Simplex Consensus: An Outline

Tomi Setsu

TOS Blockchain Teams, 2025

April 13, 2026

Abstract

This document provides an outline of the Simplex Consensus Protocol, a slot-based Byzantine Fault Tolerant (BFT) protocol designed for deterministic block finality in the TOS Blockchain. Simplex replaces the earlier Catchain consensus protocol [1] with a simpler, more predictable design that achieves one-shot finality through a three-stage voting model. The protocol tolerates up to $f < n/3$ Byzantine validators and produces blocks at a fixed target rate of 2.4 seconds per slot. The implementation resides in subdirectory `validator/consensus/simplex` of the TOS source tree.

Contents

1 Overview

The Simplex Consensus protocol is the successor to the Catchain Consensus protocol [1] in the TOS Blockchain. While both protocols achieve Byzantine fault tolerance with the standard $f < n/3$ threshold, they differ fundamentally in their approach:

- *Catchain* uses a DAG-based message model with probabilistic convergence over multiple asynchronous rounds. Block finality emerges gradually through dependency-cone analysis.
- *Simplex* uses a slot-based model with deterministic leader election and explicit three-stage voting (notarize, skip, finalize). Block finality is achieved in a single pass, making the protocol simpler to reason about and implement.

1.1. Protocol stack position. Simplex occupies the same position in the TOS protocol stack as Catchain: above the ADNL overlay broadcast layer and below the block generation and validation layer. The protocol receives candidate blocks from the collation layer, achieves consensus on which block to accept for each slot, and delivers finalized blocks to the state application layer.

1.2. Design goals. The principal design goals of Simplex are:

1. **Deterministic finality:** Once a block receives a finalization certificate, it is permanently committed. There is no possibility of reorganization.
2. **Fixed slot boundaries:** Blocks are produced at predictable intervals (target rate: 2400 ms), making the protocol behavior more uniform than Catchain's variable round times.
3. **Simplicity:** The core consensus logic is approximately 3,300 lines of C++ (versus approximately 10,000 for Catchain), achieved by separating concerns into modular actor components.
4. **Standard BFT safety:** The protocol tolerates $f < n/3$ Byzantine validators while maintaining both safety and liveness.

1.3. Architecture. The implementation is organized as a set of communicating actors connected through a typed event bus:

- `ConsensusImpl` — core slot progression and timeout logic
- `PoolImpl` — vote aggregation, certificate formation, and BFT threshold tracking
- `CandidateResolverImpl` — P2P protocol for fetching candidates and certificates
- `StateResolverImpl` — block finalization and state resolution
- `DbImpl` — persistent storage of votes and certificates
- `MetricCollectorImpl` — performance telemetry

These components communicate exclusively through the `Bus` event dispatcher, which provides typed publish/subscribe semantics. This modular design enables each component to be reasoned about and tested independently.

2 Slots and Leader Election

2.1. Slot model. The fundamental unit of consensus is a *slot*. Slots are numbered sequentially starting from zero: $s = 0, 1, 2, \ldots$. Each slot represents an opportunity to produce exactly one block. A slot may result in either:

- A *finalized block*: a valid block proposed by the slot's leader, voted on by a supermajority, and cryptographically committed.
- A *skipped slot*: the slot's leader failed to produce a valid block in time, and a supermajority agreed to skip the slot and move on.

In either case, the protocol makes progress: slots are never replayed or revisited.

2.2. Leader windows. Consecutive slots are grouped into *leader windows* of fixed size W (default $W = 4$). All slots within a single window share the same *leader*, who is responsible for proposing candidate blocks. The leader for window k is determined by round-robin:

$$\text{leader}(k) = \text{validator}[k \bmod N] \tag{1}$$

where N is the number of validators in the current validator set. Equivalently, for slot s:

$$\text{leader}(s) = \text{validator}\big[\lfloor s/W \rfloor \bmod N\big] \tag{2}$$

2.2.1. Rationale for leader windows. Grouping consecutive slots under the same leader amortizes the cost of leader transitions and allows a leader to produce multiple blocks in sequence (a "burst"), which can improve throughput when the leader is well-connected and the network is healthy. If the leader is Byzantine or unavailable, the entire window is skipped via skip votes, and the next leader takes over.

2.3. Target block rate. The protocol targets a fixed block production rate, controlled by the `target_rate` parameter (default: 2400 ms). This means one new block is expected approximately every 2.4 seconds under normal conditions. The actual block time may vary slightly due to network latency and validation time, but the slot clock provides a steady rhythm.

3 Three-Stage Voting Model

The core innovation of Simplex is its explicit three-stage voting model. Each slot progresses through up to three independent voting stages, each requiring a supermajority (more than 2/3 of total validator weight) to form a *certificate*.

