CSCI 5673
Distributed Systems

Lecture Set Eight

Raft: An Understandable Consensus Algorithm

Lecture Notes by
Shivakant Mishra
Computer Science, CU Boulder
Last update: February 28, 2017

D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. USENIX ATC 2014.

Motivation

- ***Paxos***
  - Current standard for both teaching and implementing consensus algorithms
  - Very difficult to ***understand*** and very hard to ***implement***
- ***Raft***
  - New protocol (2014)
  - Much easier to ***understand***
  - Several ***open-source implementations***

Key features of Raft

- ***Strong leader:***
  - Leader does most of the work:
    - Issues ***all*** log updates
- ***Leader election:***
  - Uses ***randomized timers*** to elect leaders.
- ***Membership changes:***
  - New ***joint consensus*** approach where the majorities of two different configurations are required

Raft consensus algorithm (I)

- Servers start by electing a ***leader***
  - Sole server habilitated to accept commands from clients
  - Will enter them in its log and forward them to other servers
  - Will tell them when it is safe to apply these log entries to their state machines

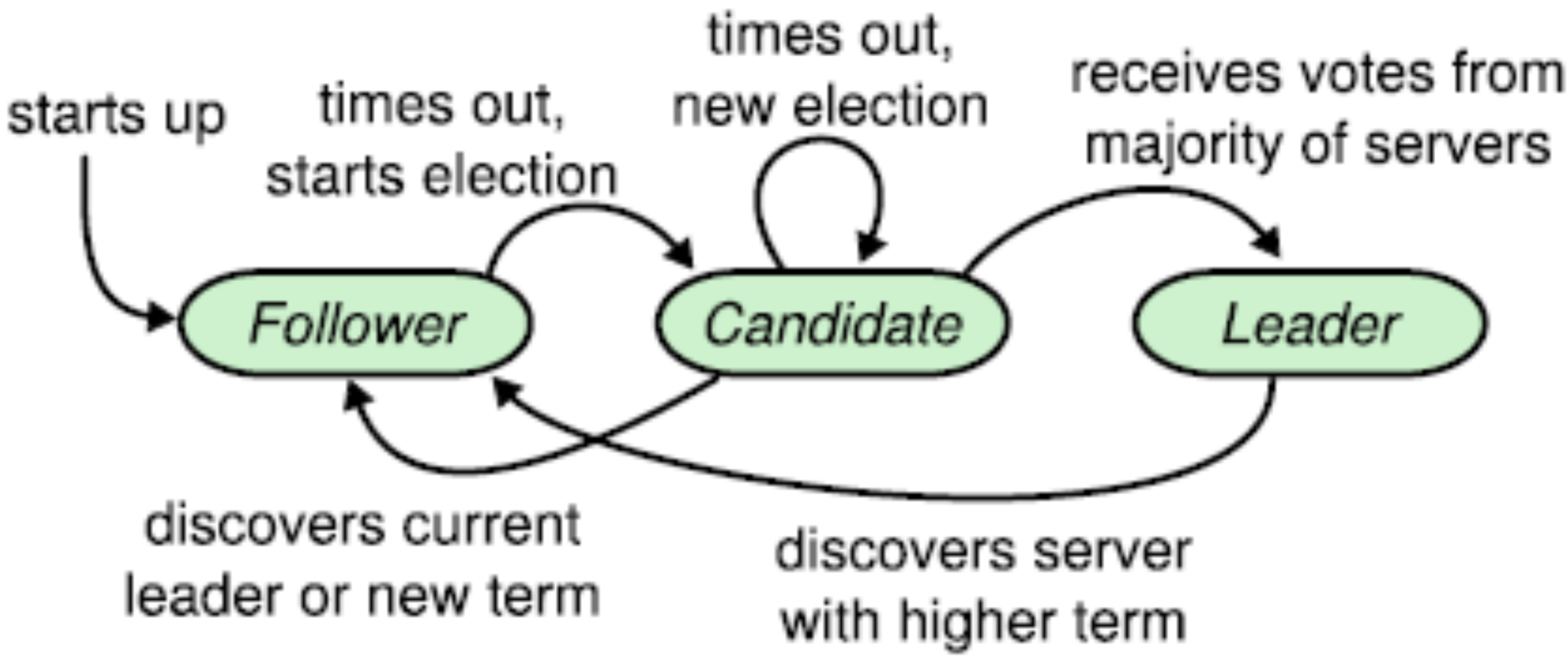
Raft consensus algorithm (II)

- Decomposes the problem into three fairly independent subproblems
  - ***Leader election:***
    How servers will pick a—***single***—leader
  - ***Log replication:***
    How the leader will accept log entries from clients, propagate them to the other servers and ensure their logs remain in a consistent state
  - ***Safety***

