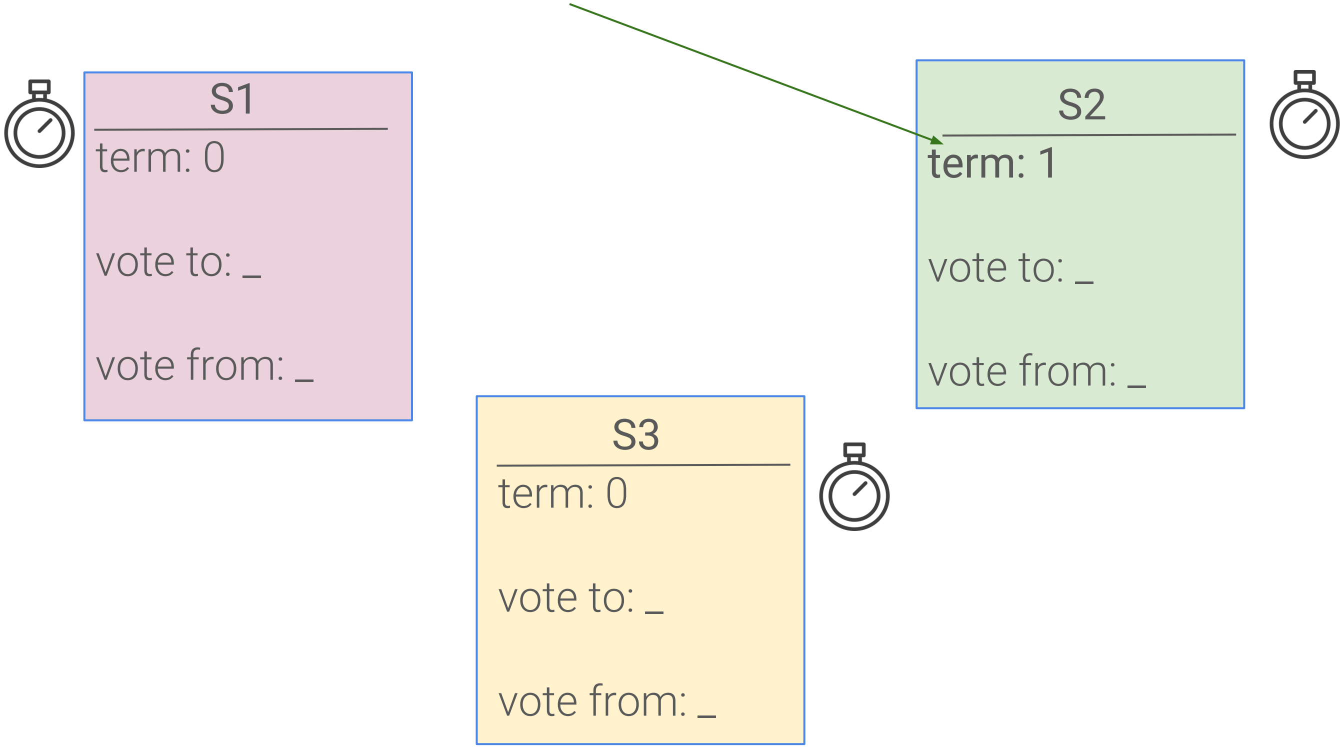
term: 0

vote to: _

vote from: _

Raft Leader Election

Step 1: declare yourself candidate, and increase the term



S1

term: 0

vote to: _

vote from: _

S2

term: 1

vote to: _

vote from: _

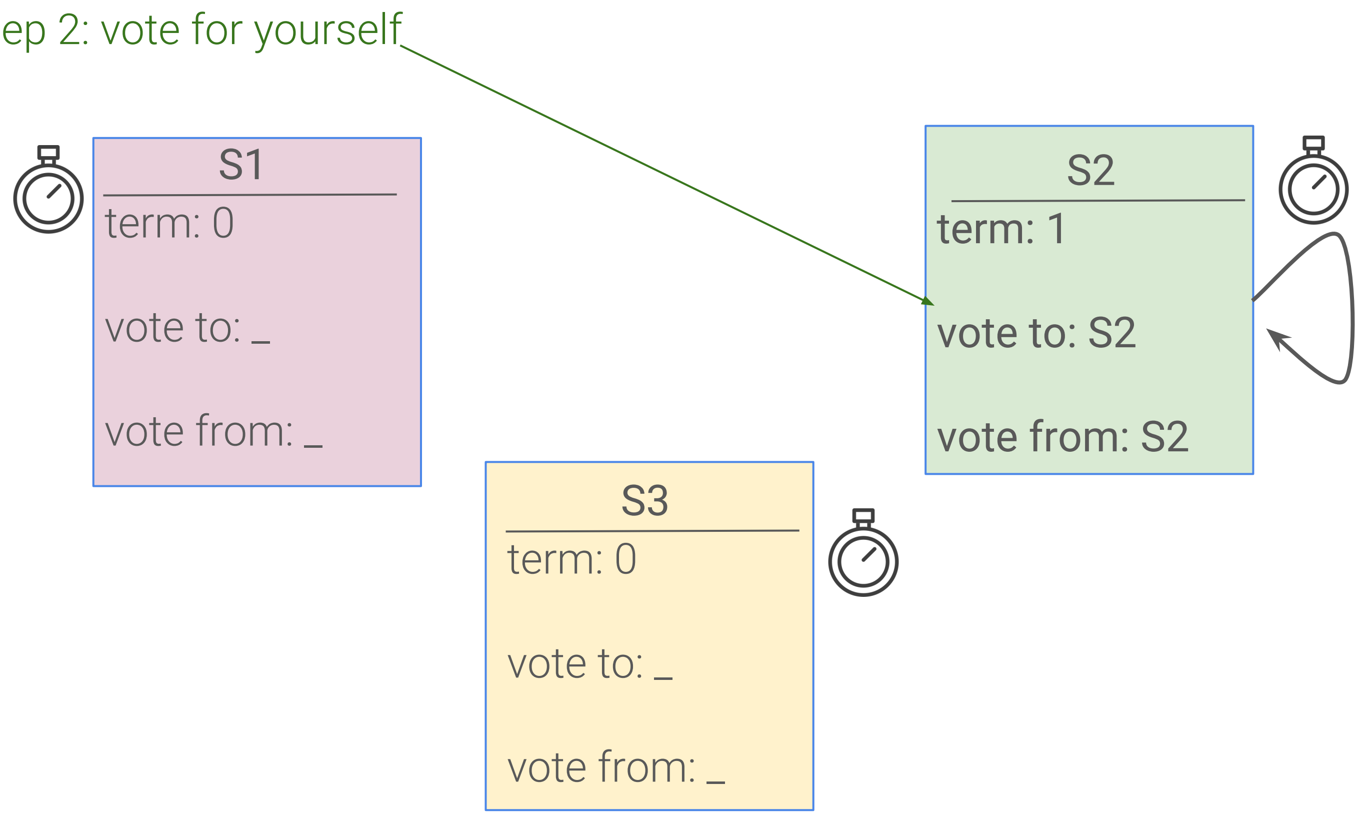
S3

term: 0

vote to: _

vote from: _

Raft Leader Election



Step 2: vote for yourself

S1

term: 0

vote to: _

vote from: _

S2

term: 1

vote to: S2

vote from: S2

S3

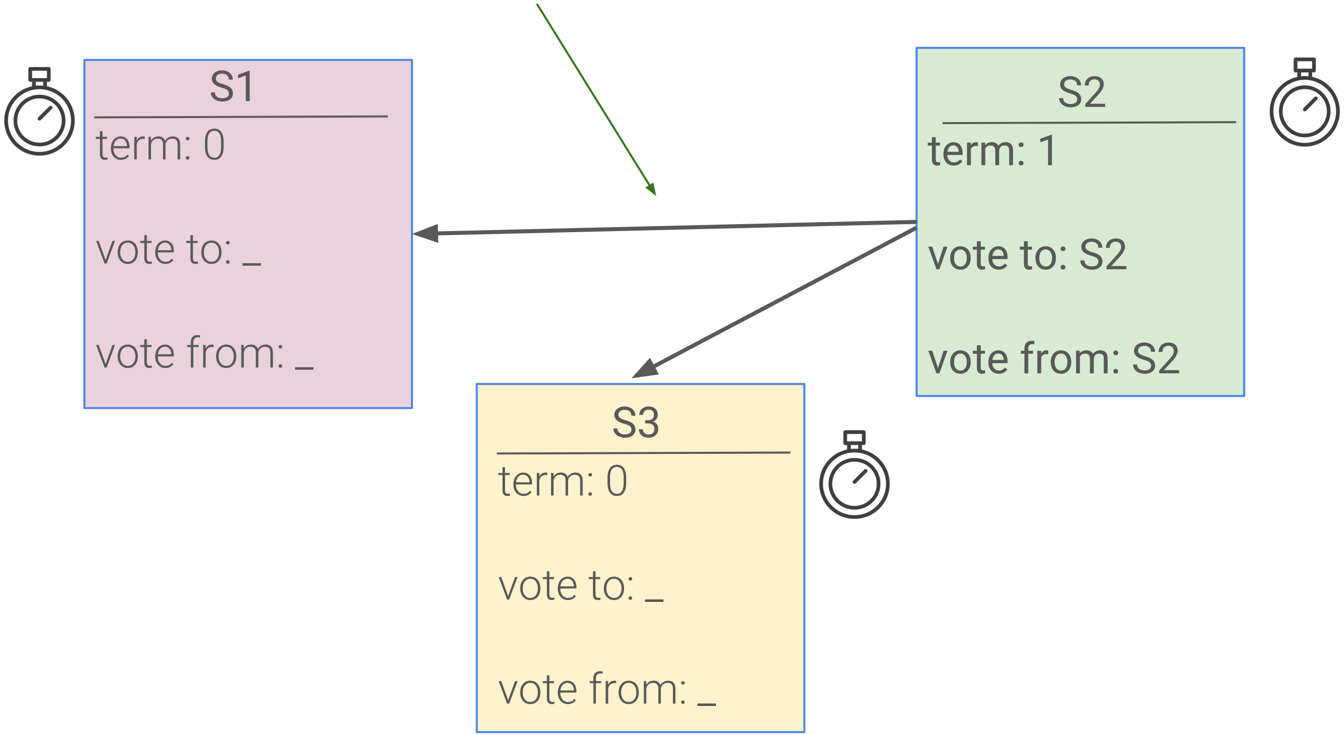
term: 0

vote to: _

vote from: _

Raft Leader Election

Step 3: ask for votes from others, prepare *RequestVoteRPC [term, ID]*



S1

term: 0

vote to: _

vote from: _

S2

term: 1

vote to: S2

vote from: S2

S3

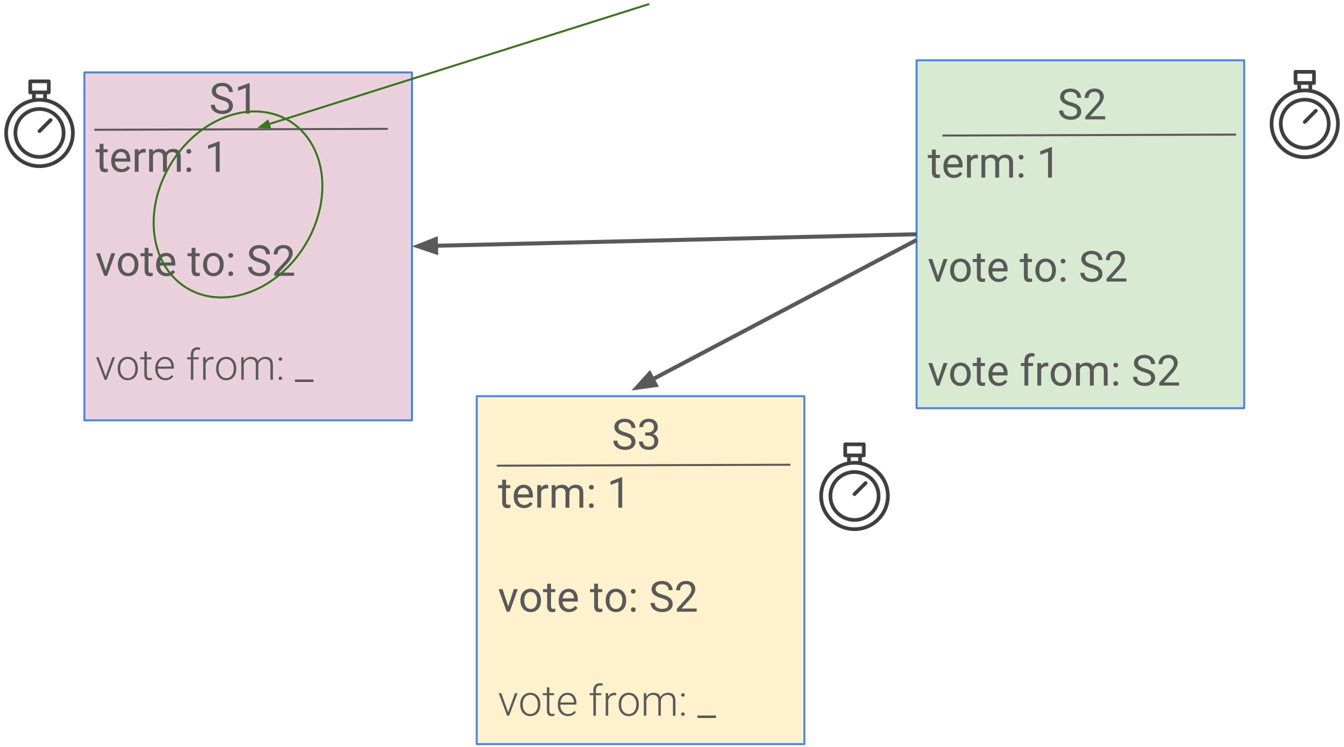
term: 0

vote to: _

vote from: _

