6.824

wu

September 24, 2022

Contents

1 map reduce

1.1 programming model

the computation takes a set of **input** key/value pairs, and produces a set of **output** key/value pairs. The user of the MapReduce library expresses the computation as two functions: **Map** and **Reduce**

Map, written by the user, takes an input pair and produces a set of **intermediate** key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the **Reduce** function

The **Reduce** function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation.

1.1.1 example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1")

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    results += ParseInt(v)
  Emit(AsString(result))
```

The `map` function emits each word plus an associated count of occurrences. The `reduce` function sums together all counts emitted for a particular word

1.1.2 Types

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

1.1.3 More examples

Distributed Grep: the map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output

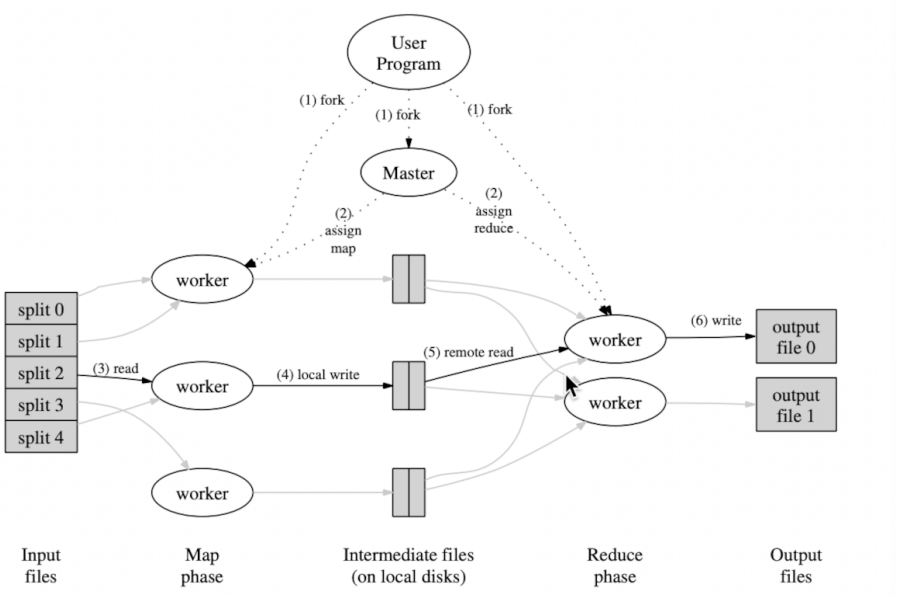
Count of URL Access Frequency: the map function processes logs of web page requests an outputs $\langle$`URL,1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL,total count`$\rangle$ pair

Term-vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname,term vector`$\rangle$ pair for each input document. The reduce function is passed all per-document term vectors for a given host. It add these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname,term vector`$\rangle$ pair

1.2 Implementation

1.2.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \mod R$). The number of partitions and the partitioning function are specified by the user



When the user program calls the `MapReduce` function, the following sequence of actions occurs

1. the MapReduce library in the user program first splits the input files into M pieces and starts up many copies of the program on a cluster of machines

2. one of the copies of the program is special - the master. The rest are workers that are assigned work by the master. there are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task

3. a worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory

4. periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers

5. when a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it **sorts** it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used

6. the reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition

7. when all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the *MapReduce* call in the user program returns back the user code

1.2.2 Master data structures

for each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), identity of the worker machine.

For each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed

1.2.3 Fault tolerance

1. worker failure The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.

   Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible

2. master failure It is easy to make the master write periodic checkpoints of the master data structures described above.

3. Semantics in the presence of failures

2 Raft paper

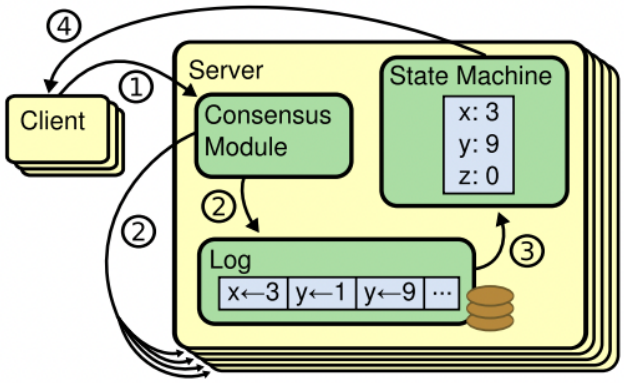
Raft is a consensus algorithm for managing a replicated log.

2.1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members

2.2 Replicated state machines

Replicated state machines are typically implemented using a replicated log, as shown in figure Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients.



Consensus algorithms for practical systems typically have the following properties:

- they ensure **safety** (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering
- they are fully functional as long as any majority of the servers are operational and can communicate with each other and with clients. Thus a typical cluster of five servers can tolerate the failure of any two servers
- they do not depend on timing to ensure the consistency of the logs
- in the common case, a command can complete as soon as a majority of the cluster has responded to a single round of remote procedure calls

2.3 The Raft consensus algorithm

link

Raft implements consensus by first electing a distinguished **leader**, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tell servers when it is safe to apply log entries to their state machines.

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

- **leader election**
- **log replication**

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 (§5.3); also used as heartbeat (§5.2).

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 (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
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 (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

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

