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

November 1, 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 (such as [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

Nakamoto consensus was introduced with the bitcoin blockchain [Nak08]. Nakamoto consensus is probabilistic, which means that there is a probability ε that consensus is not reached. Of course, ε can be so low that crucial systems can be built relying Nakamoto consensus. Nakamoto consensus is open : nodes don't have to register or get an authorisation to enter the network. Because nodes don't have to know every other node in the network and be able to communicate with everyone, Nakamoto is supposed to allow for greater scale in the network.

Nakamoto consensus definetly shaked the consensus space in computer science with cryptocurrencies. In a different context, can it be useful as well ?

2.2 My Nakamoto state machine implementation

To implement Nakamoto consensus, I chose the Snowball consensus algorithm [RYS$^+$19], which underlies the Avalanche blockchain.

The goal of snowball is to reach binary consensus starting from a network of nodes.

I adapted the protocol to fit the needs of a decentralized state machine, because snowball is described in the context of building a payment system (because it is the basis of Avalanche, which is that payment system). Snowball is built by iterations, but I will only focus on Snowball, the last iteration, because it is the most interesting and the one I tested the performances in the end. Slush and Snowflake are interesting to understand where we are coming from, but no more.

I adapted Snowball in the following ways : Each node, in addition to having all the snowball consensus related data (not much), has a version of the record. All nodes starts with a empty record. Because it is the requirement for a decentralized state machine, each node can be queried to add a log to the record. Each node can also be queried to get one log in the record.

The snowball protocol is triggered when the client asks to add a new log to an index that has not been touched yet. Indeed, the system works as if each index in the record corresponds to an independant snowball run.

In snowball, each node has a counter of "confidence" which can tip towards one way or another. In my implementation, there is no binary consensus because the only value the system has is the log request. When receiving an add-log request, the node follows this sequence :

from request (client or other node):
logToAdd (string)
logPosition (int)

from node memory :
record (map[int]string)
cnt (map[int]int)

if $record[logPosition]$ exists **then**

    reply $record[logPosition] == logToAdd$

else

    $record[logPosition] \leftarrow logToAdd$

    reply $True$

    initiateRequest($logToAdd$, $logPosition$)

end if

initiateRequest corresponds to the following sequence :

while $True$ **do**

    $success \leftarrow initiateQuery(req)$

    $initiateQuery$ picks a sample in the network and queries it

    if $success$ **then**

        $cnt[logPosition] + +$

    else

        if $cnt[logPosition] == 1$ **then**

            delete($record$, $logPosition$)

            $cnt[logPosition] \leftarrow 0$

        else

            $cnt[logPosition] - -$

        end if

    end if

    if $cnt[logPosition] \geqslant CounterThreshold$ **then**

        $record[logPosition] \leftarrow logToAdd$

        return

    end if

end while

3 parameters are used in my implementation :

- sampleSize (k) : the number of nodes that are queried each round

- majorityThreshold (α) : the number of nodes that have to reply successfully to consider that the query is successfull. (with

$$\alpha \geqslant \lfloor k/2 \rfloor$$

)

- counterThreshold (β) : the threshold of successfull requests above which the node stops initiating requests and only replies with its content. (for one position)

3 Performance Evaluation

3.1 Probability of not reaching consensus in Nakamoto Consensus

In order to compare the performances of classical consensus and Nakamoto consensus, we need to be able to know the probability of not reaching consensus ε corresponding to the set of values (n, k, β). (α is not included because we assume no byzantine nodes in this part).

The evolution of the system of nodes, for one position (one snowball run) is a Markov process. To compute to probability of not reaching consensus, we need to identify the possible states in the Markov process, the possible transitions and the transitions probabilities.

$\beta = 1$ With $\beta = 1$, the state is (c, reqs) with :

- c = Number of colored nodes ($c \leq n$)

- reqs = Number of groups of requests which are queried for next round. (They all come from neutral nodes)

If the state converges to $c = n$, consensus is reached. If the state converges to $reqs = 0$, with $c \neq 0$, consensus is not reached when the snowball run is done.

Then, the probability of reaching a certain state (1) is the sum of the probability of a different state (2) times the probability of going from state 2 to state 1. That is true if and only if states are transient, which is the case here because c can only grow and with c constant, reqs can only diminish.

