**Understanding Raft Algorithm: Consensus and Leader Election Explained**

[[](https://medium.com/@jitenderkmr?source=post_page-----faadf28fd047--------------------------------)](https://medium.com/@jitenderkmr?source=post_page-----faadf28fd047--------------------------------)

[Jitender Kumar](https://medium.com/@jitenderkmr?source=post_page-----faadf28fd047--------------------------------)

·

Follow

7 min read

·

May 15, 2024

37

A cartoon of a pink animal on a boat

Description automatically generated

**Introduction**

In the world of distributed systems, ensuring that nodes can agree on a single, consistent state is crucial. Raft is a consensus algorithm designed to achieve this by providing fault tolerance and ensuring that a single leader is elected to coordinate operations. In this article, we’ll delve into Raft’s core concepts, focusing on consensus and leader election. We’ll also provide a practical example in Go to illustrate how Raft works in action.

**What is Raft Algorithm?**

Raft is a consensus algorithm developed by Diego Ongaro and John Ousterhout at Stanford University. It’s designed to be easier to understand than previous algorithms like Paxos while providing strong fault tolerance and leader election capabilities.

**Core Concepts:**

1. **Nodes (Servers):**In Raft, the network is composed of several nodes or servers.
2. **Leader:** One of the nodes in the Raft cluster is elected as the leader. The leader is responsible for managing the replication of logs across the cluster.
3. **Follower:** All other nodes in the cluster are followers. They respond to requests from the leader and forward client requests to the leader.
4. **Candidate:** When a leader fails, a new leader needs to be elected. Nodes transition to the candidate state and initiate an election.
5. **Term:** Raft operates in terms, where each term begins with an election and ends with a new leader being elected or re-elected.
6. **Log Replication:** Raft ensures that all logs across the cluster are replicated and maintained in the same order.

**Consensus and Leader Election:**

The primary goal of Raft is to achieve consensus among nodes in the cluster regarding the state of the system. Here’s how it works:

1. **Leader Election:**

* At the beginning of each term, nodes start as followers.
* If a follower doesn’t hear from the leader for a certain period (election timeout), it transitions to the candidate state.
* The candidate requests votes from other nodes. If it receives votes from the majority, it becomes the leader.
* If no node receives a majority, a new election is started in the next term.

2. **Log Replication and Consistency:**

* The leader accepts client requests and appends them to its log.
* It then sends the log entry to followers, which replicate the log entry.
* Once a majority of followers acknowledge the entry, it’s committed to the log and applied to the state machine.

We can definitely elaborate more on the Raft consensus algorithm and its components for a deeper understanding. Here’s a more detailed explanation of the components and syncing process:

**Consensus in Raft**

Raft’s approach to consensus is crucial for ensuring the integrity and consistency of the distributed system. Consensus is achieved through a series of steps:

1. **Log Replication:**

* When a client initiates an operation, such as setting a key-value pair, the leader node receives the request.
* The leader appends the operation to its log and broadcasts this log entry to all other nodes in the cluster, including peer nodes.
* Each node in the cluster appends the log entry to its log.

2. **Majority Agreement:**

* Raft operates on the principle of majority agreement. Before committing an operation to its state machine, the leader node waits for acknowledgments from most nodes.
* If most of the nodes(Say N/2 + 1) acknowledge the operation by replicating it in their logs, the leader commits the operation to its state machine.
* This ensures that the operation is officially part of the system’s state and will be applied consistently across all nodes.

Additionally, the leader regularly sends updates to the other servers to keep them in sync. This ensures that even if a server falls behind or crashes, it can quickly catch up with the latest state of the key-value store.

Now, for a **demo**:  
Let’s say we have 3 servers, **server1, server2 and server3**. Server1 becomes the leader through a process called leader election (we’ll talk more about that later). When the client sends requests to set keys (like A, B) to certain values, server1 waits for a majority of the servers to agree before confirming to the clien

t.  
Meanwhile, it sends the request to server2 and server3 for backup. Once it gets confirmation from the majority, it tells the client it’s done and keeps the other servers updated. During this process, you’ll see messages indicating that the leader is waiting for agreement while the client waits for a response.

