How to reach consensus with strangers

Albert Garreta

Follow

Nethermind.eth

--

Listen

Share

by Albert Garreta and Isaac Villalobos Gutiérrez. Special thanks to Michal Zajac and Ahmet Ramazan Agirtas for helpful comments and revisions.

In this post, we discuss the notion of Byzantine Fault Tolerant (BFT) consensus protocols

. To start with, we offer an overview of the main concepts and their properties. Following that, we describe a large family of such protocols, and we discuss the world of synchronous BFT consensus, getting into the details of OptSync, a state-of-the-art synchronous protocol.

This post can be read either on its own or as the third installment of our Distributed Validator Technology (DVT) series and the primitives that enable it. In the first post, we gave an overview of DVT, while in the second, we discussed Distributed Key Generation protocols.

Table of contents

Introduction

Let's begin with a classic example that explains some of the terms used in the field of consensus algorithms.

Suppose an army is trying to conquer the city of Byzantium. The army is distributed across divisions, with one general in command per division. The army has managed to surround the city. Still, the defenses are more resilient than expected, and the generals are now trying to decide whether they should continue the attack or withdraw their divisions. The problem is that, for the attack to have a chance of succeeding, a majority of divisions need to remain. Otherwise, the army will not be large enough.

Hence, the generals face the following task: they need to communicate with one another to reach a joint decision. In this example, we assume the generals cannot meet in person and instead use "raven mail" to communicate . There are a couple of iss method:

- Ravens can get lost or arbitrarily delayed.
- Some of the generals could be secretly allied with the city of Byzantium. i.e., some of the generals could be Byzantine

.

This classical scenario is commonly called the Byzantine Generals Problem.

It showcases the need for robust methods that allow a group of parties to reach a coordinated decision, even if some are unreliable or malicious.

In some settings, it is common to assume that no generals are dishonest. This makes the problem less demanding, and consequently, one can devise efficient solutions which would not work in our more general setting. We will not discuss these in this post.

In modern terms, consensus protocols are fundamental to distributed computing technologies, including many blockchain systems and applications built around them. Indeed, for example, Nakamoto consensus (sometimes referred to as "Bitcoin's Proof of Work") is one

among many consensus protocols. And yes, more generally, all Layer 1's are built around such a mechanism.

Besides blockchains, consensus is essential in many other applications, such as cloud computing, clock synchronization, PageRank, etc. Additionally, as we discussed in an earlier post, it is a crucial piece of Distributed Validator Technology (DVT), where a group of nodes running an Ethereum validator must agree on what block to attest to or propose.

Next up, we will go through a few crucial notions and definitions that underpin consensus mechanisms. We will describe a "meta-protocol" that many PBFT-based protocols follow, and briefly outline IBFT/QBFT, a partially synchronous protocol used by companies working on DVT technology such as Obol and Blox.io.

We strongly recommend the blog Decentralized Thoughts for tons of excellent material on consensus and other cryptography topics.

Crash course on Byzantine Fault Tolerant consensus protocols

From now on, we call each party participating in a consensus protocol a replica

. We let n be the number of replicas

. A replica is called Byzantine

if either:

- It is actively trying to sabotage the successful functioning of the protocol.
- It is malfunctioning or offline.

Otherwise, we say the replica is honest

. Perhaps counterintuitively, in this context, a replica that is trying to follow the protocol but is offline or malfunctioning due to external circumstances would not count as an honest replica, even if it has the best of intentions.

We let f be the number ofByzantine replicas.

The problem: State Machine Replication

What goal are the replicas trying to achieve? Assume a client is interacting with a program, and as this happens, it submits requests to a mempool. The objective of each replica R is to create an ordered list of blocks

containing requests and to execute the requests in the order specified by this list. We denote such a list as C(R). Moreover, this needs to be done in a way where the following requirements are met:

- Safety (or Security)

: For any two honest replicas, R_1 and R_2, one of the lists among $C(R_1)$ and $C(R_2)$ extends the other list. For example, if $C(R_1)=(B_1, B_2)$ and $C(R_2)=(B_1, B_2, B_3)$, then $C(R_2)$ extends $C(R_1)$ and we are happy. But if $C(R_1)=(B_1, B_2)$ and $C(R_2)=(B_1, B_3)$, then there has been a security breach (Here each B_i denotes a different block).

