Solana Alpenglow Consensus

Increased Bandwidth, Reduced Latency

Quentin Kniep Jakub Sliwinski Roger Wattenhofer

Anza

White Paper v1.1, July 22, 2025

Abstract

In this paper we describe and analyze Alpenglow, a consensus protocol tailored for a global high-performance proof-of-stake blockchain.

The voting component *Votor* finalizes blocks in a single round of voting if 80% of the stake is participating, and in two rounds if only 60% of the stake is responsive. These voting modes are performed concurrently, such that finalization takes $\min(\delta_{80\%}, 2\delta_{60\%})$ time after a block has been distributed.

The fast block distribution component *Rotor* is based on erasure coding. Rotor utilizes the bandwidth of participating nodes proportionally to their stake, alleviating the leader bottleneck for high throughput. As a result, total available bandwidth is used asymptotically optimally.

Alpenglow features a distinctive "20+20" resilience, wherein the protocol can tolerate harsh network conditions and an adversary controlling 20% of the stake. An *additional* 20% of the stake can be offline if the network assumptions are stronger.

1 Introduction

"I think there is a world market for maybe five computers." – This quote is often attributed to Thomas J. Watson, president of IBM. It is disputed whether Watson ever said this, but it was certainly in the spirit of the time as similar quotes exist, e.g., by Howard H. Aiken. The quote was often made fun of in the last decades, but if we move one word, we can probably agree: "I think there is a market for maybe five *world* computers."

So, what is a *world computer*? In many ways a world computer is like a common desktop/laptop computer that takes commands ("transactions") as input and then changes its bookkeeping ("internal state") accordingly. A world computer provides a shared environment for users from all over the world. Moreover, a world computer itself is distributed over the entire world:

Instead of just having a single processor, we have dozens, hundreds or thousands of processors, connected through the internet.

Such a world computer has a big advantage over even the most advanced traditional computer: The world computer is much more fault tolerant, as it can survive a large number of crashes of individual components. Beyond that, no authority can corrupt the computer for other users. A world computer must survive even if some of its components are controlled by an evil botnet. The currently common name for such a world computer is *blockchain*.

In this paper we present Alpenglow, a new blockchain protocol. Alpenglow uses the Rotor protocol, which is an optimized and simplified variant of Solana's data dissemination protocol Turbine [Fou19]. Turbine brought erasure coded information dispersal [CT05] to permissionless blockchains. Rotor uses the total amount of available bandwidth provided by the nodes. Because of this, Rotor achieves an asymptotically optimal throughput. In contrast, consensus protocols that do not address the leader bandwidth bottleneck suffer from low throughput.

Intuition

The Votor consensus logic at the core of Alpenglow inherits the simplicity from the Simplex protocol line of work [CP23; Sho24] and translates it to a proof-of-stake context, resulting in natural support for rotating leaders without complicated view changes. In the common case, we achieve finality in a single round of voting, while a conservative two-round procedure is run concurrently as backup [SSV25; Von+24].

1.1 Alpenglow Overview

First, let us provide a high-level description of Alpenglow. We are going to describe all the individual parts in detail in Section 2.

Alpenglow runs on top of n computers, which we call nodes, where n can be in the thousands. This set of nodes is known and fixed over a period of time called an epoch. Any node can communicate with any other node in the set by sending a direct message.

Alpenglow is a proof-of-stake blockchain, where each node has a known stake of cryptocurrency. The stake of a node signals how much the node contributes to the blockchain. If node v_2 has twice the stake of node v_1, node v_2 will also earn twice the fees, and provide twice the outgoing network bandwidth.

Time is partitioned into slots. Each time slot has a slot number and a designated leader from the set of nodes. Each leader will be in charge for a fixed amount of consecutive slots, known as the leader window. A threshold verifiable random function determines the leader schedule.

While a node is the leader, it will receive all the new transactions, either directly from the users or relayed by other nodes. The leader will construct a block with these transactions. A block consists of slices for pipelining. The slices themselves consist of shreds for fault tolerance and balanced dispersal

(Section 2.1). The leader incorporates the Rotor algorithm (Section 2.2), which is based on erasure coding, to disseminate the shreds. In essence, we want the nodes to utilize their total outgoing network bandwidth in a stake-fair way, and avoid the common pitfall of having a leader bottleneck. The leader will continuously send its shreds to relay nodes, which will in turn forward the shreds to all other nodes.

As soon as a block is complete, the (next) leader will start building and disseminating the next block. Meanwhile, concurrently, every node eventually receives that newly constructed block. The shreds and slices of the incoming blocks are stored in the Blokstor (Section 2.3).

Nodes will then vote on whether they support the block. We introduce different types of votes (and certificates of aggregated votes) in Section 2.4. These votes and certificates are stored in a local data structure called Pool (Section 2.5).

With all the data structures in place, we discuss the voting algorithm Votor in Section 2.6: If the block is constructed correctly and arrives in time, a node will vote for the block. If a block arrives too late, a node will instead vote to skip the block (since either the leader cannot be trusted, or the network is unstable). If a super-majority of the total stake votes for a block, the block can be finalized immediately. However, if something goes wrong, an additional round of voting will decide whether or not to skip the block.

In Section 2.7 we discuss the logic of creating blocks as a leader, and how to decide on where to append the newly created block.

Finally, in Section 2.8 we discuss Repair – how a node can get missing shreds, slices or blocks from other peers. Repair is needed to help nodes to retrieve the content of an earlier block that they might have missed, which is now an ancestor of a finalized block. This completes the main parts of our discussion of the consensus algorithm.

Protocol

We proceed to prove the correctness of Alpenglow. First, we prove safety (we do not make fatal mistakes even if the network is unreliable, see Section 2.9), then liveness (we do make progress if the network is reliable, see Section 2.10). Finally, we also consider a scenario with a high number of crash failures in Section 2.11.

While not directly essential for Alpenglow's correctness, Section 3 examines various concepts that are important for Alpenglow's understanding. First we describe our novel Rotor relay sampling algorithm in Section 3.1. Next, we explore how transactions are executed in Section 3.2.

Then we move on to advanced failure handling. In Section 3.3 we consider how a node re-connects to Alpenglow after it lost contact, and how the system can "re-sync" when experiencing severe network outages. Then we add dynamic timeouts to resolve a crisis (Section 3.4).

In the last part, we present potential choices for protocol parameters (Section 3.5). Based on these we show some measurement results; to better understand possible efficiency gains, we simulate Alpenglow with Solana's current

Intuition

node and stake distribution, both for bandwidth (Section 3.6) and latency (Section 3.7).

In the remainder of *this* section, we present some preliminaries which are necessary to understand the paper. We start out with a short discussion on security design goals in Section 1.2 and performance metrics in Section 1.3. In Section 1.4 we discuss how Alpenglow relates to other work on consensus. Finally we present the model assumptions (Section 1.5) and the cryptographic tools we use (Section 1.6).

1.2 Fault Tolerance

Safety and security are the most important objectives of any consensus protocol. Typically, this involves achieving resilience against adversaries that control up to 33% of the stake [PSL80]. This 33% (also known as "$3f + 1$") bound is *everywhere* in today's world of fault-tolerant distributed systems.

When discovering the fundamental result in 1980, Pease et al. considered systems where the number of nodes n was small. However, today's blockchain systems consist of *thousands* of nodes! While the 33% bound of [PSL80] also holds for large n, attacking one or two nodes is not the same as attacking thousands. In a large scale proof-of-stake blockchain system, running a thousand malicious ("byzantine") nodes would be a costly endeavor, as it would likely require billions of USD as staking capital. Even worse, misbehavior is often punishable, hence an attacker would lose all this staked capital.

So, in a *real* large scale distributed blockchain system, we will probably see *significantly less* than 33% byzantines. Instead, realistic bad behavior often comes from machine misconfigurations, software bugs, and network or power outages. In other words, large scale faults are likely accidents rather than coordinated attacks.

This *attack model paradigm shift* opens an opportunity to reconsider the classic $3f + 1$ bound. Alpenglow is based on the $5f + 1$ bound that has been introduced in [DGV04] and [MA06]. While being *less* tolerant to orthodox byzantine attacks, the $5f + 1$ bound offers other advantages. Two rounds of voting are required for finalization if the adversary is strong. However, if the adversary possesses less stake, or does not misbehave all the time, it is possible for a correct $5f + 1$ protocol to finalize a block in just *a single round* of voting.

In Sections 2.9 and 2.10 we rely on Assumption 1 to show that our protocol is correct.

Assumption 1 (fault tolerance)**.** *Byzantine nodes control less than 20% of the stake. The remaining nodes controlling more than 80% of stake are correct.*

As we explain later, Alpenglow is partially-synchronous, and Assumption 1 is enough to ensure that even an adversary completely controlling the network

(inspecting, delaying, and scheduling communication between correct nodes at will) cannot violate safety. A network outage or partition would simply cause the protocol to pause and continue as soon as communication is restored, without any incorrect outcome.

However, if the network is not being attacked, or the adversary does not leverage some network advantage, Alpenglow can tolerate an even higher share of nodes that simply crash. In Section 2.11 we intuitively explain the difference between Assumption 1 and Assumption 2, and we sketch Alpenglow's correctness under Assumption 2.

Intuition

Assumption 2 (extra crash tolerance)**.** *Byzantine nodes control less than 20% of the stake. Other nodes with up to 20% stake might crash. The remaining nodes controlling more than 60% of the stake are correct.*

1.3 Performance Metrics

Alpenglow achieves the fastest possible consensus. In particular, after a block is distributed, our protocol finalizes the block in $\min(\delta_{80\%}, 2\delta_{60\%})$ time. We will explain this formula in more detail in Section 1.5; in a nutshell, δ_θ is a network delay between a stake-weighted fraction θ of nodes. To achieve this finalization time, we run an 80% and a 60% majority consensus mechanism concurrently. A low-latency 60% majority cluster is likely to finish faster on the 2δ path, whereas more remote nodes may finish faster on the single δ path, hence $\min(\delta_{80\%}, 2\delta_{60\%})$. Having low latency is an important factor deciding the blockchain's usability. Improving latency means establishing transaction finality faster, and providing users with results with minimal delay.

Another common pain point of a blockchain is the system's throughput, measured in transaction bytes per second or transactions per second. In terms of throughput, our protocol is using the total available bandwidth asymptotically optimally.

After achieving the best possible results across these main performance metrics, it is also important to minimize protocol overhead, including computational requirements and other resource demands.

Moreover, in Alpenglow, we strive for simplicity whenever possible. While simplicity is difficult to quantify, it remains a highly desirable property, because simplicity makes it easier to reason about correctness and implementation. A simple protocol can also be upgraded and optimized more conveniently.

Intuition

1.4 Related Work

Different consensus protocols contribute different techniques to address different performance metrics. Some techniques can be translated from one protocol to another without compromise, while other techniques cannot. In the following we describe each protocol as it was originally published, and not what techniques could hypothetically be added to the protocol.

Increase Bandwidth. In classic leader-based consensus protocols such as PBFT [CL99], Tendermint [BKM18] or HotStuff [Yin+19], at a given time one leader node is responsible for disseminating the proposed payload to all replicas. This bandwidth bottleneck can constitute a defining limitation on the throughput [Dan+22; Mil+16; SDV19].

DAG protocols [Dan+22; Spi+22] are a prominent line of work focused on addressing this concern. In these protocols data dissemination is performed by all nodes. Unfortunately, protocols following the DAG approach exhibit a latency penalty [Aru+25]. Some DAG protocols [Kei+22] reduce the latency penalty by foregoing "certifying" the disseminated data. For example, in Mysticeti [Bab+25] the leader block can be confirmed in two rounds of voting, i.e., after disseminating the block and observing two block layers referencing this block (corresponding to 3 network delays, or 3δ). However, most of the data (all non-leader blocks) is ordered by the protocol when a leader block "certifying" the data is finalized. In other words, most of the throughput is confirmed with a latency of 5δ. Some researchers raise concerns that this technique impacts the robustness of the protocol [Aru+24].

Another prominent technique used to alleviate the leader bottleneck for high throughput involves erasure coding [CT05; Sho24; Yan+22]. Solana [Fou19; Yak18] pioneered this approach in blockchains. In this technique, the leader erasure-codes the payload into smaller fragments. The fragments are sent to different nodes, which in turn participate in disseminating the fragments, making the bandwidth use balanced. Alpenglow follows this line of work.

A recent study [LNS25] proposes a framework to compare the impact of above-mentioned techniques on throughput and latency in a principled way. The study indicates that erasure coding of the payload (represented by DispersedSimplex [Sho24]) achieves better latency than DAG protocols.

Reduce Latency. A long line of work proposes consensus protocols that can terminate after one round of voting, typically called fast or one-step consensus. This approach has received a lot of attention, e.g., [DGV04; GV07; Kot+07; Kur02; Lam03; MA06; SR08]. Protocols DGV [DGV04] and FaB Paxos [MA06] introduce a parametrized model with $3f+2p+1$ replicas, where $p \geq 0$. The parameter p describes the number of replicas that are not needed for the fast path. These protocols can terminate optimally fast in theory (2δ, or 2 network delays) under optimistic conditions. Liveness and safety issues of

landmark papers were later pointed out [Abr+17], showcasing the complexity of the domain and thus posing the research question of fast consensus again. SBFT [Gue+19] addressed the correctness issues. SBFT can terminate after one round of voting, but is optimized for linear message complexity, therefore incurring higher latency.

As pointed out by [DGV04], and later in [KTZ21] and [Abr+21], the lower bound of $3f + 2p + 1$ actually only applies to a restricted type of protocol. These works prove the lower bound and show single-shot consensus protocols that use only $3f + 2p^* - 1$ replicas, with $p^* \geq 1$.

Interestingly, in practice, one-step protocols might *increase* the finalization latency, as one-round finalization requires voting between $n-p$ replicas, which could be slower than two rounds of voting between $n - f - p$ replicas that are more concentrated in a geographic area. Banyan [Von+24] renewed interest in fast BFT protocols, as it performs a one-round and a two-round mechanism in parallel, guaranteeing the best possible latency.

Concurrent Work. Kudzu [SSV25] is Alpenglow's "academic sibling" with a simpler theoretical model. Like Alpenglow, Kudzu features high throughput via the previously mentioned technique of erasure coding, and one- and two-round parallel finalization paths. The differences between Alpenglow and Kudzu include:

- Kudzu is specified in a permissioned model, while Alpenglow is a proof-of-stake protocol. In many protocols merely the voting weight of nodes would be impacted by this difference. However, disseminating erasure-coded data cannot be easily translated between these models.
- Alpenglow features leader windows where the leader streams the data without interruption, improving throughput. Concurrent processing of slots allows block times to be shorter than the assumed latency bound (Δ).
- Alpenglow features fast leader handoff. When the leader is rotated, the next leader can start producing a block as soon as it has received the previous block.
- With Assumption 3, Alpenglow features higher resilience to crash faults.
- In Kudzu, due to the different model, nodes can vote as soon as they receive the first fragment of a block proposal, while in Alpenglow nodes vote after reconstructing a full proposal. In theory, the former is faster, while in practice, the difference is a fraction of one network delay.
- The data expansion ratio associated with erasure coding can be freely set in Alpenglow. We suggest a ratio of 2, while in Kudzu the ratio needs to be higher.

Intuition

Follow-up Work. Hydrangea [SKN25] is a protocol proposed after Alpenglow that parametrizes resilience to byzantine and crash faults in a way related to Alpenglow. The protocol requires $n = 3f + 2c + k + 1$, and tolerates f byzantine faults and c crash faults in partial synchrony. The number of nodes not needed for finalization in one round of voting is then $p = \lfloor \frac{c+k}{2} \rfloor$. For example, to terminate in one round of voting among 80% of nodes, Hydrangea would set $p = c = k = 20\%$ and $f = 13\%$, for a total of 33% of tolerated faulty nodes. In contrast, Alpenglow can tolerate $f < 20\%$ and a total of 40% of faulty nodes, but needs Assumption 3 for fault rates higher than 20%.

