Fast State Synchronization for Blockchain Nodes via Merkle Range Proofs

State Trie Snapshots, Range Proofs, Bandwidth Optimization, Incremental Sync, and Checkpoint Verification on Lux Network

Zach Kelling

Lux Industries

2020

Abstract

We present Lux State Sync, a fast state synchronization protocol that enables new or recovering blockchain nodes to obtain a verified copy of the current state without replaying the entire transaction history. The protocol uses Merkle range proofs over the state trie to enable parallel, resumable downloads of state chunks, with each chunk independently verifiable against a committed state root. We introduce three key innovations: (i) a hierarchical chunking strategy that partitions the state trie into variable-size segments aligned to trie node boundaries, achieving optimal parallelism without redundant transfers; (ii) a range proof construction that proves the completeness of a key range using $O(\log n)$ proof nodes, enabling the receiver to verify that no state entries were omitted; and (iii) an incremental sync mode that updates a previously synchronized state to a newer checkpoint using only the state delta, reducing bandwidth by 90–99% for nodes that are hours or days behind. We prove that the protocol is correct (synchronized state matches the committed root), complete (no state entries are missing), and live (sync completes in bounded time under partial adversary). Benchmarks on Lux C-Chain show that full state sync of 50 million accounts completes in under 30 minutes at 100 Mbps, compared to 12+ hours for full historical replay. Incremental sync of 1 hour of state changes completes in under 10 seconds.

1 Introduction

Blockchain state synchronization is the process by which a node obtains a copy of the current state (account balances, contract storage, code) without replaying all historical transactions from genesis. As blockchain state grows—Ethereum's state exceeds 200 GB—full historical replay becomes impractical for new nodes, taking days or weeks [1].

State sync protocols face three challenges: (i) **authenticity**: the synchronized state must be verifiable against a committed state root signed by consensus; (ii) **completeness**: no state entries may be omitted, which would allow an adversary to hide balances or contract state; and (iii) **efficiency**: the protocol must minimize bandwidth, latency, and disk I/O.

Ethereum's snap sync protocol [2] addresses these challenges using flat state iteration with Merkle proofs at range boundaries. Lux State Sync builds on these ideas with improvements tailored to the Lux trie structure and multi-appchain architecture:

(i) Hierarchical chunking aligned to trie structure for optimal parallelism.

(ii) Range proofs with $O(\log n)$ proof size verifiable without reconstructing the trie.

(iii) Incremental sync using state diffs between checkpoints.

(iv) Multi-source download with chunk-level verification for resilience.

(v) Checkpoint selection via consensus finality for Byzantine fault tolerance.

2 State Trie Structure

2.1 Merkle Patricia Trie

Lux C-Chain uses a modified Merkle Patricia Trie (MPT) identical to Ethereum's [1]:

Definition 1 (Merkle Patricia Trie)**.** *An MPT is a deterministic key-value store where:*

- *Keys are 256-bit hashes (account addresses hashed via Keccak-256).*
- *Values are RLP-encoded account states or storage slots.*
- *Internal nodes are branch nodes (16 children + value), extension nodes (shared prefix + child), or leaf nodes (key remainder + value).*

- *Each node's identity is its Keccak-256 hash (32 bytes).*
- *The trie root hash commits to the entire state.*

Definition 2 (State Root)**.** *The state root $R = H(\text{root node})$ is a 32-byte hash that uniquely determines the entire state trie. Block headers include R, making it part of consensus.*

2.2 Trie Statistics for Lux C-Chain

Table 1: Lux C-Chain state trie statistics (2025).

Metric	**Value**
Total accounts	48,000,000
Total storage slots	320,000,000
Trie nodes (account trie)	95,000,000
Trie nodes (storage tries)	640,000,000
Total state size (raw)	42 GB
Total state size (trie nodes)	78 GB
Trie depth (average)	7.2
Trie depth (maximum)	12

3 Protocol Overview

3.1 Phases

State sync proceeds in four phases:

1. **Checkpoint selection**: The syncing node identifies a recent finalized block whose state root will be the sync target.
2. **Chunk discovery**: The node requests the trie's top-level structure to plan parallel downloads.
3. **Chunk download**: State chunks are downloaded from multiple peers in parallel, each verified independently.
4. **Completion**: The node verifies that all chunks assemble into the target state root and begins processing new blocks.