- Liveness

: Each client request is eventually included in some block of C(R), for each honest replica R.

The above is essentially the definition of the State Machine Replication (SMR)

problem. Other common (less demanding) problems in the BFT consensus literature are [Byzantine Broadcast and Byzantine Agreement

](https://decentralizedthoughts.github.io/2019-06-27-defining-consensus/).

Note that even though our terminology evokes the blockchain space, the formulation above can accommodate various unrelated applications.

Next, we focus on two of the most critical assumptions one can make in this context: the communication model and the number of tolerated Byzantine replicas.

Communication model

An important characterizing property of consensus protocols revolves around the assumptions made on how replicas communicate with each other. These are the most common assumptions in the literature:

- Synchronous communication model

: There is a time bound Δ, known to all replicas, such that, for any two replicas R_1, R_2, any message that R_1 sends to R_2 takes at most Δ time to reach R_2.

- Asynchronous communication model

: Messages can take an arbitrary amount of time to travel from one replica to another.

- Partially synchronous communication model

: There is a time bound GST (Global Stabilization Time), unknown to at least all honest replicas, such that, after GST time has passed since the start of the protocol, the communication becomes synchronous.

An example

may help illustrate the motivation behind the last model: suppose the replicas can communicate at a reasonable speed, but suddenly there is an “exceptional event” — e.g., a country-wide blackout or a big server bug — that makes communication highly delayed and unpredictable. It is reasonable to expect that, after an indeterminate amount of time, communication will return to its usual timely functioning. The partial synchronous model aims to capture this type of scenario. See this post by Ittai Abraham for further details about this model.

Further on, we will classify well-known protocols according to their communication model and Byzantine Fault Tolerance threshold (see below).

BFT protocols

This blog post considers only Byzantine Fault Tolerant (BFT)

protocols. These algorithms guarantee security and liveness even if some parties participating in them are malicious. There are other families of protocols that make alternative assumptions about the possible behavior of parties. For example, protocols used by companies may only assume that, at worst, a “party” can crash. Most definitions given here apply to such protocols as well.

Maximum fraction of Byzantine replicas, or BFT threshold

This specifies the maximum ratio f/n of Byzantine replicas the BFT protocol can tolerate while guaranteeing security and liveness. The communication model in use heavily determines this ratio. More precisely, one can achieve safety and liveness if:

- Synchronous model:

Less than half of the replicas are Byzantine, i.e., f<n/2.

- Partially synchronous model

: Less than one-third of the replicas are Byzantine, i.e., f<n/3.

- Asynchronous model

: It is impossible to build a secure and lively protocol in this model! For proof of this fact, see the following video by Tim Roughgarden or the original paper presented in 1985.

Both thresholds above are sharp, i.e., it is proven that there do not exist secure and lively protocols that can tolerate exactly or more than n/2 or n/3 Byzantine replicas, respectively. We refer to this post by Ittai Abraham for references and an overview of these types of thresholds.

Concerning asynchrony, if f<n/3, then the impossibility result disappears if one relaxes the liveness (or the security) requirements. For example, it is possible to build an asynchronous protocol with f<n/3 if, instead of liveness, one only requires that a request seen by all honest replicas is eventually included in each of the honest replicas’ list of blocks C(R) with very high probability

.

PBFT-based protocols

A classic BFT protocol is the so-called Practical Byzantine Fault Tolerant (PBFT) protocol (1999). To the best of our knowledge, this was the first BFT protocol to be used in practical settings. Many subsequent algorithms can be seen as variations and extensions of this one, e.g., Gasper, Tendermint, IBFT, QBFT, HotStuff, etc.

This post will use the non-standard term PBFT-based protocol

to refer to a family of algorithms of this kind, and going forward, we will almost exclusively focus on them.

Leaders

PBFT-based protocols (and many others) assign to some replicas the special role of leader

. Such a leader is usually responsible for :

- Proposing blocks,
- (At times) managing the communication between replicas (to reduce the communication complexity),

- Performing a variety of tasks specific to each protocol.