Hydrangea suffers from a bandwidth bottleneck at the leader and, in our view, remains underspecified for practical implementation. However, the parametrization is an interesting contribution than could also be applied to Alpenglow.

1.5 Model and Preliminaries

Names. We introduce various objects of the form $\mathsf{Name}(x, y)$. This indicates some deterministic encoding of the object type "Name" and its parameters x and y.

Epoch. To allow for changing participants and other dynamics, the protocol rejuvenates itself in regular intervals. The time between two such changes is called an epoch. Epochs are numbered as $e = 1, 2, 3$, etc. The participants register/unregister two epochs earlier, i.e., the participants (and their stake) of epoch $e + 1$ are decided at the end of epoch $e - 1$, i.e., a long enough time before epoch $e + 1$ starts. This makes sure that everybody is in agreement on the current nodes and their stake at the beginning of epoch $e + 1$.

Node. We operate on n individual computers, which we call nodes $v_1, v_2, \ldots, v_n$. The main jobs of these nodes are to send/relay messages and to validate blocks. Because of this, nodes are sometimes also called validators in the literature. While the set of nodes changes with every new epoch, as mentioned in the previous paragraph, the nodes are static and fixed during an epoch. The set of nodes is publicly known, i.e., each node knows how to contact (IP address and port number) every node v_i. Each node has a public key, and all nodes know all public keys. The information of each node (public key, stake, IP address, port number, etc.) is announced and updated by including the information in a transaction on the blockchain. This guarantees that everybody has the same information. Currently, Solana has $n \approx 1{,}500$ nodes, but our protocol can in principle scale to higher numbers. However, for practical reasons it may be beneficial to bound $n \leq n_{\max}$.

Message. Nodes communicate by exchanging authenticated messages. Our protocol never uses large messages. Specifically, all messages are less than

1,500 bytes [Pos84]. Because of this, we use UDP with authentication, so either QUIC-UDP or UDP with a pair-wise message authentication code (MAC). The symmetric keys used for this purpose are derived with a key exchange protocol using the public keys.

Broadcast. Sometimes, a node needs to broadcast the same message to *all* ($n-1$ other) nodes. The sender node simply loops over all other nodes and sends the message to one node after the other. Despite this loop, the total delay is dominated by the network delay. With a bandwidth of 1Gb/s, transmitting $n = 1{,}500$ shreds takes 18 ms (well below the average network delay of about 80 ms). To get to 80% of the total stake we need to reach $n \approx 150$ nodes, which takes only about 2 ms. Voting messages are shorter, and hence need even less time. Moreover, we can use a multicast primitive provided by an alternative network provider, e.g., DoubleZero [FMW24] or SCION [Zha+11].

Stake. Each node v_i has a known positive stake of cryptocurrency. We use $\rho_i > 0$ to denote node v_i's fraction of the entire stake, i.e., $\sum_{i=1}^{n} \rho_i = 1$. Each (fractional) stake ρ_i stays fixed during the epoch. The stake of a node signals how much the node contributes to the blockchain. If node v_2 has twice the stake of node v_1, node v_2 will also earn roughly twice the fees. Moreover, node v_2 also has twice the outgoing network bandwidth. However, all nodes need enough in-bandwidth to receive the blocks, and some minimum out-bandwidth to distribute blocks when they are a leader.

Time. We assume that each node is equipped with a local system clock that is reasonably accurate, e.g., 50 ppm drift. We do not consider clock drift in our analysis, but it can be easily addressed by incorporating the assumed drift into timeout periods. Clocks do not need to be synchronized at all, as every node only uses its local system clock.

Slot. Each epoch is partitioned into slots. A slot is a natural number associated with a block, and does not require timing agreements between nodes. The time period of a slot could start (and end) at a different local time for different nodes. Nevertheless, in normal network conditions the slots will become somewhat synchronized. During an epoch, the protocol will iterate through slots $s = 1, 2, \ldots, L$. Solana's current parameter of $L = 432{,}000$ is possible, but much shorter epochs, e.g., $L \approx 18{,}000$, could be advantageous, for instance to change stake more quickly. Each slot s is assigned a leader node, given by the deterministic function $\mathsf{leader}(s)$ (which is known before the epoch starts).

Leader. Each slot has a designated leader from the set of nodes. Each leader will be in charge for a fixed amount of consecutive slots, known as the

Protocol

leader window. A threshold verifiable random function [Dod02; MRV99] is evaluated before each epoch to determine a publicly known leader schedule that defines which node is the leader in what slot.

Timeout. Our protocol uses timeouts. Nodes set timeouts to make sure that the protocol does not get stuck waiting forever for some messages. For simplicity, timeouts are based on a global protocol parameter Δ, which is the maximum possible network delay between any two correct nodes when the network is in synchronous operation. However, timeout durations can be changed dynamically based on conditions, such that the protocol is correct irrespectively of the Δ exhibited by the network. For simplicity, we conservatively assume Δ to be a constant, e.g., $\Delta \approx 400$ ms. Importantly, timeouts do *not* assume synchronized clocks. Instead, only short periods of time are measured locally by the nodes. Therefore, the absolute wall-clock time and clock skew have no significance to the protocol. Even extreme clock drift can be simply incorporated into the timeouts - e.g. to tolerate clock drift of 5%, the timeouts can simply be extended by 5%. As explained later, Alpenglow is partially-synchronous, so no timing- or clock-related errors can derail the protocol.

Adversary. Some nodes can be byzantine in the sense that they can misbehave in arbitrary ways. Byzantine nodes can for instance forget to send a message. They can also collude to attack the blockchain in a coordinated way. Some misbehavior (e.g. signing inconsistent information) may be a provable offense, while some other misbehavior cannot be punished, e.g., sending a message late could be due to an extraordinary network delay. As discussed in Assumption 1, we assume that all the byzantine nodes together own strictly less than 20% of the total stake. Up to an additional 20% of the stake may be crashed under the conditions described in Section 2.11. The remaining nodes are *correct* and follow the protocol. For simplicity, in our analysis (Sections 2.9 to 2.11) we consider a static adversary over a period of one epoch.

Asynchrony. We consider the partially synchronous network setting of Global Stabilization Time (GST) [Con+24; DLS88]. Messages sent between correct nodes will eventually arrive, but they may take arbitrarily long to arrive. We always guarantee *safety*, which means that irrespectively of arbitrary network delays (known as the asynchronous network model), correct nodes output the same blocks in the same order.

Synchrony. However, we only guarantee *liveness* when the network is synchronous, and all messages are delivered quickly. In other words, correct nodes continue to make progress and output transactions in periods when messages between correct nodes are delivered "in time." In the model of GST, synchrony simply corresponds to a global worst-case bound Δ on mes-

sage delivery. The GST model captures periods of synchrony and asynchrony by stating that before the unknown and arbitrary time GST (global stabilization time) messages can be arbitrarily delayed, but after time GST all previous and future messages m sent at time t_m will arrive at the recipient at latest at time $\max(\mathsf{GST}, t_m) + \Delta$.

Network Delay. During synchrony, the protocol will rarely wait for a timeout. We model the actual message delay between correct nodes as δ, with $\delta \ll \Delta$. The real message delay δ is variable and unknown. Naturally, δ is not part of the protocol, and will only be used for the latency analysis. In other words, the performance of optimistically responsive protocols such as Alpenglow in the common case depends only on δ and not the timeout bound Δ. As discussed in Section 1.3, we use δ_θ to indicate how long it takes a fraction θ of nodes to send each other messages. More precisely, let S be a set of nodes with cumulative stake at least θ. In one network delay δ_θ, each node in S sends a message to every node in S. If $\theta = 60\%$ of the nodes are geographically close, then it is possible that $2\delta_{60\%}$ is *less time* than $\delta_{80\%}$, which needs only one network delay, but the involvement of 80% of the nodes.

Correctness. The purpose of a blockchain is to produce a sequence of *finalized* blocks containing transactions, so that all nodes output transactions in the same order. Every block is associated with a parent (starting at some notional genesis block). Finalized blocks form a single chain of parent-child links. When a block is finalized, all ancestors of the block are finalized as well.

Our protocol orders blocks by associating them with natural numbered slots, where a child block has to have a higher slot number than its parent. For every slot, either some block produced by the leader might be finalized, or the protocol can yield a *skip*. The blocks in finalized slots are transmitted in-order to the execution layer of the protocol stack. Definition 14 describes the conditions for block finalization. The guarantees of our protocol can be stated as follows:

- **Safety.** Suppose a correct node finalizes a block b in slot s. Then, if any correct node finalizes any block b' in any slot $s' \geq s$, b' is a descendant of b. (See also Theorem 1.)
- **Liveness.** In any long enough period of network synchrony, correct nodes finalize new blocks produced by correct nodes. (See also Theorem 2.)

1.6 Cryptographic Techniques

Hash Function. We have a collision-resistant hash function, e.g., SHA256.

Digital Signature. We have secure (non-forgeable) digital signatures. As stated earlier, each node knows the public key of every other node.

Aggregate Signature. Signatures from different signers may be combined non-interactively to form an aggregate signature. Technically, we only require non-interactive multi-signatures, which only enable signatures over the same message to be aggregated. This can be implemented in various ways, e.g. based on BLS signatures [Bon+03]. Aggregate signatures allow certificates to fit into a short message as long as $n \leq n_{\max}$.

Erasure Code. For integer parameters $\Gamma \geq \gamma \geq 1$, a (Γ, γ) *erasure code* encodes a bit string M of size m as a vector of Γ data pieces $d_1, \ldots, d_\Gamma$ of size $m/\gamma + O(\log \Gamma)$ each. The $O(\log \Gamma)$ overhead is needed to index each data piece. Erasure coding makes sure that any γ data pieces may be used to efficiently reconstruct M. The reconstruction algorithm also takes as input the length m of M, which we assume to be constant (achieved by padding smaller payloads).

In our protocol, the payload of a slice will be encoded using a (Γ, γ) Reed-Solomon erasure code [RS60], which encodes a payload M as a vector $d_1, \ldots, d_\Gamma$, where any γ d_i's can be used to reconstruct M. The data expansion rate is $\kappa = \Gamma/\gamma$.

Merkle Tree. A Merkle tree [Mer79] allows one party to commit to a vector of data $(d_1, \ldots, d_\Gamma)$ using a collision-resistant hash function by building a (full) binary tree where the leaves are the hashes of $d_1, \ldots, d_\Gamma$. Each leaf hash is concatenated with a label that marks the hash as a leaf, and each internal node of the tree is the hash of its two children. The root r of the tree is the commitment.

The *validation path* π_i for position $i \in \{1, \ldots, \Gamma\}$ consists of the siblings of all nodes along the path in the tree from the hash of d_i to the root r. The root r together with the validation path π_i can be used to prove that d_i is at position i of the Merkle tree with root r.

The validation path is checked by recomputing the hashes along the corresponding path in the tree, and by verifying that the recomputed root is equal to the given commitment r. If this verification is successful, we call d_i the data at position i with path π_i for Merkle root r. The collision resistance of the hash function ensures that no data $d_i' \neq d_i$ can have a valid proof for position i in the Merkle tree.

Encoding and Decoding. [CT05] The function *encode* takes as input a payload M of size m. It erasure codes M as $(d_1, \ldots, d_\Gamma)$ and builds a Merkle tree with root r where the leaves are the hashes of $d_1, \ldots, d_\Gamma$. The root of the tree r is uniquely associated with M. It returns $(r,\ \{(d_i, \pi_i)\}_{i \in \{1,\ldots,\Gamma\}})$, where each d_i is the data at position i with path π_i for Merkle root r.

The function *decode* takes as input $(r,\ \{(d_i, \pi_i)\}_{i \in \mathcal{I}})$, where $\mathcal{I}$ is a subset of $\{1, \ldots, \Gamma\}$ of size γ, and each d_i (of correct length) is the data at position i with path π_i for Merkle root r. Moreover, the decoding routine makes sure that the root r is correctly computed based on *all* Γ data pieces that correctly encode some message M', or it *fails*. If it fails, it guarantees that no set of γ data pieces associated with r can be decoded, and that r was (provably) maliciously constructed.

To ensure this pass/fail property, the decoding algorithm needs to check for each reconstructed data piece that it corresponds to the same root r. More precisely, *decode* reconstructs a message M' from the data $\{d_i\}_{i \in \mathcal{I}}$. Then, it encodes M' as a vector $(d'_1, \ldots, d'_\Gamma)$, and builds a Merkle tree with root r' with the hashes of $(d'_1, \ldots, d'_\Gamma)$ as leaves. If $r' = r$, *decode* returns M', otherwise it fails.

Protocol

2 The Alpenglow Protocol

In this section we describe the Alpenglow protocol in detail.

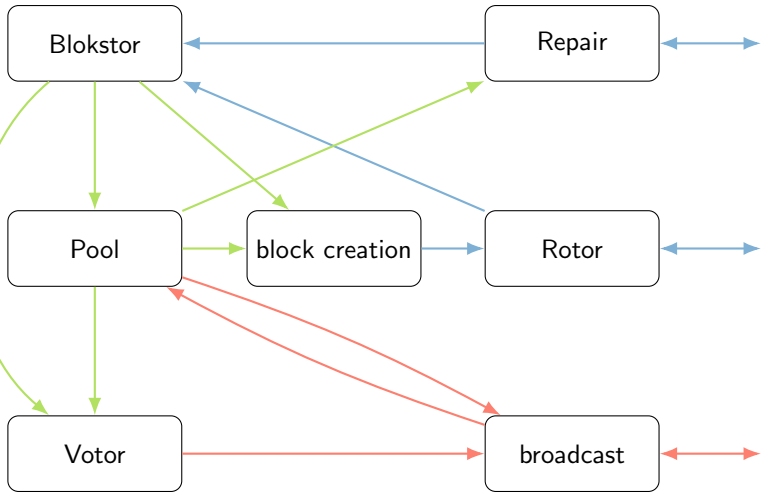

Figure 1: Overview of components of Alpenglow and their interactions. Arrows show information flow: block data in the form of shreds (blue), internal events (green), and votes/certificates (red).

Protocol

2.1 Shred, Slice, Block

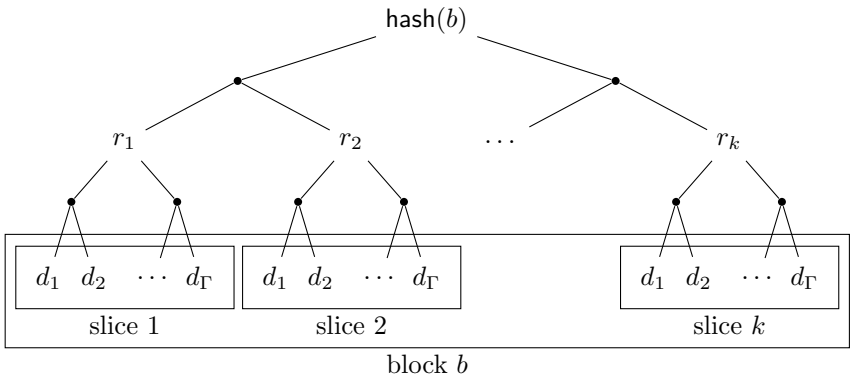

Figure 2: Hierarchy of block data, visualizing the double-Merkle construction of the block hash. Each slice has a Merkle root hash r_i, which are in turn the leaf nodes for the second Merkle tree, where the root corresponds to the block hash.

