Bibliographic Survey
Research Project 2022

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

November 2, 2022

Introduction: Why Consensus ?

The boom of cryptocurrency that the world of tech/finance has seen in the past years (2017, 2021) shed light on a domain of computer science which is in fact much older: consensus algorithms. While the Bitcoin whitepaper was released in 2008 and the first block of the bitcoin blockchain was mined in early 2009, it is after around a decade of nearly flawless functioning of the bitcoin blockchain that institutions started looking at it, and investments started flowing in. Bitcoin introduced a new consensus algorithm, based on Proof-of-Work, an energy-consuming mechanism designed to resist attackers.

Nonetheless, the research on consensus algorithms goes back way before the internet even existed. So, why is consensus needed in computed science ?

Contents

1 The need for consensus **3**
1.1 Outside computer science . . . 3
1.2 In computer science . . . 3

2 Evaluation consensus protocols **4**
2.1 Defining the consensus . . . 4
2.2 Consensus and synchrony . . . 7
2.3 Fault Tolerance . . . 7
2.4 Consensus protocols existence boundaries . . . 9
2.5 Performance . . . 10

3 Review of some of the most popular consensus algorithms **11**
3.1 Paxos . . . 11
3.2 Raft . . . 12
3.3 PBFT . . . 13
3.4 Proof-of-Work . . . 13
3.5 Proof-of-Stake . . . 14
3.6 Avalanche . . . 14

4 Conclusion **15**

1 The need for consensus

1.1 Outside computer science

In order to better understand the world of consensus algorithms in computer science, we can take an example from an early democratic systems in Greece. Let's imagine that the law of a country is defined by a unique series of decrees that are passed with a majority of voters in a parliament [Lam98] . The parliament must function even though legislators constantly go in and out of the parliament. No single record of the law can exist, because those can be corrupted, and it would require one person to be in charge of the transcribing the votes all day long.

The parliament therefore requires a consensus protocol to make sure that legislators don't have contradictory information on there version of the law (e.g. 2 different decrees with the same number).

You might think that a majority vote is all what's required to establish new decrees, but some questions on the protocol quickly come to mind:

- Who is allowed to propose new decrees, and when ?
- How to make sure the consensus is legit when voters are wandering in and out of the parliament ?
- When a new parliamentary want the current state of law, where can he get it, or from whom ?

All those interrogations are discussed in the research on consensus algorithms.

1.2 In computer science

Let's imagine that one need to run a server to access online versions of self-published books. That is fairly easy, you need a computer that handles requests from clients, processes them through a search engine and then answers with a list of books corresponding to the search.

Now as soon as you need to scale the server, you will need the database to be distributed among multiple computers or servers. Then, what happens if a user, author of one of the book, releases a new version at tries to update the online version hosted on the server ? How can one make sure that the version will be available on all instances ? If we choose to have one "leader" from which all the servers copy the state of the data, how is that leader chosen by the protocol, and changed if that node crashes ?

When there is decentralised processes, there are consensus algorithms behind, such as in etcd [etc18], AWS[1] and cloud computing in general [aws18], bitcoin and blockchain [Nak08].

[1] Amazon Web Services

2 Evaluation consensus protocols

2.1 Defining the consensus

As we stated above, distributed consensus is the problem of getting multiple entities (servers) to agree on a shared state, even in the case of failures. That shared state is usually some data structure backed by a replicated log. The property we want from a consensus system is that it is fully operational, even if a minority of the servers fail. If the majority of servers are up, system is up.

Consensus among a set of decentralized nodes can have multiple definitions. We will review some of the definitions that are available in the the scientific literature before we study any consensus algorithm to have a better understanding of the range of situations these algorithms can apply to. Most consensus definitions have variants.

