Bitxor Blockchain

Technical Reference

Version 1.1.0.1

June 15, 2022

Document Version: 1.0

"We thank you God"

Contents

Special Introduction

Bitxor had its beginnings as a test project between a community of buying and selling Cryptocurrencies (OTC), we initially held the decision to develop a collection token with the Tron network, which aims to maintain the core essence of decentralization throughout our **CosmoSystem** (August 2022).

Shortly thereafter, a multidisciplinary team of highly talented professionals from Latin America and companies such as **Kriptxor Corp, Microsula S.A. and Focus On Results S.A.**, where new ideas began to emerge, including the development of our own Blockchain.

Although there were initially disagreements about what to primarily build, we quickly decided to create something new and different. This allowed for the expansion of all design ideas, as well as the use of high coding standards. so, it gave us the opportunity to contribute something new to the blockchain landscape. As a result of a lot of effort, mainly at night or on weekends due to the various schedules available among the community, the main network was launched. **Bitxor**.
In this process we remained satisfied with what was built, but we knew that we took some shortcuts, so we should continue to improve it constantly and allow faster innovation in the future.

We thank everyone for being a contribution of inspiration to build a completely different chain: Bitxor. Our main goal was to build a high-performance *blockchain* with strong supports, for deterministic completion. In this we believe and this is how we achieve it.

This has been a long journey for all of us, but we are excited to see what is to come and what new things we bring.

``**Bitxor** protocol is made to build´´

Typographical Conventions

Description	attributes	Example
Project's name	bold, colored	**Bitxor**
Lessons	fixed width font	Xolonium
fields	fixed width font	NotoSans /Arial
file paths	fixed width font	I will compromise _step.dat
Configuration settings (preceded by configuration filename) in text mode	italic	*red:maxDifficulTy Blocks*
Configuration settings (no configuration) ration filename) in equations and formulas. the	regular	maxDifficultyBlocks
Fields in equations and formulas	regular	*T*::Public key of the signer
Function Names in Equations and Formulas the	regular	verifiable data buffer()

Table 1: Typographic conventions used in this document

1 Introduction

The protocol **Bitxor** It is based on Blockchain with a multi-layer architecture, high performance and without rust. - these are the first principles that influenced the development of **Bitxor**. Weather A handful of DLT protocols were considered, a blockchain protocol was quickly chosen because it was deemed truer to the ideal of lack of trust. Any node can download a complete copy of the chain and be able to verify it independently at all times. Nodes with enough gathering power can always create blocks and never need to rely on a leader. These options necessarily sacrifice some performance relative to other protocols, but seem more consistent with the philosophical underpinnings of Bitcoin.

Bitxor supports both probabilistic and deterministic block completion. Under probabilistic completion, the probability of any particular block being reversed decreases as more and more blocks are added, or chained together. Although the probability can become very small, it is always nonzero.[1]. In contrast, deterministic completion includes a mechanism in the protocol that allows checkpoints to be set that can never be reversed. This can lead to potentially deep reversals, but provides stronger collateral. In either case, users bear the risk of block reversals and transaction invalidation.

As part of a focus on lack of trust, the following features have been added:

- Block headers can be synchronized without transaction data, while allowing chain integrity verification.
- Transaction Merkle trees allow cryptographic proofs of contention (or not) of transactions within blocks.
- Receipts increase the transparency of indirectly triggered state changes.
- Proofs of state allow trustless verification of a specific state within the blockchain.

In **Bitxor**, there is a single server executable that can be customized by loading different plugins (for transaction support) and extensions (for functionality). exist

[1] When probabilistic completion is enabled, for performance reasons, at most *network:maxRollbackBlocks* is allowed to roll back. Older blocks are assumed, but not guaranteed, to be finished because the probability of their reverting is quite low.

three main configurations (per network), but more custom hybrid configurations are possible by enabling or disabling specific extensions.

The three main configurations are:

1. Peer: These nodes are the backbone of the network and create new blocks.
2. API: These nodes store data in a MongoDB database for easy querying and can be used in conjunction with a NodeJS REST server.
3. Dual: These nodes perform the functions of the Peer and API nodes.

A strong network will typically have a large number of peer nodes and enough API nodes to support incoming client requests. Allowing the composition of nodes to vary dynamically based on actual needs should lead to a more globally resource-optimized network.

Basing the core block and transaction pipelines on the circuit breaker pattern, and using parallel processing whenever possible, allows for high transaction rates per second relative to a typical blockchain protocol.

1.1 Network fingerprint

Each network has a unique fingerprint that is made up of the following:

1. Network Identifier: Two-byte identifier that can be shared across networks. All addresses supported by a network must have this value as the first two bytes.
2. Generation Hash Seed: 32-byte random value that must be globally unique across all networks. This value is prepended to the transaction data before the hash or signature to prevent cross-network replay attacks.

2 System

Bitxor supports high customization at both network and node level individual. The network-wide configuration, specified in the network, must be the same for all nodes in one the net. Rather, node-specific settings may vary between all nodes in the same network, and are located on the node.

Bitxor was designed to use a *plugin / extension* approach instead of supporting Turing complete smart contracts. While the latter may allow for more user flexibility, it is also more error-prone from the user's perspective. A plugin model limits the operations that can be performed on a blockchain and consequently has a smaller attack surface. Furthermore, it is much easier to optimize the performance of a discrete set of operations than an infinite set. This helps **Bitxor** achieve the high performance for which it was designed.

2.1 Transaction plugins

All nodes in a network must support the same types of transactions and process them in exactly the same way so that all nodes can agree on the global state of the blockchain. The network requires each node to load a set of transaction plugins, and this set determines the types of transactions that the network supports. This set is determined by the presence of network: plugin* sections in the network configuration. All nodes in the network must coordinate and agree to any changes, additions, or removals of these plugins. If only a subset of nodes agree to these modifications, those nodes will be in a fork. all incorporated **Bitxor** Transactions are built using this plugin model to validate their extensibility.

A plugin is a separate dynamically linked library that exposes a single entry point in the following form[2]:

```
External"C" PLUGIN_API
empty Logging subsystem (
        bitxor::plugins::manager and plugin manager);
```

The plugin manager provides access to the subset of settings that plugins need to initialize. Through this class, a plugin can register zero or more of the

[2]The format of plugins depends on the target operating system and compiler used, so all host applications and plugins must be built with the same compiler version and options.

Following:

1. Transactions: New transaction types and the mapping of those types to scan rules can be specified. Specifically, the plugin defines rules to translate a transaction into component notifications that are used in post processing. Some processing restrictions can also be specified, such as stating that a transaction can only appear within an aggregate transaction (see6.2:Added transaction).
2. Caches: New cache types and rules can be specified for serializing and unrealizing model types to and from binaries. Each state-related cache can optionally be included in a block's StateHash computation (see7.3:state hashes) when that feature is enabled.
3. Controllers: These are APIs that can always be accessed.
4. Diagnostics: These are APIs and counters that can only be accessed when the node is running in diagnostic mode.
5. Validators: Stateless and stateful validators process notifications produced by block and transaction processing. Registered validators can subscribe to general or plugin-defined notifications and reject disallowed values or state changes.
6. Observers: Observers process notifications produced by block and transaction processing. Registered observers can subscribe to general or plugin-defined notifications and update the status of the blockchain based on their values. Observers do not require any validation logic because they are only called after all applicable validators succeed.
7. Resolvers: Custom mappings from unresolved to resolved types can be specified. For example, this is used by the namespace plugin to add support for alias resolutions.

2.2 Bitxor Extensions

Individual nodes within a network can support a heterogeneous mix of capabilities. For example, some nodes may want to store data in a database or post events to a message queue. All of these capabilities are optional because none of them affect consensus. Such capabilities are determined by the set of extensions that a node loads as specified in extensions-{process}: extensions. most incorporated **Bitxor** functionality is built using this extension model to validate its extensibility.

An extension is a separate dynamically linked library that exposes a single entry point in the following form3:

```
external"C" PLUGIN_API
empty Log extension (
        bitxor :: extensions :: ProcessBootstrapper & bootstrapper
```

ProcessBootstrapper provides full access **Bitxor** configuration and services that extensions need to initialize. Providing this additional access allows extensions to be more powerful than plugins. Through this class, an extension can register zero or more of the following:

1. Services: A service represents an independent behavior. Services are passed an object that represents the state of the executable and can use it to configure a multitude of things. Among others, a service can add diagnostic counters, define APIs (both diagnostic and non-diagnostic), and add tasks to the task scheduler. You can also create dependent services and bind their lifetime to that of the hosting executable. There are very few limitations on what a service can do, allowing for the potential for significant customization.
2. Subscriptions – An extension can subscribe to any supported blockchain event. Events are generated when changes are detected. Block change, status, unconfirmed transaction, and partial transaction events are supported. Transaction status events are generated when the processing of a transaction is complete. Node events are generated when remote nodes are discovered.