Definition 1 (shred)**.** *A* shred *fits neatly in a UDP datagram. It has the form:*

$$(s, t, i, z_t, r_t, (d_i, \pi_i), \sigma_t),$$

where

- $s, t, i \in \mathbb{N}$ *are slot number, slice index, shred index, respectively,*
- $z_t \in \{0, 1\}$ *is a flag (see Definition 2 below),*
- d_i *is the data at position* i *with path* π_i *for Merkle root* r_t *(Section 1.6),*
- σ_t *is the signature of the object* $\mathsf{Slice}(s, t, z_t, r_t)$ *from the node* $\mathsf{leader}(s)$*.*

Definition 2 (slice)**.** *A* slice *is the input of Rotor, see Section 2.2. Given any* γ *of the* Γ *shreds, we can decode (Section 1.6) the slice. A slice has the form:*

$$\big(s, t, z_t, r_t, M_t, \sigma_t\big),$$

where

- $s, t \in \mathbb{N}$ *are the slot number and slice index respectively,*
- $z_t \in \{0, 1\}$ *is a flag indicating the last slice index,*
- M_t *is the decoding of the shred data* $\{(d_i)\}_{i \in \mathcal{I}}$ *for Merkle root* r_t*,*
- σ_t *is the signature of the object* $\mathsf{Slice}(s, t, z_t, r_t)$ *from the node* $\mathsf{leader}(s)$*.*

Protocol

Definition 3 (block)**.** *A* block b *is the sequence of all slices of a slot, for the purpose of voting and reaching consensus. A block is of the form:*

$$b = \{\big(s, t, z_t, r_t, M_t, \sigma_t\big)\}_{t \in \{1,\ldots,k\}},$$

where $z_k = 1$, $z_t = 0$ *for* $t < k$*. The data of the block is the concatenation of all the slice data, i.e.,* $\mathcal{M} = (M_1, M_2, \ldots, M_k)$*. We define* $\mathsf{slot}(b) = s$*. The block data* $\mathcal{M}$ *contains information about the slot* $\mathsf{slot}(\mathsf{parent}(b))$ *and hash* $\mathsf{hash}(\mathsf{parent}(b))$ *of the parent block of* b*. There are various limits on a block, for instance, each block can only have a bounded amount of bytes and a bounded amount of time for execution.*

Definition 4 (block hash)**.** *We define* $\mathsf{hash}(b)$ *of block* $b = \{\big(s, t, z_t, r_t, M_t, \sigma_t\big)\}_{t \in \{1,\ldots,k\}}$ *as the root of a Merkle tree* T *where:*

- T *is a complete, full binary tree with the smallest possible number of leaves* m *(with* m *being a power of* 2*) such that* $m \geq k$*,*
- *the first* k *leaves of* T *are* $r_1, \ldots, r_k$ *(each hash is concatenated with a label that marks the hash as a leaf),*
- *the remaining leaves of* T *are* $\perp$*.*

Definition 5 (ancestor and descendant)**.** *An ancestor of a block* b *is any block that can be reached from* b *by the parent links, i.e.,* b*,* b*'s parent,* b*'s parent's parent, and so on. If* b' *is an ancestor of* b*,* b *is a descendant of* b'*. Note that* b *is its own ancestor and descendant.*

2.2 Rotor

Intuition

Rotor is the block dissemination protocol of Alpenglow. The leader (sender) wants to broadcast some data (a block) to all other nodes. This procedure should have low latency, utilize the bandwidth of the network in a balanced way, and be resilient to transmission failures. The block should be produced and transmitted in a streaming manner, that is, the leader does not need to wait until the entire block is constructed.

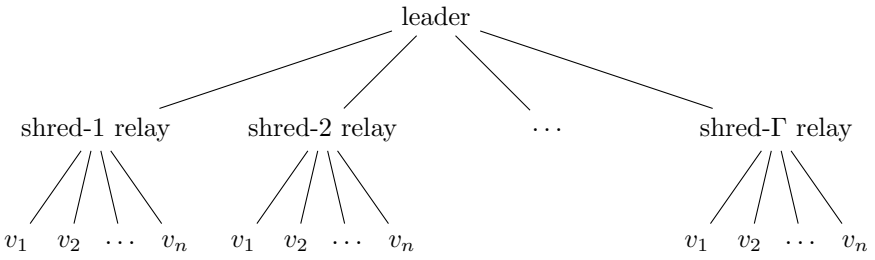

Figure 3: Basic structure of the Rotor data dissemination protocol.

Protocol

A leader uses multiple rounds of the Rotor protocol to broadcast a block. Each round considers the independent transmission of one slice of the block. The leader transmits each slice as soon as it is ready. This achieves pipelining of block production and transmission.

For each slice, the leader generates Γ Reed-Solomon coding shreds and constructs a Merkle tree over their hashes and signs the root. Each coding shred includes the Merkle path along with the root signature. Each shred contains as much data and corresponding metadata as can fit into a single UDP datagram.

Using Reed-Solomon erasure coding [RS60] ensures that, at the cost of sending more data, receiving any γ shreds is enough to reconstruct the slice (Section 1.6). After that, as an additional validity check, a receiver generates the (up to $\Gamma - \gamma$) missing shreds.

For any given slice, the leader sends each shred directly to a corresponding node selected as shred relay. We sample relays for every slice. We use a novel sampling method which improves resilience. We describe our new method in detail in Section 3.1.

Each relay then broadcasts its shred to all nodes that still need it, i.e., all nodes except for the leader and itself, in decreasing stake order. As a minor optimization, all shred relays send their shred to the next leader first. This slightly improves latency for the next leader, who most urgently needs the block.

A shred's authenticity needs to be checked to reconstruct the slice from γ of the shreds. To enable receivers to cheaply check authenticity of each shred individually, the leader builds a Merkle tree [Mer79] over all shreds of a slice, as described in Section 1.6. Each shred then includes its path in the tree and the leader's signature of the root of the tree.

When receiving the first shred of a slice, a node checks the validity of the Merkle path and the leader's signature, and then stores the verified root. For any later shred, the receiving node only checks the validity of the Merkle path against the stored root.

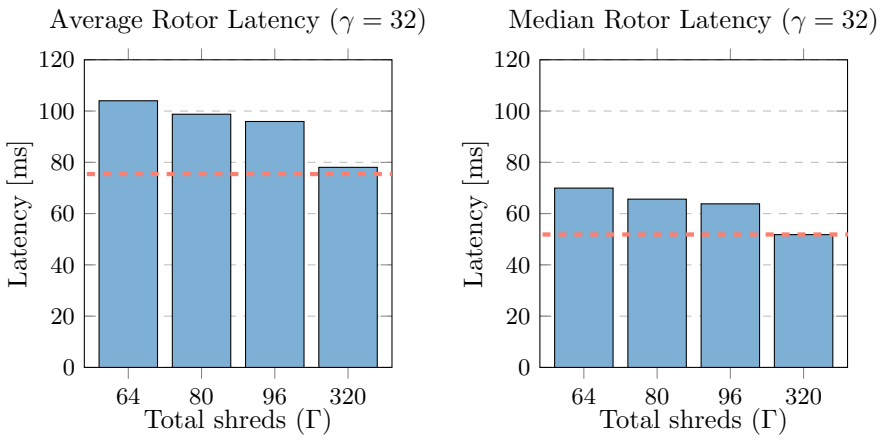

Figure 4: Rotor latency for different data expansion ratios (and thus total numbers of shreds), all with $\gamma = 32$ data shreds using our sampling from Section 3.1. The red lines indicate the average/median network latency. With a high data expansion rate ($\kappa = 10$, hence $\Gamma = 320$) we pretty much achieve the single δ latency described in Lemma 8. All our simulation results use the current (epoch 780) Solana stake distribution. Network latency is inferred from public data. Computation and transmission delays are omitted.

Definition 6. *Given a slot s, we say that Rotor is* successful *if the leader of s is correct, and at least γ of the corresponding relays are correct.*

Resilience. If the conditions of Definition 6 are met, all correct nodes will receive the block distributed by the leader, as enough relays are correct. On the other hand, a faulty leader can simply not send any data, and Rotor will immediately fail. In the following we assume that the leader is correct. The following lemma shows that Rotor is likely to succeed if we over-provision the coding shreds by at least 67%.

Lemma 7 (rotor resilience)**.** *Assume that the leader is correct, and that erasure coding over-provisioning is at least $\kappa = \Gamma/\gamma > 5/3$. If $\gamma \to \infty$, with probability 1, a slice is received correctly.*

Proof Sketch. We choose the relay nodes randomly, according to stake. The failure probability of each relay is less than 40% according to Section 1.2. The expected value of correct relays is then at least $60\% \cdot \Gamma > 60\% \cdot 5\gamma/3 = \gamma$. So strictly more than γ shreds will arrive in expectation. With $\gamma \to \infty$, applying an appropriate Chernoff bound, with probability 1 we will have at least γ shreds that correctly arrive at all nodes. □

Latency. The latency of Rotor is between δ and 2δ, depending on whether we make optimistic or pessimistic assumptions on various parameters.

Lemma 8. *(rotor latency) If Rotor succeeds, network latency of Rotor is at most 2δ. A high over-provisioning factor κ can reduce latency. In the extreme case with $n \to \infty$ and $\kappa \to \infty$, we can bring network latency down to δ. (See also Figure 4 for simulation results with Solana's stake distribution.)*

Proof Sketch. Assuming a correct leader, all relays receive their shred in time δ directly from the leader. The correct relays then send their shred to the nodes in another time δ, so in time 2δ in total.

If we over-provision the relays, chances are that many correct relays are geographically located between leader and the receiving node. In the extreme case with infinitely many relays, and some natural stake distribution assumptions, there will be at least γ correct relays between any pair of leader and receiving node. If the relays are on the direct path between leader and receiver, they do not add any overhead, and both legs of the trip just sum up to δ. □

Bandwidth. Both the leader and the shred relays are sampled by stake. As a result, in expectation each node has to transmit data proportional to their stake. This aligns well with the fact that staking rewards are also proportional to the nodes' stake. If the available out-bandwidth is proportional to stake, it can be utilized perfectly apart from the overhead.

Lemma 9 (bandwidth optimality)**.** *Assume a fixed leader sending data at rate $\beta_\ell \leq \bar{\beta}$, where $\bar{\beta}$ is the average outgoing bandwidth across all nodes. Suppose any distribution of out-bandwidth and proportional node stake. Then, at every correct node, Rotor delivers block data at rate β_ℓ/κ in expectation. Up to the data expansion rate $\kappa = \Gamma/\gamma$, this is optimal.*

Proof. Node v_i is chosen to be a shred relay in expectation $\Gamma\rho_i$ times. Each shred relay receives data from the leader with bandwidth β_ℓ/Γ, because the leader splits its bandwidth across all shred relays. Hence, in expectation, node v_i receives data from the leader at rate $\Gamma\rho_i \cdot \beta_\ell/\Gamma = \rho_i\beta_\ell$. Node v_i needs to forward this data to $n-2$ nodes. So, in expectation, node v_i needs to send data at rate $\rho_i\beta_\ell(n-2)$. Node v_i has outgoing bandwidth $\beta_i = n\bar{\beta}\rho_i$, since outgoing bandwidth is proportional to stake (Section 1.5). Since $\beta_\ell \leq \bar{\beta}$, we have $\rho_i\beta_\ell(n-2) < \beta_i$. Each node thus has enough outgoing bandwidth to support the data they need to send.

Note that we cannot get above rate β_ℓ because the leader is the only one who knows the data. Likewise we cannot get above rate $\bar{\beta}$, because all nodes need to receive the data, and the nodes can send with no more total rate than $n\bar{\beta}$. So apart from the data expansion factor κ, we are optimal. □

Note that any potential attacks on Rotor may only impact liveness, not

Analysis

safety, since the other parts of Alpenglow ensure safety even under asynchrony and rely on Rotor only for data dissemination.

2.3 Blokstor

Protocol

Blokstor collects and stores the first block received through Rotor in every slot, as described in Definition 10.

Definition 10 (Blokstor)**.** *The* Blokstor *is a data structure managing the storage of slices disseminated by the protocol of Section 2.2. When a shred* $(s, t, i, z_t, r_t, (d_i, \pi_i), \sigma_t)$ *is received by a node, the node checks the following conditions. If the conditions are satisfied, the shred is added to the Blokstor:*

- *the Blokstor does not contain a shred for indices* (s, t, i) *yet,*
- (d_i, π_i) *is the data with path for Merkle root* r_t *at position* i*,*
- σ_t *is the signature of the object* $\mathsf{Slice}(s, t, z_t, r_t)$ *from the node* $\mathsf{leader}(s)$*.*

Blokstor emits the event $\mathsf{Block}(\mathsf{slot}(b), \mathsf{hash}(b), \mathsf{hash}(\mathsf{parent}(b)))$ *as input for Algorithm 1 when it receives the first complete block* b *for* $\mathsf{slot}(b)$*.*

In addition to storing the first block received for a given slot, the Blokstor can perform the repair procedure (Section 2.8) to collect some other block b and store it in the Blokstor. If a block is finalized according to Definition 14, Blokstor has to collect and store only this block in the given slot. Otherwise, before the event $\mathsf{SafeToNotar}(\mathsf{slot}(b), \mathsf{hash}(b))$ of Definition 16 is emitted, b has to be stored in the Blokstor as well.

2.4 Votes and Certificates

Intiuition

Next we describe the voting data structures and algorithms of Alpenglow. In a nutshell, if a leader gets at least 80% of the stake to vote for its block, the block is immediately finalized after one round of voting with a fast-finalization certificate. However, as soon as a node observes 60% of stake voting for a block, it issues its second-round vote. After 60% of stake voted for a block the second time, the block is also finalized. On the other hand, if enough stake considers the block late, a skip certificate can be produced, and the block proposal will be skipped.

Protocol

Definition 11 (messages)**.** *Alpenglow uses voting and certificate messages listed in Tables 5 and 6.*

Vote Type	Object
Notarization Vote	NotarVote(slot(b), hash(b))
Notar-Fallback Vote	NotarFallbackVote(slot(b), hash(b))
Skip Vote	SkipVote(s)
Skip-Fallback Vote	SkipFallbackVote(s)
Finalization Vote	FinalVote(s)

Table 5: Alpenglow's voting messages with respect to block b and slot s. Each object is signed by a signature σ_v of the voting node v.

Certificate Type	Aggregated Votes	Condition
Fast-Finalization Cert.	NotarVote	$\Sigma \geq 80\%$
Notarization Cert.	NotarVote	$\Sigma \geq 60\%$
Notar-Fallback Cert.	NotarVote or NotarFallbackVote	$\Sigma \geq 60\%$
Skip Cert.	SkipVote or SkipFallbackVote	$\Sigma \geq 60\%$
Finalization Cert.	FinalVote	$\Sigma \geq 60\%$

Table 6: Alpenglow's certificate messages. Σ is the cumulative stake of the aggregated votes $(\sigma_i)_{I \subseteq \{1,\dots,n\}}$ in the certificate, i.e., $\Sigma = \sum_{i \in I} \rho_i$.

Protocol

2.5 Pool

Every node maintains a data structure called *Pool*. In its Pool, each node memorizes all votes and certificates for every slot.

