Consensus Protocols 101

Preliminaries

Introduction

This document is intended for those who are interested in learning more about consensus protocols. The current landscape of blockchain resources has a plethora of competing resources with no clear metric for gauging their quality. This document collates what we believe are the most important facts to understanding *blockchain consensus* in an accessible way.

By the end of this document, we'll arrive at a detailed description of the PaLa consensus algorithm. In particular, we believe PaLa to be the simplest and most performant consensus algorithm of its class and thus treat it as a first class citizen in understanding distributed consensus.

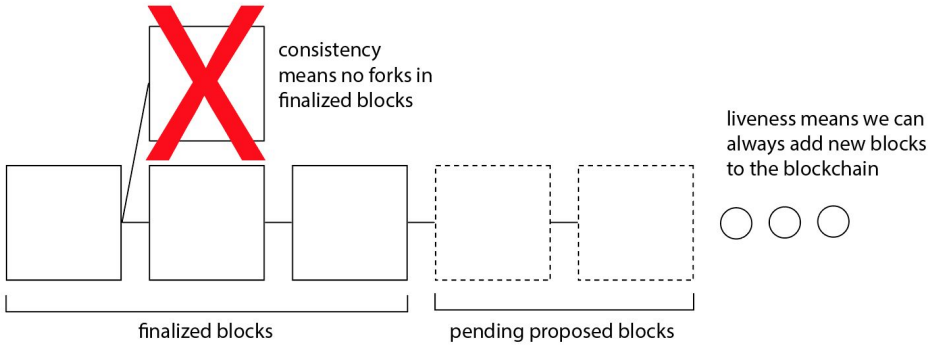
This document is written by Gengmo Qi and Peter Lu and proudly supported by ThunderCore.

What is a consensus protocol?[1]

Consensus is an abstraction for distributed systems where a set of nodes seek to agree on an ever-growing linearly-ordered log, such that two important properties are satisfied:

- ***Consistency***: all honest nodes' logs agree with each other
- ***Liveness***: all honest nodes are able to make progress on their logs

In context of blockchain, consistency (sometimes called "safety") means that there is a single canonical chain (no forks) that all honest nodes will agree on. Liveness means that honest nodes on the network are always able to add new blocks to the blockchain. When all nodes agree on a block, we say the block is *finalized*.



Major types of consensus protocols

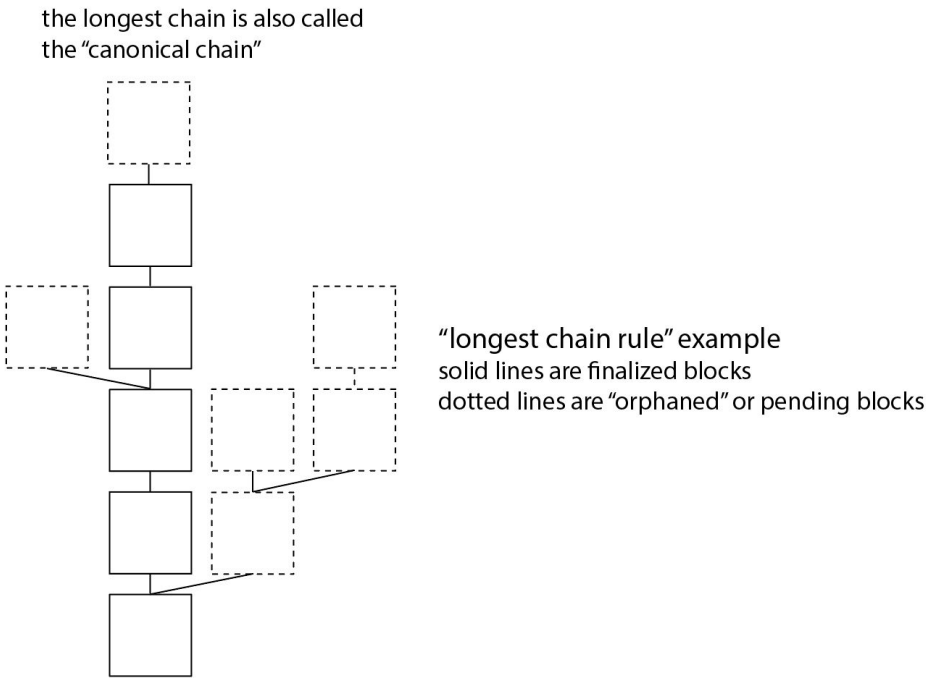
We are interested in the following two broad classes of consensus protocols:

(1) Nakamoto Consensus

Nakamoto consensus, sometimes also called chain consensus, use Nakamoto's elegant longest-chain fork choice rule for reaching consensus with high probability and is a breakthrough in distributed consensus. These protocols are conceptually simple and tolerate *minority corruptions* (50%). Further, not only has blockchains' robustness been empirically proven in [real world public blockchain networks holding billions in assets](), earlier works have also shown mathematically that Nakamoto consensus indeed achieves certain robustness properties in the presence of sporadic participation and node churn that none of the classical style protocols can attain.[2][3] Unfortunately, known Nakamoto consensus protocols suffer from

[1] The definition for consensus we give here is sufficient for all except those attempting to write formal proofs. See https://eprint.iacr.org/2017/913.pdf#page=19 for a rigorous definition.
[2] https://eprint.iacr.org/2016/919.pdf
[3] https://eprint.iacr.org/2016/918.pdf

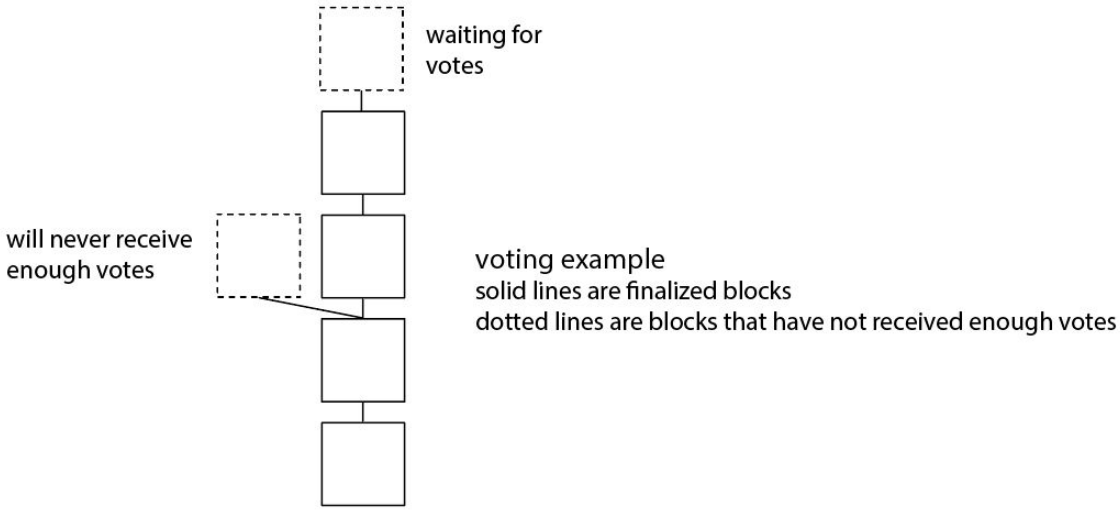
slow transaction confirmation times and low throughput. For example, Bitcoin has a 10-minute block interval and requires several blocks to confirm a transaction with sufficient confidence. Earlier works that mathematically analyze Nakamoto consensus have pointed out that *such slowness is inherent for Nakamoto protocols since the expected block interval must be set to be sufficiently large for the protocol to retain security*.[4]



(2) Classical Consensus

Classical consensus protocols reach deterministic consensus through voting. These protocols confirm transactions fast relative to Nakamoto consensus as the consensus network size is fixed and progress can be made as soon as the required votes are seen. These protocols typically use the *partially synchronous* (or partially asynchronous) network model, and thus can only tolerate at most ⅓ (33%) faults.

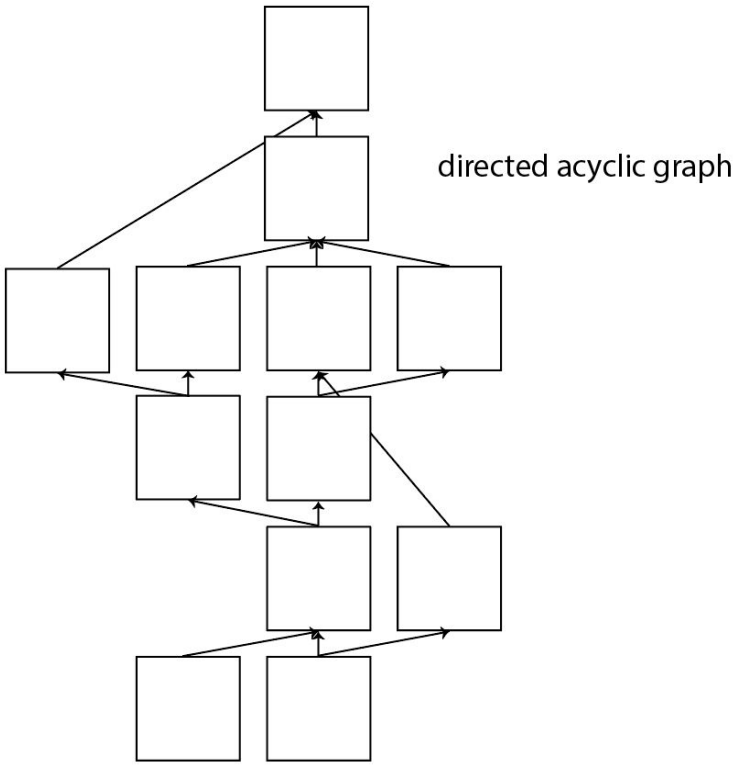
[4] https://eprint.iacr.org/2016/454.pdf



“DAG”

You may also hear of still a third class of blockchain protocols sometimes called “DAG” protocols (a name that says little about the mechanism for consensus). Such protocols include SPECTRE, The Tangle, Avalanche and PARSEC, and Hashgraph. These protocols achieve consensus on non-linear or eventually linear directed acyclic graph (DAG) of blocks using a variety of means. These protocols—which often claim to be the logical evolution of blockchain[5]—are interesting both in a theoretical and practical setting. We omit discussing them in this article to avoid complexity. This article still provides the fundamentals needed to understand such protocols so if they are of interest to you, please keep reading!

