Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
and **Dr. Anna Arpaci-Dusseau** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

0.1 Raft: Consensus Replication

As we've discussed so far, replication consistency is crucial and a difficult task. The next strategy, **Raft**, deals with such problem. There are other solutions, however, Raft is considered safe and easier to implement correctly than other solutions. First we must familiarize ourselves with the following terms:

Definition 1.1: State Machines in Distributed Systems

A **State Machine** processes sequence of inputs from a **log** and saves them in state. **Repli-**

cated state machines are implemented via **replicated logs** across multiple servers utilizing a **Consensus Algorithm**, validating log order. Such commands need be **deterministic**.

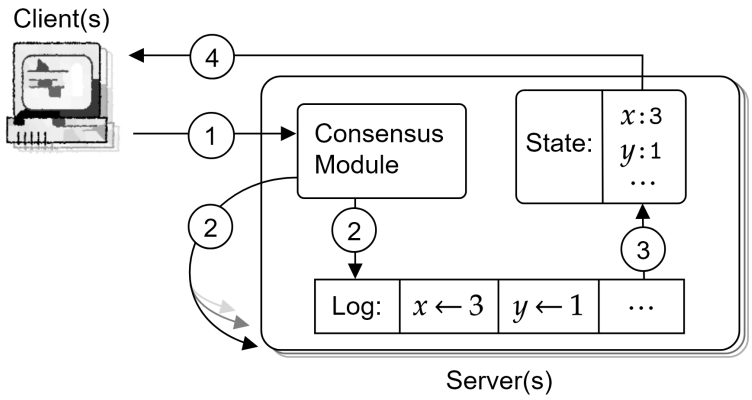

Figure 0.1: High-level framework for replicated state machines.

In the above the **Consensus Module** communicates with other servers to serve consistent logs.

0.1.1 Procedure Outline: Heartbeats, Elections, & Log Replication

Now to define the parts which make up a consensus algorithm:

Definition 1.2: Consensus Algorithm Components

A **Consensus Algorithm** involves the following components:

- **Safety**: Always returns correct results in spite, network delays, partitions, duplications, and reorderings.
- **Liveness**: A **Cluster** (group of servers) must tolerate a subset of server failures (e.g., A cluster of 5 servers can tolerate 2 failures). Offline servers may later recover and rejoin the cluster.
- **Time Agnostic**: The algorithm must not rely on synchronized clocks.
- **Majority Rule**: A majority of servers must agree on a value before it is committed. Minority slow servers must not block the system.

At a high level, The Raft Algorithm:

Definition 1.3: Raft Abstract

The Raft Algorithm involves three main components:

- **Leader Election**: A leader ℓ is elected to manage the replication process of backups β.
- **Heartbeats**: Where ℓ and β exchange consistent pulses of data to ensure liveliness.
- **Assurance**: Commit points (**??**) are established between the client, ℓ, and β.

In Raft, servers are given roles to manage the replication process.

Definition 1.4: Raft Server States

A Raft server can be in one of the following states:

- **Follower**: A server that listens to the leader.
- **Candidate**: A server that is running for leader.
- **Leader**: A server that is managing the replication process.

Followers are passive and simply listen to the leader.

Definition 1.5: Raft Heartbeats

Raft servers exchange **heartbeats** to ensure liveliness. Each server on boot is randomly assigned a timeout value. After the initial timeout, a server sends a signal akin to the out-and-in beats of a heart. **Out-beats** are origin signals, and **in-beats** are return signals. If a server β receives an out-beat from another server ℓ before its in-beat, the β timer resets and follows ℓ's beat. The leader of the beat's timer is not used.