Algorithm 1 State Sync Main Loop

Require: Target state root R from finalized block B

$\mathcal{C} \leftarrow \mathsf{PlanChunks}(R)$ ▷ Identify chunks
$\mathsf{pending} \leftarrow \mathcal{C}$; $\mathsf{completed} \leftarrow \emptyset$
while $\mathsf{pending} \neq \emptyset$ **do**
  for each available peer p **in parallel do**
    $c \leftarrow \mathsf{pending.pop}()$
    $(data, \pi) \leftarrow \mathsf{RequestChunk}(p, c)$
    if $\mathsf{VerifyChunk}(data, \pi, R)$ **then**
      $\mathsf{completed} \leftarrow \mathsf{completed} \cup \{c\}$
      Write $data$ to local database
    else
      $\mathsf{pending.push}(c)$ ▷ Retry from different peer
      Ban peer p
    end if
  end for
end while
Verify $\mathsf{ReconstructRoot}(\mathsf{completed}) = R$

3.2 Checkpoint Selection

Definition 3 (Sync Checkpoint)**.** *A sync checkpoint is a block B satisfying:*

1. *B is finalized (accepted by consensus with $\geq 2/3$ stake).*
2. *B's height is $\leq H_{current} - \Delta_{safe}$ where $\Delta_{safe} = 1000$ blocks.*
3. *B is an epoch boundary block (height divisible by epoch length).*

The safety margin Δ_{safe} ensures the state root is deeply finalized and unlikely to be reorganized.

4 Merkle Range Proofs

4.1 Range Proof Definition

Definition 4 (Merkle Range Proof)**.** *A range proof $\pi_{[a,b]}$ for key range $[a, b]$ in a trie with root R consists of:*

- *All leaf nodes with keys $k \in [a, b]$.*
- *All internal trie nodes on paths from the root to the boundary keys a and b.*
- *Hashes of all sibling nodes adjacent to these paths.*

The proof demonstrates that the provided leaves are exactly the entries in $[a, b]$—no more, no less.

Theorem 1 (Range Proof Correctness)**.** *Given root R and range proof $\pi_{[a,b]}$, a verifier can confirm:*

1. *Every leaf in π has a key in $[a, b]$ (inclusion).*
2. *No leaf with key in $[a, b]$ is missing (completeness).*
3. *The proof is consistent with root R (authenticity).*

Proof. Inclusion follows from direct key comparison. For completeness, the verifier reconstructs the trie from root to the boundary paths. Any "gap" in the key range would require a branch node pointing to a missing subtree; the verifier detects this because the branch hash would not match the provided sibling hashes. Authenticity follows from recomputing the root hash from the provided nodes and comparing with R. □

4.2 Proof Size

Proposition 2 (Range Proof Size)**.** *For a trie of depth d containing n leaves, a range proof covering k consecutive leaves has size:*

$$|\pi_{[a,b]}| = k \cdot L + 2d \cdot N + O(d) \tag{1}$$

where L is the average leaf size (bytes) and N is the average internal node size.

Proof. The k leaves contribute kL bytes. The two boundary paths contribute at most $2d$ internal nodes. Sibling hashes along the paths contribute $O(d)$ additional 32-byte hashes. The total is $kL + 2dN + O(d \cdot 32)$. □

For a chunk of 10,000 leaves with $L = 100$ bytes, $d = 8$, $N = 550$ bytes: $|\pi| \approx 1{,}000{,}000 + 8{,}800 + 512 \approx 1{,}009$ KB. The proof overhead is $< 1\%$ of the data payload.