[5] In blockchain, technological imperative and social progress are conflated with so much arrogance and grace. The history for this nascent field is still being written with affirmative shots being fired in the air by all competitors as a necessary precondition for victory. The reality must be excavated from the details strewn about in the battlefield.



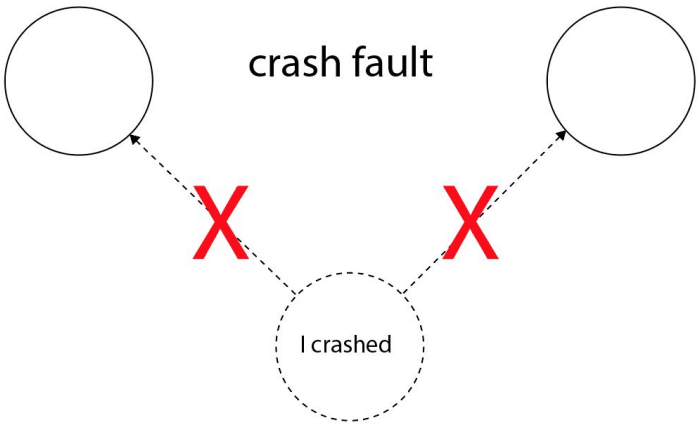
In the next two sections, we'll get into some background information and build up to a rigorous understanding of how we arrive at the tight ⅓ and ½ fault tolerance bounds mentioned above.

Byzantine Faults and Crash Faults

For the purpose of this article, we distinguish between two types of faults

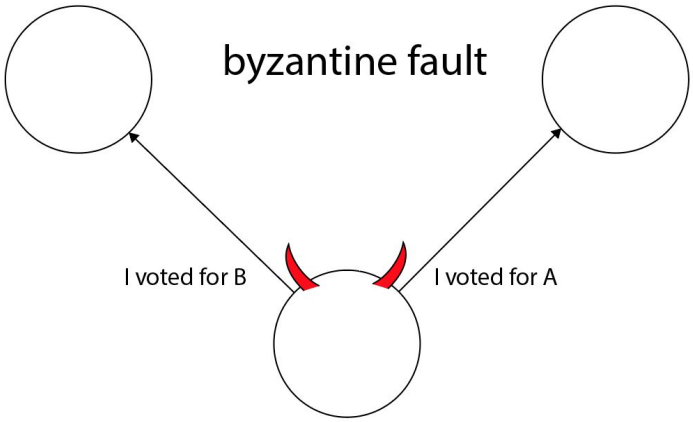
Crash Faults

A crash fault is just that. Even the best run servers do not have 100% uptime and thus crash faults must be addressed in any robust distributed system. A crashed node will stop responding to messages and may lose data. Thus a crash fault tolerant consensus protocol must handle some nodes dropping offline arbitrarily and must allow them to recover data from non-faulty nodes. Examples of crash fault tolerant consensus protocols include Raft and Zookeeper. Raft is simple and commonly taught in introductory courses to consensus. This interactive tutorial is a great place to learn more about Raft and consensus in general.



Byzantine Faults

A byzantine node may behave arbitrarily in the network including not sending messages and sending deliberately misleading messages selectively to other nodes on the network. A byzantine fault may present different symptoms to different observers. It is difficult to declare it failed and shut it out of the network, because the network must first reach a consensus regarding which component has failed in the first place. The term is derived from the Byzantine Generals' Problem, where actors must agree on a concerted strategy to avoid catastrophic system failure. Some of these actors may be traitors deliberately attempting to sabotage a coherent strategy.



A crash fault tolerant algorithm can tolerate crash faults up to a certain threshold. A byzantine fault tolerant algorithm is able to tolerate byzantine faults up to a certain threshold. In the case of blockchain consensus, we can imagine byzantine faults as malicious actors on the network trying to pull off a double spend or bring the chain to a halt.

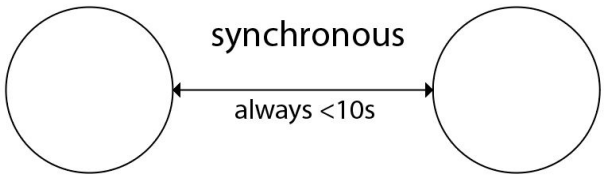
You might notice classical consensus protocols are sometimes also called “BFT consensus” protocols. This usage is prevalent and misleading. Nakamoto consensus protocols are also BFT but it is not a BFT consensus protocol in this sense. Thus we’ll encourage the use of the term “classical consensus” instead.

Synchronous, Partially Synchronous, Asynchronous

These are assumptions a consensus protocol requires from the underlying network.

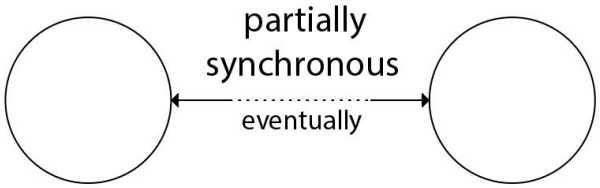
Synchronous

- Synchrony assumes that there is a **known** upper bound on all message delay. That is, all messages must be delivered in some amount of time, and all participants in the network know how long it takes for messages to be delivered.
- Can tolerate up to ½ byzantine faults
- It turns out the synchronous network model can be overly restrictive with strict limitations. The more nuanced weakly synchronous model is a more practical option.[6]



Partially Synchronous

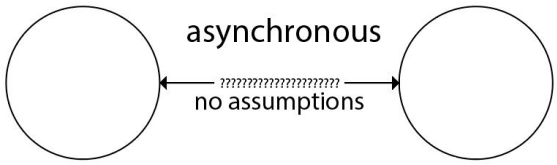
- Sometimes also known as partially asynchronous.
- Partial synchrony assumes that there is an *unknown* bound on network latency, we do not know ahead of time what it is. The system behaves like a synchronous one most of the time, but sometimes network delay that exceed the bounds may happen.
- The partially synchronous model is relatively realistic. Unknown delays do occur in real life systems and we may assume that network infrastructure is reliable enough to always deliver messages eventually.
- Can tolerate up to ⅓ byzantine faults



[6] https://eprint.iacr.org/2019/179.pdf

Asynchronous

- Asynchrony assumes that there are *no bounds* on network latency even between correctly functioning nodes. There is no common global clock thus algorithms cannot make timing assumptions and can't use timeouts.
- The FLP impossibility proves it is impossible to create an algorithm which is *guaranteed* to reach consensus in any specific finite amount of time if even a single faulty node is present.
    - That is to say, it's impossible to have both liveness and consistency.
- However, there do exist (randomized) algorithms that can achieve consensus within T seconds with *probability exponentially approaching 1* as T grows.
    - In other words, under the asynchronous setting, if you have bad actors around, no deterministic algorithms can even agree on the value of a single bit.
- Fault tolerance:
    - Deterministic algorithms: 0%, cannot tolerate faults
    - Probabilistic algorithms: up to ⅓ fault tolerance



Notes

- In practice, the right choice of consensus algorithm, and thus the right choice of the underlying consensus protocol is very context specific. For example, synchronous network assumptions may be appropriate if ½ fault tolerance is needed and if the assumed maximum network delay is very high or the underlying network is very reliable (e.g. a private network).
- You might hear that asynchronous and partially synchronous protocols are more performant than synchronous. Fitting the assumptions of the network, synchronous protocols typically wait for a period of time for all messages to arrive before reaching consensus whereas asynchronous protocols make a decision as soon as some message threshold is reached. This is not a rule. There are, for example, synchronous protocols that have "asynchronous performance".

Fault tolerance bounds: Why ½ and ⅓?

You must have seen or heard expressions like *majority honest*, *tolerate minority corruptions; tolerate ⅓ corruptions* multiple times. In the following, we aim to explain the bounds on fault

tolerance under different network assumptions. For the curious reader, please see this paper for a more thorough analysis.

Synchronous

- Intuition: Majority rule, we want the correct choice to have more than half the votes.
    - By synchronous network assumptions, we can assume that every node on the network has seen every vote after a fixed amount of time has passed at which point a decision can be made.
- Explanation[7]:
    - Assume a total of `n` nodes, among which `f` is faulty(behave arbitrarily). Once all messages are received (applying synchrony assumption), we want to have at least `f+1` correct messages to outnumber the `f` faulty messages.
        - $n \geq f+(f+1)$
        - $f \leq (n-1)/2$
        - $f < n/2$
            - "Up to ½ byzantine-fault tolerance"

Partially Synchronous

- Intuition: A single dishonest node can fool 2 other nodes by sending conflicting messages to each node.
    - The conflicting messages may not be detected in time because of the partially synchronous network assumption means there is an *unknown bound* on the network latency.
- Explanation[8]:
    - Assume a total of `n` nodes, among which `f` are faulty (may behave arbitrarily).
    - Assume a node received `x` messages.
    - The `f` faulty nodes may choose *not to send any message*, thus in the worst case a node will receive *at most* `n-f` messages, and make a decision based upon them.
        - $(1)\ x \leq n-f$
            - "There is at most `n-f` messages"
    - Among the `n-f` messages, it is unclear which ones are from the honest ones, or faulty ones. It is possible that the nodes that did not respond are not faulty, but we didn't get the message only because of network delay.
    - Therefore, `f` of those responses might be faulty, which means *all faulty nodes may have sent false messages.*
    - To make a decision, we have to guarantee there is at least `f+1` correct messages to outnumber the `f` faulty messages.
        - $(2)\ x \geq f+(f+1)$

[7] See Miller et al. for a rigorous proof.
[8] See Lamport et al. for a rigorous proof

        - "There must be enough responses from non-faulty nodes to outnumber those from faulty ones"
    - By equations `(1)` and `(2)`:
        - `f+(f+1) ≤ x ≤ n-f`
        - `f ≤ (n-1)/3`
        - `f < n/3`
            - "Up to ⅓ byzantine-fault tolerance"

Note on fault tolerance bounds

- So far we have informally established that it's possible to achieve consensus with ½ fault tolerance in a synchronous network—messages broadcasted by any honest node are guaranteed to be received by all other honest nodes within some known time period.
- The maximum achievable fault tolerance drops to ⅓ if we relax to partially synchronous network assumptions—instead of having a known upper bound, the bound of network delay is unknown.
- It's unclear if this will be used in practice, but If we add *even more assumptions* we can increase fault tolerance all the way to 99%.