Consensus on one value The simplest form of consensus is the consensus on one bit (one boolean) and is defined as follows by Michael J. Fischer [Fis83]:

> Assume a fixed number of processors, some of which are initially faulty or may fail during the execution of the protocol. Each processor i has an initial bit x_i. The consensus problem is for the non-faulty processes to agree on a bit y, called the consensus value. More precisely, we want a protocol such that each reliable process i eventually terminates with a bit y_i, and $\forall i, y_i == y$. y in general will depend in some way on the initial bits x_i. In the absence of such a requirement, the problem becomes trivial, for each process can simply choose $y_i = 0$. Some dependency requirements that have been studied, in order of increasing strength, are:
>
> 1. (non-triviality): For each $y \in (0, 1)$, there is some initial vector x_i and some admissible execution of the protocol in which y is the consensus value. (The qualification "admissible" allows for additional restrictions, such as bounds on the number of faulty processes, on the kinds of computations we are willing to consider)
> 2. (weak unanimity): If $\forall i,\ x_i = x \in (0, 1)$, then $y == x$, provided that no failures actually occur during the execution of the protocol.
> 3. (strong unanimity): If $\forall i,\ x_i = x \in (0, 1)$, then $y == x$.

This definition is simple to understand but very restrictive.

A better definition Michel Raynal gives us a broader definition of the consensus problem in his book [Ray10] :

> In the consensus problem, each process p_i proposes a value v_i and all processes have to agree (we also say “decide”) on the same value that has to be one of the proposed values.

In a more precise way, this problem is defined by the following properties :

- Validity : If a process decides a value v, then v has been proposed by some process.
- Integrity : A process decides at most once.
- Agreement : No two processes decide different values.
- Termination : Each correct process decides.

Let A be an algorithm that is assumed to solve the consensus problem. This algorithm is correct if all its runs satisfy the previous properties. Validity, integrity and agreement define the consensus safety properties, while termination defines its liveness property.

As in others studies of consensus problems, Raynal highlights that the consensus can be solved in a trivial way in failure-free asynchronous systems, for example agreeing on the value decided by the process with the smallest number (assuming processes are numbered).

Interactive consistency Interactive consistency is a notion defined by M. Pease, R, Shostak, and L. Lamport in 1980 [PLS80].

Consider a set of n isolated processors, of which it is known that no more than m are faulty. It is not known, however, which processors are faulty. Suppose that the processors can communicate only by means of two-party messages. The communication medium is presumed to be fail-safe and of negligible delay. The sender of a message, moreover, is always identifiable by the receiver. Suppose also that each processor p has some private value of information V_p. [...] The question is whether for given $m, n \geqslant 0$, it is possible to devise an algorithm based on an exchange of messages that will allow each non-faulty processor p to compute a vector of values with an element for each of the n processors, such that :

- the non-faulty processors compute exactly the same vector;
- the element of this vector corresponding to a given non-faulty processor is the private value of that processor.

Note that the algorithm need not reveal which processors are faulty, and that the elements of the computed vector corresponding to faulty processors may be arbitrary; it matters only that the non-faulty processors compute exactly the same value for any given faulty processor. We say that such an algorithm achieves interactive consistency, since it allows the non-faulty processors to come to a consistent view of the values held by all the processors, including the faulty ones. The computed vector is called an interactive consistency (i.c.) vector. Once interactive consistency has been achieved, each non-faulty processor can apply an averaging or filtering function to the i.c. vector, according to the needs of the application. Since each non-faulty processor applies this function to the same vector of values, an exact agreement is necessarily reached.

Interactive consistency differs from consensus because instead of having the nodes agreeing on one common value, we want nodes agreeing on a vector which size corresponds to the number of nodes in the system.

Consensus stability The consensus stability clause is a clause to define the type of consensus. Different levels of consensus stability can be required depending on our problem.

The three main consensus stability values are, from the strongest consensus to the weakest :

- Irrevocable: Each agent has only one output value during the whole consensus process.
- Stabilization: Each agent's output values converge in finite time : there is a final stable value.
- Asymptotic : Each agent's output values converge.

Nakamoto consensus[2] used in the bitcoin blockchain lands between irrevocable and stabilizing consensus [Bos20].

When studying irrevocable consensus, we have to keep in mind an important theorem called FLP (Fischer, Lynch, Paterson) [FLP85]: In an asynchronous system of processes, every protocol trying to solve the consensus has the possibility of non termination, even with only one faulty process. Another way of stating FLP is to say that in an asynchronous system, there is no irrevocable consensus algorithm that tolerates one crash.