Raft Leader Election

Step 4: followers increase their term, and grant vote if they haven't so far



S1

term: 1

vote to: S2

vote from: _

S2

term: 1

vote to: S2

vote from: S2

S3

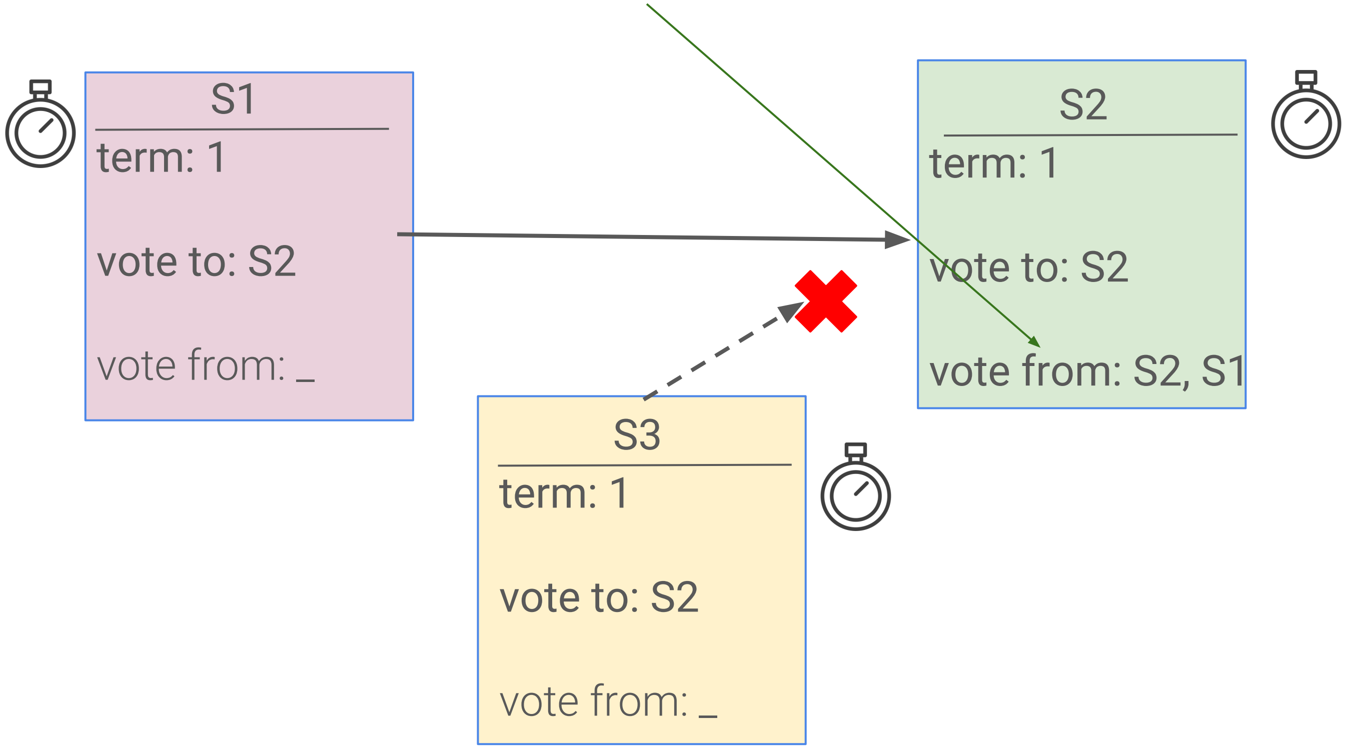
term: 1

vote to: S2

vote from: _

Raft Leader Election

Step 5: check if the majority has voted for you?



S1

term: 1

vote to: S2

vote from: _

S2

term: 1

vote to: S2

vote from: S2, S1

S3

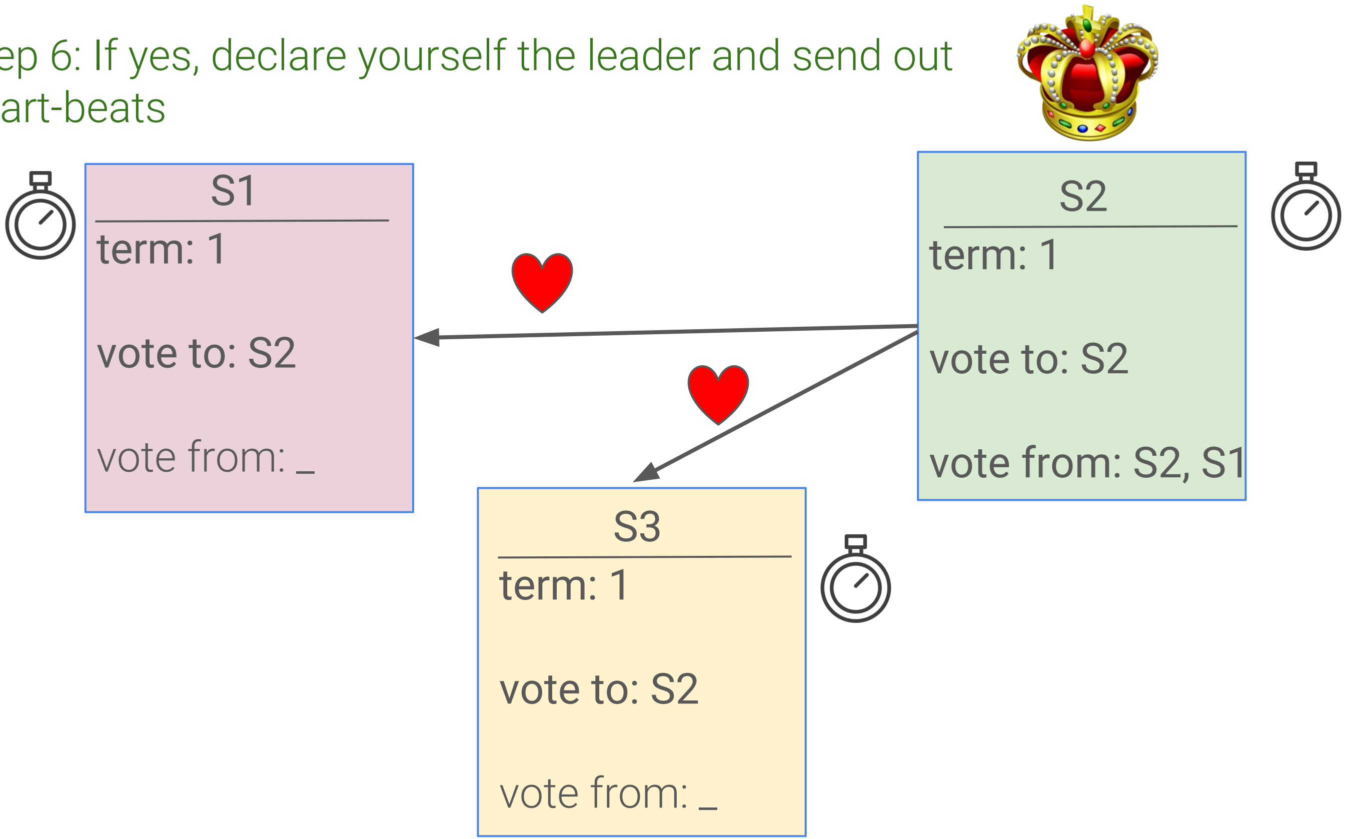
term: 1

vote to: S2

vote from: _

Raft Leader Election

Step 6: If yes, declare yourself the leader and send out heart-beats



S1

term: 1

vote to: S2

vote from: _

S2

term: 1

vote to: S2

vote from: S2, S1

S3

term: 1

vote to: S2

vote from: _

Raft Leader Election

What could go wrong here?