A screenshot of a computer

Description automatically generated

Consensus

A screenshot of a computer

Description automatically generated

**Syncing with Peer Nodes**

Syncing with peer nodes is a crucial aspect of Raft’s design to maintain consistency and bring new nodes up to speed. Here’s how it works:

1. **Syncing New Servers:**

* When a new server (e.g., ServerThree) joins the Raft cluster, it needs to catch up with the current state of the key-value store.
* To achieve this, the leader node sends a series of log entries to the new server, ensuring that it has a complete copy of the log.
* These log entries include all previous operations, such as SET A 1, SET B 2.

2. **Determining Logs for Sync-Up:**

* The leader server determines which logs to send for the sync-up process based on the last acknowledged log index for each peer node.
* If a peer node has acknowledged logs up to index k, the leader sends logs from index k+1 to the last log (n) for syncing.
* This process ensures that the new server receives all operations it missed and brings it up to date with the latest state of the key-value store.

3. **Handling Sync-Up Errors:**

* If the leader node is not aware of the last acknowledged log index of a peer node, it uses a hit-and-trial approach.
* The leader starts from the end of its log index and decrements the index based on responses from the peer node.
* If the peer node does not have the logs before the index sent as part of the sync-up request, it returns an error.
* This iterative process ensures that the new server receives the correct logs for syncing.

The sync-up process should be triggered at a pre-defined interval, taking into account factors such as the number of nodes in the system and the frequency of node additions or removals. In the above use case, a sync-up interval of 3 seconds is used, meaning the leader server sends a sync-up message every 3 seconds to ensure all peer nodes are updated.

**Leader Election**

In case the leader node fails, peer nodes monitor for a sync-up request from the leader within a set threshold. If this threshold is exceeded, a peer node assumes leadership is unavailable and triggers a leader election. It requests votes from other nodes, and upon receiving a majority, announces itself as the new leader. Each node randomizes its threshold to prevent simultaneous elections. This threshold must exceed the sync-up interval to prevent premature leader declarations.

In a demonstration, if we have three servers and server1 becomes leader, and then server1 fails, server3 detects the failure and initiates an election. After obtaining a majority vote, it becomes the new leader. The client seamlessly resumes operations with the new leader, ensuring continuity in the key-value store.

A screenshot of a computer

Description automatically generated

server1 is leader

A screenshot of a computer

Description automatically generated

server3 is leader

**Used in Production**

The Raft algorithm has been adopted by several popular distributed systems for ensuring reliability and consistency. Some notable systems that use Raft include:

* **Apache Kafka:** Kafka, a distributed streaming platform, employs Raft for managing metadata and coordinating brokers in the cluster.
* **ScyllaDB:** ScyllaDB, a high-performance NoSQL database, utilizes Raft for leader election and coordination among nodes for consistency and fault tolerance.
* **etcd:** etcd is a distributed key-value store used for configuration management, service discovery, and coordination. It relies on Raft for distributed consensus to ensure consistency and fault tolerance.
* **TiKV:** TiKV is a distributed transactional key-value database built to support the TiDB distributed SQL database. It uses Raft for distributed coordination and replication of data across nodes.
* **Consul:** Consul is a service mesh and service discovery tool that provides key-value storage for configuration, health checking, and distributed coordination. It employs Raft for consensus among nodes in a Consul cluster.

There are lot of improvements that can be done, some which are:

1. **Binary Search for Log Index:** Instead of decrementing the log index by one during sync-up, a binary search can optimize the process, particularly with large log sizes, reducing back-and-forth iterations.
2. **Handling Split Brain Problem:** Currently, the implementation doesn’t address the split brain problem, where two nodes may both assume leadership. This can lead to inconsistencies in the key-value store. Strategies need to be implemented to prevent such scenarios and ensure consistency.
3. Reference to **Martin Kleppmann’s Explanation**: Martin Kleppmann’s detailed explanation of the Raft algorithm provides valuable insights, aiding in debugging and understanding the consensus and leader election processes.

The implementation code is available on [**GitHub**](https://github.com/Jitender271/go-raft), offering a hands-on understanding of the Raft algorithm. Exploring more technical papers in this manner could be beneficial for deeper insights into distributed systems.