RAFT Consensus in OCaml

Whit Jackson

1 Introduction

In this paper, I describe the implementation of the Raft distributed consensus protocol in the OCaml programming language. While some features are yet to be implemented, this implementation covers a substantial subset of the Raft protocol specification.

2 Background

Raft was designed in the early 2010s by Diego Ongaro and John Ousterhout.[1] While Lamport's Paxos algorithm has been frequently used to achieve distributed consensus, it is notoriously difficult to understand and implement. While Raft "produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos," the protocol was designed with *understandability* in mind. The protocol separates consensus into distinct parts, each unit of which is relatively simple to implement on its own.

OCaml is a functional programming language with strong static typing and Hindley-Milner type inference. I think the combination of strong-typing, memory safety, and high-level abstractions make it a good choice for distributed systems. In particular, OCaml makes mutation of values explicit and emphasizes the use of pure functions. Given the importance of homogenous/deterministic behavior in distributed consensus, minimizing side effects makes it easier to reason and determine the state of the system at any given time.

3 Motivation

Most applications of sufficient size use a distributed architecture to manage server load and increase resiliency. However, there are inherent trade-offs between scalability, consistency, and availability of data in a distributed network. Maintaining a consistent state between different nodes in a network is difficult. One approach has a leader node maintain an authoritative global state and replicate/provide that state to followers. However, if the leader node becomes unavailable, clients will not be able to modify or read the latest global state.

Alternatively, nodes can replicate state changes to all other nodes in the network. This approach allows the network to continue operating if a node fails. Unfortunately, this approach is prone to race conditions when done naively because nodes do not have a mechanism to deterministically order transactions in the presence of network latency and transaction conflicts.

Raft is a middle-ground between these approaches. The protocol aims to maintain a consistent global state across a set of trusted servers, even if individual nodes exhibit instability or network errors. When correctly implemented, a network implementing Raft can continue operating and updating the state as long as a majority of nodes are online. If less than 50% of nodes are online, no new state changes can be committed, but the state of each node will remain consistent. Unlike the former approach, a fault in a single node is not capable of stopping the entire network. Unlike the second approach, Raft has strong consistency under non-byzantine conditions.

4 Protocol

The problem of distributed consensus is, in effect, equivalent to building a replicated state machine. Suppose we have a global state and a set of actions that modify that global state. If clients submit a set of actions $a_1, a_2, ..., a_n$ to the network, all nodes should reach the same global state g' where g' is equal to the successive application of these actions, in some consistent order, to g. Importantly, there are not guarantees about the order these actions are applied to the global state in most cases, but we do guarantee that this ordering will be consistent across all nodes. One approach to achieve this is to have all nodes store a consistent log of actions applied to the global state. To derive the current state, we can successively apply actions in the log to the initial global state.

Raft breaks down this process into two primary components: log replication and leader election. Nodes are functionally homogeneous and can be in three states: leader, follower, and candidate. The protocol first elects

[1] Ongaro and Ousterhout, "In Search of an Understandable Consensus Algorithm", 2014

a leader node, which will be responsible for managing the global state and coordinating the replication of logs across follower nodes. The leader election process ensures that only a single leader is chosen at any given time, maintaining consistency. The log replication process helps distribute the logs across all nodes in the network, ensuring each node maintains an up-to-date and consistent view of the global state.

4.1 Leader Election

During leader election, nodes in a Raft cluster select a node to act as the leader and follower nodes select a random value within a given range as an election timeout. The leader sends periodic heartbeats to clients, and an election process is initiated whenever a follower node times out while waiting for a heartbeat message from the current leader. When this occurs, the follower node transitions to a candidate and begins an election.

During an election, the candidate node increments its term number, votes for itself, and sends a RequestVote message to all other nodes in the cluster. Each node receiving the RequestVote message checks whether it hasn't already voted in the current term and if the candidate's log is at least as up-to-date as its own log. If these conditions hold, follower nodes vote for that candidate.

A candidate node becomes the leader when it receives votes from a majority of nodes in the cluster. Once elected, the new leader sends heartbeat messages (empty AppendEntries messages) to all follower nodes to confirm its leadership status and prevent new elections. If no leader is elected (e.g., due to a split vote), nodes will wait for another timeout and start a new election process. After a leader is elected, any other candidates will revert to a follower state.

4.2 Log Replication

Once a leader has been elected, it manages process of replicating the log to ensure that nodes maintain a consistent view of the global state. A client will submit actions to the leader, who will append these actions as entries in its log. The leader then sends an AppendEntries messages to each follower node, containing one or more log entries.

When a follower node recieves an AppendEntries message, that node wiill check if the leader's term is up-to-date and if the follower log contains an entry with a matching index and term number as specified in the leader's message. If both conditions are met, the follower node appends the new log entries to its log and responds with a success message. If either condition fails, the follower node responds with an error message, and the leader retries the AppendEntries message with a lower index.