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- 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 (§5.2):

- 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 RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- 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 (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

- **safty** : the key safety property for Raft is the State Machine Safety Property: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index
  - **Election Safety**: at most one leader can be elected
  - **Leader Append-Only**: a leader never overwrites or deletes entries in its log; it only append new entries
  - **Log Matching**: if two logs contain an entry with the same index and term, then logs are identical in all entries up through the given index
  - **Leader Completeness**: if a log entry is commited in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms
  - **State Machine Safety**: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index

2.3.1 Raft basics

At any given time each server is in one of three states: **leader**, **follower** or **candidate**. In normal operation there is exactly one leader and all of the other servers are followers.

- Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates.
- The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader)
- Candidate is used to elect a new leader

Raft divides time into **terms** of arbitrary length: Terms are numbered with consecutive integers. Each term begins with an **election**, in which one or more candidates attempt to become leader. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote, in this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term

Different severs may observe the transitions between terms at different times, and in some situations a server may not observe an election or even

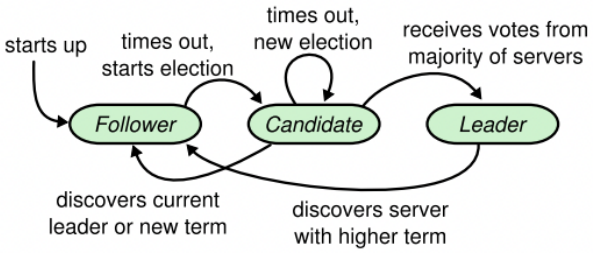


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

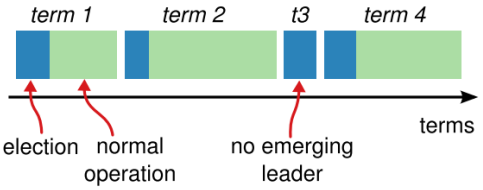


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

entire terms. Terms act as a logical clock in Raft, and they allow servers to detect obsolete information such as stable leaders.

Each server stores a **current term** number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of data, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

Raft servers communicate using remote procedure calls(RPCs), and the basic consensus algorithm requires only two types of RPCs. RequestVote RPCs are initiated by candidates during elections, and AppendEntries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat

2.3.2 Leader Election

Raft uses a heartbeat mechanism to trigger leader election.

When servers start up, they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the **election timeout**, then it assumes there is no viable leader and begins an election to choose a new leader

To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs **in parallel** to each of the other servers in the cluster. A candidate continues in this state until one of three things happens:

1. it wins the election
2. another server establishes itself as leader
3. a period of time goes by with no winner

A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis (5.4 adds an additional restriction on votes). The majority rules ensures that at most one candidate can win the election for a particular term (the **Election Safe Property**) Once a candidate wins the election, it becomes leader. It then

sends heartbeat messages to all of the other servers to establish its authority and prevent new elections

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate's current term, then the candidate rejects the RPC and continues in candidate state.

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.

Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150-300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends single server will time out.

The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election;

2.3.3 Log replication [15/15]

Check this for some explanation.

Each client request contains a command to be executed by the replicated state machines.

1. ☒ The leader appends the command to its log as a new entry, then issues AppendEntries RPCs in parallel to each of the other servers to replicate the entry.

2. ☒ When the entry has been safely replicated, the leader applies the entry to its state machine and returns the result of that execution to the client.

3. ☒ If followers crash or run slowly, or if network packets are lost, the leader retries AppendEntries RPCs indefinitely (even after it has responded to the client) until all followers

eventually store all log entries.

Los are organized as below Each log entry stores a state machine com-

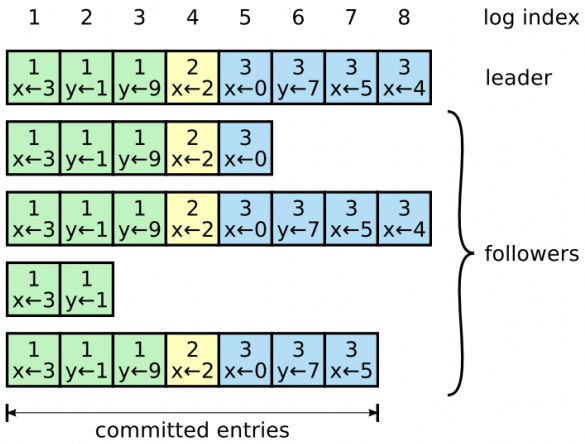


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