Multiple machines timeout at the same time?
Use randomized timeouts

None of the machines gets the majority?
Another round of elections

The candidate crashes?
If no-one hears from the new leader, another server will timeout and initiate an election

S2

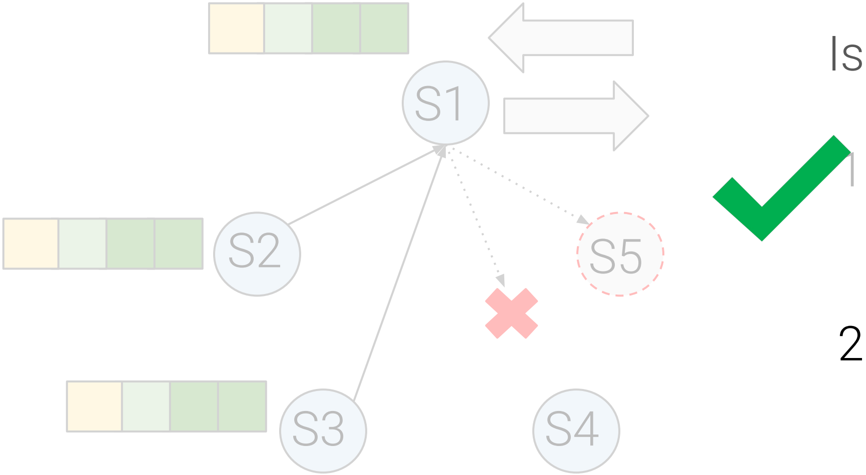
term: 1

vote to: S2

vote from: S2, S1

Raft: Basic Commit Operation

(*2f* +1) config for *f* failures (fail-stop, non-Byzantine), and use RPCs for messages



Issues to solve:

1. ✓ What happens when the leader crashes?
   Leader Election

2. How to ensure all servers have the same log entries in sequence?
   Log replication algorithm

Log Replication : Basics

(Recap): “Raft is an algorithm for managing replicated log”

The leader services client commands

- Append them in its log and send it out to followers (*AppendEntryRPC*)
- Difference between “replicated” and “committed”
- Difference between “committed” and “known to be committed”

The goal of Raft is to

- Ensure an identical log in the presence of failures
- Ensure that no committed entry is ever lost

Log Replication : the AppendEntryRPC Summary

Request :

- The current leaderID
- The current term
- prevIndex and prevTerm
- New entries[ ] to append
- Last committed entry

Identify holes/inconsistencies

Response :

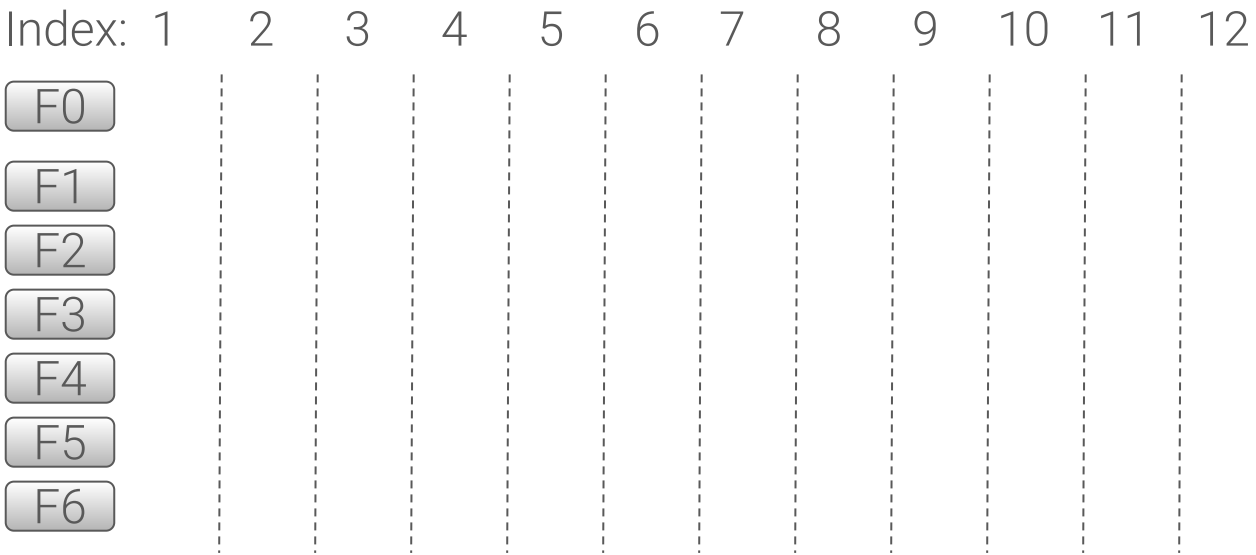
leaderLog(prevIndex, prevTerm)
==
followerLog(prevIndex, prevTerm)

- If the log matches, then accept and ACK
- Otherwise, reject and NACK
- *The current term that the follower knows*

If there was an election

Log Replication : An Example Problem

A log entry is identified by its index and its term



Index: 1 2 3 4 5 6 7 8 9 10 11 12

F0

F1

F2

F3

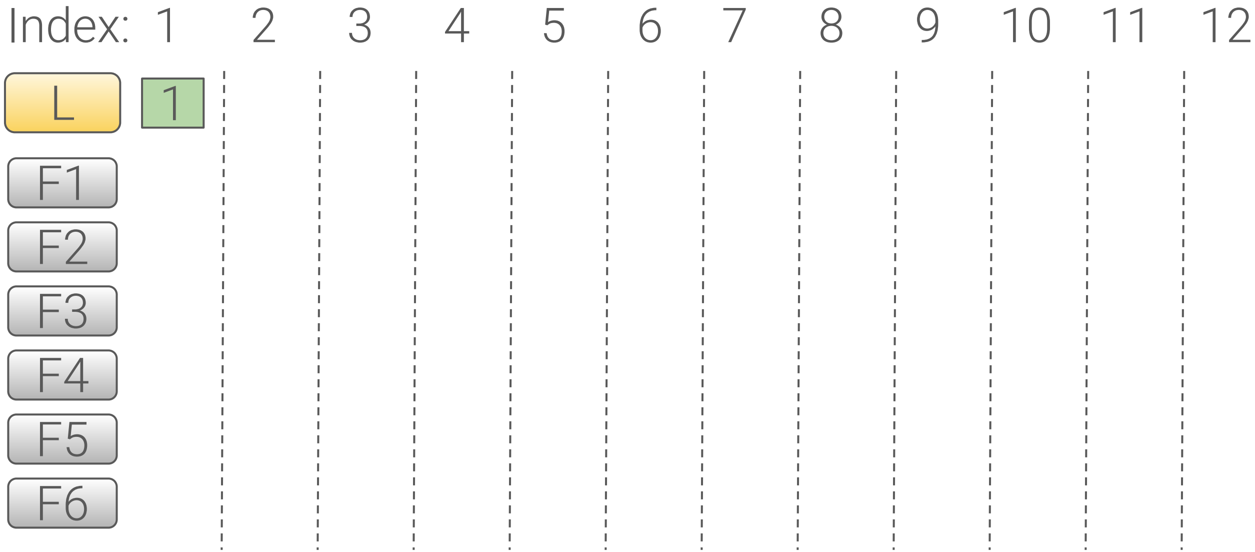
F4

F5

F6

Log Replication : An Example Problem

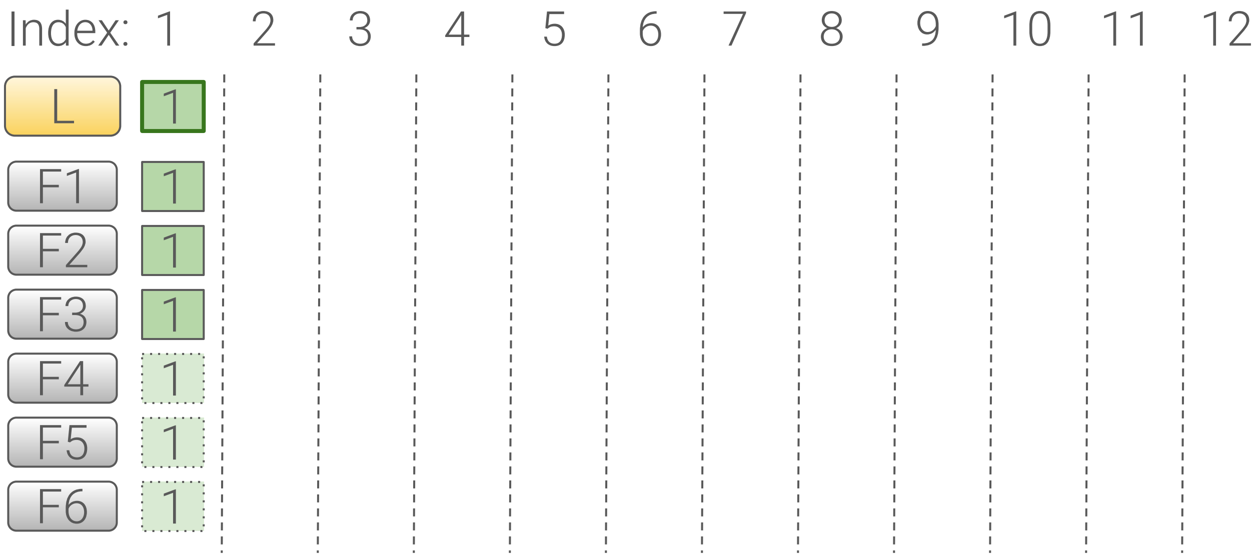
Let’s say F0 is the leader and It got a client command to commit



Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1											
F1												
F2												
F3												
F4												
F5												
F6												

Log Replication : An Example Problem

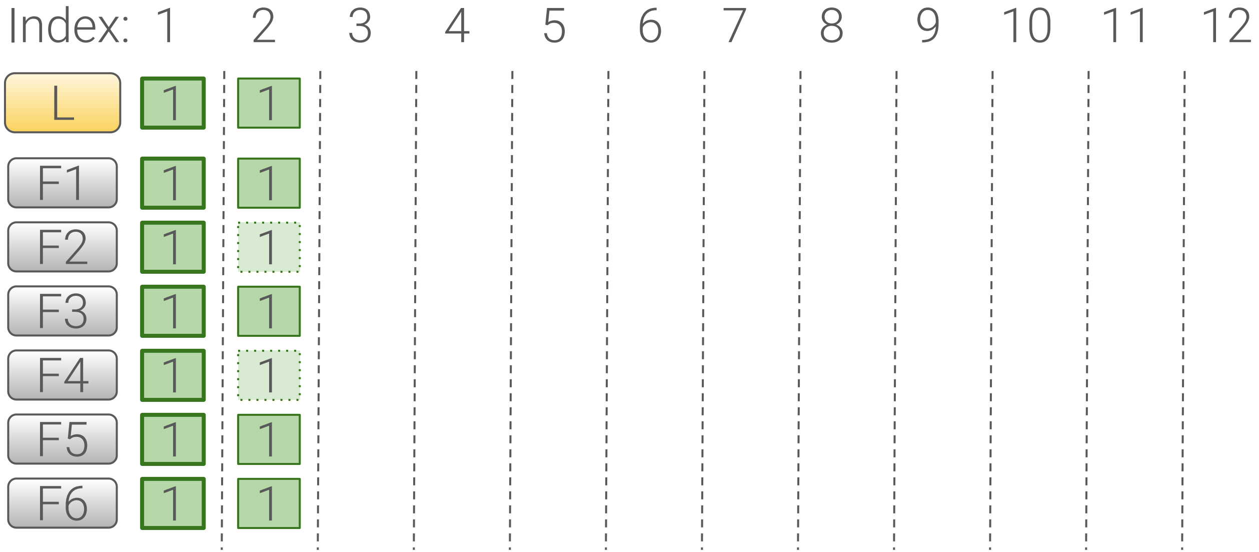
As soon as the majority has it, the entry is committed (but is not known)



Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1											
F1	1											
F2	1											
F3	1											
F4	1											
F5	1											
F6	1											