4.3 Finality

To ensure consistency, the protocol uses a commit index, which refers to the highest log entry index known to be committed by a majority of nodes. The leader maintains a separate commit index for each follower and updates these indices to reflect each node's highest log index as it receives success messages from followers. When the leader determines that a log entry has been replicated on a majority of nodes, it updates its commit index and informs the followers of the new commit index. When an entry has been replicated to a majority of nodes, it can be assumed to be *final*. In other words, it is safe to apply that entry to the state machine, as it is will not be overwritten or discarded in the future.

Follower nodes, upon receiving the new commit index from the leader, update their commit index and apply the corresponding log entries to their local state machines. This process ensures that all nodes eventually apply the same set of log entries in the same order and arrive at a consistent global state.

4.4 Membership Changes

In a distributed system, it is important that nodes are capable of joining and exiting the cluster. While the Raft paper describes a two-stage approach to cluster membership changes, I use a simplified version of the membership protocol that uses a one-node-at-a-time approach to maintain consistency.[2]

In this protocol, AddServer and RemoveServer commands are added directly to the log. Nodes update their knowledge of cluster membership as soon as they append AddServer or RemoveServer commands to their logs, rather than waiting for these commands to be committed. The first node to join the network becomes a peerless leader. When a node starts up, it specifies a bootstrap server and sends an AddServer command for itself to that server. The leader replicates RemoveServer commands to the removed server. If a node receives a RemoveServer command for itself, it exits the cluster.

The one-node-at-a-time restriction prevents splits by ensuring that any two or more split groups share nodes.These shared nodes act as arbitrators to prevent multiple leaders from being elected. For example, consider a cluster of four nodes with an additional node being added. Nodes that still think there are only four nodes in the cluster will need support from three nodes to form

[2]This approach is designed by Eileen Pangu: https://eileen-code4fun.medium.com/raft-cluster-membership-change-protocol-f57cc17d1c03

a majority. Nodes that recognize there are now five nodes in the cluster will also need support from three nodes. Since there are only five nodes in total, the two groups of three nodes are guaranteed to have overlaps. This overlap prevents multiple leaders, as a node will only vote for one candidate in any given term (as described in Pangu 2020).

5 Implementation

The implementation of the Raft consensus protocol in OCaml leverages the limited concurrency provided by the Async library. There is no global mutable state. Most functions take in a $State.t$ type and produce a new $State.t$ type. This approach allows for more straightforward reasoning about state changes and greatly simplifies testing.

5.1 RPC Servers

In this implementation, nodes run two separate RPC servers: one for the protocol itself, and another for admin actions and client interactions with the network. Both RPC servers utilize binary RPCs, which are efficient and type-safe. I use the bin_io library, which is designed for efficient serialization and deserialization of OCaml data structures.

Using the async-rpc library, a persistent connection to each peer is established and maintained. These connections are represented as file descriptors and are stored in the $State.t$ type.

5.1.1 Protocol RPC

The protocol RPC is implemented using Async's one-way RPCs, which ensures that the event loop does not hang while waiting for a response. This choice is suitable for Raft because the protocol is resilient even if some messages are not communicated. Responses are sent as independent messages.

When the protocol RPC receives events, it places them in an event queue implemented with Async's Pipe.t datatype. Pipe.t is a useful tool for managing event queues and it uses file descriptors to create an ordered stream of values between a published and a consumer. This design allows for ordered handling of incoming events.

When processing the event queue, the system first checks for any unprocessed messages. If there are pending messages, they are processed accordingly. If there are no incoming messages, the server checks whether the heartbeat or election timeouts have passed and handles those events as needed.

One of the strengths of using the Async library is that it does not rely on preemptive threading. This design choice ensures that it is impossible to process two events simultaneously, which could lead to inconsistencies. However, the lack of preemptive threading also means that the implementation cannot use a setTimeout function to handle timeouts like Petros does. An event queue helps manage the ordering of events, as well as buffer responses during bursty periods.

5.1.2 Client RPC

The client RPC server is responsible for sending commands to the network. In this implementation, the state machine is represented as an integer, with increment and decrement commands available for manipulation. The RemoveServer command can also be called through the client RPC.

When a node's client RPC receives a message, it forwards the message to the leader. Once the leader receives the message, it appends it to the log and initiates the AppendEntries process.

5.2 Membership Changes

The implementation supports dynamic membership sizing as described above. When a node starts up, it sends an AddServer message to a provided bootstrap node or starts a new network if no bootstrap node is specified. The list of peers is generated from the log, ensuring that all up-to-date nodes maintain a consistent set of peers.