Definition 12 (storing votes)**.** *Pool stores received votes for every slot and every node as follows:*

- *The first received notarization or skip vote,*
- *up to 3 received notar-fallback votes,*
- *the first received skip-fallback vote, and*
- *the first received finalization vote.*

Definition 13 (certificates)**.** *Pool generates, stores and broadcasts certificates:*

- *When enough votes (see Table 6) are received, the respective certificate is generated.*
- *When a received or constructed certificate is newly added to Pool, the certificate is broadcast to all other nodes.*

- *A single (received or constructed) certificate of each type corresponding to the given block/slot is stored in Pool.*

Note that the conditions in Table 6 imply that if a correct node generated the Fast-Finalization Certificate, it also generated the Notarization Certificate, which in turn implies it generated the Notar-Fallback Certificate.

Definition 14 (finalization)**.** *We have two ways to finalize a block:*

- *If a finalization certificate on slot s is in Pool, the unique notarized block in slot s is finalized (we call this slow-finalized).*
- *If a fast-finalization certificate on block b is in Pool, the block b is finalized (fast-finalized).*

Whenever a block is finalized (slow or fast), all ancestors of the block are finalized first.

Definition 15 (Pool events)**.** *The following events are emitted as input for Algorithm 1:*

- $\mathsf{BlockNotarized}(\mathsf{slot}(b), \mathsf{hash}(b))$*: Pool holds a notarization certificate for block b.*
- $\mathsf{ParentReady}(s, \mathsf{hash}(b))$*: Slot s is the first of its leader window, and Pool holds a notarization or notar-fallback certificate for a previous block b, and skip certificates for every slot s' since b, i.e., for $\mathsf{slot}(b) < s' < s$.*

Protocol

As we will see later (Lemmas 20 and 35), for every slot s, every correct node will cast exactly one notarization or skip vote. After casting this initial vote, the node might emit events according to Definition 16 and cast additional votes.

The event $\mathsf{SafeToNotar}(s, b)$ indicates that it is not possible that some block $b' \neq b$ could be fast-finalized (Definition 14) in slot s, and so it is safe to issue the notar-fallback vote for b.

Similarly, $\mathsf{SafeToSkip}(s)$ indicates that it is not possible that any block in slot s could be fast-finalized (Definition 14), and so it is safe to issue the skip-fallback vote for s.

Intiuition

Definition 16 (fallback events)**.** *Consider block b in slot $s = \mathsf{slot}(b)$. By $\mathsf{notar}(b)$ denote the cumulative stake of nodes whose notarization votes for block b are in Pool, and by $\mathsf{skip}(s)$ denote the cumulative stake of nodes whose skip votes for slot s are in Pool. Recall that by Definition 12 the stake of any node can be counted only once per slot. The following events are emitted as input for Algorithm 1:*

- $\mathsf{SafeToNotar}(s, \mathsf{hash}(b))$*: The event is only issued if the node voted in*

slot s already, but not to notarize b. Moreover:

$$\mathit{notar}(b) \geq 40\% \ \textbf{\textit{or}} \ \Big(\mathit{skip}(s) + \mathit{notar}(b) \geq 60\% \ \textbf{\textit{and}} \ \mathit{notar}(b) \geq 20\%\Big).$$

If s is the first slot in the leader window, the event is emitted. Otherwise, block b is retrieved in the repair procedure (Section 2.8) first, in order to identify the parent of the block. Then, the event is emitted when Pool contains the notar-fallback certificate for the parent as well.

- $\mathit{SafeToSkip}(s)$*: The event is only issued if the node voted in slot s already, but not to skip s. Moreover:*

$$\mathit{skip}(s) + \sum_b \mathit{notar}(b) - \max_b \mathit{notar}(b) \geq 40\%.$$

Protocol

2.6 Votor

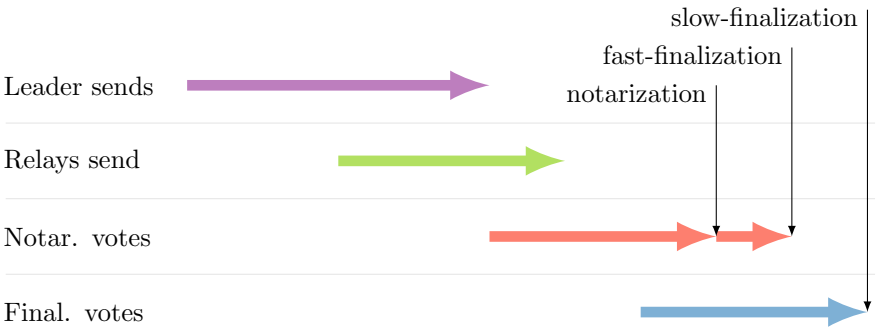

Figure 7: Protocol overview: a full common case life cycle of a block in Alpenglow.

The purpose of voting is to notarize and finalize blocks. Finalized blocks constitute a single chain of parent references and indicate the output of the protocol.

The protocol ensures that for every slot, either a skip certificate is created, or some block b is notarized (or notarized-fallback), such that all ancestors of b are also notarized. Condition thresholds ensure that a malicious leader cannot prevent the creation of certificates needed for liveness. If many correct nodes produced notarization votes for the same block b, then all other correct nodes will make notar-fallback votes for b. Otherwise, all correct nodes will broadcast skip-fallback votes.

By Definition 14, a node can finalize a block as soon as it observes enough notarization votes produced by other nodes immediately upon receiving a block. However, a lower participation threshold is required to make a nota-

Intuition

rization certificate. Then the node will send the finalization vote. Therefore, blocks are finalized after one round of voting among nodes with 80% of the stake, or two rounds of voting among nodes with 60% of the stake.

Nodes have local clocks and emit timeout events. Whenever a node v's Pool emits the event $\mathsf{ParentReady}(s, \dots)$, it starts timeout timers corresponding to all blocks of the leader window beginning with slot s. The timeouts are parametrized with two delays (pertaining to network synchrony):

- Δ_{block}: This denotes the protocol-specified block time.
- $\Delta_{\mathsf{timeout}}$: Denotes the rest of the possible delay (other than Δ_{block}) between setting the timeouts and receiving a correctly disseminated block. As a conservative global constant, $\Delta_{\mathsf{timeout}}$ can be set to $(1\Delta + 2\Delta) >$ (time needed for the leader to observe the certificates) + (latency of slice dissemination through Rotor).

Definition 17 (timeout)**.** *When a node v's Pool emits the first event $\mathsf{ParentReady}(s, \dots)$, $\mathsf{Timeout}(i)$ events for the leader window beginning with s (for all $i \in \text{WINDOWSLOTS}(s)$) are scheduled at the following times:*

$$\mathsf{Timeout}(i) : \mathsf{clock}() + \Delta_{\mathsf{timeout}} + (i - s + 1) \cdot \Delta_{\mathsf{block}}.$$

The timeouts are set to correspond to the latest possible time of receiving a block if the leader is correct and the network is synchronous. Timeouts can be optimized, e.g., by fine-grained Δ estimation or to address specific faults, such as crash faults.

Note that $\mathsf{ParentReady}(s, \dots)$ is only emitted for the first slot s of a window. Therefore, $(i - s + 1) \geq 1$ and $\mathsf{Timeout}(i)$ is never scheduled to be emitted in the past.

Definition 18 (Votor state)**.** *Votor (Algorithms 1 and 2) accesses state associated with each slot. The state of every slot is initialized to the empty set: $\mathsf{state} \leftarrow [\emptyset, \emptyset, \dots]$. The following objects can be permanently added to the state of any slot s:*

- *$\mathsf{ParentReady}(\mathsf{hash}(b))$: Pool emitted the event $\mathsf{ParentReady}(s, \mathsf{hash}(b))$.*
- *Voted: The node has cast either a notarization vote or a skip vote in slot s.*
- *$\mathsf{VotedNotar}(\mathsf{hash}(b))$: The node has cast a notarization vote on block b in slot s.*
- *$\mathsf{BlockNotarized}(\mathsf{hash}(b))$: Pool holds the notarization certificate for block b in slot s.*
- *$\mathsf{ItsOver}$: The node has cast the finalization vote in slot s, and will not cast any more votes in slot s.*

Protocol

- *BadWindow: The node has cast at least one of these votes in slot s: skip, skip-fallback, notar-fallback.*

Additionally, every slot can be associated with a pending block, which is initialized to bottom: pendingBlocks $\leftarrow [\bot, \bot, \ldots]$. *The* pendingBlocks *are blocks which will be revisited to call* tryNotar(), *as the tested condition might be met later.*

Algorithm 1 Votor, event loop, single-threaded

```
1:  upon Block(s, hash, hash_parent) do
2:      if tryNotar(Block(s, hash, hash_parent)) then
3:          checkPendingBlocks()
4:      else if Voted ∉ state[s] then
5:          pendingBlocks[s] ← Block(s, hash, hash_parent)

6:  upon Timeout(s) do
7:      if Voted ∉ state[s] then
8:          trySkipWindow(s)

9:  upon BlockNotarized(s, hash(b)) do
10:     state[s] ← state[s] ∪ {BlockNotarized(hash(b))}
11:     tryFinal(s, hash(b))

12: upon ParentReady(s, hash(b)) do
13:     state[s] ← state[s] ∪ {ParentReady(hash(b))}
14:     checkPendingBlocks()
15:     setTimeouts(s)                ▷ start timer for all slots in this window

16: upon SafeToNotar(s, hash(b)) do
17:     trySkipWindow(s)
18:     if ItsOver ∉ state[s] then
19:         broadcast NotarFallbackVote(s, hash(b))      ▷ notar-fallback vote
20:         state[s] ← state[s] ∪ {BadWindow}

21: upon SafeToSkip(s) do
22:     trySkipWindow(s)
23:     if ItsOver ∉ state[s] then
24:         broadcast SkipFallbackVote(s)                ▷ skip-fallback vote
25:         state[s] ← state[s] ∪ {BadWindow}
```

Algorithm 2 Votor, helper functions

```
1: function windowSlots(s)
2:     return array with slot numbers of the leader window with slot s

3: function setTimeouts(s)                                   ▷ s is first slot of window
4:     for i ∈ windowSlots(s) do                             ▷ set timeouts for all slots
5:         schedule event Timeout(i) at time clock()+Δ_timeout+(i−s+1)·Δ_block

6: ▷ Check if a notarization vote can be cast. ◁
7: function tryNotar(Block(s, hash, hash_parent))
8:     if Voted ∈ state[s] then
9:         return false
10:    firstSlot ← (s is the first slot in leader window)                    ▷ boolean
11:    if (firstSlot and ParentReady(hash_parent) ∈ state[s]
       or (not firstSlot and VotedNotar(hash_parent) ∈ state[s − 1]) then
12:        broadcast NotarVote(s, hash)                                  ▷ notarization vote
13:        state[s] ← state[s] ∪ {Voted, VotedNotar(hash)}
14:        pendingBlocks[s] ← ⊥                          ▷ won't vote notar a second time
15:        tryFinal(s, hash)                                  ▷ maybe vote finalize as well
16:        return true
17:    return false

18: function tryFinal(s, hash(b))
19:    if BlockNotarized(hash(b)) ∈ state[s] and VotedNotar(hash(b)) ∈ state[s]
       and BadWindow ∉ state[s] then
20:        broadcast FinalVote(s)                                        ▷ finalization vote
21:        state[s] ← state[s] ∪ {ItsOver}

22: function trySkipWindow(s)
23:    for k ∈ windowSlots(s) do                                 ▷ skip unvoted slots
24:        if Voted ∉ state[k] then
25:            broadcast SkipVote(k)                                         ▷ skip vote
26:            state[k] ← state[k] ∪ {Voted, BadWindow}
27:            pendingBlocks[k] ← ⊥                      ▷ won't vote notar after skip

28: function checkPendingBlocks()
29:    for s : pendingBlocks[s] ≠ ⊥ do                        ▷ iterate with increasing s
30:        tryNotar(pendingBlocks[s])
```

2.7 Block Creation

The leader v of the window beginning with slot s produces blocks for all slots WINDOWSLOTS(s) in the window. After the event ParentReady(s, hash(b_p)) is emitted, v can be sure that a block b in slot s with b_p as its parent will be valid. In other words, other nodes will receive the certificates that resulted in v emitting ParentReady(hash(b_p)), and emit this event themselves. As a result, all correct nodes will vote for b.

In the common case, only one ParentReady(s, hash(b_p)) will be emitted for a given s. Then, v has to build its block on top of b_p and cannot "fork off" the chain in any way. If v emits many ParentReady(s, hash(b_p)) events for different blocks b_p (as a result of the previous leader misbehaving or network delays), v can build its block with any such b_p as its parent.

Algorithm 3 introduces an optimization where v starts building its block "optimistically" before any ParentReady(s, hash(b_p)) is emitted. Usually v will receive some block b_p in slot $s-1$ first, then observe a certificate for b_p after additional network delay, and only then emit ParentReady(s, hash(b_p)). Algorithm 3 avoids this delay in the common case. If v started building a block with parent b_p, but then only emits ParentReady(s, hash(b'_p)) where $b'_p \neq b_p$, v will then instead indicate b'_p as the parent of the block in the content of some slice t. In this case, slices $1, \ldots, t-1$ are ignored for the purpose of execution.

We allow changing the indicated parent of a block only once, and only in blocks in the first slot of a given window.

When a leader already observed some ParentReady($s, \ldots$), the leader produces all blocks of its leader window without delays. As a result, the first block b_0 always builds on some parent b_p such that v emitted ParentReady(s, hash(b_p)), b_0 is the parent of the block b_1 in slot $s+1$, b_1 is the parent of the block b_2 in slot $s+2$, and so on.

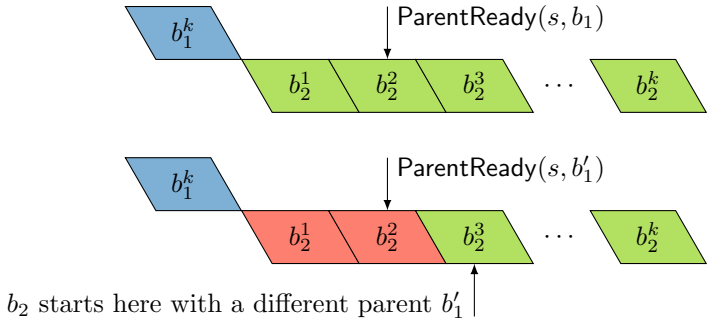

Figure 8: Handover between leader windows with k slices per block. The new leader starts to produce the first slice of its first block (b_2^1) as soon as it received the last slice (b_1^k) of the previous leader. The common case is on top and the case where leader switches parents at the bottom, see also Algorithm 3.

Protocol

Algorithm 3 Block creation for leader window starting with slot s

wait until block b_p in slot $s-1$ received **or** $\mathsf{ParentReady}(\mathsf{hash}(b_p)) \in \mathsf{state}[s]$
$b \leftarrow$ generate a block with parent b_p in slot s ▷ *block being produced*
$t \leftarrow 1$ ▷ *slice index*
while $\mathsf{ParentReady}(\dots) \notin \mathsf{state}[s]$ **do** ▷ *produce slices optimistically*
    $\mathsf{Rotor}(\text{slice } t \text{ of } b)$
    $t \leftarrow t + 1$
if $\mathsf{ParentReady}(\mathsf{hash}(b_p)) \notin \mathsf{state}[s]$ **then** ▷ *change parent, reset block*
    $b_p \leftarrow$ any b' such that $\mathsf{ParentReady}(\mathsf{hash}(b')) \in \mathsf{state}[s]$
    $b \leftarrow$ generate a block with parent b_p in slot s starting with slice index t
$\mathsf{start} \leftarrow \mathsf{clock}()$ ▷ *some parent is ready, set timeout*
while $\mathsf{clock}() < \mathsf{start} + \Delta_{\mathsf{block}}$ **do** ▷ *produce rest of block in normal slot time*
    $\mathsf{Rotor}(\text{slice } t \text{ of } b)$
    $t \leftarrow t + 1$
for remaining slots of the window $s' = s+1, s+2, \dots$ **do**
    $b \leftarrow$ generate a block with parent b in slot s'
    $\mathsf{Rotor}(b)$ over Δ_{block}

2.8 Repair

Repair is the process of retrieving a block with a given hash that is missing from Blokstor. After Pool obtains a certificate of signatures on $\mathsf{Notar}(\mathsf{slot}(b), \mathsf{hash}(b))$ or $\mathsf{NotarFallback}(\mathsf{slot}(b), \mathsf{hash}(b))$, the block b with hash $\mathsf{hash}(b)$ according to Definition 4 needs to be retrieved.