Therefore, the probability of being at state (c, reqs) is (the term in the sum is considered 0 if it does not make sense, which is the case for some t) : The sum represents a step of the resolution of one group of requests (t represent the number of neutral nodes touched)

$$P(c, reqs) = \sum_{t=0}^{k+1} \binom{n - c + t}{t}\binom{c - t - 1}{k - t} P(c - t, reqs + 1 - t)$$

$\beta = 2$ With $\beta = 2$, the state is (c, s, rc, rs), with:

- c = Number of colored nodes (cnt = 2)

- s = Number of semi-colored nodes (cnt = 1)

- rc = Number of groups of requests from colored nodes

- rs = Number of groups of requests from semi-colored nodes

6

We must separate requests in those two groups because a request cannot be directed to the node initiating it. Therefore the probabilities of hitting a colored node is smaller if the request comes from a colored node than from a semi-colored node.

The term in the sums is considered 0 if it does not make sense. First sum represents a step of the resolution of one group of requests, coming from a semi-colored node, with tc (ts) the number of colored (semi colored) nodes in the sample. Second sum represents a step of the resolution of one group of requests, coming from a colored node. To do the calculation, we assume that requests from semi-colored nodes resolve faster.

$$
P(c, s, rc, rs) =
$$
$$
\sum_{tv=0}^{k+1} \sum_{ts=0}^{k+1} \binom{n-c-s+tv}{tv}\binom{s-tv-1+ts}{ts}\binom{c-ts}{k-tv-ts}P(c-ts, s-tv+ts, rc-ts, rs+1-tv)
$$
$$
+ \sum_{ts=0}^{k+1} \binom{n-c-s+rs}{tv}\binom{s-rs+ts}{ts}\binom{c-ts-1}{k-rs-ts}P(c-ts, s-rs+ts, rc+1-ts, 0)
$$

With these formulas, we can compute the probability of not reaching consensus : view Appendices

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

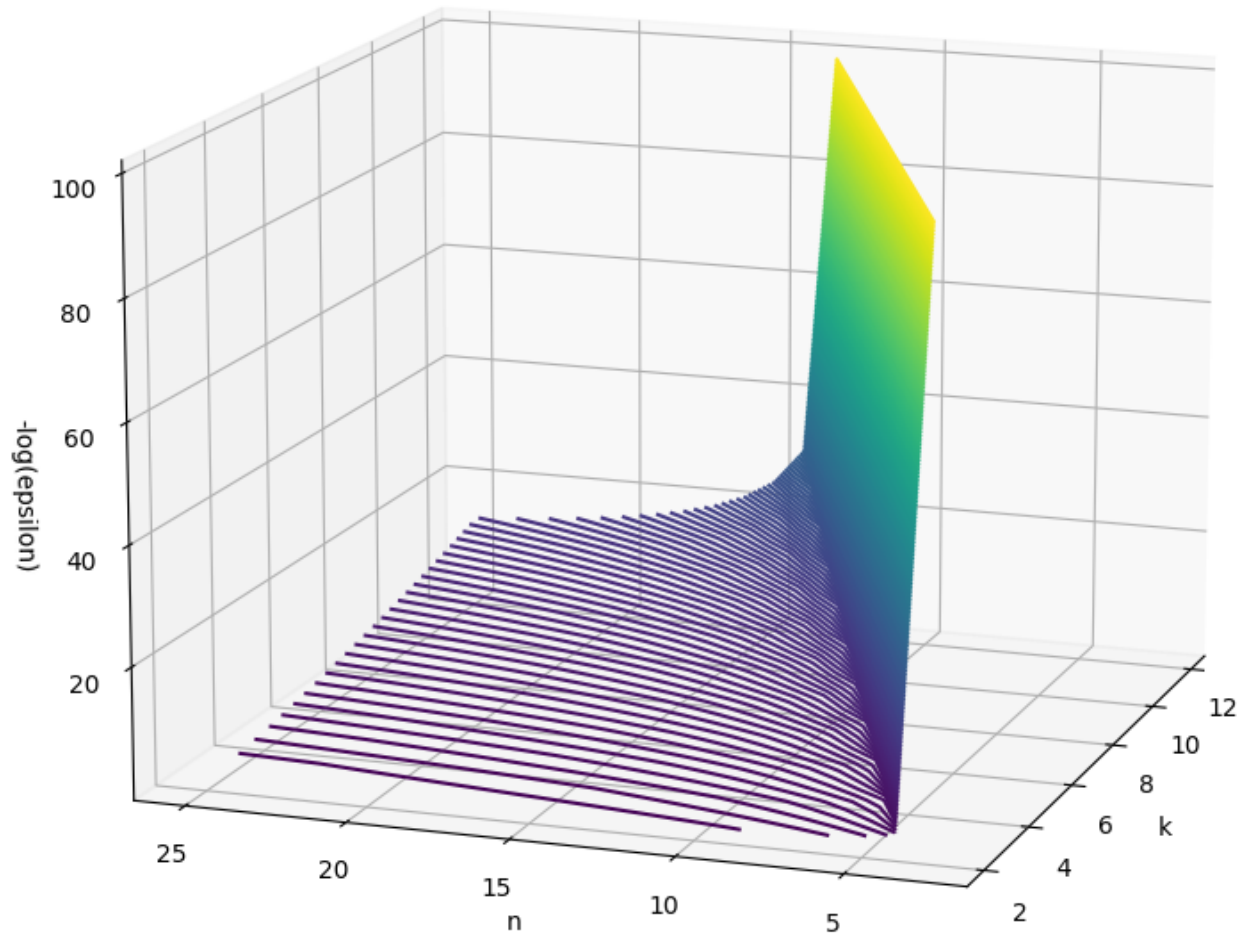[CL99] M Castro and B Liskov. Practical byzantine fault tolerance. 1999. `https://pmg.csail.mit.edu/papers/osdi99.pdf`.

