Robust and Fast Blockchain State Synchronization

Enrique Fynn
Università della Svizzera italiana (USI), Lugano, Switzerland

Ethan Buchman
Informal Systems, Guelph, Canada

Zarko Milosevic
Informal Systems, Guelph, Canada

Robert Soulé
Yale University, New Haven, CT, USA

Fernando Pedone
Università della Svizzera italiana (USI), Lugano, Switzerland

Abstract

State synchronization, the process by which a new or recovering peer catches up with the state of other operational peers, is critical to the operation of blockchain-based systems. Existing approaches to state synchronization typically involve downloading snapshots of system state. Such approaches introduce an attack vector from malicious peers that can significantly degrade performance. Moreover, the process of creating snapshots leads to performance hiccups. This paper presents a technique for peers to catch up with operational peers without trusting any particular peer and gracefully recover from misbehavior during the process. We have integrated our design into a production blockchain middleware. Our evaluation shows that during operation, the transaction throughput is consistently higher without pauses for snapshot construction. Moreover, the time it takes for a new peer to join the blockchain is halved, while at the same time tolerating Byzantine peers.

2012 ACM Subject Classification Computer systems organization → Reliability; Computer systems organization → Availability; Computer systems organization → Redundancy; Computing methodologies → Distributed algorithms

Keywords and phrases state synchronization, replication, blockchain

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.8

Funding This work was supported in part by the Swiss National Science Foundation (grant 175717) and NSF FMitF (grant 2019285).

1 Introduction

Intuitively, a blockchain provides an append-only log of transactions implemented by geographically distributed peers. The execution of these transactions determines the system state. Each peer stores the system state in a Merkle tree [23], or similar data structures (e.g., Merkle-Patricia-tree [26]). Executing a transaction involves performing operations on the tree. By executing the log of transactions in the same order, each peer transitions through the same state changes (i.e., state machine replication model [22]). The fact that the state is stored in a Merkle tree allows a peer to validate the consistency of the state by computing a hash on the reconstructed tree. A Merkle-tree is a tree in which every leaf node stores a cryptographic hash of its value, and every non-leaf node stores the hash of its children. Every block header in the blockchain stores the hash of the tree root constructed by the transactions in a previous block (e.g., block n contains the hash of the tree root computed with transactions in block $n-1$).

In theory, a new peer joining the network (or a recovering peer) could download the transactions it misses, possibly the entire blockchain, and reconstruct the state by re-executing every missing transaction. However, this is impractical, since the number of transactions grows too large over time. Therefore, existing blockchain-based systems (e.g., [2, 26]) rely on periodic snapshots of the system state. The snapshot is a serialized representation of the tree data structure. When a new peer joins the blockchain (or a failed peer recovers), it downloads the blockchain blocks with transactions (or block headers, with a summary of the block) and the snapshot. The peer then installs the snapshot and replays all transactions since the snapshot was taken, to reconstruct the current system state. To validate a snapshot, a peer simply computes the hash on its tree and compares the computed value with the value stored in the trusted block header. To further decrease the time it takes for a new peer to join the blockchain, a snapshot can be divided into *chunks*, each one containing multiple nodes of the tree. This allows new peers to download chunks from many peers concurrently. A natural strategy to assign tree nodes to chunks is to traverse the tree (e.g., using depth-first search) and build fixed-sized chunks [3].

Unfortunately, this approach to state synchronization suffers from two subtle, but important problems. The first problem is due to the relationship between chunks and validation. If nodes of the state tree are assigned to chunks in a naïve way, as described above, then the chunks cannot be validated independently. Instead, a peer must download the entire tree before it can compute the hash. This introduces an effective attack vector for misbehaving peers. If a single malicious peer shares an invalid chunk, the new peer cannot identify the problem until it has downloaded all of the chunks. This wastes network and disk resources of both the sender and the receiver, and can significantly prolong the time it takes for a new peer to join the blockchain. The second problem is with performance. When a peer computes a snapshot, it needs to search the tree, serialize all nodes, build chunks, and save them to local storage. We show in the paper that existing blockchain-based systems based on this approach experience periodic hiccups, dropping transaction throughput.

In this paper, we provide data structures and algorithms for robust and fast blockchain state synchronization. By robust, we mean that our solution can tolerate Byzantine failures. By fast, we mean that (i) validation and state reconstruction can be performed quickly, and (ii) peers do not have to pause operation in order to compute a snapshot. The key component of our solution is the design of a novel data structure targeted specifically to the state synchronization use case. This data structure, which we call an AVL* tree, is a Merklized AVL tree in which the tree leaves are organized into chunks. However, the assignment of nodes to chunks ensures that each chunk always contains a sub-tree of the system state. This enables batches of leaves to be securely and efficiently downloaded concurrently, while also permitting chunks to be verified for integrity using a compact proof. Finally, the structure of the tree is such that after a transaction is executed, a peer needs only to recompute the hash of the affected chunk, and propagate the hash up the tree to the root. It does not need to recompute the entire snapshot. Thus, during normal operation, the peer never has to pause transaction processing.

Incorporating leaf batching into a Merkle-ized AVL tree might seem straightforward. In reality, the problem introduces several challenges. First among these is identifying the proper invariants that must be maintained so that the integrity of the chunks can be checked independently. Second is designing the non-trivial changes to the AVL tree's operation methods to preserve the invariants. Third, during state transfer, peers will download different chunks in parallel, and can start reconstructing the tree. In general, inserting the data into a tree can result in different trees. To ensure correctness, we need to guarantee that the tree construction algorithm deterministically builds the same tree on different peers.

We have implemented the AVL* tree data structure and integrated it into the Tendermint blockchain middleware. We show that the time it takes for a new peer to join the blockchain using AVL* is halved, while at the same time tolerating Byzantine peers. Moreover, the improvements in the performance of state synchronization do not degrade the performance of steady execution. In fact, AVL* slightly increases throughput and reduces latency. Finally, we show that these improvements increase with the system size.

The rest of this paper is organized as follows. We first provide the necessary background (§2). We then describe the basic structure and operations of our AVL* tree (§3), followed by algorithms that enable fast state synchronization (§4). We then provide a thorough evaluation of AVL* trees, comparing to an AVL tree (§5). Finally, we present related work (§6) and conclude (§7). The Appendix contains a correctness argument for tree operations and state synchronization, and the detailed algorithms proposed in the paper.

2 Background

2.1 Blockchain

A blockchain is a distributed ledger, an append-only log of transactions implemented by geographically distributed peers. Clients submit transactions to the blockchain, which are appended to the log and executed by peers. Clients and peers may be *honest*, if they follow the protocol specification, or *malicious* (Byzantine), if nothing can be assumed about their behavior. The blockchain behaves correctly if a fraction of the peers, typically more than two-thirds, is honest [20].

The append-only log is structured as a linked-list of blocks, each block divided into a header and a body. The header contains, among other information, a cryptographic link to the previous block and a hash of the state. The body contains a list of transactions, each transaction cryptographically signed by the client that submitted it. Some blockchain systems (e.g., [24, 26]) allow the chain of blocks to momentarily fork, a situation in which multiple blocks are linked to a previous block. An alternative design is to ensure a total order on linked blocks (e.g., [15]). In this case, peers must agree on the next block to be appended by means of a byzantine fault-tolerant consensus protocol (e.g., [16, 21]).

2.2 Merkle trees

Generally, the state of a blockchain is stored locally by each peer in a key-value store structured as a Merkle-tree [25], or similar data structure (e.g., Merkle-Patricia-tree [26]). In a Merkle-tree, each leaf holds the value and its hash, and each inner node holds the hash of its children. The hash of the tree's root (Merkle-root) is stored in the blockchain header, which ensures its integrity. The key function of Merkle-trees is to provide succinct proofs of data integrity of any node of the tree. One can prove in logarithmic time and space that a leaf v belongs to the tree by providing the hash of the siblings of the tree nodes in the path from v to the Merkle-root m. In Figure 1, these hashes are h_0 and h_3. One can verify that v belongs to the tree by recalculating h_2, h_1 and m, and then verifying that the recalculated m is equal to the known trusted Merkle-root hash, stored in the blockchain.