5.3 Future Improvements

There are a few areas where the implementation can be further improved. Data persistence as described in the Raft paper has not yet been implemented. Log compaction has not yet been implemented. Despite this, nodes can still catch up quickly using AppendEntries. Optimizing the log representation could also be beneficial, as the current linked list implementation isn't well-suited for large logs due to the time complexity of lookups.

6 Results

The evaluation of this Raft protocol implementation in OCaml has been primarily focused on the correctness of the system rather than its performance. Nevertheless, the performance observed during local testing was promising.

Unfortunately, due to cross-compilation issues, the code was not run on the unix lab machines. Instead, a local testing environment was utilized. This testing

was focused on scaling and fault tolerance. The cluster scaled up seamlessly from 1 to 30 nodes (using one-at-a-time-semantics) with a 100 millisecond heartbeat timeout and a 300-600ms randomized election timeout, without changing terms.

6.1 Fault Tolerance and Leader Election

The robustness of the leader election process was evaluated by abruptly terminating the leader node. The system proved its resilience in such scenarios, consistently electing a new leader within two terms. This quick recovery demonstrates the protocol's ability to handle sudden faults effectively. Nodes that were killed could successfully rejoin the network, further attesting to the system's resilience.

6.2 Performance

Even though the primary focus was not on performance, the results are encouraging. The parameters used for heartbeat and election timeouts suggest that in normal operating conditions, state updates could reach finality in about 100 milliseconds. In the worst observed cases, this could increase to less than 1.2 seconds.

These estimates do not include network latency, which would be expected to increase the probability of election failure and latency. In higher latency environments, it would be necessary to increase timeout durations. However, when these parameters are well-tuned, excessive leader reelections should be avoided.

A resilience test was conducted simulating a faulty network by setting nodes to ignore one third of the messages they received. In this situation, the election timeout was doubled. The scaling properties remained consistent, although in some instances, the cluster took three terms to elect a new leader. This is understandable because entries/heartbeats are resent when a success message is not received. However, RequestVote RPCs are sent only once, and if they are dropped, the chances of a failed term increase.

6.3 Analysis of Results

The findings from this study offer insight into the potential of implementing Raft in OCaml, specifically regarding its scalability, resilience, and performance. Despite the limitations imposed by the testing environment, the results provide a reliable indication of the protocol's capabilities.

The ability of the system to scale up from 0 to 30 nodes without changing terms is a clear testament to the **scalability** of this implementation. In practical scenarios, this characteristic is particularly beneficial for systems that require dynamic scaling in response to varying workloads. This adaptability can ensure efficient resource allocation and utilization while maintaining system performance. While the dynamic scaling protocol is a simplification compared to the original paper, it provides a proven and scalable method for changing group membership.

The system's **fault tolerance** capability, as evidenced by its leader election process, is particularly impressive. Upon abrupt termination of the leader node, the system was quick to elect a new leader. This prompt recovery showcases Raft's resilience, as well as its ability to minimize downtime, which ensures the continuous availability and reliability of the system. Moreover, the ability of terminated nodes to successfully rejoin the network is indicative of the system's recovery capabilities, further reinforcing the **robustness** of the protocol/implementation.

While the testing was not primarily focused on performance, the observed performance metrics were encouraging. The observed latencies are acceptable in many real-world applications, indicating the potential usefulness of this implementation. Timeouts can likely be further reduced from the test parameters on local networks without significantly increasing the fault rate.

However, introduction of network latency and message loss in the simulated faulty network test revealed important areas for further optimization. Although the system proved to be resilient, the election timeout had to be doubled, and in some cases, the cluster took three terms to elect a new leader. This suggests that while the system handles faults efficiently, its performance can possibly be impacted in scenarios with high network latency or packet loss through retries. However, these changes would need to be thoroughly analyzed to ensure they don't compromise any protocol guarantees.

7 Conclusion

The implementation of the Raft consensus protocol in OCaml demonstrates its effectiveness and potential in building distributed systems with scalability, fault tolerance, and performance in mind. The design choices, such as strong typing and explicit state changes, contribute to the understandability and correctness of the implementation. The use of the Async library provides limited concurrency and efficient event handling, ensuring the protocol's resilience and responsiveness.

The evaluation of the implementation indicates that the Raft protocol can handle dynamic membership changes, scale well with a growing number of nodes, recover quickly from faults, and achieve acceptable performance in various scenarios. While there are areas for further optimization, such as log persistence, log com-

paction, and log representation, the implementation already showcases promising results.

Overall, the Raft protocol implemented in OCaml provides a solid foundation for building distributed systems that require consensus, replication, and fault tolerance. Its understandability, resilience, and scalability make it an attractive choice for applications that prioritize consistency, availability, and partition tolerance. With further improvements and optimizations, this implementation can be extended and utilized in real-world distributed systems with confidence.