In addition to the above, extensions can configure the node in more complex ways. For example, an extension can register a custom network time provider. In fact, there is a specialized extension that establishes a time provider based on algorithms described insixteen:Weather Synchronization. This is an example of the high levels of customization that this extension model. To understand the full range of extensibility allowed by extensions, consult the project code or developer documentation.4.

2.3 Server

the simplest **Bitxor** the topology consists of a single executable server. Transaction plugins required by the network and **Bitxor** The extensions desired by the node operator are loaded and initialized by the server.

[3]The format of the extensions depends on the target operating system and compiler used, so all host applications and plugins must be compiled with the same version and compiler options.

4 for details https://github.com/BitxorCorp

Bitxor stores all your data in a data directory. The content of the data directory. they are as follows:

1. Block version directories: These directories contain block information in a proprietary format. Binary data, transactions, and associated data for each confirmed block is stored in these directories in files with .dat and .stmt extensions. statements (see7.2:Income) generated by processing each block are also stored here for quick access. An example of a versioned directory is 00000, which contains the first group of blocks. hashes.dat contains a mapping of block heights to lock hashes.
   When deterministic completion is enabled, these directories will also contain versions .proof files that contain proof information for each completion epoch.
2. audit: audit files created by the audit consumer (see9.1.1:Common Consumers) are stored in this directory.
3. Importance: Versioned files that contain information about important accounts at each importance recalculation point. Using files in this directory, the set of important accounts can be reconstituted at any important recalculation point. These files allow deep reversals that require undoing several important calculations. This directory is only created when deterministic completion is enabled.
4. logs – Versioned log files created when file-based logging is enabled are stored in this directory. Active log files associated with running processes are stored directly in the data directory. Each log file is usually prefixed with the name of the source process.
5. spool - Subscription notifications are written to this directory. They are used as a message queue to pass messages from the server to the broker. They are also used by the recovery process to recover data in the event of a bad termination.
6. condition -Bitxor stores its proprietary archive files in this directory. Supplemental.dat and files ending with summary.dat store summary data. Files ending with Cache.dat store full cache data.
7. saidb: When node: enableCacheDatabaseStorage is set, this directory will contain RocksDB files.
8. transfer message: When user: enableDelegatedHarvestersAutoDetection is set, this directory will contain delegated harvest requests fetched for the current node.
9. commit step.dat - Stores the most recent step of the commit process initiated by the server. This is mainly used for recovery purposes.

10. index.dat – This is a counter containing the number of blocks stored on disk.
11. proof.index.dat – This is a counter containing the number of completion proofs stored on disk.
12. Voting status.dat – Stores information about the last completion message sent by a node.

2.3.1 Cache database

The server can run with or without a cache database. When node:enableCacheDatabaseStorage is set, RocksDB is used to store cache data. Verifiable status (see7.3:state hashes) requires a cache database and is expected to be enabled in most network configurations.

A cache database should only be disabled when all of the following are true:

1. A high rate of transactions per second is desired.
2. The trustless verification of the cache state is not important.
3. The servers are configured with a large amount of RAM.

In this mode, all cache entries always reside in memory. At power off, cache data is written to disk in multiple flat files. At startup, this data is read and used to fill memory caches.

When a cache database is enabled, summarized cache data is written to disk in multiple flat files. This summary data is derived from data stored in caches. An example is the list of all high value accounts that have a balance of at least net: minHarvesterBalance. While this list can be generated by (re)inspecting all accounts stored in the account state cache, it is saved and loaded from disk as an optimization.

2.4 Runner

The intermediation process allows for more complex solutions. **Bitxor** Behaviors will be added without sacrificing parallelization. Transaction plugins required by the network and **Bitxor** The extensions desired by the node operator are loaded and initialized by the broker. Although the broker supports all the features of transaction plugins, it only supports a subset of **Bitxor** Features of the extensions. For example, overriding the network time provider in the broker is not supported. Broker extensions are primarily intended to register subscribers and react to events forwarded to those subscribers. Consequently, it is expected that the server and

broker has different extensions loaded. See the project code or developer documentation for more details.

The broker monitors the spool directories for changes and forwards any event notifications to subscribers registered by loaded extensions. Extensions register their subscribers to process these events. For example, a database extension can read these events and use them to update a database to accurately reflect the global state of the blockchain.

Spool directories function as one-way message queues. The server writes messages and the broker reads them. There is no way for the broker to send messages to the server. This decoupling is intentional and was done for performance reasons.

The server generates subscription events in the blockchain synchronization consumer (see9.1.2:bye-National bloc consumers) when you have an exclusive lock on the blockchain data. These operations are offloaded to the broker to avoid slow database operations when the server has an exclusive lock. Server overhead is minimal because most of the data used by the broker is also required to retrieve the data after a server crash.

2.5 Recovery

The recovery process is used to repair the global state of the blockchain after an incorrect server and/or broker termination. Transaction plugins required by the network and **Bitxor** The extensions desired by the node operator are loaded and initialized by the recovery process. When a broker is used, the recovery process must load the same extensions as the broker.

The specific recovery procedure depends on the configuration of the process and the value of the confirmation step.dat file. Generally, if the server exited after the state changes were flushed to disk, those changes will be reapplied. The state of the blockchain will be the same as if the server had applied and committed those changes. Otherwise, if the server exited before the state changes were flushed to disk, the pending changes will be discarded. The state of the blockchain will be the same as if the server had never attempted to process those changes.

Once the recovery process is complete, the state of the blockchain should be indistinguishable from the state of a node that never ungracefully terminated. The spooled directories are repaired and processed. Block and cache data stored on disk is reconciled and updated. Pending status changes are applied, if applicable. Other files that indicate the presence of a bad termination are updated or deleted.

2.6 Common topologies

Although a network can be made up of a large number of heterogeneous topologies, most nodes are likely to fall into one of three categories: Peer, API, or Dual. The same server process is used in all of these topologies. The only difference is in which extensions each loads.

Peer nodes are light nodes. They have enough functionality to add security to the blockchain network, but little beyond that. They can sync with other nodes and collect new blocks.

API nodes are heavier nodes. They can sync with other nodes, but cannot collect new blocks. They support hosting bonded aggregate transactions and collecting co-signatures to complete them. They require a broker process, which is configured to write data to a MongoDB database and propagate changes through public message queues to subscribers. The REST API depends on both of these capabilities and is typically co-located with an API node for performance reasons to minimize latency.

Dual nodes are simply a superset of Peer and API nodes. They support the full capabilities of both node types. Since these nodes support all the capabilities of API nodes, they also require an intermediary.

3 Cryptography

Blockchain is a technology requires the use of some crypto concepts. **Bitxor** uses cryptography based on elliptic curve cryptography (ECC). The choice of the underlying curve is important to ensure safety and speed. **Bitxor** uses the Ed25519 digital signature algorithm. East algorithm uses the following *Twisted Edwards curve*:

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

over the finite field defined by the prime number $2^{255}-19$. The base point for the corresponding group G is called B. The group has q = $2^{252}+27742317777372353535851937790883648493$ items. It was developed by DJ Bernstein et al. and it is one of the safest and fastest digital signature algorithms [Ber+11].

It is important for **Bitxor** The algorithm produces short 64-byte signatures and supports fast signature verification. Neither key generation nor signing is used during block processing, so the speed of these operations is not important.

3.1 Public/private key pair

A private key is a 256-bit random integer k. To derive from it the public key, the public key A, the following steps are followed:

$$\mathrm{H}(k) = (h_0, h_1, \ldots, h_{511}) \tag{1}$$

$$a = 2^{254} + \sum_{3 \le i \le 253} 2^i h_i \tag{2}$$

$$A = aB \tag{3}$$

Since A is a group element, it can be encoded into a 256-bit integer A, which serves as the public key.

3.2 Signature and Verification

Given a message M, private key k, and its associated public key A, the following steps are followed to create a signature:

$$\mathrm{H}(k) = (h_0, h_1, \ldots, h_{511}) \tag{4}$$
$$r = \mathrm{H}(h_{256}, \ldots, h_{511}, M) \text{ where the comma means concatenation} \tag{5}$$
$$R = rB \tag{6}$$
$$S = (r + \mathrm{H}(\underline{R}, \underline{A}, M)a) \bmod q \tag{7}$$

Then (R, S) is the signature of the message M under the private key k. Note that only signatures where S < q and S > 0 are considered valid to avoid the problem of *malleability of the firm*.