Definition 19 (repair functions)**.** *The protocol supports functions for the repair process:*

- *sampleNode*(): *Choose some node v at random based on stake.*
- *getSliceCount*($\mathsf{hash}(b), v$)*: Contact node v, which returns (k, r_k, π_k) where:*
  - *k is the number of slices of b as in Definition 4,*
  - *r_k is the hash at position k with path π_k for Merkle root $\mathsf{hash}(b)$.*

  The requesting node needs to make sure r_k is the last non-zero leaf of the Merkle tree with root $\mathsf{hash}(b)$. It verifies that the rightward intermediate hashes in π_k correspond to empty sub-trees.
- *getSliceHash*($t, \mathsf{hash}(b), v$)*: Contact node v, which returns (r_t, π_t) where r_t is the hash at position t with path π_t for Merkle root $\mathsf{hash}(b)$.*
- *getShred*(s, t, i, r_t, v)*: Contact node v, which returns the shred $(s, t, i, z_t, r_t, (d_i, \pi_i), \sigma_t)$ as in Definition 1.*

Protocol

The functions can fail verification of the data provided by v and return $\perp$ (e.g. if invalid data is returned or v simply does not have the correct data to return).

Algorithm 4 Repair block b with $\mathsf{hash}(b)$ in slot s

```
1: k ← ⊥
2: while k = ⊥ do                                   ▷ find the number of slices k in b
3:     (k, r_k, π_k) ← getSliceCount(hash(b), sampleNode())
4: for t = 1, ..., k concurrently do
5:     while r_t = ⊥ do                             ▷ get slice hash r_t if missing
6:         (r_t, π_t) ← getSliceHash(t, hash(b), sampleNode())
7:     for each shred index i concurrently do
8:         while shred with indices s, t, i missing do   ▷ get shred if missing
9:             shred ← getShred(s, t, i, r_t, sampleNode())
10:            store shred if valid
```

2.9 Safety

In the following analysis, whenever we say that a certificate exists, we mean that a correct node observed the certificate. Whenever we say that an ancestor b' of a block b exists in some slot $s = \mathsf{slot}(b')$, we mean that starting at block b and following the parent links in blocks with the given hash we reach block b' in slot $s = \mathsf{slot}(b')$.

Lemma 20 (notarization or skip)**.** *A correct node exclusively casts only one notarization vote or skip vote per slot.*

Proof. Notarization votes and skip votes are only cast via functions tryNotar() and trySkipWindow() of Algorithm 2, respectively. Votes are only cast if $\mathsf{Voted} \notin \mathsf{state}[s]$. After voting, the state is modified so that $\mathsf{Voted} \in \mathsf{state}[s]$. Therefore, a notarization or skip vote can only be cast once per slot by a correct node. □

Lemma 21 (fast-finalization property)**.** *If a block b is fast-finalized:*

(i) no other block b' in the same slot can be notarized,

(ii) no other block b' in the same slot can be notarized-fallback,

(iii) there cannot exist a skip certificate for the same slot.

Proof. Suppose some correct node fast-finalized some block b in slot s. By Definition 14, nodes holding at least 80% of stake cast notarization votes for b. Recall (Assumption 1) that all byzantine nodes hold less than 20% of stake. Therefore, a set V of correct nodes holding more than 60% of stake cast notarization votes for b.

(i) By Lemma 20, nodes in V cannot cast a skip vote or a notarization vote for a different block $b' \neq b$. Therefore, the collective stake of nodes casting a notarization vote for b' has to be smaller than 40%.

(ii) Correct nodes only cast notar-fallback votes in Algorithm 1 when Pool emits the event SafeToNotar. By Definition 16, a correct node emits $\mathsf{SafeToNotar}(s, \mathsf{hash}(b'))$, if either a) at least 40% of stake holders voted to notarize b', or b) at least 60% of stake holders voted to notarize b' or skip slot s. Only nodes $v \notin V$ holding less than 40% of stake can vote to notarize b' or skip slot s. Therefore, no correct nodes can vote to notar-fallback b'.

(iii) Skip-fallback votes are only cast in Algorithm 1 by correct nodes if Pool emits the event SafeToSkip. By Definition 16, a correct node can emit SafeToSkip if at least 40% of stake have cast a skip vote or a notarization vote on $b' \neq b$ in slot s. Only nodes $v \notin V$ holding less than 40% of stake can cast a skip vote or a notarization vote on $b' \neq b$ in slot s. Therefore, no correct nodes vote to skip-fallback, and no nodes in V vote to skip or skip-fallback slot s. □

Lemma 22. *If a correct node v cast a finalization vote in slot s, then v did not cast a notar-fallback or skip-fallback vote in s.*

Proof. A correct node adds ItsOver to its state of slot s in line 21 of Algorithm 2 when casting a finalization vote. Notar-fallback or skip-fallback votes can only be cast if $\mathsf{ItsOver} \notin \mathsf{state}[s]$ in lines 18 and 23 of Algorithm 1 respectively. Therefore, notar-fallback and skip-fallback votes cannot be cast by v in slot s after casting a finalization vote in slot s.

On the other hand, a correct node adds BadWindow to its state of slot s when casting a notar-fallback or skip-fallback vote in slot s. A finalization vote can only be cast if $\mathsf{BadWindow} \notin \mathsf{state}[s]$ in line 19 of Algorithm 2. Therefore, a finalization vote cannot be cast by v in slot s after casting a notar-fallback and skip-fallback vote in slot s. □

Lemma 23. *If correct nodes with more than 40% of stake cast notarization votes for block b in slot s, no other block can be notarized in slot s.*

Proof. Let V be the set of correct nodes that cast notarization votes for b. Suppose for contradiction some $b' \neq b$ in slot s is notarized. Since 60% of stake holders had to cast notarization votes for b' (Definition 11), there is a node $v \in V$ that cast notarization votes for both b and b', contradicting Lemma 20. □

Lemma 24. *At most one block can be notarized in a given slot.*

Proof. Suppose a block b is notarized. Since 60% of stake holders had to cast notarization votes for b (Definition 11) and we assume all byzantine nodes hold less than 20% of stake, then correct nodes with more than 40% of stake cast notarization votes for b. By Lemma 23, no block $b' \neq b$ in the same slot can be notarized. □

Lemma 25. *If a block is finalized by a correct node, the block is also notarized.*

Proof. If b was fast-finalized by some correct node, nodes with at least 80% of the stake cast their notarization votes for b. Since byzantine nodes possess less than 20% of stake, correct nodes with more than 60% of stake broadcast their notarization votes, and correct nodes will observe a notarization certificate for b.

If b was slow-finalized by some correct node, nodes with at least 60% of stake cast their finalization vote for b (Def. 11 and 14), including some correct nodes. Correct nodes cast finalization votes only if $\mathsf{BlockNotarized}(\mathsf{hash}(b)) \in \mathsf{state}[s]$ in line 19 of Algorithm 2 after they observe some notarization certificate. By Lemma 24, this notarization certificate has to be for b. □

Lemma 26 (slow-finalization property)**.** *If a block b is slow-finalized:*

(i) no other block b' in the same slot can be notarized,

(ii) no other block b' in the same slot can be notarized-fallback,

(iii) there cannot exist a skip certificate for the same slot.

Proof. Suppose some correct node slow-finalized some block b in slot s. By Definition 14, nodes holding at least 60% of stake cast finalization votes in slot s. Recall that we assume all byzantine nodes to hold less than 20% of stake. Therefore, a set V of correct nodes holding more than 40% of stake cast finalization votes in slot s. By condition in line 19 of Algorithm 2, nodes in V observed a notarization certificate for some block. By Lemma 24, all nodes in V observed a notarization certificate for the same block b, and because of the condition in line 19, all nodes in V previously cast a notarization vote for b. By Lemmas 20 and 22, all nodes in V cast no votes in slot s other than the notarization vote for b and the finalization vote. Since nodes in V hold more than 40% of stake, and every certificate requires at least 60% of stake holder votes, no skip certificate or certificate on another block $b' \neq b$ in slot s can be produced. □

Lemma 27. *If there exists a notarization or notar-fallback certificate for block b, then some correct node cast its notarization vote for b.*

Proof. Suppose for contradiction no correct node cast its notarization vote for b. Since byzantine nodes possess less than 20% of stake, every correct node observed less than 20% of stake voting to notarize b. Both sub-conditions for emitting the event $\mathsf{SafeToNotar}(s, \mathsf{hash}(b))$ by Definition 16 require observing 20% of stake voting to notarize b. Therefore, no correct node emitted $\mathsf{SafeToNotar}(s, \mathsf{hash}(b))$. In Algorithm 1, emitting $\mathsf{SafeToNotar}(s, \mathsf{hash}(b))$ is the only trigger that might lead to casting a notar-fallback vote for b. Therefore, no correct node cast a notar-fallback vote for b. However, at least 60%

of stake has to cast a notarization or notar-fallback vote for b for a certificate to exist (Definition 11), leading to a contradiction. □

Lemma 28. *If a correct node v cast the notarization vote for block b in slot $s = \mathsf{slot}(b)$, then for every slot $s' \leq s$ such that $s' \in \text{WINDOWSLOTS}(s)$, v cast the notarization vote for the ancestor b' of b in slot $s' = \mathsf{slot}(b')$.*

Proof. If s is the first slot of the leader window, there are no slots $s' < s$ in the same window. Since v voted for b in s we are done. Suppose s is not the first slot of the window.

Due to the condition in line 11 of Algorithm 2, v had to evaluate the latter leg of the condition (namely (**not** firstSlot **and** VotedNotar($\mathsf{hash}_{\mathsf{parent}}$) $\in$ state$[s-1]$)) to true to cast a notarization vote for b. The object VotedNotar(hash) is added to the state of slot $s-1$ only when casting a notarization vote on a block with the given hash in line 13. By induction, v cast notarization votes for ancestors of b in all slots $s' < s$ in the same leader window. □

Lemma 29. *Suppose a correct node v cast a notar-fallback vote for a block b in slot s that is not the first slot of the window, and b' is the parent of b. Then, either some correct node cast a notar-fallback vote for b', or correct nodes with more than 40% of stake cast notarization votes for b'.*

Proof. SafeToNotar conditions (Definition 16) require that v observed a notarization or notar-fallback certificate for b', and so nodes with at least 60% of stake cast notarization or notar-fallback votes for b'. Since byzantine nodes possess less than 20% of stake, either correct nodes with more than 40% of stake cast notarization votes for b', or some correct node cast a notar-fallback vote for b'. □

Lemma 30. *Suppose a block b in slot s is notarized or notarized-fallback. Then, for every slot $s' \leq s$ such that $s' \in \text{WINDOWSLOTS}(s)$, there is an ancestor b' of b in slot s'. Moreover, either correct nodes with more than 40% of stake cast notarization votes for b', or some correct node cast a notar-fallback vote for b'.*

Proof. By Lemma 27, some correct node voted for b. By Lemma 28, for every slot $s' \leq s$ such that $s' \in \text{WINDOWSLOTS}(s)$, there is an ancestor b' of b in slot s'.

Let b' be the parent of b in slot $s-1$. Suppose correct nodes with more than 40% of stake cast notarization votes for b'. Then, the result follows by Lemma 28 applied to each of these nodes.

Otherwise, by Lemma 29, either some correct node cast a notar-fallback vote for b', or correct nodes with more than 40% of stake cast notarization votes for b'. By induction, the result follows for all ancestors of b in the same leader window. □

Lemma 31. *Suppose some correct node finalizes a block b_i and b_k is a block in the same leader window with $\mathit{slot}(b_i) \leq \mathit{slot}(b_k)$. If any correct node observes a notarization or notar-fallback certificate for b_k, b_k is a descendant of b_i.*

Proof. Suppose b_k is not a descendant of b_i. By Lemmas 21 and 26, $\mathsf{slot}(b_i) \neq \mathsf{slot}(b_k)$. Therefore, $\mathsf{slot}(b_i) < \mathsf{slot}(b_k)$ and b_k is not in the first slot of the leader window. By Lemmas 27 and 25, some correct node v cast a notarization vote for b_k. By Lemma 28, there is an ancestor of b_k in every slot $s' < \mathsf{slot}(b_k)$ in the same leader window.

Let b_j be the ancestor of b_k in slot $\mathsf{slot}(b_i) + 1$. b_k is not a descendant of b_i, so the parent b_i' of b_j in the same slot as b_i is different from b_i.

By Lemma 30, either correct nodes with more than 40% of stake cast notarization votes for b_j, or some correct node cast a notar-fallback vote for b_j. If a correct node cast a notar-fallback vote for b_j, by Definition 16, the parent b_i' of b_j in the same slot as b_i is notarized, or notarized-fallback. That would be a contradiction with Lemma 21 or 26. Otherwise, if correct nodes with more than 40% of stake cast notarization votes for b_j, by Lemma 28, these nodes also cast notarization votes for b_i', a contradiction with Lemma 23. □

Lemma 32. *Suppose some correct node finalizes a block b_i and b_k is a block in a different leader window such that $\mathit{slot}(b_i) < \mathit{slot}(b_k)$. If any correct node observes a notarization or notar-fallback certificate for b_k, b_k is a descendant of b_i.*

Proof. Let b_j be the highest ancestor of b_k such that $\mathsf{slot}(b_i) \leq \mathsf{slot}(b_j)$ and b_j is notarized or notarized-fallback. If b_j is in the same leader window as b_i, we are done by Lemma 31; assume b_j is not in the same leader window as b_i. By Lemmas 27 and 28, some correct node v cast a notarization vote for an ancestor b_j' of b_j in the first slot s of the same leader window. Due to the condition in line 11 of Algorithm 2, v had to evaluate the former leg of the condition (namely $\mathsf{firstSlot}$ **and** $\mathsf{ParentReady}(\mathsf{hash}(b)) \in \mathsf{state}[s]$) to true (with $s = \mathsf{slot}(b_j')$) to cast a notarization vote for b_j', where b is the parent of b_j'. $\mathsf{ParentReady}(\mathsf{hash}(b))$ is added to $\mathsf{state}[s]$ only when $\mathsf{ParentReady}(s, \mathsf{hash}(b))$ is emitted. Note that by Definition 15, if a correct node has emitted $\mathsf{ParentReady}(s, \mathsf{hash}(b))$, then b is notarized or notarized-fallback. If $\mathsf{slot}(b) < \mathsf{slot}(b_i)$, by Definition 15 Pool holds a skip certificate for $\mathsf{slot}(b_i)$, contradicting Lemma 21 or 26. If $\mathsf{slot}(b) = \mathsf{slot}(b_i)$, since b is notarized or notarized-fallback, again Lemma 21 or 26 is violated. Due to choice of b_j, $\mathsf{slot}(b_i) < \mathsf{slot}(b)$ is also impossible. □

Theorem 1 (safety)**.** *If any correct node finalizes a block b in slot s and any correct node finalizes any block b' in any slot $s' \geq s$, b' is a descendant of b.*

Proof. By Lemma 25, b' is also notarized. By Lemmas 31 and 32, b' is a descendant of b. □

2.10 Liveness

Lemma 33. *If a correct node emits the event* $\mathit{ParentReady}(s, \dots)$*, then for every slot* k *in the leader window beginning with* s *the node will emit the event* $\mathit{Timeout}(k)$*.*

Proof. The handler of event $\mathsf{ParentReady}(s, \dots)$ in line 12 of Algorithm 1 calls the function setTimeouts(s) which schedules the event $\mathsf{Timeout}(k)$ for every slot k of the leader window containing s (i.e., $k \in$ windowSlots(s)). □

If a node scheduled the event $\mathsf{Timeout}(k)$, we say that it set the timeout for slot k.

