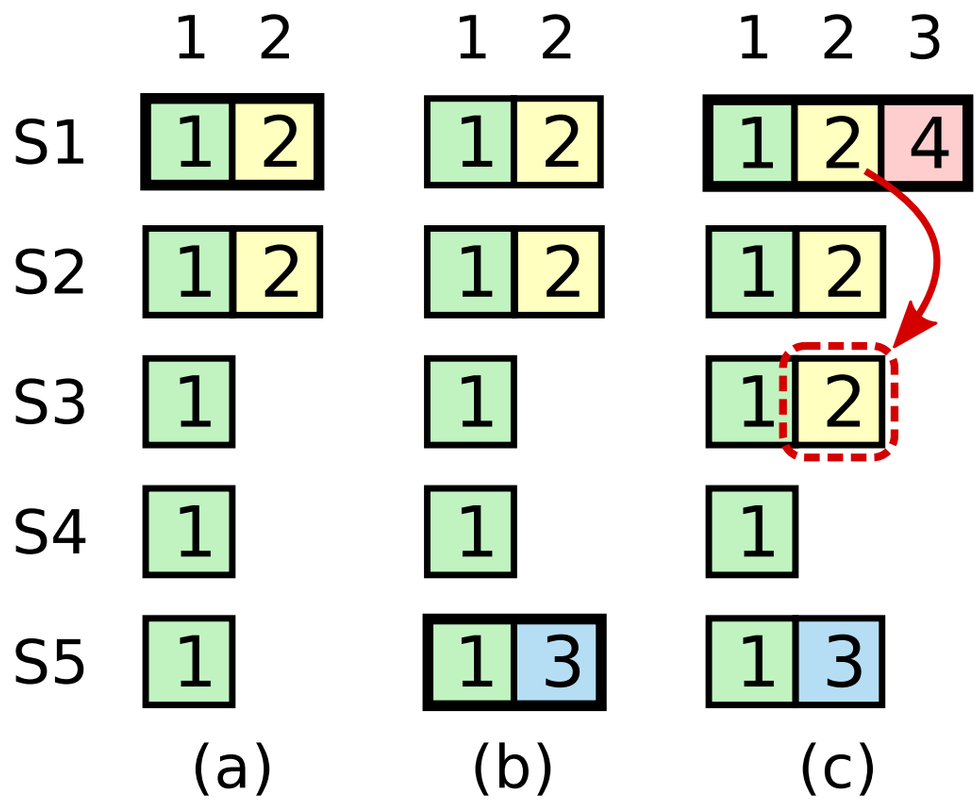
Distributed Systems

Spring Semester 2020

Lecture 7: Raft

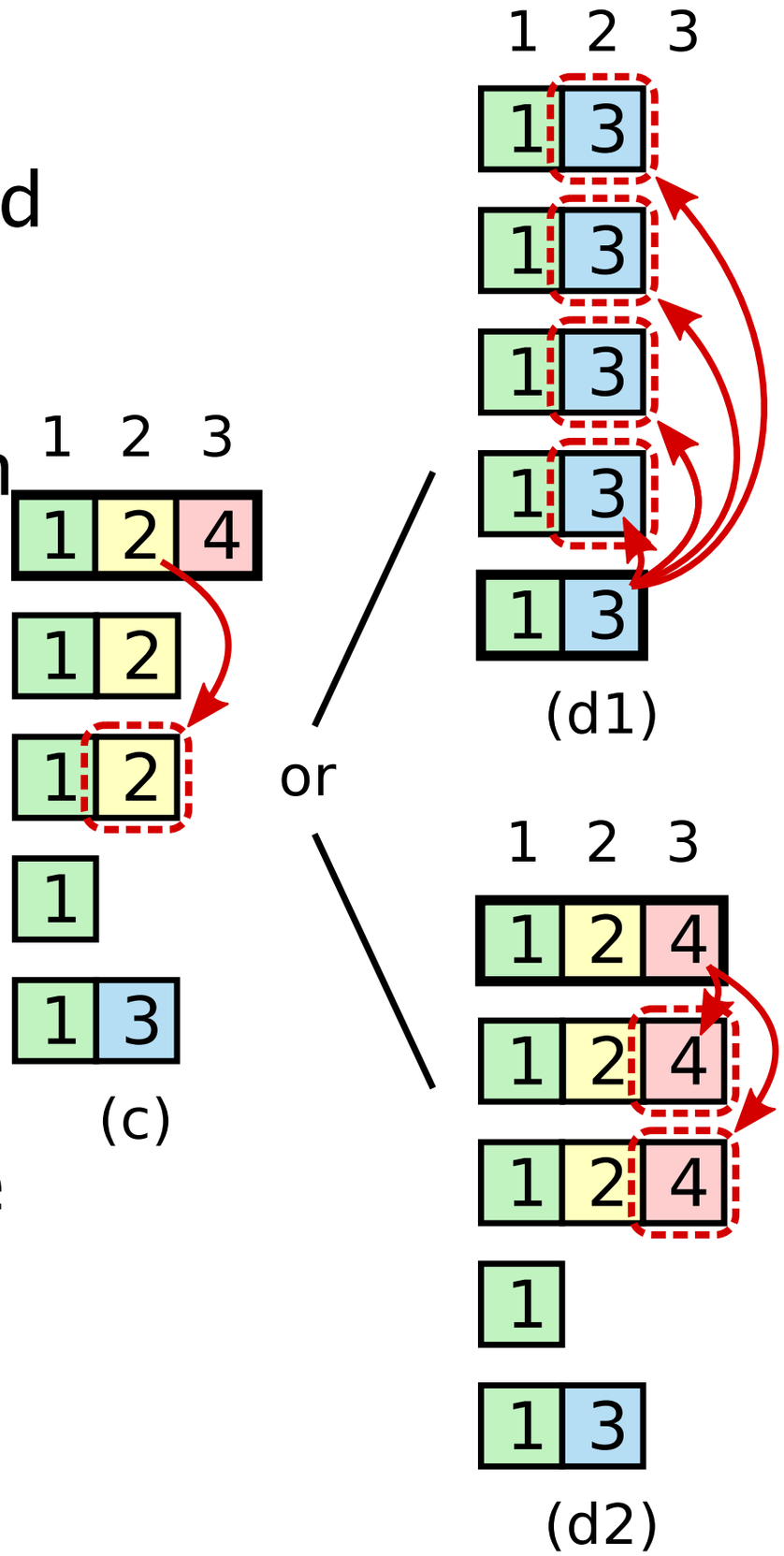
John Liagouris
liagos@bu.edu

A time sequence showing why a leader cannot determine commitment using log entries from older terms.



S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed.

If S1 crashes as in (d1), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (d2), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.



committed means that the entry:

1) reached a majority in the term it was initially sent out

OR

2) a subsequent log entry is committed

Assignment Note

- Quick Rollback: End of section 5.3
  - Given aggressive nature of failure testing you will likely need this:
  - “When rejecting an AppendEntries request follower can include the term of the conflicting entry and the first index it stores for that term. … the leader can decrement nextIndex to bypass all of the conflicting, rather than one RPC per entry”

Configuration Changes

- What happens when:
  - One ore more servers cannot recover and we need a replacement?
  - We want to scale out/in by adding/removing servers?

