New Constellation*

Multiple Concurrent Proposers for Alpenglow

Quentin Kniep and Roger Wattenhofer

Solana Foundation

Version 1.0
November 14, 2026

Abstract

New Constellation (NewCon) extends Solana's Alpenglow consensus protocol with multiple concurrent proposers. Instead of one leader, k co-located proposers each build a block for every slot. The transactions of all accepted concurrent blocks are merged into one canonical order, so no single proposer controls inclusion or ordering. In this paper, we present NewCon, a design for multiple concurrent proposers on top of Solana's Alpenglow consensus protocol.

1 Introduction

1.1 Motivation

The promise of a blockchain is that no single party is in control. In practice, however, most blockchains, Solana included, still grant a *temporary monopoly* on transaction inclusion and ordering. Block production rotates according to a leader schedule, and the current leader can observe all pending transactions, reorder them, censor any it does not like, and insert its own transactions to front-run other users.

The profit from this temporary monopoly is known as maximal extractable value (MEV) [Dai+20]. Multiple concurrent proposers (MCP) [GNR25; Kni+26] are a promising way to break the temporary monopoly: if several proposers include transactions in the same slot, and their transactions are merged, no single proposer controls inclusion or ordering.

*Earlier drafts of this paper called the protocol Spectral. For marketing reasons ("Constellation = MCP of Solana"), the protocol is now called Constellation. Its design, however, differs fundamentally from the Constellation protocol we published earlier [Kni+26]. To avoid confusion, this paper calls our protocol *New Constellation*, or *NewCon* for short, and the earlier protocol *Legacy Constellation*, or *LegCon* for short. Outside this paper, our protocol may simply be called Constellation.

1.2 Properties

NewCon is designed to keep everything that makes Alpenglow fast and simple, but with multiple concurrent proposers. NewCon offers the following properties.

MEV Resistance.

- *Leaderless.* There is no leader. The proposers of a window are equals, unlike designs in which concurrent proposers feed a single leader. No single party controls inclusion or ordering in any slot (Section 2.1).
- *Censorship resistance.* Unless the slot is skipped entirely, a transaction only needs one non-censoring proposer among the proposers of a slot to be included (Theorem 2.2).
- *Deterministic ordering.* The transactions of all accepted blocks are merged into one deterministic order by priority fee (Section 3.3).
- *Block hiding.* Blocks are disseminated encrypted, and the keys are only released with the attestations, after all proposers have committed to their blocks. This commit-and-reveal scheme ensures that no proposer sees the transactions of the others before its own block is fixed (Section 2.2 and Theorem 2.3).
- *Transaction hiding.* A user can encrypt a transaction and split it among several proposers, so that no single proposer can read it on its own; it only becomes readable once enough parts are included. This trades some censorship resistance against hiding (Section 3.2).
- *Accountability.* Block hashes and attestations are signed, so a late proposer is identifiable and can be sanctioned, and an equivocating proposer can be proven guilty and slashed (Section 2.1 and Theorem 2.4).

Performance.

- *Shorter blocks.* Transactions are included every 50 ms, but validators only vote every few slots, so the number of consensus rounds does not grow (Section 2.1).
- *No extra latency.* Unlike designs that route transactions through an additional set of attesters, NewCon adds no global hop: blocks are disseminated by Rotor as in Alpenglow, and the local timing check runs in parallel (Section 1.3).
- *High bandwidth.* Dissemination via Rotor is unchanged, and the proposers share the capacity of a slot, so relays and validators carry the same load as with a single Alpenglow leader of the same throughput (Section 2.2).

- *No slot stretching.* Co-located proposers exchange signed block hashes directly and attest to the validators which of them were on time (Section 2.2). This bounds how late a proposer can be to a few milliseconds instead of the global Alpenglow timeout, so slots keep their nominal pace, and timing games and slot stretching are limited.

Robustness and Simplicity.

- *Generalizes Alpenglow.* Alpenglow runs unchanged on cblocks, each consisting of a few consecutive slots of all proposers (Section 2.4), so NewCon inherits Alpenglow's safety and liveness. With a single proposer and a vote on every slot, NewCon is essentially Alpenglow, so it can be deployed as an upgrade.
- *Execution before voting.* Validators execute every slot as soon as it has arrived and vote on the executed state (Section 2.4). Most MCP designs instead execute asynchronously, after consensus, so their votes cannot certify the resulting state.
- *No synchronized clocks.* Every timeout, including the check of the co-proposers' hashes, is measured with the local clock of a node and started by a local event, so NewCon, like Alpenglow, needs no physical clock synchronization (Section 1.4).
- *Crash tolerance.* A crashed proposer only loses its own blocks; the blocks of its co-proposers are still accepted (Section 2.3). In other words, the skip rate is *lower* than the fraction of crashed nodes.

1.3 Protocol Overview

Slots and Windows. As in Alpenglow, time is divided into slots, and consecutive slots are grouped into windows of w slots (we suggest $w = 16$). A slot lasts only $\Delta_{\mathsf{block}} = 50$ ms.

Proposers. Instead of a single leader, the schedule assigns k concurrent proposers to every window (initially $k = 2$ or 3; k can be increased later). The proposers of a window are co-located, i.e., they are within a small network delay Δ_{local} of each other, e.g., in the same city. Over an epoch, the windows rotate among such locations around the world (Section 3.1). Figure 1 shows NewCon from a bird's-eye view.

Block Production. Users send their transactions to one or several proposers of a slot (Section 3.2). In every slot, each proposer first generates a fresh symmetric key for its block, and then builds the block from the transactions it receives. Throughout the slot, it streams the block out in slices, as in Alpenglow. Every slice is encrypted with the block's key and disseminated to all validators via Rotor as soon as it is ready.

Hash Exchange. When streaming out the last slice of its block, each proposer also sends a signed hash of the whole block directly to its co-proposers (Figure 2). Since the proposers are co-located, a hash takes at most Δ_{local} to arrive. However, the proposers do not start their slots at exactly the same time, since their ParentReady may differ by up to Δ_{local}. So, measured from the time a proposer sends its own hash, the hash of a correct co-proposer arrives within $2\Delta_{\mathsf{local}}$: up to Δ_{local} because the co-proposer may have started later, and another Δ_{local} for the message itself. If the hash of a co-proposer does not arrive within these $2\Delta_{\mathsf{local}}$, the proposer does not wait any longer, but considers this co-proposer late. Since a single lost hash can cause a skip, we add fault tolerance, as discussed in the next section.

Attestations. After receiving all hashes, each proposer *attests* which of its co-proposers were on time: it directly sends a signed *attestation* to all validators, together with its own block encryption key. Validators do not accept a block whose proposer is marked as late. Since a proposer only reveals its key after it has seen the hashes of all others, no proposer can see the transactions of the others before it has committed to its own block.

Latency. The attestations and keys reach the validators at roughly the same time as the blocks themselves: the attestations are delayed by one extra local message exchange of at most $2\Delta_{\mathsf{local}}$ before they are sent directly to the validators, while the blocks take the extra relay hop of Rotor (which, on a network like Solana mainnet, adds a detour of roughly 20% of the global network delay [KSW25]).

Execution. The k proposer blocks of a slot form one *execution block* (*xblock*). As soon as a validator holds the blocks, the keys and the attestations of a slot, it decrypts the blocks, merges their transactions into one deterministic order by priority fee, and executes them. Figure 2 shows one slot.

Voting. Validators do not vote on every slot, but only on every ν-th slot (we suggest $\nu = 4$). Alpenglow runs unchanged on *consensus blocks* (*cblocks*), each

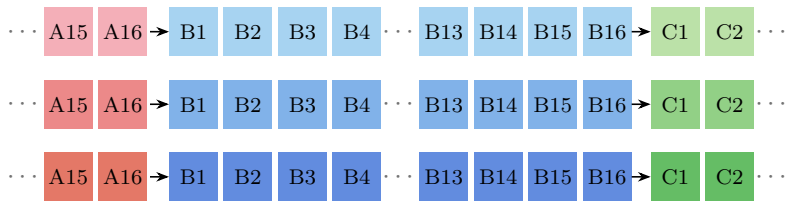


Figure 1: Overview of NewCon. Time is divided into windows A, B, C, ... of $w = 16$ slots each. Every window has k co-located proposers (here $k = 3$, one per row), and each of them produces a block in every slot.

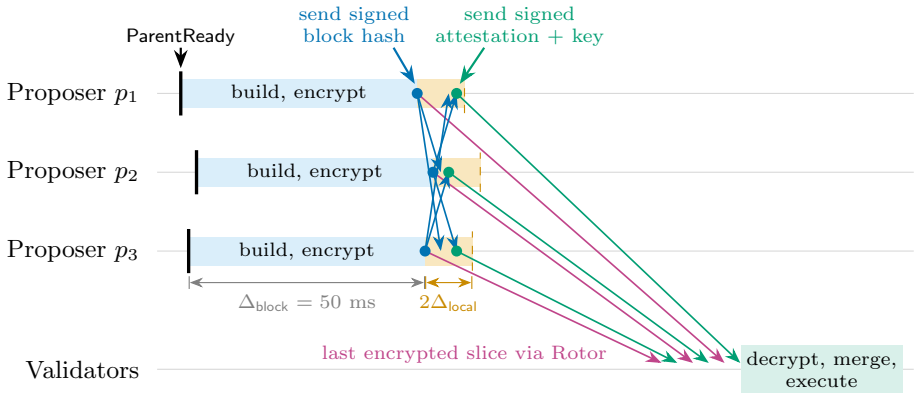


Figure 2: The first slot of a window with $k = 3$ proposers. Each proposer builds its block for Δ_{block} after its own ParentReady, which may differ by up to Δ_{local} between the co-located proposers. Throughout these Δ_{block}, it streams the block out encrypted, slice by slice, via Rotor to the validators (for clarity only the last slice per proposer is drawn). Concurrently, it sends a signed hash of its block directly to the other proposers. As soon as it has received the hashes of all others, within $2\Delta_{\text{local}}$ after its own sending time, a proposer sends a signed attestation with its key to all validators. Attestations, keys and blocks reach the validators at about the same time. Once everything has arrived, validators decrypt, merge and execute the slot.

consisting of ν consecutive xblocks; a window of $w = 16$ slots thus corresponds to an Alpenglow leader window of four cblocks. A vote does not need to cover all k proposers: a validator votes for a cblock if all attestations it received agree on the same set of on-time proposers, normally all k of them, but at least a majority, and if it has validated and executed the blocks of exactly these proposers in time. The vote then covers only the blocks of these proposers. While the attestation of some proposer is still missing, the validator holds this vote back for a bounded time, so that colluding proposers cannot exclude a co-proposer whose attestation is still on its way (Section 2.3). If the attestations disagree, or something is missing in time, the validator votes skip.

Pipelining. The proposers do not wait for the votes, but continue with the next slot right away. Validators keep decrypting, merging and executing every slot as it arrives, and vote once per cblock (Figure 3).

Failures. As in Alpenglow, a skipped cblock also skips the rest of the window, and the next window starts right away (Figure 4). A proposer that was crashed from the start of its window, however, does not cause a skip: all its co-proposers agree that it is late, so the attestations still agree, and the vote simply covers the blocks of the other proposers. Since the attestation of the crashed proposer

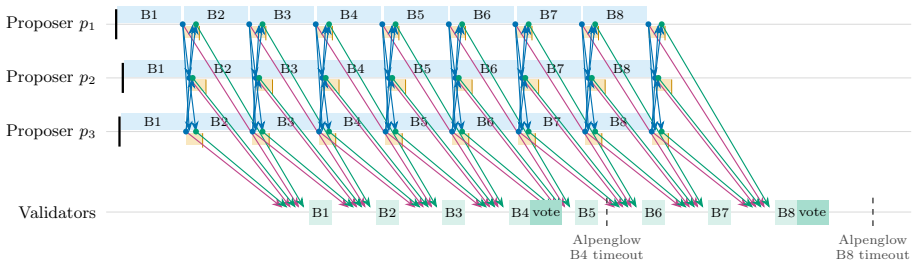


Figure 3: Pipelining: the proposers produce their blocks back to back, without waiting for any votes. With $\nu = 4$, B5 follows right after B4, while the validators are still voting on B1 to B4. Validators decrypt, merge and execute every slot as soon as it arrives, and vote once per cblock, after B4 and after B8, each before its Alpenglow timeout. Each individual slot works as in Figure 2.

never arrives, only the first vote of such a window has to wait for the attestation timeout before voting (Section 2.3).