Since the function setTimeouts(s) is called only in the handler of the event $\mathsf{ParentReady}(s, \dots)$ in Algorithm 1, we can state the following corollary:

Corollary 34. *If a node sets a timeout for slot* s*, the node emitted an event* $\mathit{ParentReady}(s', \mathsf{hash}(b))$*, where* s' *is the first slot of the leader window* windowSlots(s)*.*

Lemma 35. *If all correct nodes set the timeout for slot* s*, all correct nodes will cast a notarization vote or skip vote in slot* s*.*

Proof. For any correct node that set the timeout for slot s, the handler of event $\mathsf{Timeout}(s)$ in line 6 of Algorithm 1 will call the function trySkipWindow(s), unless $\mathsf{Voted} \in \mathsf{state}[s]$. Next, either $\mathsf{Voted} \notin \mathsf{state}[s]$ in line 24 of Algorithm 2, and the node casts a skip vote in slot s, or $\mathsf{Voted} \in \mathsf{state}[s]$. The object Voted is added to $\mathsf{state}[s]$ only when the node cast a notarization or skip vote in slot s, and therefore the node must have cast either vote. □

Lemma 36. *If no set of correct nodes with more than 40% of stake cast their notarization votes for the same block in slot* s*, no correct node will add the object* $\mathit{ItsOver}$ *to* $\mathit{state}[s]$*.*

Proof. Object $\mathsf{ItsOver}$ is only added to $\mathsf{state}[s]$ in line 21 of Algorithm 2 after testing that $\mathsf{BlockNotarized}(\mathsf{hash}(b)) \in \mathsf{state}[s]$. The object $\mathsf{BlockNotarized}(\mathsf{hash}(b))$ is only added to $\mathsf{state}[s]$ when the event $\mathsf{BlockNotarized}(s, \mathsf{hash}(b))$ is handled in Algorithm 1. By Definition 15, Pool needs to hold a notarization certificate for b to emit the event. The certificate requires that 60% of stake voted to notarize b (Def. 11). Since we assume that byzantine nodes hold less than 20% of stake, correct nodes with more than 40% of stake need to cast their notarization votes for the same block in slot s for any correct node to add the object $\mathsf{ItsOver}$ to $\mathsf{state}[s]$. □

Lemma 37. *If all correct nodes set the timeout for slot* s*, either the skip certificate for* s *is eventually observed by all correct nodes, or correct nodes with more than 40% of stake cast notarization votes for the same block in slot* s*.*

Proof. Suppose no set of correct nodes with more than 40% of stake cast their notarization votes for the same block in slot s.

Since all correct nodes set the timeout for slot s, by Lemma 35, all correct nodes will observe skip votes or notarization votes in slot s from a set S of correct nodes with at least 80% of stake (Assumption 1).

Consider any correct node $v \in S$. As in Definition 16, by $\mathsf{notar}(b)$ denote the cumulative stake of nodes whose notarization votes for block b in slot $s = \mathsf{slot}(b)$ are in v's Pool, and by $\mathsf{skip}(s)$ denote the cumulative stake of nodes whose skip votes for slot s are in Pool of v. Let w be the stake of nodes outside of S whose notarization or skip vote v observed. Then, after v received votes of nodes in S: $\mathsf{skip}(s) + \sum_b \mathsf{notar}(b) = 80\% + w$. Since no set of correct nodes with more than 40% of stake cast their notarization votes for the same block in slot s, $\max_b \mathsf{notar}(b) \leq 40\% + w$. Therefore,

$$\begin{aligned} \mathsf{skip}(s) + \sum_b \mathsf{notar}(b) - \max_b \mathsf{notar}(b) = \\ 80\% + w - \max_b \mathsf{notar}(b) \geq \\ 80\% + w - (40\% + w) = 40\%. \end{aligned}$$

Therefore, if v has not cast a skip vote for s, v will emit the event $\mathsf{SafeToSkip}(s)$. By Lemma 36, v will test that $\mathsf{ItsOver} \notin \mathsf{state}[s]$ in line 23 of Algorithm 1, and cast a skip-fallback vote for s.

Therefore, all correct node will cast a skip or skip-fallback vote for s and observe a skip certificate for s. □

Lemma 38. *If correct nodes with more than 40% of stake cast notarization votes for block b, all correct nodes will observe a notar-fallback certificate for b.*

Proof. Reason by induction on the difference between $\mathsf{slot}(b)$ and the first slot in $\textsc{windowSlots}(\mathsf{slot}(b))$.

Suppose $\mathsf{slot}(b)$ is the first slot in the window. Suppose for contradiction some correct node v will not cast a notarization or notar-fallback vote for b. Since v will observe the notarization votes of correct nodes with more than 40% of stake, by Definition 16 v will emit $\mathsf{SafeToNotar}(\mathsf{slot}(b), \mathsf{hash}(b))$.

The object $\mathsf{ItsOver}$ is added to $\mathsf{state}[\mathsf{slot}(b)]$ in line 21 of Algorithm 2 after casting a finalization vote. The condition in line 19 ensures that v cast a notarization vote for a notarized block b'. However, by Lemma 23, there can be no such $b' \neq b$ in the same slot, and v has not cast the notarization vote for b.

When triggered by $\mathsf{SafeToNotar}(\mathsf{slot}(b), \mathsf{hash}(b))$, v will test that $\mathsf{ItsOver} \notin \mathsf{state}[s]$ in line 18 and cast the notar-fallback vote for b, a contradiction.

Therefore, all correct nodes will cast a notarization or notar-fallback vote for b, and observe a notar-fallback certificate for b.

Next, suppose $\mathsf{slot}(b)$ is not the first slot in the window and assume the induction hypothesis holds for the previous slot.

Suppose for contradiction some correct node v will not cast a notarization or notar-fallback vote for b. Since v will observe the notarization votes of correct nodes with more than 40% of stake, by Definition 16 v will retrieve block b and identify its parent b'. By Lemma 28, the correct nodes that cast notarization votes for b also voted for b', and $\mathsf{slot}(b') = \mathsf{slot}(b) - 1$. By induction hypothesis, v will observe a notar-fallback certificate for b', and emit $\mathsf{SafeToNotar}(\mathsf{slot}(b), \mathsf{hash}(b))$. Identically to the argument above, v will cast the notar-fallback vote for b, causing a contradiction.

Therefore, all correct nodes will cast a notarization or notar-fallback vote for b, and observe a notar-fallback certificate for b. □

Lemma 39. *If all correct nodes set the timeouts for slots of the leader window* $\textsc{windowSlots}(s)$*, then for every slot* $s' \in \textsc{windowSlots}(s)$ *all correct nodes will observe a notar-fallback certificate for* b *in slot* $s' = \mathsf{slot}(b)$*, or a skip certificate for* s'*.*

Proof. If correct nodes observe skip certificates in all slots $s' \in \textsc{windowSlots}(s)$, we are done. Otherwise, let $s' \in \textsc{windowSlots}(s)$ be any slot for which a correct node will not observe a skip certificate. By Lemma 37, there is a block b in slot $s' = \mathsf{slot}(b)$ such that correct nodes with more than 40% of stake cast the notarization vote for b. By Lemma 38, correct nodes will observe a notar-fallback certificate for b. □

Lemma 40. *If all correct nodes set the timeouts for slots* $\textsc{windowSlots}(s)$*, then all correct nodes will emit the event* $\mathsf{ParentReady}(s_+, \dots)$*, where* $s_+ > s$ *is the first slot of the following leader window.*

Proof. Consider two cases:

(i) all correct nodes observe skip certificates for all slots in $\textsc{windowSlots}(s)$;

(ii) some correct node does not observe a skip certificate for some slot $s' \in \textsc{windowSlots}(s)$.

(i) Consider some correct node v. By Corollary 34, v had emitted an event $\mathsf{ParentReady}(k, \mathsf{hash}(b))$, where k is the first slot of $\textsc{windowSlots}(s)$. By Definition 15, there is a block b, such that v observed a notar-fallback certificate for b, and skip certificates for all slots i such that $\mathsf{slot}(b) < i < k$. Since v will observe skip certificates for all slots in $\textsc{windowSlots}(s)$, v will observe skip certificates for all slots i such that $\mathsf{slot}(b) < i < s_+$. By 15, v will emit $\mathsf{ParentReady}(s_+, \mathsf{hash}(b))$.

(ii) Let s' be the highest slot in $\textsc{windowSlots}(s)$ for which some correct node v will not observe a skip certificate. By Lemma 39, v will observe a notar-fallback certificate for some block b in slot $s' = \mathsf{slot}(b)$. By definition of

s', v will observe skip certificates for all slots i such that $\mathsf{slot}(b) < i < s_+$. By 15, v will emit $\mathsf{ParentReady}(s_+, \mathsf{hash}(b))$. □

Lemma 41. *All correct nodes will set the timeouts for all slots.*

Proof. Follows by induction from Lemma 33 and Lemma 40. □

Lemma 42. *Suppose it is after GST and the first correct node v set the timeout for the first slot s of a leader window* WINDOWSLOTS(s) *at time t. Then, all correct nodes will emit some event $\mathsf{ParentReady}(s, \mathsf{hash}(b))$ and set timeouts for slots in* WINDOWSLOTS(s) *by time $t + \Delta$.*

Proof. By Corollary 34 and Definition 15, v observed a notar-fallback certificate for some block b and skip certificates for all slots i such that $\mathsf{slot}(b) < i < s$ by time t. Since v is correct, it broadcast the certificates, which were also observed by all correct nodes by time $t + \Delta$. Therefore, all correct nodes emitted $\mathsf{ParentReady}(s, \mathsf{hash}(b))$ by time $t + \Delta$ and set the timeouts for all slots in WINDOWSLOTS(s). □

Theorem 2 (liveness)**.** *Let v_ℓ be a correct leader of a leader window beginning with slot s. Suppose no correct node set the timeouts for slots in* WINDOWSLOTS(s) *before GST, and that Rotor is successful for all slots in* WINDOWSLOTS(s)*. Then, blocks produced by v_ℓ in all slots* WINDOWSLOTS(s) *will be finalized by all correct nodes.*

Proof. The intuitive outline of the proof is as follows:

(1) We calculate the time by which correct nodes receive blocks.

(2) Suppose for contradiction some correct node v cast a skip vote. We argue that v cast a skip vote in every slot $k' \geq k$, $k' \in$ WINDOWSLOTS(s).

(3) We consider different causes for the first skip vote cast by v. We determine that some $\mathsf{Timeout}(j)$ resulted in casting a skip vote by v before any $\mathsf{SafeToNotar}$ or $\mathsf{SafeToSkip}$ is emitted in the window.

(4) We argue that $\mathsf{Timeout}(k)$ can only be emitted after v has already received a block and cast a notarization vote in slot k, a contradiction.

(1) By Lemma 41, all correct nodes will set the timeouts for s. Let t be the time at which the first correct node sets the timeout for s. Since $t \geq \mathsf{GST}$, by Lemma 42, v_ℓ emitted $\mathsf{ParentReady}(s, \mathsf{hash}(b))$ for some b and added $\mathsf{ParentReady}(\mathsf{hash}(b))$ to $\mathsf{state}[s]$ in line 13 of Algorithm 1 by time $t + \Delta$. Conditions in lines 1 and 4 of Algorithm 3 imply that after $\mathsf{ParentReady}(\mathsf{hash}(b)) \in \mathsf{state}[s]$, v_ℓ proceeded to line 10 by time $t + \Delta$. According to lines 11 and 16, v_ℓ will finish transmission of a block b_k in slot $k \in$ WINDOWSLOTS(s) by time $t + \Delta + (k - s + 1) \cdot \Delta_{\mathsf{block}}$. Since Rotor is successful for slots in WINDOWSLOTS(s),

correct nodes will receive the block in slot $k \in \text{WINDOWSLOTS}(s)$ by time $t + 3\Delta + (k - s + 1) \cdot \Delta_{\mathsf{block}}$.

(2) Suppose for contradiction, some correct node v will not cast a notarization vote for some b_k, and let k be the lowest such slot. Since v_ℓ is correct, the only valid block received by any party in slot k is b_k, and v cannot cast a different notarization vote in slot k. By Lemma 35, v will cast a skip vote in slot k. Moreover, v cannot cast a notarization vote in any slot $k' > k$ in the leader window, due to the latter leg of the condition in line 11 of Algorithm 2 (i.e. **not** firstSlot **and** VotedNotar($\mathsf{hash}_{\mathsf{parent}}$) $\in \mathsf{state}[k' - 1]$). Therefore, v cast a skip vote in every slot $k' \geq k$, $k' \in \text{WINDOWSLOTS}(s)$.

(3) Skip votes in slot k are cast by TRYSKIPWINDOW(j) in Algorithm 2, where $j \in \text{WINDOWSLOTS}(s)$. The function TRYSKIPWINDOW(j) is called after handling SafeToNotar($j, \dots$), SafeToSkip(j), or Timeout(j) in Algorithm 1. Let j be the slot such that the first skip vote of v for a slot in WINDOWSLOTS(s) resulted from handling SafeToNotar($j, \dots$), SafeToSkip(j), or Timeout(j). Consider the following cases:

- SafeToNotar($j, \dots$): If $j < k$, by definition of k, all correct nodes cast notarization votes for b_j. Therefore, SafeToNotar($j, \dots$) cannot be emitted by a correct node. Therefore, $j \geq k$. SafeToNotar($j, \dots$) requires v to cast a skip vote in slot j first. Therefore, v cast a skip vote for slot j before emitting SafeToNotar($j, \dots$), a contradiction.

- SafeToSkip(j): Similarly to SafeToNotar, the event cannot be emitted by a correct node for $j < k$, and requires that v cast some skip vote for slot $j \geq k$ before it is emitted, a contradiction.

- Timeout(j): Due to the condition when handling the event in line 6 of Algorithm 1, the event does not have any effect if v cast a notarization vote in slot j. Moreover, v cannot cast a notarization vote in slot j if Timeout(j) was emitted beforehand. Since v cast notarization votes in slots of the window lower than k, then $j \geq k$. Since the event Timeout(j) is scheduled at a higher time for a higher slot in line 5 of Algorithm 2, the time at which Timeout(k) is emitted is the earliest possible time at which v cast the first skip vote in the window.

(4) Since t is the time at which the first correct node set the timeout for slot s, v emitted Timeout(k) at time $t' \geq t + \Delta_{\mathsf{timeout}} + (k - s + 1) \cdot \Delta_{\mathsf{block}} \geq t + 3\Delta + (k - s + 1) \cdot \Delta_{\mathsf{block}}$. However, as calculated above, v has received b_i for all $s \leq i \leq k$ by that time. Analogously to Lemma 42, v has also emitted ParentReady($s, \mathsf{hash}(b)$) and added ParentReady($\mathsf{hash}(b)$) to $\mathsf{state}[s]$, where b is the parent of b_s. The condition in line 11 is satisfied when v calls TRYNOTAR(Block($s, \mathsf{hash}(b_s), \mathsf{hash}(b)$)), and v cast a notarization vote for b_s. Since CHECKPENDINGBLOCKS() is called in lines 3 and 14 of Algorithm 1 when handling Block and ParentReady events, v cast a notarization vote for b_i for all $s \leq i \leq k$ by the time Timeout(k) is emitted, irrespectively of the

order in which b_i were received. This contradicts the choice of v as a node that did not cast a notarization vote for b_k.