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

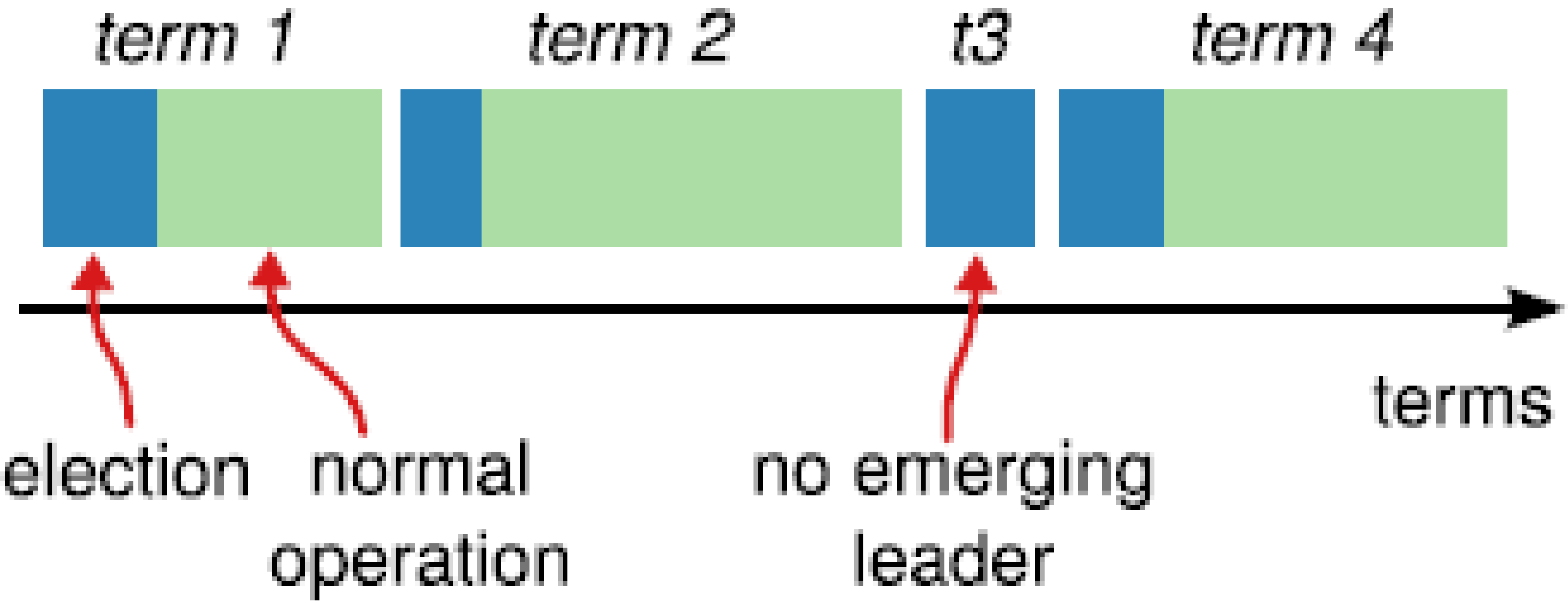
Server states



Raft basics: terms (I)

- Epochs of arbitrary length
  - Start with the election of a leader
  - End when
    - No leader can be selected (split vote)
    - Leader becomes unavailable
- Different servers may observe transitions between terms at different times or even miss them

Raft basics: terms (II)



Raft basics: terms (III)

- Terms act as logical clocks
  - Allow servers to detect and discard obsolete information (messages from stale leaders, …)
- Each server maintains a current term number
  - Includes it in all its communications
- A server receiving a message with a high number updates its own number
- A leader or a candidate receiving a message with a high number becomes a follower

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***

Starting an election

- When a follower starts an election, it
  - Increments its current term
  - Transitions to candidate state
  - Votes for itself
  - Issues ***RequestVote*** RPCs in parallel to all the other servers in the cluster.

Acting as a candidate

- A candidate remains in that state until
  - It wins the election
  - Another server becomes the new leader
  - A period of time goes by with no winner

Winning an election

- Must receive votes from a majority of the servers in the cluster for the same term
  - Each server will vote for at most one candidate in a given term
    - The first one that contacted it
- Majority rule ensures that at most one candidate can win the election in a given term
- Winner becomes ***leader*** and sends heartbeat messages to all of the other servers
  - To assert its new role

Hearing from other servers

- Candidates may receive an ***AppendEntries*** RPC from another server claiming to be leader
- If the leader's term is at greater than or equal to the candidate's current term, the candidate recognizes that leader and returns to follower state
- Otherwise the candidate ignores the RPC and remains a candidate

Split elections

- No candidate obtains a majority of the votes in the servers in the cluster
- Each candidate will time out and start a new election
  - After incrementing its term number