Configuration Changes

- "Stop the world" and restart
- Can we do better?
  - Two-phase approach: first phase disables old configuration, second phase enables new
  - Raft solution is "joint consensus": allows different servers to transition to the new configuration at different times

Joint Consensus

- First phase: both configurations are active
  - Two sets of servers that define separate majorities
  - Any server from either configuration can be a leader
  - Elections and Commits require majorities from both configurations
- Second phase: all servers have switched to the new configuration

Joint Consensus

- Cluster configurations are treated as special entries
- A server uses the latest (possibly joint) configuration in its log, even if it is not committed
- Once a configuration is committed, all servers must use this configuration

Example

Example

L

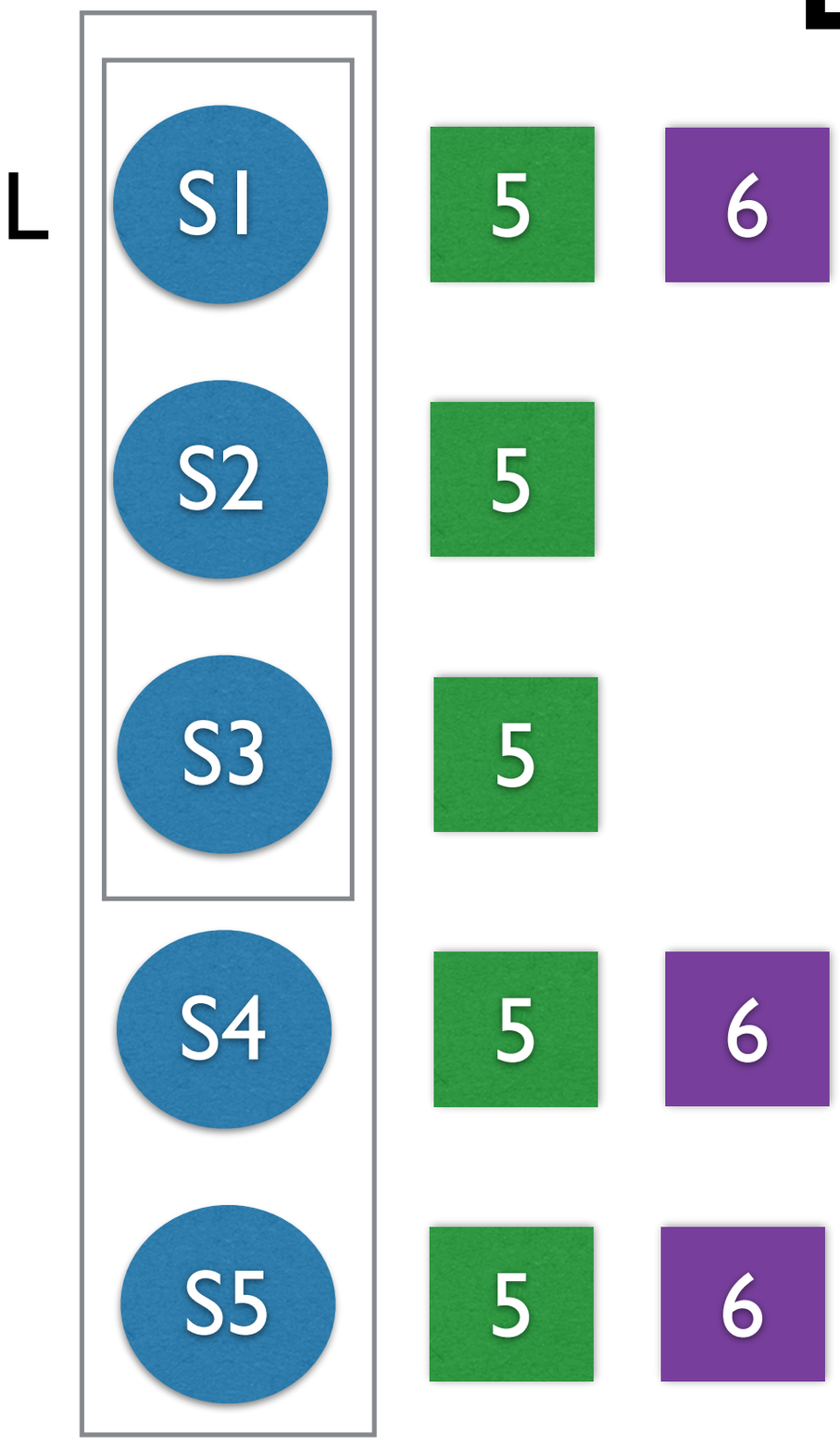
Example

Leader receives re-configuration command

Example

Leader sends RPCs to followers

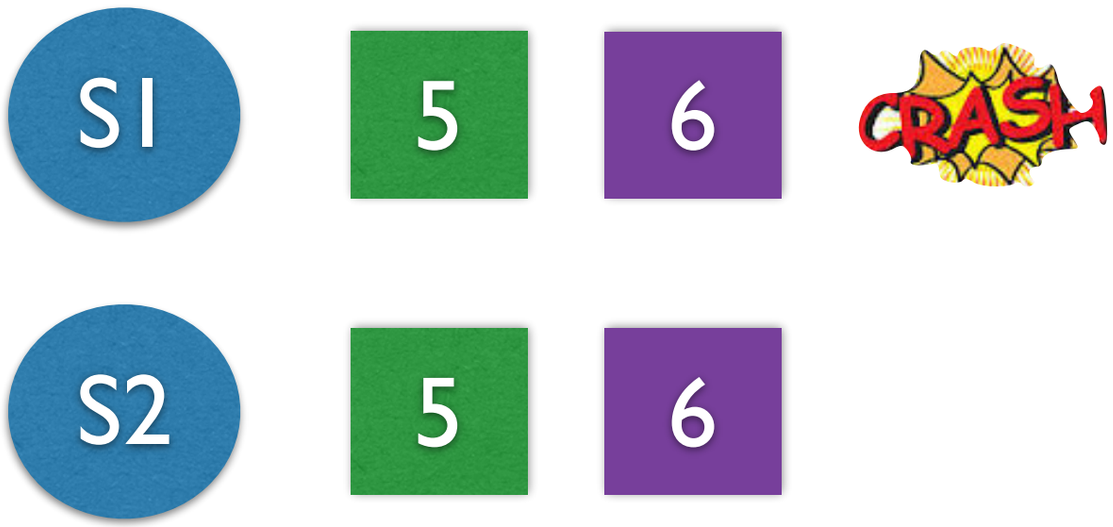
Example



Re-configuration command appears in the new majority, not the old

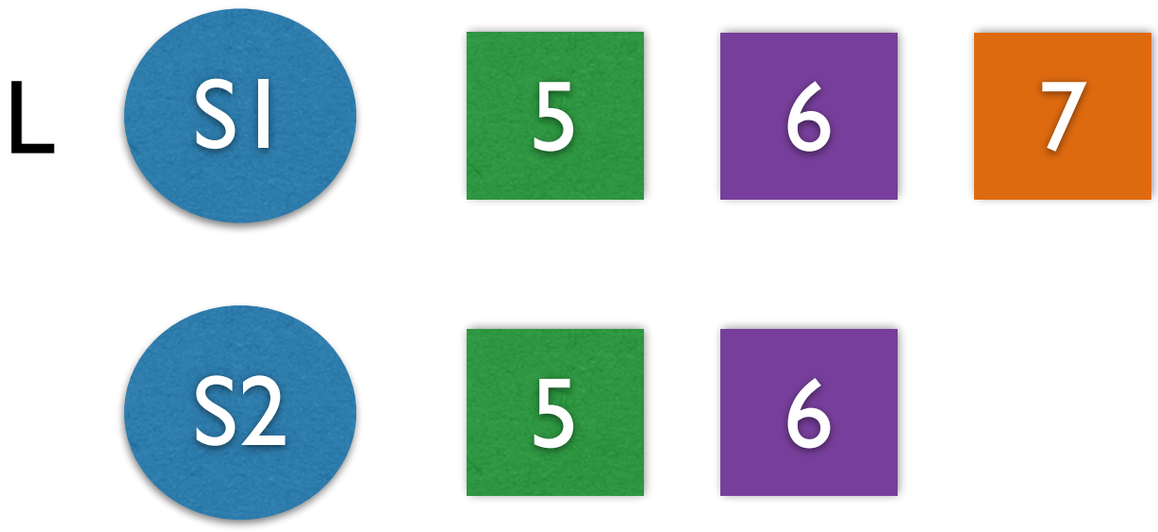
Example

L



Leader crashes again

Example



S1 wins the election with term 7 and receives a new client request

Example

Leader sends append RPCs to the servers

Example

Re-configuration command is committed

Persistence

- Persistent state (figure 2): currentTerm, votedFor, log[]
- Must be on "disk" so that on restart can be initialized to last known state