mand along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties. Each log entry also has an integer index identifying its position in the log

The leader decides when it is safe to apply a log entry to the state machines; such an entry is called **committed**.

- ☒ A log entry is committed once the leader that created the entry has replicated it on a majority of the servers. This also commits all preceding entries in the leader's log, including entries created by previous leaders.
- ☒ The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out.
- ☒ Once a follower learns that a log entry is committed, it applies the entry to its local state machine

Raft maintains the following properties, which together constitute the Log Machine Property:

- If two entries in different logs have the same index and term, then they store the same command
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries

The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log.

The second property is guaranteed by a simple consistency check performed by AppendEntries.

1. ⊠ When sending an AppendEntries RPC, the leader includes the index and the term in its log that immediately precedes the new entries.
2. ⊠ If the follower does not find an entry in its log with the same index and term, then it refuses the new entries

The consistency check acts as an induction

During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails.

However, leader crashes can leave the log inconsistent,

In Raft, the leader handles inconsistencies by forcing the follower' log to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log

To bring a follower's log into consistency with its own, the leader must

1. ⊠ find the latest log entry where the two logs agree
2. ⊠ delete any entries in the follower's log after that point
3. ⊠ send the follower all of the leader's entries after that point

All of these actions happen in response to the consistency check performed by AppendEntries RPCs.

The leader maintains a `nextIndex` for each follower, which is the index of the next log entry the leader will send to that follower

1. ⊠ When a leader first comes to power, it initializes all `nextIndex` values to the index just after the last one in its log (11 in figure)
2. ⊠ If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC.

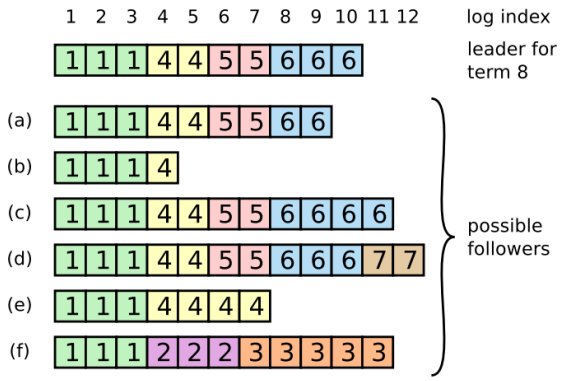


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

3. ☒ After a rejection, the leader decrements `nextIndex` and retries the AppendEntries RPC. Eventually `nextIndex` will reach a point where the leader and follower logs match.

4. ☒ When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log.

Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term

If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs.

- When rejecting an AppendEntries request, the follower can include the term of the conflicting entry and the first index it stores for that term. With this information, the leader can decrement `nextIndex` to bypass all of the conflicting entries in that term

- One AppendEntries RPC will be required for each term with conflicting entries, rather than one RPC per entry

2.4 Safety

2.4.1 election restriction

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries.

A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log in that majority, then it will hold all the committed entries.

The RequestVote RPC implements this restriction: the RPC includes information about candidate's log, and the voter denies its vote if its own log is more up-to-date than that of the candidate

Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs.

2.4.2 Committing entries from previous terms

- A leader knows that an entry from its current term is committed once that entry is stored on a majority of the serves.

- If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry.
- However, a leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers

Below is an illustration:

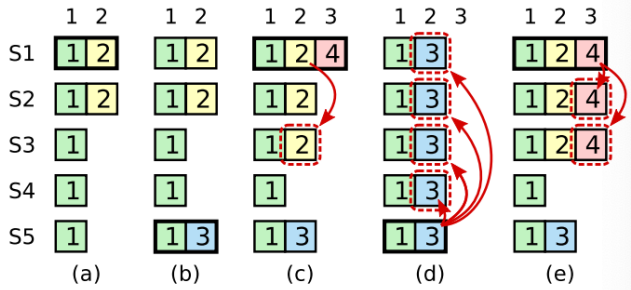


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) 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 (d), 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 (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

To eliminate problems like the one in figure, Raft never commits log entries from previous terms by counting replicas. Once an entry from the current term has been committed, then all prior entries are committed indirectly because of the Log Machting Property

2.4.3 Safety argument

We argue the Leader Completeness Property here (if a log entry is commited in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms)

We assume that the Leader Completeness Property does not hold, then we prove a contradiction.

Suppose the leader for term T (leader_T) commits a log entry from its term, but that log entry is not stored by the leader of some future term. Consider the smallest term U>T whose leader (leader_U) does not store the entry