We cannot have the protocol rely on the leader replica to function honestly. Otherwise, such a leader may just as well process and finalize blocks on its own. For this reason, all protocols that rely on leaders implement a leader-change mechanism

(sometimes called a view-change mechanism

).

Such mechanism is used when the network realizes that the leader is not doing its job as expected, or there is a scheduled change of the leader.

Steady-state and view-change

PBFT-based protocols progress through a series of views

. Each view v is a positive integer and corresponds to a leader L_v. A view-change

is a subprotocol that makes it possible to increase the view number v by one so that the replica L_u replaces replica L_v as the leader, where u=v+1.

There are two trends in regards to when view-changes are executed:

- Some protocols do not perform a view-change until the leader is detected to misbehave. In particular, in this type of protocol, if a leader is perpetually honest, there will never be a view-change.
- Other protocols execute a view-change every time a block is finalized. This is common of PBFT-based protocols used in Layer 1's such as Gasper, Tendermint, IBFT, QBFT, HotStuff, etc. Another example is Nakamoto consensus (though this one is not "PBFT-based" and does not use the "view" terminology).

These protocols are sometimes referred to as leader-rotating.

As long as a replica is not trying to change the leader, we say the replica is in steady-state

.

Block finalization, latency, throughput, and communication complexity

We say that an honest replica R finalizes

(or commits

) a block when it includes it in its list of blocks C(R). The latency

of the protocol is the amount of time that passes between the moment a designated replica (for example, the leader) includes the request in a block proposal and when all honest replicas have finalized a block containing such a request.

Once a communication model and a Byzantine Fault Tolerance threshold have been fixed, the latency of the protocol is perhaps the property that receives the most attention in the literature.

The protocol's throughput

is the inverse of the time T that passes between the finalization of two consecutive blocks into all the lists C(R) of honest replicas. This T is at most the latency, but T can be significantly lower. For example, several "instantiations" of the same core consensus protocol can be run in parallel, giving the overall setting a high throughput without altering the latency.

The communication complexity

refers to the total number of messages the replicas send each other, from when a designated replica (the leader) includes a client request in a block proposal, to when the request is included in a finalized block by all honest replicas.

Sometimes only digital signatures are counted regarding communication complexity, as these are often the largest pieces of data interchanged between replicas.

Below we list a few well-known consensus protocols and their properties.

(Links for each protocol are provided at the end of the post)

(1): The security of Nakamoto Consensus is a nontrivial topic. See, for example,this paper by Ling Ren (2019). Roughly, the "right setting" consists of having blocks produced at least every 10 minutes.

(2): Algorand's proof of liveness assumes synchronous communication. However, Algorand retains security under asynchronous communication.

(3): Security and liveness proofs in Snowflake are guaranteed to hold with high probability p. The BFT threshold can be increased at the expense of reducing p.

A general recipe for PBFT-based consensus

As we mention, PBFT-based protocols are quite similar. For this reason, in this section, we present a “general scheme” for such protocols. Later, we will describe a particular instantiation of this scheme as the so-called OptSync protocol.

We assume that honest replicas use digital signatures and a Public Key Infrastructure (PKI) to guarantee that messages are untampered and authenticated.

For each view v we let L_v denote the leader during view v. While in view v, each replica R (including the leader) follows the next steps (see the figure below as well):

Propose step

The replica R ignores this step unless R is the leader L_v.

- L_v collects requests from a mempool and bundles them into a block B.
- L_v broadcasts a

message containing B, the block B is attached to, and a signature of L_v (and perhaps more data).

Voting step(s)*

Intuitively, R verifies that a block proposal satisfies several security properties (specified by the particular protocol in use). If so, it broadcasts a message to all replicas attesting to this fact. Otherwise, it broadcasts a complaint against the leader, and it may proceed to execute the view-change mechanism.

More precisely:

1) Upon receiving a

message, the replica makes several security checks. These are specific to each protocol, but usually, they include making sure that the following (or similar) conditions are met:

- The message signature is valid.
- B is well-formed and points to a “valid” parent block.
- The message

is not incompatible with other “propose” messages received during view v.