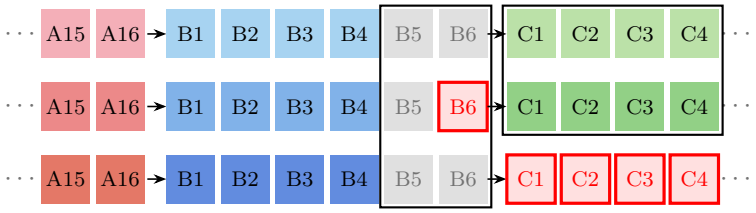


Figure 4: Failure cases, in the same setting as in Figure 1. In this example, colorful blocks are accepted, the red framed block B6 is late or incorrect, and gray blocks are correct but not accepted. B1 to B4 are accepted together by the vote at B4. The left black frame shows the vote on slots B5 and B6, so the skip vote caused by B6 also skips B5, for everybody. B7 and B8 do not stand a chance, since the rest of the window is skipped as well, and the next window starts right away with C1. In this next window, the third proposer has crashed, so its blocks C1 to C4 never arrive (red). Its co-proposers do not give it a positive attestation, and the vote (right black frame) covers only the blocks of the other two proposers, which are accepted.

1.4 Model

Adversarial Model. We assume the Alpenglow model: less than 20% of the stake is byzantine, up to another 20% may be crashed or offline, and equivocation is detectable. Safety holds as long as byzantine stake is below 20%, liveness requires 60% correct and online stake, and fast finalization requires 80%.

Timing Model. Nodes need no synchronized clocks: each timeout is measured with the local clock of a node, and is started by a local event, such as the arrival of a protocol message. This includes the $2\Delta_{\mathsf{local}}$ window of the hash exchange, which every proposer measures with its own clock, starting when it sends its own hash. So we only assume that every node has a local clock, and that these clocks run at about the same rate: over a leader window, their drift is small enough to be absorbed by the slack in Δ_{local} and Δ_{global}.

Network Model. We assume the partial synchrony model, where the protocol maintains safety even during periods of asynchrony and liveness only during long enough periods of synchrony. During synchrony, message delays for messages between arbitrary nodes across the entire network are bounded by Δ_{global}, and between the co-located proposers of a window by the much smaller $\Delta_{\mathsf{local}} \ll \Delta_{\mathsf{global}}$.

Geo-Distribution. We further assume that the validators form local *hubs*, e.g., in the same city, with a network delay of at most Δ_{local} between any two correct nodes of the same hub (Section 3.1). The proposers of a window are always drawn from one such hub (Section 3.1).

2 The Protocol

This section specifies NewCon. We first define the blocks and messages (Section 2.1), then describe what a proposer does (Section 2.2, Algorithm 1) and what a validator does (Section 2.3, Algorithm 2), and finally how NewCon runs on top of Alpenglow (Section 2.4). The reasoning behind some of the design choices follows in Section 3.

2.1 Blocks and Messages

Proposer Blocks. Time is divided into *slots* $s = 0, 1, 2, \ldots$ of $\Delta_{\mathsf{block}} = 50$ ms each, grouped into windows of w slots. The schedule assigns k co-located proposers $p_1, \ldots, p_k$ to every window (Section 3.1). In every slot s of its window, proposer p_q builds one *proposer block* $b_{s,q}$.

Execution Blocks. The accepted proposer blocks of slot s form its *execution block* (*xblock*) $x_s = (b_{s,q})_{q \in P_s}$, where $P_s \subseteq \{1, \ldots, k\}$ is the set of accepted proposers, normally all k of them (Section 2.3). The xblocks are merged and executed in order, one every Δ_{block}.

Consensus Blocks. Alpenglow, however, does not see the individual xblocks. It runs unchanged on *consensus slots* $i = 0, 1, 2, \ldots$, numbered just as Alpenglow numbers its slots. Consensus slot i consists of ν slots $\nu i, \ldots, \nu i + \nu - 1$, and its

consensus block (*cblock*) is

$$c_i = (i_{\mathsf{parent}}, h(c_{i_{\mathsf{parent}}}), x_{\nu i}, \ldots, x_{\nu i+\nu-1}, h_{\mathsf{state}}),$$

$$\text{e.g., } c_i = (i-1, h(c_{i-1}), x_{4i}, x_{4i+1}, x_{4i+2}, x_{4i+3}, h_{\mathsf{state}}) \text{ for } \nu = 4.$$

Here, $c_{i_{\mathsf{parent}}}$ is the parent of c_i (see below), and h_{state} is the hash of the state after executing c_i. Validators vote only once per consensus slot, i.e., every ν slots, on all ν xblocks at once. We suggest $\nu = 4$, so transactions are included every 50 ms, with only one vote every 200 ms. Whenever we use Alpenglow's notions of blocks, votes, certificates and ParentReady, they refer to cblocks and consensus slots; a plain *slot* always means a 50 ms slot. In particular, a NewCon window of $w = 16$ slots is an Alpenglow leader window of $w/\nu = 4$ consensus slots. Hence, a single Alpenglow instance decides a single chain of cblocks, at most one per consensus slot, and each decision covers up to νk proposer blocks.

Proposer Block Content. A proposer block $b_{s,q}$ consists of a header and an encrypted payload. The header contains the slot s and the proposer index q; in the first slot of a window, it also names a parent cblock (see below). The payload contains the transactions. A proposer block is not sent at once, but streamed out in slices throughout its slot, as in Alpenglow (Section 2.2). The payloads of all slices of a block are encrypted under the same fresh symmetric key $K_{s,q}$, one key per block.

Parent. Only the first cblock c_{i_0} of a window has a choice of parent. The first slot of the window is slot νi_0, since consensus slot i_0 consists of the slots $\nu i_0, \ldots, \nu i_0 + \nu - 1$. In the header of its block of this slot, each proposer names the cblock for which it emitted ParentReady (slot and hash), normally the last cblock of the previous window. The co-proposers might name different parents; the parent of c_{i_0} is the one named most often by the accepted blocks of slot νi_0, with ties broken by lower proposer index. Every later cblock c_i of the window has no choice: as in Alpenglow, its parent is always c_{i-1}. So its proposer blocks name no parent; instead, the hash $h(c_i)$ includes $h(c_{i-1})$ (Section 2.4), and the cblocks of a window form a hash chain.

Commitments. The *commitment* $h_{s,q} = h(b_{s,q})$ of a proposer block is the hash of the block as it is disseminated, i.e., with its payload encrypted. The commitment must also bind the key $K_{s,q}$, e.g., by including $H(K_{s,q})$ in the header, so that p_q cannot later reveal a different key that decrypts the same ciphertext to a different payload (Theorem 2.3). Proposer p_q sends its commitment to its co-proposers in the signed message $\mathsf{Commit}(s, q, h_{s,q})$.

Attestations. The *attestation* of proposer p_q for slot s is the signed message $\mathsf{Attestation}(s, q, \tau, K_{s,q})$. The vector τ has one entry per proposer: $\tau[j] = h_{s,j}$ if p_q received the commitment of p_j on time (Section 2.2), and $\tau[j] = \perp$ otherwise; p_q always counts itself as on time. So an attestation states which proposers

Algorithm 1 Proposer p of the window starting with consensus slot i_0. ParentReady is the event of its own Alpenglow instance and validator (Algorithm 2).

```
1: P                                                          ▷ proposers for consensus slot i_0
2: start[s] ← ∞, done[s] ← false                              ▷ for all slots s
3: received[s][j] ← ⊥, proposal[s][j] ← ⊥                     ▷ for all slots s and proposers j ∈ P

4: upon the first of ParentReady(i_0, ·) or its validator executing slot νi_0 − 1 do
5:     for i = i_0, ..., i_0 + w/ν − 1 do                     ▷ back to back, without waiting for votes/certs
6:         for s = iν, ..., iν + ν − 1 do                     ▷ back to back, without voting
7:             K_s ← fresh symmetric key
8:             b ← pack transactions, encrypt payload with K_s
9:             stream the slices of b through Rotor during the next Δ_block
10:            if s = i_0ν then                               ▷ first slot of the window
11:                wait for ParentReady(i_0, h(c_j)), keep streaming until Δ_block after it
12:                b.parent ← (j, h(c_j))                     ▷ parent c_j, in the last slice
13:            end if
14:            start[s] ← min(now, start[s])
15:            proposal[s][p] ← h(b)
16:            send signed Commit(s, p, h(b)) to all co-proposers, a few times   ▷ redundancy
17:            if s is the last slot of the window then send b directly to the next proposers
18:        end for
19:    end for

20: upon receiving first Commit(s, q, h) from a co-proposer q do
21:    start[s] ← min(now, start[s])
22:    received[s][q] ← now
23:    proposal[s][q] ← h

24: upon ¬done[s] and proposal[s][p] ≠ ⊥ and (received[s][q] ≠ ⊥ for all q ≠ p or now >
    start[s] + 2Δ_local) do
25:    done[s] ← true
26:    τ = [⊥, ..., ⊥]
27:    for q ∈ P do
28:        if q = p or received[s][q] ≤ start[s] + 2Δ_local then      ▷ received[s][q] = ⊥ → late
29:            τ[q] ← proposal[s][q]
30:        end if
31:    end for
32:    send signed Attestation(s, p, τ, K_s) to all validators
```

were on time and to which blocks they committed, and it reveals the key of the attester's own block. We call $P(\tau) = \{j : \tau[j] \neq \bot\}$ the set of proposers *attested* by τ, and two attestations *agree* if they have the same vector τ. A proposer *equivocates* if it signs two different proposer blocks, commitments or attestations for the same slot; equivocation is slashable.

2.2 Proposer

Algorithm 1 describes a proposer p, and Figures 2 and 3 illustrate it.

Window Start. A proposer starts its window when its local Alpenglow instance emits ParentReady for the first consensus slot i_0 of the window, or already

earlier with the fast leader handover described below. Its co-proposers emit their ParentReady at about the same time, since they are co-located and receive the same certificates. This difference is bounded by Δ_{local}, because proposers forward the first certificates they construct or receive that triggered their ParentReady directly to their co-proposers (as well as all other validators) as part of regular Alpenglow operation.

Block Production. From then on, the proposer produces its blocks for all w slots of its window back to back, one every Δ_{block}, without waiting for any votes or certificates (Figure 3). In every slot s, it generates a fresh key K_s and builds its block b_s from the transactions that users sent to it (Section 3.2). Throughout the slot, it streams the block out in slices, as in Alpenglow: every slice is encrypted with K_s and disseminated via Rotor as soon as it is ready.

Dissemination. Each slice is erasure coded into $\Gamma = 64$ shreds, any $\gamma = 32$ of which reconstruct it, as in Alpenglow. With shreds of about 1.2 KB, a slice carries about 38 KB, small enough to send at least one slice per 50 ms slot. Rotor relays are sampled separately for each slot, slice index, and even for each co-proposer.

Hash Exchange. Together with the last slice of b_s, the proposer sends its signed commitment $\mathsf{Commit}(s, p, h(b_s))$ directly to its co-proposers. For every slot s, it starts the hash exchange at time $\mathsf{start}[s]$, measured with its own clock: the earlier of the time when it sends its own commitment and the time when it receives the first commitment of a co-proposer for slot s. A co-proposer is *on time* if its commitment arrives by $\mathsf{start}[s] + 2\Delta_{\mathsf{local}}$, and *late* otherwise. Since $\mathsf{start}[s]$ is never later than its own sending time, a proposer waits at most $2\Delta_{\mathsf{local}}$ after sending its own commitment, as in Figure 2.

Robust Hash Exchange. The commitments are the only messages of the protocol without built-in redundancy. All other messages are fault tolerant by nature: slices are erasure coded and disseminated via Rotor, and attestations and votes go to all validators, so a few lost messages do not matter. A commitment, in contrast, is a single message from one proposer to one co-proposer. If it is lost, the co-proposer considers the sender late although it was on time, and the disagreeing attestations may cause a skip. Fortunately, commitments are few: in every slot, each proposer sends only $k - 1$ of them, so they can easily be protected. A simple measure is to send every commitment several times, a fraction of a millisecond apart, since it suffices if any copy arrives. A better, optional measure is a second round of local communication: a proposer not only sends its own commitment to its co-proposers, but also forwards every commitment it receives to its other co-proposers. For $k = 3$, this is just two more messages per proposer and slot, and a commitment is only missed if both its direct and its forwarded copy are lost. The price is latency: a forwarded commitment needs a second hop, so with this option the waiting interval grows

from $2\Delta_{\mathsf{local}}$ to $3\Delta_{\mathsf{local}}$, i.e., a co-proposer is on time if its commitment arrives by $\mathsf{start}[s] + 3\Delta_{\mathsf{local}}$. The rest of this paper describes the protocol without this option, with a waiting interval of $2\Delta_{\mathsf{local}}$.