[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`.

[Nak08] Satoshi Nakamoto. Bitcoin : a peer-to-peer electronic cash system. 2008. `https://bitcoin.org/en/bitcoin-paper`.

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

[RYS⁺19] Team Rocket, M Yin, K Sekniqi, R van Renesse, and E Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability. 2019. `https://assets.website-files.com/5d80307810123f5ffbb34d6e/6009805681b416f34dcae012_Avalanche%20Consensus%20Whitepaper.pdf`.

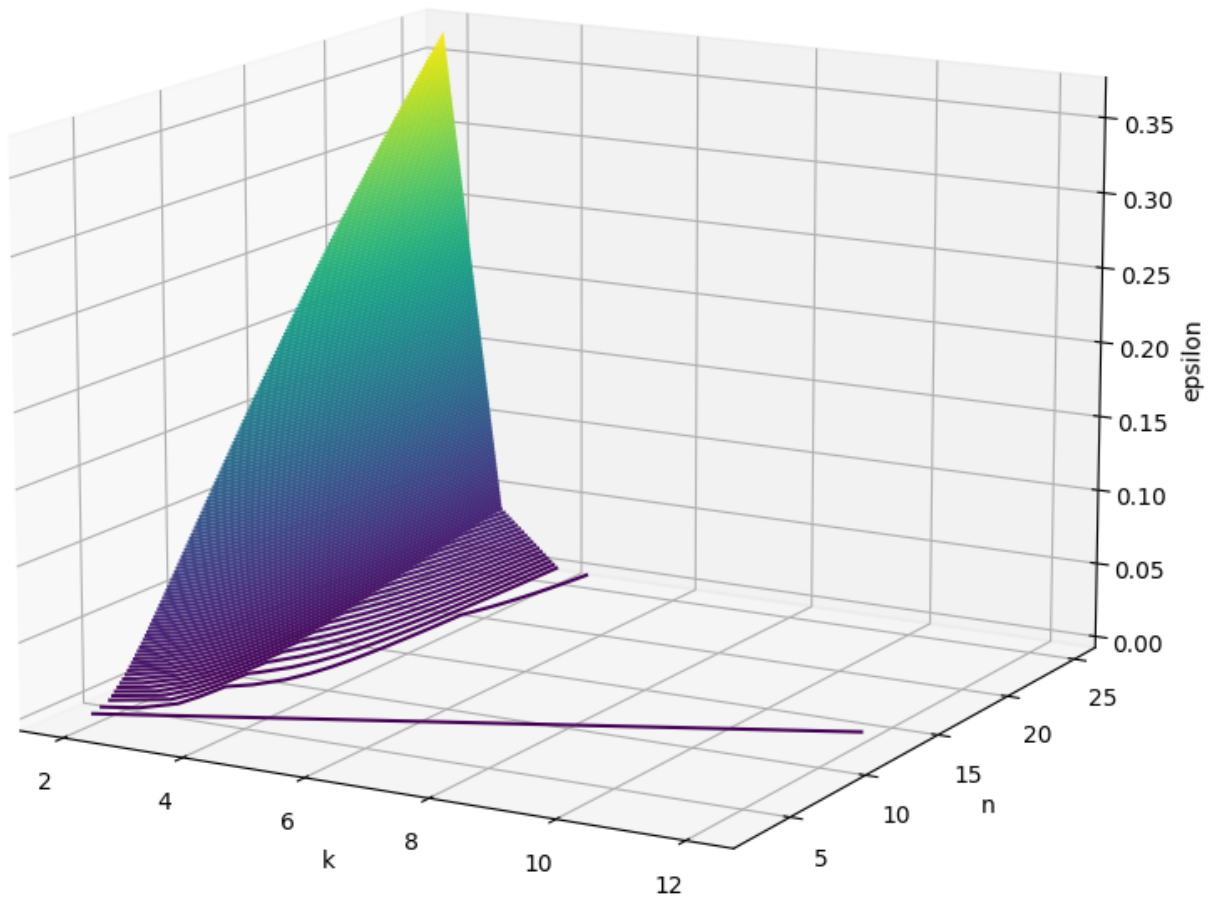[Sch90] F B. Schneider. Implementing fault-tolerant services using the state machine approach : a tutorial. 1990. `https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf`.