- Less than (1-f)n replicas have complained against the leader (such complaints may be triggered by the leader failing to propose valid blocks — e.g., by being inactive).

2) If some of these checks fail, the replica broadcasts a complaint message along with evidence supporting such a complaint (if possible). If the detected issues are severe enough, the replica exits the view and proceeds to execute the view-change mechanism. By “severe” we refer to issues that unequivocally prove that the leader is Byzantine.

3) Otherwise if some (protocol specific) extra conditions are met R sends a

message to all replicas.

The figure below schematizes how the “vote step” works.

(*) Different protocols have a different number of instantiations of this step. That is, state-of-the-art protocols usually have two or even three “vote steps” (with a different name each). For example, Tendermint has two vote steps, Prevote and Pre-commit; and HotStuff has three, Prepare, Pre-commit, and Commit.

The 2nd vote step is essentially like the 1st one, except that the replica now expects to receive messages of the type

instead of messages of the form

, and it broadcasts

once certain protocol-specific conditions are met (e.g. once it has received at least (1-f)n

messages for the same block B). Similarly, in the 3rd vote step, R receives messages of the form

, and it broadcasts messages as

, etc.

Finalization and Execution step

This step works similarly to an extra vote step, with the difference that, at the moment when R would vote for a message, here R includes B in its list of finalized blocks C(R), and executes the requests contained in the block.

As we mentioned, PBFT-based consensus also relies on a subprotocol that allows to replace the current leader. For brevity, we will not describe this in this post.

The protocols used in Distributed Validator Technology

As we mentioned in the table above, the consensus mechanism currently used byObol and Blox is the so-called Quorum BFT (or just QBFT) consensus protocol. This is a variation of the Istanbul BFT (or just IBFT) protocol, which achieves a better message complexity of $O(n^2)$. It follows our general scheme closely, having two vote steps (called “prepare” and “commit”). The conditions in Step 2.3 consist precisely in receiving at least 2n/3 Propose or Vote messages from the previous step. Similarly, in Step 3, a block B is executed and included in the replica’s list of finalized blocks only when at least 2n/3 messages of the form

are received.

The protocol includes a so-called block-locking mechanism,

akin to Tendermint’s, to ensure safety after performing a view-change.

Synchronous protocols

In what follows, we opt to specialize our discussion to a synchronous communication model. This is because synchronous BFT protocols are less known in the blockchain space than partially synchronous ones. For further information on specific partially synchronous protocols, we refer, for example, to this excellent post by Ittai Abraham discussing PBFT, SBFT, Tendermint, and HotStuff. We note that most of these protocols adhere quite closely to the general scheme we presented above.

Overview

In this section, we assume a synchronous communication model. We let Δ be the publicly known pessimistic maximum communication delay, and we let δ denote the actual network delay

. This parameter is an approximation of the actual delivery time of messages within the network of replicas, while Δ is a pessimistic “worst-case scenario” bound for such time. In general, δ is much smaller than Δ.

Perhaps counterintuitively, synchronous protocols generally have significantly more latency than partially synchronous ones.

Indeed, partially synchronous protocols can achieve a latency of $O(\delta)$ (think, for example, something like 4δ to 8δ) when communication works normally. For example, HotStuff is one such protocol — we note that Tendermint has a $O(\Delta)$ bottleneck in its view-change mechanism, even during periods of fast communication. In contrast, synchronous protocols must have a latency of $O(\Delta)$ when f is close to n/2, even if all honest replicas communicate at network speed δ

(see this paper by Shrestha et al).

A sub-threshold: fast path and slow path

On the other hand, a protocol calledThunderella was presented in 2017, which is capable of finalizing blocks at network speed — i.e., with $O(\delta)$ time — when $f<n/4$ (i.e., when less than a fourth of the replicas are Byzantine). This and subsequent protocols (for example, Sync HotStuff) feature fast

and slow paths

. Each of these “paths” is a subprotocol that is activated when $f<n/4$ or $n/4\leq f <n/2$, respectively. The corresponding latencies are $O(\delta)$ and $O(\Delta)$, respectively.