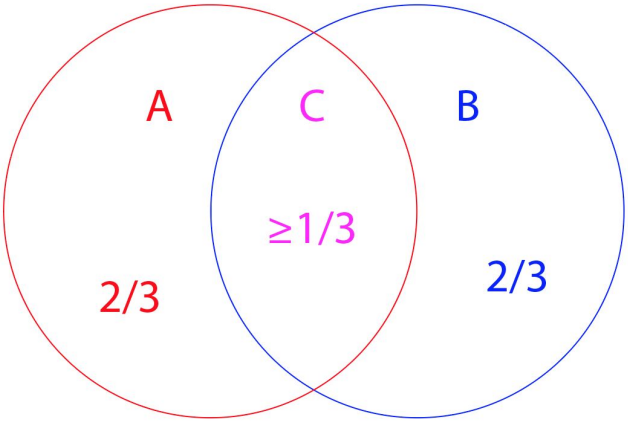
Takeaway

It doesn't make sense to talk about a protocol's fault tolerance without knowing the assumptions it is based upon 🤔. When you are comparing across different protocols in the future, do not be misled by claims such as "*Our protocol X is 50% fault tolerant, thus more secure than protocol Y which is 33% fault tolerant* 😈".

Liveness vs Consistency

You may sometimes hear that PoW favors liveness over consistency. Any PoW node has a chance to mine a new block and contribute to the blockchain's liveness whereas there is no consistency until sufficient time has passed. In Nakamoto consensus, liveness comes for free as any node can make a new block whereas consistency must be carefully reasoned.

In classical consensus, the opposite is true. Consistency is a straightforward argument based on the pigeonhole principle. With a ⅔ voting threshold, it's clear that at least ⅓ nodes must be byzantine to create a split vote. In the picture below, groups A and B are both ⅔ votes for conflicting proposals. Honest voters do not cast conflicting votes. Thus, group C is byzantine and votes for both groups. Group C must contain at least ⅓ of the nodes in order to create two ⅔ majority votes for conflicting proposals.



With only honest nodes, there might be a split vote (say ½ vote in group A and ½ vote in group B). In this case neither groups A or B will ever reach ⅔ majority since no honest node casts conflicting votes. Without further mechanisms to make progress, liveness comes to a halt. Thus, in classical consensus, consistency is easy and liveness is hard.

Proof-of-Work

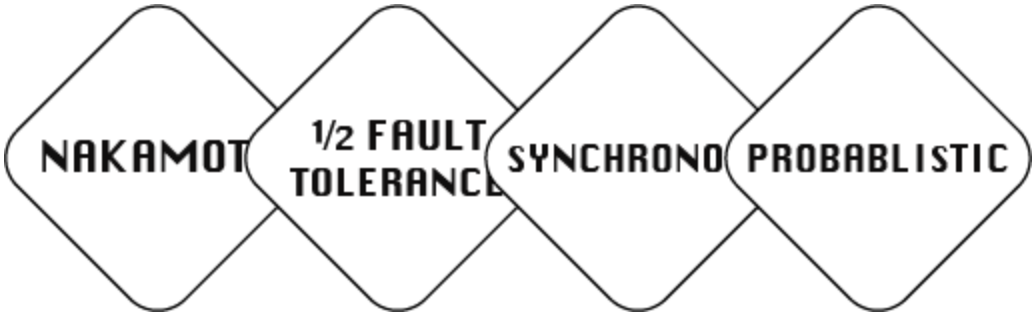
So we now know enough to understand the formal properties of the Proof-of-Work (PoW) consensus protocol. PoW is the very original Nakamoto consensus protocol. Using the longest chain fork choice rule, blocks are finalized when they are deep enough into the history. This works because the more blocks that have been mined on top of any given block, the more "work" is required to fork it. The protocol has *probabilistic* finality meaning consistency is achieved with *very high probability*.[9]

While the bitcoin paper did not make explicit network assumptions, we can interpret the protocol as functioning in the synchronous network setting. Blocks are required to propagate through the entire network such that each node is likely mining on top of the most recent block. Another way to look at this is to say that PoW is not *partition tolerant*. A network is partitioned if there are subsets who can speak within their group but not to each other. Since hash power is the only requirement for creating blocks, we can see that each partition in a PoW network is capable of extending and finalizing their own chain.

Finally, as we read earlier, the synchronous network have up to ½ fault tolerance. Indeed PoW achieves this tight bound. This is clear if you understand the "51% attack" where an adversary with majority of the network hash power creates a fork from before a finalized block that becomes the new longest chain thus breaking consistency.

---

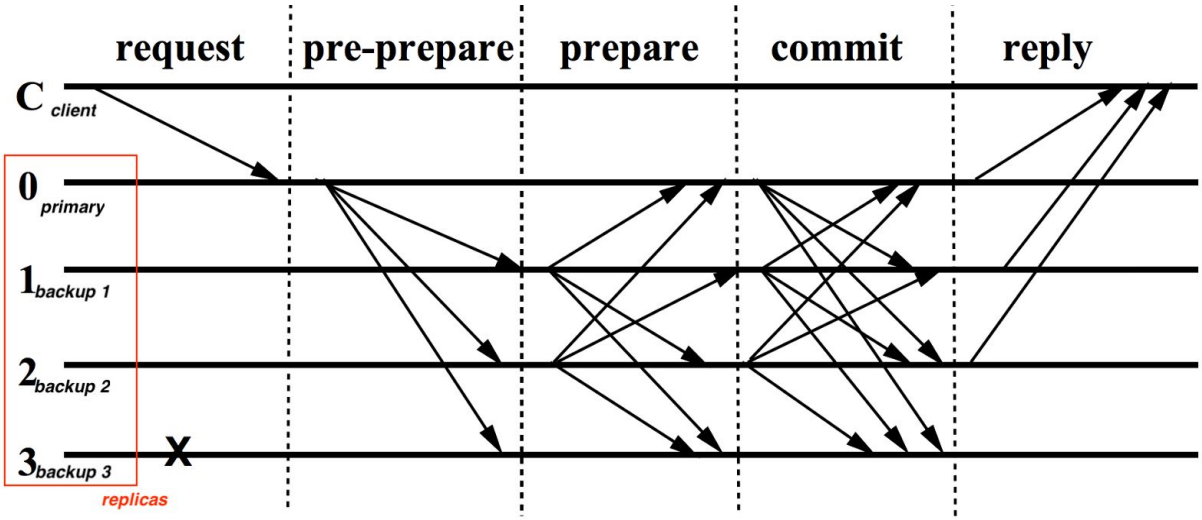
[9] This might seem bad at first, but remember that blockchain is built on cryptography primitives which themselves are only secure to attack with very high probability? How high you might ask? Well let's just say you'd have better luck finding a specific atom in the universe.


NAKAMOT
1/2 FAULT
TOLERANCE
SYNCHRONO
PROBABLISTIC

PBFT (Practical Byzantine Fault Tolerance)

In this section, we'll give a high level overview of the PBFT consensus algorithm. While we consider newer protocols including PaLa to be a strict upgrade over PBFT. Understanding PBFT is still crucial. Many of the terminology and ideas behind newer classical consensus protocols come from this protocol.

We use PBFT as a learning tool from which we will establish the language and ideas that PaLa and Thunderella protocols will build on. Note that the protocol presented here is modified and simplified to be applicable for blockchain consensus.[10] The core concepts remain the same.



Source: Practical Byzantine Fault Tolerance,OSDI, 1999.

Terminology

The algorithm is modeled as a state machine in a distributed system that is replicated across different *replicas*, among which one of the replicas is the *primary* and the rest are *backups*.

Normal-case operation

On a very high level, PBFT consists of the following steps in the normal-case operation:

[10] In particular, this modified protocol omits checkpoints (which are an implementation detail) and null proposals (which allow multiple outstanding proposals during a view change, some of which may be be skipped).

- Request: the user sends *transactions* to the primary.
- Pre-prepare: the primary produces a *proposal* containing transactions and forwards to all replicas.
- Prepare: Upon receiving a proposal, backups will verify it, and if it succeeds, they will broadcast *prepare* message to *all other replicas*. Backups do nothing if verification fails. This is the *first round* of voting.
- Commit: Upon receiving *prepare* messages from ⅔ of all backups, replicas will now broadcast *commit* messages. This is the *second round* of voting.
- Reply: the client sees the result of consensus.

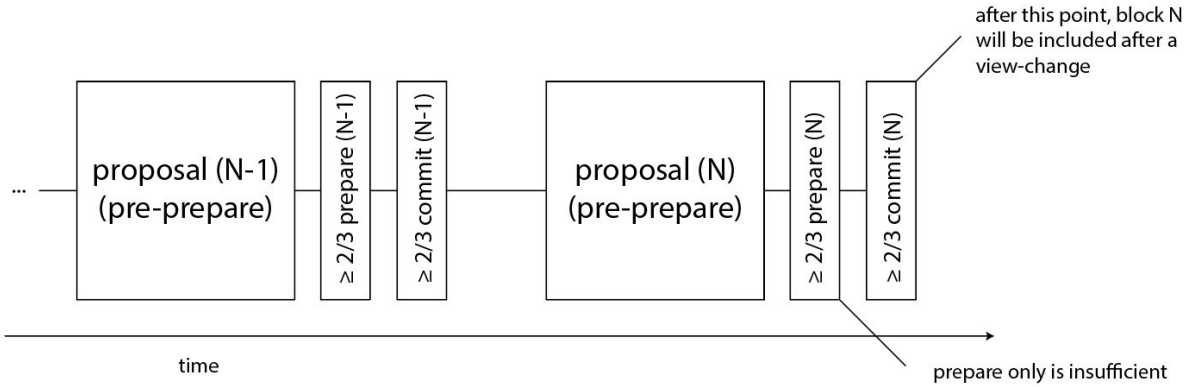
In most circumstances, normal-case operation requirements are met and the primary is able to rapidly process new transactions. Since only ⅔ of backups need to agree in the prepare and commit phase, PBFT can already tolerate up to ⅓ of backups going or even acting byzantine. But what happens if the primary goes down?