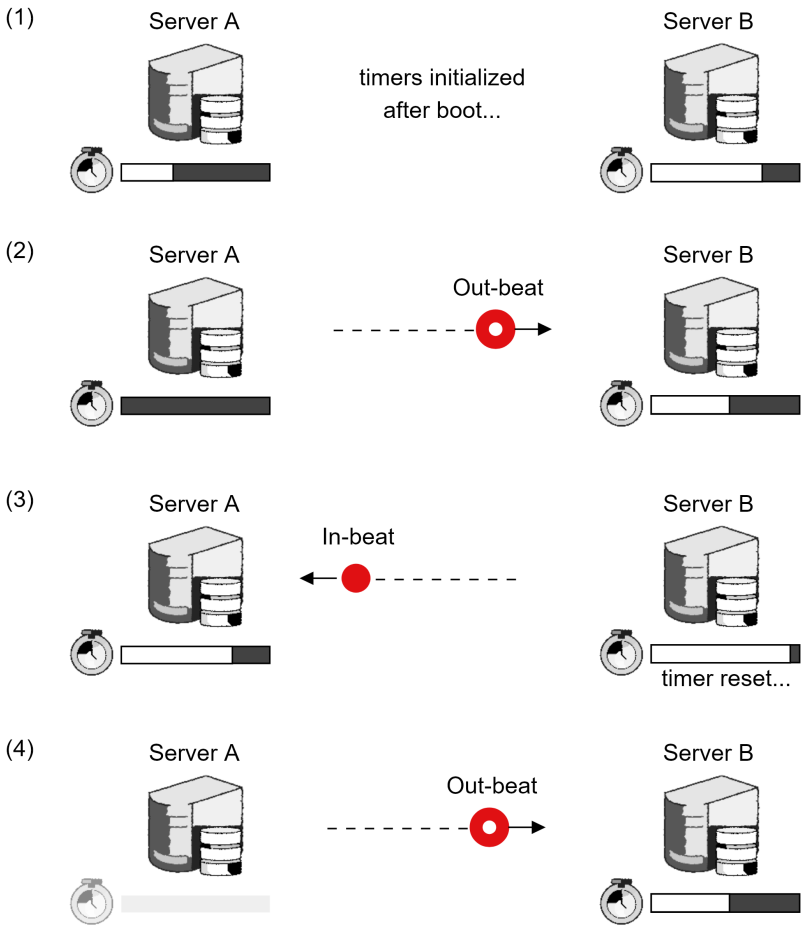

Figure 0.2: Server A and B exchanging heartbeats after system initialization (1). Server A's timer runs out first, sending the first signal to B (2). B has received A's out-beat, and now follows them. B resets their timer, and returns the signal to A (3). A's timer isn't effect, **nor is it used** whilst a leader of the beat (4).

Definition 1.6: Raft Election Terms

Let us denote an arbitrary server as β, then β is either of class γ (**Follower**), ς (**Candidate**), or ℓ (**Leader**). I.e., we have types $\{\beta : \gamma \mid \varsigma \mid \ell \}$ The Raft Election Process involves the following:

1. **Timeout ($\gamma \to \varsigma$):** First, all β initialize as γ with a random timer (e.g., 150-300ms). Once the timer runs out, γ transitions to ς.
2. **Candidate Election ($\varsigma \to \ell$):** ς votes for itself and sends a **RequestVote RPC** to all β, informing them of the new term and approval request. All β vote once per term for the RPC they received first. If ς receives the majority vote, it transitions to ℓ.
3. **Split Vote:** If all top ς tie in votes, all ς timeout, waiting for the next term. At this point, if a γ's timer runs out before the ς turn-around, then $\gamma \to \varsigma$ starting a new term election.
4. **Behind Leaders & Candidates ($\ell \to \gamma, \varsigma \to \gamma$):** If a ℓ or ς ever receives a heartbeat from a higher term β, it reverts to γ. Additionally, lower term signals are **ignored**.
5. **Leader Timeout ($\gamma \to \varsigma$):** If γ does not receive a heartbeat from the ℓ (server failure) whilst their timer runs out, they transition to ς and start a new election.