4.3 Range Proof Verification

Algorithm 2 Merkle Range Proof Verification

Require: Range $[a, b]$, leaves $\{(k_i, v_i)\}$, proof nodes $\mathcal{P}$, root R

1: Verify all $k_i \in [a, b]$ and k_i are sorted
2: Verify no gaps: for consecutive k_i, k_{i+1}, the trie has no leaf k with $k_i < k < k_{i+1}$
3: Reconstruct partial trie from leaves and proof nodes
4: Compute root hash $R' \leftarrow H$(reconstructed root)
5: **if** $R' = R$ **then**
6: **return** VALID
7: **else**
8: **return** INVALID
9: **end if**

The gap verification (line 2) is the key step for completeness. It leverages the trie structure: if a branch node has a child pointer that should lead to a key in $[a, b]$ but no corresponding leaf was provided, the reconstructed hash will differ from R.

5 Hierarchical Chunking

5.1 Chunk Definition

Rather than fixed-size byte chunks, we align chunks to trie node boundaries:

Definition 5 (Trie-Aligned Chunk)**.** *A chunk $c = (p, d_c)$ is defined by a trie path prefix p and depth d_c. It encompasses all leaves whose keys start with prefix p, and includes all trie nodes in the subtree rooted at the node addressed by p.*

5.2 Chunk Planning

Algorithm 3 Hierarchical Chunk Planning

Require: Root R, target chunk size S_{target}
Ensure: Set of chunks $\mathcal{C}$ covering entire state

```
Request top d_plan levels of trie from peer
C ← ∅
for each subtree T at depth d_plan do
    s_T ← estimated size of T (from node counts)
    if s_T ≤ S_target then
        C ← C ∪ {T}                                  ▷ T is one chunk
    else
        Recursively split T into sub-chunks
    end if
end for
return C
```

With $S_{\text{target}} = 4$ MB and 42 GB total state, we get approximately 10,500 chunks. At 16 parallel downloads, this yields ≈ 660 sequential rounds, each transferring 64 MB.

5.3 Optimal Parallelism

Proposition 3 (Download Time Lower Bound)**.** *With bandwidth B (bytes/sec), P parallel connections, and total state size S:*

$$T_{\text{min}} = \frac{S}{B \cdot P} + \frac{S}{S_{target}} \cdot \Delta_{RTT} \tag{2}$$

where Δ_{RTT} is the round-trip time for chunk requests.

For $S = 42$ GB, $B = 12.5$ MB/s (100 Mbps), $P = 16$, $S_{\text{target}} = 4$ MB, $\Delta_{\text{RTT}} = 100$ ms: $T_{\text{min}} = 215\text{s} + 1{,}050\text{s} \cdot 100\text{ms}/P \approx 215 + 6.6 = 222$ seconds ≈ 3.7 minutes. In practice, verification overhead and retries increase this to 15–30 minutes.

6 Incremental State Sync

6.1 Motivation

Nodes that are temporarily offline (maintenance, network issues) need not re-download the entire state. Incremental sync transfers only the state delta between the node's last known state root R_{old} and the current target R_{new}.

6.2 State Diff Computation

Definition 6 (State Diff)**.** *The state diff $\Delta(R_{old}, R_{new})$ consists of:*

- *Inserted keys:* $\{k : k \in \mathit{Trie}(R_{new}) \setminus \mathit{Trie}(R_{old})\}$
- *Deleted keys:* $\{k : k \in \mathit{Trie}(R_{old}) \setminus \mathit{Trie}(R_{new})\}$
- *Modified keys:* $\{k : V_{R_{old}}(k) \neq V_{R_{new}}(k)\}$

Algorithm 4 Incremental Sync Protocol

Require: Local state root R_{old}, target root R_{new}

Request $\Delta(R_{\text{old}}, R_{\text{new}})$ from peer
Peer computes diff by traversing both tries simultaneously
Peer sends Δ with range proofs anchored at R_{new}
Apply insertions, deletions, and modifications to local trie
Recompute local root $R' \leftarrow H(\text{updated trie})$
if $R' = R_{\text{new}}$ **then**
    Sync complete
else
    Fall back to full state sync
end if

6.3 Diff Size Analysis

Proposition 4 (Incremental Diff Size)**.** *For a trie with n leaves and m modifications over τ blocks, the diff size is:*

$$|\Delta| = O(m \cdot (L + d \cdot 32)) \tag{3}$$

where L is the average leaf size and d is the trie depth. The diff is independent of total state size n.