View-Change

If the primary has failed, a subroutine called *view-change* will be carried out. On a high level, *the view-change protocol provides liveness by allowing the system to designate a new primary when the former fails*. When replicas see no new progress after some time, they broadcast a view-change message indicating that they want to advance to the next *view*. The view-change begins when *view-change messages* from ⅔ of all replicas are collected.

During a view change, replicas may not agree on the latest state of consensus—remember that the primary was responsible for leading consensus up until now. Since we are in the partially synchronous network setting, we can not guarantee that all replicas will see a ⅔ majority vote in the first round if it exists. A replica who sees a ⅔ majority vote for a proposal in the first round can not assume that others have seen it as well and therefore there is no agreement.

Thankfully, PBFT required 2 rounds of voting where replicas do not vote in the second round until they've seen results from the first round. Thus, upon seeing ⅔ majority vote in the *second round*, the replica is guaranteed that at least ⅔ of replicas have seen results from the *first round*. This knowledge about knowledge replaces the synchrony assumption where nodes are simply assumed to have seen a message (a round of votes in this case) after a certain amount of time has passed.



With this mechanism in place, the new primary *will* continue from the last proposal that received ⅔ majority vote in the second round.[11] *This is because view-change messages also include the last proposal a backup has seen ⅔ majority vote in the first round*.[12] The proposals in the view-change messages dictate where the proposer must continue. Assuming fewer than ⅓ nodes are byzantine, one can show that if *anyone* has seen a ⅔ majority vote in the second round, there must be at least 1 honest backup who successfully broadcasted a view-change message indicating that they have seen ⅔ majority vote in the first round for that proposal. The proof is an application of the pigeonhole principle and described in detail in 4.5.1 of the PBFT paper.[13]

Takeaway

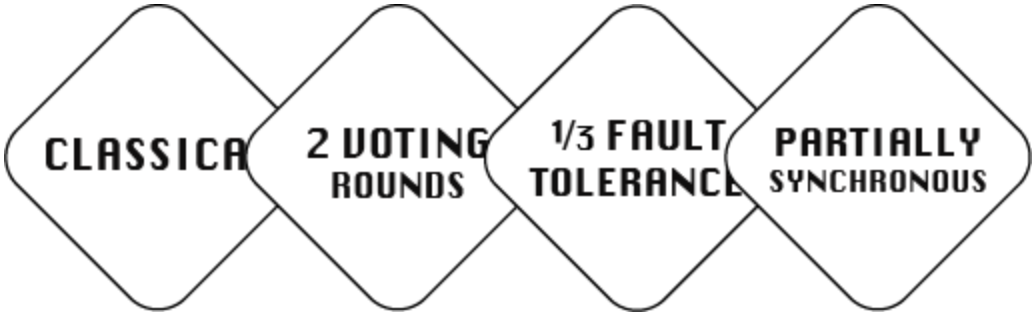
PBFT is a seminal classical consensus protocol employing 2 voting rounds for finality. It is in the partially synchronous network and achieves the optimal ⅓ fault tolerance. Understanding at a high level how 2 rounds of voting enable safe view-changes is the basis of all classical consensus protocols. The details of PBFT are not so important and we'll see soon how PaLa is a strict improvement and simplification over the PBFT protocol.

---

[11] In the original unmodified protocol, proposals since the last checkpoint need to be voted on again after a view-change. I did not understand why this was necessary.

[12] This is a crucial detail which I admit is not well spoken here. We'll see something similar in the Thunderella protocol.

[13] Not surprisingly, the proof looks very similar to the argument we presented earlier of the ⅓ fault tolerance bound in the partially synchronous case.



PBFT is a message heavy algorithm. There are 3 rounds of messages and each message must be sent to all other replicase. It thus has $O(n^2)$ message overhead where n is the number of replicas.

Alternatives

PBFT employs the *view-based* approach to consensus where each view is run by a primary leader who is in charge of creating new proposals. The *ballot-based* approach is based on the Paxos consensus protocol and used in Federated Byzantine Agreement (FBA) protocols such as the Stellar Consensus Protocol (SCP). In the ballot based approach, there is no explicit leader. Instead anyone may share a proposal for a *slot* and, after consensus, the final value for that slot is created from all agreed upon proposals (e.g. the union of all transactions of all proposed blocks). This approach avoids censorship issues of having a single node responsible for creating new proposals in the short term. The ballot based approach still employs two rounds of voting to finalize a decision. As ThunderCore prioritize the performance of view-based approaches, we do not go into details of the ballot based approach. For more information on FBA and ballot based approach, please see the SCP paper and the timeless Paxos historical document.

Proof-of-Stake

Proof-of-Work Recap

Proof-of-Work, consensus is based only on computation power. Participation is fluid. Thus, from the protocol standpoint, it is entirely *permissionless*. Anyone with the capability of solving the hash puzzle has a chance to contribute to consensus. This capability is in no way given to anyone by the protocol itself.[14] As more hash power joins the network problem difficulty goes up. In economic equilibrium, problem difficulty scales with token value. Energy consumption goes up and chance of mining a block per unit of hash power goes down.

Proof-of-Stake (PoS) protocols switch the participation requirement to be based on *stake in the underlying currency* instead of hash power. This section introduces the major themes of PoS and some of its variants.

Original Proof-of-Stake

The original usage of “Proof-of-Stake” refers to a class of Nakamoto consensus algorithms where a node’s holdings or stake in the underlying currency is converted to virtual computation power. The name is derived from its similarity to Proof-of-Work somewhat literally replacing the “Work” with “Stake” in the algorithm. For example, the PoW mining problem might be changed from:

```
hash(parent, nonce) <= difficulty
```

to something like:

```
hash(parent, address A) <= balance(address A) * time * difficulty
```

with the remainder of the protocol unchanged.

Since original PoS still uses the longest fork choice rule, the immediate issue is that it is easy for nodes to “mine” blocks on many forks at once. This is byzantine behavior as honest nodes only mine on a single longest block. However, since there is no *cost* for acting byzantine, we can not assume nodes will not do so. It is strict economic gain as more mined blocks on different forks

[14] The financial barriers of obtaining this capability (hash power) is a different story.

means a higher chance of being included in the canonical chain and earning block rewards. This is referred to as the the *nothing at stake* problem and can be addressed with *slashing* which we'll discuss soon. There are still other difficult to address vulnerabilities.

Committee based Proof-of-Stake

The term Proof-of-Stake has also been applied to classical consensus protocols that use some staking based scheme for choosing its participants. We call the chosen set of participants a *committee* and individually we refer to them as *voters*. In these protocols, participants put down a deposit in the underlying currency for the right to become a voter in the network. The committee is responsible for creating new blocks on the blockchain. The nothing at stake problem can be addressed by slashing a voter's deposit when there is cryptographic evidence of the voter's bad behavior.

Though we consider this bad use of Proof-of-Stake, it seems far past the point where this usage can be corrected. Thus Proof-of-Stake refers to a broad class of blockchain protocols where participation is based on stake which include both Nakamoto and classical consensus protocols.

Other approaches include *permissioned* networks where the set of consensus nodes are known beforehand. A similar (arguably indistinguishable) approach is to allow each individual node on the network to choose the nodes they trust to achieve consensus. Public networks like Stellar and Ripple deploy the latter. Permissioned networks are the obvious choice for private consortium chains.

Long-Range Attacks

With staking, deposited stake must be released eventually at which point it can no longer be slashed. After this point, voters (who may have moved or sold their stake) can release their old private keys with no consequence. If an attacker obtains enough of these keys (by purchasing them for example) they can readily create a new blockchain that forks the current one from a long time ago. This is referred to as a *long range attack*. Thankfully, there is a straightforward defense. Nodes on the network will simply reject any blockchain that doesn't extend recently from their own blockchain thus thwarting this attack entirely. For new nodes joining the network, the only solution is to pick a set of trusted nodes to provide them with the canonical chain. In practice, this solution seems satisfactory at least when considering the fact that the $$$ value (of a cryptocurrency) is a social construct to begin with.

Delegating Stake

PoW mining pools combine the hash power of many nodes distributing risk and rewards across all participants. Analogous to this, PoS participants may delegate their stake to another consensus node on the network who will take a cut of the rewards. Delegation is simpler than

pool mining as it does not involve the delegator to run any of their hardware. Delegation might be built directly into the protocol or trustlessly mediated by a smart contract. If it's not supported in the protocol, delegation can also be done off-chain with a trusted service provider. Most PoS chains support non-custodial (trustless) delegation and there are many staking as a service providers out there to make delegating your stake even easier.

DPOS BFT

For clarity, we will briefly talk about the DPOS BFT[15] consensus algorithm which is a simple classical-like consensus algorithm. DPOS BFT takes the top 21 (say) highest staking nodes (which include delegated stake from other users) to be nodes participating in consensus. Consensus is achieved through round robin proposing and voting. Offline proposers are skipped after sufficient time. Two round robin rounds of proposed blocks are needed to finalize a block. DPOS BFT is the name of this consensus algorithm. The name also refers to the mechanism of delegating stake to choose participants which is a feature available in many other PoS blockchains. Similar *consensus* algorithms include Aura and Clique. These algorithms are also known as Proof-of-Authority, which refers to the participants being chosen based on a-priori authority and says nothing about the mechanism for consensus.