Uniform consensus The uniform consensus is introduced [CBS00] to tackle a specific case of consensus reach, which fits the consensus definition, but could still lead to system having a unwanted / unsustainable behaviour. In the case of a failure occurring on a process a long time after it was solicited, because the traditional consensus definition ignores faulty processes (and a faulty process could show a failure after being solicited), two processes could have reached consensus on two different values, one of the processes fail, and this way consensus is reached.

Indeed, the uniform consensus adds up a new condition (called uniform agreement) to the consensus and thereby strengthens the agreement: No two processes (whether faulty or not) can decide differently.

In synchronous systems, in the crash-failure model (no byzantine failure[3]), if the uniform consensus is required, one additional round is needed to decide, so uniform consensus is harder than regular consensus for crash failures.

[2]Proof-of-Work (3.4)
[3]Byzantine failures (2.3)

2.2 Consensus and synchrony

Synchrony and asynchrony Consensus algorithms and results will differ a lot between a synchronous and an asynchronous system. Let's define those terms : A synchronous system of processes has :

- message delays $< \Delta$
- relative computing speed $< \Phi$
- Δ and Φ are known

Those are the hypothesis that correspond to synchrony. Asynchronous systems are all systems not bound by those hypothesis

Assuming the system is synchronous changes the impossibility theorem for irrevocable consensus. While in an asynchronous system, the FLP result restricts most of the work on consensus, the impossibility result in synchronous system is a bit less strict on what cannot be done.

That result is called SPL, standing for Shostak, Pease & Lamport and dates back to 1982 [LSP82]. It states the following : In a synchronous system, there is no irrevocable consensus algorithm that tolerates $n/3$ byzantine agents.

We will discuss in the next section what a byzantine agent is.

Partial synchrony Finally, the last result worth mentioning states a possibility and not an impossibility in partially synchronous systems [DLS88].

Partially synchronous systems are systems laying between synchronous and asynchronous systems. One example would be a system in which messages delays and relative computing speed are bound, but we don't know the value of the bounds a priori. In another case, bounds might be known, but only apply starting as some unknown time T, whereas the system must work correctly at any time.

The result demonstrated by Dwork, Lynch & Stockmeyer (DLS) is the following : For partially synchronous systems, there is an irrevocable consensus algorithm that tolerates a minority of crashes.

The "minority" value depends on the type of crashes that the system is facing, the number of processes affected by those crashes and the type of partial synchrony that constitutes our hypothesis. The DLS paper gives the values for each situation of how many processes can be faulty for the whole system to stay resilient.

2.3 Fault Tolerance

Let's dive deeper in what we mean by fault tolerance in consensus algorithms. We already stated that having a consensus among a set of processes, of which none is faulty is trivial. We are now going to differentiate failure types in order to be able to assess different types of consensus algorithms.

Failures definitions come from [DLS88]. The definitions take place in a system defined as follows :

The communication system is modeled as a collection of N sets of messages, called buffers, one for each processor. The buffer of pi represents messages that have been sent to p_i , but not yet received. Each processor follows a deterministic protocol involving the receipt and sending of messages. Each processor p_i can perform one of the following instructions in each step of its protocol: $Send(m, p_j)$: places message m in p_j's buffer $Receive(p_i)$: removes some (possibly empty) set S of messages from p_j's buffer and delivers the messages to p_i .

Crash failures The crash failure is the most basic type of crash, and one of the most common. The process basically stops running. Therefore, it is unable to handle any requests for an undetermined period of time. Crash failures can occur in real life when the internet is down on a server, when there is a black out in the area (no more electricity), or if there is a maintenance for example.

Recently, the OVHcloud datacenter fire incident reminded the public that crashes can occur any time and anywhere.

The formal hypothesis for a processor p_i with crash failure is : processor p_i executes correctly, but can stop at any time. Once crashed it cannot restart

Omission failures The omission failure is considered a higher level of failure severity compared to the crash. A process p_i is subject to omission failure when :

- $Send(m, p_j)$ by p_i might not place m in p_j's buffer.
- $Receive(p_i)$ might cause only a subset of the delivered messages to be actually received by p_i.

Crash failures are sometimes referred to as a special case of omission failures, where the system can detect with certainty if that the process has crashed.

Byzantine failures Byzantine failures are the most malicious failures that can occur in a distributed system. They were defined by Lamport in the famous [LSP82] .