The Merkle-tree of a blockchain is multi-version: a new tree is created for every new blockchain block, typically implemented with copy-on-write for performance reasons. When a new version of the tree is instantiated, the Merkle-root from the current version is copied to a new Merkle-root and made the Merkle-root of the new version. Modified nodes are duplicated and saved under the new tree. Although it is not common for peers to navigate on older

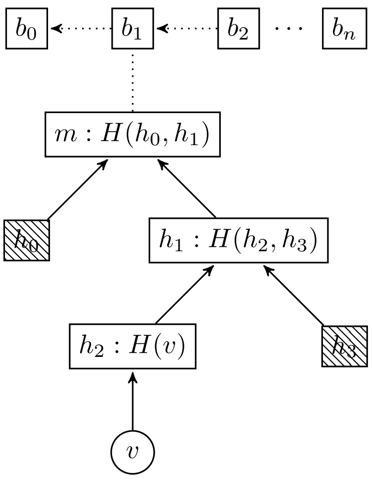


Figure 1 Blockchain $b_0, ..., b_n$ and Merkle-proof h_0;h_3 of v. Hashes m, h_1 and h_2 are computed from v, h_0, and h_3, and hash function $H()$; v is valid if m matches the value stored in block b_1.

versions of the tree, it is required for certain inter-blockchain communication protocols [6], where peers serve as relayers between blockchains, allowing one blockchain to verify state proofs from another at a recent block.

2.3 State synchronization problem

A new (or recovering) blockchain peer needs to retrieve and execute transactions from all the blocks it has missed in order to catch up with operational peers. Merkle-trees enable fast catch up techniques based on traditional state machine replication [16]: Instead of re-executing all missing transactions, the peer retrieves a recent version of the Merkle-tree (i.e., a snapshot) from other peers and applies only the transactions in blocks that succeed the retrieved Merkle-tree. This technique has been implemented in popular blockchain clients (e.g., Geth [7], Ethereum's main client).

The obvious question, though, is how should a peer download the snapshot? On the one hand, since a snapshot can be large, the download process can be accelerated by partitioning the tree into chunks, and downloading chunks from several peers in parallel. On the other hand, in blockchain systems, leaf sizes tend to be small (e.g., a hundred bytes), and the number of leaves quite large, creating significant I/O overhead if one were to serve each leaf independently. It makes sense to batch leaves together to reduce the I/O overhead.

Existing systems choose a fixed-size chunk and assign nodes to the chunk until it is full [2]. A natural implementation strategy is to perform a traversal of the tree (e.g., in depth-first order) and assign nodes to chunks. However, this approach fails to capitalize on the key property of Merkle-trees: that subtrees can be validated independently. Consequently, the snapshot cannot be validated until a peer has downloaded the complete state. Put another way, the peer must download all chunks before it can check if they are valid. Consequently, a single misbehaving peer serving an invalid chunk can initiate a type of "denial of service" attack, significantly prolonging the time it takes for a new peer to join the blockchain.

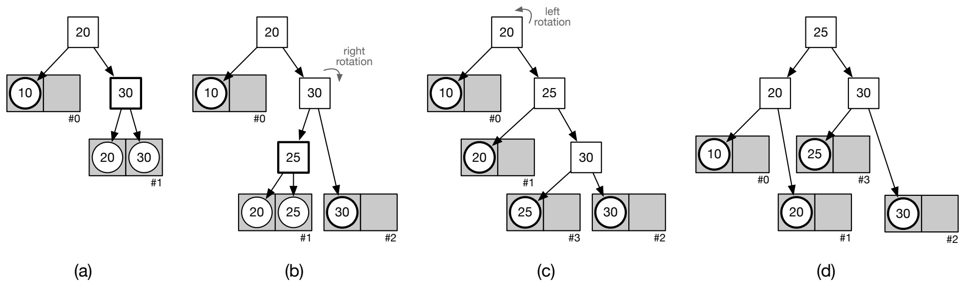


Figure 2 Insertion and rotations in AVL* (white square: inner node; white circle: leaf; gray rectangle: chunk with chunk id; bold square/leaf: chunk root). (a) a balanced AVL tree, (b) an imbalanced AVL tree after the insertion of 25, (c) an imbalanced AVL tree after right rotation of previous tree at inner node 30, (d) a balanced AVL tree after left rotation of previous tree at pivot (inner node 20).

3 The AVL* chunked tree

The AVL* tree is a Merkle-ized, balanced binary search tree. As an AVL+ tree (§3.1), it preserves the same invariant: the height difference between its left and right children is at most 1. The key difference between the two is that the leaf nodes in an AVL* tree are organized into *chunks*, or batches (§3.2), and each AVL* chunk can be validated individually. The AVL* tree requires new data structures (§3.3), and algorithms for searching (§3.4), inserting (§3.5), and deleting (§3.6) nodes. Moreover, organizing the tree in chunks has implications on re-balancing (§3.7), multi-versioning (§3.8), and the correctness of the tree (discussed in the Appendix). A chunked AVL* tree can be downloaded in parallel during state synchronization, and independently checked for integrity.

3.1 AVL+ trees

While the techniques described in the paper could be used with any Merkle-ized trees, we focus the discussion on AVL trees [12, 5]. An AVL+ tree is a self-balanced ordered binary tree that implements a key-value storage API, where all values are stored in leaves and inner nodes store keys, used to keep the tree ordered. Leaves store the hash of values, and inner nodes the hash of their children and keys. As an immutable data structure, all updates are performed using copy-on-write. To add a key-value pair, the tree is traversed from its root until a leaf is found. Then, a new inner node is created, from which the found leaf and the new leaf (with the key-value to be included) will descend.

When the height difference between the left and right subtrees of an inner node is greater than one, the tree is said to be imbalanced. To re-balance the tree, one or two rotations are needed involving the lowest imbalanced subtree, whose root is called pivot node. Figures 2 (b) and (c) show imbalanced trees with inner node 20 as pivot. A left rotation at the pivot is enough if the subtree on the right of the pivot is higher than the subtree on the left of the pivot (Figure 2 (c)). But a right rotation at the right child of the pivot must happen first, if the left side of the pivot's right child is higher than the right side of the pivot's right child. This is what happens in Figure 2 (b) since the left side of inner node 30, on the right of pivot 20, is higher than its right side. The cases for right and left-right rotations are symmetric.

3.2 Chunks

A chunk is a set of nodes that are grouped together, and serves as the unit of access for the persistent store, communication, and integrity check. Grouping the nodes of the tree into batches facilitates parallel downloads. But, the assignment of nodes to chunks is important. If done incorrectly, it would not allow individual chunks to be checked for integrity. Each chunk has a *root* which is defined as follows:

▶ **Definition 1.** *The root of a chunk C, root(C), is the lowest (i.e. deepest) node that has all the nodes in C as descendants.*

Given this definition, the important invariant preserved by AVL* when assigning nodes to chunks is as follows:

▶ **Property 1.** *For any chunks C_a and C_b, neither root(C_a) is a descendant of root(C_b) nor root(C_b) is a descendant of root(C_a).*

Property 1 ensures an overall minimal proof size for checking the integrity of a chunk. A client peer can check the integrity of chunk C with C and the proof that *root*(C) is a valid node. If *root*(C) is valid (see §2.2) then all data it stores is valid, including the hash of its subtree. The client peer then computes the hash that should be stored in *root*(C) from the leaves in C all the way up to *root*(C) and then checks whether the computed hash matches the hash stored in *root*(C).