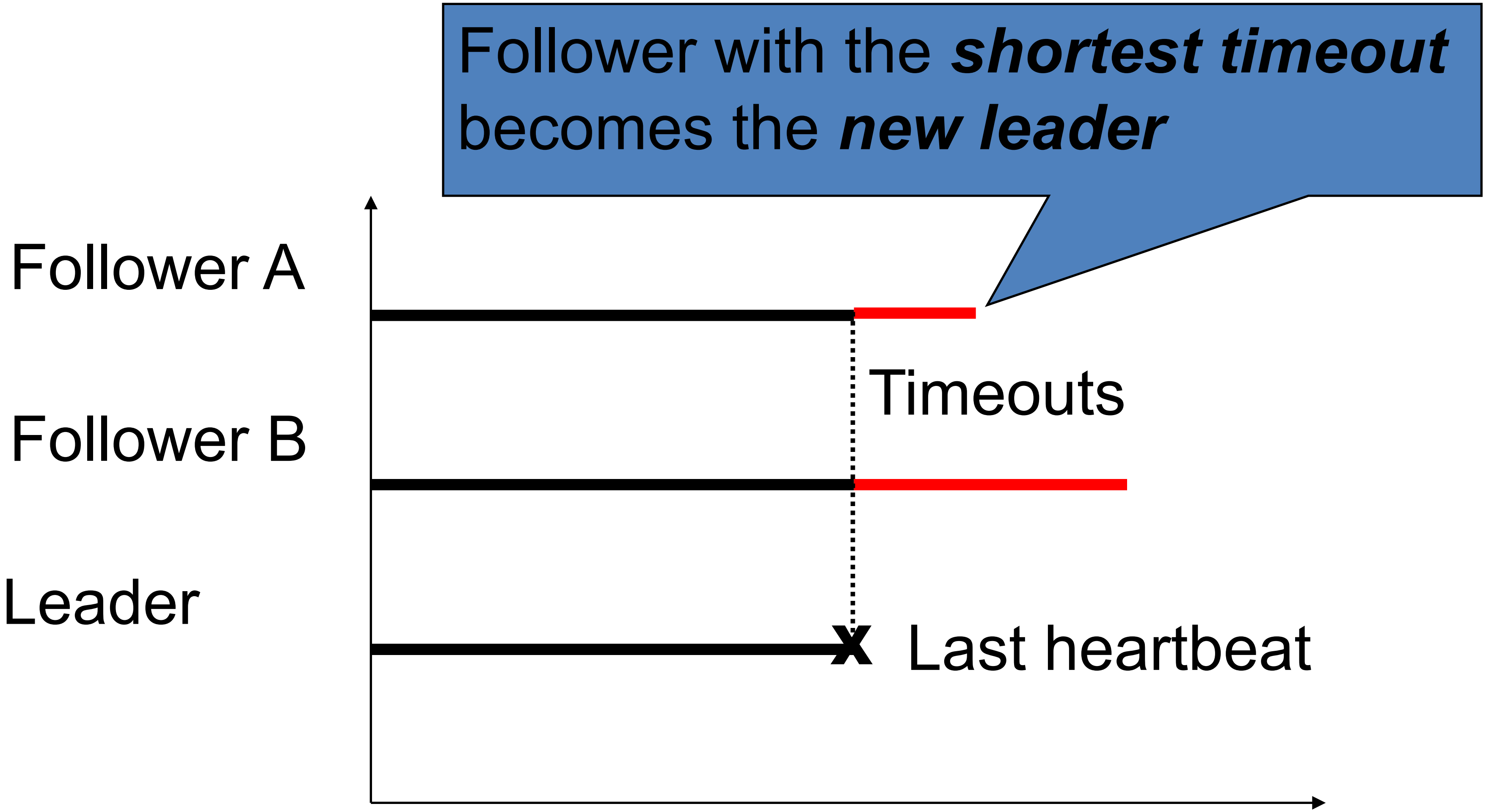
Avoiding split elections

- Raft uses randomized election timeouts
  - Chosen randomly from a fixed interval
- Increases the chances that a single follower will detect the loss of the leader before the others

Reacp: Key features of Raft

- ***Strong leader:***
  - Leader does most of the work:
    - Issues ***all*** log updates
- ***Leader election:***
  - Uses ***randomized timers*** to elect leaders.
- ***Membership changes:***
  - New ***joint consensus*** approach where the majorities of two different configurations are required