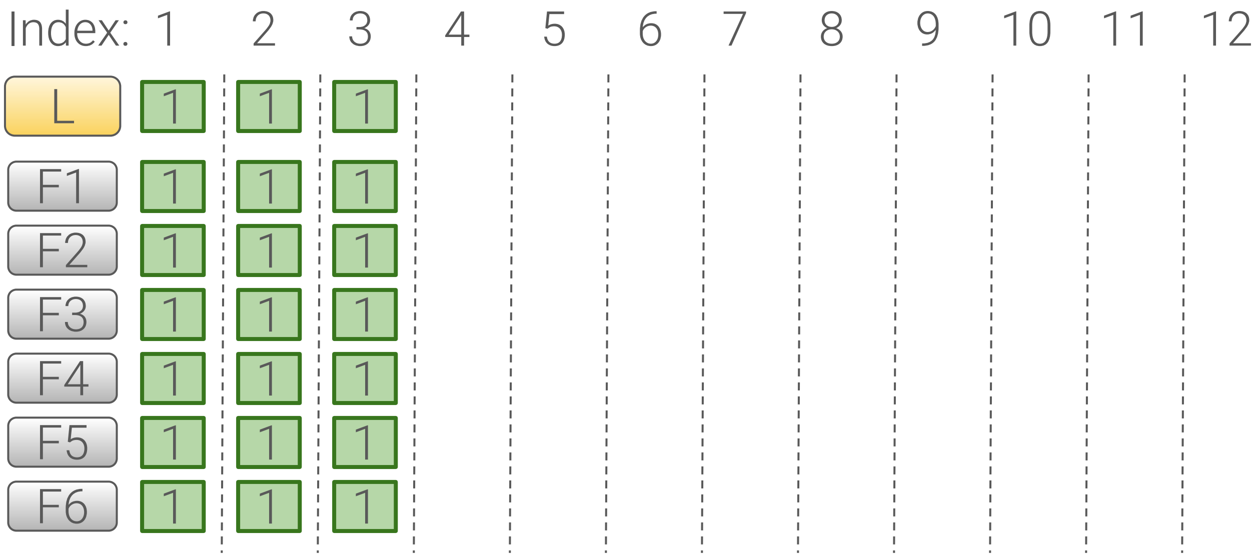
Log Replication : An Example Problem

With the next entry, the commitment of the last index can be announced



Log Replication : An Example Problem

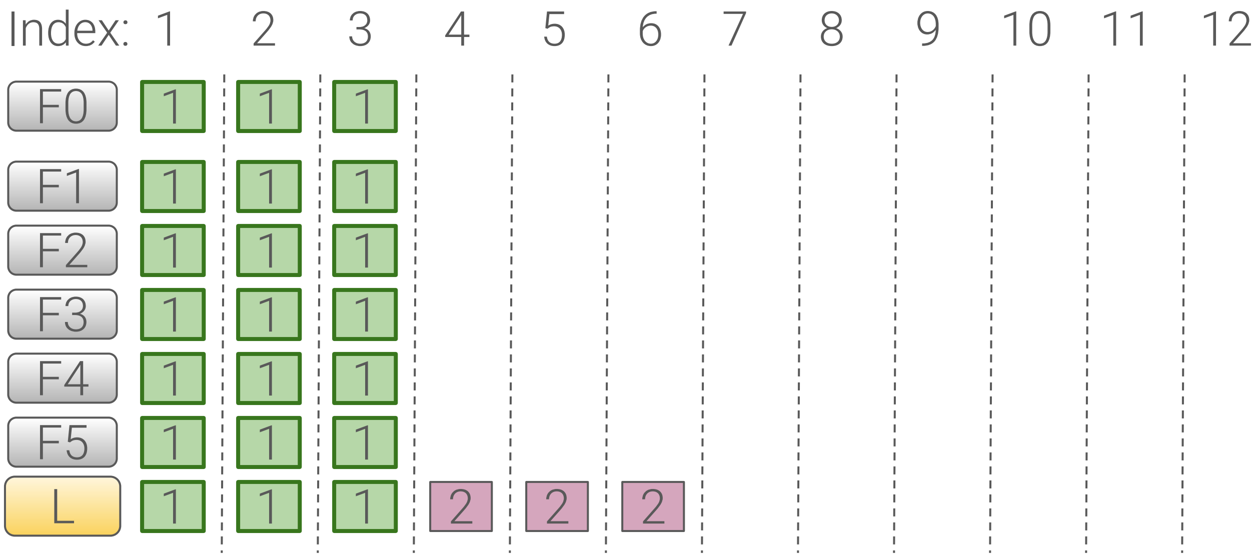
All good so far, then someone calls an election



Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1									
F1	1	1	1									
F2	1	1	1									
F3	1	1	1									
F4	1	1	1									
F5	1	1	1									
F6	1	1	1									

Log Replication : An Example Problem

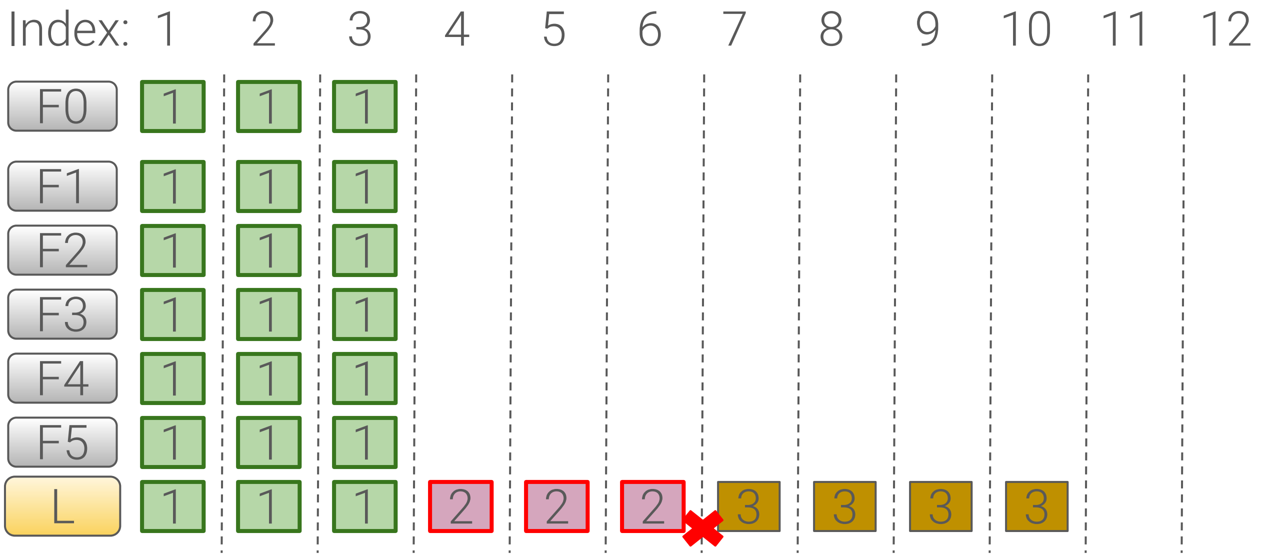
F6 become the leader and accepts 3 entries for the term 2



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1									
F1	1	1	1									
F2	1	1	1									
F3	1	1	1									
F4	1	1	1									
F5	1	1	1									
L	1	1	1	2	2	2						

Log Replication : An Example Problem

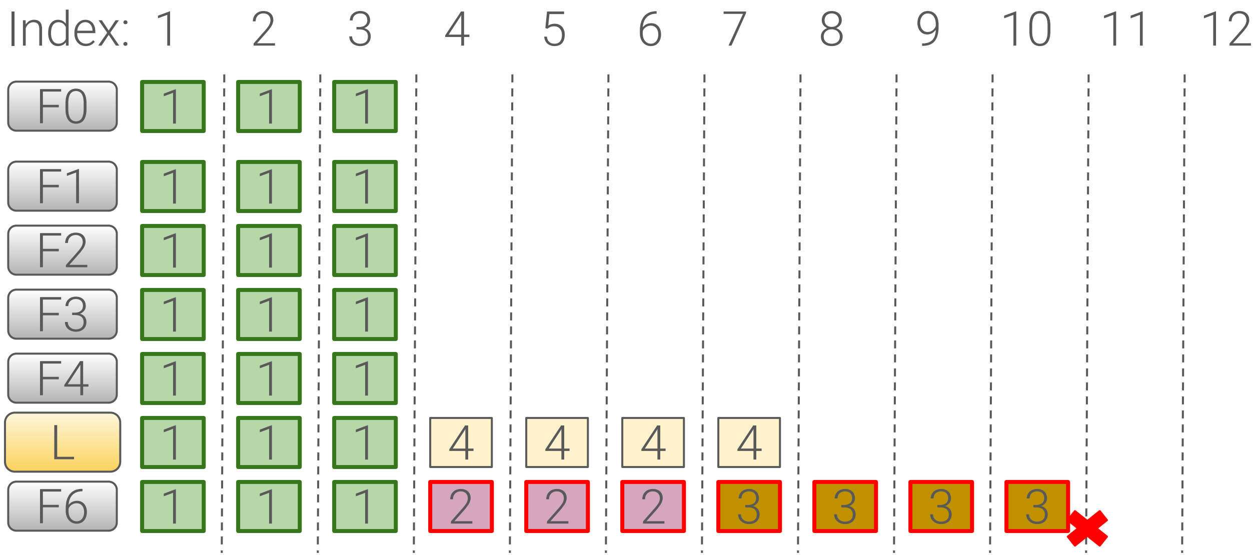
… becomes slow, someone calls an election, but F6 wins again !



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1									
F1	1	1	1									
F2	1	1	1									
F3	1	1	1									
F4	1	1	1									
F5	1	1	1									
L	1	1	1	2	2	2	3	3	3	3		

Log Replication : An Example Problem

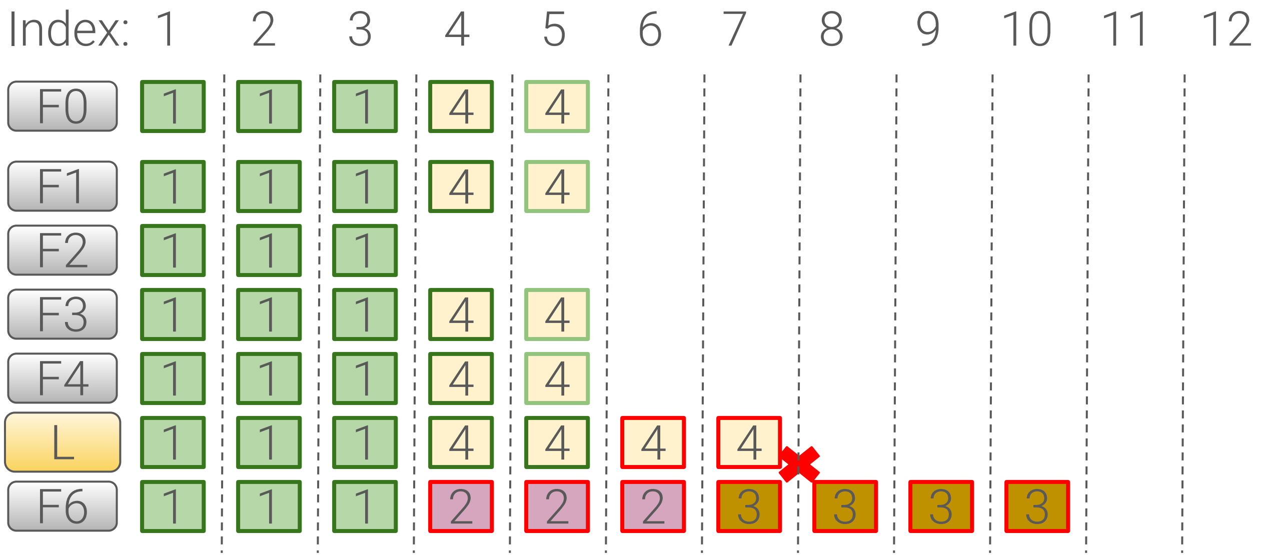
This time F5 wins and accepts 4 new entries for term 4