Let's observe a cluster of three servers:

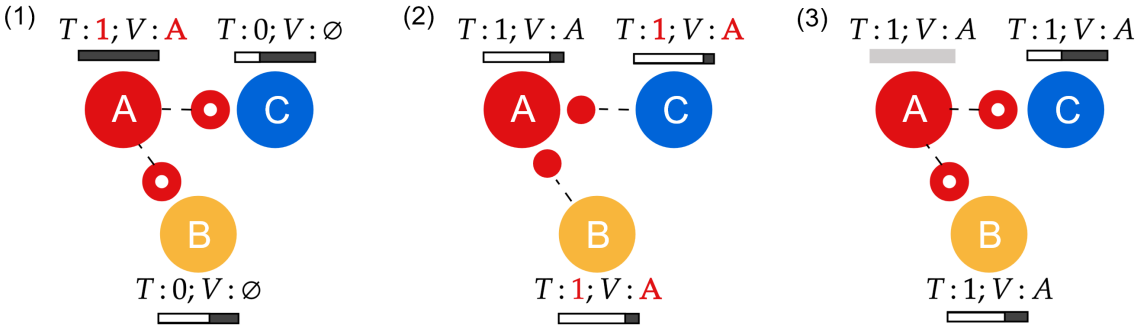

Figure 0.3: First Election after boot where A, B, and C nodes in a cluster of three have been initialized. (1) A times out first, becoming a Candidate and sending a RequestVote RPC to B and C. (2) B and C vote for A. (3) A receives the majority vote and becomes the Leader. Now A's timer is no longer used.

Observe a cluster of four who is dealing with a split vote:

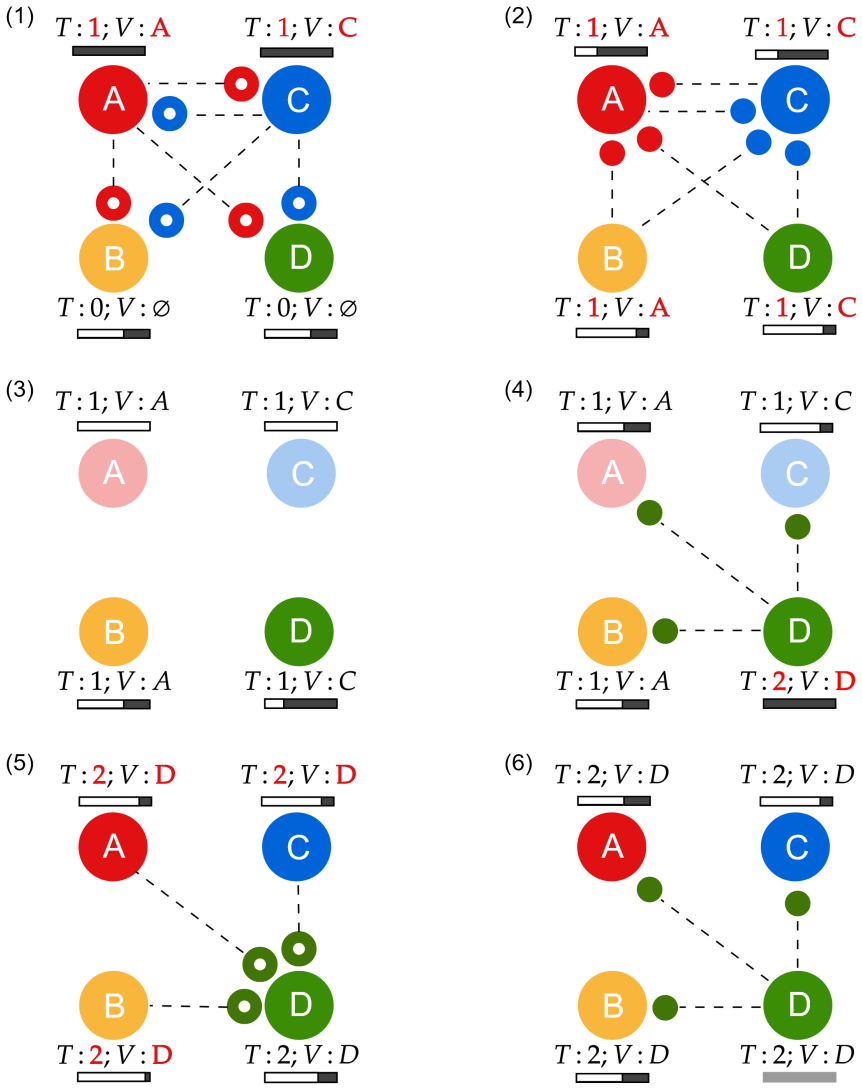

Figure 0.4: (1) A and C's timers run out first, sending out a RequestVote RPC to all other servers. (2) B votes for A, and D votes for C. (3) A and C timeout as per split vote. (4) D times out starting a new term, casting their Request RPC. (5) All servers respond to the higher order term. (6) D becomes the Leader (Their time no longer used).