Attestation. As soon as the commitments of all co-proposers have arrived, but at the latest at $\mathsf{start}[s] + 2\Delta_{\mathsf{local}}$, the proposer sends its signed attestation $\mathsf{Attestation}(s, p, \tau, K_s)$ directly to all validators. It never does so before it has sent its own commitment, even if the commitments of all co-proposers arrive earlier. Only now, in the attestation, does it reveal its key.

Proposal Hiding. If the block of one proposer reached its co-proposers without any delay, the slowest proposer could get a last look, with the option of changing its own block based on the blocks of all others. This is prevented by encrypting the blocks and tying the release of the keys to the attestations. When a proposer releases its key, it has already committed to its own block, and its block can only be accepted together with the blocks whose commitments it attests, which were all fixed before its payload was revealed (Theorem 2.3). This comes at essentially no cost (just some additional bytes in the attestation), except for the execution delay inherent in not revealing the block contents before every proposer has committed.

Fast Leader Handover. To improve performance, in Alpenglow the leader sends its last block directly to the next leader. Likewise, in NewCon the proposers send the full cblock of the last consensus slot of the window directly to the proposers of the next window, so that these learn the latest transactions sooner and do not include them again. The next proposers thus usually hold the last cblock c_{i_0-1} of the previous window before the certificate for it arrives. As in Alpenglow, they do not wait for these certificates: a proposer starts to build and stream the first block of its window as soon as its own validator has executed c_{i_0-1}, or when it emits $\mathsf{ParentReady}$, whichever happens first. It still completes this block Δ_{block} after $\mathsf{ParentReady}$, exactly as without the early start, so the co-proposers complete their first blocks at most Δ_{local} apart, no matter when each of them started. Only the first block of the window depends on the parent, and its parent reference, although part of the header, is sent with its last slice. So a proposer that started early never has to switch parents, and validators lose nothing, since they cannot decrypt the block before its attestation anyway.

2.3 Validator

Algorithm 2 describes a validator v. A validator receives the shreds of all proposer blocks via Rotor, and the attestations directly from the proposers.

Validity. A proposer block $b_{s,q}$ is valid if

Algorithm 2 Validator. Alpenglow runs unchanged on consensus slots and cblocks, except that its timeouts and its Block events come from this algorithm.

$P[s], x[s], \Sigma[s] \leftarrow \bot$ ▷ for all slots s
$b[s][q], V[s][q] \leftarrow \bot$; $\mathsf{eq}[s][q] \leftarrow \mathsf{false}$ ▷ for all slots s and proposers q
$\mathsf{voted}[i] \leftarrow \bot$ ▷ for all consensus slots i

upon $\mathsf{ParentReady}(i_0, \cdot)$ **do**
  for $i = i_0, \ldots, i_0 + w/\nu - 1$ **do**
    $\mathsf{Timeout}(i) \leftarrow \mathsf{now} + \Delta_{\mathsf{timeout}} + (i - i_0 + 1)\nu\Delta_{\mathsf{block}} + \Delta_{\mathsf{block}}$
    $\mathsf{Deadline}(i_0) \leftarrow \mathsf{now} + \Delta_{\mathsf{block}} + 2\Delta_{\mathsf{global}} + 2\Delta_{\mathsf{local}}$
  end for

upon $\mathsf{now} = \mathsf{Deadline}(i_0)$ **do** ▷ stop waiting for missing attestations
  TryVote(i_0)

upon receive new shred of proposer q for slot s from Rotor **do**
  if saw equivocation evidence **then** $\mathsf{eq}[s][q] \leftarrow \mathsf{true}$
  if reconstructed a valid block $b_{s,q}$ of q for slot s **then**
    $b[s][q] \leftarrow b_{s,q}$
    TryExecute(s)
  end if

upon $V_{s,q} = \mathsf{Attestation}(s, q, \tau, K)$ **do**
  if $V[s][q] \neq \bot$ **then**
    if $V_{s,q} \neq V[s][q]$ **then** $\mathsf{eq}[s][q] \leftarrow \mathsf{true}$ ▷ equivocation, slashable
  else
    $V[s][q] \leftarrow V_{s,q}$
    TryExecute(s); TryVote($\lfloor s/\nu \rfloor$)
  end if

procedure TryExecute(s)
  $i \leftarrow \lfloor s/\nu \rfloor$; $A \leftarrow$ Agreed(s)
  if $P[s] \neq \bot$ **or** $A = \bot$ **or** $b[s][q] = \bot$ **for some** $q \in A$ **then return**
  if s is not the first slot of its window **and** $P[s-1] \neq A$ **then return**
  $x[s] \leftarrow (b_{s,q})_{q \in A}$
  $T_q \leftarrow \mathsf{Dec}_{V[s][q].K}(b[s][q])$ **for all** $q \in A$
  if some T_q is invalid **or** $h(b[s][q]) \neq V[s][q].\tau[q]$ **for some** $q \in A$ **then return**
  if s is the first slot of its window **then**
    $\pi \leftarrow$ the most common $b[s][q].\mathsf{parent}$ over $q \in A$, resolve ties to the lowest q
    if $\Sigma[\pi] = \bot$ **then return** ▷ parent not executed yet
    $\sigma \leftarrow \Sigma[\pi]$; $\mathsf{par}[i] \leftarrow \pi$
  else
    $\sigma \leftarrow \Sigma[s-1]$; **if** $s = i\nu$ **then** $\mathsf{par}[i] \leftarrow (i-1, h(c_{i-1}))$
  end if
  $\Sigma[s] \leftarrow \mathsf{Execute}\big(\sigma, \mathsf{Merge}(s, (T_q)_{q \in A})\big)$ ▷ e.g. Algorithm 4
  $P[s] \leftarrow A$
  if $s = i\nu + \nu - 1$ **then** TryVote(i)
  TryExecute($s+1$) ▷ slot $s+1$ may be complete already
end procedure

Algorithm 2 Validator (continued).

```
43: procedure TryVote(i)                                   ▷ emits Block for c_i, at most once
44:     if voted[i] ≠ ⊥ or P[s] = ⊥ or Agreed(s) ≠ P[s] for a slot s of c_i then return
45:     first ← i is the first consensus slot of its window
46:     if first and V[iν][q] = ⊥ for some q and now < Deadline(i) then return      ▷ wait
47:     if not first and voted[i − 1] = ⊥ then return                               ▷ vote in order
48:     voted[i] ← P[iν]
49:     c_i = (par[i], (x_s)_{s∈{iν,...,iν+ν−1}}, h(Σ[iν + ν − 1]))                 ▷ with the state hash
50:     emit Block(i, h(c_i), par[i])
51:     TryVote(i + 1)                                                              ▷ c_{i+1} may be waiting for this vote
52: end procedure

53: function Agreed(s)                                     ▷ the accepted proposers of slot s, or ⊥
54:     if V[s][q] = ⊥ for all q, or eq[s][q] for some q then return ⊥
55:     if V[s][q].τ ≠ V[s][q′].τ for some V[s][q], V[s][q′] ≠ ⊥ then return ⊥      ▷ all agree
56:     A ← {p : τ[p] ≠ ⊥} with τ = V[s][q].τ for some V[s][q] ≠ ⊥
57:     if |A| ≤ ⌊k/2⌋ or V[s][q] = ⊥ for some q ∈ A then return ⊥
58:     return A
59: end function
```

(i) all shreds are valid, signed by p_q, their Merkle roots match, and together with the reconstructed shreds they reproduce the Merkle root included in the received shreds,

(ii) the block is complete and was received before the timeout of its consensus slot,

(iii) it is the only block of p_q for slot s (otherwise p_q equivocated),

(iv) the block hashes match the ones the attestations attest to,

(v) its payload decrypts under the key $K_{s,q}$ of p_q's attestation, and the key matches the commitment,

(vi) if it is a block of the first slot of a window, its parent reference is correct (Section 2.1), and

(vii) it respects size, format and compute limits.

Agreement. A validator considers slot s *agreed* on the set P_s if

(a) all attestations for slot s it received agree, and P_s is the set attested by them,

(b) it holds the attestation of every proposer in P_s,

(c) P_s is a strict majority of the proposers, i.e., $|P_s| > \lfloor k/2 \rfloor$, and

(d) it has seen no equivocation of any proposer in slot s.

Normally, P_s contains all k proposers. A proposer that has crashed is attested as late by all its co-proposers, so the attestations still agree, and P_s consists of the others. With $k = 3$, NewCon thus tolerates one crashed proposer, while with $k = 2$ a crash leaves no majority. A slot can be agreed while the attestation of a proposer outside P_s is still missing; if this attestation arrives later and disagrees, the slot is no longer agreed.

Execution. Once slot s is agreed on P_s and the validator holds valid blocks of all proposers in P_s, it decrypts these blocks with the keys from their attestations, merges their transactions into one deterministic order (Section 3.3), and executes the resulting xblock x_s. The blocks of proposers outside P_s are ignored. The slots are executed in order, each on the state after the previous slot, and the first slot of a window on the state after the parent cblock (Section 2.1).

Timeouts. As in Alpenglow, a validator sets the timeouts of all consensus slots of a window when it emits ParentReady for the first consensus slot i_0 of the window. Compared to Alpenglow, a validator needs a bit more time: the attestations are delayed by the local hash exchange of up to $2\Delta_{\mathsf{local}}$, and execution takes up to Δ_{block} once the last block and attestation of a cblock have arrived. The $2\Delta_{\mathsf{local}}$ come for free as long as $2\Delta_{\mathsf{local}} \leq \Delta_{\mathsf{global}}$, since the attestations go directly to the validators and thus save a hop of up to Δ_{global} compared to the blocks, which take two hops via Rotor. So only Δ_{block} is added to the Alpenglow timeout: the timeout of consensus slot i expires

$$T(i) = \Delta_{\mathsf{timeout}} + (i - i_0 + 1)\Delta_{\mathsf{cblock}} + \Delta_{\mathsf{block}}, \quad \text{with } \Delta_{\mathsf{cblock}} = \nu\Delta_{\mathsf{block}},$$

after ParentReady. More fine-grained timeouts in Δ_{block}-increments would be possible at the cost of some additional complexity, by blurring the line between slots and consensus slots.

Voting. A validator casts a notarization vote $\mathsf{NotarVote}(i, h(c_i))$ for consensus slot i if, before the timeout $T(i)$, all ν slots of c_i are agreed on the same set P, and it has validated and executed the blocks of all proposers in P; the hash $h(c_i)$ covers the hash h_{state} of the resulting state. Otherwise, it casts a skip vote $\mathsf{SkipVote}(i)$. In particular, a validator votes skip if two attestations disagree, e.g., because one proposer considers another late; it never votes for a subset of the attested blocks.

If P does not contain all k proposers, the attestation of a proposer outside P may still be missing, and the validator cannot tell whether this proposer is offline or whether its attestation is still on its way. In this case, the validator executes the slots without delay, but holds back its vote for the first cblock of the window until it has received the attestations of all k proposers for the first slot of the window, at the latest until $\Delta_{\mathsf{block}} + 2\Delta_{\mathsf{global}} + 2\Delta_{\mathsf{local}}$ after ParentReady. By then, the attestation of every correct proposer for this slot has arrived: Its ParentReady is at most Δ_{global} later than that of the validator, it completes its first block Δ_{block} later and attests at most $2\Delta_{\mathsf{local}}$ after that, and the attestation takes at

most Δ_{global}. So colluding proposers cannot exclude a correct co-proposer by attesting only each other (Theorem 2.2). If an attestation that disagrees arrives in the meantime, the validator votes skip. The wait is needed only once per window: the set cannot change after the first slot, so a validator that has voted for c_{i-1} of the same window votes for c_i right away. The wait is also shorter than the timeout, since attestations do not travel via Rotor: in a window with an offline proposer, the first cblock is voted on at this deadline, and the later ones as soon as they are executed. This way, with the proposed parameters (specifically with $w\nu\Delta_{\mathsf{block}} \geq \Delta_{\mathsf{block}} + 2\Delta_{\mathsf{global}} + 2\Delta_{\mathsf{local}}$), the last cblock of the window is finalized no later than it would be in a window with only correct proposers.