The adjective "byzantine" comes from an allegory to describe the problem, where several divisions of the byzantine army are camped outside an enemy city. Each division obeys its general and generals can communicate together via messenger. Their goal is to plan an attack. The notion of consensus comes from the need to plan something together.

However, not all generals are considered loyal. Some of them might be traitors to the cause. Indeed the generals must have a algorithm taking into account the possibility of traitor generals ("byzantine" processes in the system) to achieve the consensus (all loyal generals agree on the plan).

Formally, a byzantine process can have any arbitrary behaviour. We only assume that the receiver knows the identity of the sender of a message.

In the paper, the problem is described as so :

Setting Given a system of n components, t of which are dishonest, and assuming only point-to-point channel between all the components. Whenever a component A tries to broadcast a value x, the other components are allowed to discuss with each other and verify the consistency of A's broadcast, and eventually settle on a common value y.

Property The system is said to resist Byzantine faults if a component A can broadcast a value x, and then: If A is honest, then all honest components agree on the value x. In any case, all honest components agree on the same value y.

Variants The problem has been studied in the case of both synchronous and asynchronous communications.

The paper studies a few examples, and contains demonstrations for important results. For example, no solution can be found for oral messages if there are 3 generals, one of whom is a traitor.

Byzantine failures are studied in systems with and without authentication. With authentication, messages can be signed with the name of the sending processor. The signature is presumed impossible to counterfeit. Without authentication, there is no guarantee regarding authentication for byzantine processes.

2.4 Consensus protocols existence boundaries

To conclude on the possibilities to reach consensus depending on our system of processes, the type of fault that we want to study, the kind of consensus that we want and the level of synchrony that we assume, we display a table from [DLS88] with the smallest number of processors Nmin for which a t-resilient consensus protocol exists. (t being the number of faulty processes)

Table 1: SMALLEST NUMBER OF PROCESSORS N_{min} FOR WHICH A t-resilient CONSENSUS PROTOCOL EXISTS

Failure Type	Synchronous	Asynchronous	Partially synchronous communication and synchronous processors	Partially synchronous communication and processors	Synchronous communication and partially synchronous processors
Fail-Stop (crash)	t	∞	2t+1	2t+1	t
Omission	t	∞	2t+1	2t+1	[2t, 2t+1]
Authenticated Byzantine	t	∞	3t+1	3t+1	2t+1
Byzantine	3t+1	∞	3t+1	3t+1	3t+1

$Nmin = \infty$ means that irrevocable consensus is a priori impossible with these hypothesis [FLP85] The closed interval $[2t, 2t + 1]$ means that $2t \leqslant Nmin \leqslant 2t + 1$

2.5 Performance

Performance evaluation of a consensus algorithm One way of evaluating consensus protocols is to assess, for one system of processes, whether or not they resist a fail-stop failure, an omission failure and/or a byzantine failure. But if we want to be more precise in our assessment, for example to compare multiple Byzantine fault-tolerant protocol we need more criteria.

Studying the performance of consensus algorithms is not easy, as reported in [CUBS02] ("Consensus protocols are typically too complex for analytical approaches to performance evaluation") and [SDS01] limits the study to the pure speed of the protocol (providing that the consensus is indeed reached). According to Urban and Schiper [US05],

> Most of the time, consensus algorithms are evaluated using simple metrics like time complexity (number of communication steps) and message complexity (number of messages). This gives, however, little information on the real performance of those algorithms. [. . .]
>
> Most papers on the performance of agreement algorithms only consider failure free executions, which only gives a partial and incomplete understanding of the behavior of the algorithms

They suggest a benchmark based on measuring what they call the early latency of atomic broadcast. This value is measured under multiple conditions of workload (how much the system has to process) and faultloads (how many failing processors the system has).

Some protocols, like Bitcoin, implement Proof-of-Work as a part of their consensus algorithm, and we will discuss why Proof-of-Work is used, but many publications (like [Tru18]) raised the issue of its electricity consumption. Some even track the electric consumption of the network on a day to day basis [cam]. So energy consumption is a criteria to keep in mind when studying consensus algorithms. There are basically two states : either the consensus algorithm implements Proof-of-Work, or not and if not then the electricity consumption will be like that of any server, depending on the number of nodes participating.