Trivially, we see that this property would be satisfied if we were to assign every node to the same chunk, or assign each node to its own chunk, begging the question of how big a chunk should be? There is a trade-off: if a chunk is too large, then we lose the ability to download multiple chunks concurrently, but if a chunk is too small, we lose the benefits of batching. The chunk size is a constant set when the tree is instantiated. In our experiments, we empirically evaluate different chunk sizes.

The key challenge in designing the AVL* tree is updating the insertion and delete algorithms of the AVL+ tree to respect this invariant, despite tree rotations.

3.3 Data structures

In an AVL* tree, inner nodes are stored in volatile memory only (DRAM); leaves are embedded in chunks, which are stored in volatile memory and in a persistent store. Table 1 details the structure of inner nodes, leaf nodes, and chunks.

An inner node contains a *key*, used to search the tree; a pointer to a *chunk*, if the key is the root of the chunk; pointers to *left* and *right* nodes (inner or leaf); the *height* of the node; a *hash*, discussed below; and a boolean *is_leaf* that asserts that a pointer to the node references an inner node.[1]

A leaf node contains a key-value pair; a pointer to a *chunk*, if the key is the root of the chunk; an *i_node* pointer to an inner node, if the leaf's key is also an inner node; the *height* of the node, which is zero if the key is not stored as an inner node or the height of the inner node that stores the same key as the leaf; a *hash*; and a constant boolean *is_leaf*.

A chunk contains a unique chunk identifier *cid*, a number in $0..(m-1)$, where m is the number of chunks; the *version* of the chunk, the number of leaves stored in the chunk, *size*; a pointer to the chunk *root*, which can be an inner node or a leaf; and a set *leaf* with all the leaves stored in the chunk. A chunk can store up to C_p leaves, a parameter of the system.

[1] For simplicity, we assume that pointers can reference either inner nodes or leaves.

Table 1 The data structures used in the AVL*.

Inner node	
key	unique item identifier
chunk	pointer to chunk, if chunk root
left	pointer to the left node, inner or leaf
right	pointer to the right node, inner or leaf
height	the height of the node
hash	needed by the Merkle-ized tree
is_leaf	false
Leaf node	
key	unique item identifier
value	arbitrary data held in the node
chunk	pointer to chunk, if chunk root
i_node	pointer to matching inner node, if any
height	0 or i_node height (when sending chunk)
hash	needed by the Merkle-ized tree
is_leaf	true
Chunk	
cid	unique chunk id, starting in 0
version	version number of the chunk
size	number of leaves in the chunk
root	pointer to the chunk root, inner or leaf
leaf[0..$(C_p - 1)$]	set of leaf nodes in the chunk

Hashes are computed after all changes in the tree have been performed, that is, the blockchain block the tree corresponds to has been fully processed. The hash of a leaf takes as input the key, value, height of the tree, and the chunk id and version, if the leaf is the chunk root. The hash of an inner node includes the key, the hashes of its children, and the chunk id and version, if the leaf is the chunk root.

3.4 Search

Searching for a key in the AVL* tree is just like a search in a regular AVL tree. The only difference is that if the search traverses a part of the tree that is not stored in main memory, then the corresponding chunk is read from disk and its entire subtree is stored in main memory.

3.5 Insertion

The insertion starts by searching down the tree for a leaf with a key that is immediately smaller or bigger than the key to be inserted. This will determine the position of the new leaf, either left or right of the found leaf, with the key-value element to be inserted. When searching down the tree, the root of a chunk is eventually found. From Property 1 (see §3.2), only one root chunk is traversed when inserting an element in the tree. If the root chunk points to a full chunk, before searching further, the chunk is split. The split, detailed next, does not change the structure of the tree but ensures that there is an available position in the chunk to accommodate the new leaf. To insert the new element, one inner node is created, pointing to the found leaf and the new leaf. When unrolling the recursion that leads to the insertion of an element, the height of every visited tree node is recomputed. If the subtree rooted at the visited node becomes imbalanced, a rotation is executed.

For example, in Figure 2 we add a node with key 25 in the tree depicted in (a), which has two chunks. Chunk #0 has one leaf and chunk #1 is full with two leaves. The insertion starts from the tree root and traverses to the right child at node 30, the chunk root of chunk

#1, which is at maximum capacity, and splits it, creating chunk #2. The algorithm continues traversing the tree until leaf 20 is reached. Since the new key is bigger than the leaf's key, a new inner node is added copying the value of the new key and rearranging the leaves accordingly, resulting in the tree (b) before balancing is done.

To split a chunk, we create two new chunks and run a depth-first search (DFS) from the left and right children of the chunk root to be split. Each call to DFS builds a new chunk that is assigned to the left and right children of the old chunk root. At the end of the split the old chunk is deallocated and its chunk root set to nil.

3.6 Deletion

Deleting a node from the AVL* tree is similar to deleting a node from an AVL tree. We discuss next the basic procedure and then two extensions that account for chunks.

To delete key k, we start by searching the tree to find a pivot node p such that (i) p's left node is a leaf, or (ii) p is an inner node with key k. Then, there are four cases to consider:

- Case (a): p's left child is the leaf with key k. In this scenario, p's left node is deleted, p's right node takes the place of the pivot, and the original pivot is deleted. This case happens when we want to remove key 10 in Figure 2 (a), where the pivot is inner node 20.
- Case (b): p's right child is the leaf with key k. In this case, p is necessarily the inner node with the key to be deleted. Both p and its right child are deleted and p's left child takes the place of the pivot. This case happens when we want to remove key 30 in Figure 2 (a), where the pivot is inner node 30.
- Case (c): The left child of p's right child is the leaf with key k. The leaf with k is deleted, p is replaced with p's right child, and the original pivot is deleted. For this case, consider the deletion of key 20 in Figure 2 (a), where the pivot is inner node 20.
- Case (d): Otherwise, we find the inner node x with the lowest value at the sub-tree on the right of p, delete x's left leaf, replace pivot p with x, and delete the inner node p. We illustrate this case with the deletion of key 20 in Figure 2 (b), where the pivot is inner node 20 and x is inner node 25.

Akin to the AVL-tree deletion, when unrolling from the recursion that found pivot p, the nodes involved in the deletion have their heights updated and possibly rotated. Differently from the insertion, a deletion may involve rotations at all nodes in the way up to the root.

Deletion of a node in an AVL* tree differ from deletion in an AVL tree in two aspects. First, in cases (c) and (d) above, when an inner node x replaces a deleted inner node (i.e., the pivot). If x is a chunk root, then it will no longer be root, and the root of the chunk it referred to will be a node that descends from x.

Second, when deleting a leaf, it may happen that a chunk ends up with no leaves. This situation requires attention since the validity of a chunk is attested by the chunk root (discussed in §4), and an empty chunk has no root. To handle this case, when a chunk becomes empty, we take the chunk with the current largest unique id, assign to this chunk the id of the empty chunk, and decrement by one the number m of existing chunks (see §3.3). This ensures that every chunk with id in $0..(m-1)$ has at least one node, and therefore a chunk root.

3.7 Re-balancing

If, as a result of an insertion or a deletion, there is a height difference between two child subtrees, then the parent tree must be re-balanced. Similar to the AVL tree, a rotation in an AVL* tree happens if the height difference from the children of a node is greater than one. There can be four types of rotations: left, right-left, right, and left-right. Left and right rotations happen with a single rotation to the left or right, respectively. When rotating to the left on a pivot, the right child of the pivot takes the place of the pivot (called new pivot), the right child of the old pivot becomes the left child of the new pivot, and the left child of the new pivot becomes the old pivot. For the right-left rotation, first the right child of the pivot is rotated to the right and finally the pivot is rotated to the left. The right and left-right rotations are symmetrical to the cases explained before. The tree can be balanced with at most two rotations after an insertion.