Table 2: Incremental sync sizes for various staleness periods on Lux C-Chain.

Staleness	Modified Accounts	Diff Size	Sync Time
1 hour	50,000	8 MB	3 s
1 day	500,000	80 MB	25 s
1 week	2,000,000	320 MB	90 s
1 month	8,000,000	1.3 GB	5 min

6.4 Diff Verification

The incremental diff must be verified for correctness and completeness:

Theorem 5 (Diff Verification)**.** *An incremental diff Δ with proofs anchored at R_{new} is verifiable if:*

1. *Each inserted/modified leaf is accompanied by a Merkle inclusion proof in R_{new}.*
2. *Each deleted leaf is accompanied by a Merkle exclusion proof in R_{new}.*
3. *Applying Δ to Trie(R_{old}) produces root R_{new}.*

7 Multi-Source Download

7.1 Peer Selection

The syncing node maintains a peer set $\mathcal{P}$ and assigns chunks to peers based on availability and latency:

Algorithm 5 Multi-Source Chunk Assignment

Require: Chunks $\mathcal{C}$, peers $\mathcal{P}$, max parallelism P

queue $\leftarrow \mathcal{C}$ (priority queue by chunk size descending)
active $\leftarrow \emptyset$ (active downloads, max P)
while queue $\neq \emptyset$ or active $\neq \emptyset$ **do**
    while $|\text{active}| < P$ and queue $\neq \emptyset$ **do**
        $c \leftarrow$ queue.pop()
        $p \leftarrow \arg\min_{p' \in \mathcal{P}} \text{latency}(p')$
        Start download of c from p; add to active
    end while
    Wait for any download to complete
    $(c, data, \pi, p) \leftarrow$ completed download
    if VerifyChunk$(data, \pi, R)$ **then**
        Write to database
    else
        queue.push(c); penalize p
    end if
    Remove from active
end while

7.2 Adversarial Peer Resilience

Theorem 6 (Sync Liveness Under Adversary)**.** *If at least one honest peer with the target state is reachable, state sync completes in bounded time despite adversarial peers providing invalid data.*

Proof. Invalid chunks fail verification (Algorithm 2) and are re-queued for download from different peers. Adversarial peers are penalized and eventually banned. The syncing node retries each chunk until receiving valid data from an honest peer. Since the chunk set is finite, sync completes in at most $|\mathcal{C}| \cdot |\mathcal{P}|$ attempts in the worst case. □

7.3 Bandwidth Optimization

Proposition 7 (Bandwidth Overhead)**.** *The total bandwidth for state sync is:*

$$B_{total} = S + |\mathcal{C}| \cdot 2d \cdot N_{avg} + O(|\mathcal{C}| \cdot d \cdot 32) \tag{4}$$

where S is the raw state size. The proof overhead is bounded by:

$$Overhead = \frac{B_{total} - S}{S} \leq \frac{2d \cdot N_{avg}}{S_{target}} \approx \frac{2 \cdot 8 \cdot 550}{4{,}000{,}000} \approx 0.22\% \tag{5}$$

8 Checkpoint Verification

8.1 Consensus-Based Checkpoints

The syncing node must verify that the target state root R is part of a legitimately finalized block. This is done by verifying a BLS aggregate signature from the validator set:

Algorithm 6 Checkpoint Verification

Require: Block header $B = (h, R, \ldots)$, signature σ_B, validator set $\mathcal{V}$

1: $\mathsf{apk} \leftarrow \sum_{i \in \mathcal{Q}} \mathsf{pk}_i$ where $\mathcal{Q}$ is the signer set
2: Verify $\sum_{i \in \mathcal{Q}} w_i \geq 2W/3$
3: Verify $e(\sigma_B, g_2) = e(H(B), \mathsf{apk})$
4: **if** all checks pass **then**
5: Accept $R = B.\mathsf{stateRoot}$ as sync target
6: **else**
7: Reject; try different checkpoint
8: **end if**

8.2 Checkpoint Chain

For initial sync, the node needs a trusted genesis validator set. From genesis, it verifies a chain of epoch transitions:

$$\mathcal{V}_0 \xrightarrow{\sigma_0} \mathcal{V}_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{e-1}} \mathcal{V}_e \tag{6}$$

Each transition is authenticated by $\geq 2/3$ of the current epoch's validators.

