Building a Fault-Tolerant Online Store Using Raft Consensus Algorithm

By Paul Cozzi, Peter Paiste, Eyenunwana Nwoko, Atakan Guney

GitHub With Source Code Available at: https://github.com/petethemeat/raftSubmission

Abstract

For students studying distributed consensus, the motivation behind consensus algorithms as well as certain algorithmic implementations of consensus can be difficult to grasp. In this paper, we explore a consensus algorithm that is designed to be easy to understand, Raft. Using Raft, we successfully implemented an asynchronous, fault-tolerant online store in order to give a concrete example of how consensus algorithms and replicated state machines can be used in a real-world setting.

2. Introduction

Our design goal was to implement a replicated state machine using Raft consensus algorithm. Consensus algorithms seek to provide three things: agreement (data matches on all servers), nontriviality (the servers requests are taken into account when deciding which data will be consistently present) and termination (we try to prevent the algorithm from going on forever, but this is impossible in asynchronous systems, so we provide an element of randomness to minimize this possibility). We chose Raft because of its simplicity compared to other consensus algorithms. The system (from Assignment 4) involves a cluster of servers which all store the inventory for an online store. The servers must have the ability to receive commands from clients and update the inventory accordingly. Additionally, each server must agree on the inventory updates (There should be no inventory conflicts between servers).

Our original solution to this problem involved using Lamport's Mutex algorithm to prevent conflict by allowing only one server to modify the inventory at a time. This system was fault tolerant down to a single server, but it required the assumption the system was synchronous. The Raft algorithm described in [1] is designed for asynchronous consensus, so it provides an excellent opportunity to

improve our solution to Assignment 4 by keeping a fault-tolerance of n/2 - 1 (where there are n servers) and allowing servers to shut down and come back up or have undefined message delay.

Additionally, [1] references several implementations of Raft, but the purpose of a replicated state machine can be unclear to someone new to distributed systems. Therefore, our project aims to provide a concrete example of the utility of a distributed state machine using Raft, rather than just an implementation of Raft itself.

3. Description of Project

Our project was to create a fault-tolerant method of responding to clients' requests to purchase and view items by updating a store's inventory. The fault-tolerance was obtained through duplicating the inventory on multiple servers, and the agreement of the duplication was obtained by utilizing the Raft consensus algorithm. Because we were provided with a paper on Raft to start with, our research on the proper algorithm to use did not need to go further than the paper cited, [1]. Raft is generally a good choice for consensus because it has comparable performance to, and greater simplicity than, other consensus algorithms (such as Paxos). Raft centers itself around a single elected leader, with the goal of replicating request logs that are received from clients. Therefore, our design is broken into three parts: leader election, log replication, and client interaction. This section will give a brief description of each of those parts.

Raft has a few components which must be introduced before understanding our design. The first is a term, which is Raft's form of a clock. The term starts at 0, and increments for each election cycle. There is also a log, which is the list of requests which may only be committed to a server's inventory when they are duplicated on the logs of a majority of servers. Finally, there are two kinds of message: requestVote and Append. These will be explained later.

Raft has a single elected leader server, whose purpose is to handle all client requests and serve as a ground truth for all logs. When our system starts up, each server is a follower, and each follower always

has a countdown timer of random length within a specified range (for example, server might time out in 300 to 500 ms). When a server's timer times out, it increments its term, becomes a candidate, and starts an election by sending a requestVote message (which includes the term) to all other servers. Other servers only vote once per term, and the majority vote wins. This ensures that there is only one leader per term. Additionally, servers only vote for other servers whose logs are up-to-date with their own. This ensures that the most recent entry that was applied to the inventory persists through all future elections. Leaders periodically send empty Append messages with their term to all other servers to ensure that the other servers do not time out.

Logs are replicated on all servers through Append messages. Whenever a request is received, it is added to the leader's log. The leader then sends this request to all other servers so they can update their logs. They send responses when their logs are updated, and when the leader receives a response from the majority of servers, it updates its inventory, sends a response to the client, and notifies the other servers to update their inventories with that request. The leader requires the majority of responses to ensure that if it crashes, at least one other server will know about the latest log entry, and the information can persist.

We ensure that only the leader can receive requests from clients by having each client first contact a random server. This server, if not the leader, sends the ID of the last leader it knew about to the client, so the client can try again.

There are many holes in the description here, but they were all covered in our implementation, and [1] describes the Raft algorithm more completely.

4. Design Alternatives

The consensus algorithm we used for our system was Raft because our problem statement specified that Raft must be used. However, we had to make many key decisions for our particular implementation of the algorithm. We decided to implement Raft from scratch rather than use one of several pre-built libraries in the spirit of education. We coded in Java and used TCP for message passing

because these things were more familiar to us than C++ or RMI. Finally, we omit certain Raft concepts to simplify the problem and more coherently test and explain our solution. These concepts include log compaction, which is a memory-saving mechanism that is not useful on the small scale we tested, and server cluster changes, which are meant more for system maintenance in the real world.

5. Evaluation and Results

Distributed systems are notoriously difficult to test, so our methodology for evaluation was to validate that our project worked in stages through user testing. In the base case for each test, we used 3 servers, but we tested our system with up to 5. The basis for raft is leader election, so we tested this first. We made sure a leader could be elected, then we killed the leader to make sure a secondary leader was elected. The first and second leader elections succeeded, and we then moved on to the second test phase: client interaction. To test that clients could communicate with the server cluster and that every request was eventually being redirected to the leader, we sent a dummy message and had the servers echo the request to the console, and we had the leader echo the response back to the client, along with its ID, to be printed. The client received a response only from the leader. This validated that the leader was serving requests. The last set of tests involved log replication. In Raft, the leader only applies a log entry to its state machine and responds to the client after the log entry is duplicated on a majority of servers. To validate this, we had the leader print the response of each other server, then print before it sent the message to client. These validation tests worked (and thus, our project could be demonstrated to persist after server crashes). The final test was to check if servers that missed log entries could gain them back after restarting. To validate this, we started up the last server after servicing multiple client requests, and printed from that server the last committed message. All messages printed in order.

Overall, our system consists of a demonstrable Raft implementation that persists through crashes and has server agreement. To take this project further, we could add logic for server cluster changes (IE, servers being reset) and log compaction to allow for longer operation periods.

6. Conclusion

For this project, we implemented the keeping of a fault-tolerant multi-server online inventory that could be used to service client requests. We showed that with the Raft algorithm, this implementation was possible, thus giving an asynchronous approach to Assignment 4. We implemented our project in Java, and validated that leader election, client interaction, and log replication functioned properly with a fault tolerance of up to half of the servers. Further study and modification of our publication could include logic for cluster changes and log compaction.

REFERENCES

[1] D. Ongaro and J. Ousterhout, *In Search of an Understandable Consensus Algorithm* [Online]. Available: *https://raft.github.io/raft.pdf*