However, such protocols need an extra mechanism that allows honest replicas to know which path they should follow (i.e., the replicas need to spend communication resources trying to estimate the current value of f) and how to switch securely from one path to the other.

This “path-switching” process represents a costly one in terms of latency. For example, in Thunderella, this cost is $O(\kappa\Delta)$ where κ is a security parameter. Hence, protocols following the “fast-and-slow path paradigm” may not be the best fit for environments where f changes frequently. Furthermore, n/4 coordinated Byzantine replicas could cause endless “path-switching” triggers by successively misbehaving for a while and then behaving correctly for another while. This could effectively DoS the whole protocol.

Optimistic and responsive block finalization

One of the first synchronous BFT protocols to address this issue is the [OptSync

](https://eprint.iacr.org/2020/458) protocol (see also thePiLi and Hybrid-BFT protocols).

Intuitively speaking, instead of switching from a fast to a slow path and vice-versa, OptSync has two subprotocols running simultaneously. Let’s call them the optimistic subprotocol

and the synchronous subprotocol

. Roughly, they work and interplay as follows:

- If f<n/4, then the optimistic subprotocol

is guaranteed to finalize blocks with latency O(δ).

- If n/4≤ f<n/2, in general, the optimistic subprotocol will not be able to reach a consensus. However, the slower synchronous subprotocol will be there, waiting to save the day

, guaranteeing that blocks are finalized within time O(Δ).

Next, we provide a rather detailed description of OptSync's steady-state. We describe it as an instantiation of the general scheme outlined previously. Some arguments we use are standard arguments from the PBFT-based consensus literature; hence they are of general interest.

Description of OptSync (steady-state)

The authors introduce two versions of the protocol which are Byzantine Fault Tolerant. We refer to these as OptSync V1 and V2. They also describe a third version that does not tolerate Byzantine replicas. The latencies of OptSync V1 and V2 are annotated below. Note that the first version has a slightly faster steady-state, while the second is able to replace leaders at network speed when f<n/4.

We will describe the simplest of OptSync's versions, namely V1 in the table above.

Block organization

For our explanations to make complete sense, we need to be a little more elaborate regarding how replicas store the blocks they receive.

In the protocol, each block points to a predecessor block.

Honest replicas store a [Directed Acyclic Graph (DAG)

](https://en.wikipedia.org/wiki/Directed_acyclic_graph)of blocks

, which is updated with each non-erroneous (more on this later) block they receive.

As with the list of blocks C(R), each honest replica R stores its version of such a DAG. Two blocks in the DAG equivocate

one another if neither is an ancestor of the other (i.e., there is no path from one to the other).

If the protocol functions correctly, then for each replica R, the stored DAG has a path C(R) of finalized blocks. In particular, C(R) contains no two equivocating blocks.

Notice that the DAG is only used to organize the blocks that each replica sees. Adding a block to the DAG does not necessarily mean that it is finalized or has any validity.

Below we schematize how such a DAG may look like.

We now proceed to describe the steady-state of OptSync.

We present a simplified protocol version, omitting several technicalities for clarity and space.

As usual, we let v be a view number and let L_v be the leader of view v. While in view v, an honest replica R follows the below protocol.

Propose step

This step is essentially the same as the one outlined in our general scheme.

Vote step

Upon receiving

from the leader, if no error

is detected (see below), then:

- R broadcasts a message of the form
- R adds B to its DAG.

We call each of the following an error

:

- B equivocates a block from R's list of finalized blocks C(R).
- While in view v, R has previously broadcasted a message of the form

for a block B' that equivocates B (and since equivocation is a symmetric relation, B equivocates B').

- The contents of B are faulty in some application-specific sense (for example, focusing on the blockchain world, B could contain a smart contract transaction with an invalid signature).
-

does not have a valid signature.

Technical note:

includes the message

. Otherwise, Byzantine replicas could broadcast a vote for an arbitrary equivocating block, causing a DoS attack on the system due to Bullet 1.

Additionally, R ignores proposals from other views.

What if a valid message of the form

never arrives?

Roughly speaking, the replica broadcasts a "complaint" against the leader, which we call a blame message

.