Security

As of writing, PoW blockchains still hold a large majority of the cryptocurrency market cap and proponents advocate for their completely permissionless and decentralized nature. From an economic standpoint, PoW networks fall short of PoS in terms of security.

The main issue is that the cost of attack is proportional only to lost mining time. Since a longer adversarial fork becomes the canonical chain, any rewards lost are immediately returned. Thus the cost of attack is proportional to rewards lost during the attack which are returned if the attack succeeds.

With services such as online hash power markets, coordinating such an attack is entirely trivial. In addition, Volatile token prices means mining may not be consistently profitable. If mining not profitable, there will be an excess of unused hash power (bitcoin ASICS have few other uses) that could be sold for for a profitable attack. Hash power owners who have withdrawn their bitcoin would gladly take a premium in exchange for renting their unused hardware. In practice we've already seen several 51% attacks on major a PoW chains.

Proof-of-Stake, subverts these issues by relying on deposits and slashing schemes to disincentivize attacks. The cost of attack in a PoS blockchain can be precisely set by the deposit

[15] This single 7 character acronym blurs the terminology for delegating stake with a consensus algorithm, a Nakamoto style consensus algorithm with a committee selection policy, and a fault tolerance mode with finality guarantees 🥇.

requirement and slashing amount. In practice, these prices should be set so high that such an attack would never be profitable. Thus, from an economic standpoint, *PoS is more secure than PoW* by design. For the curious reader, we suggest this blog post for a more detailed discussion on the design philosophy of PoS.

Summary

So we see there are many ways to look at consensus and choosing participants for consensus. In Nakamoto protocols, the mechanism for choosing participants and consensus are inseparable. It seems that in continuing a dialog that started with Bitcoin, many non-Nakamoto blockchain protocols still combine these two concepts and add to the confusion. The usage of Proof-of-Stake to refer to a means of choosing participants rather than a mechanism for consensus is perhaps for a similar reason. We hope this brief exposition here will clarify some of the ambiguous terminology.

PaLa

The following adapted from the ThunderCore Whitepaper. It is extended with more details and explanations.

PaLa is a blockchain consensus protocol based on partially synchronous network assumptions and tolerates up to ⅓ corruptions. Below, we describe a simplified version of the protocol called **Basic Pala** to illustrate its simplicity and effectiveness. Basic Pala is the foundation for understanding the complete version of our protocol which is outlined afterwards. The full details of the PaLa protocol are readily available in the PaLa research paper.

Setup

Assume a fixed **committee** of **voters**. How these nodes are chosen is described later. Each node maintains a **local epoch** counter *e* and a local view of the blockchain. Each block contains an epoch number, a list of transactions, and the hash of its parent block. The epoch number of a chain is defined as the epoch number of the last block in the chain. Each epoch has a single unique **proposer** which is known to all nodes in the network. In this simplified version, every voter is also a proposer for some epochs.

Consensus proceeds one block at a time. Proposers propose a block if they are eligible to propose in the current epoch. Voters vote on blocks if a set of conditions are met. A collection of ⅔ of the committee's votes on a single block is a **notarization** for that block. If there is a notarization for a block, the block is **notarized**. Each block has an epoch which advances monotonically. If an epoch *e* block has an epoch *e-1* parent, the block is a **normal block** otherwise it is a **timeout block**. A block is **finalized** if it is the parent of a notarized normal block. A finalized block is part of the immutable history of the blockchain and indicates consensus has been achieved.

Protocol

Each node keeps their local current blockchain **fresh**. Whenever it sees a valid blockchain that is **fresher** than its current chain–it has higher epoch number than the epoch number of their current chain–they switch to this chain. A valid blockchain should satisfy the following conditions:

1. The epoch numbers of all blocks should be strictly increasing.
2. Every block in the blockchain is notarized

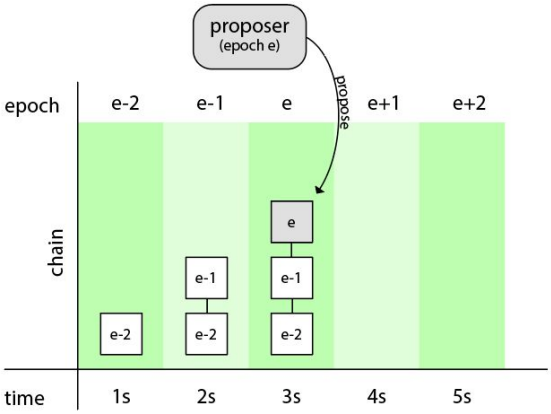
In addition each node will do the following:

- Increase local epoch counter to *e* if

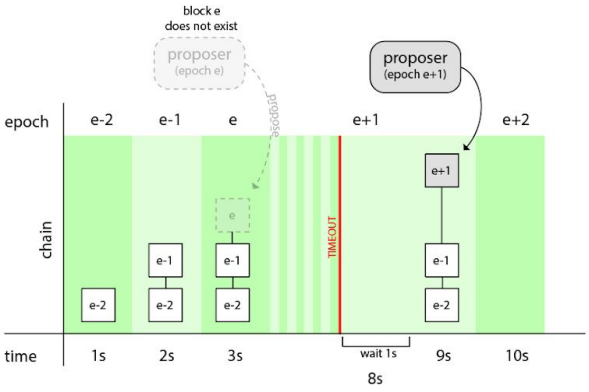
- Their current local epoch is smaller than *e* AND
  - They see a notarized chain for epoch *e*-1 OR
  - They see at least ⅔ committee members' validly signed *clock*(*e*) messages.

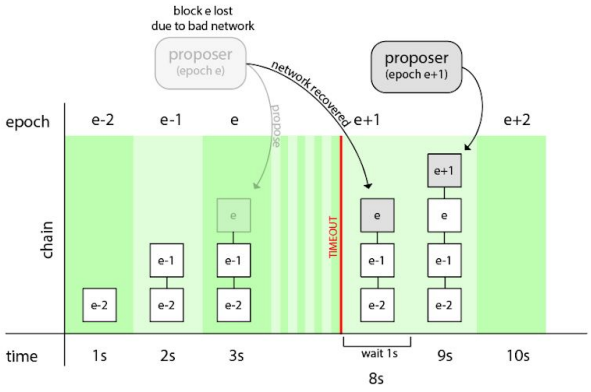
- If they have stayed in epoch *e-1* for more than a fixed amount of time
  - Broadcast the message *clock*(*e*).

- If they are the proposer of epoch *e*
  - If their current chain ends with the block of epoch *e-1*, immediately propose a new block for epoch *e* extending their own current notarized chain

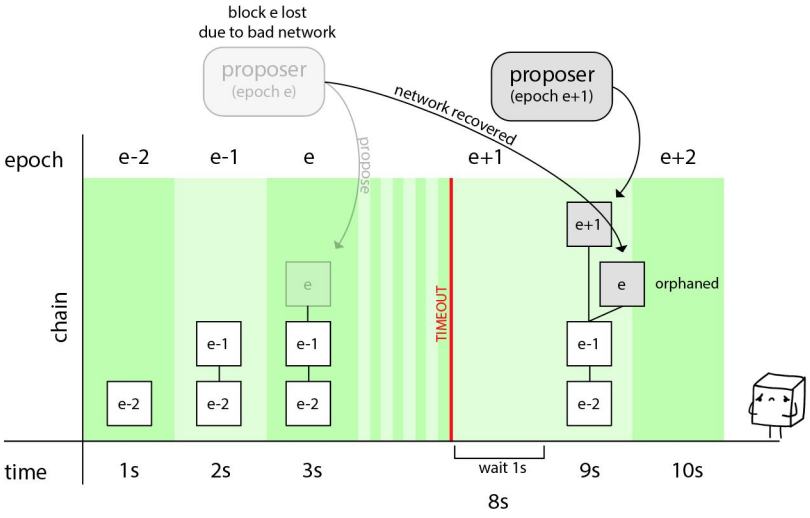


  - If the epoch number of their current blockchain is *e* (due to a timeout), wait for a fixed period of time (hoping to receive a fresher notarized chain) and propose a new block for epoch *e* extending their own current chain.



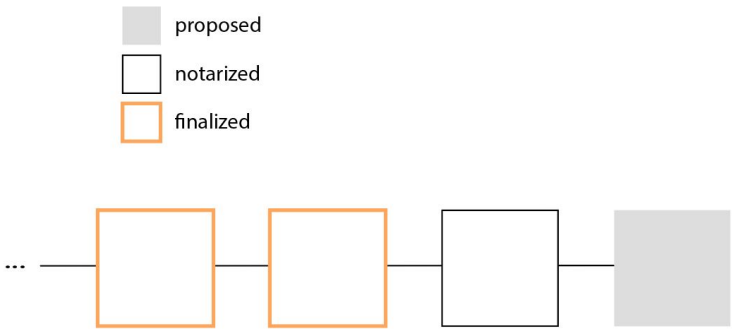


  - Note that it is possible for a notarized block to be orphaned if it arrives after the fixed delay.