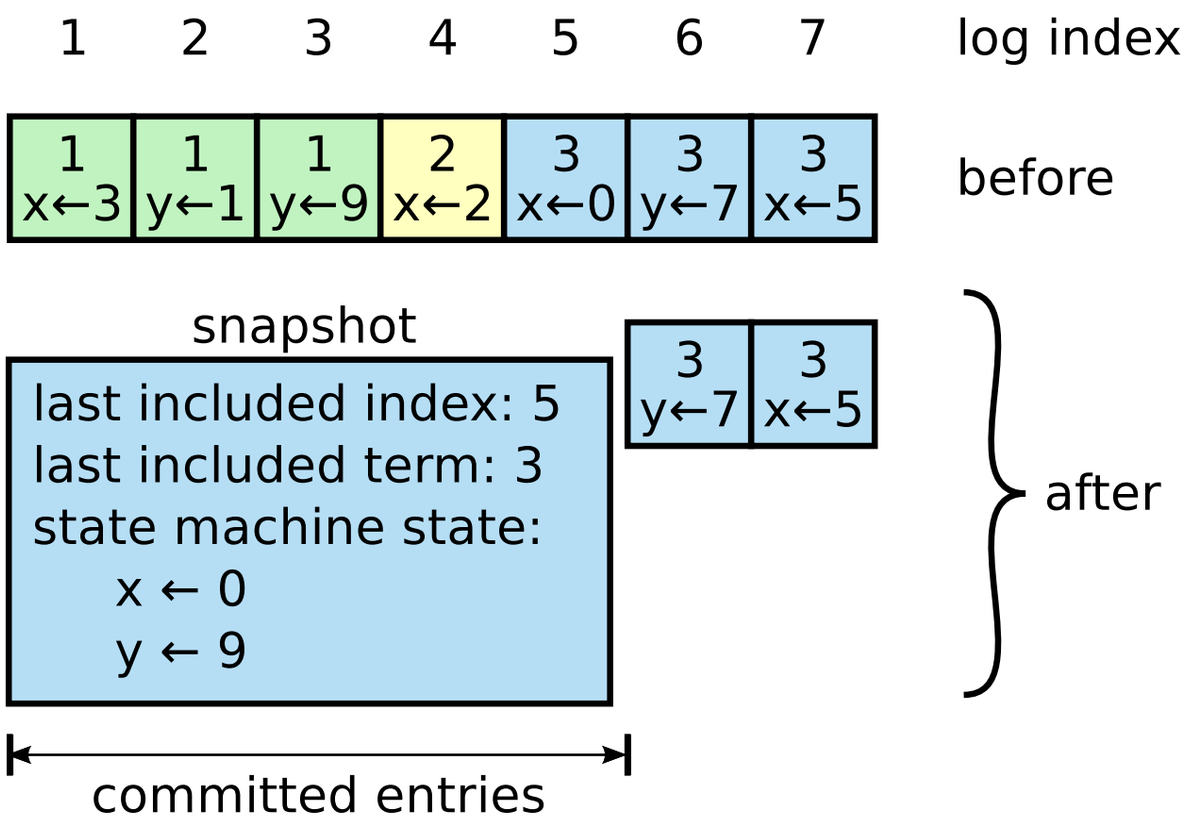
Persistence and Performance

- Disk : 10ms write, SSD 0.1 ms — RPC << 1ms (in datacenters)
- So if every op needs a disk write then limited to speed of write.
- Better performance
  - batch many client ops into each RPC (disk write)
  - faster storage non-volatile storage (battery backup RAM)