To verify the signature (R, S) for the given message M and public key A, the verifier checks S < q and S > 0 and then computes

$$\tilde{R} = SB - \mathrm{H}(R, A, M)A$$

and verify that

$$\tilde{R} = R \tag{8}$$

If S was calculated as shown in (7) after

$$SB = rB + (\mathrm{H}(R, A, M)a)B = R + H(R, A, M)A$$

Y (8) will hold.

3.2.1 batch verification

When many signatures need to be verified, a batch signature verification can speed up the process by about 80%. **Bitxor** uses the algorithm described in [Ber+11]. Given a batch of (Mi, Ai, Ri, Si) where (Ri, Si) is the signature of the message Mi with the public key Ai, let Hi = H(Ri, Ai, Mi). Also, suppose that a corresponding number of independent uniformly distributed 128-bit random integers zi are generated. Now consider the equation:

$$\left(-\sum_i z_i S_i \bmod q\right) B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod q) A_i = 0 \tag{9}$$

Setting $P_i = 8R_i + 8H_iA_i - 8S_iB$, then if (9) holds, it implies

$$\sum_i z_i P_i = 0 \tag{10}$$

All Piso are elements of a cyclic group (remember that q is prime). If any Pin is zero, for example P2, it means that for given integers z0, z1, z3, z4. . ., there is exactly one option for z2 to satisfy (10). The chance of that is 2−128. Therefore, if (9) holds, it is almost certain that Pi= 0 for all i. This implies that the signatures are valid.

Yes (9) is not met, it means that there is at least one invalid signature. Then, **Bitxor** uses single signature verification to identify invalid signatures.

3.3 Verifiable Random Function (VRF)

A verifiable random function (VRF) uses a public/private key pair to generate pseudorandom values. Only the owner of the private key can generate a value such that it cannot be predetermined by an adversary. Anyone with the public key can verify whether or not the value was generated by its associated private key. **Bitxor** uses the ECVRF-EDWARDS25519-SHA512-TAI.

To generate a test[5] given a public key Y corresponding to a private key SK = xB and an input alpha seed[6]:

$$
\begin{aligned}
H &= \text{map\_to\_group\_element}(alpha,\ Y) \\
\gamma &= xH \\
k &= generate\_nonce(H) \\
c &= \text{letfHash}(3,\ 2,\ H,\ \gamma,\ kB,\ kH)[0..15] \\
s &= (k + cx)\ mod\ q \\
proof &= (\gamma,\ c,\ s)
\end{aligned}
$$

The proof produced by the above function can be verified with the following procedure:

$$
\begin{aligned}
H &= \text{map\_to\_group\_element}\ (alpha,\ Y) \\
U &= sB - cY \\
V &= sH - c\gamma \\
verification\ hash &= \text{letfHash}(3,\ 2,\ H,\ \gamma,\ U,\ V)[0..15]
\end{aligned}
$$

When the computed verification hash matches part c of the test, the verification of the random value is successful.

[5] This is usually called proof, not to be confused with verification, because the owner of the private key must prove that they generated the random value with their private key.

[6] The listings provided in this section do not define helper functions. Full descriptions of these functions can be found in [Goal+20].

A test hash, also called a VRF hash result, can be derived from γ of a validated test:

$$Test\_Hash = \text{IetfHash}(3, 3, 8\gamma) \quad \text{(eleven)}$$

3.4 Voting Key List

Final voters must specify the range of final epochs in which a root voting key is eligible to vote. Before announcing a root voting key, voters must create a voting key list containing voting keys fixed by epoch. The construction of the list is roughly based on a simplified Bellare-Miner [BM99] building. Each key in the list is bound to a single epoch and is deleted when the finalization process progresses to a later epoch. This provides some forward secrecy. Even if an attacker obtains a list of voting keys, completed epochs cannot be modified or repudiated.

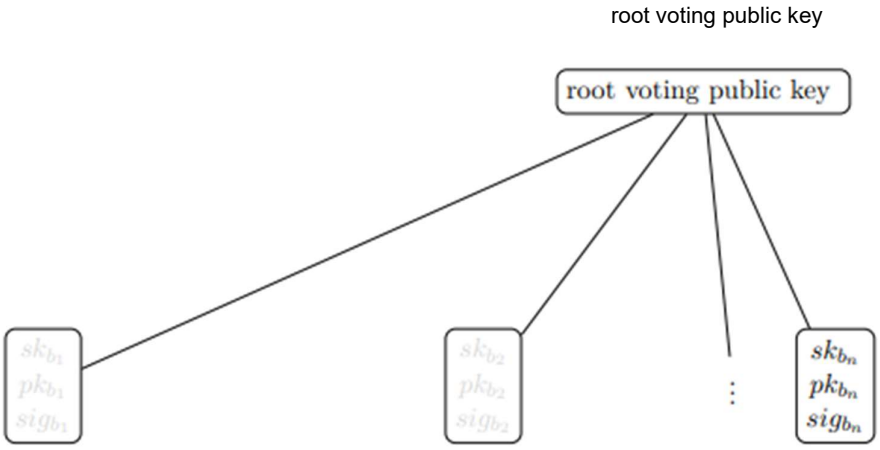


Figure 1: List of voting keys

The list is fully built before announcing a voting key link transaction. First, the root voting key pair is generated. The root voting public key is signed with the signing public key of an account as part of the voting key link transaction.

Next, keys linked to the epoch are generated. For each key pair, the root key pair signs the epoch-bound public key concatenated with its respective epoch. Once all the keys bound to the epoch are generated, the root voting secret key is discarded. In the equations, i refers to the respective epoch.

$$sig_{bi} = Sign_{root\ secret\ key}(pkbi \,||\, \text{IntToBin(i)})$$

Signing a voting message for a given epoch creates a message signature:

$$^{sig}_{message\text{-}i} = \text{Sign}_{skbi}\ (\text{message})$$

3.4.1 Signature

The signature for a vote at a given epoch is made up of two pairs:

- *(root voting public key, sig_{bi})*
- *(pk_{bi} , $sig_{message\text{-}i}$)*

The signature of a key voting list is considered verified when:

- *root voting public key* is registered to participate in the given epoch.
- *Message* the signer's key matches the key linked to the epoch.
- All component signatures are cryptographically verified.

4 Trees

Bitxor uses tree structures to support thin clients without confidence. merkle trees allow a client to cryptographically confirm the existence of data stored in them. Patricia trees allow a customer to confirm cryptographically the existence orlack of data stored in them.

4.1 Tree merkle

A Merkle tree is a tree of hashes that allows efficient proofs of existence. Within **Bitxor**, all basic merkle trees are constrained to be balanced and binary. each leaf The node contains a hash of some data. Each non-leaf node is built by hashing the hashes stored in the child nodes. In the **Bitxor** implementation, when any (non-root) layer contains an odd number of hashes, the last hash is doubled when calculating the main hash.

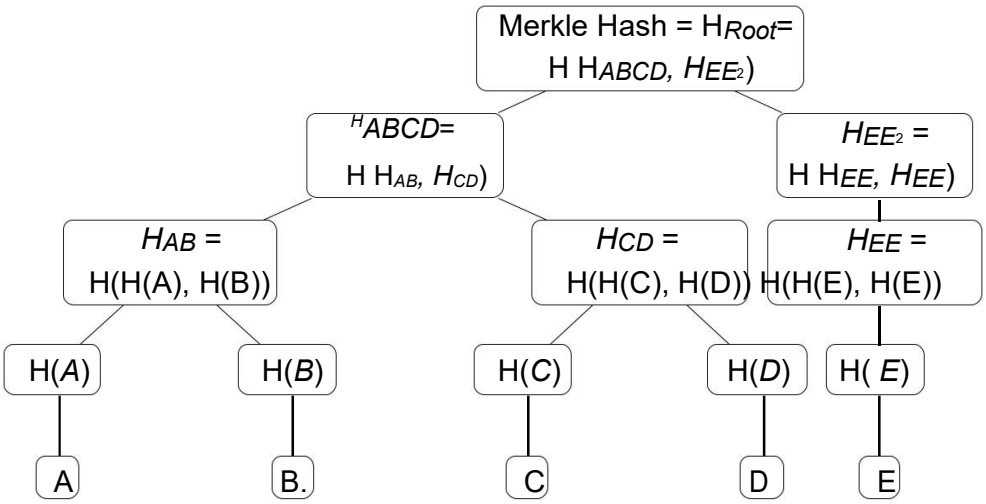


Figure 2: Four-level Merkle tree made up of five data elements

An advantage of using merkle trees is that the existence of a hash in a tree can be proven only with log(N) hashes. This enables proof of existence with relatively low bandwidth requirements.