Definition 1.7: Log Replication

Given { β (Server) : γ (Follower)| ς (Candidate)| ℓ (Leader) }, the Raft Log Replication Process involves the following:

- **Leader Log Replication:** Once a ℓ is elected, it services command requests from the client. The ℓ first appends the command to its **indexed log**, then sends an **AppendEntries RPC** in parallel to all β. Once a majority of β have replicated the log, the command is **committed**, and thus, safe to apply to their state machine.
- **Order of Execution:** The ℓ maintains a `nextIndex[]`, indicating the next expected log entry for all β. The ℓ sends missing log entries to lagging β; otherwise, it sends empty **AppendEntries RPCs** as heartbeats.
- **Leader Redirection:** All β that receive client requests, redirect the client to the ℓ.

Theorem 1.1: Log Matching Property

If two entries in different logs have the same index and term, then:

- They store the same command.
- All proceeding entries are the same.

Definition 1.8: Log Correction

Given { β (Server) : γ (Follower)| ς (Candidate)| ℓ (Leader) },

If ℓ fails, the new ℓ may be missing log entries. Let i denote the last index of ℓ's log entry, and j the last index of other βs' logs. If after an **AppendEntries RPC**, β's $j \neq i$, then ℓ decrements β's **nextIndex** to $(j-1)$ **per call**, until $j = i$. Once majority coherence is met, the log is committed.

This holds as we assume the preceding Theorem (1.1) is true. This is to say, the leader has a—**Append Only Property**—that it does not modify its own entries, but **only** appends new ones.

Note: A slight optimization for log correction, is for γ to tell ℓ the conflicting term and its first index. With that, ℓ can skip the log entries of such term. Though this isn't necessary as in real world applications, these conflicts are infrequent.

0.1.2 Safety: Restricting Leader Election

Since previously discussed, a new leader may be missing log entries, from which they tell other servers to discard mismatches. This can become problematic when the previous leader and other servers have committed several entries.

To fix this, we introduce constraints on leader election:

Definition 1.9: Leader Completeness

All proceeding leaders must have all committed entries of the previous leader.

Definition 1.10: Leader Election Restriction

Given { β (Server) : γ (Follower)| ς (Candidate)| ℓ (Leader) },
New ℓ must demonstrate **Leader Completeness**. To ensure this, β upon receiving a **RequestVote RPC** from ς checks:

- If ς's last log is from a lower term, the vote is **rejected**.
- If ς's last log is from the same term, but shorter, the vote is **rejected**.
- If ς's log is **at least** as up-to-date as the voter, the vote is **accepted**.

Theorem 1.2: Present Term Commitment

Leaders cannot assume previous term entries are committed, as they may not have been fully replicated. Hence they only commit entries from their term, and by the **Log Matching Property** (1.1), previous entries are also committed.

Theorem 1.3: Ensuring a Leader Through Timing

To ensure that there is always a leader in the face of probabilistic timing, these magnitude requirements should be satisfied:

$$broadcastTime \ll electionTimeout \ll MTBF$$

$broadcastTime$ measures the average time of heartbeat reciprocation for a server (e.g., .5ms-20ms). Then $electionTimeout$, the randomly assigned timeout times for each server (e.g, 10ms-500ms), and $MTBF$, the average time between failures of a server (e.g., several months or more).