Execute step (non-blocking)

R finalizes a block B (and all predecessors of B) into its list of blocks C(R) as soon as the following two conditions are met:

Condition 1

: R has not decided to change the leader L_v (more on this later).

Condition 2

: One of the following is true:

- Responsive finalization

: R has received at least 3n/4 messages of the form

from different replicas. If R voted for B previously, we also count such a vote towards the 3n/4 votes.

- Synchronous finalization

: 2Δ time has passed since R broadcasted its message

.

When to initiate a view-change?

R decides to change the leader, i.e. R begins a view-change, when:

- R receives evidence of the leader being faulty: this may be at least $\lfloor n/2 \rfloor + 1$ blame messages, or cryptographic proof of a severe issue with the leader's behavior (sent by another replica).
- R receives an erroneous (in the sense described in the vote step) vote or block proposal for a block B. In this case R broadcasts cryptographic evidence of such an event.

This is the entirety of the steady-state of our simplified version of the OptSync protocol. Describing the view-change protocol is out of scope.

Security analysis

Next, we sketch the main points that make the steady-state secure. Don't let the math deter you — it's easier than it looks.

Recall that in our context security

refers to the fact that for any two replicas R, R', there are no two equivocating blocks in the lists C(R) and C(R'). Assuming the leader L_v is not Byzantine, we now discuss why this is indeed the case.

We see that no two honest replicas R, R' can finalize blocks B, B', respectively, that equivocate each other. This means that, as long

as all honest replicas stay in view v, C(R) and C(R) do not contain two equivocating blocks. Hence, one is an extension of the other, as required. To prove so, we argue that if this is not the case, then a contradiction occurs. Hence let us assume R finalizes B while in view v and R' finalizes a block B' that equivocates B when in view v. There are four scenarios:

- Both B and B' were finalized responsively.
- Both B and B' were finalized synchronously.
- B was finalized responsively and B' synchronously, or vice-versa.

Case 1: Both B and B' were finalized responsively

The following type of argument is ubiquitous in the literature about consensus protocols.

Denote by S, S' the sets of replicas that have voted for the blocks B and B' when in view v, respectively.

We claim

that there must exist an honest replica that belongs to both S and S'. This contradicts the fact that, by design, while in view v, an honest replica R_h never votes for two blocks that equivocate one another (see the "Vote" step above).

Proof of the claim

: We present the general argument. Along the way we give the actual numbers for the example n=8 and f < n/2 = 4.

Since B and B' have been finalized responsively, we have |S|, |S'|≥ 3n/4= 6.

Hence the sets S and S' must have at least n/2=4 replicas in common. In the example, if there are less than n/2=4 replicas in common, say there are 3 replicas in common, then there must be at least 3 replicas belonging to B and not to B', and 3 other replicas belonging to B' but not to B, forcing |S|≥ 9, a contradiction.

Since f < n/2 =4, and S and S' have at least n/2=4 replicas in common, there must be at least one honest replica R among these. This means that R voted for the two equivocating blocks B, B': a contradiction.■

(Figure: Illustration of the example discussed above. The blue dots (1, 2, 3, 5, 6) represent honest replicas, while the Byzantine replicas are represented as hollow green dots (4, 7, 8). S' and S are indicated as orange and dashed gray lines, respectively. The reader may convince themselves that it is impossible to draw two such sets with at least 6 replicas each in such a way that no honest replica lies in both sets).

Case 2: Both B and B' were finalized synchronously

Suppose the honest replicas R and R' make a synchronous finalization for the equivocating blocks B and B'. Suppose the finalizations occur at times t and t', respectively, and t>t', i.e., B' is the first among the two blocks to be finalized.

By design, R voted for B at time t-2Δ, and it did not detect errors

later since, otherwise, it would have started the view-change protocol and exited view v (see Condition 1 in the description of the Execute step). This implies that no other replica voted for an equivocating block proposal before time t-Δ.

Indeed, otherwise, R would have seen the vote before time t, and it would not have finalized B; see the figure below:

However, since t'<t, we know that R' voted for B' at time t'-2Δ < t — 2Δ. This is a contradiction. Hence, Case 2 cannot occur.