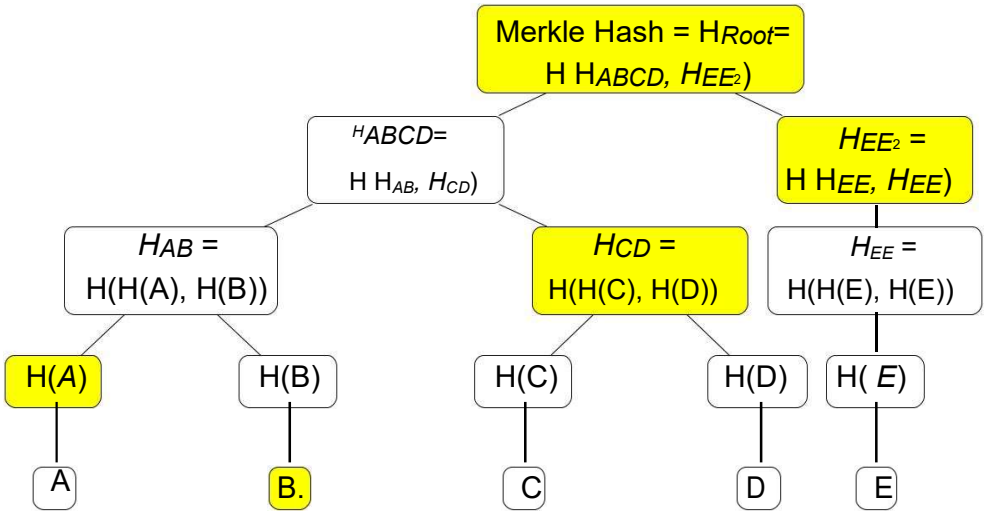


Figure 3: Merkle test required to prove the existence of B in the tree

A Merkle proof of existence requires a single hash of each level of the tree. To prove the existence of B, a client must:

1. Calculate H(*B*).
2. Get H_{Root}; in **Bitxor**, this is stored in the block header.
3. Application H(A), H_{CD}, H_{EE2}.
4. Compute H_{Root}'= H(H(H(H(A), H(B)), HCD), hUS2).
5. Compare H_{Root} and H_{Root}'; if they match, H(*B*) must be stored in the tree.

4.2 Patricia Tree

A Patricia tree[mor68] is a deterministically ordered tree. It is built from key value pairs and supports both existence and non-existence tests that only require log(N) hashes. Non-existence tests are possible because this tree is deterministically ordered by keys. Applying the same data, in any order, will always result in the same tree.

Inserting a new pair of key values into the tree breaks the key into nibbles, and each nibble is logically its own node in the tree. All keys within a single tree are required to have the same length, which allows for slightly optimized algorithms.

For illustration, consider the following key value pairs inTable 2. Some examples will use ASCII keys to elucidate concepts more clearly, while others will use hexadecimal keys to represent more precisely. **Bitxor** implementations.

Figure 4 It represents a complete Patricia tree where each letter is represented by a separate node. Although this tree is logically correct, it is quite large and uses a lot of memory. A typical key is a 32-byte hash value, which means that storing a single value could require up to 64 nodes. To work around this limitation, successive empty branch nodes can be collapsed to a branch node with at least two connections or to a leaf node. This leads to a different but more compact tree, as shown inFigure 5.

key	hex-key	value
do**	646F0000	verb
dog*	646F6700	puppy
doge	646F6765	mascot
hours	686F7273	stallions

Table 2: Sample data from Patricia's tree

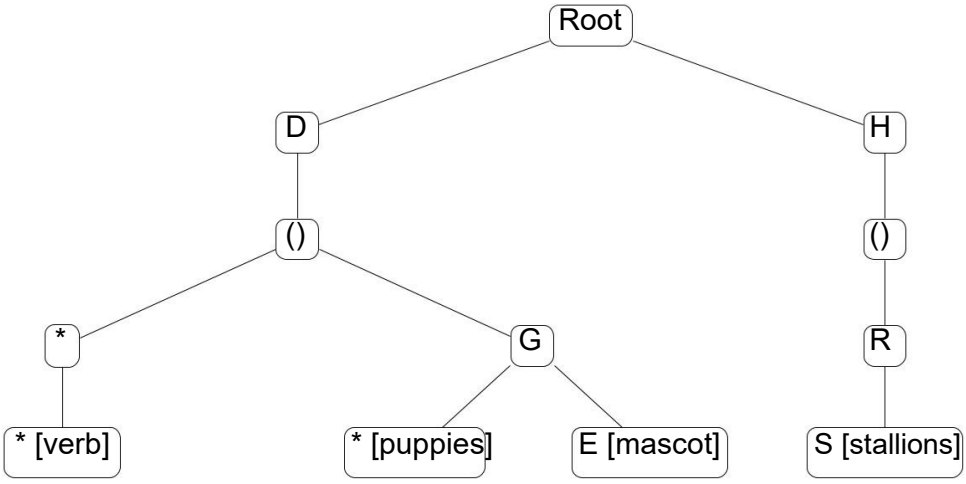


Figure 4: Conceptual Patricia tree (extended) composed of four data elements

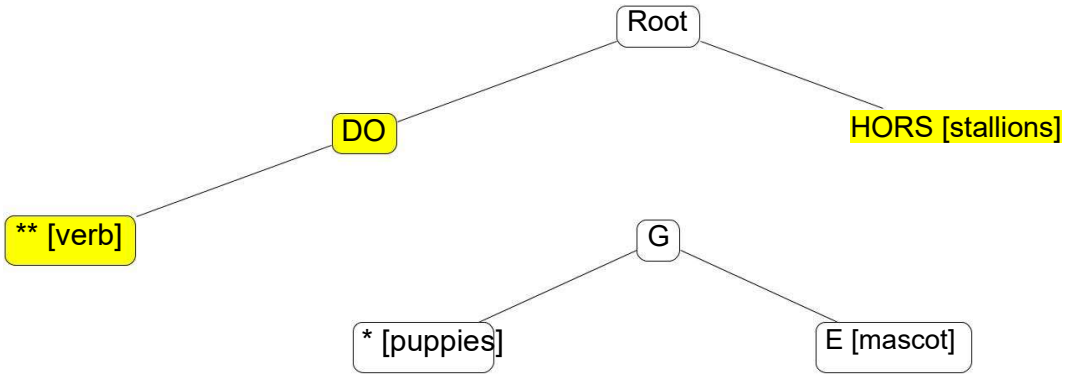


Figure 5: Conceptual (compact) patrician tree composed of four data elements

4.3 TreePatricia Merkle

A Merkle Patricia tree is a combination of Merkle and Patricia trees. The **Bitxor** the implementation focuses on two types of nodes: leaf nodes and branch nodes. Each leaf node contains a hash of some data. Each branch node contains up to sixteen pointers to child nodes.

As in a basic Merkle tree, each Merkle Patricia tree has a root hash, but the hashes stored in the Merkle Patricia tree are a bit more complex.

Each node in the tree has a tree node path. This route consists of a sentinel nibble followed by zero or more route nibbles. If the path represents a leaf node, 0x2 will be set in the sentinel nibble. If the path consists of an odd number of nibbles, the sentinel nibble will be set to 0x1 and the second nibble will contain the first nibble of the path. If the path is made up of an even number, the second nibble will be set to 0x0 and the second byte will contain the first nibble of the path.

A leaf node is made up of the following two elements:

1. TreeNodePath - Encoded tree node path (with set of leaf bits).
2. ValueHash: Hash of the value associated with the key that ends in the leaf.

The hash of a leaf node can be computed by hashing its components:

$$H(Leaf) = H(TreeNodePath, ValueHash)$$

.

A branch node is made up of the following elements:

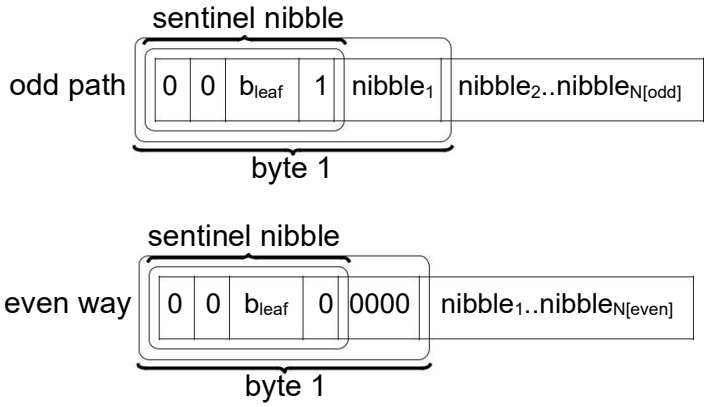


Figure 6: Tree Node Path Encoding

1. Tree Node Path: Encoded tree node path (with the leaf bit not set).
2. *hashlink*$_0$*, . . . ,* *LinkHash*$_{15}$: hashes of children where the index is the next part of the path nibble. When there is no child present in an index, a zero hash should be used instead.