If the rotation involves a chunk root, then there are two cases to consider. First, if the root of the chunk is the pivot of the rotation, then the node that takes the position of the pivot becomes the new chunk root. Second, if the pivot of the rotation is not the root of a chunk, but the node that takes the position of the pivot is the root of a chunk, then we split the chunk before doing the rotation. This is done to ensure Property 1. As a consequence, there will be extra splits even when a chunk is not full; in our experimental evaluation these splits amount for around 4% of the total number of splits.

Continuing from the example in Figure 2, the tree (b) is imbalanced after the insertion. Since the height of the root's right child is greater than the height of the root's left child, it triggers a left or a right-left rotation. Since the root's right child has a higher height on its left child than the right one we do a right-left rotation. First we rotate inner node 30 to the right and then rotate the inner node 20 (root) to the left. When rotating the inner node 30 to the right, we observe that its left child is a chunk root and split the chunk before continuing with the rotation. After this step, the tree is depicted in (c); observe that the tree has an extra chunk #3 created by the split and still is imbalanced. After the final rotation to the left, the tree is finally balanced as shown in (d).

3.8 Multi-versioning

An AVL* tree is multi-versioned, and a new version of the tree is created for every new blockchain block. For performance, a new tree is created using copy-on-write. Differently from an AVL+ tree, in the AVL* tree the unit of allocation is a chunk. This means that when a node is modified in the new version of the tree, the complete chunk that contains the node is copied. This chunk-based allocation suggests a tradeoff: small chunks reduce the overhead of creating the new tree, but increase the number of chunks that need to be propagated upon state synchronization. We experimentally evaluate this tradeoff in §5.

3.9 Correctness of tree operations

We initially argue that, in the absence of rotations, the insertion and deletion algorithms preserve Property 1.

In the case of an insertion, the property could only be invalidated when creating new chunks. When splitting a chunk, two new disjoint chunks are created and assigned to the left and right subtrees as the original chunk is extinguished. These steps do not violate Property 1 since they do not create descendants below or above each chunk root.

To see why deletion preserves Property 1, assume node x replaces the pivot, and x is root of its chunk. From the protocol, a descendant y of x becomes chunk root in place of x. Since x was chunk root, none of the nodes in its subtree are chunk root, and thus, no descendant of y is a chunk root.

We now argue that a right rotation preserves Property 1; the same reasoning applies for a left rotation. When rotating a node x to the right, x's left child is assigned as the right child of x's original left child. The only case a rotation would lead to a violation of Property 1 is when there are chunk roots in x's siblings. To deal with this case, before the assignment, we split the left chunk root in case of a right rotation (and right chunk root in case of a left rotation). After the split, the assigned node is a chunk root and can be safely moved to the opposite part of the tree.

4 Robust state synchronization

In general, there may be many ways to build a balanced binary tree from the same data. To be correct, we must ensure that all peers build the same tree. This is ensured if a peer collects all chunks that make up a tree, these chunks are valid, and the re-construction of the tree is deterministic. More formally, the tree reconstruction algorithm ensures the following two properties:

▶ **Property 2.** *Let T be the AVL* tree built by a honest peer after executing the n-th block of the blockchain, and $C_T = \{C_0, ..., C_{m-1}\}$ the chunks of T. Upon receiving chunk C_k from a peer, a client peer can check whether C_k is valid (i.e., $C_k \in C_T$) before receiving any other chunks in C_T.*

▶ **Property 3.** *Let T and T' be two AVL* trees with the same nodes. If we build T' by inserting node by node following their order of height in T, then T and T' are isomorphic.*

We now describe a procedure to reconstruct the tree that satisfies Properties 2 and 3. To check that T is valid, the client needs the trusted Merkle-root hash of T and the number m of chunks in this tree. Both the Merkle-root hash and the number of chunks in the tree must be included in the block headers of the blockchain to ensure that they are trusted. In a blockchain, this information is available in the headers of a block that succeeds the n-th block (e.g., in the $(n + 1)$-th block).

A client peer requests a chunk by its identifier (i.e., a value in $0..(m - 1)$) and receives as a response the requested chunk with the proof of inclusion of the chunk's root. The peer then rebuilds the subtree rooted at the chunk's root with all the nodes in the chunk. The procedure that checks the validity of the chunk (Property 2) proceeds in two steps. In the first step, the peer verifies the proof's validity by reconstructing the path from the chunk's root until the Merkle-root based on the hashes provided in the proof. The proof is valid if and only if the reconstructed Merkle-root matches the trusted one. In the second step, the peer recomputes the hashes of the subtree from the leaves up to the chunk root. The chunk is valid if the computed hash of the chunk root matches the hash provided, known to be valid from the first step.

To ensure that the client peer builds a proper subtree with the leaves in a chunk (Property 3), the peer first sorts all leaves by height. Recall from §3.3 that the height stored in a leaf is either 0, if the leaf's key is not an inner node, or the inner node's height in the tree. Then, the peer builds the tree by adding one node at a time, in descending order of the node height (i.e., root first). Multiple subtrees can be built in parallel to speed up the process. When the last chunk is built, the peer links all the chunk subtrees: The peer sorts all subtrees in descending order of their root node height, at which point the tree's root lies necessarily in the first position of the array. Each subtree is then added sequentially in the tree. When the last element is added the tree is complete. The correctness of this procedure is shown below. After the tree is built, the peer recomputes all the hashes.

4.1 Correctness state synchronization

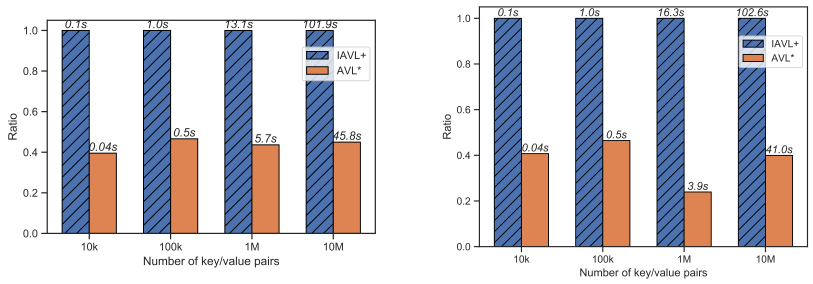
We initially consider Property 2. The first step of the protocol checks the validity of the chunk root using the proof of integrity of a single node of the tree. This follows immediately from the properties of Merkle trees (§2.2). A valid chunk root provides a trusted hash of its subtree. Then, in the second step, we compute the hashes from the leaves all the way up to the chunk root. If the trusted and the computed hash match, then the chunk is valid.

We now consider Property 3. Let h be the height of T, and $T(n)$ be a subtree of T that contains nodes from height h down to height n. The proof is by backwards induction on n.

Base step. Trivially true for $n = h$ since there is a single node with height h in subtrees $T(h)$ and $T'(h)$, the root.

Inductive step. We assume that $T(n+1)$ and $T'(n+1)$ are isomorphic and show that $T(n)$ and $T'(n)$ are isomorphic too, for $0 \le n < h$. Let k be a node in $T(n)$ that is not in $T(n+1)$. Since T is a binary search tree, there is only one path in $T(n+1)$ that leads to k. From the inductive hypothesis, $T(n+1)$ and $T'(n+1)$ are isomorphic, thus, k will end up in the same location in $T'(n)$. Since $T(0) = T$ and $T'(0) = T'$, we conclude that T and T' are isomorphic.

5 Evaluation



(a) 10 peers.

(b) 80 peers.

Figure 3 Time for a peer to recover from scratch, 100k chunks, varying number of key/value pairs.

5.1 Implementation and environment

To evaluate the behavior of our data structure and algorithms, we integrated them into Tendermint and conducted experiments under different conditions. Tendermint is a blockchain middleware that supports the replication of arbitrary applications. Tendermint provides applications with an AVL+ tree to manage state, called IAVL+. Peers can join the network by fetching a snapshot of the system state. To speed up the process, a peer can fetch a snapshot in chunks of fixed size.