Log Compaction and snapshots

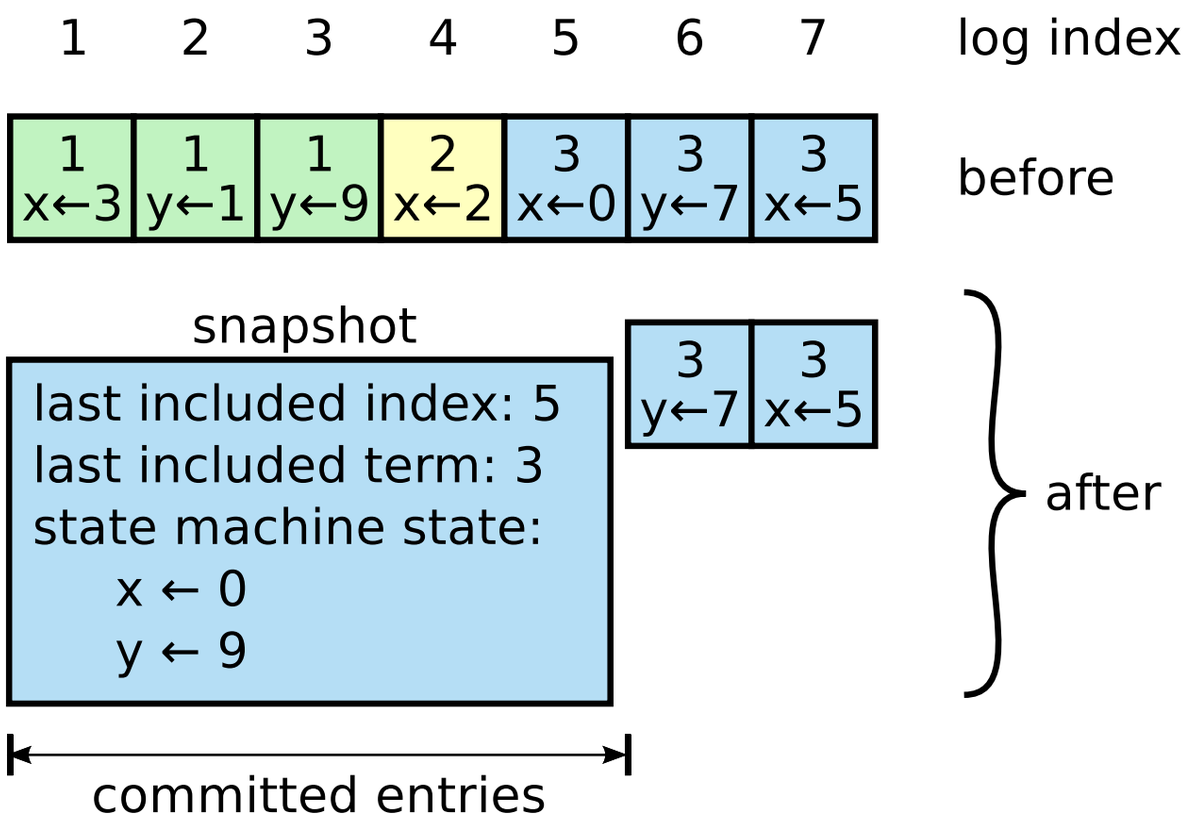
- Raft uses snapshotting to keep the size of the log bounded
- The entire current state is written to reliable storage as a snapshot
- Snapshots use copy-on-write so that the server can still reply to requests during snapshotting
- Snapshots can be periodic or based on an upper bound in the log size