The hash of a branch node can be computed by hashing its components:

$$H(\text{Branch}) = H(\textit{Tree Node Path, LinkHash}_0\textit{,...,LinkHash}_{15})$$

.

Rebuilding the example above with hexadecimal keys produces a tree that illustrates a more precise view of how a **Bitxor** the tree is built. Note that each branch node index makes up a single nibble of the path. This is represented in Figure 7.

4.4 tree tests Patricia Merkle

A Merkle proof of existence requires a single node from each level of the tree. To prove the existence of {key = 646F6765, value = H(mascot)}, a client must:

1. Compute H(*pet*) (*remember, all leaf values are hashes*).
2. Request all nodes on path **646F6765**: *Node*$_6$, *Node*$_{646F}$, *Node*$_{646F67}$.
3. Verify that *Node*$_{646F67}$::*Link*[6] is equal to H(Leaf(*mascot*)).

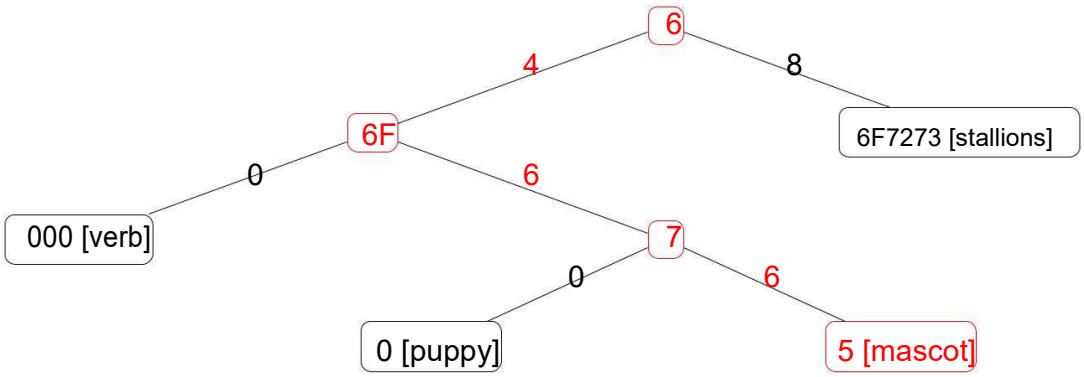


Figure 7: Realistic Patricia tree with branch and leaf nodes and all optimizations. The path to *mascot* [**646F6765**] is highlighted.

4. Compute H($Node_{646F67}$) and verify that $Node_{6467}$::*Link*[6] is equal to H($Node_{646F67}$).
5. Compute H(*Node*6467) and verify that $Node_{6}$:: *Link*[4] is equal to H($Node_{6467}$).
6. Existence is tested if all computed and actual hash values match.

A Merkle test for non-existence requires a single node from each level of the tree. To prove the non-existence of {*key* = **646F6764**, value = H(*mascot*)}, a client must:

1. Compute H(*mascot*) (remember, all leaf values are hashes).
2. Request all nodes on path 646F6764: Node6, Node646F, Node646F 67.
3. Verify that $Node_{646F67}$ :: *Link*[5] is equal to H(*Leaf*(*mascot*)). Since *Link*[5] is not configured, this check will fail. If the value being searched for was in the tree, it should be bound to this node due to the determinism of the tree.

5 Accounts and Addresses

Bitxor uses elliptic curve cryptography to guarantee the confidentiality, authenticity and non-repudiability of all transactions. An account is uniquely identified by an address, which is partially derived from a one-way mutation of its public signing key. Each account is linked to a mutable state that is updated when transactions are made accepted by the network. This state is globally consistent and can contain zero or more public keys.

5.1 Addresses

A decoded address is a 24-byte value made up of the following three parts:

- 2-byte network
- 160-bit hash of an account's signing public key
- 2-byte checksum

The checksum allows fast recognition of misspelled addresses. It is possible to send tokens[7] to any valid address even if the address has not previously participated in any transactions. If no one owns the private key of the account to which the token are sent, the tokens will most likely be lost forever.

An encoded address is a Base32[8] encoding of a decoded address. It is human readable and is used in clients.

Note that the Base32 encoding of binary data has a spread factor of $\frac{5}{8}$, and the size of a decoded address is not divisible by five. Consequently, any Base32 representation of a decoded address will contain additional information. While this is not inherently problematic, when encoding a decoded address, the extra input byte is set to zero by convention for consistency between clients. Also, the last byte of the resulting 40-character string is discarded.

[7] A token is a digital asset defined in **Bitxor**. The Token **Bitxor (BXR)** is the official token of

5.1.1 Address derivation

To convert a public key to an address, the following steps are performed:

1. Run 256-bit SHA3 on the public key.
2. Perform 160-bit RIPEMD 160 of the hash resulting from step 1.
3. Prepend the network version two-bytes to the RIPEMD hash 160.
4. Run 256-bit SHA3 on the result, take the first two bytes as checksum.
5. Concatenate the output from step 3 and the checksum from step 4.
6. Encode the result using Base32.

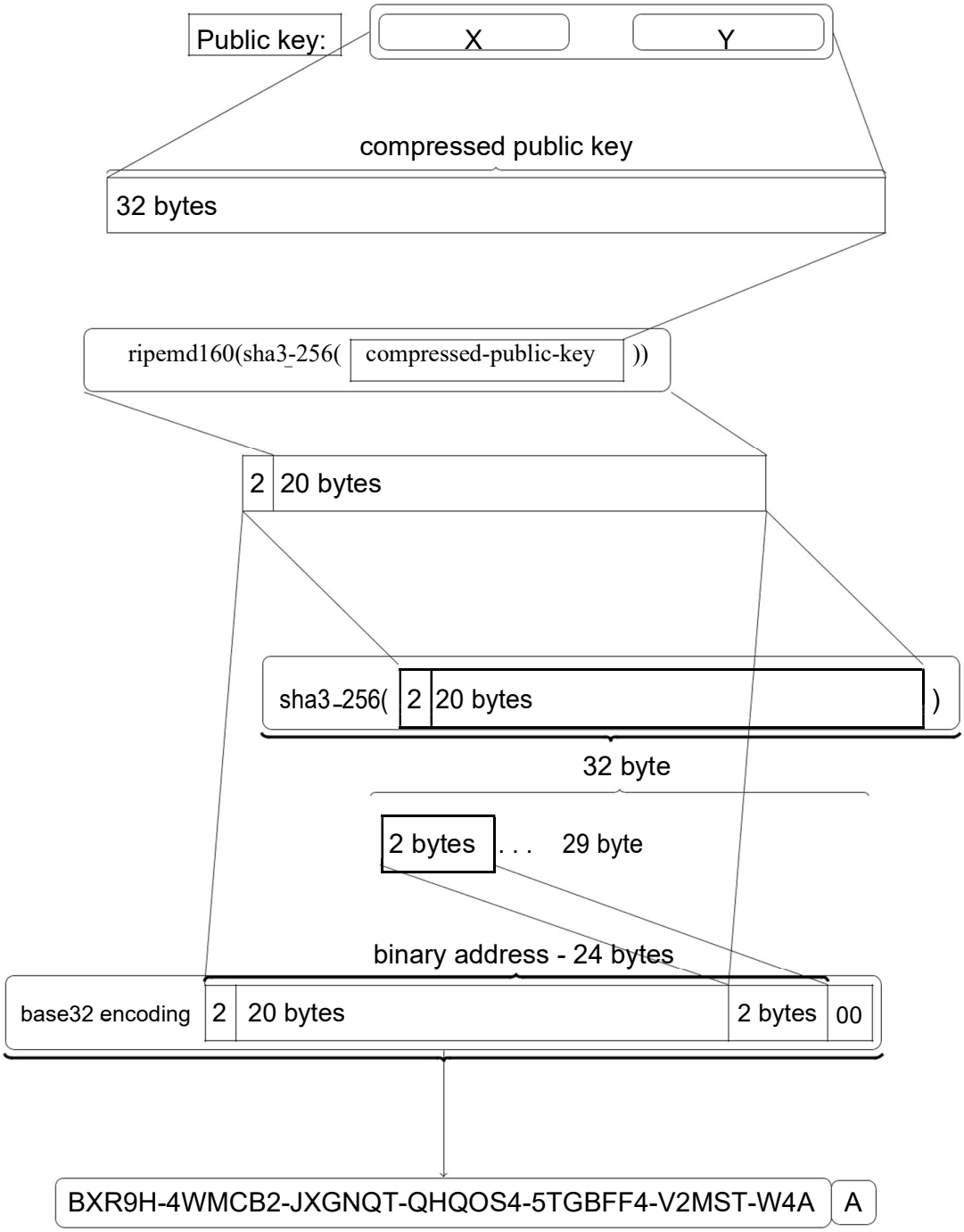


Figure 8: Address Generation

5.1.2 Aliasof direction

An address can have one or more aliases assigned through an address alias transaction9. All transactions that accept addresses support the use of a public key-derived address or address alias. For such transactions, the format of the alias address field is:

- network two-bytes ORnet with value 1
- 8-byte namespace ID that is an alias
- 14 byte zero

5.1.3 Intentional address collision

It is possible for two different public signing keys to produce the same address. If that address contains valuable assets AND has not previously been associated with a public key (for example, when sending a transaction from the account), an attacker could withdraw funds from that account.