Using the state machine approach Even if some papers had described the concept years before, the state-machine approach is first 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.

Using the state machine approach allows us to better assess the performance of consensus algorithm, because they are put in a situation where testing is easier : "Do I get the expected result when there is a crash fault / a byzantine fault on 1/2/n nodes in my system using this algorithm ? How long does it take ? ”

The Andrew benchmark [HKM+88] for example is used in the PBFT[4] paper [CL99] to state that their replicated file system supporting NFS[5] adds very little time cost compared to a standard NFS daemon.

3 Review of some of the most popular consensus algorithms

3.1 Paxos

Paxos was presented by Lamport [Lam98] to be a fault-tolerant consensus protocol. It can cope with processors experiencing failures, as long as they are not byzantine failures. Other assumptions are that the functioning processors are connected to each other. No speed requirement is done : processors may operate at any speed, and the same goes with connection in the network : messages are sent asynchronously and might take a long time to be delivered, or even get lost.

As we saw earlier in [FLP85], in a completely asynchronous environment, progress is never guaranteed, even if only one processor fails. However, by measuring time it is easy to fall in the “partial synchrony” case of our table. The conditions that would prevent Paxos from making progress are rarely met in practice. In the paper, it is said that

> Achieving the progress condition required that legislators be able to measure the passage of time.

Paxos is now considered a family of protocols because of the many variants that have been created over time. We will first review the most basic version of it and then some of the developments that can be made.

Vanilla Paxos In the Paxos algorithm, each processor can play a role at a certain time. Those roles are Acceptor (voters), Proposer, Learner and Leader

In the Part-Time Parliament, multiples protocols are described, always in the metaphoric description of the parliament. Priests (p) in the parliament are the processors in the protocol. The first protocol is called the preliminary protocol but following this one a second protocol called the basic protocol is the one who stayed as the Paxos algorithm. It is easier to implement than the preliminary protocol because the priests (the processors) have to keep less information in their ledger (in memory) to be a part of the protocol.

Those informations are (for a priest p) :

1. $lastTried[p]$: The number of the last ballot that p tried to initiate, or $-\infty$ if there was none.

[4]Practical Byzantine Fault-Tolerance

[5]Network File System

2. $prevVote[p]$: The vote cast by p in the highest-numbered ballot in which is voted, or $-\infty$ if he never voted

3. $nextBal[p]$: The largest value of b for which p has sent a $LastVote(b, v)$ message, or $-\infty$ if he has never sent such a message.

The basic protocol (which I won't copy here) guarantees consistency, which for recall means no two ledgers can contain contradictory information (different decrees with the same number); but it doesn't guarantee progress because it only states what the priest can do, not what they have to do. In order to achieve progress, Lamport introduces the Complete Synod Protocol, where priest must complete some of the steps of the protocol as soon as they can. With a few hypothesis on priest reacting to messages and messages being delivered, the Complete Synod Protocol guarantees progress if and only if only one priest who doesn't live the chamber propose new decrees. The complete Paxos protocol therefore includes a process to select a leader (the president). The president selection requirement is the following :

> If no one entered or left the Chamber, then after T minutes exactly one priest in the Chamber would consider himself to be the president.

Some details are given on how the new president becomes the new president, but it is similar to a decree proposal process.

Variants Multi-Paxos is the main extent of Paxos, introduced in the Part-Time Parliament as the Multi-decree parliament. It addresses passing multiple decrees instead of just focusing on passing one. This leads to some improvements in the runs, compared to running the basic Paxos time and time again. For example, if the leader is elected in one Paxos run and does not fail, he can be used in the next run as well, without compromising any of the properties of Paxos.

Let's say we use Paxos to provide access to a distributed state machine. Therefore, computing a Paxos run each time the service is queried by a client would result in a significant amount of unecessary processing, compared to a leader-based process with Paxos runs only to tackle failures. Most of the time, variants of Paxos would be implemented, not the original Paxos mentioned as the Basic Protocol.

Some people have since worked on other variants. Lamport himself worked on a Fast Paxos algorithm [Lam05] (basically reducing the number of message delays when the information comes from a client query).

3.2 Raft

The Raft algorithm was introduced in In search for an understandable consensus algorithm [OO13]. The Raft is presented as as efficient as Paxos and producing a result equivalent to (multi-)Paxos.