Since for all $k \in \text{WINDOWSLOTS}(s)$ all correct nodes cast notarization votes for b_k, all correct nodes will observe the fast-finalization certificate for b_k and finalize b_k. □

2.11 Higher Crash Resilience

In this section we sketch the intuition behind Alpenglow's correctness in less adversarial network conditions, but with more crash faults.

In harsh network conditions Alpenglow can be attacked by an adversary with over 20% of stake. However, such an attack requires careful orchestration. Unintentional mistakes, crash faults and denial-of-service attacks (which are functionally akin to crash faults) have historically caused more problems for blockchain systems. In the rest of this section, we will consider Assumption 2 instead of Assumption 1. Additionally, Assumption 3 captures on a high level the attacker's lesser control over the network.

Assumption 3 (Rotor non-equivocation)**.** *If a correct node receives a full block b via Rotor (Section 2.2), any other correct node that receives a full block via Rotor for the same slot, receives the same block b.*

Note that crashed nodes are functionally equivalent to nodes exhibiting indefinite network delay. In Section 2.9 we have demonstrated that Alpenglow is safe with arbitrarily large network delays, which are possible in our model. Therefore, safety is ensured under Assumption 2.

The reasoning behind liveness (Section 2.10) is affected by Assumption 2 whenever we argue that correct nodes will observe enough votes to trigger the conditions of Definition 16 (SafeToNotar and SafeToSkip). However, with the additional Assumption 3 that two correct nodes cannot reconstruct a different block in the same slot, either SafeToNotar or SafeToSkip has to be emitted by all correct nodes after they observe the votes of other correct nodes. If correct nodes with at least 20% of stake voted to notarize a block, then the condition:

$$\Big(\mathsf{skip}(s) + \mathsf{notar}(b) \geq 60\% \textbf{ and } \mathsf{notar}(b) \geq 20\%\Big)$$

will be satisfied after votes of all correct nodes are observed. Otherwise,

$$\mathsf{skip}(s) + \sum_b \mathsf{notar}(b) - \max_b \mathsf{notar}(b) \geq \mathsf{skip}(s) \geq 40\%$$

will be satisfied.

Corollary 43. *Theorem 2 holds under Assumptions 2 and 3 instead of Assumption 1.*

Analysis

Note that if the leader is correct or crashed, Assumption 3 is never violated, as the leader would produce at most one block per slot. Therefore, crash-only faults amounting to less than 40% of stake are always tolerated.

To conclude, we intuitively sketch the conditions in which Assumption 3 can be violated by an adversary distributing different blocks to different parties. If there are also many crash nodes in this scenario, correct nodes might not observe enough votes to emit SafeToNotar or SafeToSkip, and the protocol could get stuck.

Suppose a malicious leader attempts to distribute two different blocks b and b' such that some correct nodes reconstruct and vote for b, while other correct nodes reconstruct and vote for b'. If a correct node receives two shreds not belonging to the same block (having a different Merkle root for the same slice index) before being able to reconstruct the block, the node will not vote for the block. Therefore, network topology and sampling of Rotor relays determines the feasibility of distributing two different blocks to different correct nodes.

Example 44. *Consider two clusters of correct nodes A and B, such that the network latency within a cluster is negligible in relation to the network latency between A and B. Each A and B are comprised of nodes with 31% of stake. The adversary controls 18% of stake, and 20% of stake is crashed. The Rotor relays in A receive shreds for a block b_A from a malicious leader, while Rotor relays in B receive shreds for a block b_B. The Rotor relays controlled by the adversary forward shreds of b_A to A, and shreds of b_B to B. Due to the delay between A and B, nodes in A will reconstruct b_A before observing any shred of b_B. Similarly for B and b_B. Assumption 3 is violated in this scenario.*

If the network topology has uniformly distributed nodes, it is harder to arrange for large groups to receive enough shreds of a slice of b before receiving any shreds of a corresponding slice of b'.

3 Beyond Consensus

Intiuition

This section describes a few issues that are not directly in the core of the consensus protocol but deserve attention. We start with three issues (sampling, rewards, execution) closely related to consensus, then we move on to advanced failure handling, and we finish the section with bandwidth and latency simulation measurement results.

3.1 Smart Sampling

To improve resilience of Rotor in practice, we use a novel committee sampling scheme. It is inspired by FA1 [GKR23] and improves upon FA1-IID. It takes the idea of reducing variance in the sampling further.

Definition 45. *Given a number of bins k and relative stakes $0 < \rho_1, \ldots, \rho_n < 1$. A* partitioning *of these stakes is a mapping*

$$p : \{1, \ldots, k\} \times \{1, \ldots, n\} \to [0, 1]_{\mathbb{R}},$$

such that:

- *stakes are fully assigned, i.e., $\forall v \in \{1, \ldots, n\} : \sum_{b \in \{1,\ldots,k\}} p(b, v) = \rho_v$, and*
- *bins are filled entirely, i.e., $\forall b \in \{1, \ldots, k\} : \sum_{v \in \{1,\ldots,n\}} p(b, v) = 1/k$.*

A procedure that for any number of bins k and relative stakes $\rho_1, \ldots, \rho_n$ calculates a valid partitioning is called a partitioning algorithm.

Definition 46. *Our committee sampling scheme, called* partition sampling *or PS-P, is instantiated with a specific partitioning algorithm P. It then proceeds as follows to generate a single set of Γ samples:*

1. *For each node with relative stake $\rho_i > 1/\Gamma$, fill $\lfloor \rho_i \Gamma \rfloor$ bins with that node. The remaining stake is $\rho'_i = \rho_i - \frac{\lfloor \rho_i \Gamma \rfloor}{\Gamma} < 1/\Gamma$. For all other nodes, the remaining stake is their original stake: $\rho'_i = \rho_i$*
2. *Calculate a partitioning for stakes $\rho'_1, \ldots, \rho'_n$ into the remaining $k = \Gamma - \sum_{i \in [n]} \lfloor \rho_i \Gamma \rfloor$ bins according to P.*
3. *From each bin, sample one node proportional to their stake.*

Protocol

One simple example for a partitioning algorithm randomly orders nodes, and make cuts exactly after every $1/k$ relative stake. PS-P instantiated with this simple partitioning algorithm is already better than the published state of the art [GKR23]. However, this topic deserves more research.

Next, we show that PS-P improves upon IID and FA1-IID. Let $\mathcal{A}$ denote the adversary and $\rho_{\mathcal{A}}$ the total stake they control, possibly spread over many nodes. Further, assume $\rho_{\mathcal{A}} < \gamma/\Gamma = 1/\kappa$ and therefore $\gamma < \rho_{\mathcal{A}}\Gamma$.

Lemma 47. *For any stake distribution with $\rho_i < 1/\Gamma$ for all $i \in \{1, \ldots, n\}$, any partitioning algorithm P, adversary $\mathcal{A}$ being sampled at least γ times in PS-P is at most as likely as likely as in IID stake-weighted sampling.*

Proof. For any partitioning, in step 3 of Definition 46, the number of samples for the adversary is Poisson binomial distributed, i.e., it is the number

of successes in Γ independent Bernoulli trials (possibly with different probabilities). The success probability of each trial is the proportion of stake in each bin the adversary controls. Consider the case where $\mathcal{A}$ achieves to be packed equally in all Γ bins. In that case, the number of samples from the adversary follows the Binomial distribution with $p = \rho_{\mathcal{A}}$. This is the same as for IID stake-weighted sampling. Also, the Binomial case is also known to be maximizing the variance for Poisson binomial distributions [Hoe56], thus maximizing the probability for the adversary to get sampled at least $\gamma < \Gamma$ times. □

Theorem 3. *For any stake distribution, adversary $\mathcal{A}$ being sampled at least γ times in PS-P is at most as likely as in FA1-IID.*

Proof. Because of step 1 of in Definition 46, applying our scheme directly is equivalent to using FA1 with our scheme as the fallback scheme it is instantiated with. Therefore, together with Lemma 47, the statement follows. □

Finally, we practically analyze how this sampling scheme compares to regular stake-weighted IID sampling and FA1-IID on the current Solana stake distribution.

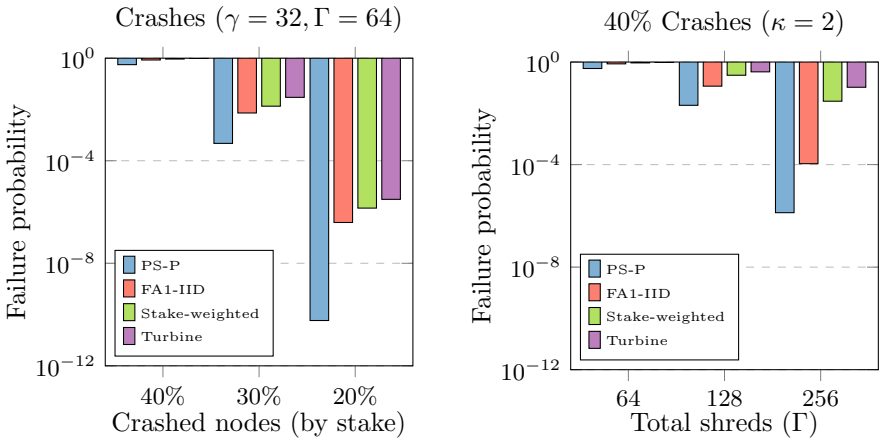

Figure 9: Probabilities that Rotor is not successful when experiencing crash failures, when instantiated with PS-P (with fully randomized partitioning) compared to other sampling techniques. This assumes 64 slices per block (Rotor is only successful for the block if it is successful for every slice).

3.2 Voting vs. Execution

Intiuition

In Section 2, we omitted the execution of the blocks and the transactions therein. Currently, Solana uses the synchronous execution model described below.

Eager (Synchronous) Execution. The leader executes the block before sending it, and all nodes execute the block before voting for it. With the slices being pipelined (the next slice is propagated while the previous slice is executed), this may add some time to the critical path, since we need to execute the last slice before we can send a notarization vote for the block.

Lazy (Asynchronous) Execution. We can also vote on a block before executing it. We need to make sure that the Compute Units (CUs) reflect actual execution costs. This way the CU bounds on transactions and the whole block guarantee that blocks are executed timely. If CUs are unrealistically optimistic, this cannot work since execution delays may grow without bounds.

Distributed Execution. Another active area of research is distributed execution, which is related to this discussion about execution model. In distributed execution validators use multiple machines (co-located for minimal latency) for executing transactions. Ideally, in contrast to executions on a single machine, this allows the system to scale to higher transaction throughputs. It also allows nodes to respond to surges in traffic without always over-provisioning (this is called elasticity). Examples of this line of research are Pilotfish [Kni+25] and Stingray [SSK25].

3.3 Asynchrony in Practice

Intiuition

In our model assumptions of Section 1.5 we assumed that delayed messages are eventually delivered. While this is a standard model in distributed computing, in reality (as well as in the original formulation of partial synchrony with GST [DLS88]) messages might be lost. Note that we already allow asynchrony (arbitrarily long message delays), so our protocol is safe even if messages are dropped. In this section we discuss two mechanisms enhancing Alpenglow to address network reality in practice, to restore liveness if the protocol makes no progress.

Joining. Nodes might go offline for a period of time and miss all of the messages delivered during that time. We note that if a rebooting or newly joining node observes a finalization of block b in slot s, it is not necessary to observe any vote or certificate messages for earlier slots. Due to safety (Theorem 1), any future block in a slot $s' \geq s$ that might be finalized will be a descendant of b, and if any correct node emits the event $\mathsf{ParentReady}(s', b')$,

b' has to be a descendant of b.

Rebooting or joining nodes need to observe a fast-finalization certificate for a block b in slot s, or a finalization certificate for s together with a notarization certificate for b in the same slot s. Block b can be retrieved with Repair Section 2.8. The parent of b can be identified and retrieved after b is stored, and so on. A practical implementation might retrieve any missing blocks for all slots in parallel, before verifying and repairing all ancestors of b.

Standstill. Eventual delivery of messages needs to be ensured to guarantee liveness after GST. As noted above, if a correct node observes a finalization in slot s, no vote or certificate messages for slots earlier than s are needed for liveness. Lack of liveness can be detected simply by observing a period of time without new slots being finalized. After some parametrized amount of time, e.g., $\Delta_{\mathsf{standstill}} \approx 10$ sec in which the highest finalized slot stays the same, correct nodes trigger a re-transmission routine. Then, nodes broadcast a finalization certificate for the highest slot observed (either a fast-finalization certificate for a block b in slot s, or a finalization certificate for s together with a notarization certificate for b in the same slot s). Moreover, for all higher slots $s' > s$, nodes broadcast observed certificates and own votes cast in these slots.

Protocol

3.4 Dynamic Timeouts

Alpenglow is defined in the partially synchronous model, but strictly speaking, epochs deviate from partial synchrony. For epoch changes to work, at least one block needs to be finalized in each epoch. A finalized block in epoch e makes sure that the previous epoch $e-1$ ended with an agreed-upon state. This is important for setting the stage of epoch $e+1$, i.e., to make sure that there is agreement on the nodes and their stake at the beginning of epoch $e+1$.

Intiuition

Our solution is to extend timeouts if the situation looks dire. More precisely, if a node does not have a finalized block in $\Delta_{\mathsf{standstill}} \approx 10$ sec of consecutive leader windows, the node will start extending its timeouts by $\varepsilon \approx 5\%$ in every leader window.

Note that the nodes do not need coordination in extending the timeouts. As soon as nodes see finalized blocks again, they can return to the standard timeouts immediately as described in Section 2.6. Also when returning to normal timeouts, no agreement or coordination is needed, and some nodes can still have longer timeouts without jeopardizing the correctness of the system.

Increasing timeouts by $\varepsilon \approx 5\%$ in every leader window results in exponential growth. With exponential growth, timeouts quickly become longer than any extraordinary network delay caused by any network/power disaster. Therefore, it can be guaranteed that we have a finalized slot in each epoch.

Protocol

3.5 Protocol Parameters

Throughout the document we have introduced various parameters. Table 10 shows how we set the parameters in our preliminary simulations. Testing is needed to ultimately decide these parameters.

Some parameters are set implicitly, and will be different in every epoch. This includes in particular the parameter for the number of nodes n. Throughout this paper we used $n \approx 1{,}500$ for the number of nodes. The reality at the time of writing is closer to $n \approx 1{,}300$.

Blocks per leader window w	4
Data shreds per slice γ	32
Coding shreds per slice Γ	64
Timeout increase ε	5%
Maximum nodes $n_{\max}$	2,000
Standstill trigger $\Delta_{\mathsf{standstill}}$	10 sec
Block time Δ_{block}	400 ms
Epoch length L	18,000 slots

Table 10: Protocol Parameters.

Protocol

3.6 Bandwidth

In this section we analyze the bandwidth usage of Alpenglow. Table 11 lists the size of Votor-related messages. As a bandwidth optimization, only one of the finalization certificates should be broadcast (whichever is observed first). Then, in the common case, every node broadcasts a notarization vote, finalization vote, notarization certificate and one of the finalization certificates for every slot. If we account for the larger of the finalization certificates (fast-finalization), for $n = 1{,}500$, a node transmits $(196 + 384 + 384 + 164) \cdot 1{,}500$ bytes for every 400 ms slot, which corresponds to 32.27 Mbit/s. The total

outgoing bandwidth is plotted in Figure 12.