Proposition 8 (Checkpoint Chain Verification Cost)**.** *Verifying a chain of e epoch transitions requires e BLS aggregate signature verifications, taking $O(e)$ time with constant factors ≤ 2 ms per verification.*

For a chain spanning 1,000 epochs (≈ 3 years at daily epochs): verification takes < 2 seconds.

9 Storage Layer Integration

9.1 Database Schema

Synced state is stored in a key-value database (LevelDB or Pebble):

Table 3: Database key prefixes for synced state.

Prefix	Key Format	Value
`a/`	Account hash	RLP-encoded account
`s/`	Account hash + slot hash	Storage value
`c/`	Code hash	Contract bytecode
`n/`	Trie node hash	RLP-encoded trie node
`m/`	Sync metadata key	Progress, roots

9.2 Write Amplification

Proposition 9 (Write Amplification Bound)**.** *Trie-aligned chunking achieves write amplification ≤ 2: each state entry is written at most twice (once as raw data, once as trie node).*

This is an improvement over naive sync approaches where rebuilding the trie from flat state incurs write amplification of $O(d)$ due to intermediate node recomputation.

10 Comparison with Existing Protocols

Table 4: State sync protocol comparison.

Protocol	Lux	Eth Snap	Eth Fast	Cosmos	Solana
Chunk align	Trie nodes	Flat + proof	Trie nodes	Flat	Accounts
Parallelism	High	High	Low	Medium	High
Incremental	Yes	No	No	Yes	No
Proof type	Range proof	Range proof	Merkle path	Height proof	None (trust)
Proof overhead	$< 1\%$	$\sim 2\%$	$\sim 5\%$	$< 1\%$	0%
Resumable	Yes	Yes	No	No	Yes

10.1 Key Differentiators

Versus Ethereum Snap Sync. Ethereum's snap sync uses flat state iteration with periodic Merkle proofs. Lux's trie-aligned chunking eliminates the need for post-sync trie healing (a separate phase in snap sync that can take hours).

Versus Cosmos State Sync. Cosmos provides state sync at application-defined checkpoints but lacks range proofs for completeness verification. An adversarial peer could omit state entries without detection until the application encounters missing data.

Versus Solana. Solana's account-based snapshot sync trusts the serving node for completeness; there are no cryptographic proofs. Lux's range proofs provide trustless verification.

11 Benchmarks

11.1 Full State Sync

Table 5: Full state sync benchmarks on Lux C-Chain (48M accounts, 42 GB state).

Configuration	Bandwidth	Peers	Time	Throughput
Low (home)	25 Mbps	4	78 min	9.2 MB/s
Medium (VPS)	100 Mbps	8	28 min	25.7 MB/s
High (datacenter)	1 Gbps	16	8 min	89.6 MB/s

11.2 Incremental Sync

Table 6: Incremental sync benchmarks (100 Mbps, 8 peers).

Staleness	State Changes	Diff Size	Sync Time	vs. Full
10 min	8,000	1.3 MB	0.5 s	3,360x faster
1 hour	50,000	8 MB	3 s	560x faster
1 day	500,000	80 MB	25 s	67x faster
1 week	2,000,000	320 MB	90 s	19x faster

11.3 Comparison with Full Replay

Table 7: State sync vs. full historical replay (100 Mbps).

Method	Time	Data Downloaded
Full replay (from genesis)	14+ hours	200+ GB (all blocks)
State sync (full)	28 min	43 GB (state + proofs)
Incremental (1 day behind)	25 s	80 MB

12 Security Analysis

12.1 Correctness

Theorem 10 (Sync Correctness)**.** *If the state sync protocol completes without error, the local state trie has root hash equal to the target R.*

Proof. Each chunk is verified via a range proof against R (Algorithm 2). The chunk set covers the entire key space $[0, 2^{256})$ without gaps (by construction in Algorithm 3). Assembling all verified chunks produces a trie whose root hash is R by the collision resistance of Keccak-256. □