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1									
F1	1	1	1									
F2	1	1	1									
F3	1	1	1									
F4	1	1	1									
L	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3	✖	

Log Replication : An Example Problem

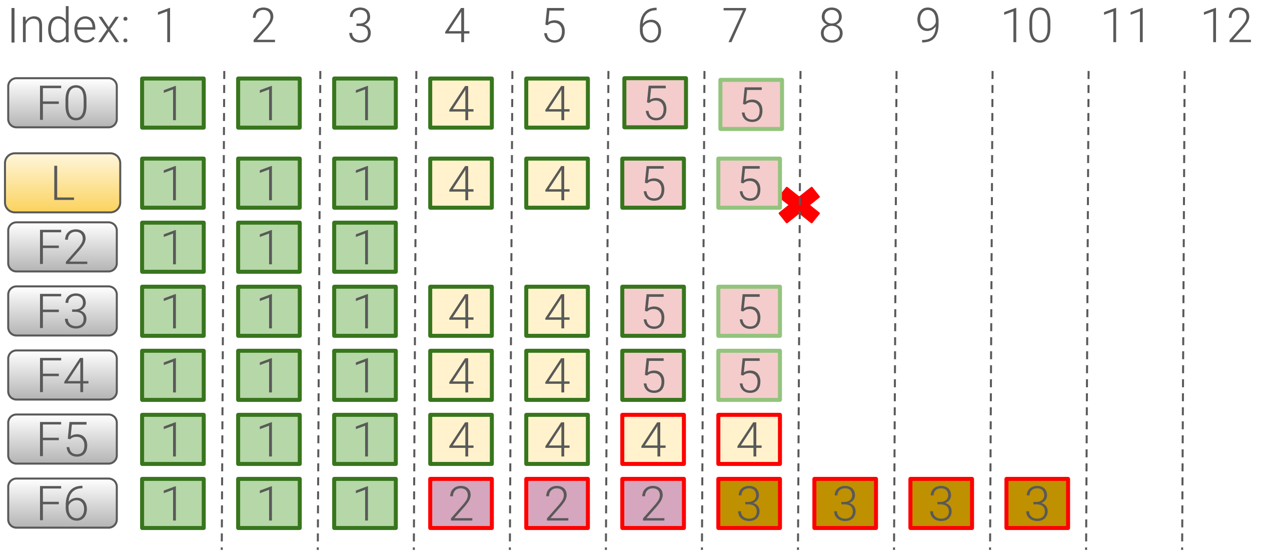
It manages to replicate 4th and 5th entry but failed after that. F2 is slow. Here, 4th index is committed and known, and 5th is implicitly committed.



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1	4	4							
F1	1	1	1	4	4							
F2	1	1	1									
F3	1	1	1	4	4							
F4	1	1	1	4	4							
L	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Log Replication : An Example Problem

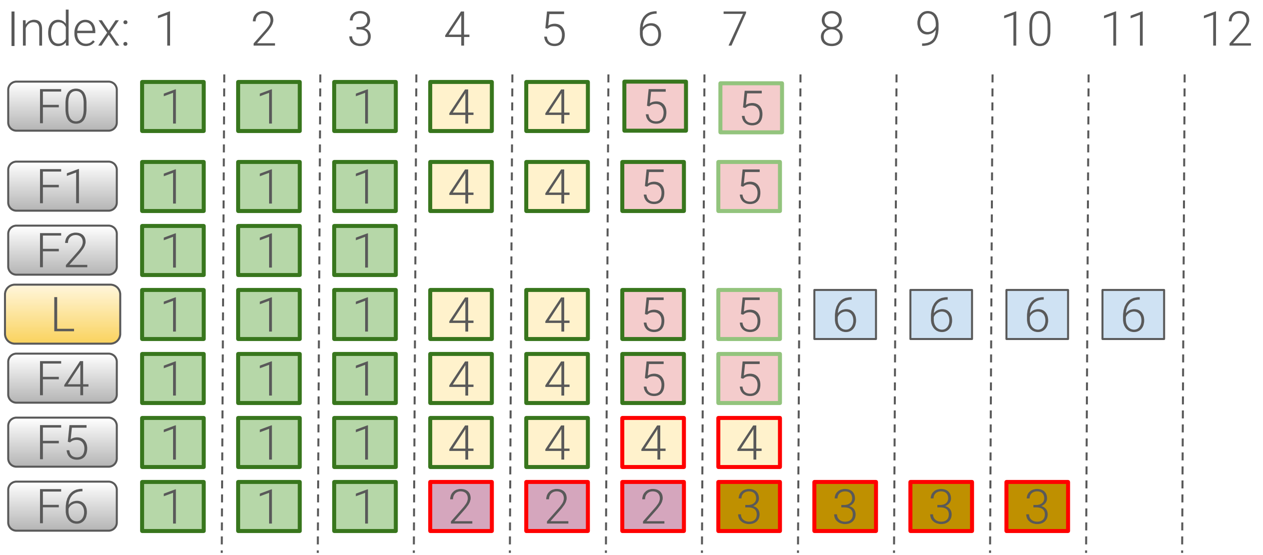
Now, F1 wins the election for the term 5 and replicates 6th and 7th entries. When it tries to commit the 6th entry, it will announce that index 5 is committed.



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1	4	4	5	5					
L	1	1	1	4	4	5	5	✖				
F2	1	1	1									
F3	1	1	1	4	4	5	5					
F4	1	1	1	4	4	5	5					
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Log Replication : An Example Problem

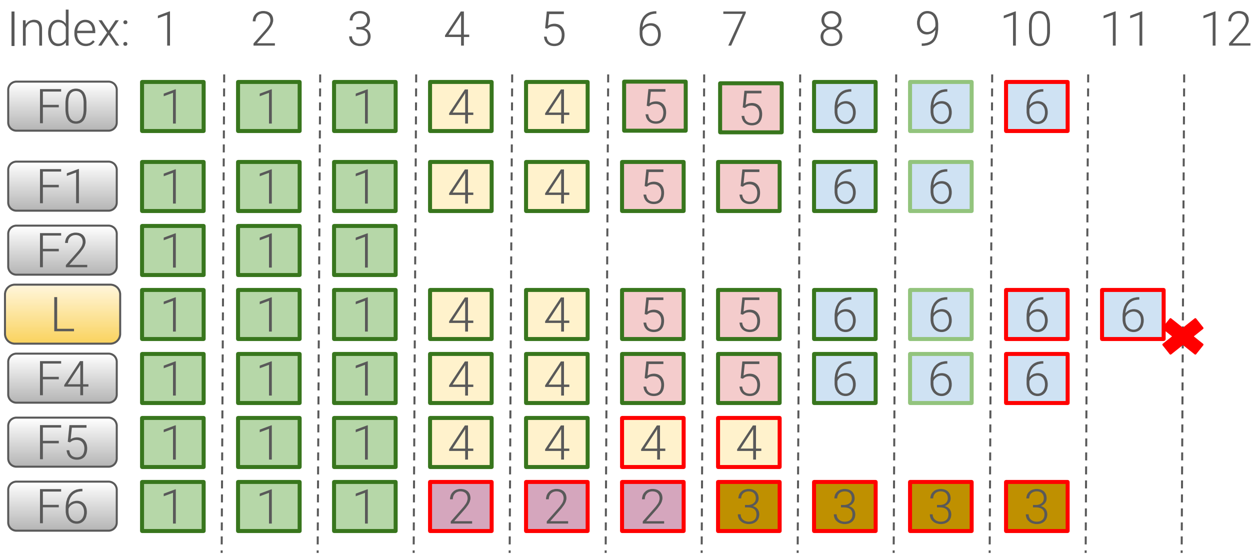
This time F3 wins the election for the term 6 and accepts 4 entries



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1	4	4	5	5					
F1	1	1	1	4	4	5	5					
F2	1	1	1									
L	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5					
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Log Replication : An Example Problem

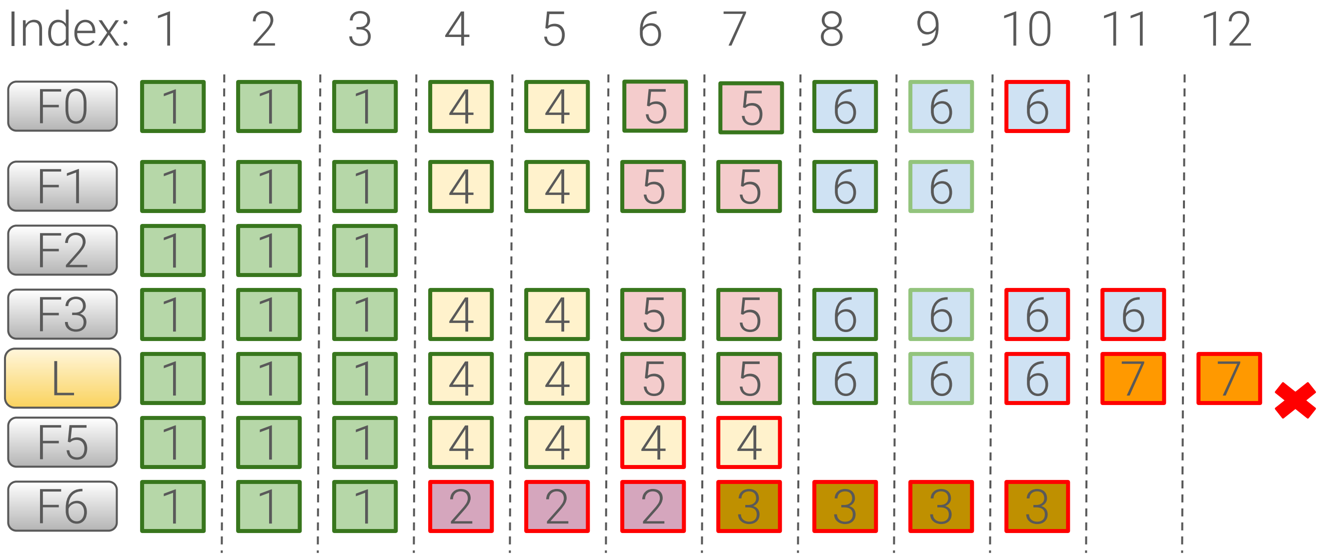
While replicating the index 8, F3 confirmed that index 7 is committed.