It is also supposed to be more understandable than Paxos, and with a different structure. Just like Paxos, the Raft can handle stop failures, but not byzantine failures.

Raft basically relies on a stronger leader than Paxos (in which the leader is introduced only for performance purpose) with followers only responding to the leader's request as long as there is no failure. The third state (leader and follower being the two first) is candidate and is used in the case of a leader failure triggering a new leader election.

3.3 PBFT

PBFT, standing for Practical Byzantine Fault Tolerance is a consensus algorithm introduced in 1999 by Miguel Castro and Barbara Liskov [CL99]

PBFT is described in the context of the state machine approach, which means that the goal is a bit more than just reach consensus between the nodes, but we will here isolate the consensus algorithm laying behind the description of the paper. The algorithm improves protocols presented in other papers, Rampart and SecureRing, which relied on the synchrony assumption.

The algorithm tolerates f faulty nodes if there are 3f+1 nodes in the system, and as said before adds very little time cost compared to a standard implementation of the same service (NFS in this case).

PFBT uses cryptography to authenticate the nodes in the exchange. In the basic version (section 4 of the paper), digital signatures are used to authenticate all the messages. In the final version described in the paper however, in order to optimize performance, digital signatures are used only for view-change and new-view messages and all other messages are authenticated using message authentication code (MACs).

3.4 Proof-of-Work

Even if it is often referred as so, Proof-of-Work is not in itself a consensus algorithm. Proof-of-Work was at inception a denial-of-service counter-measure [has]. Proof-of-Work algorithms allow users to show that they computed some amount of work, for example when using a service.

Without Proof-of-Work, an attacker can potentially send millions of request to a service that has to process them in what is known as a denial-of-service attack. If the services requires the request to contain a Proof-of-Work stamp, the attacker would need for example one second to compute each stamp, which would need a lot of power and time to do the attack, in practice avoiding this kind of attacks completely.

In the bitcoin blockchain [Nak08], Proof-of-Work is used to build the consensus algorithm which consists in giving trust to the longest chain[6]. This is sometimes referred to as the Nakamoto consensus, like in [Bos20]. The longest chain can only be built if you have the majority of computing power on the network. Economical rationality makes it that nodes want to play with the rules to win

[6]Longest chain of blocks in the blockchain, see [blo] or [Nak08] for details

the newly created coins (when one block is mined, the successful node (miner) gets a compensation for his work), therefore “avoiding” the byzantine generals problem.

The bitcoin network is permissionless, and theoretically, any number of byzantine nodes (without computing power) can join the network without affecting its security.

I included Proof-of-Work because it is the technology that allowed all the developments of the crypto world but studying it deeply is beyond the scope of this project.

3.5 Proof-of-Stake

The first occurrence of Proof-of-Stake can be found behind the cryptocurrency PPCoin (or PeerCoin) [KN12], a project receiving very little interest nowadays. However, the technology gained popularity over time and nowadays much more cryptocurrencies are based on Proof-of-Stake than on Proof-of-Work [coi]. The second biggest cryptocurrency by market cap[7], Ethereum, even moved from Proof-of-Work to Proof-of-Stake not long before these lines were written, in one of the biggest event the world of crypto has ever seen (known as The Merge).

Proof-of-Stake consists in replacing the trust from computing power with the trust from holding the coins of the ecosystem, therefore proving that someone has something to loose by being dishonest.

Again, I’m not going to go further in the details of the different implementations of Proof-of-Stake, how inflation is managed in these ecosystems, that is a whole other topic. The Ethereum Foundation claims that Proof-of-Stake is more secure, less energy-intensive, and better for implementing new scaling solutions [eth]. At least, it tackles the biggest criticism on cryptocurrencies from the mainstream media, which is that Proof-of-Work uses too much power [bit21].

3.6 Avalanche

Avalanche introduces a new family of Byzantine Fault Tolerant Protocols [RYS+19]. The Avalanche blockchain which relies on the Snowball protocol reached 30 billion USD market cap at its peak. Avalanche consistently stands in the top 10 public blockchains since 2020.

Slush, Snowflake, Snowball are the three main variants that are presented, each being more complex than the previous one.