All tests were conducted in a wide-area network (WAN) using Amazon's Elastic Computing (EC2) platform. We evaluated the system with 10 peers (small setup) and 80 peers (large setup). Our large setup is a fair approximation of Cosmos/Tendermint's current production system, with 125 peers. Peers were deployed in datacenters in seven Amazon regions: three datacenters in North America (Oregon, Ohio, Canada Central), two in Asia (Tokyo, Hong

Kong), three in Europe (Paris, Frankfurt, and London), one in South America (Sao Paulo), one in the Middle East (Bahrain), and one in Africa (South Africa). We used t3.xlarge and r5.xlarge instances.

We used most of Tendermint's default parameters, and set the mempool cache with 50k transactions, and block interval of 1 second for executions with 10 peers and 5 seconds for for executions with 80 peers. These parameters led to the best results for throughput and latency for both the IAVL+ and the AVL* experiments. In the setup with 10 peers, each peer is connected to every other peer; in the setup with 80 peers, each peer is connected to 25 random peers.

We developed a key-value store application using Tendermint and benchmarked the application using the IAVL+ tree and the AVL* tree. Clients submit transactions that add new key-value pairs to the store. A key contains 20 bytes and a value contains 100 bytes, both generated randomly by clients. We evaluated the IAVL+ and the AVL* trees with chunks that can contain up to 10k and 100k key-value pairs, amounting to roughly 1MB and 10MB chunks, respectively. We considered executions in which clients include 10k, 100k, 1M and 10M key-value entries to the store.

5.2 State synchronization

The first set of experiments evaluates the performance of the state synchronization operation. The main metric of concern is the time it takes to perform synchronization for a new peer joining the blockchain.

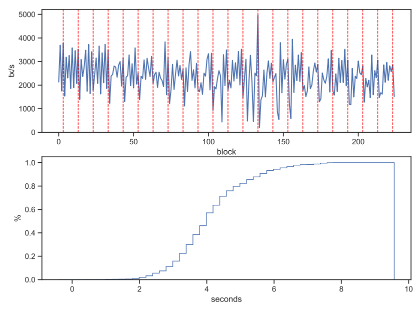
In these experiments, all peers but one are pre-initialized with a full tree. When the experiment begins, the new peer recovers the state by downloading chunks in parallel from the operational peers. We vary three parameters: (i) the size of the tree, (ii) the number of validators (10 and 80), and (iii) the size of the chunks (10k and 100k).

Figure 3 shows the state synchronization times for IAVL+ and AVL*. The results for 10k chunks and 100k chunks are similar; thus, we show results for 100k chunks only. We report the results as a ratio, with the absolute time printed at the top of each column. There are two important features to note. First, for both IAVL+ and AVL*, the synchronization time increases linearly with the size of the tree, and it does not depend on the size of the system. Second, the synchronization time for AVL* is roughly half the time required by IAVL+. This happens because in the AVL*, when a chunk is received, it can be validated individually and, if valid, the subtree it contains can be built before all chunks are received.

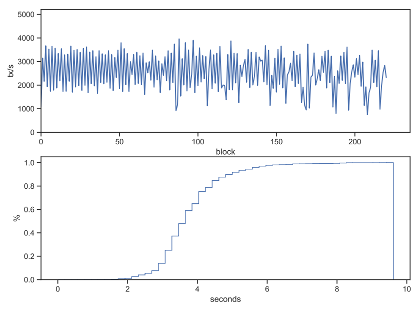
Table 2 Throughput and latency for IAVL+ and AVL* in different configurations.

	10 peers							
	10k chunks				100k chunks			
	Throughput (tx/s)		Latency (s)		Throughput (tx/s)		Latency (s)	
	average	std.	average	std.	average	std.	average	std.
IAVL+	2373.86	764.03	4.18	1.1	2362.76	713.46	4.16	1.02
AVL*	2538.63	798.46	3.85	0.94	2491.69	802.94	3.96	0.99
variation	+7%		-8%		+5%		-5%	

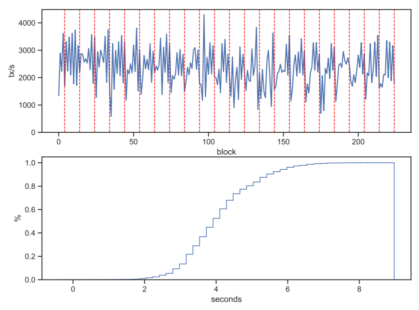
	80 peers							
	10k chunks				100k chunks			
	Throughput (tx/s)		Latency (s)		Throughput (tx/s)		Latency (s)	
	average	std.	average	std.	average	std.	average	std.
IAVL+	893.28	218.33	8.63	2.27	892.82	235.59	8.71	2.6
AVL*	988.42	124.39	7.64	1.53	1001.25	131.8	7.54	1.55
variation	+11%		-11%		+12%		-13%	



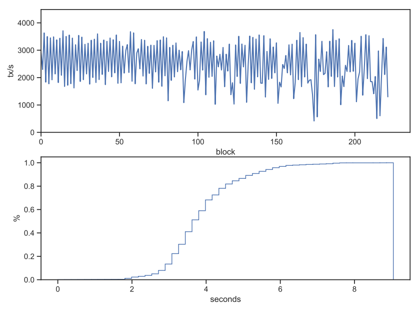
(a) IAVL+ with 10k chunks.



(b) AVL* with 10k chunks.



(c) IAVL+ with 100k chunks.



(d) AVL* with 100k chunks.

Figure 4 Throughput and latency of transaction execution, 10 peers, 1M key/value pairs.

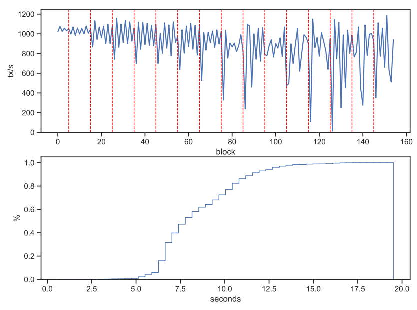
We do note one detail. Recall that the AVL* does not guarantee fixed-sized chunks. So, in these experiments, the AVL* chunks tend to be smaller than those made by snapshots in the IAVL+. Snapshots from the AVL* have around 16% more chunks than the IAVL+ tree. However, although there are more chunks in the application using the AVL* tree, the data stored in each chunk is increased, since it includes the proofs of inclusion for each chunk.

Overall, the state synchronization time when using the AVL* is on average 58% faster compared to the IAVL+ tree, despite having to download more chunks.

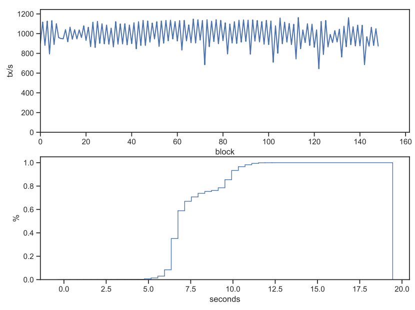
5.3 Steady-state operation

In the second set of experiments, we compare transaction throughput and latency of Tendermint using an AVL* and an IAVL+ tree. One distinctive aspect of AVL* is that peers do not need to periodically build state snapshots. To understand the overhead of IAVL+, we measure the time it takes to compute a snapshot. Finally, we assess AVL* space utilization.

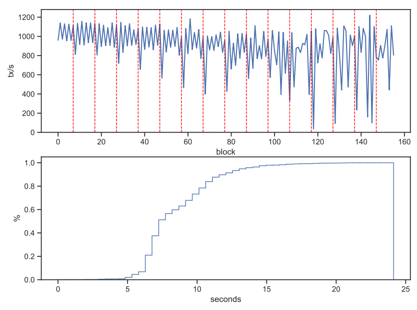
In all the experiments, clients operate in a closed-loop, meaning a client only submits a new transaction after it receives the response for the previously submitted transaction. The client subscribes to the blockchain and delivers the blockchain blocks. Latency is computed as the time it takes for a transaction to be included in a block. Throughput is the number of transactions in a block divided by the time it took for the block to be ordered (i.e., interval between the current block and the previous one). In the experiments with the IAVL+ tree, snapshots are built every ten blocks.