3.1. BFT weight threshold. Let W_{total} denote the total weight of all validators in the current validator set. The BFT weight threshold for certificate formation is:

$$W_{\text{thresh}} = \left\lfloor \frac{2 \cdot W_{\text{total}}}{3} \right\rfloor + 1 \tag{3}$$

A certificate is valid if and only if the sum of weights of the signing validators meets or exceeds W_{thresh}. Under the standard BFT assumption $f < n/3$, honest validators always control at least $W_{\text{total}} - W_f > 2W_{\text{total}}/3$ weight, guaranteeing that certificates can always be formed by the honest majority.

3.2. Stage 1: Notarization. The first stage establishes *block availability*. When a validator receives a candidate block from the slot's leader and verifies its validity:

1. The validator broadcasts a NotarizeVote(candidate_id) signed with its Ed25519 key.
2. The `PoolImpl` aggregates incoming notarize votes per candidate.
3. When the accumulated weight for a candidate reaches W_{thresh}, a *notarization certificate* NotarCert is formed by collecting the individual signatures.

The notarization certificate proves that at least 2/3+1 of validator weight has seen and validated the candidate block. This is a necessary prerequisite for finalization.

3.3. Stage 2: Skip. The skip stage provides *liveness* when the leader fails. If a validator does not receive a valid candidate block within the configured timeout:

1. The validator broadcasts a SkipVote(slot).
2. When skip votes accumulate to W_{thresh}, a *skip certificate* SkipCert is formed.

The skip certificate proves that at least $2/3 + 1$ of validator weight has agreed that this slot should be empty. The protocol then advances to the next slot.

3.3.1. Timeout mechanism. The timeout for producing a skip vote is managed by `ConsensusImpl`:

- When a new leader window begins, a timer is set to `first_block_timeout` (default: 1000 ms).
- If no valid candidate arrives before the timeout, skip votes are broadcast for all remaining slots in the window.
- If the previous window was also skipped, the timeout is multiplied by `first_block_timeout_multiplier` (default: 1.2×), capped at `first_block_timeout_cap` (default: 100 s). This exponential backoff prevents skip storms when the network is partitioned.
- When a successful window is observed, the timeout is reset to its default value.

3.4. Stage 3: Finalization. The finalization stage achieves *permanent commitment*. After observing a notarization certificate (or casting one's own notarize vote):

1. The validator broadcasts a FinalizeVote(candidate_id).
2. When finalize votes accumulate to W_{thresh}, a *finalization certificate* FinalCert is formed.

A finalization certificate is *irreversible*: once formed, the corresponding block is permanently part of the blockchain. The finalized block is delivered to the state application layer.

3.5. Voting invariants. The protocol enforces strict per-validator, per-slot voting invariants to prevent equivocation:

1. Each validator casts at most one notarize vote per slot.
2. Each validator casts at most one skip vote per slot.
3. Each validator casts at most one finalize vote per slot.

4. If a validator casts both a notarize vote and a finalize vote for the same slot, they must reference the same candidate.

5. A validator *cannot* cast both a skip vote and a finalize vote for the same slot (these are mutually exclusive outcomes).

Any violation of these invariants constitutes a *misbehavior report* containing the two conflicting signed votes as cryptographic proof of equivocation.

3.6. Safety argument. The mutual exclusivity of notarization certificates and skip certificates follows from the BFT threshold. Suppose both NotarCert(c) and SkipCert(s) exist for the same slot. The signers of NotarCert have total weight $\geq W_{\text{thresh}}$ and the signers of SkipCert also have total weight $\geq W_{\text{thresh}}$. Their intersection must have weight $\geq 2W_{\text{thresh}} - W_{\text{total}} > W_{\text{total}}/3$, which means more than 1/3 of the total weight signed both. But Invariant 5 prohibits any honest validator from doing so. Therefore, at least $W_{\text{total}}/3$ weight must be Byzantine, contradicting our assumption $f < n/3$. This proves that for any slot, at most one of $\{\text{NotarCert}, \text{SkipCert}\}$ can exist.

Similarly, two finalization certificates for different candidates in the same slot cannot coexist under $f < n/3$.

4 Certificate Formation

4.1. Certificate structure. A certificate is a pair consisting of a vote and a set of validator signatures:

$$\text{Certificate}\langle T\rangle = (v : T,\ \text{sigs} : \{(\text{idx}_i, \sigma_i)\}_{i \in S}) \tag{4}$$

where $T \in \{\textsc{NotarizeVote}, \textsc{FinalizeVote}, \textsc{SkipVote}\}$, idx_i is the validator index, σ_i is an Ed25519 signature over the canonical serialization of v within the session context, and S is a subset of validators with $\sum_{i \in S} w_i \geq W_{\text{thresh}}$.