Example

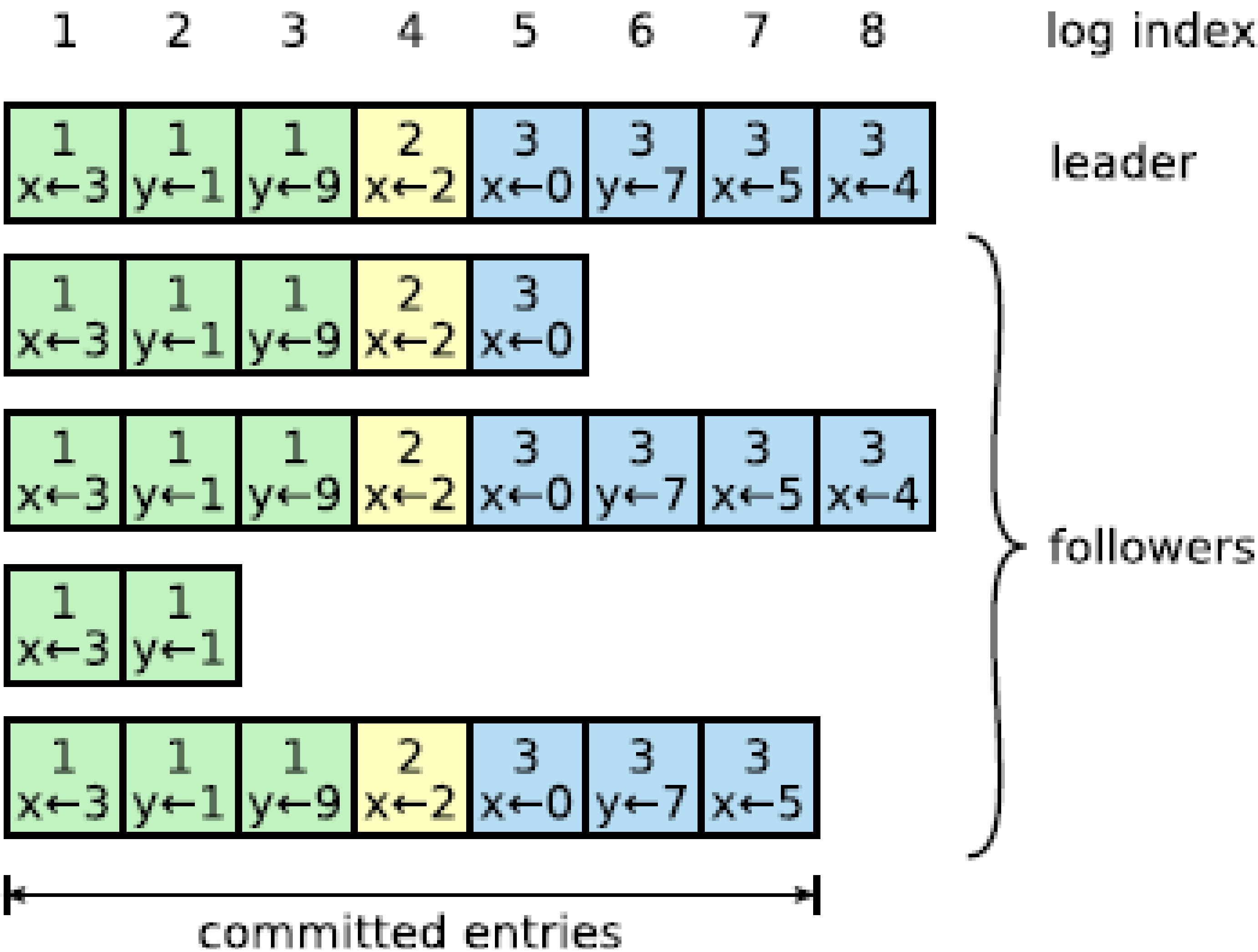


Follower with the ***shortest timeout*** becomes the ***new leader***
Follower A
Timeouts
Follower B
Leader
x
Last heartbeat

Log replication

- Leader
  - 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***

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 available state machines
- Committing an entry also commits all previous entries
  - All AppendEntry RPCs—including heartbeats—include the index of its most recently committed entry

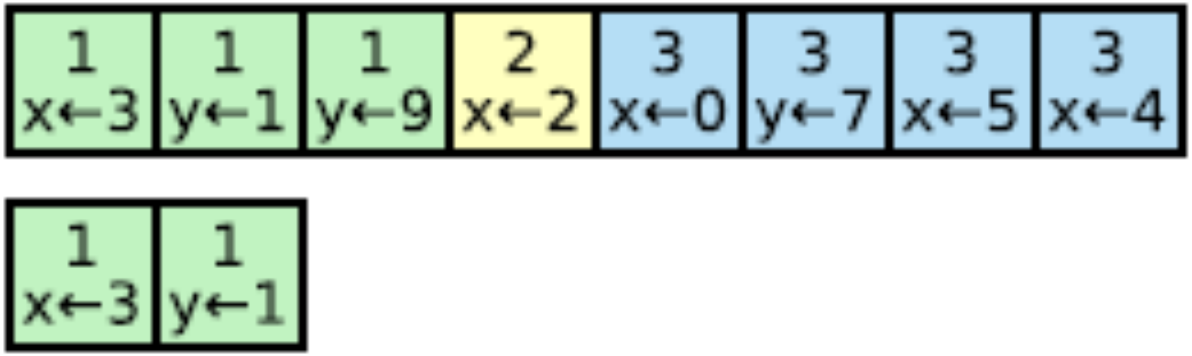
Why?

- Raft commits entries in ***strictly sequential order***
  - A log entry is committed once the leader that created the entry has replicated it on a majority of the servers
  - Requires followers to accept log entry appends in the same sequential order
    - ***Cannot "skip" entries***