Unlike Paxos, Raft and PBFT, in which all correct nodes reach the consensus with absolute certainty the Snow protocol family, like the Nakamoto consensus, may revert with some probability ε. It is not an irrevocable consensus algorithm.

I won’t get here into the details of the algorithm and why it works. Basically each node, when it receives a new state, updates it own state with the information and picks a small sample of the network to propagate its new own state. The details vary between Slush (non BFT), Snowflake (BFT) and Snowball (adding “confidence”).

[7]Market capitalization (or market cap) is the total value of all the coins that have been mined. It is calculated by multiplying the number of coins in circulation by the current market price of a single coin.

4 Conclusion

The question of reaching consensus among a set of distributed nodes in a vast topic, which started from distributed computing and distributed database mostly to prevent hardware failures and scale infrastructures. Nowadays most of the research is done around blockchains and how to prevent Byzantine failures with fast and secure consensus protocols.

Sacrificing the assurance of reaching a irrevocable consensus for a probabilistic approach seems to be the norm among public blockchains. Indeed, as they are public, open and permissionless, having one node trying to reach consensus by sending requests to a quorum of nodes which have to validate, like Paxos algorithms do is too slow and heavy.

References

[aws18] Aws decentralized data management, 2018. https://docs.aws.amazon.com/whitepapers/latest/running-containerized-microservices/decentralized-data-management.html.

[bit21] Bitcoin uses more electricity than many countries. how is that possible? *The New York Times*, 2021. https://www.nytimes.com/interactive/2021/09/03/climate/bitcoin-carbon-footprint-electricity.html.

[blo] Wikipedia: Blockchain. https://en.wikipedia.org/wiki/Blockchain.

[Bos20] Bernadette Charron Bost. Problèmes de consensus, 2020. https://www.lix.polytechnique.fr/news/253/view.

[cam] Cambridge bitcoin electricity consumption index. https://ccaf.io/cbeci/index.

[CBS00] B Charron-Bost and A Schiper. Uniform consensus harder than consensus. 2000. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.7181&rep=rep1&type=pdf.

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

[coi] Coinmarketcap. https://coinmarketcap.com.

[CUBS02] A Coccoli, P Urban, A Bondavalli, and A Schiper. Performance analysis of a consensus algorithm combining stochastic activity networks and measurements. 2002. https://infoscience.epfl.ch/record/49941/files/CUB+02.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.

[etc18] etcd.io, 2018. https://etcd.io.

[eth] Ethereum proof-of-stake. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems : A brief survey. 1983. https://apps.dtic.mil/sti/pdfs/ADA129834.pdf.

[FLP85] M J. Fischer, N A. Lynch, and M S. Paterson. Impossibility of distributed consensus with one faulty process. 1985. https://ilyasergey.net/CS6213/_static/02-consensus/flp.pdf.

[has] Hashcash. http://hashcash.org/.

[HKM$^+$88] J H. Howard, M L. Kazar, S G. Menees, D A. Nichols, M. Satyanarayanan, R N. Sidebotham, and M J. West. Scale and performance in a distributed file system. 1988. https://www.cs.cmu.edu/~satya/docdir/howard-tocs-afs-1988.pdf.

[KN12] S King and S Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012. https://people.cs.georgetown.edu/~clay/classes/fall2017/835/papers/peercoin-paper.pdf.

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

[Lam05] Leslie Lamport. Fast paxos. 2005. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.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.

[PLS80] M Pease, L Lamport, and R Shostak. Reaching agreement in the presence of faults. 1980. https://dl.acm.org/doi/pdf/10.1145/322186.322188.

[Ray10] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems.* 2010.

[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.

[SDS01] N Sergent, X Defago, and A Schiper. Impact of a failure detection mechanism on the performance of consensus. 2001. https://infoscience.epfl.ch/record/49973/files/SDS01.pdf.

[Tru18] Jon Truby. Decarbonizing bitcoin: Law and policy choices for reducing the energy consumption of blockchain technologies and digital currencies. 2018. https://www.sciencedirect.com/science/article/abs/pii/S2214629618301750.

[US05] P Urban and A Schiper. Comparing distributed consensus algorithms. 2005. https://infoscience.epfl.ch/record/49903.