2.4 Consensus

Alpenglow on cblocks. NewCon runs a single Alpenglow instance on cblocks. For Alpenglow, c_i is simply the block of consensus slot i, identified by $h(c_i)$. It contains the slot and hash of its parent cblock, the agreed sets P_s and the attested commitments of all ν slots of c_i, and the state hash h_{state}; so $h(c_i)$ binds c_i to its parent, to exactly one set of proposer blocks, and to the state after executing them. Only two things differ from Alpenglow: the validity of a cblock, which is given by the voting rule above, and the timeouts, which leave room for the execution (Section 2.3). Votes and certificates are those of Alpenglow, on the hash $h(c_i)$.

Execution before Voting. Validators vote on the post-execution state: the hash $h(c_i)$ in a notarization vote covers the hash h_{state} of the state after executing c_i, and a notarization certificate thus certifies not just the proposer blocks, but also the resulting state. This is possible since all proposer blocks of a consensus slot are decided at once: a validator that votes notarize knows the exact set P of accepted proposers, and has executed exactly their blocks. So the proposer blocks of a consensus slot are decided and executed just like a single block in Alpenglow. In Alpenglow with synchronous execution, the leader could even include the resulting state hash in the block it distributes itself, and validators would check it as a validity rule. In NewCon, this is not possible: the state after a slot depends on the blocks of all proposers, and a proposer only learns the blocks of its co-proposers like any other validator. Instead, the state hash is part of the cblock, which the validators assemble themselves. So NewCon allows execution *before* voting, but not execution before finishing block production. This is the main advantage of deciding all proposer blocks of a consensus slot together, compared to MCP designs that decide the proposer blocks independently, and therefore require lazy (or "asynchronous") execution.

2.5 Parameters

In Table 1 we propose a specific choice of parameters, designed to be similar to the Alpenglow protocol parameters and the deployed parameters of the Solana

Parameter	Description	Value
k	concurrent proposers	3
ν	slots per consensus slot	4
w	slots per proposer window	16
Δ_{local}	max. local network delay	10 ms
Δ_{block}	length of a slot	50 ms
Δ_{global}	max. network delay	250 ms

Table 1: NewCon protocol parameters.

blockchain. Specifically, the leader windows stay at 800 ms per window ($w \cdot \Delta_{\mathsf{block}}$). With $k = 3$ proposers we have the smallest number of proposers that can support one crashed proposer. For the proofs to work without adapting the Alpenglow timeouts, we require $2\Delta_{\mathsf{local}} \leq \Delta_{\mathsf{global}}$, which is the case for the provided parameter set.

2.6 Correctness

We treat Alpenglow [KSW25] almost as a black box (with only some slight changes to the definition of the block hash and the preconditions to emitting the local events) and make the following assumptions.

(M) *Model.* The byzantine stake f among the validators is less than 20%. After GST, at least 60% of the stake is correct and online, and messages between correct nodes take at most Δ_{global}. Messages from a correct proposer reach the co-proposers in at most Δ_{local}.

(D) *Determinism.* Merging, transaction ordering and execution are deterministic.

(T) *Timing.* The parameters satisfy $\Delta_{\mathsf{timeout}} \geq 3\Delta_{\mathsf{global}}$ and $2\Delta_{\mathsf{local}} \leq \Delta_{\mathsf{global}}$. After GST, a block of a correct proposer reaches all correct validators via Rotor at most $2\Delta_{\mathsf{global}}$ after the proposer completed it. A correct validator executes a slot within Δ_{block}, once it holds the blocks and attestations of the slot and the state it builds on.

(AS) *Alpenglow safety.* If correct nodes follow the Alpenglow voting rules, the finalized blocks form a single chain. At any point in time the finalized chains of different correct validators are prefixes of one or the other. This holds for every block-validity predicate, regardless of when correct nodes cast their initial votes, and also if correct nodes omit fallback votes.

(AL) *Alpenglow liveness.* Consider it is after GST. (i) If every correct validator eventually casts either a notarization or skip vote for every slot, every slot obtains a notar-fallback or skip certificate at every correct node. (ii) If the

same valid block in a slot reaches all correct nodes before the timeouts, it is finalized. This is especially the case if the leader is correct.

Theorem 2.1 (NewCon correctness)**.** *NewCon is safe and every certified state hash is correct. After GST, every window obtains* ParentReady*. Moreover, a cblock is finalized in every consensus slot of every window that starts after GST, has* $> \lfloor \frac{k}{2} \rfloor$ *correct proposers while all its other proposers are crashed, and whose parent every correct validator has executed when it emits* ParentReady *for the window. Here, a window starts when the first correct node emits* ParentReady *for it, and a crashed proposer sends no messages in the window.*

Proof. Safety. We view the cblock of consensus slot i as the block of a virtual leader, with hash $h(c_i)$, i.e., validators run the Alpenglow voting rules on this block hash. Like a regular block hash in Alpenglow, $h(c_i)$ determines the block and its parent: it covers the hash of the parent cblock, the agreed sets and the attested commitments of all slots of c_i (Section 2.4), and every commitment covers its proposer block. Per consensus slot, a correct validator emits at most one Block event (Algorithm 2, Lines 44 and 50), and thus casts at most one notarization vote. Otherwise, only the validity predicate of blocks and the timeouts differ from Alpenglow, and both only decide whether and when a correct validator casts its initial vote. So (AS) gives a single finalized chain of cblocks. Since a commitment binds both the ciphertext and the key of its proposer block (Section 2.1), this chain binds uniquely to a set of proposer blocks and their payloads across the corresponding slots.

State hashes. With (D), the slots of a chain up to a cblock thus have a unique post-execution state, the result of executing their merged transactions in order; we call its hash the *correct* state hash of the cblock. A correct validator computes h_{state} for c_i by executing the slots of c_i on the state it computed for the parent of c_i (Algorithm 2, Lines 33, 38 and 50), so by induction along the chain, a correct NotarVote is for a cblock that contains the correct state hash. Every notarization and notar-fallback certificate has 60% of the stake and thus contains a correct vote. A correct NotarFallbackVote requires seeing at least $20\% > f$ notarization votes for the same hash, and hence a correct NotarVote. So every certified cblock contains the correct state hash.

ParentReady. We show by induction over the windows that every correct node eventually emits ParentReady for every window. Suppose that every consensus slot before some window has a notar-fallback or skip certificate at every correct node, which is trivially true for the first window. Take the last of these slots with a notar-fallback certificate, or the genesis block if there is none: all consensus slots after it have skip certificates, so every correct node emits ParentReady for the window. All correct validators then set the timeouts of the window (Algorithm 2, Line 6) and cast a notarization or skip vote in each of its consensus slots by these timeouts. By (AL)(i), every consensus slot of the window obtains a notar-fallback or skip certificate at every correct node, which completes the induction.

Finalization. Consider a window as in the theorem, with first consensus slot i_0, and let A be the set of its correct proposers. All times are real times; as in

Section 1.4, we ignore clock drift within a window. Let t_u be the time when a correct node u first emits ParentReady for i_0. By then, u has broadcast the certificates that triggered it, as part of the certificate broadcast in Alpenglow. With (M), $|t_p - t_v| \leq \Delta_{\mathsf{global}}$ for every proposer $p \in A$ and every correct validator v, and $|t_p - t_q| \leq \Delta_{\mathsf{local}}$ for all $p, q \in A$.

Consider the m-th slot of the window. A proposer $p \in A$ completes its first block Δ_{block} after ParentReady, even if it started early, and then produces its blocks back to back, so it completes its block of this slot and sends its commitment at time $\sigma_p = t_p + m\Delta_{\mathsf{block}}$ (Algorithm 1, Lines 9, 11 and 16). It starts the hash exchange at $\mathsf{start}_p \leq \sigma_p$, the earlier of σ_p and the arrival of the first commitment of a co-proposer (Algorithm 1, Lines 14 and 21). Let $\sigma = \min_{q \in A} \sigma_q$, so $\sigma_q \leq \sigma + \Delta_{\mathsf{local}}$ for all $q \in A$. Crashed proposers send no commitments, and correct ones send none before σ, so $\mathsf{start}_p \geq \sigma$. The commitment of $q \in A$ reaches p by $\sigma_q + \Delta_{\mathsf{local}} \leq \sigma + 2\Delta_{\mathsf{local}} \leq \mathsf{start}_p + 2\Delta_{\mathsf{local}}$, i.e., on time (Algorithm 1, Line 28). So all proposers in A send the same attestation vector: it attests exactly the proposers in A, with the commitments of their blocks. Proposer p sends its attestation at the latest at $\mathsf{start}_p + 2\Delta_{\mathsf{local}} \leq \sigma_p + 2\Delta_{\mathsf{local}}$ (Algorithm 1, Lines 24 and 32).

The attestation of p reaches a correct validator v by $\sigma_p + 2\Delta_{\mathsf{local}} + \Delta_{\mathsf{global}}$, and the block of p by $\sigma_p + 2\Delta_{\mathsf{global}}$ by (T). As $\sigma_p \leq t_v + \Delta_{\mathsf{global}} + m\Delta_{\mathsf{block}}$ and $2\Delta_{\mathsf{local}} \leq \Delta_{\mathsf{global}}$, validator v holds the blocks and the attestations of all proposers in A for the m-th slot by time $a_m = t_v + m\Delta_{\mathsf{block}} + 3\Delta_{\mathsf{global}}$. Correct proposers do not equivocate, and nobody else can sign in the name of a proposer, so v never sees an equivocation or any other attestation for a slot of the window: at the latest from a_m on, the m-th slot is agreed on A (Algorithm 2, Lines 54, 55 and 57), the same set in every slot (Algorithm 2, Line 27). The parent of c_{i_0} is named by some $p \in A$ (Section 2.1), which emitted ParentReady for it at t_p; the certificates of p reach v by $t_p + \Delta_{\mathsf{global}} \leq a_1$, and v has executed this parent by assumption (executing it by a_1 would suffice). By (T), v executes a slot within Δ_{block} once it holds its blocks, its attestations and the preceding state. By induction on m, it has executed the m-th slot by $a_m + \Delta_{\mathsf{block}}$: its blocks and attestations are there by a_m, and so is the preceding state, as $a_{m-1} + \Delta_{\mathsf{block}} = a_m$ (for $m = 1$, this is the state of the parent).

Consensus slot i of the window ends with the $j\nu$-th slot of the window, where $j = i - i_0 + 1$. So v has executed c_i by

$$a_{j\nu} + \Delta_{\mathsf{block}} = t_v + 3\Delta_{\mathsf{global}} + j\Delta_{\mathsf{cblock}} + \Delta_{\mathsf{block}} \leq t_v + T(i),$$

i.e., before its timeout, which expires $T(i) = \Delta_{\mathsf{timeout}} + j\Delta_{\mathsf{cblock}} + \Delta_{\mathsf{block}}$ after ParentReady (Section 2.3; Algorithm 2, Line 6); the inequality is $\Delta_{\mathsf{timeout}} \geq 3\Delta_{\mathsf{global}}$ from (T). If some proposers are crashed, their attestations are missing, and v waits for them before its first vote of the window, but at most until $t_v + \Delta_{\mathsf{block}} + 2\Delta_{\mathsf{global}} + 2\Delta_{\mathsf{local}} \leq a_1$ (Algorithm 2, Lines 7 and 46); so in any case, v emits Block for c_i before its timeout (Algorithm 2, Line 50).

By (D), all correct validators obtain the same cblock, including its parent and state hash, and thus the same hash $h(c_i)$. So in every consensus slot of the

window, the same valid block reaches all correct nodes before the timeouts, just as with a correct leader, and (AL)(ii) applies. □

If (D) does not hold (e.g. due to an implementation bug), correct validators compute different state hashes for the same proposer blocks, and thus different cblock hashes, which is detected in the voting phase or when a validator learns the consensus outcome. The conditions on the crashed proposers and on the parent are both needed for finalization. A proposer that crashes within a window may reach only some nodes with its last commitment or attestation, so that the attestations disagree and the rest of the window may be skipped. And since validators execute before they vote, a validator that has not yet executed the parent of a window, e.g., because it has to repair it first, may miss its timeouts.