For the attack to be successful, the attacker would need to find a private+public key pair such that the SHA3 256 of the public key was at the same time equal to the RIPEMD preimage 160 of the 160-bit hash mentioned above. Since SHA3 256 offers 128 bits of security, it is mathematically unlikely that a single SHA3 256 collision will be encountered. Due to similarities between **Bitxor** addresses and Bitcoin addresses, the probability of causing a **Bitxor** Address collision is roughly the same as causing a Bitcoin address collision.

5.2 public keys

An account is associated with zero or more public keys. Supported key types are:

1. signature:
   Public key ED25519 that is used to sign and verify data. Any account can receive data, but only accounts with this public key can send data. This public key is the only one used as input in the calculation of an account's address.
2. bound:
   This public key links a primary account to a remote collector. For a main account,

this public key specifies the account of the remote collector that can sign blocks on your behalf. For a remote account, this public key specifies the primary account for which you can sign blocks. These links are bidirectional and are always established in pairs.

3. node:
   This public key is set to a primary account. Specifies the public key of the node to which you can delegate the collection. It is important to note that this does not indicate that the remote is actively collecting on the node, only that it has permission to do so. As long as the collection account or node owner is honest, the account is restricted from delegating collection to only one node at a time.
   An honest collector should only send their remote collector private key to a single node at a time. Changing your remote will invalidate all previous remote collection permissions granted to all other nodes (and implies forwarding security of delegated keys). Old private keys from remote collectors will no longer be valid and cannot be used to collect blocks.
   An honest node owner should only remotely harvest with a remote harvester private key that is currently linked to their node public key
4. VRF:
   Public key ED25519 that is used to generate and verify random values. This public key must be set to a primary account for the account to be eligible for collection.
5. vote:
   BLS public key used to sign and verify completion messages. All voting keys are temporary and must be registered with a start and end time. This public key must be set up on a parent account for the account to be eligible to vote. It is only valid when the current epoch is within its registered range. To allow for a seamless key change, an account can have at most network: maxVotingKeysPerAccount voting keys registered at one time.

6 Proceedings

Transactions are instructions that modify the
global chain status. Is it so atomically processed and grouped into blocks.
If any part of a transaction fails processing, the state of the chain
global is reset to the state before the transactionapplication attempt.

There are two fundamental types of transactions: basic transactions and aggregate transactions. Basic transactions represent a single operation and require a single signature. Aggregated transactions are containers for one or more transactions that may require multiple signatures.

Aggregated transactions allow basic transactions to be combined into potentially complex operations and executed atomically. This increases developer flexibility relative to a system that only guarantees atomicity for individual operations while restricting the overall set of allowed operations to a finite set. It does not require the introduction of a Turing complete language and all its inherent drawbacks. Developers don't need to learn any new languages or develop custom contract implementations from scratch. The composition of transactions must be less error-prone and lead to fewer errors than the implementation of computationally complete programs.

6.1 Basic transaction

A basic transaction is made up of cryptographically verifiable and non-verifiable data. All verifiable data is contiguous and signed by the transaction signer. Any non-verifiable data is either ignored (eg padding bytes) or can be computed deterministically from verifiable data. Every basic transaction requires verification of exactly one signature.

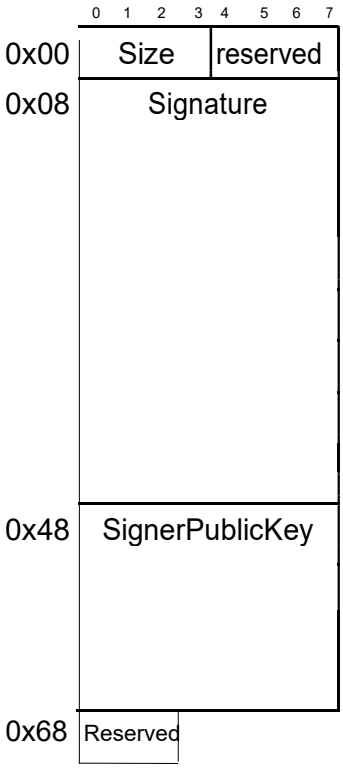
You don't need to check any of the unverifiable header fields. The size is the serialized size of the transaction and can always be derived from the verifiable data of the transaction. The signature is an output to the signature and an input to the verification. SignerPublicKey is an input for both signing and verification. For a transaction T to pass signature verification, both Signature and SignerPublicKey must match verifiable data, which

it has length relative to size.

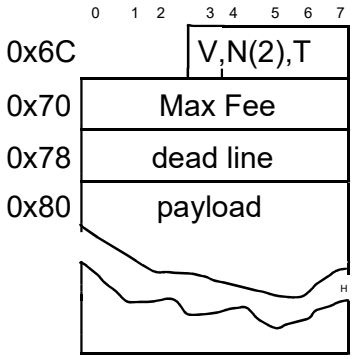
verify(*T*::Signature, *T*::Signer's Public Key, Verifiable Data Buffer(*T*))

Reserved bytes are used to complete transactions so that all integral fields and cryptographic primitives have natural alignment. Since these bytes are meaningless, they can be removed without invalidating any cryptographic guarantees.

Binary layouts for all types of transactions are specified in **Bitxor** open source schema language[10]. Consult published schematics for the most up-to-date specifications.



(a) Header (unsigned)



(b) Verifiable data

Figure 9: Basic transaction binary design

In the figures, (V)ersion, (N)etwork two-bytes and (T)ype are abbreviated due to space concerns.

[10]Schematics can be found at https://github.com/bitxorcorp/.

6.2 Added transaction

The design of an aggregate transaction is more complex than that of a basic transaction, but there are some similarities. An aggregated transaction shares the same unverifiable header as a basic transaction, and this data is processed in the same way. Also, an aggregated transaction has an unverifiable data footer followed by embedded transactions and co-signatures.

An aggregated transaction can always be sent to the network with all required co-signatures. In this case, it is said to be complete and is treated like any other transaction without any special processing.

API nodes may also accept linked aggregate transactions that have incomplete joint natures. The sender must pay a security deposit that is returned if and only if all required joint signatures are collected before the transaction is finalized. Assuming this bonus is paid in advance, an API node will collect the co-signatures associated with this transaction until it has enough signatures or times out.

TransactionsHash is the most important field in an aggregate transaction. It is the root Merkle hash of the embedded transaction hashes stored within the aggregate. To compute this field, a Merkle tree is built by adding each embedded transaction hash in natural order. The resulting root hash is assigned to this field.

You don't need to check any of the unverifiable footer fields. PayloadSize is a calculated size field that must be correct to extract the exact same embedded transactions that were used to calculate TransactionsHash. Reserved bytes, again, are used for padding and have no intrinsic meaning.