4.2. Signature scope. Each vote is signed over a canonical byte string that includes:

- The session identifier (derived from the validator set hash and catchain sequence number).
- The vote type tag (notarize, finalize, or skip).
- The vote payload (candidate ID with slot number and content hash, or just slot number for skip).

This prevents cross-session replay attacks and ensures that a signature for one vote type cannot be repurposed as another.

4.3. Aggregation in PoolImpl. The `PoolImpl` component maintains per-slot, per-candidate weight accumulators. When a new signed vote arrives:

1. The signature is verified against the validator's Ed25519 public key and the session identifier.
2. The vote is checked against voting invariants (cf. **3.5**). If it conflicts with a previously recorded vote from the same validator, a misbehavior report is generated.
3. The validator's weight is added to the appropriate accumulator.
4. If the accumulator reaches W_{thresh}, the certificate is formed by collecting all recorded signatures for that vote.
5. The certificate is broadcast to all validators and persisted to the database.

4.4. Certificate uniqueness. For a given slot, the following certificates are mutually exclusive:

- NotarCert(c) and SkipCert(s) for the same slot (proved in **3.6**).
- FinalCert(c) and FinalCert(c') for different candidates $c \neq c'$ in the same slot.
- FinalCert(c) and SkipCert(s) for the same slot.

A slot may have both NotarCert(c) and FinalCert(c) for the *same* candidate; indeed, this is the normal progression for a successfully finalized block.

5 Candidate Resolution

5.1. The candidate availability problem. After a notarization certificate is formed, all validators know that the block *exists* and has been validated by a supermajority. However, some validators may not have received the actual block data (for example, if the leader selectively broadcast to only part of the network). The `CandidateResolverImpl` component solves this availability problem.

5.2. Resolution protocol. When a validator needs a candidate block that it has not yet received:

1. It sends a RequestCandidate(candidate_id, want_candidate, want_notar) message to a randomly selected peer.
2. The peer responds with the candidate block data and/or the notarization certificate, if available.
3. If no response arrives within `candidate_resolve_timeout` (default: 1000 ms), the request is retried with a different peer.
4. Retries use exponential backoff: timeout × `candidate_resolve_timeout_multiplier` (default: 1.2×), capped at `candidate_resolve_timeout_cap` (default: 10 s).

5.3. Rate limiting. To prevent denial-of-service attacks, candidate requests are rate-limited to `candidate_resolve_rate_limit` (default: 10 requests per second per peer). Requests exceeding this limit are silently dropped.

6 State Resolution and Finalization

6.1. Finalization flow. When a finalization certificate is observed for slot s with candidate c:

1. The `StateResolverImpl` checks whether the parent block of c has been finalized.
2. If the parent is finalized, c is applied to the blockchain state, producing a new state root.
3. If the parent is *not* finalized (e.g., the parent slot was notarized but not yet finalized), the parent is finalized first (recursively).
4. The finalized block ID and parent reference are persisted to the database.
5. A FinalizeBlock event is emitted to the state application layer.

6.2. Skipped slot handling. A skipped slot does not produce a block, but the protocol still advances its internal slot counter. The next non-skipped block's *parent* reference points to the most recently finalized block, which may be several slots earlier. This creates a sparse chain where consecutive finalized blocks may have non-consecutive slot numbers.

6.3. State caching. The `StateResolverImpl` caches computed states to avoid redundant re-application of blocks. This is important because multiple slots may finalize in rapid succession (e.g., after a network partition heals), and the finalization of later blocks requires the state produced by earlier blocks.

7 Persistence and Bootstrap

7.1. Database schema. The `DbImpl` component persists consensus state using the following key-value structure:

- db.ourVote[vote_hash] $\to$ (vote, seqno) — locally cast votes for crash recovery.
- db.cert[vote_hash] $\to$ certificate — formed certificates.
- db.poolState $\to$ first_nonannounced_window — pool progress marker.
- db.finalizedBlock[candidate_id] $\to$ (block_id, parent) — finalized blocks.

7.2. Bootstrap from crash. When a validator restarts, it recovers its state from the database:

1. Load all locally cast votes (to avoid equivocation after restart).
2. Load all certificates (to reconstruct the consensus state).
3. Load the pool state marker (to know which windows have been processed).
4. For the last incomplete window before the crash, publish skip votes for any un-finalized slots (conservative recovery).