Next we show that NewCon satisfies the selective-censorship resistance property [GNR25].

Lemma 2.2 (Selective-censorship resistance)**.** *In a window that starts after GST, for every slot, either the proposer blocks of all correct proposers are part of the slot or the corresponding consensus slot is skipped.*

Proof. Let p be a correct proposer of the window, c_{i_0} the first cblock of the window, and v a correct validator; we use the notation of the proof of Theorem 2.1. Proposer p sends its attestation for a slot only after its own commitment, so the attestation attests its own block (Algorithm 1, Lines 24 and 28), and at most $2\Delta_{\mathsf{local}}$ after completing this block, since $\mathsf{start}[s]$ is never later than that (Algorithm 1, Lines 14 and 24). It completes its block of the first slot of the window at $t_p + \Delta_{\mathsf{block}}$, so its attestation for this slot reaches v by $t_p + \Delta_{\mathsf{block}} + 2\Delta_{\mathsf{local}} + \Delta_{\mathsf{global}} \leq t_v + \Delta_{\mathsf{block}} + 2\Delta_{\mathsf{global}} + 2\Delta_{\mathsf{local}}$, i.e., by the deadline of v for missing attestations (Algorithm 2, Line 7).

Now suppose that v emits Block for a cblock c_i of the window whose slots are agreed on a set P that does not contain p. Then v has emitted Block for c_{i_0} before (Algorithm 2, Line 47), and all slots of the window up to c_i are agreed on the same set P (Algorithm 2, Line 27), in particular the first slot of the window. So when v emitted Block for c_{i_0}, all attestations for this first slot that it had received attested exactly P (Algorithm 2, Lines 44 and 55), and that of p was not among them. But at that time, v either held the attestations of all proposers for the first slot, or its deadline had passed (Algorithm 2, Line 46); in both cases it held the attestation of p, a contradiction.

So no correct validator casts a notarization vote for a cblock of the window in which the block of a correct proposer is missing. As in the proof of Theorem 2.1, such a cblock obtains neither a notarization nor a notar-fallback certificate, so its consensus slot can only be skipped. □

Lemma 2.3 (Proposal hiding)**.** *It is impossible for a malicious proposer to craft a proposer block based on any information within the payload of a correct proposer's proposer block and have the attacker and victim block appear in the same cblock.*

Proof. Until a proposer receives the attestation containing the decryption key, they hold at most the full encrypted proposer block of a specific other proposer (Algorithm 1, Line 8), and thus do not learn anything about its payload. Further, the key is only sent in the attestation (Algorithm 1, Line 32), which contains the commitments of the other proposers (including the attacker) to their proposer blocks. So, the attestation of the victim proposer includes either (a) a positive attestation about the attacker proposer of a commitment for a block that was created without knowing the payload of the victim block or (b) a negative attestation about the attacker proposer. By the binding property of the commitment, and since the commitment binds both the ciphertext and the key, the attacker can also not later open the commitment to a different block, especially not one chosen after the commitment was sent. If the victim proposer block is included in the cblock that a correct validator voted notarize for, the attestations of all accepted proposers agree with the victim's (Algorithm 2, Line 55), so that same validator's notarization vote (depending on the case above) covers (a) a block of the attacker proposer which was crafted without knowing the contents of the victim block or (b) no block of the attacker proposer. This holds for all correct validators' notarization votes and thus also for the notarize(-fallback) certificate and thus for the finalized cblock. □

Even though there are multiple proposers, together building a cblock, an effective equivocation on that cblock can be blamed on some misbehaving proposer, as shown below.

Lemma 2.4 (Accountable equivocation)**.** *If two correct nodes vote notarize for a different cblock, there exists proof of equivocation blaming at least one of the proposers between the two correct nodes. A correct proposer is never falsely blamed.*

Proof. For the correct nodes to vote notarize on different cblocks $c_i \neq c'_i$, there must be xblocks $x_s \neq x'_s$, such that one correct node saw unanimous attestations of $> \lfloor \frac{k}{2} \rfloor$ proposers supporting the proposer blocks making up x_s (Algorithm 2, Lines 55 and 57), and similarly for the other block and the other proposer. Thus there is at least one proposer who signed both conflicting attestations over different sets of proposer blocks for the same slot. This proposer can be blamed as misbehaving. A correct proposer would never sign two different attestations for the same slot (Algorithm 1, Line 24), and is thus never blamed by such. □

This only holds because the threshold for the attestations is $> \lfloor \frac{k}{2} \rfloor$. To show this consider the following example, in which the protocol is changed to allow $\leq \lfloor \frac{k}{2} \rfloor$ attestations to decide the block content.

Example 2.5. *Consider a NewCon execution with $k = 2$ proposers. The local network has an intermittent problem, causing both proposers to miss the other's proposal commitment before the deadline, so they each include only their own proposal in their attestation. Further, the global network behaves such that some correct validators see only the one attestation and some only the other, each are considered valid and either set of validators votes on a different cblock made*

up of only one of the proposer blocks each. This is effectively an equivocation in the Alpenglow protocol. Because this happened even though both proposers were honest, neither proposer can clearly be blamed and no strong penalty, like slashing, can be applied.

3 Design Considerations

This section discusses the parts of NewCon that are not core protocol rules, but determine whether the design works in practice.

3.1 Geo-Distribution

NewCon requires the proposers of a window to be co-located (Section 1.4). This requirement looks harmless at first glance, but it has far-reaching consequences.

Co-location. A proposer cannot be alone: a validator that is far away from all other validators cannot be a proposer, since it does not have viable co-proposers. More generally, every place where proposers are drawn needs both enough stake (because stake implies security), and enough independent validators, so that no single operator controls all proposers of a window. Of course, an operator can do a Sybil attack and run several validators to look independent. However, every validator has to pay the validator admission ticket (VAT) of Alpenglow [KSW25], so being Sybil means paying the VAT several times, which discourages such behavior. On the positive side, many validators are flexible about their location: most of them run in data centers, and moving to a data center close to other validators is a modest cost if it makes them eligible as proposers. What matters is the network latency between co-proposers, since they have to exchange their commitments within $2\Delta_{\mathsf{local}}$.

Hubs. We call a group of co-located validators a *hub*. Hubs are not planned: neither the protocol nor a central authority decides where hubs are. A hub forms wherever enough validators gather, and it disappears when they leave. The protocol only sets the conditions that a hub has to fulfill to take part in the proposer schedule. Validators remain free to choose their location.

Hub Registration. Anybody can register a new hub with an on-chain transaction. A hub is registered with a unique, human-readable name (e.g., an airport code such as FRA) and with its location (given as an integer ECEF coordinate triple, as in *Validator Location Registration* (SIMD-0674 [KW26a]). Registering a name requires a deposit, and the name is released if the hub stays inactive for too long. A newly registered hub is not yet active; it only becomes active once enough validators have joined.

Joining a Hub. A validator joins a hub by its name with an on-chain transaction, either as part of its registration or later; a validator that does not join any hub only votes. Leaving and moving between hubs are on-chain transactions as well, and all of them take effect only at an epoch boundary announced in advance, so that schedules are fixed ahead of time, and a cooldown prevents validators from hopping between hubs.

Proposer Stake. Let ρ_i denote the stake of validator i, as a fraction of the total stake. Its *proposer stake* $\hat{\rho}_i$ determines how often it is drawn as a proposer. A validator that is not a member of an active hub has proposer stake $\hat{\rho}_i = 0$ and only votes. For a member of a hub, the proposer stake is usually just its stake, $\hat{\rho}_i = \rho_i$, unless it is too large for its hub: every window needs k distinct proposers, and a single member cannot be more than one, however much stake it holds. Let $\rho_1 \geq \rho_2 \geq \cdots \geq \rho_n$ be the stakes of the members of a hub h, with $n \geq k$. For $j = 0, 1, \ldots, k-1$, let

$$c_j = \frac{1}{k-j} \sum_{i>j} \rho_i$$

be the stake of all but the j largest members, divided by the remaining $k - j$ slots, and let $c = c_j$ for the smallest j with $\rho_{j+1} \leq c_j$; such a j always exists, at the latest for $j = k-1$. The proposer stake of member i is then $\hat{\rho}_i = \min(\rho_i, c)$, and the *hub stake* is $\hat{\rho}_h = \sum_{i \in h} \hat{\rho}_i = kc$. Thus the j largest members are capped at exactly $1/k$ of the hub stake, and they are proposers in every window of the hub, while the other $k - j$ proposers of every window are drawn among the remaining members, proportionally to their stake. For example, with $k = 3$ and stakes of 6%, 2%, 1.2%, and 0.8% of the total stake, the cap is $c = 2\%$: the two largest members are proposers in every window, the third proposer is one of the two small members, in 60% and 40% of the windows, respectively, and the hub stake is 6%. The excess stake of capped members still counts for voting and validator rewards, but not for proposing.

Large Validators. A large validator therefore gains nothing from concentrating its stake in one hub, and is better off by splitting its stake across various hubs in different regions.

Active Hubs. A hub h is *active* if

(i) it has at least $n_{\mathsf{min}} \geq k$ members,

(ii) its hub stake is large enough, $\hat{\rho}_h \geq \rho_{\mathsf{min}}$, and

(iii) its members reach each other within Δ_{local}.

Condition (iii) is enforced by the attestations, and it is individually verifiable: a fast hash exchange proves to one validator that the other one is close, since no message travels faster than light. A member that cannot keep up is marked late by its co-proposers and loses its blocks, so it has every incentive to move to a hub it can actually reach.

Parameters over Time. The hub parameters $(\Delta_{\mathsf{local}}, \rho_{\mathsf{min}}, n_{\mathsf{min}})$ start permissively, to include as many validators as possible initially. We can tighten these parameters over time to improve the user experience.

Who Is Affected? Table 2 shows how much of today's stake is not yet in a place that could form an active hub. A realistic start for today's validators is a generous Δ_{local} of 25 ms; with $\rho_{\mathsf{min}} = 1\%$, only 1.2% of the stake would then not yet be in a hub. With a smaller Δ_{local} and a larger ρ_{min}, we can improve the economic security of the protocol, but more stake would have to move.

	stake not in a hub today			
$\rho_{\mathsf{min}} \setminus \Delta_{\mathsf{local}}$ (distance)	≈2 ms (100 km)	≈5 ms (300 km)	≈10 ms (500 km)	≈25 ms (1000 km)
0.1%	1.9%	1.9%	1.9%	0.3%
0.5%	2.7%	2.7%	2.7%	0.7%
1%	3.7%	3.7%	3.7%	1.2%
2%	6.6%	6.6%	6.6%	1.2%
3%	11.0%	11.0%	6.6%	4.3%
5%	28.2%	28.2%	10.4%	8.8%
7.5%	33.5%	33.5%	22.5%	22.0%
10%	41.3%	41.3%	31.2%	26.7%

Table 2: Share of today's stake that could not yet be part of an active hub, if a hub needs at least two validators that reach each other within Δ_{local} and together hold at least ρ_{min} of the stake. Since we have no latency measurements between all validators, Δ_{local} is approximated by distance, with a generous margin for routing detours, jitter, and processing on top of the roughly 1 ms per 100 km of fiber. Based on the 30 largest validator locations (95.9% of the stake, IP geolocation, validators.solutions, October 30, 2026); the remaining 4.1% at smaller locations are not included. The floor of 1.9% comes from a few isolated sites without any other validator nearby, independently of ρ_{min}. Since hubs grow bottom-up, these numbers describe a starting point, not a permanent exclusion: wherever enough validators gather, a new hub can form.

Candidate Hubs. Even though hubs grow bottom-up, it is interesting to ask where they might emerge. Table 3 ranks candidate hubs by four criteria: importance for traditional finance, importance for crypto, today's stake, and geographic diversity. The list suggests that starting with just 3-4 hubs in New York, Frankfurt, Tokyo and London, might be a good starting point.