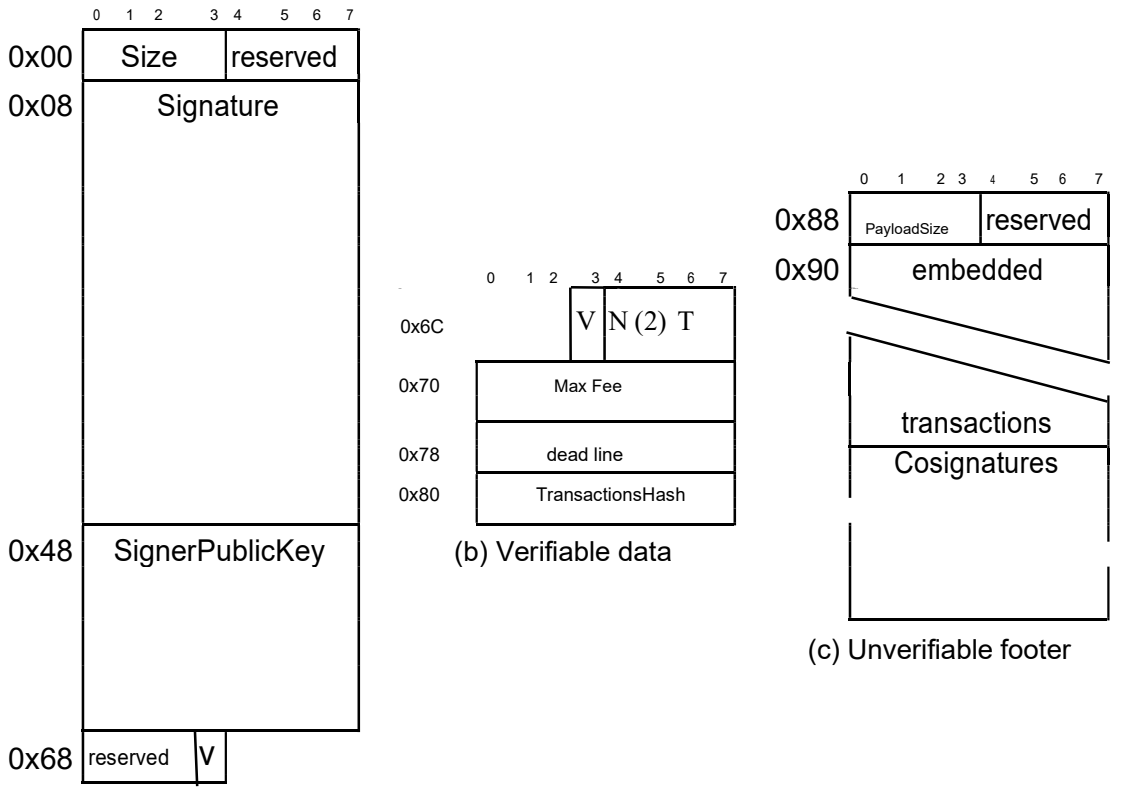


Figure 10 – Added Transaction Header Binary Layout

6.2.1 Integrated transaction

An embedded transaction is a transaction that is contained within an aggregate transaction. Compared to a basic transaction, the header is smaller, but the transaction-specific data is the same. The signature is removed because all the signature information is contained in the joint signatures. MaxFee and Deadline are removed because they are specified by the main aggregate transaction.

Client implementations can use the same code to build the custom parts of a basic or embedded transaction. The only difference is in the creation and application of different headers.

Not all transactions are supported as built-in transactions. For example, an aggregate transaction cannot be embedded within another aggregate transaction.

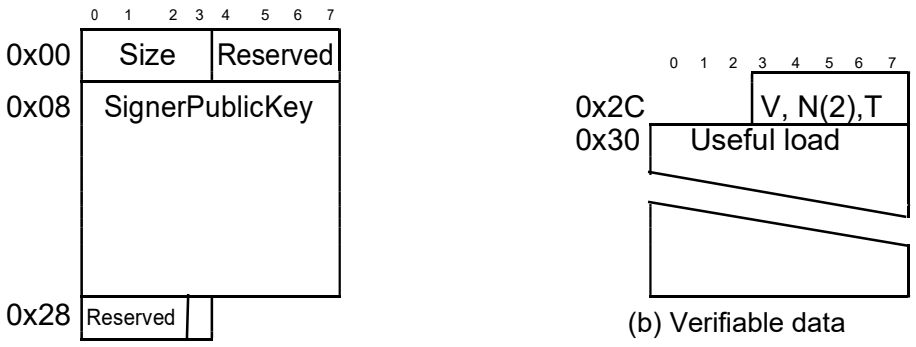


Figure 11: Integrated Transaction Binary Layout

6.2.2 joint signature

A joint signature consists of a version[eleven], a public key and its corresponding signature. Zero or more joint signatures are added to the end of an aggregate transaction. Joint signatures are used to cryptographically verify an aggregate transaction involving multiple parties.

[e] The version is reserved for future extensions. Currently, it is expected to always be zero.

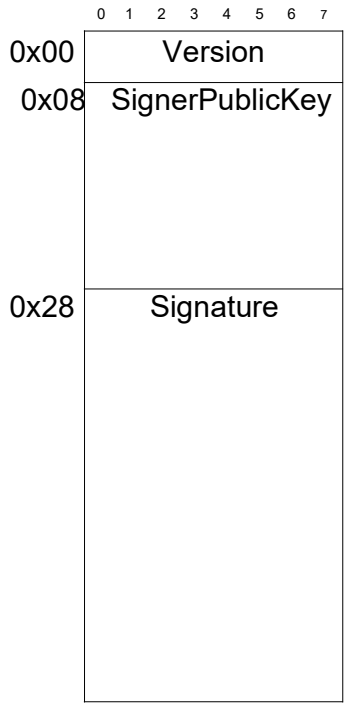


Figure 12: Cosignature Binary Design

For an aggregated transaction A to pass verification, it must pass basic transaction signature verification and have a joint signature for each embedded transaction signer[12].

Like any transaction, an aggregated transaction must pass basic transaction signature verification.

Verify (*A*::Signature, *A*::SignerPublicKey, VerifiableDataBuffer(*A*))

Additionally, all joint signatures must pass signature verification. Note that cosigners sign the data hash of an aggregated transaction, not the data itself.

$\sum_{0 \le i \le N_C}$ Verify (C::Signature, C::SignerPublicKey, H(VerifiableDataBuffer(A)))

Finally, there must be a joint signature that corresponds to and satisfies each signer of the incorporated transaction.

[12] For multisignature accounts, there must be enough co-signatures to satisfy the multisignature account restrictions.

6.2.3 Extended design

The aggregate transaction design described above was correct with one simplification. All embedded transactions are padded to completion on 8-byte boundaries. This ensures that all embedded transactions and joint signatures also start at 8-byte boundaries. Padding bytes are never signed or included in any hash.

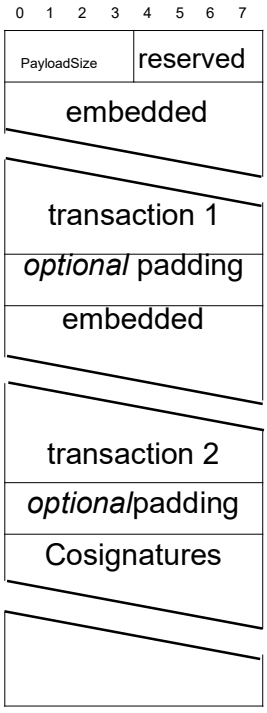


Figure 13: Added transaction footer with padding

6.3 Transaction hashes

Each transaction has two associated hashes: an entity hash and a Merkle component hash. The former uniquely identifies a transaction and is used to prevent multiple commits of the same transaction. The latter is more specific and is used when calculating Transactions Hash (see6.2:Added transaction).

The entity hash of a transaction is calculated as a hash of the following:

1. Transaction Signature – If this field is not included, an adversary could prevent a transaction from being included in the network by loading it with a nearly identical transaction containing a malformed signature.

2. Transaction Signer PublicKey – If this field were not included, an adversary could prevent a transaction from being included in the network by loading it with a nearly identical transaction that contained a malformed signer public key.
3. Network Generation Hash Seed: This field prevents cross-network transaction replay attacks.[13].
4. Verifiable transaction data

All confirmed transactions must have a unique entity hash. The entity hash of an aggregate transaction is independent of their joint signatures. This prevents the same aggregated transaction from being committed multiple times with different sets of valid co-signatures.

The hash of the Merkle component of a normal transaction is identical to the hash of its entity. The hash of the Merkle component of an aggregated transaction is calculated by concatenating the hash of its entity with all the public keys of its joint signatures.[14]. This ensures that the Transactions Hash reflects all the co-signatures that allowed an aggregate transaction to be confirmed.

[13] Also, when signing and verifying transaction data, the Generation Hash Seed network prepends the data so that transaction signatures are only verified on networks with a match.

Hash seed generation.

[14] Co-signature signatures are not included because they can only have a single value given a specific public key and payload.

7 Blocks

Bitxor it is, in essence, a chain of blocks. A blockchain is a ordered collection of blocks. understand the parts of a **Bitxor** block is essential to understand the capabilities of the platform.

The design of a block is similar to the design of an aggregate transaction (see6.2:Agriculture-aggregate transaction). A block shares the same unverifiable headerfifteenWhat an aggregate transaction, and this data is processed in the same way. Also, a block has a footer of unverifiable data followed by transactions. Unlike an aggregate transaction, a block is followed by basic, non-embedded transactions, and each transaction within a block is signed independently of the block's signer.sixteen. This allows any transaction that satisfies all conditions to be included in any block.

You don't need to check any of the unverifiable footer fields. Reserved bytes are used for padding and have no intrinsic meaning.