- On receiving a block proposal for epoch *e* from the proposer of epoch *e*, sign the proposal and multicast the signature if the following conditions hold:
    - They have seen the notarization for the parent block of the proposal.
    - The block is at least as fresh as their current chain at the beginning of epoch *e*.
    - They have not signed any other block for epoch *e.*

- A node reports a block as finalized if it is the parent of a notarized block



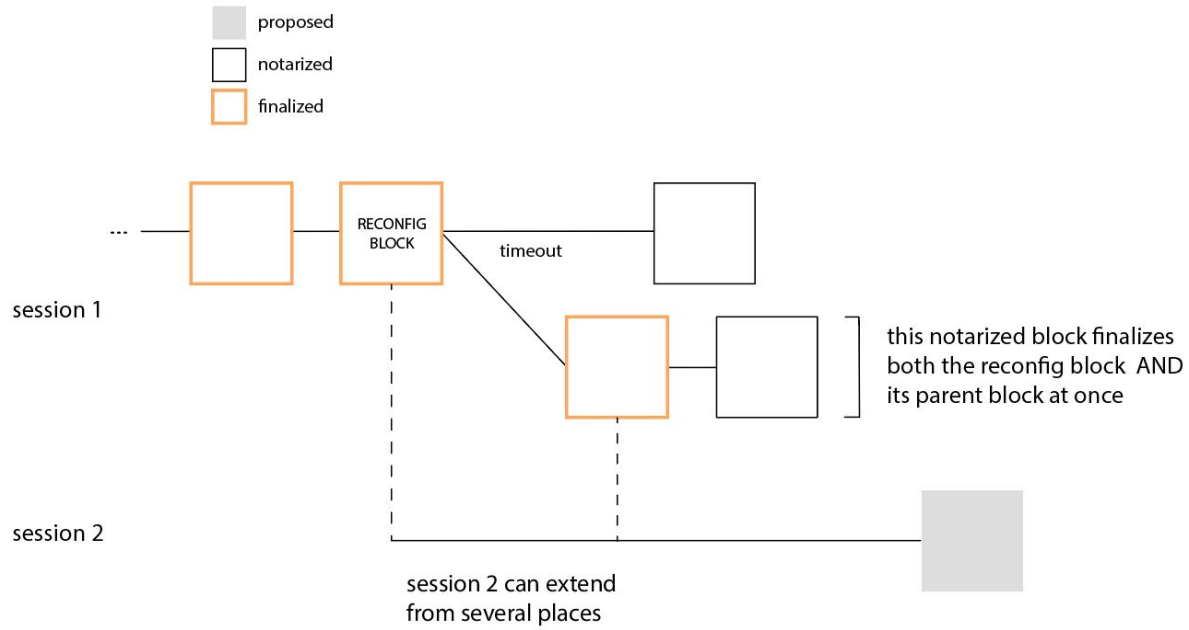
With these simple rules, the PaLa consensus protocol achieves liveness and consistency. The PaLa protocol is simple and rigorously proven. PaLa is designed based on partially synchronous network assumption. It is inherently partition tolerant and responsive. If there is ever a network partition, there is only a temporary outage in liveness. It will resume progress when the partition heals with no conflict in state.

Committee Reconfiguration

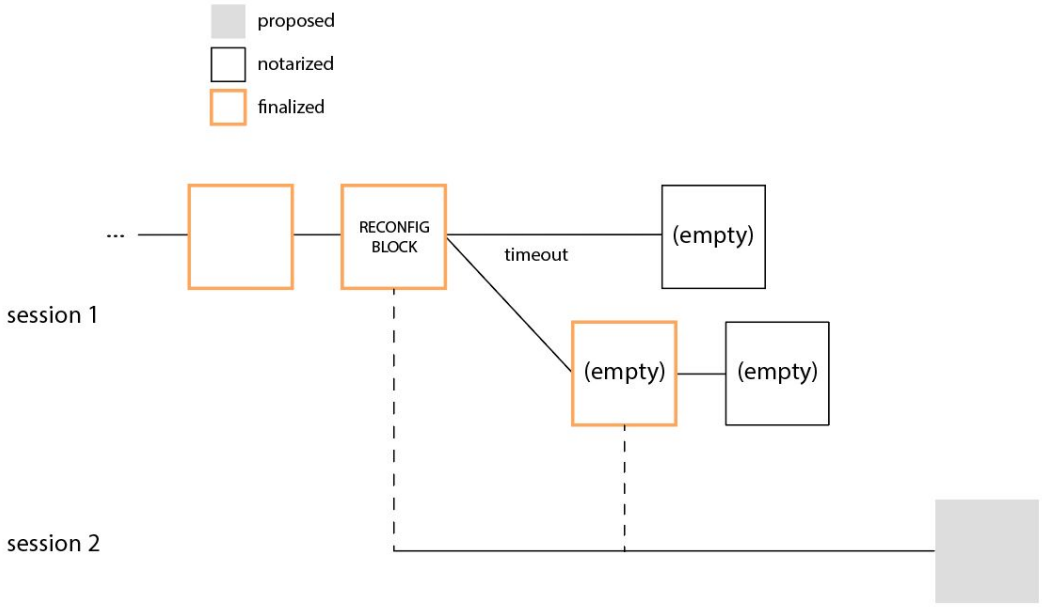
In a real world Proof-of-Stake blockchain, we want to periodically switch committees as stake changes hands in the system. The PaLa consensus algorithm supports fast and seamless committee switches. The basic idea is that the previous committee must finalize a special **reconfiguration block** before the next committee may serve to ensure consistent continuity.

The following assumes a mature understanding of the PaLa consensus protocol so don't be alarmed if you're reading it for the first time. It's nonetheless published here for readers to revisit as reference in the future.

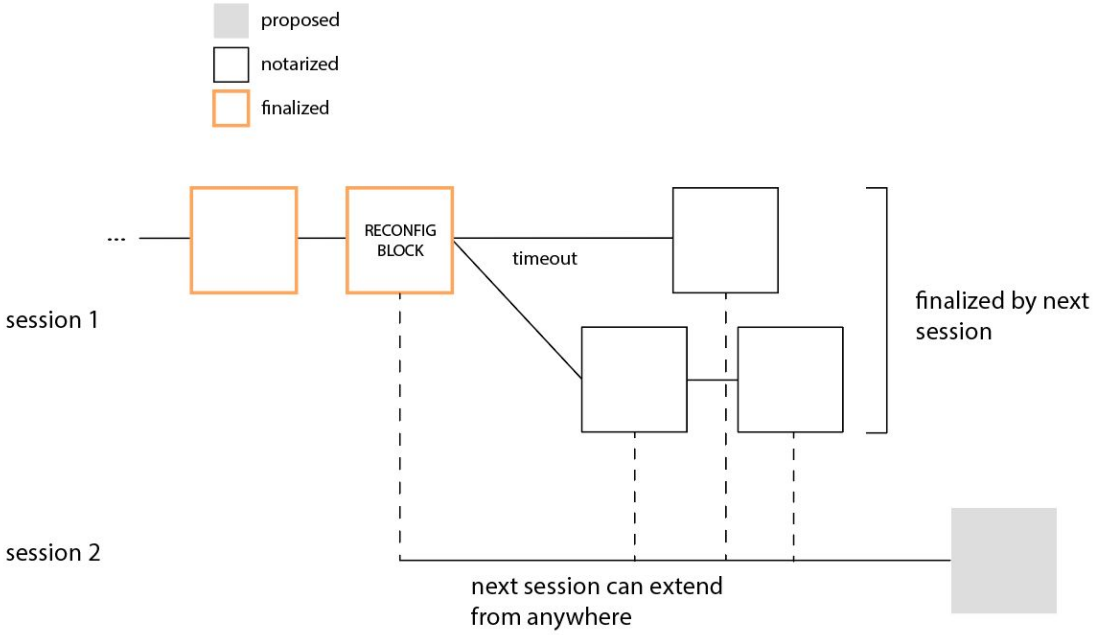
So far, PaLa only guarantees consistency history with a fixed committee. It's possible, just before a reconfiguration, that the previous committee might finalize an extension of the reconfiguration block. Remember 2 notarized *normal* blocks in a row are needed for the first one to be finalized. Thus timeouts allow for multiple blocks in a row to be notarized and finalized all at once when the 2 normal blocks condition is met. If the next sessions extends from the reconfig block, the extended finalized chain from the reconfig block is lost and we lose consistency. There is no guarantee the new committee will see this extended finalized chain and so we can not require that they extend from it.



ThunderCore's implementation always extends from the reconfig block and requires that all blocks after the reconfiguration block must be empty. They only serve to finalize the reconfiguration block, and it doesn't matter if their content is lost.

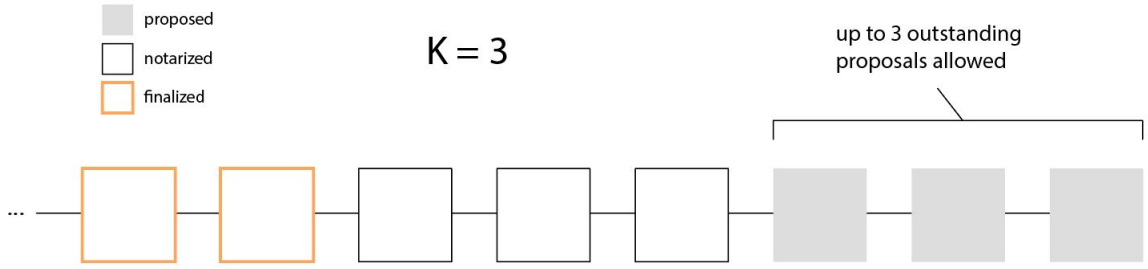


Another solution solution is to have blocks after a reconfiguration block be finalized by the *next* committee.

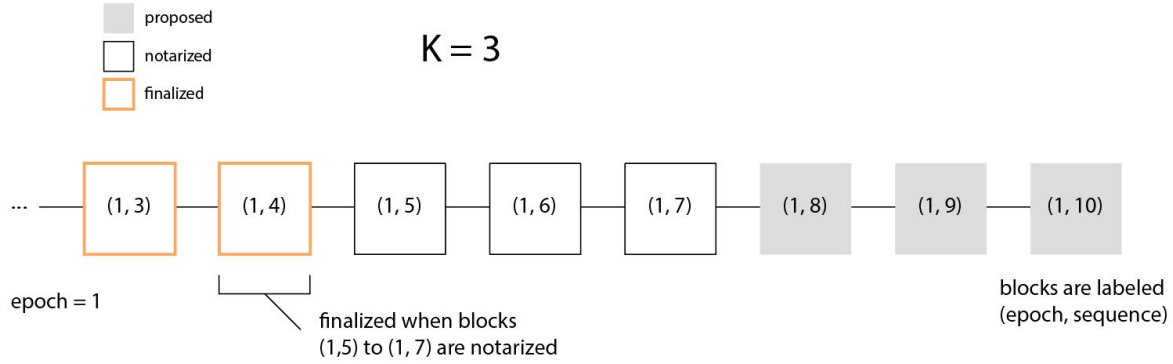


Doubly Pipelined PaLa

The full PaLa protocol described in the paper is called **Doubly Pipelined Pala** which supports the **proposal pipeline**[16] feature. This version allows for consensus to proceed *k* blocks at a time where *k* is a protocol set parameter. The proposal pipeline allows newer blocks to be buffered while older ones are still being voted on. Based on empirical testing, we found that when network latency is high relative to block times, a higher *k* value can improve throughput volume.