Index:	1	2	3	4	5	6	7	8	9	10	11	12
F0	1	1	1	4	4	5	5	6	6	6		
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
L	1	1	1	4	4	5	5	6	6	6	6	✖
F4	1	1	1	4	4	5	5	6	6	6		
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Log Replication : An Example Problem

This time F4 becomes the leader and accepts 2 entries for the term 7 and but dies



Index:	1	2	3	4	5	6	7	8	9	10	11	12	
F0	1	1	1	4	4	5	5	6	6	6			
F1	1	1	1	4	4	5	5	6	6				
F2	1	1	1										
F3	1	1	1	4	4	5	5	6	6	6	6		
L	1	1	1	4	4	5	5	6	6	6	7	7	✖
F5	1	1	1	4	4	4	4						
F6	1	1	1	2	2	2	3	3	3	3			

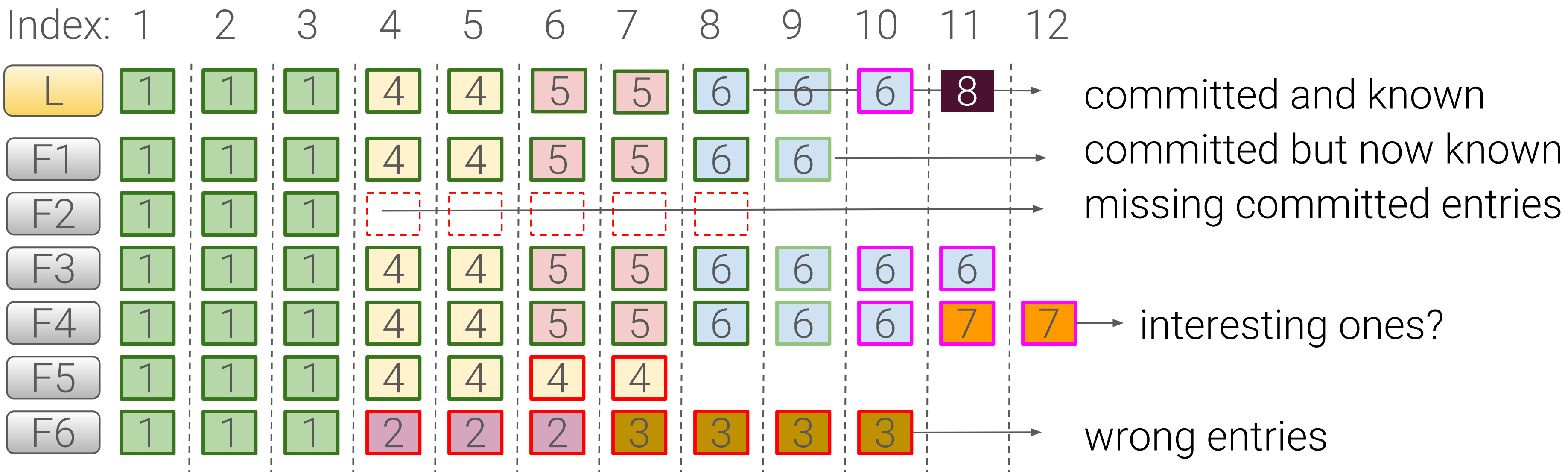
Log Replication : An Example Problem

Now L0 becomes the leader again and we have …

Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Log Replication : An Example Problem

Now L0 becomes the leader again and we have …



Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

committed and known

committed but now known

missing committed entries

interesting ones?

wrong entries

Log Replication : Raft’s Solution

The leader’s log is THE log

No provisions for finding and reconciling holes

Log information flows in one direction, from the leader to followers

The leader can overwrite follower’s log with its entries

The leader also tells which entries are committed (to apply the state machine)

How does the Leader do it: find the last point where its log and a follower’s log disagree and from there on re-write follower’s log

Log Replication : An Example Problem

Try to match index by index

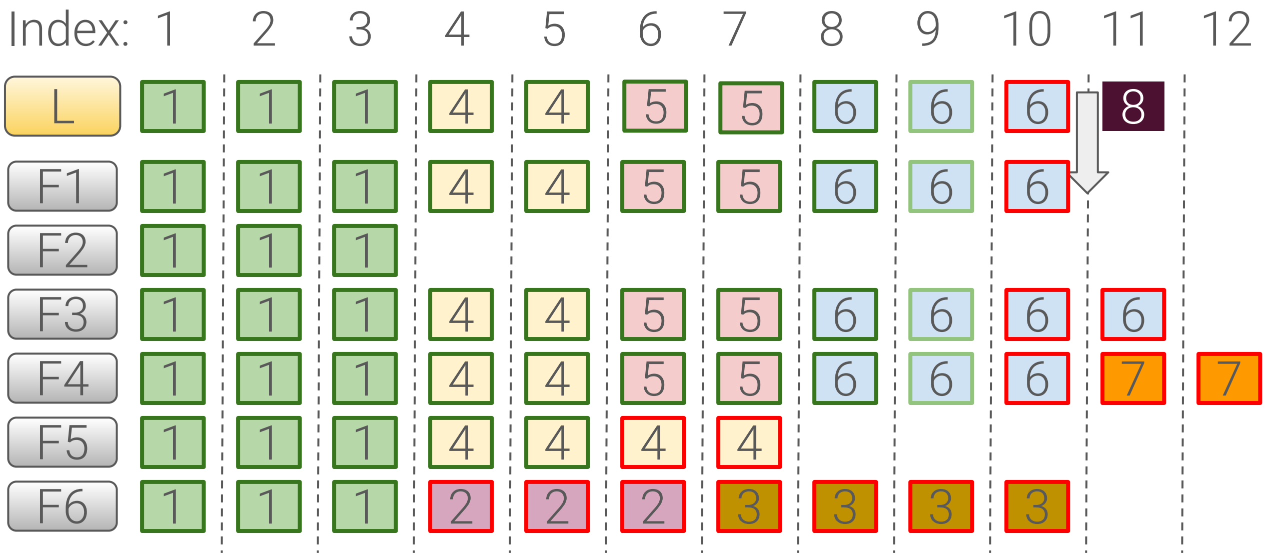
Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6		8	
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Commit 8 with (prevIndex:10, prevTerm:6)?

No

Log Replication : An Example Problem

Try to match index by index



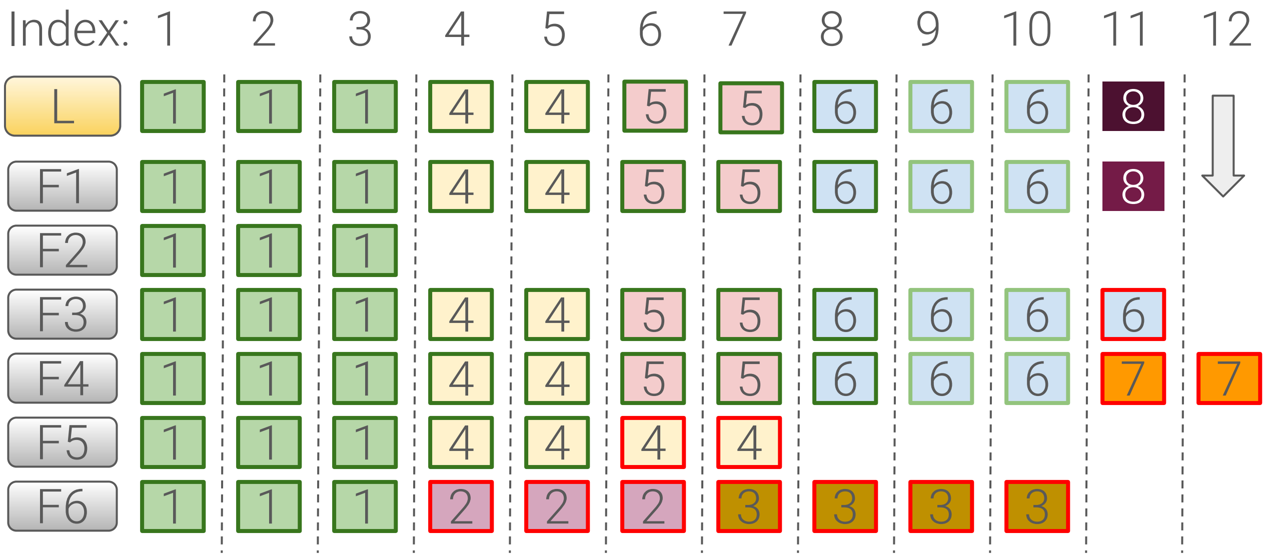
Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6	6		
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Commit 6 with (prevIndex:9, prevTerm:6)?

YES!

Log Replication : An Example Problem

Try to match index by index (see index 10 is now implicitly committed)



Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6	6	8	
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Commit 8 with
(prevIndex:10, prevTerm:6)?
YES!

Log Replication : An Example Problem

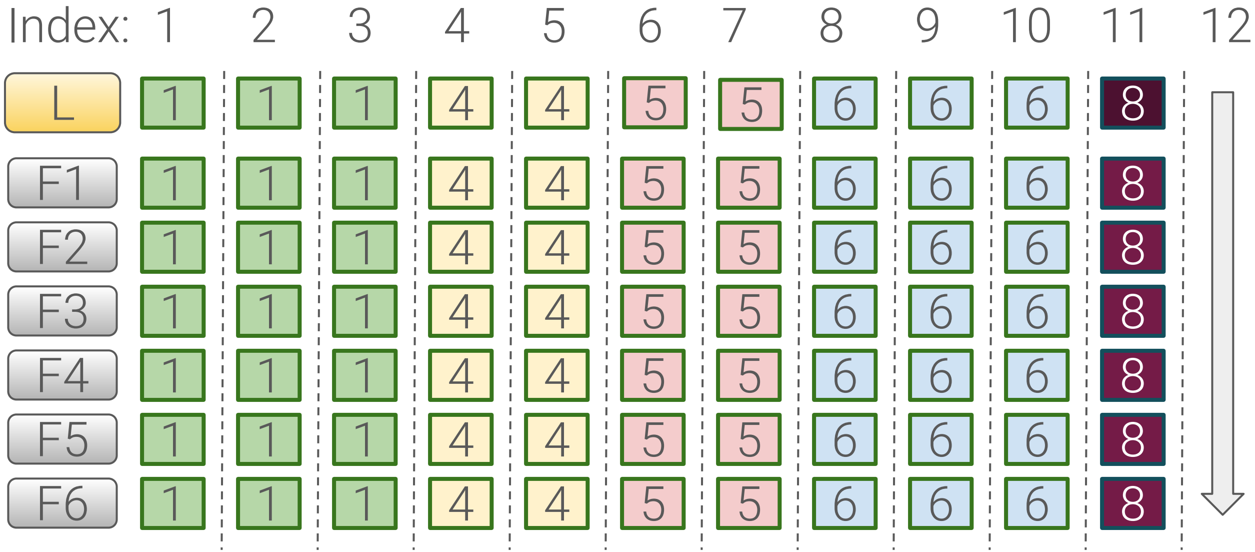
And implicitly discover which entries are now committed (see index 9 and 10)

Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6	6	8	
F2	1	1	1	4	4	5	5	6	6	6	8	
F3	1	1	1	4	4	5	5	6	6	6	8	
F4	1	1	1	4	4	5	5	6	6	6	8	
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

You overwrite entries in the follower's log

Log Replication : An Example Problem

If no failures then all log entries converges and all committed entries and discovered and announced