STATE	
Persistent state on all servers (stored on stable storage before responding to RPCs)	
currentTerm	Latest term server has seen (initialized to 0 on first boot, increases monotonically).
votedFor	Candidate ID that received vote in current term (or null if none).
log[]	Log entries; each entry contains a command for the state machine and the term when it was received by the leader (first index is 1).
Volatile state on all servers	
commitIndex	Index of the highest log entry known to be committed (initialized to 0, increases monotonically).
lastApplied	Index of highest log entry applied to the 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).
RequestVote RPC	
Invoked by candidates to gather votes.	
term	Candidate's term.
candidateId	Candidate requesting vote.
lastLogIndex	Index of candidate's last log entry.
lastLogTerm	Term of candidate's last log entry.
Results	
term	Current term, for candidate to update itself.
voteGranted	**true** means candidate received vote.
Receiver Implementation	
1. Reply **false** if **term** < **currentTerm**. 2. If **votedFor** is **null** or **candidateId**, and candidate's log is at least as up-to-date as receiver's log, grant vote.	

AppendEntries RPC	
Invoked by leader to replicate log entries; also used as heartbeat.	
Arguments	**Description**
term	Leader's term.
leaderId	Allows follower to 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	Current term, for leader to update itself.
success	**true** if follower contained entry matching **prevLogIndex** and **prevLogTerm**.
Receiver Implementation	

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

0.1.3 Cluster Reconfiguration (Adding, Removing, and Replacing Servers)

Cluster reconfiguration involves adding, removing, replacing servers, or adjusting routines:

Definition 1.11: Stop and Restart & Pause and Resume

- **Stop and Restart:** Immediately halt operations, save state, reconfigure, and restart.
- **Pause and Resume:** Pause executing incoming requests and instead save them in a buffer, finish current processes, reconfigure, restart and resume.

Raft takes a two phase approach to reconfigure the cluster:

Definition 1.12: Raft Joint Consensus Reconfiguration (Part 1)

Joint consensus allows for a cluster to service requests while reconfiguring. It does this via a transition state, which is a combination of the old and new configurations (Joint consensus).

Reconfiguration: Once a leader receives a configurations request to swap from C_{old} to C_{new}, it appends $C_{old,new}$ to its logs, proceeding as follows:

- **Phase 1:**
  1. Propagate $C_{old,new}$ to all servers.
  2. Servers receiving new configurations (e.g., $C_{old,new}$), acknowledge it (**regardless** if it's committed or not).
  3. A leader crashing leads to reelections under C_{old} or $C_{old,new}$ (committed or not).
  4. Once $C_{old,new}$ is committed, the leader appends C_{new} to its log and replicates it.
- **Phase 2:**
  1. C_{new} is committed and live, any server not under C_{new} can be shut down.
  2. The leader who committed C_{new} becomes a follower to account for **Deletion**.

Addition: New Servers enter with **empty logs** and added in the following manner:

1. The leader upon receiving C_{new} is notified of new server existence.
2. New servers **cannot vote**, but only receive logs (as to avoid splitting the cluster).
3. Once new servers are synchronized in logs, $C_{old,new}$ may be committed.
4. $C_{old,new}$ clusters require the majority vote from **both old and new servers**.

Deletion: To remove servers, they are excluded in the C_{new} configuration; **With the only exception** that leaders who do not exist in C_{new} may continue to manage the cluster until C_{new} is committed.

Though we must address the situation where servers outside of C_{new} try to cast votes:

Definition 1.13: Raft Joint Consensus Reconfiguration (Part 2)

Since Raft processes requests asynchronously, during the transition to the new configuration from $C_{old,new}$ to C_{new}, the following could happen:

- A server not yet notified of the new configuration may not know it has been removed from it. Hence, it may not receive heartbeats and attempt to start a new election.
- A server with some other configuration comes online and begins to send out RPCs to the cluster.

In any such case of unwanted votes, Raft enacts the following protocol to ensure safety:

- **Leader Liveliness:** If servers within C_{new} believe they have a leader, they ignore votes (regardless of higher order terms).
- **Majority Rule:** Even if an election occurs, C_{new} members will vote amongst themselves, for which servers outside of C_{new} will never win.

Ideally, we would like to immediately terminate servers that are removed, but this may not always be possible or practical.

0.1.4 Log Compaction & Snapshotting

In finite systems, logs cannot expand without bound. So as to clean log entries without losing state consensus, snapshotting is employed.