Since a single proposer may propose multiple blocks in sequence, a more nuanced block numbering scheme is needed. Blocks now have an *epoch* and *sequence* number. Proposers are chosen based on the epoch as before and each new proposal from the same sequence advances the sequence number which starts from 0 at the beginning of each epoch. A block is now finalized if its *k'th* child in the same epoch is notarized.



[16] Also referred to as "doubly streamlined" in an unpublished version of the PaLa paper. The author of this article prefers to term "proposal pipeline".

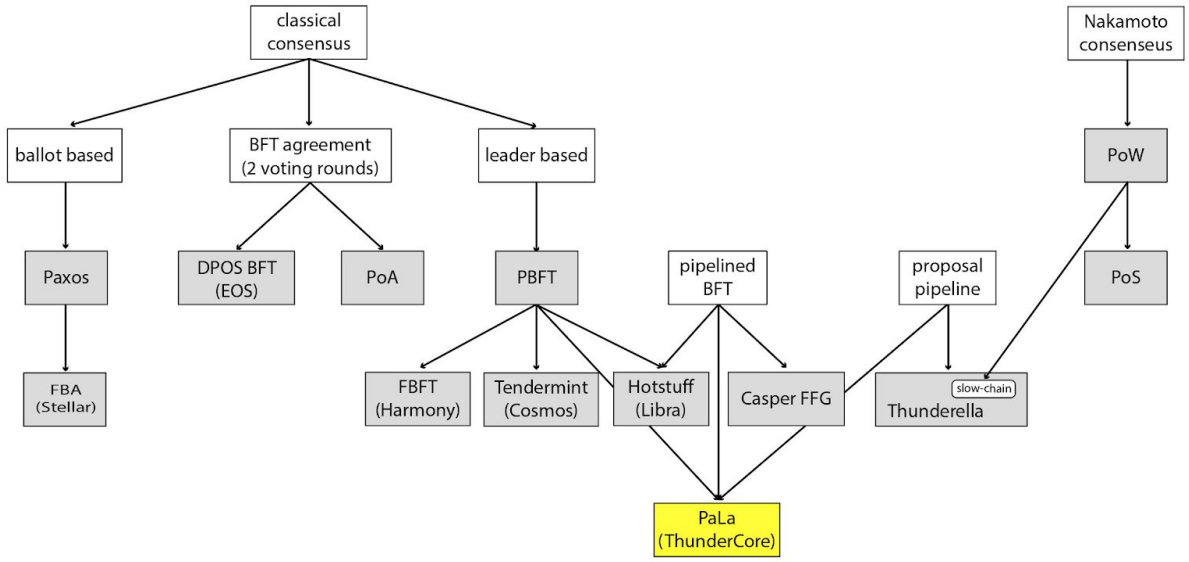
The remainder of the protocol is very similar to PaLa. The proofs for consistency and liveness also follow suite. We'll dive into the details of this in a future blog post comparing PaLa and Hotstuff.

Performance

From a performance standpoint, PaLa is a significant improvement on prior classical consensus protocols that require two rounds of voting per block and $O(n^2)$ messages. PaLa makes use of the pipelined BFT idea where the second round of voting in this classical consensus protocol is piggy-backed on the first round of voting for the next block. The active proposer uses BLS multi-signatures to collect votes and distribute notarizations. Together with a multi-hub-and-spoke network topology, PaLa can achieve consensus with just O(n) messages.

Takeaway

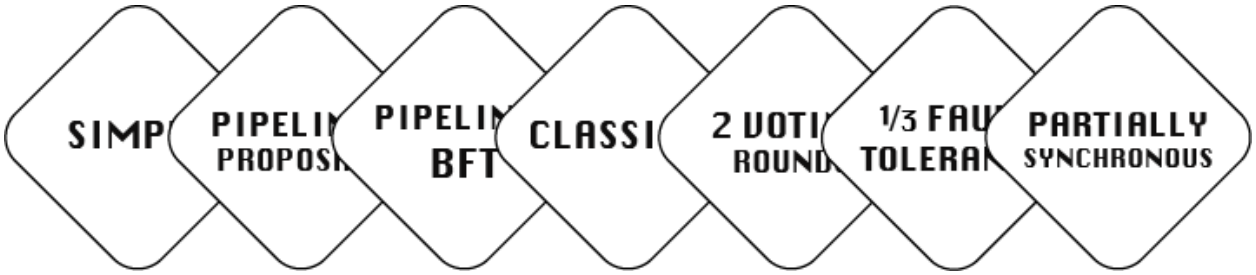
PaLa is the culmination of decades of distributed consensus research (spurned and funded by the emerging cryptocurrency paradigm). It brings together bold ideas from many protocols elegantly combining them into a single efficient and simple protocol. This should be apparent from the very short description of the algorithm presented above. The PaLa research paper provides simple and rigorous proofs to its security properties based on realistic network assumptions.



Newer BFT consensus algorithms such as Tendermint, FBFT, Casper FFG and Hotstuff use many of the same innovations as PaLa. However, none of these algorithms are as simple, elegant and optimal as PaLa.

We expect BFT consensus to be equated with PaLa the same way Distributed Hash Table (DHT) is equated with Kademlia.

The reference implementation of the PaLa consensus protocol is already released and contains thorough documentation and educational material.



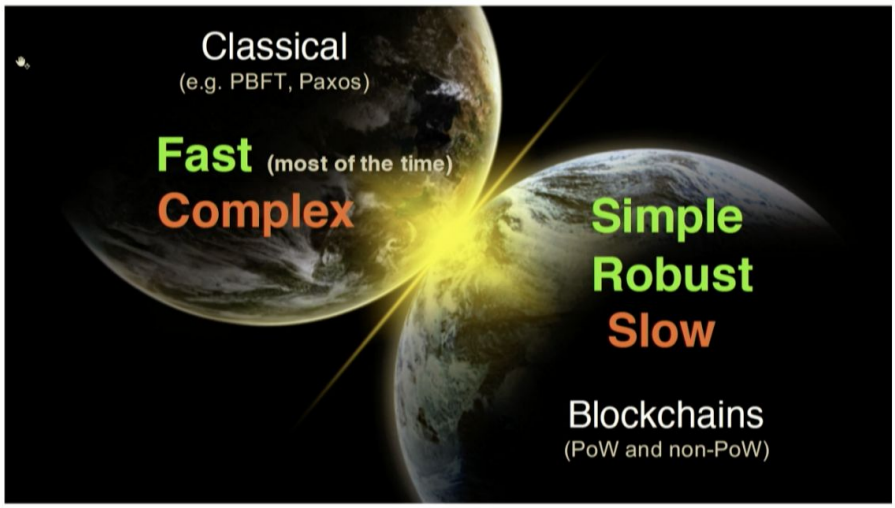
Conclusion and More!

Wow, awesome job for making it to the end! Are you expecting a reward? Knowledge is a reward in and of itself!!! This is just the beginning. We hope you are better equipped to navigate the world of blockchain consensus. For more information about ThunderCore please check out our website or drop us a line @ThunderProtocol.

Of course, this conclusion is not the conclusion! Below are follow-up articles on still more consensus protocols. More will be added over time. Let us know what you're interested in seeing next!

Thunderella

The major innovation of the [Thunderella consensus protocol] is the combination of classical consensus with Nakamoto's chain-style consensus, which brings the best of both worlds.



Classical protocols are extremely fast at a small scale. Transactions can be confirmed at speeds approaching centralized deployments. However, these protocols can be complex, difficult to implement, and do not scale in number of participants. In contrast, Nakamoto protocols are simple, have additional robustness properties, can scale in participants indefinitely, but take minutes to confirm transactions with high probability.

Recall in our description of PBFT, a view-change is executed if no proposal is seen from the primary for a sufficiently long period of time. During a view-change, nodes exchange messages to agree upon the following:

1. A view change has indeed happen
2. What the state of the network is just before the switch

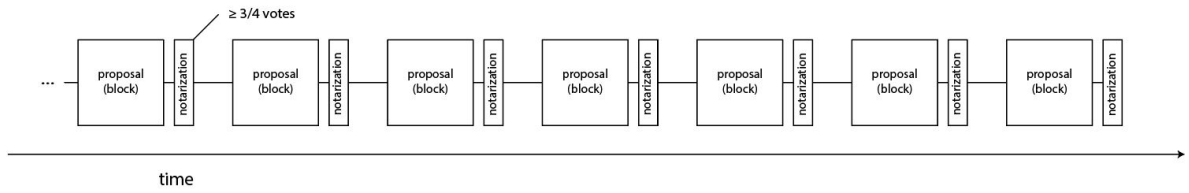
In effect, nodes are coming to consensus *without* a primary proposer. The key observation for the Thunderella protocol is that we can rely on a Proof-of-Work *slow-chain* for consensus when it comes to 1. and 2. which we'll call *fallback* and *recovery* respectively.