Log Compaction and snapshots



Each server periodically
snapshots
to bound resource
consumption

Log Compaction and snapshots



Each server periodically snapshots to bound resource consumption

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

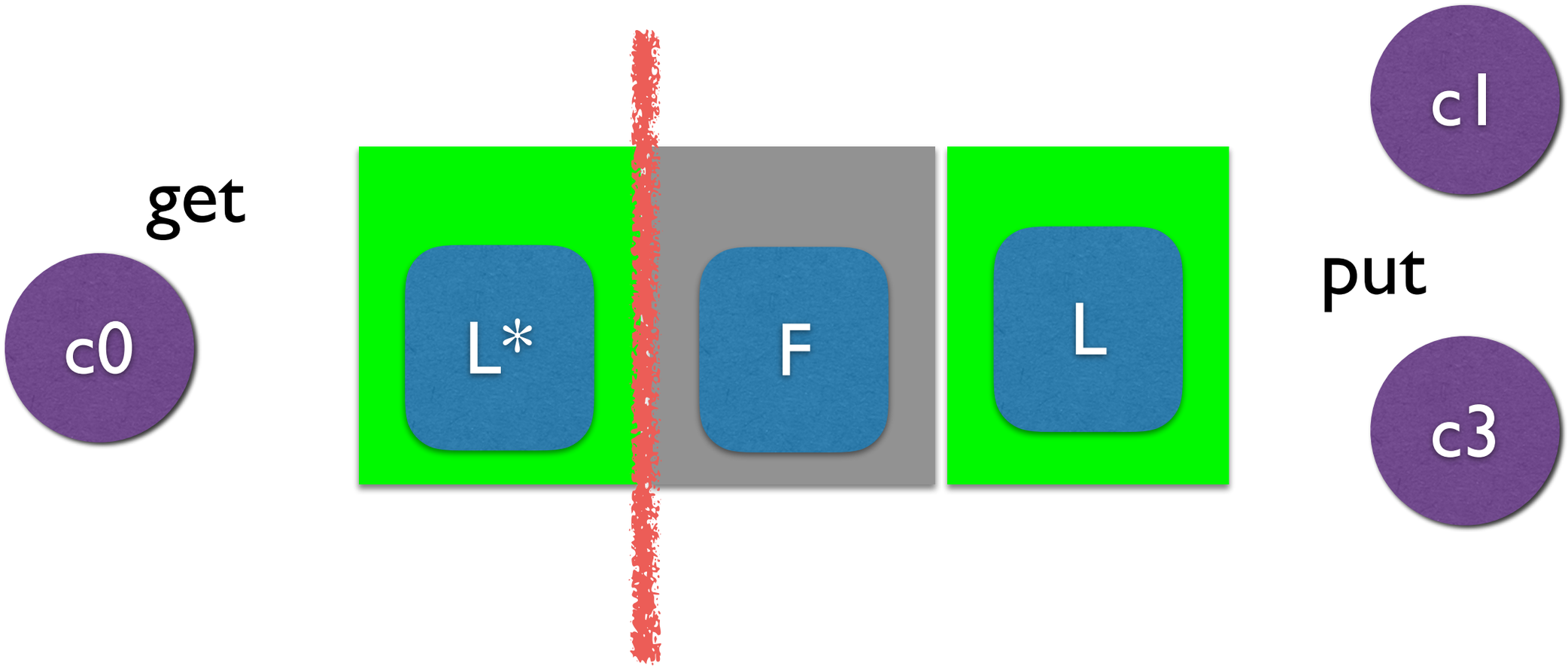
Figure 13: A summary of the InstallSnapshot RPC. Snapshots are split into chunks for transmission; this gives the follower a sign of life with each chunk, so it can reset its election timer.

Leader may need to use its snapshot to update stale clients

Client Behavior Section 8

- Client loops: retry until success — leads to duplicates on leader change
- Duplicate detection: Raft can send to service repeat commands
- **Service** required to keep history of commands to reject duplicates
  - replica must maintain a duplicate table (as leaders can change)

Important : read-only



If read's did not go to log and there is a partition we no longer need majority to reply so we do ;-) and why could this be a problem?

"read your last write"

Linearizability

- Raft supports linearizable semantics
- Clients cannot read stale data
- See also: http://www.bailis.org/blog/linearizability-versus-serializability/