Greatly simplifies the protocol

Raft log matching property

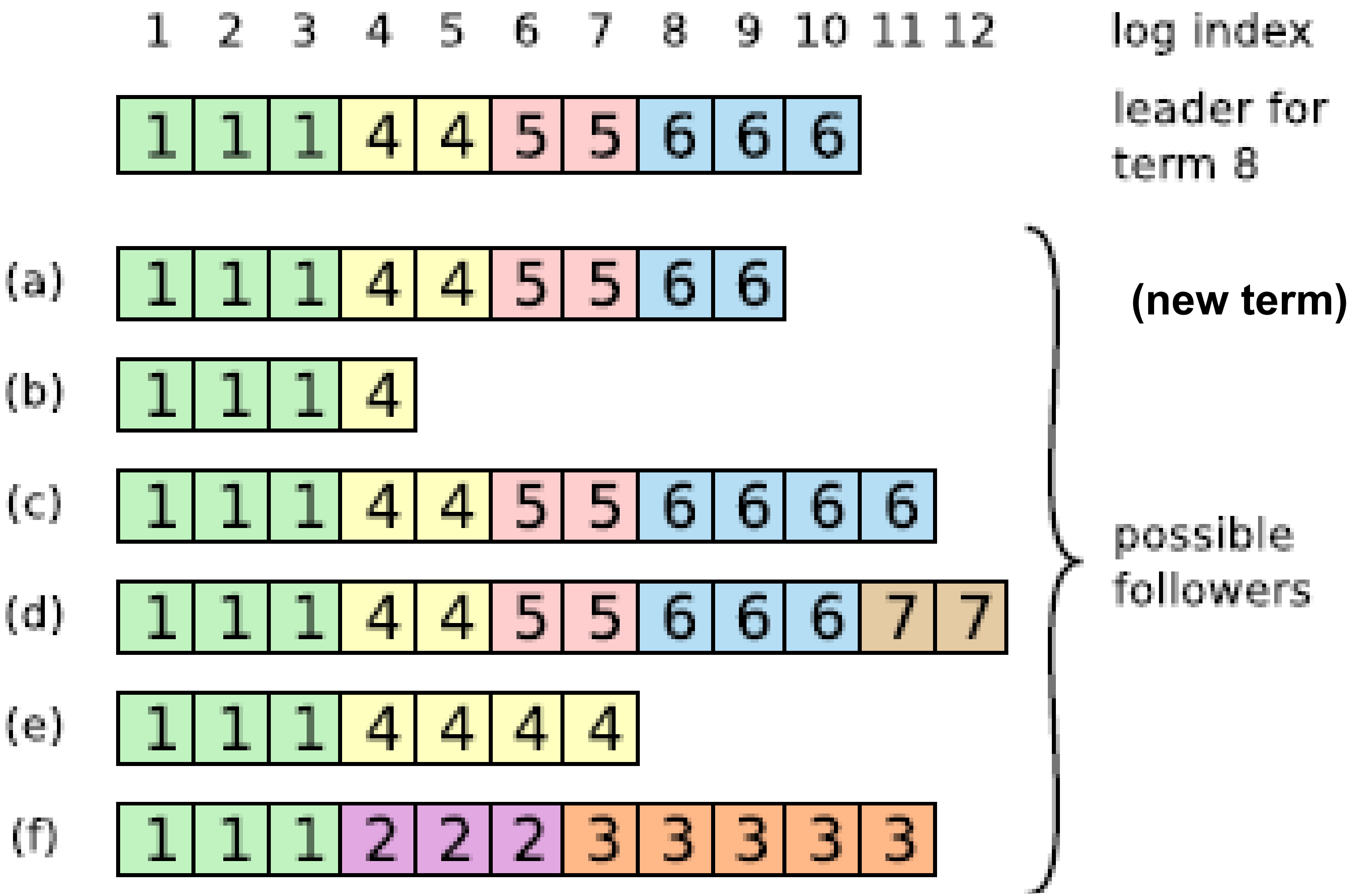
- If two entries in different logs have the same index and term
  - These entries store the same command
  - ***All previous entries*** in the two logs are ***identical***



Handling leader crashes (I)

- Can leave the cluster in a inconsistent state if the old leader had not fully replicated a previous entry
  - Some followers may have in their logs entries that the new leader does not have
  - Other followers may miss entries that the new leader has

Handling leader crashes (II)



Handling Crashes

- Missing entries: a-b
- Extra uncommitted entries: c-d
- Both: e-f

Handling leader crashes (IV)

- Raft solution is to let the new leader to force followers' log to duplicate its own
  - Conflicting entries in followers' logs will be ***overwritten***

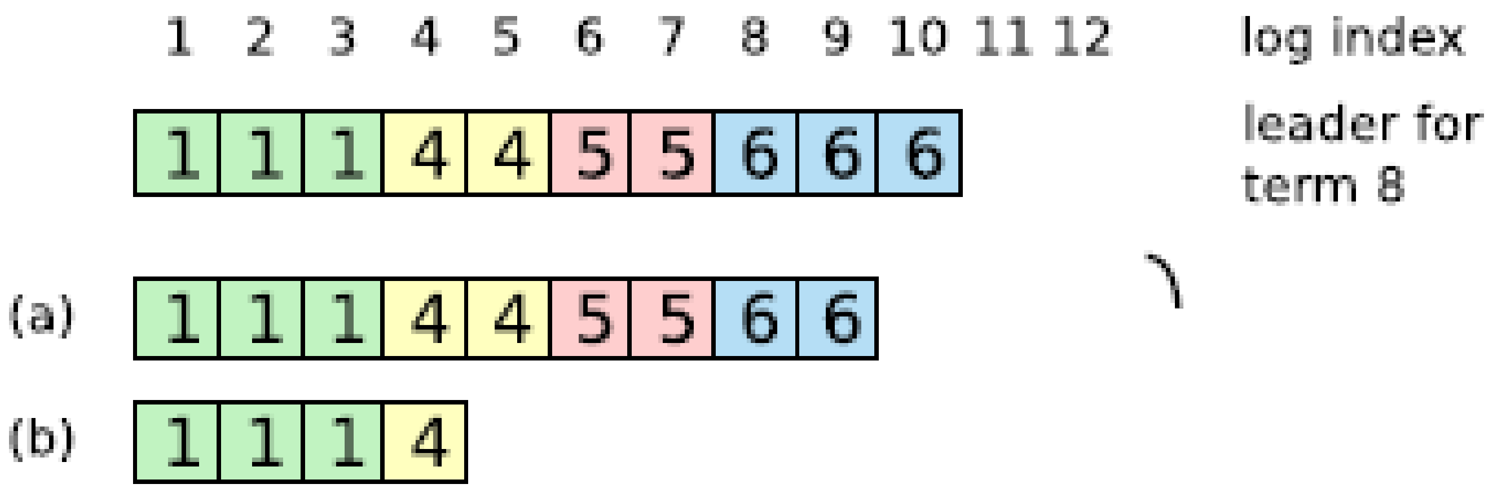
How? (I)