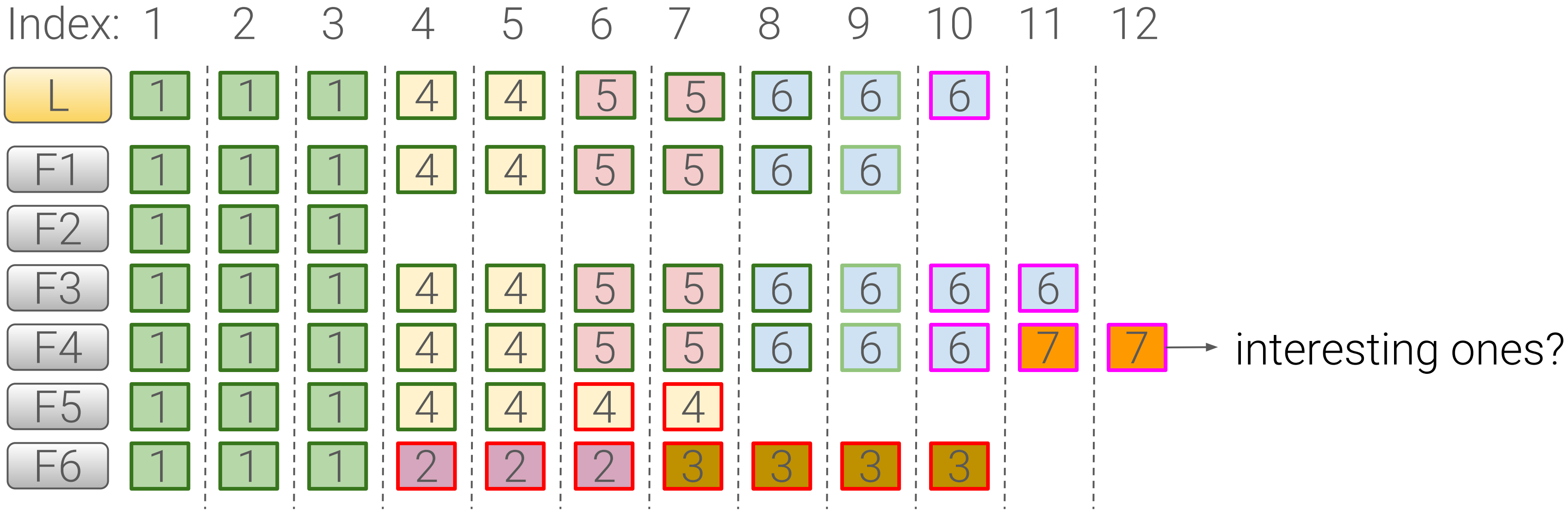


Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6	8	
F1	1	1	1	4	4	5	5	6	6	6	8	
F2	1	1	1	4	4	5	5	6	6	6	8	
F3	1	1	1	4	4	5	5	6	6	6	8	
F4	1	1	1	4	4	5	5	6	6	6	8	
F5	1	1	1	4	4	5	5	6	6	6	8	
F6	1	1	1	4	4	5	5	6	6	6	8	

But wait, there is more !

Index 10 survived and got committed. What if F3 became a leader? Or F4?

Or worse F6?



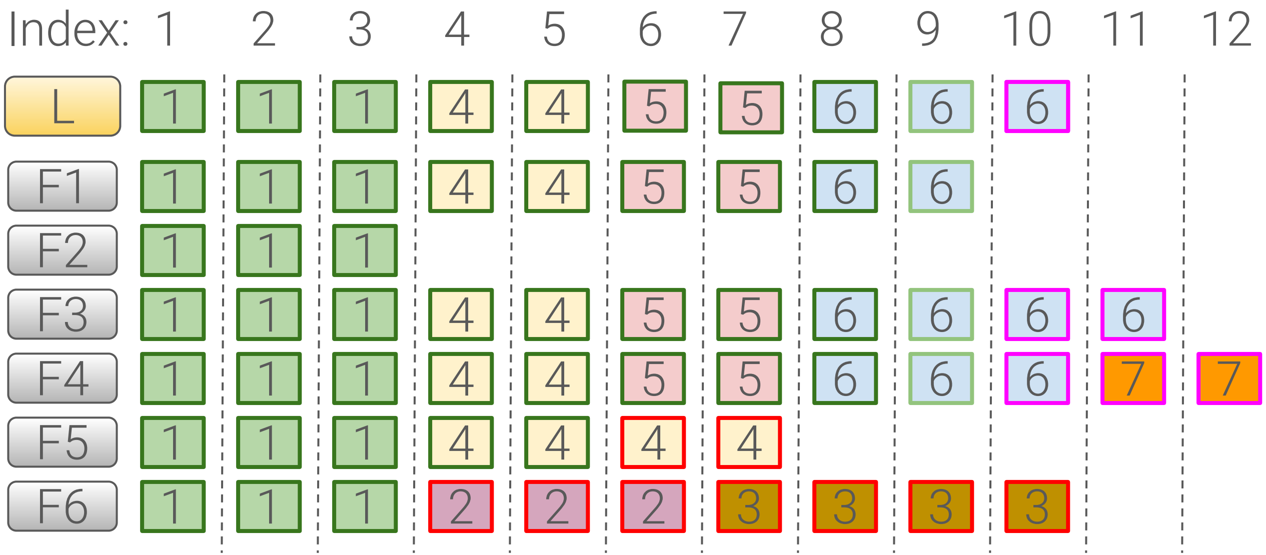
Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6		
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

interesting ones?

Raft Leader Election: Revisited

RequestVoteRPC [term, ID]

ResponseVoteRPC : the first one (with a higher term) gets the vote



Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6		
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Raft Leader Election: Revisited

RequestVoteRPC [term, ID] **+ (last index and term)**

ResponseVoteRPC : the first one (with a higher term) gets the vote **+ (given that candidate's log is as up-to-date as the voter's)**

Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6		
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Up-to-date?

1. Compare terms
2. Compare length

Raft Leader Election: Revisited

RequestVoteRPC [term, ID] **+ (last index and term)**

ResponseVoteRPC : the first one (with a higher term) gets the vote **+ (given that candidate's log is as up-to-date as the voter's)**

Index:	1	2	3	4	5	6	7	8	9	10	11	12
L	1	1	1	4	4	5	5	6	6	6		
F1	1	1	1	4	4	5	5	6	6			
F2	1	1	1									
F3	1	1	1	4	4	5	5	6	6	6	6	
F4	1	1	1	4	4	5	5	6	6	6	7	7
F5	1	1	1	4	4	4	4					
F6	1	1	1	2	2	2	3	3	3	3		

Vote?

F0 : higher term in the log

F1: higher term in the log

F2: OK

F3: higher term in the log

F4: higher term in the log

F5: higher term in the log

F6: OK

Raft Leader Election: Revisited

RequestVoteRPC [term, ID] **+ (<u>last index and term</u>)**

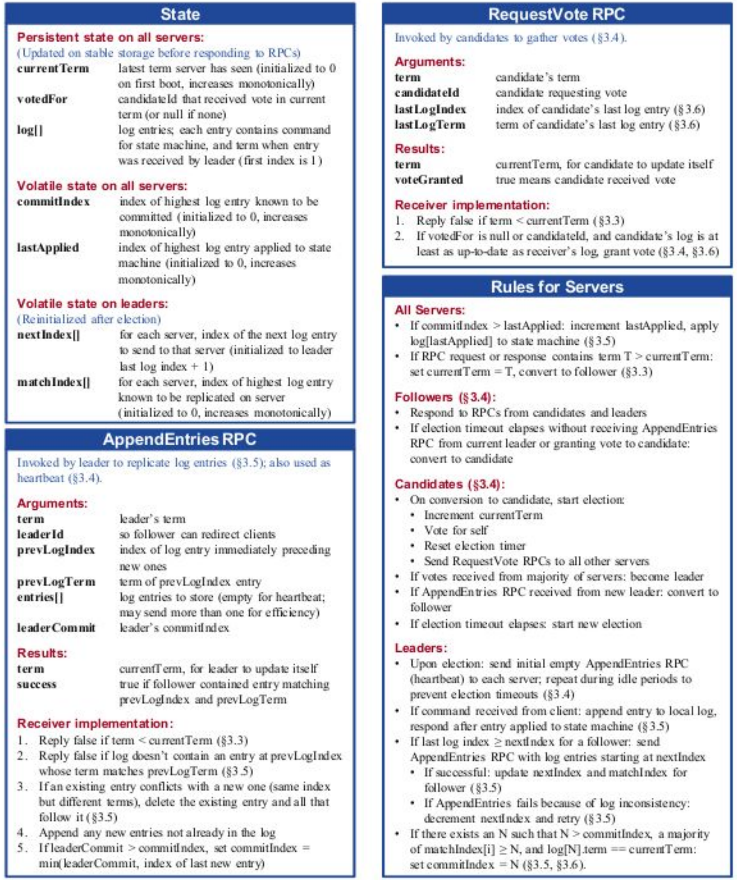
ResponseVoteRPC : the first one (with a higher term) gets the vote **+ (<u>given that candidate's log is as up-to-date as the voter's</u>)**

Index:	1	2	3	4	5	6	7	8	9	10	11	12	
L	1	1	1	4	4	5	5	6	6	6			L
F1	1	1	1	4	4	5	5	6	6				L
F2	1	1	1										
F3	1	1	1	4	4	5	5	6	6	6	6		L
F4	1	1	1	4	4	5	5	6	6	6	7	7	L
F5	1	1	1	4	4	4	4						
F6	1	1	1	2	2	2	3	3	3	3			✗

That is it !

Obviously there are more details about

- Client interaction
- Membership management
- Log Compaction
- State-machine management
- Checkpointing and restore
- Practical guidelines
  - How to deploy
  - Timeout recommendations
- ...



State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:

(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§3.5); also used as heartbeat (§3.4).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§3.4).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§3.6)
lastLogTerm	term of candidate's last log entry (§3.6)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§3.5)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§3.3)

Followers (§3.4):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§3.4):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPC (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§3.4)
- If command received from client: append entry to local log, respond after entry applied to state machine (§3.5)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§3.5)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§3.5)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§3.5, §3.6).

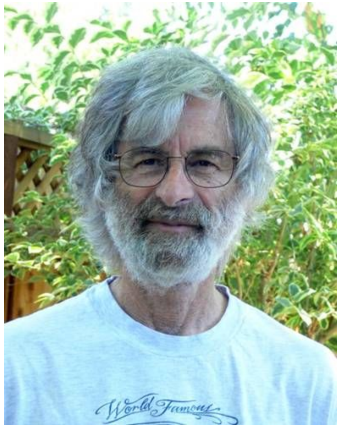
Figure 3.1: A condensed summary of the Raft consensus algorithm (excluding membership changes, log compaction, and client interaction). The server behavior in the lower-right box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §3.4 indicate where particular features are discussed. The formal specification in Appendix B describes the algorithm more precisely.

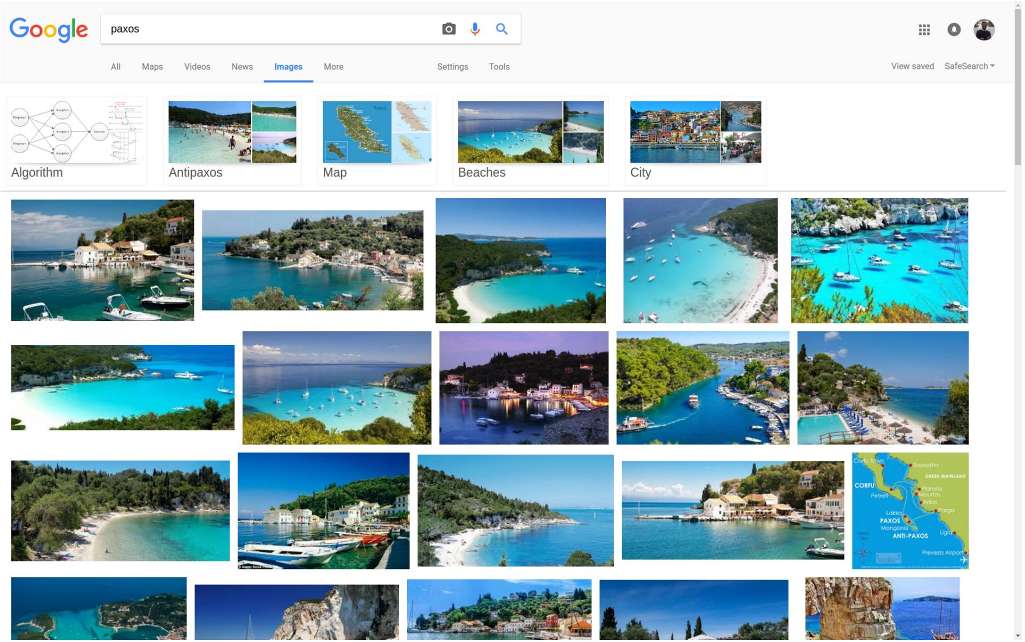
Understandability

...in Comparison to What?