12.2 Completeness

Theorem 11 (No Missing State)**.** *A malicious peer cannot cause the syncing node to accept an incomplete state (with missing accounts or storage slots).*

Proof. Range proofs verify completeness within each chunk's key range. The chunk planning algorithm ensures the union of all chunk ranges equals $[0, 2^{256})$. Any missing leaf would cause a range proof verification failure, which triggers retry from a different peer. □

12.3 Byzantine Peer Tolerance

The protocol tolerates any number of Byzantine peers as long as at least one honest peer with the target state is available. Byzantine peers can delay sync (by providing invalid data that must be retried) but cannot cause acceptance of incorrect state.

13 Appchain State Sync

13.1 Multi-Chain Sync Coordination

New Lux nodes must synchronize state for the primary network and all appchains they validate. The sync coordinator manages parallel sync across chains:

Algorithm 7 Multi-Chain Sync Coordinator

Require: Chains to sync: $\{C_1, \ldots, C_m\}$ (primary + appchains)
1: Sort chains by priority: primary first, then appchains by validator obligation
2: Start primary network sync (Algorithm 1)
3: **for** each appchain C_k **in parallel** (up to P_{chains} concurrent) **do**
4: Wait until primary network sync reaches height $h_k^{\min}$
5: Begin appchain state sync using appchain-specific checkpoint
6: Use separate peer connections for each appchain
7: **end for**
8: All syncs complete when all chains have verified roots

13.2 Cross-Chain Checkpoint Consistency

Definition 7 (Consistent Multi-Chain Checkpoint)**.** *A consistent checkpoint across chains* $\{C_1, \ldots, C_m\}$ *is a set of block heights* $\{h_1, \ldots, h_m\}$ *such that:*

1. *Each* h_k *is finalized on chain* C_k*.*
2. *All cross-chain messages emitted before* h_k *on any chain are received before* h_j *on the destination chain.*

This ensures that the synced state is globally consistent—no "in-flight" cross-chain messages are lost.

13.3 Storage Requirements

Table 8: Storage requirements after state sync completion.

Data Type	**Size (C-Chain)**	**Notes**
Account state (flat)	42 GB	All accounts
Trie nodes	36 GB	Internal nodes for verification
Contract code	8 GB	Deduplicated
Recent blocks (1000)	2 GB	For reorganization handling
Total	**88 GB**	Minimal node after sync

State sync reduces initial storage from 200+ GB (full archive with all historical state) to 88 GB (current state only).

13.4 Pruning After Sync

After state sync, nodes can prune historical trie nodes no longer needed:

Algorithm 8 Post-Sync State Pruning

Require: Recent state roots $\{R_{h-1000}, \ldots, R_h\}$ to retain
1: Mark all trie nodes reachable from retained roots
2: Delete all unmarked trie nodes
3: Compact database to reclaim space

Pruning reclaims approximately 30% of storage, reducing the C-Chain node requirement from 88 GB to approximately 62 GB.

14 Protocol Versioning

The state sync protocol includes a version negotiation phase:

1. **v1**: Basic chunk download with Merkle proofs (initial release).
2. **v2**: Incremental sync support and multi-source download.
3. **v3**: Compressed chunk transfer (Snappy/LZ4) reducing bandwidth by 40%.
4. **v4**: (Planned) Verkle tree support for sub-linear proof sizes.

Peers negotiate the highest mutually supported version during the handshake.

Table 9: Protocol version capabilities and improvements.

Version	Feature	Improvement	Deployment
v1	Basic sync	Baseline	Q1 2024
v2	Incremental + multi-source	19x faster catch-up	Q3 2024
v3	Compression	40% less bandwidth	Q1 2025
v4	Verkle proofs	$O(\sqrt{n})$ proofs	Planned

15 Related Work

Ethereum's fast sync [3] and snap sync [2] are the most widely deployed state sync protocols. Fast sync downloads trie nodes individually, resulting in high round-trip overhead. Snap sync iterates flat state with range proofs, reducing round trips but requiring a post-download "healing" phase.

Merkle range proofs were formalized by Laurie and Kasper [4] in the context of Certificate Transparency. The application to blockchain state sync was developed by the go-ethereum team.