Appendices

Probability of not reaching consensus ($\beta = 1$) : ε
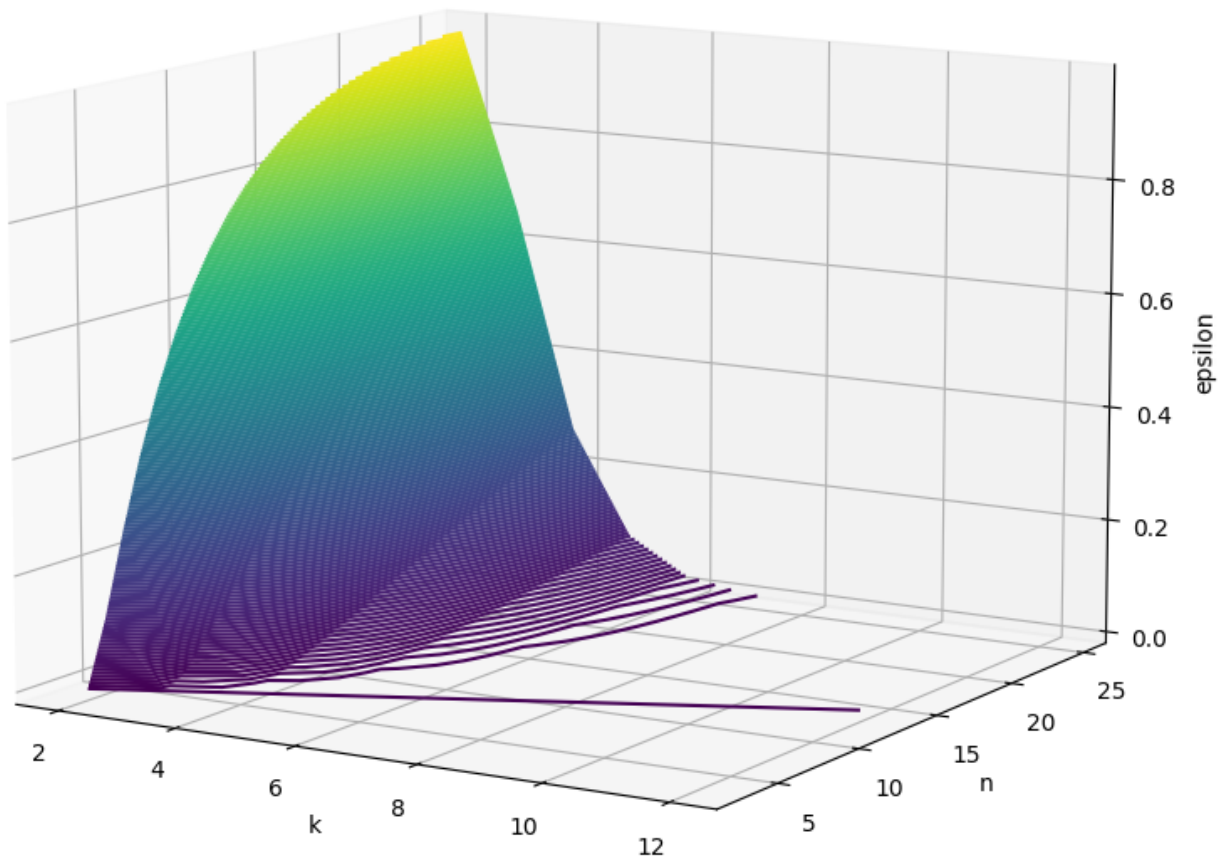
Probability of not reaching consensus ($\beta = 1$) : $-\log(\varepsilon)$

Probability of not reaching consensus (β = 2) : ε

Probability of not reaching consensus (β = 2) : -log(ε)