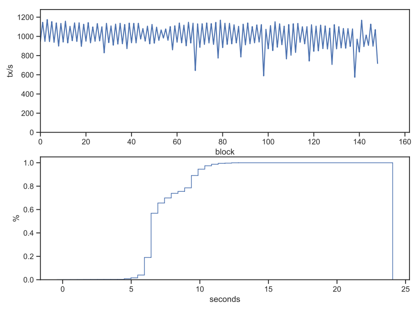
(a) IAVL+ with 10k chunks.



(b) AVL* with 10k chunks.



(c) IAVL+ with 100k chunks.



(d) AVL* with 100k chunks.

Figure 5 Throughput and latency of transaction execution, 80 peers, 1M key/value pairs.

5.3.1 Steady-state performance

We show the throughput as a time series and the latency CDF for Tendermint using both IAVL+ and AVL*. Figure 4 presents the results for a blockchain with 10 peers. Figure 4 (a) and (b) report the results for chunks of size 10k, and Figure 4 (c) and (d) report results for chunks of size 100k. Figures 5 shows the results for the same set of experiments when the number of peers is increased to 80.

The graphs show that using the AVL* tree results in more predictable performance. One of the reasons for the volatility of the IAVL+ tree is that there are periodic drops in performance that occur during a snapshot operation. In the graph, the dashed-red lines indicate the time that a snapshot occurs.

Although it is somewhat difficult to tell from the graphs, the performance of AVL* is not only more predictable, but it is also better, on average, than the IAVL+. Table 2 shows the mean values, the corresponding standard deviation, and the relative improvement of AVL* over IAVL+ for all these experiments.

In a blockchain with 10 validators, AVL* increases the throughput and reduces the latency by at least 5% when compared to the IAVL+ tree. These improvements increase with the size of the blockchain: with 80 validators, the improvements in throughput and latency are at least 11%.

5.3.2 Time for a snapshot

One of the major differences between IAVL+ and AVL* is that AVL* trees do not need to pause execution to compute a snapshot. This happens because in AVL* trees, chunks are an integral part of the tree data structure. In IAVL+, chunks are built from tree snapshots. Because snapshot computation has a significant impact on the IAVL+ performance, we wanted to quantify the overhead.

Figure 6 shows the time it takes to compute a snapshot with IAVL+ trees as the number of tree leaves (i.e., key-value pairs) is increased, varying from 0 to one million leaves. As the tree gets bigger, snapshots take more time to complete. That happens because each snapshot has to serialize the whole tree. Changing the chunk sizes does not have a significant impact on the experiment's performance.

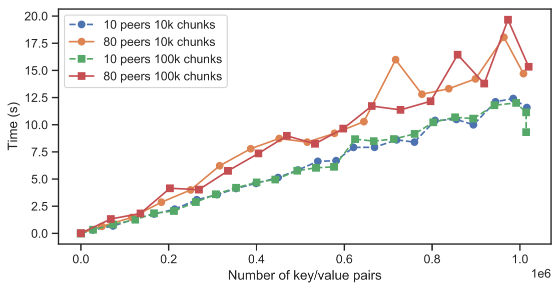


Figure 6 Time for an IAVL+ snapshot.

5.3.3 Space efficiency

The trade-off for using an AVL* is space efficiency, as the algorithm may fill chunks up to their capacity. We define space efficiency as the number of chunks in the AVL* divided by the ideal number of chunks (ceiling of elements divided by chunk size). For instance, if the space efficiency were 2, it would mean that the AVL* uses twice as much chunks as the ideal scenario. Note that when creating snapshot for the IAVL+, the space efficiency is always 1, because the serialization process always serializes the tree from scratch. We evaluated the space efficiency of the AVL* tree as we insert one million random keys into an empty AVL* tree. The space efficiency stabilizes at around 1.4 for random data, which we believe is an acceptable overhead.

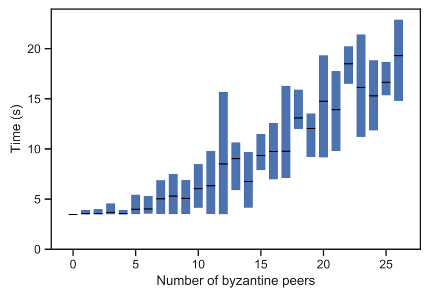
5.4 State synchronization under attack

A key benefit of an AVL* tree over the IAVL+ tree is that it can gracefully recover from Byzantine or malicious behavior from peers. An IAVL+ cannot be checked for validity until the entire tree has been downloaded and reconstructed. If, after reconstructing the tree, the tree's root hash is different from the one in the trusted block header, then the client peer must refetch all chunks again. In contrast, the AVL* tree allows the client peer to detect invalid chunks easily, and remove misbehaving peers from their peer list.

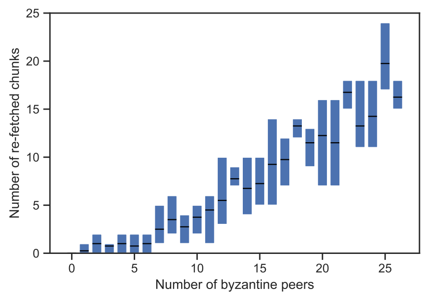
To quantify the performance of AVL* in the presence of malicious peers, we again performed the state synchronization experiment, but introduced malicious peers that respond with invalid chunks. In these experiments, we used a fixed number of 80 peers and a tree with one million entries. We varied the number of malicious peers from 1 to 26. We performed the experiment five times, and report mean synchronization time, as well as the maximums and minimums in bars and whiskers. Figure 7 shows the results.

In Figure 7 (a), we see that, as expected, the presence of malicious peers degrades the state synchronization time. The state synchronization time grows linearly with the number of malicious peers. Because a peer can respond to multiple requests for chunks in parallel before they are verified, the number of invalid chunks sent by peers can vary. In Figure 7 (b) we see that the number of chunks that need to be re-fetched is proportional to the state synchronization time.

From §5.2, we know that without any byzantine peers, it takes 16.3 seconds for a client peer using IAVL+ to complete state synchronization (with 80 peers, 100k chunks and 1M key-value pairs). With one byzantine peer, this time would double since the client peer would have to start from scratch. And even after retrying, there is no guarantee that the second attempt would succeed. Thus, a coordinated attack could substantially increase state sync time of IAVL+. AVL* outperforms IAVL+ even under attack by 20 byzantine peers.



(a) Time for state synchronization.



(b) Number of re-fetched chunks.

Figure 7 State sync with byzantine peers, 100k chunks, 1M key/value pairs.

6 Related work

Many systems for state machine replication with byzantine actors have addressed the problem of fast state synchronization without executing the entire transaction log. This is typically done by taking snapshots of the state, often called checkpoints. Such checkpoints can then be downloaded by new or recovering peers. While PBFT [17] proposed the use of a Merkle tree for its checkpoints, the Upright [18] and BFT-SMaRt [13] systems consider Merkle trees and copy-on-write semantics to be too invasive in general to the application developer. Upright outlines three simple approaches to state transfer [18], and BFT-SMaRt [13] describes a detailed Collaborative State Transfer protocol, where the full state is downloaded from a single peer and verified against hashes from other peers. In PBFT, a binary Merkle-tree is built at each checkpoint by partitioning the application state in 4KB pages. The pages are stored as leaves of the Merkle-tree using copy-on-write to only persist pages that have been modified since the previous checkpoint. This approach is not efficient to encode a key-value storage, since operations on the key space (e.g., searches) do not have logarithmic complexity.