Message	*BLS Sig.*	*Slot Number*	*Block Hash*	*Node Bitmap*	*MAC*	*Headers*	*Total*
notar. vote	96	8	32	–	32	28	196
notar. cert.	96	8	32	188	32	28	384
fast-final. cert.	96	8	32	188	32	28	384
final. vote	96	8	–	–	32	28	164
final. cert.	96	8	–	188	32	28	352
skip vote	96	8	–	–	32	28	164
skip cert.	96	8	–	188	32	28	352

Table 11: Estimation of message sizes in bytes for a network comprised of 1,500 nodes.

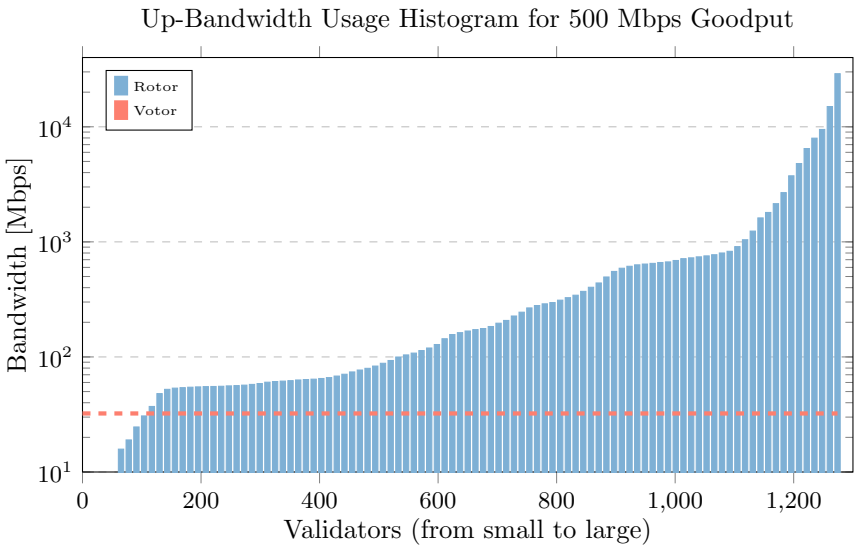

Figure 12: Bandwidth usage to achieve consistent goodput of 500 Mbps, i.e., where the leader role requires sending at 1 Gbps for $\kappa = 2$.

3.7 Latency

We simulated Alpenglow in a realistic environment. In particular, in our simulation, the stake distribution is the same as Solana's stake distribution at the time of writing (epoch 780), and the latencies between nodes correspond to real-world latency measurements. Some possible time delays are not included in the simulation, in particular block execution time. Moreover, a different stake distribution would change our results.

Figure 13 shows a latency histogram for the case when the block leader is located in Zurich, Switzerland, our location at the time of writing. The leader is fixed in Zurich, and each bar shows the average over 100,000 simulated executions. The Rotor relays are chosen randomly, according to stake. We plot simulated latencies to reach different stages of the Alpenglow protocol against the fraction of the network that arrived at that stage.

- The green bars show the network latency. With the current node distribution of Solana, about 65% of Solana's stake is within 50 ms network latency round-trip time) of Zurich. The long tail of stake has more than 200 ms network latency from Zurich. The network latency serves as a natural lower bound for our plot, e.g., if a node is 100 ms from Zurich, then any protocol needs at least 100 ms to finalize a block at that node.
- The yellow bars show the delay incurred by Rotor, the first stage of our protocol. More precisely, the yellow bars show when the nodes received γ shreds, enough to reconstruct a slice.
- The red bars mark the point in time when a node has received notarization votes from at least 60% of the stake.
- Finally, the blue bars show the actual finalization time. A node can finalize because they construct a fast-finalization certificate (having received 80% stake of the original notarization votes), or a finalization certificate (having received 60% of the finalization votes), or having received one of these certificates from a third party, whatever is first.

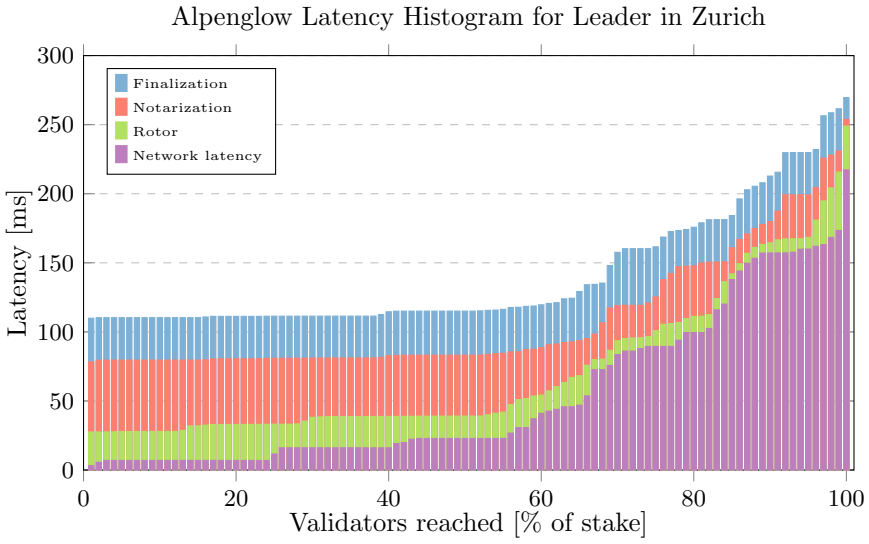

Figure 13: For a fixed leader in Zurich with random relays we have: (i) the last node in the network finalizes in less than 270 ms, (ii) the median node finalizes almost as fast as the fastest ones, in roughly 115 ms.

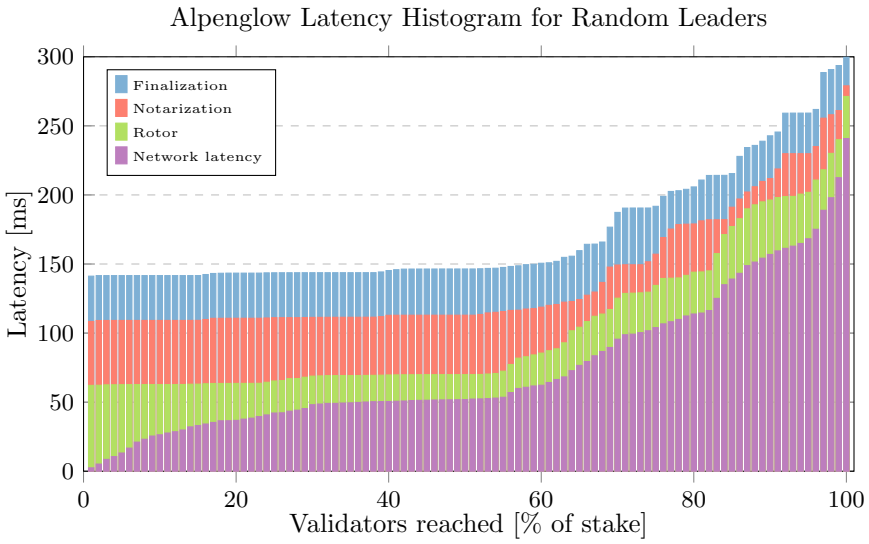

Figure 14: This plot is a generalized version of Figure 13, where the leader is chosen randomly according to stake. While Zurich is *not* "the center of the Solana universe," it is more central than the average leader. Hence the numbers in this plot are a bit higher than in Figure 13, and the median finalization time is roughly 150 ms.

Thanks. We thank the following people for their input: Ittai Abraham, Zeta Avarikioti, Emanuele Cesena, Igor Durovic, Yuval Efron, Pranav Garimidi, Sam Kim, Charlie Li, Carl Lin, Julian Loss, Zarko Milosevic, Gabriela Moreira, Karthik Narayan, Joachim Neu, Alexander Pyattaev, Ling Ren, Max Resnick, Tim Roughgarden, Ashwin Sekar, Victor Shoup, Philip Taffet, Yann Vonlanthen, Marko Vukolić, Josef Widder, Wen Xu, Anatoly Yakovenko, Haoran Yi, Yunhao Zhang.

References

[Abr+21] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. "Good-case Latency of Byzantine Broadcast: A Complete Categorization". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 2021, pages 331–341.

[Abr+17] Ittai Abraham et al. "Revisiting fast practical byzantine fault tolerance". In: *arXiv preprint arXiv:1712.01367* (2017).

[Aru+24] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. *Shoal++: High Throughput DAG-BFT Can Be Fast!* `https://decentralizedthoughts.github.io/2024-06-12-shoalpp/`. 2024.

[Aru+25] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. "Shoal++: High Throughput DAG BFT Can Be Fast and Robust!" In: *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2025.

[Bab+25] Kushal Babel et al. "Mysticeti: Reaching the Latency Limits with Uncertified DAGs". In: *32nd Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2025.

[Bon+03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. "Aggregate and verifiably encrypted signatures from bilinear maps". In: *Advances in Cryptology (Eurocrypt), Warsaw, Poland*. Springer. 2003, pages 416–432.

[BKM18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. *The latest gossip on BFT consensus*. arXiv:1807.04938, `http://arxiv.org/abs/1807.04938`. 2018.

[CT05] Christian Cachin and Stefano Tessaro. "Asynchronous verifiable information dispersal". In: *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2005, pages 191–201.

[CL99] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*. New Orleans, Louisiana, USA, 1999, pages 173–186.

[CP23] Benjamin Y. Chan and Rafael Pass. "Simplex Consensus: A Simple and Fast Consensus Protocol". In: *Theory of Cryptography (TCC), Taipei, Taiwan*. Taipei, Taiwan: Springer-Verlag, 2023, pages 452–479.

[Con+24] Andrei Constantinescu, Diana Ghinea, Jakub Sliwinski, and Roger Wattenhofer. "Brief Announcement: Unifying Partial Synchrony". In: *38th International Symposium on Distributed Computing (DISC)*. 2024.

[Dan+22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus". In: *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*. 2022, pages 34–50.

[Dod02] Yevgeniy Dodis. "Efficient construction of (distributed) verifiable random functions". In: *Public Key Cryptography (PKC), Miami, FL, USA*. Springer. 2002, pages 1–17.

[DGV04] Partha Dutta, Rachid Guerraoui, and Marko Vukolic. *The Complexity of Asynchronous Byzantine Consensus*. `https://infoscience.epfl.ch/server/api/core/bitstreams/19ce5930-31af-4489-9551-d5d014b8c1f1/content`. 2004.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. "Consensus in the Presence of Partial Synchrony". In: *J. ACM* 35.2 (1988), pages 288–323.

[FMW24] Austin Federa, Andrew McConnell, and Mateo Ward. *DoubleZero Protocol*. `https://doublezero.xyz/whitepaper.pdf`. 2024.

[Fou19] Solana Foundation. *Turbine–Solana's Block Propagation Protocol Solves the Scalability Trilemma*. `https://solana.com/news/turbine---solana-s-block-propagation-protocol-solves-the-scalability-trilemma`. 2019.

[GKR23] Peter Gazi, Aggelos Kiayias, and Alexander Russell. "Fait Accompli Committee Selection: Improving the Size-Security Tradeoff of Stake-Based Committees". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS), Copenhagen, Denmark*. ACM, 2023, pages 845–858.

[GV07] Rachid Guerraoui and Marko Vukolić. "Refined Quorum Systems". In: *Proceedings of the 26th Annual ACM Aymposium on Principles of Distributed Computing (PODC)*. 2007, pages 119–128.

[Gue+19] Guy Golan Gueta et al. "SBFT: A scalable and decentralized trust infrastructure". In: *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pages 568–580.

[Hoe56] Wassily Hoeffding. "On the distribution of the number of successes in independent trials". In: *The Annals of Mathematical Statistics* (1956), pages 713–721.

[Kei+22] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. "Cordial miners: Fast and efficient consensus for every eventuality". In: *arXiv:2205.09174* (2022).

[Kni+25] Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. "Pilotfish: Distributed Execution for Scalable Blockchains". In: *Financial Cryptography and Data Security (FC), Miyakojima, Japan*. Apr. 2025.

[Kot+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. "Zyzzyva: speculative byzantine fault tolerance". In: *Proceedings of 21st Symposium on Operating Systems Principles (SOSP)*. 2007, pages 45–58.

[Kur02] Klaus Kursawe. "Optimistic byzantine agreement". In: *21st IEEE Symposium on Reliable Distributed Systems (DSN)*. IEEE. 2002, pages 262–267.

[KTZ21] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. "Revisiting Optimal Resilience of Fast Byzantine Consensus". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. Virtual Event, Italy, 2021, pages 343–353.

[Lam03] Leslie Lamport. "Lower Bounds for Asynchronous Consensus". In: *Future Directions in Distributed Computing: Research and Position Papers*. 2003, pages 22–23.

[LNS25] Andrew Lewis-Pye, Kartik Nayak, and Nibesh Shrestha. *The Pipes Model for Latency and Throughput Analysis*. Cryptology ePrint Archive, Paper 2025/1116. 2025. URL: `https://eprint.iacr.org/2025/1116`.

[MA06] J-P Martin and Lorenzo Alvisi. "Fast byzantine consensus". In: *IEEE Transactions on Dependable and Secure Computing* (2006), pages 202–215.

[Mer79] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. Stanford University, 1979.

[MRV99] Silvio Micali, Michael Rabin, and Salil Vadhan. "Verifiable random functions". In: *40th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 1999, pages 120–130.

[Mil+16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. "The Honey Badger of BFT Protocols". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016, pages 31–42.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2 (1980), pages 228–234.

[Pos84] Jon Postel. *Standard for the Interchange of Ethernet Frames*. RFC 894. Apr. 1984.

[RS60] Irving S Reed and Gustave Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pages 300–304.

[Sho24] Victor Shoup. "Sing a Song of Simplex". In: *38th International Symposium on Distributed Computing (DISC)*. Volume 319. Leibniz International Proceedings in Informatics. Dagstuhl, Germany, 2024, 37:1–37:22.

[SSV25] Victor Shoup, Jakub Sliwinski, and Yann Vonlanthen. "Kudzu: Fast and Simple High-Throughput BFT". In: *arXiv:2505.08771* (2025).

[SKN25] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. "Hydrangea: Optimistic Two-Round Partial Synchrony with One-Third Fault Resilience". In: *Cryptology ePrint Archive* (2025).

[SR08] Yee Jiun Song and Robbert van Renesse. "Bosco: One-step byzantine asynchronous consensus". In: *International Symposium on Distributed Computing (DISC)*. Springer. 2008, pages 438–450.

[Spi+22] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. "Bullshark: DAG BFT protocols made practical". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022, pages 2705–2718.

[SSK25] Srivatsan Sridhar, Alberto Sonnino, and Lefteris Kokoris-Kogias. "Stingray: Fast Concurrent Transactions Without Consensus". In: *arXiv preprint arXiv:2501.06531* (2025).

[SDV19] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. "Mir-BFT: High-Throughput BFT for Blockchains". In: *arXiv:1906.05552* (2019).

[Von+24] Yann Vonlanthen, Jakub Sliwinski, Massimo Albarello, and Roger Wattenhofer. "Banyan: Fast Rotating Leader BFT". In: *25th ACM International Middleware Conference, Hong Kong, China*. Dec. 2024.

[Yak18] Anatoly Yakovenko. *Solana: A new architecture for a high performance blockchain v0.8.13*. `https://solana.com/solana-whitepaper.pdf`. 2018.

[Yan+22] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. "DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Renton, WA, Apr. 2022, pages 493–512.

[Yin+19] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. 2019, pages 347–356.

[Zha+11] Xin Zhang et al. “SCION: Scalability, Control, and Isolation on Next-Generation Networks”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2011, pages 212–227.