Paxos.

A set of protocols (not islands), proposed by Leslie Lamport





Paxos: History

http://lamport.azurewebsites.net/pubs/pubs.html#lamport-paxos

“[...]Writing about a lost civilization allowed me to eliminate uninteresting details and indicate generalizations by saying that some details of the parliamentary protocol had been lost. To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist, replete with Stetson hat and hip flask.”

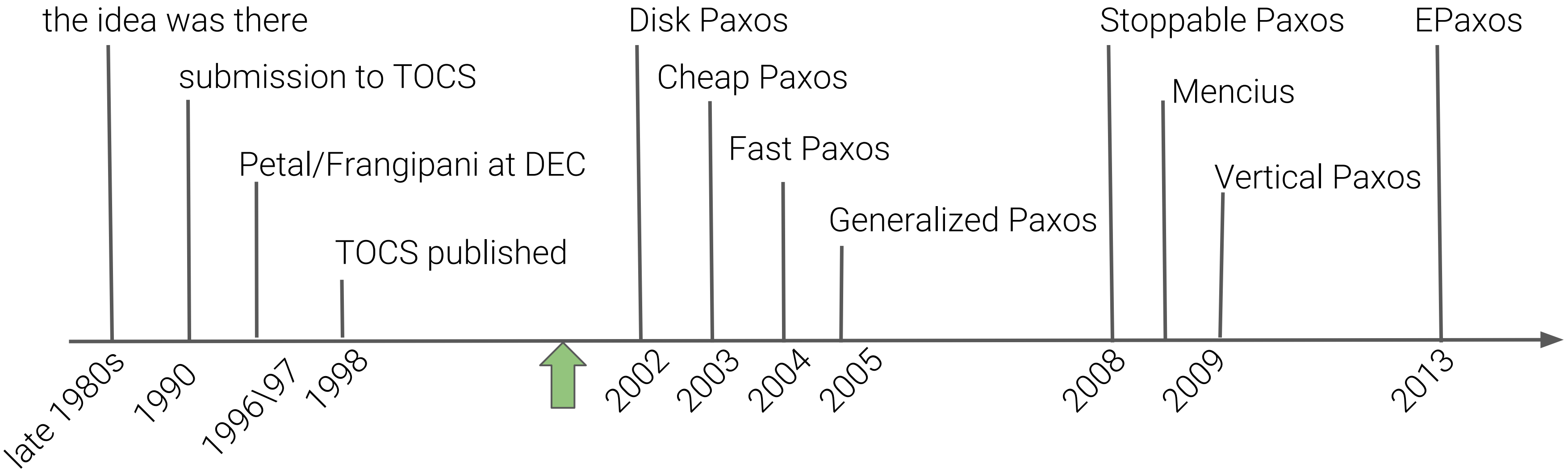
“My attempt at inserting some humor into the subject was a dismal failure. People who attended my lecture remembered Indiana Jones, but not the algorithm. People reading the paper apparently got so distracted by the Greek parable that they didn't understand the algorithm.[...]”

“I submitted the paper to *TOCS* in 1990. All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed. [...]”

“When Ed Lee and I (Chandu Thekkath) were working on Petal we needed some sort of commit protocol to make sure global operations in the distributed system completed correctly in the presence of server failures. [...] Leslie gave Ed a copy of the *Part-Time Parliament* tech report, which we both enjoyed reading. I particularly liked its humour and to this day, cannot understand why people don't like that tech report.[...]”

“Meanwhile, the one exception in this dismal tale was Butler Lampson, who immediately understood the algorithm's significance. He mentioned it in lectures and in a paper, and he interested Nancy Lynch in it.[...]”

Which Paxos: A Brief History



the idea was there

submission to TOCS

Petal/Frangipani at DEC

TOCS published

Disk Paxos

Cheap Paxos

Fast Paxos

Generalized Paxos

Stoppable Paxos

Mencius

Vertical Paxos

EPaxos

late 1980s

1990

1996\97

1998

2002

2003

2004

2005

2008

2009

2013

Taken from http://paxos.systems/variants.html

Paxos

Paxos has become synonym with consensus

Paxos is a broad term for a whole family of consensus protocols

Paxos’s claim to fame is its elegant and thorough treatment of the problem

The Problem: “reaching consensus on a single value in an asynchronous setting”

+ Very well understood and analyzed problem
- How often does a system choose a single value? (Multi-Paxos is not specified precisely in literature)

Paxos Challenges

Reaching a consensus on a single-value is just the first step towards highly-available systems

1. How to reach a consensus on a single value?

fundamental problem

2. How to reach consensus on multiple values?
3. How to maintain the consensus/decision log?
4. How to maintain the state-machine?
5. How to manage cluster membership?
6. How clients interact with the system and what kind of guarantees they get?
7. Bootstrapping and recovery of the system?
8. Performance optimizations, and maintenance ?

Paxos Followups

- *How to build a highly available system using consensus*. Lampson, 1996
- *Paxos made simple*, Lamport, SIGACT 2001
- *The ABCD's of Paxos*, Lampson, PODC 2001
- *Deconstructing Paxos*, Boichat et al., SIGACT 2003
- *Generalized consensus and Paxos*, Lamport, 2005
- *Paxos made live: an engineering perspective*, Chandra et al., PODC 2007
- *Paxos Made Practical*. Mazieres, 2007
- *Paxos for system builders: an overview*, Kirsch et al., LADIS 2008
- *Paxos made moderately complex*, Van Renesse, 2012
- … and many more

"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." - Chandra et al., PODC'07

"The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-). —NSDI reviewer" - Ongario, Raft presentations

Paxos Challenges

Reaching a consensus on a single-value is just the first step towards highly-available systems

1. How to reach a consensus on a single value?
2. How to reach consensus on multiple values?
3. How to maintain the consensus/decision log?
4. How to maintain the state-machine?
5. How to manage cluster membership?
6. How clients interact with the system and what kind of guarantees they get?
7. Bootstrapping and recovery of the system?
8. Performance optimizations, and maintenance ?

Raft Approach

Reaching a consensus on a single-value is just the first step towards highly-available systems

1. ~~How to reach a consensus on a single value?~~
2. ~~How to reach consensus on multiple values?~~
3. How to maintain the consensus/decision log?
4. How to maintain the state-machine?
5. How to manage cluster membership?
6. How clients interact with the system and what kind of guarantees they get?
7. Bootstrapping and recovery of the system?
8. Performance optimizations, and maintenance ?

fundamental problem

My Subjective Impression

Is Raft easier to understand and explain? **Yes**

My Subjective Impression

Is Raft easier to understand and explain? **Yes**

+ An interesting read of a complete system design
+ Fewer numbers of states to consider
+ Intuition is easy, e.g., do timeouts reset after an election? Yes. As they meant to break ties in case of a split vote, each machine sets a new timeout for a new election/term

But we are comparing a Raft system with Paxos protocol

My Subjective Impression

Is Raft easier to understand and explain? **Yes**

- + An interesting read of a complete system design
- + Fewer numbers of states to consider
- + Intuition is easy, e.g., do timeouts reset after an election? Yes. As they meant to break ties in case of a split vote, each machine sets a new timeout for a new election/term

But we are comparing a Raft system with Paxos protocol

Is everything sorted out in Raft?

- - Flaky machine (liveness vs. safety), client interaction is fuzzy (!), Byzantine faults, slow leader, non-commutative optimizations,...

Some Amusement from the User Study

“Cool idea, I think it’s what Paxos should have been.”

“The quizzes were too long. Could not complete in the time provided. [...]”

“Paxos is eaiser to understand because it does not have any many details as Raft. [...]”

“Both are super complex!”

“Good job on the lecture videos.”

“Videos were a bit dry”

“I obviously noticed that Raft and Paxos are very similar - to the point that I feel like Raft is actually paxos presented differently. […]”

“Ousterhout is a boss. Thanks for the lectures!”

Discussion

Raft: Client Interaction - 1/3

How does a client finds a Raft cluster servers?

- Use network broadcast or DNS lookups (requires network help)

How does a client find the leader?

- A follower can tell the identity of the leader to a client

How does a client request is serviced?

- The leader processes a the request by replicated it across the cluster

Raft: Client Interaction - 2/3

Consistency semantics? As such it would provide “at-least once” semantics, why?

- Leader failure before acknowledgement
- Network duplicates

With some modifications, Raft can provide Linearizability : “instantaneously, exactly once”

- Identify duplicate commands
  - give clients session IDs and command IDs
  - Manage session (addition, expiration, stale, etc.)

Raft: Client Interaction - 2/3

Read-only command optimizations

Can anyone serve? No, as all servers are not in the same state

Can only-leader server? No, it might have lost its leader status

- A leader voluntarily steps down if it cannot maintain heart-beats

Combination of the two? Yes, but without complete log replication protocol

- Identify committed entries after re-election with no-op command
- Ensure your leader status with heart-beats and reply
- Alternatively, “sequential-consistency” can be provided as well

Raft Cluster Membership Changes - 1/2

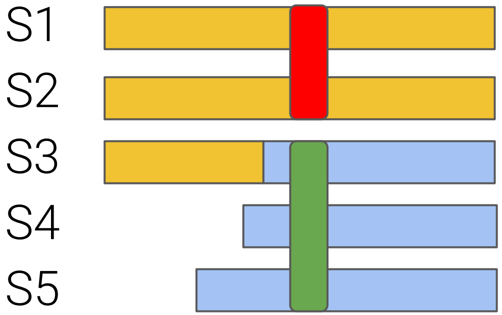
Avoid complicated multi-server changes

Series of ±1 server changes

- Maintain a majority overlap between C_{old} and C_{new} configurations

Servers act immediately on a new configuration as soon as it is added to the log

Next round of changes are allowed as soon as the current one is committed



two distinct majorities, when moving from a 3 to 5 nodes cluster

Raft Cluster Membership Changes - 2/2

Servers

- Give votes without checking the current configuration
- Accept entries without checking the current configuration

Optimizations/Concerns/Recommendations

- A new servers are caught up as a non-voting member
  - The leader can abort the change if the the new server is slow or non-responsive
- Add servers before removing to maintain fault-tolerance level
- Bootstrap the first server with a configuration with itself in it

What are the Other Options?

Leader-based protocols : Viewstamped Replication, ZAB (ZooKeeper), etc.

Process of the election

- Detection of a failed leader (how?, practical recommendations?)
- Who can become leader (any or specific server(s)?)

Performance considerations

- Rotating leaders (Mencius)
- Offloading to clients (Fast-Paxos)
- Exploiting commutativity (Generalized-Paxos, EPaxos)