The Protocol

Assume a prior agreed upon set of accelerators and committee members (same as proposers and voters). A simplified version of the Thunderella protocol, which creates blocks on the *fast-path* can be described simply as follows:

Fast-Path

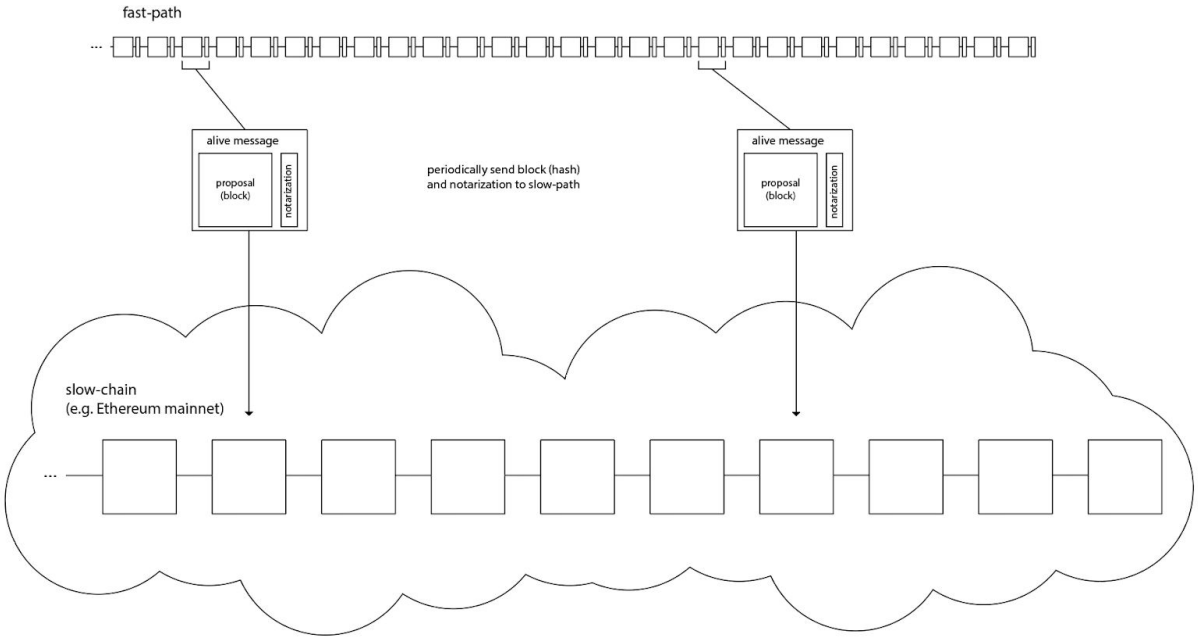
1. The active accelerator signs a proposal (block) to be added to the blockchain (linearly ordered log) and sends it to the committee.
2. Each committee member votes on the proposal by signing it if it is a valid extension of their blockchain and returns their vote to the accelerator.
3. Upon collecting ¾ of the committee's votes, the active accelerator combines them into a *notarization* and broadcasts it.
4. If a notarization for a proposal is observed, that proposal is finalized and part of the immutable history of the blockchain.



Slow-Chain

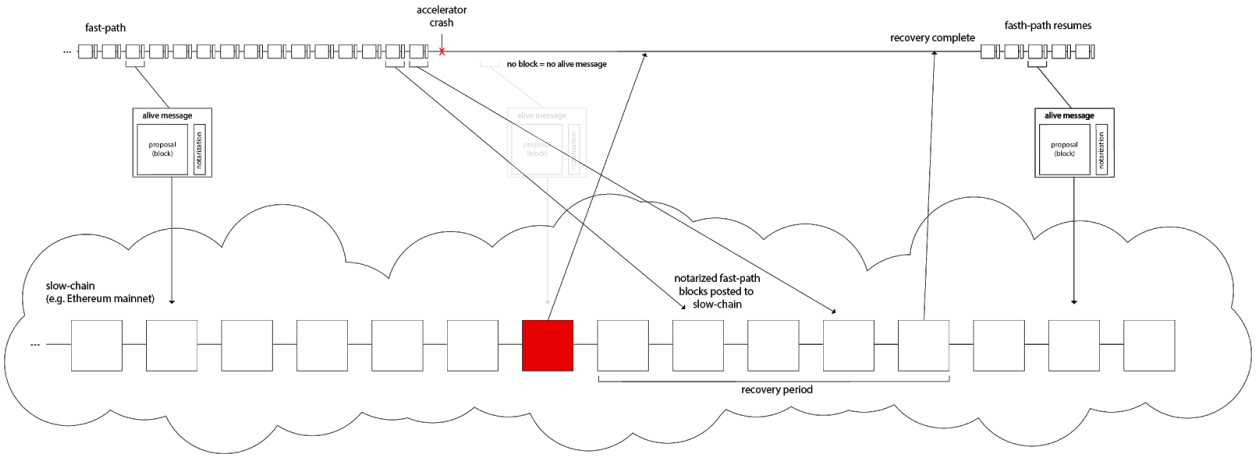
The slow-chain is another Proof-of-Work blockchain where consistency and liveness are guaranteed. Here when we say something is "seen on the slow-chain" we mean that it is safe to assume (by the finality assumptions of the slow-chain blockchain) that all participants of the network have also seen it. In practice, this may mean waiting a few blocks for probabilistic finality.

1. Every 100th fast-path blocks (say), a hash and notarization of the block must be posted to the slow-chain in a timely manner. This is called an *alive* message.
    a. This can be accomplished by including the alive message payload in the slow-chain transaction data field (say).
    b. Note, that anyone can post this alive message and alive messages can not be forged as they require a valid notarization of the block.



Fallback and Recovery

1. When no new alive message is seen on the slow-chain after 20 slow-chain blocks (say) the fast-path is down and recovery begins. Committee members stop signing new proposals.
2. Recovery lasts for 10 slow-chain blocks (say) during which time participants will post a valid most recently seen block hash and notarization to the slow-chain.
3. After recovery, the next accelerator chosen based on a round robin policy (say) comes online and proposes from the latest block reported during the recovery phase.



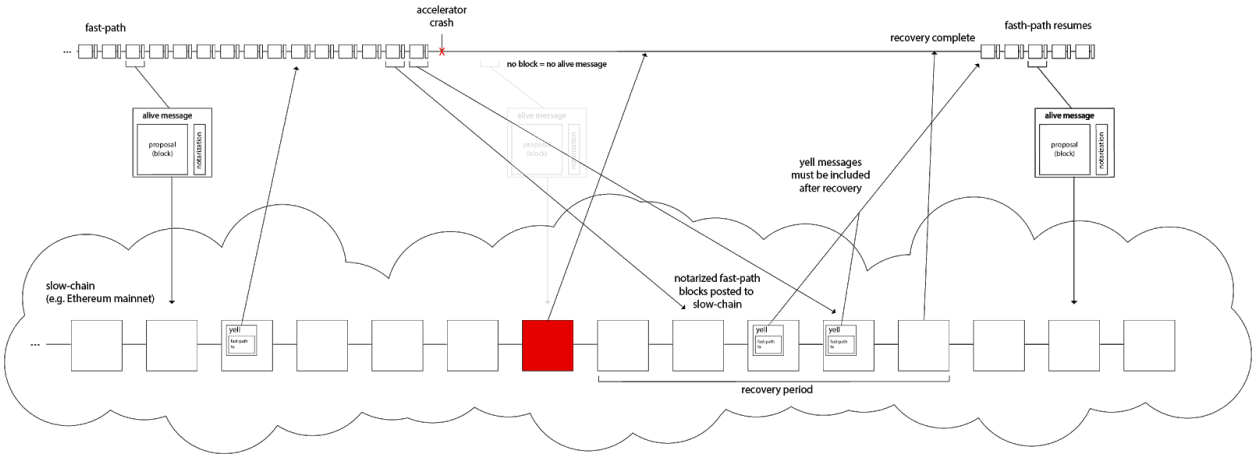
Note that step 2. should look a little familiar. In the view-change sub-protocol of PBFT, replicas (nodes) are required to send the last prepared (notarized) block in their view to coordinate the

recovery point. In comparison, by leveraging the slow-chain, this recovery coordination process is simpler and more reliable (thus requiring weaker assumptions).

Yell Messages

As an additional tool for ensuring censorship resistance and liveness, Thunderella defines *yell messages*. Yell messages are fast-path transactions that are sent to the slow-chain. These transactions are included in the fast-path by rules defined by the protocol (rather than the propose-notarize process for regular transactions). Thus even during recovery, a finalized yell message on the slow-chain is also a finalized fast-path transaction.

1. A user can wrap a valid signed fast-path transaction into a *yell* message (in the data field of a slow-chain transaction say) and post to the slow-chain.
2. When the slow-chain block containing the yell message is finalized, committee members expect to see this transaction appear on the fast-path. If it does not show up, fallback is triggered and the protocol enters recovery.
3. During recovery, each node on the network extends their blockchain from the last fast-path block based on yell messages they see on the slow-chain according a deterministic set of rules.[17] Since we can always rely on PoW consensus on the slow-chain, every node independently arrives at the same extended fast-path.
4. Upon recovery, the new accelerator must extend from the extended fast-path chain in the previous step.



In the full protocol description, Thunderella goes even faster by allowing for multiple outstanding proposals (blocks that are not notarized). This trick is called *proposal pipelining* and it's the same trick used to improve throughput in PaLa. The full Thunderella protocol is described in the [Thunderella litepaper].

[17] The high level concept for yell messages is straightforward. However, the exact set of rules for yell message is a crucial and nuanced implementation detail.

Takeaway

As Thunderella leverages a synchronous Proof-of-Work blockchain with ½ fault tolerance for the slow path, the voting threshold is ¾ which gives 50% fault tolerance for consistency on the partially synchronous fast path. Typically a partially asynchronous could achieve at best ⅓ fault tolerance for liveness as explained previously. However, by leveraging a synchronous slow-chain for recovering from a failed accelerator, Thunderella also achieves ½ fault tolerance for liveness. Note that the slow-chain need not be a PoW chain. However, the security of the protocol inherits the security of the underlying chain. For example, using a partially synchronous classical consensus slow-chain will result in the protocol being partially synchronous and having up to only ⅓ fault tolerance.

In practice, Thunderella provides confirmation after just 1 round of votes! With a PoW slow chain such as Ethereum, recovery may take several minutes where progress is slow and expensive. If the fast-path is reliable, recovery is rarely needed. The slow recovery issue can be addressed by using a faster classical consensus blockchain for the slow chain. While it was originally intended to be used as a layer 1 scaling solution for ThunderCore, Thunderella may be even more appropriate for a layer 2 side chain with an appropriate cross-chain bridge between the fast-path and slow-chain. For example, multiple untrusted operators (perhaps chosen through staking an ERC20 token) could use Thunderella as a Plasma side chain and have one message round finality!