Definition 1.14: Raft - Log Compaction & Snapshots

Raft compacts logs via state snapshots (**??**). Servers independently take snapshots, and **are not** initiated by the leader. Snapshots employ the following protocol:

1. Trigger snapshot when logs reach a fixed size in **bytes** or time (e.g., see below tip).
2. The snapshot replaces **committed** log entries and retains entries yet to be committed.
3. The snapshot includes: last committed log index and term, and machine state (e.g., key-value pairs).

Tip: Simple Snapshot Initiate Protocol - One possible solution suggested by the Raft paper is to initiate the snapshot when the log reaches a certain size in bytes. Ideally a size significantly larger than the size of the expected snapshot. Since snapshots have a **fixed cost** in setup, it reduces the overall cost overtime if snapshots occur at fewer intervals.

This works for the following reason:

Proof 1.1: Raft - Log Compaction & Snapshots

Having the leader initiate snapshots may overcomplicated its role, RPCs, and or interfere with bandwidth by sending state over the network. Since the leader has already vetted committed log entries, it is safe to assume that resulting independent snapshot consisting of such entries are also safe. ■

Though it's not ideal to send an snapshots over the network, in some cases it might have to.

Definition 1.15: Raft Resynchronization (InstallSnapshot RPC)

There are two cases where a snapshot may be sent:

1. **Missing Entry**: If the leader has already compacted a log entry it needs to send to a server, it sends the snapshot instead.
2. **Server Recovery**: A server coming online may be severely behind, and thus, simply sending AppendEntries RPCs may take too long.

In such cases the leader will send a **InstallSnapshot RPC**. Such RPC includes the last log index. The server will discard all entries before the last included index, and retain the rest.

Copying the entire dataset is costly, as to avoid this, Raft recommends a lazy snapshotting approach:

Definition 1.16: Raft Snapshots - Copy-on-Write (CoW)

When snapshots occur, the server marks existing data as shared between the snapshot and the live state. Only when a write (modification) touches shared data, does the server copy to preserve the snapshot.

InstallSnapshot RPC	
Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.	
Arguments	**Description**
term	Leader's term.
leaderId	Allows follower to 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.
Response to Sender	
term	Current term, for leader to update itself.
Reciever Implementation	

1. Reply immediately if **term** < **currentTerm**.
2. Create new snapshot file if first chunk (**offset** = 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 the 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).

0.1.5 Raft Algorithm Paper

Here we have provided the general outline for the Raft Algorithm. For a more detailed explanation, please refer to the original paper, which includes proofs:
• https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf.
And the extended version here:
• https://raft.github.io/raft.pdf
And these two websites which offer interactive visualizations of the Raft Algorithm:

- http://thesecretlivesofdata.com/raft/
- https://raft.github.io/

The **next page** will include the schematics for the Raft Algorithm's **State**, **AppendEntries RPC**, and **RequestVote RPC** [2].

0.2 Failure Models

0.2.1 Defining Failures

This section discusses, what to do or how to classify when failures occur.

Definition 2.1: Failure Model

A **Failure Model** is a set of assumptions about the types of failures that can occur in a distributed system. Such failures may be **Correlated** or **Independent**. Processes that do not fail are considered **Correct**.

In particular, we are concerned with the following types of failures:

- Crash and or omission of responses, and how we might recover from them.
- Deviation protocol, arbitrarily or maliciously.

Definition 2.2: Crash-Stop Failure

A **Crash-Stop Failure** occurs when a process halts and does not resume (e.g., power failure, software crash). The failure is not explicitly detectable by other processes. **Permanent hardware failures** typically fall under category of crash-stop failures.

Definition 2.3: Fail-Stop Failure

A **Fail-Stop Failure** is a detectable crash-stop failure (e.g., timeouts, heartbeats).

Definition 2.4: Omission Failure

An **Omission Failure** can be categorized into two types:

- **Send Omission**: The process fails to send messages according to the protocol.
- **Receive Omission**: The process fails to receive messages that were sent by other processes.

In particular, The process itself may still be operational but unable to correctly communicate (e.g., network disruptions, software errors, or buffer overflows).

Definition 2.5: Crash-Recovery Failure

A **Crash-Recovery Failure** occurs when a process halts due to a crash but retains the **ability to recover** and resume execution.