Hub Schedule. Every window is assigned to an active hub, and its k proposers are drawn among the hub's members according to their proposer stake (as discussed earlier). If the windows were assigned proportionally to the hub stake $\hat{\rho}_h$ alone, a large hub could dominate the schedule. Instead, a window is assigned proportionally to hub stake only with probability α, and to an active hub chosen uniformly at random otherwise, e.g., with $\alpha = 1/2$. A hub with a fraction x of the total hub stake thus gets a fraction $\alpha x + (1-\alpha)/|H|$ of all

#	Hub	TradFi	Crypto	Stake	Diversity
1	New York	**#1**	US institutions, ETFs, Coinbase, Gemini	9%	—
2	Frankfurt	#15; ECB, Eurex	EU under MiCA	**46%**	6200 km New York
3	Tokyo	#10	**Exchange engines**, Binance, Hyperliquid	5%	9300 km Frankfurt
4	London	**#2**	Institutional desks, options, Deribit	27%	640 km Frankfurt
5	Singapore	#4	#1; **Major Asian hub**, trading firms, Bybit	4%	5300 km Tokyo
6	Hong Kong	**#3**; Shenzhen #9	#3; Licensed exchanges, gateway to China, OKX	0.7%	2600 km Singapore
7	San Francisco	**#5**	Crypto companies, venture capital	4%	4100 km New York
8	Dubai	#7	#4; **Fast-growing hub**, many exchanges	0%	4800 km Frankfurt
9	Seoul	#8	#8; **Large retail volume**, Upbit, Bithumb	0.3%	1150 km Tokyo
10	Chicago	#14, **derivatives**	Crypto futures, trading firms, CME	2%	1150 km New York
11	São Paulo	#98; largest in Latin America	Strong adoption in Latin America	0.4%	**7700 km** New York
12	Mumbai	#52; India's financial capital	**High adoption**, restrictive regulation	0%	1900 km Dubai
13	Sydney	#24; largest in Oceania	Moderate	0%	**6300 km** Singapore
14	Johannesburg	#80; largest in Africa	High adoption in Africa, mostly retail	0%	**6400 km** Dubai
15	Zurich	#11	#2; **Crypto Valley**, Alpenglow HQ	0.5%	300 km Frankfurt

Table 3: Candidate hubs, in a possible order of growth; bold marks the main reason for a hub's position. TradFi is the rank in the Global Financial Centres Index (GFCI 39, Z/Yen, March 2026); Crypto starts with the rank in the Crypto Friendly Cities Index 2026 (Multipolitan) where available, and exchange locations are as reported by traders. *Stake* is the share of today's stake that is closest to the hub (validators.solutions, October 30, 2026); for example, London includes Amsterdam, which could also replace London and Frankfurt as a single European hub. *Diversity* is the distance to the closest hub higher up in the list.

windows, where H is the set of active hubs. Stake in a large hub therefore earns fewer proposer roles than stake in a small hub, which keeps any single hub from dominating the schedule and gives validators an incentive to join smaller hubs. Since every active hub gets at least a fraction $(1-\alpha)/|H|$ of the windows, however little stake it holds, the conditions for an active hub, in particular ρ_{min} and n_{min}, must be strict enough that this share is not cheap to obtain. The windows are ordered with SIMD-0675 [KW26b], with the hubs in the role of the leaders: a run of consecutive windows stays within one region, typically moving between nearby hubs, before the schedule moves on to another region. Handovers within a run cost a few milliseconds, and handovers between regions are the same as

between distant leaders in Alpenglow. For users, a distant hub is no different from a distant leader in Alpenglow; on average, NewCon is not slower, and users near any hub regularly have their proposers nearby.

Proposer Schedule. The hub schedule fixes the hub of every window of an epoch; it remains to draw the k proposers of each window among the members of its hub h. For every window, the members of h are placed in random order on a line of length $\hat{\rho}_h = kc$, where member i covers an interval of length $\hat{\rho}_i$. With a uniformly random offset $u \in [0, c)$, the proposers are the members whose intervals contain the k points $u, u + c, \ldots, u + (k-1)c$, and their proposer indices are assigned at random. Since no interval is longer than c, the k proposers are distinct, and member i is a proposer with probability exactly $\hat{\rho}_i/c$: a capped member is a proposer in every window of its hub, and every other member in proportion to its stake. The proposers of a window are thus always k different members of the same active hub, so they are co-located, and no validator holds more than one of the k roles of a window. Proposer roles are proportional to proposer stake within every hub, but only partly across hubs, since a fraction $1 - \alpha$ of the windows is assigned independently of hub stake. All draws use a deterministic PRNG seeded from the epoch, and stakes and hub memberships are taken from the same snapshot as for the current leader schedule, so that all validators compute the same schedule. Algorithm 3 gives the complete procedure, including the hub schedule.

Algorithm 3 NewCon proposer schedule of an epoch with N windows.

Require: active hubs H; proposer stakes $\hat{\rho}_i$ and hub stakes $\hat{\rho}_h$; PRNG seeded from the epoch
$\omega_h \leftarrow \alpha \cdot \hat{\rho}_h / \sum_{g \in H} \hat{\rho}_g + (1-\alpha)/|H|$ for all $h \in H$
for $i = 1, \ldots, N$ **do** ▷ hub schedule
  $h_i \leftarrow$ random hub from H, where hub h has probability ω_h
end for
$(h_1, \ldots, h_N) \leftarrow$ SIMD-0675$(h_1, \ldots, h_N)$ ▷ reorder, with hubs as leaders
for $i = 1, \ldots, N$ **do** ▷ proposers of window i
  $(m_1, \ldots, m_n) \leftarrow$ shuffle(members of h_i); $c \leftarrow \hat{\rho}_{h_i}/k$
  $u \leftarrow$ uniformly random in $[0, c)$; $s \leftarrow 0$; $W_i \leftarrow ()$
  for $j = 1, \ldots, n$ **do** ▷ m_j covers $[s, s + \hat{\rho}_{m_j})$, the next point is u
    $s \leftarrow s + \hat{\rho}_{m_j}$
    if $u < s$ **then** append m_j to W_i; $u \leftarrow u + c$
  end for
  $W_i \leftarrow$ shuffle(W_i) ▷ proposer indices
end for
return $(W_1, \ldots, W_N)$

3.2 Threshold Transactions

***t*-of-*n* Transactions.** A user sends each transaction to a set of n proposers of the target slot ($1 \le n \le k$) with a threshold $1 \le t \le n$, a *t-of-n transaction*. A larger n buys censorship resistance: the transaction survives $n - t$ receivers that censor it or whose blocks are not accepted. A higher threshold t buys privacy: the content stays temporarily hidden from any $t - 1$ colluding receivers. For

$t = 1$, the plain transaction is sent to every receiver, and duplicates are removed during the merge. For $t \geq 2$, the user follows Krawczyk [Kra94]: it encrypts the transaction with a fresh key, erasure codes the ciphertext into n fragments (any t of which reconstruct it), and splits the key into n Shamir shares with threshold t. Each receiver gets one fragment, one key share, and a public header signed by the fee payer (transaction ID, fee payer, compute limit, writable accounts, priority fee, target slot), which is all that validity, merging, and selection use. Once t fragments are accepted, the transaction is decrypted and executed; until then, its fragments are cached (Section 3.3).

Partial Hiding. This hiding is only partial: it ends once t fragments have been revealed, which can happen before the transaction has reached a point of no return. If all receivers include their fragments in the target slot, this does little harm: parties that together have seen t fragments can only react in later slots, which are ordered after the transaction, so they can back-run but not front-run it. Two cases remain. First, a receiver that holds back its fragment until the other $t - 1$ fragments have been revealed can decrypt the transaction early, and add a front-running transaction to the block that also carries its fragment; with a higher priority fee, this front-running transaction is executed first (Section 3.3). This is only possible within the expiry period of the cached fragments, so a short expiry period limits it, at the cost of less tolerance for skipped slots. Second, if the fragments are disseminated but the slot is skipped, the transaction is revealed without being executed, and it can be front-run in a later slot when it is sent again.

3.3 Transaction Ordering

Transaction Ordering. The transactions of all accepted proposer blocks of a slot are merged into one order, the xblock of the slot, and the xblocks are executed one after the other. After removing duplicates (see below), the transactions of a slot are ordered by priority, as in LegCon [Kni+26]: by decreasing priority fee per compute unit prio, then by increasing compute limit cu, and finally by increasing transaction hash. We write SortTxs(txs) for a list of transactions txs sorted in this order. The order is deterministic, the proposers have no influence on it, and front-running is expensive, since one has to outbid a transaction to get ahead of it. Other orders are possible, e.g., round robin among the proposers, or letting each application define the order of its own transactions, such as a batch auction per market. Application-defined ordering is non-trivial, since a transaction may touch several applications whose preferred orders conflict, and the result must remain composable; Canidio [Can26] proposes to follow an order whenever all affected applications agree on it, and priority ordering otherwise. Alternatively, each application could define a priority function, and a transaction would only be valid if all applications it touches assign it the same priority, so that sorting by this priority still satisfies all of them.

Removing Duplicates. A user may send the same transaction to several proposers ($n > 1$), or send it again to the proposers of a later window, so the merge first removes duplicates:

- *Plain transactions* are identified by the hash of their signed message: copies with the same message are the same transaction, and only one copy is kept.
- *Private transactions* ($t > 1$) are identified by their transaction ID, the commitment in their public header: the fragments that different proposers included are collected under this ID, and the transaction is kept once.
- *Earlier slot wins:* if the same transaction was included in different slots, for example in slots 1 and 3, only the copy in the earlier slot counts. In particular, a transaction that has already been executed in an earlier slot is recognized by its message hash and dropped.
- *Incomplete private transactions:* a private transaction can only be decoded once t of its fragments have been accepted, and these may land in different slots. Validators therefore keep received fragments in a cache, indexed by transaction ID. As soon as the t-th fragment is received in some slot, the transaction is decoded and executed in that slot, ordered like any other transaction of the slot. If fewer than t fragments have been accepted after an expiry period, the cached fragments are discarded.

Slot Capacity. Every transaction declares a compute limit cu and the set acc of writable accounts it touches; if its execution exceeds cu, it is canceled but still pays its fees. A slot is limited by a virtual parallel execution schedule with m execution *tracks* (we suggest $m = 4$), in which each track may use at most b_{cu} compute units; b_{cu} thus bounds the time it takes to replay the slot. Validators fill this schedule deterministically, walking down the merged order (Algorithm 4); Figure 5 shows an example.

- A transaction is placed at the earliest time on any track at which it does not overlap with an earlier transaction (in the merged order) that writes one of the same accounts, and at which it still finishes by b_{cu}. If several tracks are free at that time, the track with the smallest index is used.
- This may leave gaps in the schedule. A later transaction may fill a gap, as long as no conflicting earlier transaction starts after it.