- Leader maintains a ***nextIndex*** for each follower
  - Index of entry it will send to that follower
- New leader sets its ***nextIndex*** to the index ***just after its last log entry***
  - 11 in the example
- Broadcasts it to all its followers

How? (II)

- Followers that have missed some AppendEntry calls will refuse all further AppendEntry calls – *consistency check*
- Leader will decrement its nextIndex for that follower and redo the previous AppendEntry call
  - Process will be repeated until a point where the logs of the leader and the follower **match**
- Will then send to the follower all the log entries it missed

How? (III)



- By successive trials and errors, leader finds out that the first log entry that follower (b) will accept is log entry 5
- It then forwards to (b) log entries 5 to 10

Interesting question

- How will the leader know which log entries it can commit
  – Cannot always gather a majority since some of the replies were sent to the old leader

- Fortunately for us, any follower accepting an AcceptEntry RPC implicitly acknowledges it has processed all previous AcceptEntry RPCs

Followers' logs cannot skip entries

A last observation

- Handling log inconsistencies does not require a special sub algorithm
  - Rolling back AppendEntry calls is enough

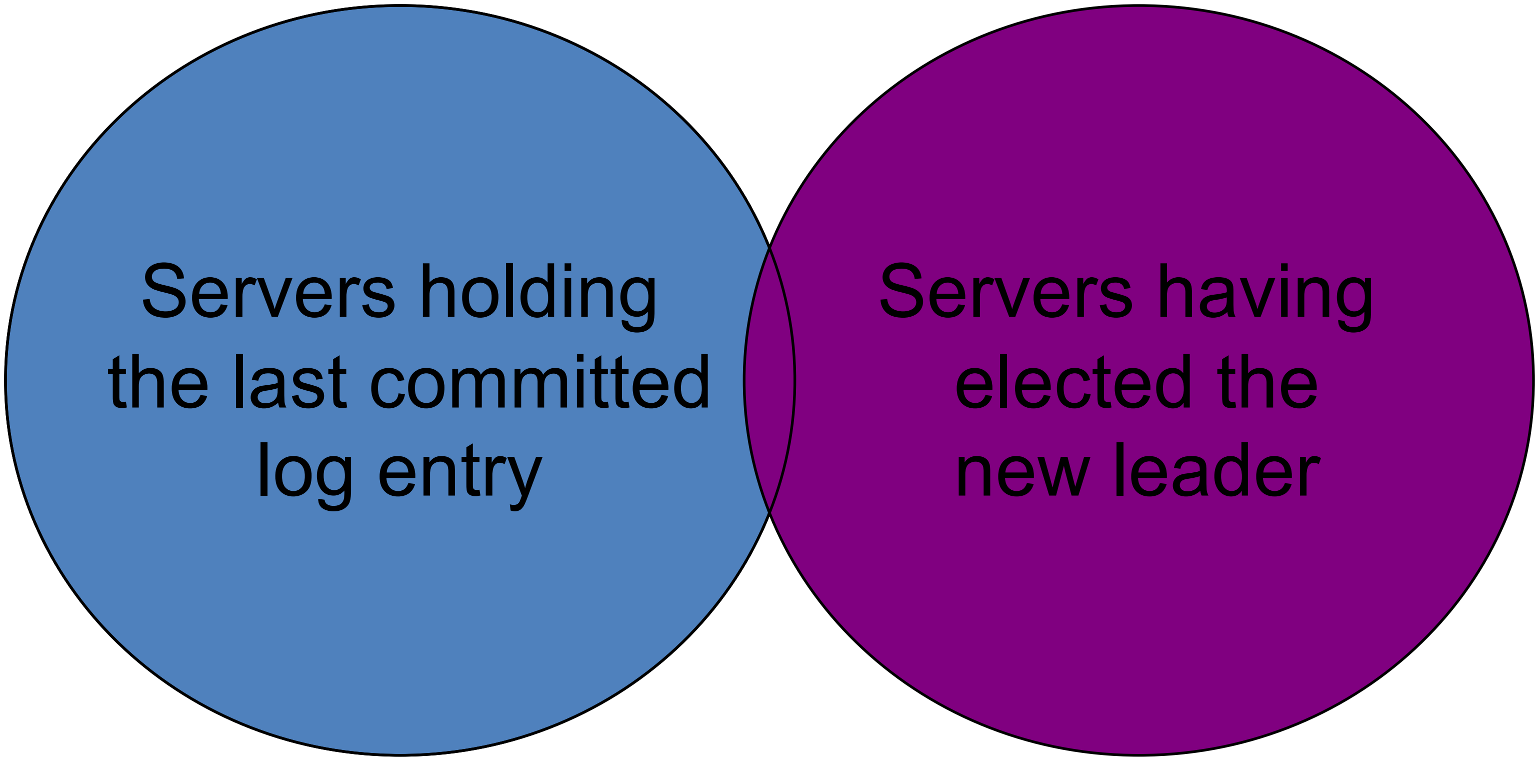
Safety

- Two main issues
  - What if the log of a new leader did not contain all previously committed entries?
    - Must impose conditions on new leaders
  - 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
    - *Details in the paper*
  - 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 a previous term

- A leader cannot immediately conclude that an entry from a previous term is committed even if it is stored on a majority of servers.
  - *See paper*
- Leader should never commit 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

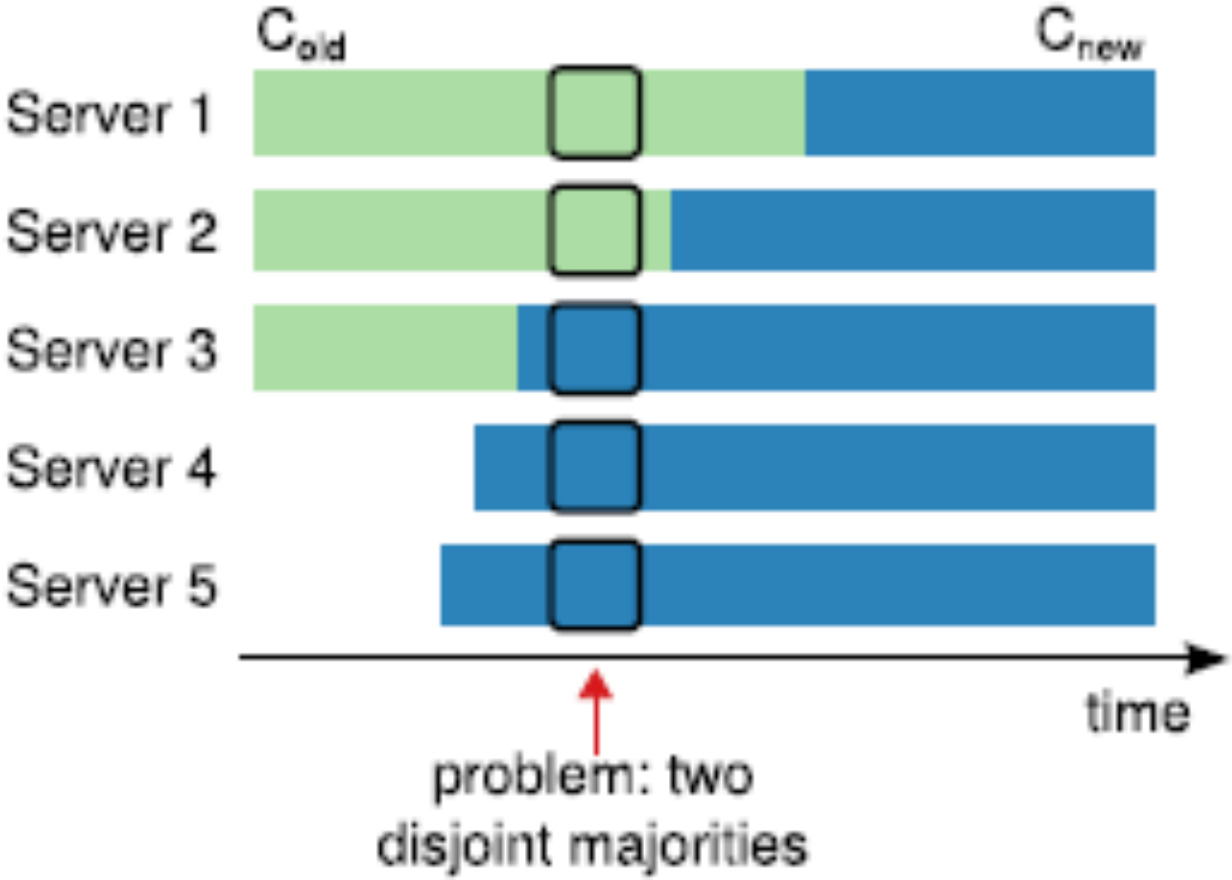
Cluster membership changes

- Not possible to do an atomic switch
  - Changing the membership of all servers at once
- Will use a two-phase approach:
  - Switch first to a transitional ***joint consensus*** configuration
  - Once the joint consensus has been committed, transition to the new configuration

The joint consensus configuration

- Log entries are transmitted to all servers, old and new
- Any server can act as leader
- Agreements for entry commitment and elections requires majorities from both old and new configurations
- Cluster configurations are stored and replicated in special log entries