Unlike SMR systems, which either consider Merkle-trees too expensive [18, 13], or construct them only for state synchronization at periodic checkpoints [17], many blockchain systems already use Merkle-ized data structures to store the state. The primary use case for such Merkle-trees is to facilitate light clients, who can efficiently query for particular leaves of the tree and verify their integrity, without ever downloading the entire state or transaction history. The use of Merkle-trees for state synchronization has received much less attention, but as the state of blockchain systems grow, synchronizing it becomes more expensive.

In Geth [7], state synchronization is performed by requesting individual nodes of the tree. Peers do not have a way of knowing how long the state synchronization will last, because they do not know the total number of nodes [4]. Since the Merkle-tree is part of the consensus rules (i.e., Merkle-roots are stored in block headers), peers can verify that a received node from the tree is correct. However, given the small size of nodes (less than one KB), their randomized distribution in the underlying database, and the large size of the state (tens of GBs), requesting nodes individually leads to performance degradation for peers requesting and providing nodes. Batching nodes is a promising solution, however, it is challenging to batch nodes in a manner that can be securely verified by peers, and limits attacks on honest peers. For instance, in OpenEthereum [8], snapshots are taken periodically by serializing the entire state, and dividing it in large chunks. The hashes of each chunk are published in a manifest file. Since the manifest is not part of the consensus process, there is no way to verify that a chunk is correct before downloading all of them. Successfully completing the state synchronization in such a system thus depends on retrieving a correct manifest, which requires strong assumptions, for instance, that a particular peer can be trusted or that a majority of connected peers are correct. This is stronger than the usual assumption of a single (though unspecified) correct peer commonly used in blockchain systems.

Other blockchain systems have proposed to take snapshots of the Merkle-tree by periodically dividing it in chunks. In Codechain snapshots [9], chunks are built from nodes within a certain depth from a common root, and the hash of the snapshot is included in the block header so chunks can be verified incrementally by peers. Chunks may contain entire sub-trees, or may be limitted to the upper nodes in a sub-tree. In Tendermint IAVL+ snapshots, tree nodes are serialized in order and assembled into chunks of a given size [1]. However, since Tendermint's block header does not currently support snapshot hashes, chunks cannot be incrementally verified by peers. Hence the need for a tree that incorporates chunking directly into its structure.

Motivated by their use in the blockchain context, numerous Merkle-tree designs have been proposed lately. TurboGeth [10] separates the key-value storage from the Merkle tree structure, and batches Merkle tree nodes in chunks to reduce the number of lookups during Merkle operations. This has the effect of greatly improving performance without changing the structure of the hash tree itself. Sparse Merkle-trees [19] have been adopted by other blockchain projects [11]. Recent advances in cryptography have even offered a glimpse into generalizations of Merkle trees called accumulators, which enable O(1) proofs of set-membership and state-less blockchain clients [14]. While there has been a wide diversity of proposed and implemented tree designs, the AVL* is the only known tree to target both the light client and state synchronization use cases found in blockchain systems.

7 Conclusion

State synchronization is a significant bottleneck for blockchain-based systems. In this paper, we have presented a novel extension to Merkle-ized AVL+ trees that incorporates chunks. This extension allows peers to download portions of the state in parallel, and validate each chunk independently. We have also described an algorithm for deterministic tree reconstruction, ensuring that all peers have the same state. We have extensively tested our algorithms in a geo-distributed environment that show the benefits when reconstructing the state from snapshots and gracefully coping with byzantine failures.

References

1 Adr 053: State sync prototype. https://github.com/tendermint/tendermint/blob/master/docs/architecture/adr-053-state-sync-prototype.md.

2 Cosmos network. https://cosmos.network/.

3 Cosmos sdk. https://github.com/cosmos/cosmos-sdk.

4 Go ethereum faq. https://geth.ethereum.org/docs/faq.

5 Iavl+ implementation. https://github.com/tendermint/iavl.

6 The interblockchain communication protocol. https://github.com/cosmos/ics/blob/master/spec.pdf.

7 Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum.

8 Openethereum warpsync. https://openethereum.github.io/wiki/Warp-Sync.

9 Snapshot sync proposal. https://research.codechain.io/t/snapshot-sync-proposal/21.

10 Turbo-geth programmer's guide. https://github.com/ledgerwatch/turbo-geth/blob/master/docs/programmers_guide/guide.md.

11 Urkel tree implementation. https://github.com/handshake-org/urkel.

12 George M Adel'son-Vel'skii and Evgenii Mikhailovich Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

13 Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 169–180, 2013.

14 Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

15 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. arXiv:1807.04938.

16 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002. doi:10.1145/571637.571640.

17 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

18 Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.

19 Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.

20 Ittay Eyal and Emin Gün Sirer. Majority is not enough: bitcoin mining is vulnerable. *Commun. ACM*, 61(7):95–102, 2018.

21 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58, 2007. doi:10.1145/1294261.1294267.

22 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

23 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, 1987. doi:10.1007/3-540-48184-2_32.

24 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, bitcoin, 2008.

25 Michael Szydlo. Merkle tree traversal in log space and time. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 541–554. Springer, 2004.

26 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

A Appendix: Algorithms

Algorithm 1 Insert.

```
next_cid := 0                                                    ▷ next chunk identifier
procedure INSERT(node, key, val)                                 ▷ node is a pointer to root
    return INSERT_AUX(node, key, val, $\epsilon$)                ▷ return ptr to new node
procedure INSERT_AUX(node, key, val, last_chunk)
    if node = $\epsilon$ then                                    ▷ if the tree is empty:
        chunk := NEW_CHUNK(next_cid)                             ▷ first chunk
        node := INSERT_IN_CHUNK(chunk, key, val)
        node→chunk := chunk                                      ▷ node becomes chunk root
        chunk→root :=node                                        ▷ ditto
        return node
    if node→chunk $\neq \epsilon$ then                           ▷ if node is the chunk root:
        c := node→chunk                                          ▷ let c be this chunk
        if $c$ →size = $C_p$ then                                ▷ if c is a full chunk:
            SPLIT_CHUNK(node)                                    ▷ split node (i.e., c's root)
    else
        c := last_chunk                                          ▷ chunk root is a higher node
    if node→is_leaf then                                         ▷ if node is a leaf:
        node_chunk := node→chunk                                 ▷ keep its chunk, if any
        node→chunk := $\epsilon$                                 ▷ node can't be chunk root
        if key < node→key then                                   ▷ normal AVL left insert
            left := INSERT_IN_CHUNK(c, key, val)
            node := NEW_INNER(node→key, left, node, 1)
        else                                                     ▷ normal AVL right insert
            right := INSERT_IN_CHUNK(c, key, val)
            node := NEW_INNER(key, node, right, 1)
        node→right→inner_node := node                            ▷ leaf points to its inner
        node→chunk := node_chunk                                 ▷ inner gets kept chunk
        if node→chunk $\neq \epsilon$ then
            c→root := node
    else                                                         ▷ if node is an inner node:
        if key < node→key then                                   ▷ go down left or...
            node→left := INSERT_AUX(node→left, key, val, c)
        else                                                     ▷ ...go down right...
            node→right := INSERT_AUX(node→right,key,val,c)
    UPDATE_HEIGHT(node)                                          ▷ new height from children heights
    return BALANCE(node)                                         ▷ if needed, rotate to keep balance
procedure UPDATE_HEIGHT(node)
    node→height :=                                               ▷ height gets max children height plus one
                                              max(node→left→height,node→right→height) + 1
procedure BALANCE(node)
    h := node→left→height - node→right→height
    if h < -1 then                                               ▷ if right branch is bigger than left branch:
        return ROTATE_RL(node)                                   ▷ rotate [right] left
    if h > 1 then                                                ▷ if left branch is bigger than right branch:
        return ROTATE_LR(node)                                   ▷ rotate [left] right
    return node
```