- **Crash Phase**: The process halts in some way (e.g., stops sending or receiving messages).
- **Recovery Phase**: The processes may recover to the last correct state via snapshot (**??**). Certain types of memory may persist through the crash:
  - **Volatile memory** is lost during a crash (e.g, mid-execution variables).
  - **Stable storage** is retained through a crash (e.g., disk storage, assuming no disk failure).

However, if the processes crashes indefinitely, it is considered a **Crash-Stop** failure (2.2).

Definition 2.6: Byzantine Failure

A **Byzantine Failure** occurs when a process exhibits arbitrary or malicious behavior, leading to unpredictable system behavior.

- **Arbitrary Behavior**: The process deviates from the expected protocol, such as:
  - Sending corrupted or inconsistent messages to different nodes.
  - Updating its state in an unintended or unpredictable manner.
- **Malicious Behavior**: The process actively attempts to disrupt the system, such as:
  - Exploiting protocol vulnerabilities to manipulate outcomes (e.g., double-spending in a blockchain).

In short, these failures occur due to **bugs** (unintentional) or **attacks** (intentional).

Tip: The term **Byzantine** in computer science comes from the *Byzantine Generals Problem*, introduced by Leslie Lamport in 1982. It describes a scenario where generals must coordinate an attack but cannot trust all messengers—some may be traitors sending conflicting information.

The name *Byzantine* is inspired by the Byzantine Empire, which was historically known for its complex and often deceptive political intrigues. While the term is widely used in distributed systems, some argue it unfairly portrays Byzantine history.

In computing, a **Byzantine failure** refers to a system component acting unpredictably, whether due to bugs, faults, or malicious intent, making consensus difficult.

0.2.2 Failures Model Hierarchy

Here we compare failure models as extensions of each other, though we will omit **fail-stop** as it is more of a detection mechanism. To quickly recap:

- **Crash-Stop**: Process halts and cannot resume (undetectable).
- **Omission**: Process fails to properly communicate.
- **Crash-Recovery**: Process halts but can recover and resume.
- **Byzantine**: Process exhibits arbitrary or malicious behavior.

Theorem 2.1: Failure Model Hierarchy

The failure models can be arranged in a hierarchy, where each model is an extension of the previous one. The hierarchy is as follows:

$$\textbf{Crash-Stop} \subset \textbf{Omission} \subset \textbf{Crash-Recovery} \subset \textbf{Byzantine}$$

In particular,

- **Crash-Stop** $\subset$ **Omission**: A crash-stop is a full crash rather than a partial communication failure.
- **Omission** $\subset$ **Crash-Recovery**: During recovering the last correct state, volatile memory lost may exhibit omission-like behavior. Meaning some messages are lost due to "**amnesia**."
- **Crash-Recovery** $\subset$ **Byzantine**: as a process may recover and exhibit arbitrary or malicious behavior. Moreover, a Byzantine failure may be **any type of failure**.

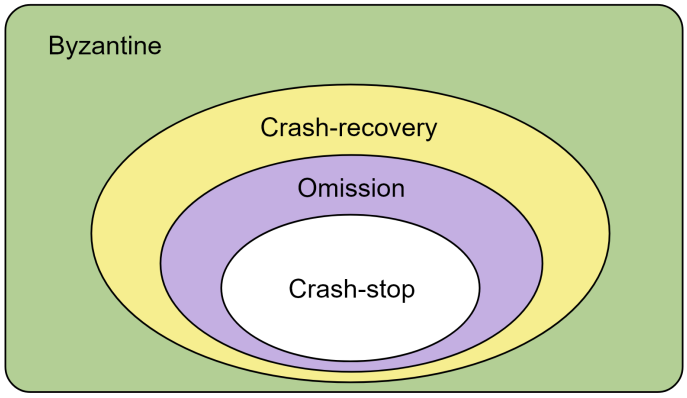

Figure 0.5: Failure Hierarchy depicting: Crash-stop $\subset$ Omission $\subset$ Crash-Recovery $\subset$ Byzantine

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.

[2] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.