The joint consensus configuration



Implementations

- Two thousand lines of C++ code, not including tests, comments, or blank lines.
- About 25 independent third-party open source implementations in various stages of development
- Some commercial implementations

- A good description of how Raft works:

  http://thesecretlivesofdata.com/raft/

Primary Backup Replication

- Primary/backup: ensure a single order of ops:
  - Primary orders operations
  - Backups execute operations in order

Case study: Hypervisor

Bressoud and Schneider. Hypervisor-Based Fault Tolerance. SOSP 1995

- Goal: fault tolerant computing
  - Banks, NASA etc. need it
  - CPUs are most likely to fail due to complexity
- Hypervisor: primary/backup replication
  - If primary fails, backup takes over
  - Caveat: assuming failure detection is perfect

Hypervisor replicates at VM-level

- Why replicating at VM-level?
  - Hardware fault-tolerant machines were big in 80s
  - Software solution is more economical
  - Replicating at O/S level is messy (many interfaces)
  - Replicating at app level requires programmer efforts
  - Replicating at VM level has a cleaner interface (and no need to change O/S or app)
- Primary and backup execute the same sequence of machine instructions

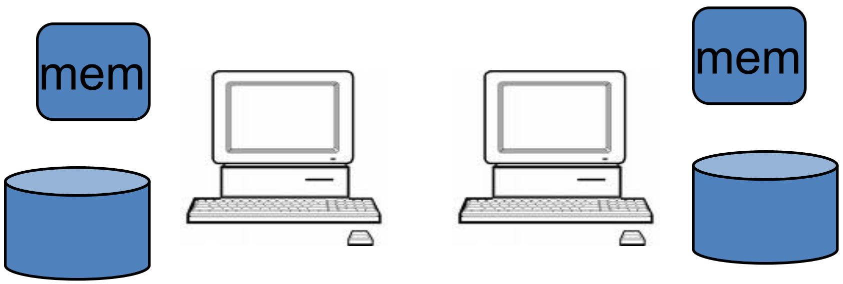
A Strawman design

- Two identical machines
- Same initial memory/disk contents
- Start execute on both machines
- Will they perform the same computation?

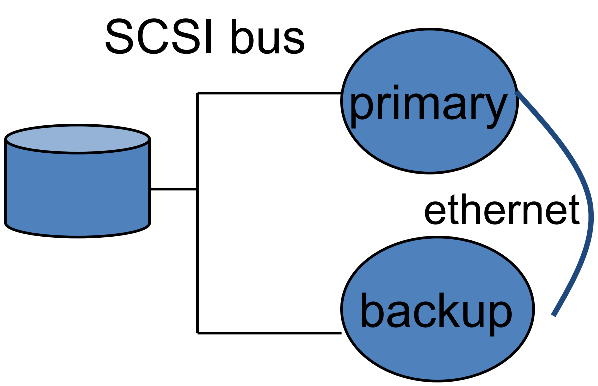
Strawman flaws

- To see the same effect, operations must be deterministic
- What are deterministic ops?
  - ADD, MUL etc.
  - Read time-of-day register, cycle counter, privilege level?
  - Read memory?
  - Read disk?
  - Interrupt timing?
  - External input devices (network, keyboard)

Hypervisor’s architecture



Strawman replicates disks
at both machines
Problem: disks
might not behave identically
(e.g. fail at different sectors)



Hypervisor connects devices to
to both machines

- Only primary reads/writes to devices
- Primary sends read values to backup
- Only primary handles interrupts from h/w
- Primary sends interrupts to backup

Hypervisor executes in epochs

- Challenge: must execute interrupts at the same point in instruction streams on both nodes
- Strawman: execute one instruction at a time
  - Backup waits from primary to send interrupt at end of each instruction
  - Very slow….
- Hypervisor executes in epochs
  - CPU h/w interrupts every N instructions (so both nodes stop at the same point)
  - Primary delays all interrupts till end of an epoch
  - Primary sends all interrupts to backup

Hypervisor failover

- If primary fails, backup must handle I/O
- Suppose primary fails at epoch E+1
  - In Epoch E, backup times out waiting for [end, E+1]
  - Backup delivers all buffered interrupts at the end of E
  - Backup starts epoch E+1
  - Backup becomes primary at epoch E+2

Hypervisor failover

- Backup does not know if primary executed I/O epoch E+1?
  - Relies on O/S to re-try the I/O
- Device needs to support repeated ops
  - OK for disk writes/reads
  - OK for network (TCP will figure it out)
  - How about keyboard, printer, ATM cash machine?

Hypervisor implementation

- Hypervisor needs to trap every non-deterministic instruction
  - Time-of-day register
  - HP TLB replacement
  - HP branch-and-link instruction
  - Memory-mapped I/O loads/stores
- Performance penalty is reasonable
  - A factor of two slow down
  - How about its performance on modern hardware?

Caveats in Hypervisor

- Hypervisor assumes failure detection is perfect
- What if the network between primary/backup fails?
  - Primary is still running
  - Backup becomes a new primary
  - Two primaries at the same time!
- Can timeouts detect failures correctly?
  - Pings from backup to primary are lost
  - Pings from backup to primary are delayed