Algorithm 2 Auxiliary procedures.

```
 1: procedure NEW_INNER(key, ln, rn, ht)                          ▷ create inner node
 2:     new_inode := allocate new inner node
 3:     new_inode→key := key
 4:     new_inode→left := ln
 5:     new_inode→right := rn
 6:     new_inode→height := ht
 7:     return new_inode
 8: procedure NEW_CHUNK(cid)                                      ▷ create chunk
 9:     new_c := allocate new chunk
10:     new_c→cid := cid
11:     new_c→size := 0
12:     new_c→root := $\epsilon$
13:     return new_c
14: procedure INSERT_IN_CHUNK(c, key, val)                        ▷ insert key,val
15:     c→leaf[c→size].key := key
16:     c→leaf[c→size].value := val
17:     node_addr := pointer to c→leaf[c→size]
18:     c→size := c→size + 1
19:     return node_addr
20: procedure DELETE_FROM_CHUNK(c, key)
21:     for node in c→leaf do
22:         if node.key = key then                                ▷ found leaf
23:             SWAP(node, c→leaf[c→size])                        ▷ swap with the last
24:     deallocate c→leaf[c→size]                                 ▷ free last leaf
25:     c→size := c→size - 1                                      ▷ decrement number of leaves
26:     if c→size = 0 then                                        ▷ chunk is empty
27:         next_cid := next_cid - 1                              ▷ decrement number of chunks
28:         last_chunk := GET_CHUNK(next_cid)                     ▷ get chunk with highest id
29:         last_chunk→cid := c→cid                               ▷ replace chunk's id
30:         deallocate c                                          ▷ free empty chunk
31: procedure SPLIT_CHUNK(node)                                   ▷ split chunk rooted at node
32:     new_c := NEW_CHUNK(node→chunk→cid)
33:     DFS(node→left, node, new_c)
34:     node→left→chunk := new_c
35:     new_c→root := node→left                                   ▷ assign left chunk
36:     next_cid := next_cid + 1
37:     new_c := NEW_CHUNK(next_cid)
38:     DFS(node→right, node, new_c)
39:     new_c→root := node→right                                  ▷ assign right chunk
40:     node→right→chunk := new_c
41:     deallocate node→chunk
42:     node→chunk := $\epsilon$                                  ▷ x is no longer chunk root
43:     return
44: procedure DFS(ptr, pnt, c)
45:     if ptr→is_leaf then
46:         c→leaf[c→size].key := ptr→key
47:         c→leaf[c→size].value := ptr→value
48:         c→leaf[c→size].i_node := ptr→i_node
49:         if ptr→key < pnt→key then                             ▷ assign parent
50:             pnt→left := pointer to chunk→leaf[c→size]
51:         else
52:             pnt→right := pointer to chunk→leaf[c→size]
53:         c→size := c→size + 1                                  ▷ one more leaf in chunk
54:     else
55:         DFS(ptr→left, ptr, c)
56:         DFS(ptr→right, ptr, c)
57:     return
```

Algorithm 3 Delete.

```
procedure DELETE(node, key)
    if node.key = key and node.is_leaf then                         ▷ only one node
        DELETE_FROM_CHUNK(node.chunk, key)
        return $\epsilon$
    return DELETE_AUX($\epsilon$, node, key, $\epsilon$)                          ▷ return ptr to new node
procedure DELETE_AUX(node, key, last_chunk)
    if node→chunk $\neq \epsilon$ then                                 ▷ if node is the chunk root:
        c := node→chunk                                              ▷ let c be this chunk
    else
        c := last_chunk                                      ▷ chunk root is a higher node
    if node→left→key = key and node→left→is_leaf then
        if c = $\epsilon$ then                                  ▷ chunk is necessarily on the left
            c := node→left→chunk
        DELETE_FROM_CHUNK(c, key)
        promoted := node→right
        if node→chunk $\neq \epsilon$ then
            promoted→chunk := node→chunk                             ▷ node is a chunk root
        deallocate node                                           ▷ no need for inner-node
        return promoted
    if node→key = key and node→right→is_leaf then
        if c = $\epsilon$ then                                 ▷ chunk is necessarily on the right
            c := node→right→chunk
        DELETE_FROM_CHUNK(c, key)
        promoted := node→left
        if node→chunk $\neq \epsilon$ then
            promoted→chunk := node→chunk                             ▷ node is a chunk root
        deallocate node                                           ▷ no need for inner-node
        return promoted
    if node→key = key and node→right→left→is_leaf then
        if c = $\epsilon$ then                                                ▷ chunk is lower
            if node→right→chunk $\neq \epsilon$ then                            ▷ chunk on right
                c := node→right→chunk
                node→rightNode→rightNode→chunk = c
                node→chunk = $\epsilon$
            else
                c := node→right→left→chunk                               ▷ chunk on the left
        DELETE_FROM_CHUNK(c, key)
        aux := node→right
        node→key := aux→key
        node→right := aux→right
        deallocate aux
        return node
    if node→key = key then
        n, p := DELETE_LEAF(node→right, c)
        node→key = n→key
        node→right := p
        deallocate n
    else
        if key < node→key then
            node→left := DELETE_AUX(node→left, key, c)
        else
            node→right := DELETE_AUX(node→right, key, c)
    UPDATE_HEIGHT(node)                              ▷ new height from children heights
    return BALANCE(node)                              ▷ if needed, rotate to keep balance
```

Algorithm 4 Rotations.

```
 1: procedure ROTATE_RL(node)                          ▷ normal AVL [right] left rotation
 2:     rl_height := node→right→left→height
 3:     rr_height := node→right→right→height
 4:     if rl_height > rr_height then
 5:         node→right := ROTATE_R(node→right)
 6:         UPDATE_HEIGHT(node)
 7:     return ROTATE_L(node)
 8: procedure ROTATE_LR(node)                          ▷ normal AVL [left] right rotation
 9:     ll_height := node→left→left→height
10:     lr_height := node→left→right→height
11:     if ll_height < lr_height then
12:         node→left := ROTATE_L(node→left)
13:         UPDATE_HEIGHT(node)
14:     return ROTATE_R(node)
15: procedure ROTATE_L(node)                           ▷ node is the pivot
16:     if node→chunk $\neq \epsilon$ then             ▷ if subtree rooted at node in a chunk:
17:         node→chunk→root := node→right
18:         node→right→chunk := node→chunk             ▷ node's right child...
19:         node→chunk := $\epsilon$                   ▷ ...becomes new chunk root
20:     else                                           ▷ else, rotation may involve two chunks
21:         if node→right→chunk $\neq \epsilon$ then   ▷ if r child is chunk root:
22:             SPLIT_CHUNK(node→right)
23:     new_pivot := node→right                        ▷ rotation in three steps: one, ...
24:     node→right := new_pivot→left                   ▷ ...two, and...
25:     new_pivot→left := node                         ▷ ...three!
26:     UPDATE_HEIGHT(node)                            ▷ update moved node height
27:     UPDATE_HEIGHT(new_pivot)                       ▷ update moved node height
28:     return new_pivot
29: procedure ROTATE_R(node)                           ▷ node is the pivot
30:     if node→chunk $\neq \epsilon$ then             ▷ if subtree rooted at node in a chunk:
31:         node→left→chunk := node→chunk              ▷ node's left child...
32:         node→chunk := $\epsilon$                   ▷ ...becomes new chunk root
33:     else                                           ▷ else, rotation may involve two chunks
34:         if node→left→chunk $\neq \epsilon$ then    ▷ if l child is chunk root:
35:             SPLIT_CHUNK(node→left)
36:     new_pivot := node→left                         ▷ rotation in three steps: one, ...
37:     node→left := new_pivot→right                   ▷ ...two, and...
38:     new_pivot→right := node                        ▷ ...three!
39:     UPDATE_HEIGHT(node)                            ▷ update moved node height
40:     UPDATE_HEIGHT(new_pivot)                       ▷ update moved node height
41:     return new_pivot
```