8 Standstill Detection

8.1. Standstill timeout. The `PoolImpl` runs a periodic alarm every `standstill_timeout` (default: 10 s). If the alarm fires during normal operation, it indicates that the consensus has not made progress within the expected time. In response:

1. The current voting state for all tracked slots is logged for diagnostics.
2. The last finalized certificate is re-broadcast to all validators.
3. All certificates in the currently tracked interval are re-broadcast, rate-limited to `standstill_max_egress_bytes_per_s` (default: 6.5 MB/s).

This mechanism helps validators that missed certificates (due to transient network issues) to catch up without requiring a full state sync.

9 Misbehavior Detection

9.1. Conflicting votes. When the `PoolImpl` detects that a validator has cast two votes that violate the invariants of **3.5**, it generates a *misbehavior report* containing both signed votes. This report serves as a cryptographic proof of equivocation that can be used for:

- Slashing the misbehaving validator's stake.
- Excluding the validator from future consensus rounds.
- Forensic analysis of the consensus failure.

9.2. Signature-based banning. Validators that send messages with invalid signatures are temporarily banned for `bad_signature_ban_duration` (default: 5 s). During the ban period, all messages from the banned validator are dropped without processing. This prevents a Byzantine validator from consuming network and computation resources with garbage messages.

10 Configuration Parameters

The following parameters control Simplex consensus behavior. They are set via the blockchain configuration (on-chain) or as validator-engine command-line flags.

Parameter	Default	Description
`target_rate`	2400 ms	Target interval between consecutive blocks
`first_block_timeout`	1000 ms	Initial wait for a candidate block in a leader window before skip
`first_block_timeout_multiplier`	1.2×	Exponential backoff factor for repeated skips
`first_block_timeout_cap`	100 s	Maximum timeout after backoff
`min_block_interval`	0 ms	Minimum time between consecutive blocks
`slots_per_leader_window`	4	Number of slots per leader window
`max_leader_window_desync`	250	Maximum allowable clock skew in leader windows
`standstill_timeout`	10 s	Interval for broadcasting missing certificates
`standstill_max_egress`	6.5 MB/s	Rate limit for standstill rebroadcasts
`candidate_resolve_timeout`	1000 ms	Initial timeout for candidate fetch
`candidate_resolve_multiplier`	1.2×	Backoff factor for candidate fetch retries
`candidate_resolve_cap`	10 s	Maximum candidate fetch timeout
`candidate_resolve_rate_limit`	10/s	Per-peer request rate limit
`bad_signature_ban_duration`	5 s	Temporary ban for invalid signatures
`BFT threshold`	$\lfloor 2W/3 \rfloor + 1$	Weight required for certificate formation

11 Comparison with Catchain Consensus

Aspect	Catchain	Simplex
Message model	DAG-based with dependency cones	Slot-based with explicit votes
Finality	Probabilistic (multiple rounds converge)	Deterministic (one-shot finalize certificate)
Block time	Variable (depends on message delays)	Fixed slots (2.4 s target)
Voting	Implicit via dependency-cone merge	Explicit 3-stage: notarize, skip, finalize
Liveness	Message-based timeout	Slot timeout + skip votes + exponential backoff
State update	Recursive cone-merge (f, g functions)	Linear slot progression with cached state
Implementation	∼10K LOC	∼3.3K LOC
Byzantine threshold	$f < n/3$	$f < n/3$
Cryptography	Ed25519 signatures	Ed25519 signatures

11.1. Why Simplex was introduced. The Catchain consensus protocol, while theoretically sound, has several practical drawbacks that motivated the development of Simplex:

1. **Complexity:** The DAG-based state model with dependency cones, vector timestamps, and recursive merge functions is difficult to reason about and prone to subtle implementation bugs.

2. **Variable latency:** Catchain's round time depends on message propagation delays across the dependency DAG, leading to unpredictable block times.

3. **Gradual finality:** Blocks in Catchain achieve progressively stronger finality over multiple rounds, but never reach the "one signature, done" certainty of Simplex's finalization certificates.

4. **Operational opacity:** Debugging Catchain consensus failures requires understanding the full DAG structure, while Simplex failures reduce to straightforward questions about which validators voted for which slot.

References

[1] T. Setsu, *Catchain Consensus: An Outline*, TOS Blockchain documentation, 2025.

[2] N. Durov, *Telegram Open Network*, 2017.

[3] N. Durov, *Telegram Open Network Blockchain*, 2018.

[4] L. Lamport, R. Shostak, M. Pease, *The byzantine generals problem*, *ACM Transactions on Programming Languages and Systems*, **4/3** (1982), p. 382–401.

[5] M. Castro, B. Liskov, et al., *Practical byzantine fault tolerance*, *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (1999), p. 173–186.

[6] M. Yin, D. Malkhi, M. Reiter, G. Golan-Gueta, I. Abraham, *HotStuff: BFT consensus with linearity and responsiveness*, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), p. 347–356.