Case 3: B was finalized responsively, and B' was finalized synchronously

Suppose R finalized B responsively at time t. By a similar argument as before, we know that no replica voted for an equivocating block before time t-Δ, since otherwise, R would have seen such a vote before finalizing B. Hence R' broadcasted its vote at a time at least t-Δ and made its synchronous finalization at a time at least t-Δ + 2Δ = t+Δ.

Since R finalized B responsively, some honest replica must have voted for B before time t (for example, R itself). All replicas will receive such a vote before time t+Δ.

Hence it is impossible for the honest replica R' to finalize B' synchronously. This is because we know that such finalization occurs at a time at least t+Δ, but we also know that R' receives a vote for an equivocating block before such time; see the following figure:

Final remarks

This concludes our overview of consensus protocols. We started with the basics and made our way into understanding some low-level details of OptSync, a state-of-the-art synchronous BFT protocol. We have also described two of the three main primitives that enable Decentralized Validator Technology (DVT). Stay tuned for our next post of the series discussing the last one: Threshold Signature Schemes!

References

- Decentralized thoughts blog: https://decentralizedthoughts.github.io/
- Byzantine broadcast and Byzantine agreement: https://decentralizedthoughts.github.io/2019-06-27-defining-consensus/
- Communication models: https://decentralizedthoughts.github.io/2019-06-01-2019-5-31-models/
- Partial synchrony communication: https://decentralizedthoughts.github.io/2019-09-14-flavours-of-partial-synchrony/
- Authenticated Synchronous BFT: https://decentralizedthoughts.github.io/2019-11-11-authenticated-synchronous-bft/
- Good-case latency of Byzantine Broadcast: a complete characterization (2022)
- Byzantine Agreement, Byzantine Broadcast, and State Machine Replication with near optimal latency (2020)
- Byzantine threshold cheat sheet: https://decentralizedthoughts.github.io/2021-10-29-consensus-cheat-sheet/
- Asynchronous Agreement: https://decentralizedthoughts.github.io/2022-03-30-asynchronous-agreement-part-one-defining-the-problem/
- What is the difference between PBFT, Tendermint, SBFT and HotStuff?:https://decentralizedthoughts.github.io/2019-06-23-what-is-the-difference-between/
- What is a DAG?: https://en.wikipedia.org/wiki/Directed_acyclic_graph
- Presentation on the impossibility result: https://www.youtube.com/watch?v=vJhm9uhd34E&list=PLEGCF-WLh2RLOHv_xUGLqRts_9JxrckiA&index=17
- Original proof of the impossibility result:https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf
- Partial Byzantine Fault Tolerance: https://pmg.csail.mit.edu/papers/osdi99.pdf
- Gasper: https://arxiv.org/abs/2003.03052
- Tendermint's homepage: https://tendermint.com/
- Tendermint: https://arxiv.org/pdf/1807.04938.pdf
- IBFT consensus: https://github.com/ethereum/EIPs/issues/650
- QBFT whitepaper original idea: https://arxiv.org/pdf/2002.03613.pdf
- SyncHotStuff — Decentralized Thoughts post: https://decentralizedthoughts.github.io/2019-11-12-Sync-HotStuff/
- SyncHotStuff: https://eprint.iacr.org/2019/270
- Hotstuff: https://arxiv.org/abs/1803.05069
- Nakamoto consensus: https://bitcoin.org/bitcoin.pdf
- Analysis of Nakamoto Consensus: https://eprint.iacr.org/2019/943
- OptSync — Presentation: https://www.youtube.com/watch?v=4nZHp1-kM4U&t=883s
- OptSync: https://eprint.iacr.org/2020/458.pdf
- Thunderella: https://eprint.iacr.org/2017/913.pdf
- On the Optimality of Optimistic Responsiveness:https://decentralizedthoughts.github.io/2020-06-12-optimal-optimistic-responsiveness/
- Authenticated Synchronous BFT: https://decentralizedthoughts.github.io/2019-11-11-authenticated-synchronous-bft/
- PiLi: https://eprint.iacr.org/2018/980.pdf
- Hybrid-BFT: https://eprint.iacr.org/2020/406
- Phase-King through the lens of Gradecast: A simple unauthenticated synchronous Byzantine Agreement protocol