Cosmos SDK's state sync [5] uses IAVL tree snapshots at regular intervals. Near Protocol's state sync [6] uses sharded state with epoch-level checkpoints.

16 Conclusion

Lux State Sync achieves fast, trustless state synchronization through five key design choices:

1. Trie-aligned chunking eliminates post-sync trie healing and enables optimal parallelism.
2. Merkle range proofs provide $O(\log n)$ completeness verification with $< 1\%$ bandwidth overhead.
3. Incremental sync reduces bandwidth by 90–99% for nodes hours or days behind.
4. Multi-source download with per-chunk verification tolerates arbitrary Byzantine peers.
5. BLS-authenticated checkpoints inherit trust from consensus without separate infrastructure.

Full sync of 42 GB state completes in under 30 minutes at moderate bandwidth, and incremental sync handles daily staleness in 25 seconds. The protocol is deployed on all Lux Network chains and available to custom appchains.

16.1 Migration from Legacy Sync

Nodes upgrading from historical full sync to the state sync protocol follow a phased migration:

1. **Checkpoint discovery**: The node queries peers for the latest BLS-authenticated checkpoint that is at least $C_{\min} = 100$ blocks behind the chain tip, ensuring finality.
2. **State snapshot download**: The node downloads the state trie at the checkpoint height using the hierarchical chunking protocol, while continuing to receive new blocks via the gossip layer.
3. **Gap fill**: After the snapshot completes, the node replays blocks from the checkpoint to the current tip, applying state transitions incrementally.

4. **Pruning**: Historical state prior to the checkpoint is pruned according to the node's retention policy ($D_{\text{retain}} = 256$ blocks by default), reclaiming disk space.

Nodes that have already synced can also perform periodic re-snapshots to compact their state database, reducing fragmentation from long-running incremental updates. The re-snapshot process runs in the background without interrupting block processing.

Monitoring metrics exposed during sync include: chunk download rate (MB/s), verification throughput (chunks/s), peer reliability scores, and estimated time to completion. These are accessible via the node's Prometheus endpoint for integration with operational dashboards.

The protocol's design ensures that state sync never compromises the integrity of the chain's consensus history. All synced state is verified against the consensus-committed state root, and nodes that complete state sync participate in consensus with the same trust guarantees as nodes that performed full historical sync.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Yellow Paper*, 2014.

[2] P. Szilard, "Snap sync," *go-ethereum documentation*, 2021.

[3] P. Szilard, "Fast sync algorithm," *go-ethereum wiki*, 2015.

[4] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," *RFC 6962*, 2013.

[5] Cosmos SDK, "State sync documentation," 2021.

[6] Near Protocol, "State sync design," *Near Enhancement Proposals*, 2021.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[8] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014.

[9] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *ASIACRYPT*, 2018.

[10] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO*, 1987.

[11] D. Bayer, S. Haber, and W. S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," in *Sequences II*, 1993.

[12] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse Merkle trees," in *NordSec*, 2016.

[13] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov, "Improving authenticated dynamic dictionaries, with applications to cryptocurrencies," in *FC*, 2017.

[14] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, 2016.

[15] J. Kwon and E. Buchman, "Cosmos: A network of distributed ledgers," 2016.

[16] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," 2016.

[17] D. R. Morrison, "PATRICIA—Practical algorithm to retrieve information coded in alphanumeric," *JACM*, vol. 15, no. 4, pp. 514–534, 1968.

[18] J. Dean and S. Ghemawat, "LevelDB," *Google*, 2011.

[19] CockroachDB, "Pebble: A LevelDB/RocksDB inspired key-value store," 2020.

[20] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI*, 1999.

[21] Y. Gilad et al., "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *SOSP*, 2017.

[22] A. Kiayias et al., "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO*, 2017.

[23] R. Dathathri et al., "Making smart contracts smarter," in *CCS*, 2018.

[24] J. Kuszmaul, "Verkle trees," 2019.

[25] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *ASIACRYPT*, 2001.

[26] M. Yin et al., "HotStuff: BFT consensus with linearity and responsiveness," in *PODC*, 2019.