Decentralized state machine using Nakamoto (probabilistic) consensus

Research Project 2022

Nicolas COURAUD (nicolas.couraud@etu.emse.fr)
École Nationale Supérieure des Mines de Saint-Étienne

October 31, 2022

Abstract

As defined by Schneider [Sch90], the state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas.

In this article, we look at state machines as a way to hold in multiple servers copies of the same record. That record, in my implementation, is a standard key-value data structure (a Go map).

I show how one can build a decentralized state machine using Nakamoto Consensus (in our implementation we used the Snowball consensus algorithm), and I study the advantages and drawbacks of such an approach, compared to classical consensus algorithms like Raft [OO13].

The implementation of the project is available at github.com/Nicolascrd/distributed-state-machine

Contents

1 Traditional state machine replication

1.1 Introduction

Decentralized state machine systems are currently built, for the most part, around classical consensus algorithms.

In classical consensus, all of the nodes have to know each other. It is a permissioned model, which means that nodes have to get an approval to get into the network. That is very normal for private infrastructure but might not work for public infrastructure. Because all nodes must communicate with each other, the communication cost in generally considered high.

Many classical consensus algorithms exists, the most famous one being Paxos [Lam98]. In my project, I used the Raft consensus algorithm [OO13] because it has the same performances as Paxos, and is simpler to understand and implement.

In Raft, each node can be either Leader, Follower or Candidate. In regular functioning, one node will be the leader and all the others will follow. The Leader node sends regular heartbeat requests to notify followers that it is still running. If they stop receiving the heartbeat, they switch to the follower status.

The client can ask any node to add a log to the record. The request includes the number of the log that the client wants to add and the log himself. If the client as a node which is not the leader, the request is transfered to the leader. If there is already a record at that number, the leader returns an error message to the client. Otherwise, the leader updates its record and asks all his followers to update themselves as well.

When requesting a log, the client can also query any node in the network. The node just responds with the log at that number in the local record that it is holding.

The only parameter than I included in my implementation is *updateSystem*, which is a boolean. If True, if the leader sends heartbeats and a node does not reply, it will be eliminated from the network in the knowledge of all the nodes. This make it possible to crash a majority of nodes and still have the state machine running and usable on all the surviving nodes.

1.2 Byzantine fault-tolerance (BFT) with classical consensus

Contrary to Crash Fault, Byzantine Fault [LSP82] is a type of failure where you consider that the node can fail in any possible way. In practice, it means that the node will stay up and can send malicious requests to interfere with the functioning of the network.

My implementation of the state machine does not tolerate Byzantine Faults at all. With only one byzantine node, it already endangers the whole system. Indeed, the malicious node can "elect himself" in Raft, and then send heartbeats to all the node in the network, including the legitimate leader to become the leader de facto. Then, it can ignore requests, reply whatever it wants and write any log it wants in any node it wants.

To tackle this issue, BFT algorithms were designed [CL99]. They include a layer of authentification, but are not necessarily much slower. However, there is a maximum number of malicious nodes that the algorithm can handle safely [DLS88]. For example, assuming partial synchrony of the nodes, consensus protocols can handle t crash failures with 2t+1 nodes but t byzantine failures with no less than 3t+1 nodes.

2 Nakamoto (probabilistic) state machine replication

2.1 The Nakamoto Consensus

The Naka consensus The Naka consensus The Naka consensus The Naka consensus The Naka consensus The Naka consensus The Naka consensus

2.2 My Nakamoto state machine implementation

Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine Nakamoto state machine

3 Performance Evaluation

3.1 Probability of not reaching consensus in Nakamoto Consensus

Markov Markov Markov Markov Markov Markov Markov Markov Markov Markov

3.2 Number of requests required in regular functioning

Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests Number of requests

3.3 In case of a failure

Failure ? Failure ? Failure ? Failure ? Failure ? Failure ? Failure ? Failure ? Failure ? Failure ?

4 Conclusion

Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion

References

[CL99] M Castro and B Liskov. Practical byzantine fault tolerance. 1999. `https://pmg.csail.mit.edu/papers/osdi99.pdf`.

[DLS88] C Dwork, N Lynch, and L Stockmeyer. Consensus in the presence of partial synchrony. 1988. `https://groups.csail.mit.edu/tds/papers/Lynch/jacm88.pdf`.

[Lam98] Leslie Lamport. The part-time parliament. 1998. `https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf`.

[LSP82] L Lamport, R Shostak, and M Pease. The byzantine generals problem. 1982. `https://lamport.azurewebsites.net/pubs/byz.pdf`.

[OO13] D Ongaro and J Ousterhout. In search of an understandable consensus algorithm. 2013. `https://raft.github.io/raft.pdf`.

[Sch90] F B. Schneider. Implementing fault-tolerant services using the state machine approach : a tutorial. 1990. `https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf`.