Priority Inversion. Filling gaps can invert the priority order of two conflicting transactions. If $\mathsf{tx1}$ has a higher priority than $\mathsf{tx2}$ but conflicts with many other transactions, including transactions on its own track, $\mathsf{tx1}$ may only fit late in the schedule, while $\mathsf{tx2}$, which has few conflicts, fits into an earlier gap and is executed strictly before $\mathsf{tx1}$. We accept this: $\mathsf{tx1}$ has used its priority fee for its

Algorithm 4 Transaction selection

```
1: function SelectTxs(txs)
2:     txList ← [ ]
3:     spaces ← {(0, b_cu, 1), ..., (0, b_cu, m)}          ▷ free locations: start, end, track
4:     scheduled ← ∅          ▷ scheduled txs: position, length, accounts, track
5:     txs ← SortTxs(txs)                                              ▷ by priority
6:     for tx ∈ txs do                                                 ▷ iterate in order
7:         newSpaces ← ∅
8:         placed ← false
9:         if VM.chargePriorityFee(tx) failed then
10:            placed ← true                          ▷ skip tx (still charge for inclusion)
11:        end if
12:        while not placed and spaces ≠ ∅ do
13:            (start, end, ℓ) ← min spaces
14:            spaces ← spaces \ {(start, end, ℓ)}
15:            stxs ← sortBy(scheduled ∪ {0, 0, ∅}, (pos, len, acc) ↦ pos + len)
16:            for (pos, len, _, _) ∈ stxs do
17:                newPos ← max{start, pos + len}
18:                C ← {(p, l, _, _) ∈ stxs : p + l > newPos ∧ p < newPos + tx.cu}
19:                busyAcc ← {a : (_, _, a, _) ∈ C}
20:                S ← FindSpace(tx, newPos, (start, end, ℓ), busyAcc)
21:                if S ≠ {(start, end, ℓ)} then
22:                    scheduled ← scheduled ∪ {(newPos, tx.cu, tx.acc, ℓ)}
23:                    txList ← txList ∪ {tx}
24:                    placed ← true
25:                    break
26:                end if
27:            end for
28:            newSpaces ← newSpaces ∪ S
29:        end while
30:        if placed then
31:            VM.chargeInclusionFee(tx)
32:        end if
33:        spaces ← spaces ∪ newSpaces
34:    end for
35:    txList ← SortTxs(txList)
36:    return txList
37: end function

38: function FindSpace(tx, pos, (start, end, ℓ), busyAcc)          ▷ returns new spaces
39:    if end − pos < tx.cu or tx.acc ∩ busyAcc ≠ ∅ then
40:        return {(start, end, ℓ)}
41:    end if
42:    S ← ∅
43:    if pos > start then
44:        S ← S ∪ {(start, pos, ℓ)}
45:    end if
46:    if pos + tx.cu < end then
47:        S ← S ∪ {(pos + tx.cu, end, ℓ)}
48:    end if
49:    return S
50: end function
```

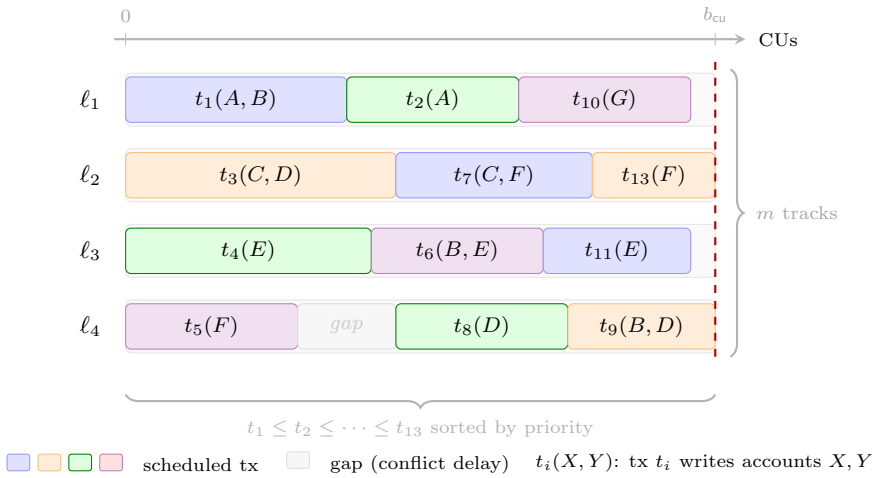


Figure 5: Example transaction selection outcome based on a list of transactions $[t_1, t_2, \ldots]$ (accessing accounts $A, B, \ldots$). The gap before t_8 occurs because it has to wait on t_3, takes precedence over other transactions that could be scheduled immediately after t_5, and the gap is not large enough to fit any later transactions. There is also $\mathsf{tx}_{12}(A, D)$ with $t_{12}.\mathsf{cu} = t_{13}.\mathsf{cu}$, which could not be scheduled because it has to wait on t_9 (both access account D).

position relative to the transactions it conflicts with, and we expect most transactions to care mainly about their most congested accounts. Forbidding such inversions would let a single transaction destroy all the available parallelism.

Deterministic, Not Optimal. The greedy schedule does not always fit the most transactions into a slot. This is unavoidable: already for transactions that touch disjoint accounts and whose compute limits add up to exactly $m \cdot b_{\mathsf{cu}}$, deciding whether they all fit on the m tracks is NP-complete. What matters instead is that the schedule is deterministic and follows the merged order, so all validators select the same transactions, and higher-priority transactions are scheduled first.

Parallel Execution. The virtual schedule only determines which transactions are executed in a slot, not how they are executed. The actual execution may be faster and adaptive: for instance, if a transaction uses fewer compute units than declared, later transactions may start earlier. The only requirement is that the execution respects the merged order for all transactions that write a common account; then the result is independent of how the execution is parallelized.

3.4 Fees

Fees can work much as on Solana today: a base fee, part of which is burned, and a priority fee, which also determines the order of the transactions. In addition, every included copy or fragment of a transaction pays an *inclusion fee* to the proposer whose accepted block carries it. This gives proposers a reason to include transactions, also those that are not executed in the end, and it makes redundant copies ($n > 1$) cost something for the user. We leave the details of the fee mechanism to future work.

Fee Payers. Proposers build their blocks on an outdated view of the chain, so they may include a transaction whose fee payer can no longer pay, or several transactions of the same fee payer that together exceed its balance. To make sure that the inclusion fee can always be paid, we adopt the reserve rule of LegCon [Kni+26]. Every fee payer account has to keep a small reserve φ, e.g., 0.001 SOL, which can only be withdrawn by closing the account, and closing takes effect only two windows later. A proposer includes transactions only once it has executed all slots before the previous window, and it includes transactions of a given fee payer with inclusion fees of at most $\varphi/(2k)$ per window. The inclusions that a proposer cannot yet observe therefore stem from at most two windows, i.e., from at most $2k$ proposers, and together they never exceed the reserve. Validators check this rate, and a proposer block that exceeds it is invalid. During the selection (Algorithm 4), the inclusion fee is always charged, even if the balance drops below the reserve; a transaction is only selected if its priority fee can be charged without touching the reserve; and a transaction whose execution would touch the reserve fails.

4 Possible Extensions

In this section, we discuss possible extensions of NewCon: better crash availability, and Lanes, a variant in which every proposer is decided on its own (Section 4.2). We also give an outlook on a possible future extension in which several hubs around the world, each with multiple proposers, propose concurrently (Section 4.3).

4.1 Crash Availability, Censorship Resistance, BFT

The current choices of NewCon require unanimous attestations from a majority of proposers and voting skip if any disagree, maximize censorship resistance while tolerating crashed but otherwise honest proposers, but give malicious proposers the power to skip a slot. Towards the other extreme the proposers could run a local synchronous one-shot BFT consensus protocol to agree on a set of proposer blocks that will make up the cblock. This would allow a $> 2/3$ super-majority to always propose a block, in the face of a minority of not just crashed but also byzantine proposers. On the other hand, that design gives absolute selective-censorship power to any super-majority coalition of proposers.

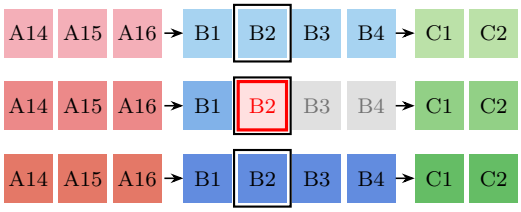


Figure 6: Lanes: every proposer is decided by its own vote (black frames). If the second proposer is late with B2, only this proposer loses B2 and the rest of its window, while the other two proposers continue with B3 and B4.

4.2 Lanes

In Lanes, every proposer runs its own lane, an independent instance of Alpenglow, and every lane is decided by its own votes (Figure 6). The transactions of the accepted blocks of all lanes are then merged as in NewCon. NewCon already leaves out crashed and late proposers through its attestations and the majority rule, so Lanes would only help in three remaining cases. First, a byzantine proposer can deliberately cause disagreeing attestations, e.g., by sending its hash to only some of its co-proposers; in NewCon, this skips the rest of the window for everybody, in Lanes only for its own lane. A local agreement among the proposers (Section 4.1) would address the same problem without Lanes. Second, Lanes tolerate a crashed proposer also for $k = 2$, where the majority rule of NewCon requires both proposers. Third, Lanes do not need co-location for the decision itself, which may help when proposers in several regions propose concurrently (Section 4.3). The price is high: since a validator does not know which lanes of a slot are accepted when it votes, execution becomes asynchronous; there are k consensus instances with k times as many votes; and the protocol and its proofs become considerably more complex.

4.3 Concurrent Hubs

In NewCon, all proposers of a window are located in one place, and over time the windows rotate among the hubs of the world (Section 3.1). A natural long-term extension is to let several hubs propose at the same time: in every slot, a co-located group of proposers in each of, say, New York, Frankfurt, and Tokyo builds blocks, and the blocks of all hubs are merged into one order. Every hub would run the local part of NewCon as before, including its own hash exchange and attestations.

Concurrent hubs would make NewCon truly global: every major region proposes in every slot instead of waiting for its turn, users anywhere have a proposer nearby, and censoring a transaction would require colluding proposers in several jurisdictions.

The details, however, are far from clear. Hubs are Δ_{global} apart rather than Δ_{local}, so the attestations cannot simply span several hubs, and a hub far from

most of the stake would always look late. A hub that sees the blocks of the other hubs before committing to its own could play timing games across hubs, so hiding would need a mechanism that works at global distances. The hubs could be connected like the lanes of Section 4.2, each hub deciding its own blocks, but then execution would become asynchronous again. We consider concurrent hubs a promising direction for future work.

5 Related Work

Maximal Extractable Value. The concept of maximal extractable value (MEV) was introduced and systematically studied by Daian et al. [Dai+20], who documented widespread exploitation of transaction ordering in decentralized exchanges through front-running, sandwich attacks, and priority gas auctions. They showed that block producers can extract substantial profit by reordering, inserting, or censoring transactions, and identified consensus instability as a consequence: once MEV revenues are large enough, rational validators have an incentive to deviate from the protocol to capture them. Zhou et al. [Zho+23] systematize DeFi attacks, many of which are enabled or amplified by MEV, and Heimbach et al. [HW22] give a taxonomy of transaction reordering manipulations and the defenses proposed against them. The harm is twofold: users suffer direct losses through adverse ordering, and large MEV opportunities give validators incentives to deviate from the protocol, eroding both security and fairness.

Proposer-Builder Separation. The dominant practical response to MEV has been proposer-builder separation (PBS) [But21], which accepts MEV as inevitable and instead democratizes access to it: specialized builders compete to construct the most profitable block, and proposers select the highest bid. PBS was deployed on Ethereum via MEV-Boost [Fla22]; Heimbach et al. [Hei+23] analyze its achievements and limitations in practice, including persistent centralization in the builder market. The fundamental limitation of PBS is structural: it optimizes and redistributes MEV rather than removing the harm to users. Transaction fee mechanisms in this setting were studied by Roughgarden [Rou21], who analyzes the incentive properties of EIP-1559 [But+19], and by Bahrani, Garimidi, and Roughgarden [BGR24], who examine which guarantees remain achievable in the presence of MEV. Stouka, Ma, and Thiery [SMT25] extend this framework to censorship under bribery with multiple proposers and inclusion lists. These works clarify the tension between revenue maximization and user protection, but do not provide protocol mechanisms that remove the leader's monopoly over inclusion and ordering.

Order Fairness. A parallel line of work defines and enforces order fairness as a property of consensus. Kelkar et al. [Kel+20] introduced receive-order fairness: if a supermajority of nodes observe transaction t_1 before t_2, then t_1 should be confirmed before t_2; they showed that this property is in tension with liveness

and yields impossibility results in some settings. Themis [Kel+23] achieves a practical variant, batch order-fairness, in which nodes vote on the relative order of transactions and batches are cleared with a social-choice rule. These approaches address the ordering dimension of MEV, but not selective censorship, in which a leader simply excludes targeted transactions. NewCon addresses both dimensions structurally: no single proposer decides which transactions are included, and the transactions of all proposers are merged by a deterministic rule, so that no single proposer controls the order (Section 3.3); applications can additionally define their own order [Can26].

Censorship Resistance. A growing body of work provides formal censorship-resistance guarantees by restructuring who assembles block content. Fox, Pai, and Resnick [FPR23] formalize censorship resistance in on-chain auctions as the cost an adversary must incur to delay a transaction as a function of its tip, and show that a single proposer with exclusive ordering power cannot satisfy censorship resistance and rational participation at the same time, which motivates architectures that dilute proposer power.

Multiple Concurrent Proposers. Multiple concurrent proposers (MCP) are the central architectural response: block content is sourced from several proposers at once, so that no single party can censor or reorder all submitted transactions. Garimidi, Neu, and Resnick [GNR25] give a rigorous treatment of MCP and show that it can achieve selective censorship resistance and hiding with a single additional round trip over a standard consensus protocol, as a gadget on top of existing consensus. Abraham, Efron, and Ren [AER25] formalize censorship-resistant Byzantine broadcast and prove a lower bound on its latency overhead, showing that this one-round overhead is asymptotically optimal. Marsh, Ranchal-Pedrosa, and Resnick [MRR26] systematize MCP protocols into four families (fixed proposal vectors, selected proposal collections, persistent proposal streams, and causal proposal DAGs) and contrast them with constrained assembly, in which several parties restrict a block assembled by a single party. Their comparison also shows that additional proposers alone do not guarantee inclusion, and that hiding does not settle who can react last. Several further MCP and censorship-resistant designs appeared recently. Cadence [Bab+26] runs an independent consensus instance per slot with several proposers each ("extreme pipelining"), and provides short-term censorship resistance and hiding at the fast-path latency of single-leader consensus. AMP [Cas+26] is a multi-proposer layer for BFT consensus with bounded inclusion: a payload attested by sufficiently many validators at one height is finalized at the next. Prefix Consensus [XTS26] relaxes consensus so that outputs only need to extend the maximum common prefix of the honest inputs, and builds a leaderless BFT protocol from it in which, after GST, at most f slots miss an honest proposal. Gatling [SRN26] runs several staggered instances of an atomic broadcast protocol and merges their outputs with a deterministic interleaving rule, similar in spirit to Lanes, to make the time between propos-

als arbitrarily small. Sedna [Ran+25] addresses how users send transactions to concurrent proposers, using rateless coding instead of full replication, and offers until-decode privacy, a notion of hiding close to that of our t-of-n transactions (Section 3.2). Sei Giga [MLJ25] is a multi-proposer EVM blockchain built on Autobahn: proposals are made available first, consensus orders them, and blocks are executed asynchronously afterwards, whereas NewCon executes before voting. Landers and Marsh [LM25] analyze MEV channels specific to MCP, such as duplicate steals within a tick, auctions between proposers, and timing races, and argue that deterministic priority scheduling and duplicate-aware payouts mitigate them; NewCon also uses a deterministic priority order. On Ethereum, fork-choice enforced inclusion lists (FOCIL) [Thi+24] let a committee of validators force transactions into blocks, a lighter-weight route to censorship resistance that keeps a single block builder. On the economic side, Saraf et al. [Sar+26] quantify censorship resistance and throughput with rational concurrent proposers, and Moallemi et al. [Moa+26] study where proposers choose to locate, with a single and with concurrent proposers, which relates to our co-located proposers and to the geo-aware schedule of SIMD-0675 [KW26b].

Constellation. LegCon [Kni+26] is an MCP protocol for Solana that runs as a preprocessor to Alpenglow [KSW25]: proposers send their transactions to attesters, which timestamp them and forward them to a leader, and the leader must assemble each batch from the attested transactions. This yields a protocol-enforced economic tick of 50 ms, at the cost of an additional layer of attesters, synchronized clocks, and a leader that, although tightly constrained, still assembles the block. NewCon has neither a leader nor attesters: the k proposers of a slot produce Alpenglow blocks themselves, and co-located proposers replace synchronized clocks. In the terms of Marsh, Ranchal-Pedrosa, and Resnick [MRR26], LegCon is a form of constrained assembly, while NewCon is closest to a fixed proposal vector, with k proposals per slot. Table 4 compares the two protocols along the properties of Section 1.2.

Batch Auctions. That discrete clearing intervals reduce manipulation is not unique to blockchains. Budish, Cramton, and Shim [BCS15] show that continuous limit order books create a high-frequency-trading arms race in which speed rather than information drives profits, and propose frequent batch auctions, in which orders are collected and cleared at regular intervals, removing the benefit of sub-interval latency advantages. The idea was soon transferred to blockchains, e.g., [Kur19], and Canidio and Fritsch [CF23] show that batching trades on automated market makers reduces the value extractable through ordering. These insights motivate the short slots of NewCon: the transactions of all proposers are merged slot by slot, every $\Delta_{\mathsf{block}} = 50$ ms (Section 2.1), so that what counts is the slot a transaction lands in, not its exact arrival time.

Property	NewCon	LegCon
MEV resistance		
Leaderless	yes	no, leader assembles block
Censorship resistance	yes	yes
Deterministic ordering	yes, priority fee	yes, priority fee
Block hiding	yes, commit–reveal	no
Transaction hiding	yes, t-of-n	no
Accountability	lateness, equivocation	equivocation
Performance		
Shorter blocks	yes, 50 ms	yes, 50 ms
No extra latency	yes	no, attester hop
High bandwidth	yes	no, data sent twice
No slot stretching	yes, local check	yes, synced clocks
Robustness and simplicity		
Generalizes Alpenglow	yes	yes, preprocessor
Execution before voting	yes	yes
No synchronized clocks	yes	no
Crash tolerance	yes	proposers only

Table 4: The properties of Section 1.2 in NewCon and LegCon.

References

[AER25] Ittai Abraham, Yuval Efron, and Ling Ren. *The Latency Cost of Censorship Resistance*. Cryptology ePrint Archive, Paper 2025/2136. 2025. URL: https://eprint.iacr.org/2025/2136.

[Bab+26] Kushal Babel et al. *Cadence: Extreme Pipelining with Multiple Concurrent Proposers*. 2026. arXiv: 2607.02275 [cs.DC].

[BGR24] Maryam Bahrani, Pranav Garimidi, and Tim Roughgarden. *Transaction Fee Mechanism Design in a Post-MEV World*. Cryptology ePrint Archive. 2024.

[BCS15] Eric Budish, Peter Cramton, and John Shim. "The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response". In: *The Quarterly Journal of Economics* 130.4 (2015), pages 1547–1621. DOI: 10.1093/qje/qjv027.

[But+19] Vitalik Buterin et al. *EIP-1559: Fee Market Change for ETH 1.0 Chain*. https://eips.ethereum.org/EIPS/eip-1559. 2019.

[But21] Vitalik Buterin. *Proposer/Builder Separation-friendly Fee Market Designs*. Ethereum Research forum post. 2021.

[Can26] Andrea Canidio. *Ordering by Unanimity: Giving Applications Sequencing Rights Without Breaking Composability*. arXiv preprint arXiv:2607.06144. 2026.

[CF23] Andrea Canidio and Robin Fritsch. "Batching Trades on Automated Market Makers". In: *5th Conference on Advances in Financial Technologies (AFT)*. 2023.

[Cas+26] Daniel Cason et al. *AMP: Arc Multi-Proposer Protocol with Bounded Inclusion Guarantees*. arXiv preprint arXiv:2605.23677. 2026.

[Dai+20] Philip Daian et al. "Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges". In: *IEEE Symposium on Security and Privacy (SP)*. 2020. DOI: 10.1109/SP40000.2020.00040.

[Fla22] Flashbots. *MEV-Boost: Proposer-Builder Separation for Ethereum*. https://boost.flashbots.net. 2022.

[FPR23] Elijah Fox, Mallesh M. Pai, and Max Resnick. "Censorship Resistance in On-Chain Auctions". In: *5th Conference on Advances in Financial Technologies (AFT)*. Volume 282. LIPIcs. 2023, 19:1–19:18. DOI: 10.4230/LIPIcs.AFT.2023.19.

[GNR25] Pranav Garimidi, Joachim Neu, and Max Resnick. "Multiple Concurrent Proposers: Why and How". In: *arXiv preprint arXiv:2509.23984* (2025).

[Hei+23] Lioba Heimbach, Lucianna Kiffer, Christof Ferreira Torres, and Roger Wattenhofer. "Ethereum's Proposer-Builder Separation: Promises and Realities". In: *ACM Internet Measurement Conference (IMC)*. 2023, pages 406–420.

[HW22] Lioba Heimbach and Roger Wattenhofer. "SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance". In: *4th ACM Conference on Advances in Financial Technologies (AFT)*. 2022, pages 47–60.

[Kel+23] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. "Themis: Fast, Strong Order-Fairness in Byzantine Consensus". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023, pages 35–48. DOI: 10.1145/3576915.3616658.

[Kel+20] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. "Order-Fairness for Byzantine Consensus". In: *Advances in Cryptology (CRYPTO)*. Volume 12172. LNCS. 2020, pages 451–480. DOI: 10.1007/978-3-030-56877-1_16.

[Kni+26] Quentin Kniep, Max Resnick, Jakub Sliwinski, and Roger Wattenhofer. *Solana Constellation: Internet Capital Markets*. https://anza.xyz/constellationwhitepaper. White Paper v0.9. Mar. 2026.

[KSW25] Quentin Kniep, Jakub Sliwinski, and Roger Wattenhofer. *Solana Alpenglow Consensus: Increased Bandwidth, Reduced Latency*. https://anza.xyz/alpenglow-1-1. 2025.

[KW26a] Quentin Kniep and Roger Wattenhofer. *SIMD-0674: Validator Location Registration*. Solana Improvement Document, https://github.com/solana-foundation/solana-improvement-documents/pull/674. 2026.

[KW26b] Quentin Kniep and Roger Wattenhofer. *SIMD-0675: Geo-Aware Leader Schedule*. Solana Improvement Document, https://github.com/solana-foundation/solana-improvement-documents/pull/675. 2026.

[Kra94] Hugo Krawczyk. "Secret Sharing Made Short". In: *Advances in Cryptology – CRYPTO '93*. Volume 773. Lecture Notes in Computer Science. Springer, 1994, pages 136–146.

[Kur19] Klaus Kursawe. "Wendy, the Good Little Fairness Widget: Achieving Order Fairness for Blockchains". In: *1st ACM Conference on Advances in Financial Technologies (AFT)*. 2019, pages 25–36. DOI: 10.1145/3318041.3355468.

[LM25] Steven Landers and Benjamin Marsh. *MEV in Multiple Concurrent Proposer Blockchains*. arXiv preprint arXiv:2511.13080. 2025.

[MLJ25] Benjamin Marsh, Steven Landers, and Jayendra Jog. *Sei Giga*. arXiv preprint arXiv:2505.14914. 2025.

[MRR26] Benjamin Marsh, Alejandro Ranchal-Pedrosa, and Max Resnick. *SoK: Multiple Concurrent Proposer Blockchains*. Cryptology ePrint Archive, Paper 2026/2684. 2026. URL: https://eprint.iacr.org/2026/2684.

[Moa+26] Ciamac C. Moallemi, Mallesh M. Pai, Dan Robinson, and Jan Christoph Schlegel. *The Geography of (Concurrent) Block Proposing*. arXiv preprint arXiv:2610.18812. 2026.

[Ran+25] Alejandro Ranchal-Pedrosa, Benjamin Marsh, Lefteris Kokoris-Kogias, and Alberto Sonnino. *Sedna: Sharding Transactions in Multiple Concurrent Proposer Blockchains*. arXiv preprint arXiv:2512.17045. 2025.

[Rou21] Tim Roughgarden. "Transaction Fee Mechanism Design". In: *ACM SIGecom Exchanges* 19.1 (2021), pages 52–55.

[Sar+26] Aditya Saraf, Ioannis Kaklamanis, Sarisht Wadhwa, and Fatima Elsheimy. "Price of Censorship: Censorship Resistance and Throughput under Rational Concurrent Proposers". In: *8th Conference on Advances in Financial Technologies (AFT 2026)*. Volume 395. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026, 26:1–26:24. DOI: 10.4230/LIPIcs.AFT.2026.26.

[SRN26] Giulia Scaffino, Max Resnick, and Joachim Neu. *Gatling: Rapid-Fire Consensus from Parallel Composition*. 8th Conference on Advances in Financial Technologies (AFT), LIPIcs. 2026. DOI: 10.4230/LIPIcs.AFT.2026.4.

[SMT25] Aikaterini-Panagiota Stouka, Julian Ma, and Thomas Thiery. *Multiple Proposer Transaction Fee Mechanism Design: Robust Incentives Against Censorship and Bribery*. arXiv preprint arXiv:2505.13751. 2025.

[Thi+24] Thomas Thiery et al. *EIP-7805: Fork-Choice Enforced Inclusion Lists (FOCIL)*. https://eips.ethereum.org/EIPS/eip-7805. 2024.

[XTS26] Zhuolun Xiang, Andrei Tonkikh, and Alexander Spiegelman. *Prefix Consensus for Censorship Resistant BFT*. arXiv preprint arXiv:2602.02892. 2026.

[Zho+23] Liyi Zhou et al. "SoK: Decentralized Finance (DeFi) Attacks". In: *IEEE Symposium on Security and Privacy (SP)*. 2023, pages 2444–2461. DOI: 10.1109/SP46215.2023.10179435.