7.1 Block fields

The height is the sequence number of the block. The first block, called the genesis block, has a height of one. Each successive block increases the height of the previous block by one.

The timestamp is the number of milliseconds that have passed since the genesis block. Each successive block must have a higher timestamp than the previous blocks because the block time is strictly increasing. Each network tries to keep the average time between blocks close to the target block time.

Difficulty determines how difficult it is to harvest a new block, based on previous blocks. The difficulty is described in detail in8.1:block difficulty.

GenerationHashProof is the VRF proof generated with the block collector's VRF private key. It consists of a 32-byte γ, a 16-byte check hash (c), and a 32-byte scalar(s) (see3.3:Verifiable Random Function (VRF)). This is used to ensure that future block harvesters are unpredictable (see8.3:Block generation).

[15] To emphasize, this is not referring to the verifiable block header. Refers to fields such as Signature that precede the verifiable block header.

[16] In an aggregated transaction, the account creating the aggregated transaction must sign the transaction data for it to be valid. In a block, the block signer does not need to sign the data of any transaction contained in it.

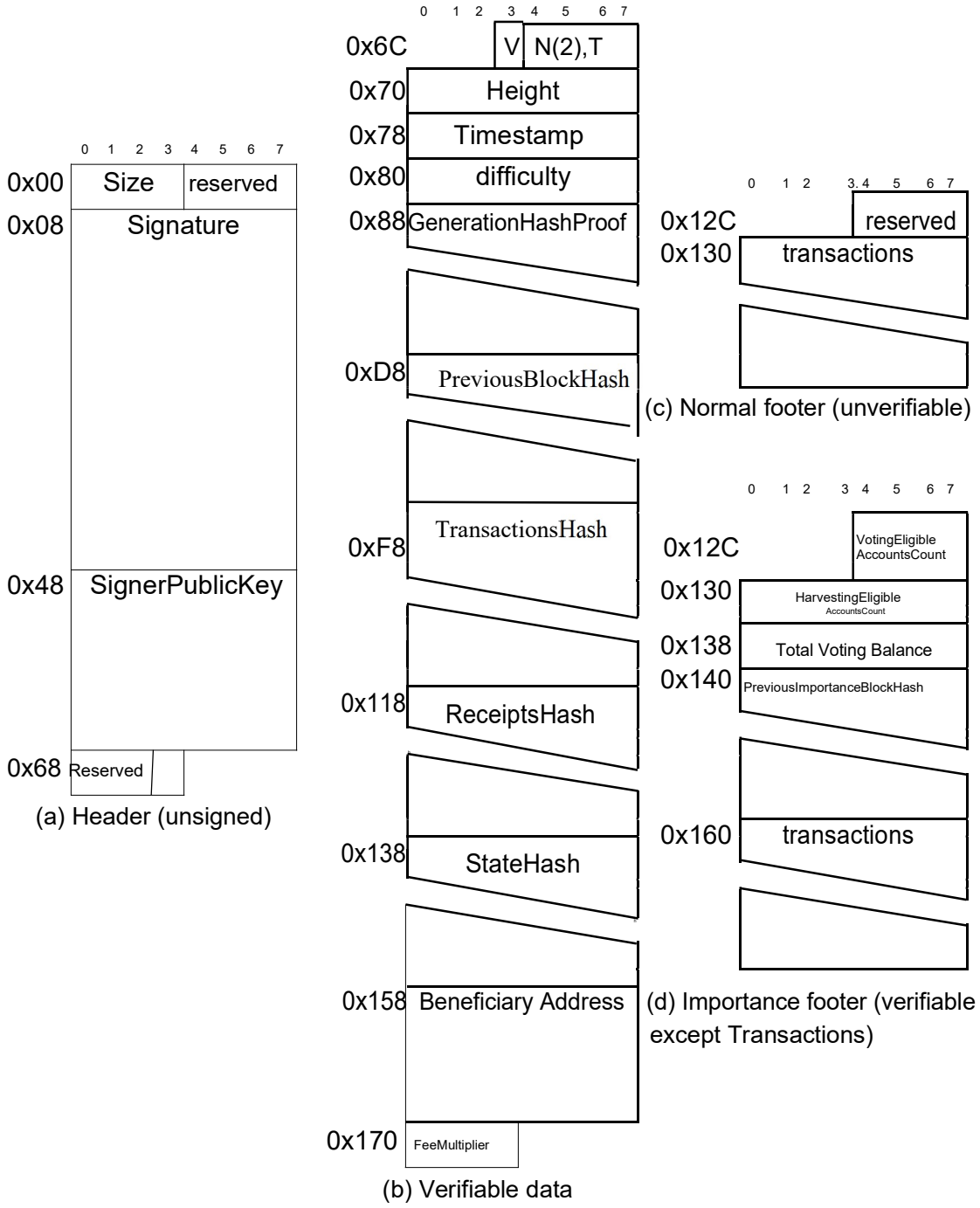


Figure 14: Block Header Binary Layout

Previous Block Hash is the hash of the previous block. This is used to ensure that all blocks within a blockchain are deterministically linked and ordered.

Transactions Hash is the Merkle root hash of the transaction hashes stored within the block[17]. To compute this field, a Merkle tree is built by adding each transaction hash in natural order. The resulting root hash is assigned to this field.

Receipts Hash is the Merkle root hash of the receipt hashes produced while processing the block. When a network is configured without network: enable Verifiable Receipts, this field must be set to zero in all blocks (see7.2:Income).

StateHash is the hash of the global state of the blockchain after processing the block. When a network is configured without network: enable Verifiable State, this field must be set to zero in all blocks. Otherwise, it is calculated as described in7.3:state hashes.

Beneficiary Address is the account to which the beneficiary share of the block will be assigned when the network has a non-zero network: mining profit percentage. This field can be set for any account, even one that is not yet known to the network. This field is set by the node owner when a new block is mined. If this account is set up as owned by the node owner, the node owner will share in the fees paid on all harvested blocks on their node. This, in turn, incentivizes the node owner to run a strong node with many delegated collectors.

Fee Multiplier is a multiplier used to calculate the effective fee for each transaction contained within a block. The node: min Fee Multiplier is set by the owner of the node and can be used to achieve various goals, including profit maximization or confirmed transactions. Assuming that a block B contains a transaction T, the effective transaction fee can be calculated as:

$$\text{Effective fee } (T) = B::\text{Fee Multiplier size of}(T)$$

If the effective fee is greater than the transaction Max Fee, the transaction signer keeps the difference. Only the actual fee is deducted from the transaction signer and credited to the harvester. More information on fee multipliers can be found at8.3:Block Generation.

7.1.1 Importance Block Fields

Each block in which an importance calculation is performed contains an expanded footer with additional verifiable information[18]. This information reflects the global state of the blockchain after processing the block.

[17] This field has the same purpose as the field with the same name in an aggregate transaction.

[18] *network: voting Set Grouping*expected to be a multiple of network: importance Grouping.

Voting Eligible Accounts Count is the number of accounts eligible to vote at the end time that covers the following importance group. This is an exact number and only includes accounts that meet all voting requirements. For example, accounts that do not have a voting key on file are excluded.

Harvesting Eligible Accounts Count is the number of accounts with a balance of at least network: min Harvester Balance. This is an estimate and a maximum. The actual number of accounts eligible for collection may be less because accounts not eligible for collection are not excluded. For example, accounts that do not have a VRF key on file are not excluded.

Total Voting Balance is the sum of the balance of all voting-eligible accounts at the time of completion covering the following importance group. This value allows trustless verification of completion tests. It can be used as the authoritative denominator when calculating the cumulative voter turnout percentage of a completion test.

Previous Importance BlockHash is the hash of the previous importance block. This is a longer lasting guarantee, relative to the Previous Block Hash, that all blocks within a blockchain are deterministically linked and ordered. It can be used to enable a trustless fast sync protocol, which only needs to download important block headers instead of all block headers.

7.2 Income

Zero or more receipts are generated during the execution of a block. Receipts are primarily used to communicate status changes caused by secondary effects to clients. In this way, they allow simpler clients to remain aware of complex state changes.

For example, the expiration of a namespace is triggered by the number of blocks that have been committed since the namespace was created. While the triggering event is on the blockchain, there is no indication of this state change in the block in which the expiration occurs. Without receipts, a client would need to keep track of all expiration namespaces and heights. With receipts, a customer simply needs to monitor receipts that are due.

Another example is related to mining rewards. Receipts are produced indicating the main, non-delegated account being credited and the divisions of beneficiaries. They also communicate the amount of currency created by inflation.

Receipts are grouped into three different types of statements and collated by receipt sources. The three types of statements are transactions, address resolution, and token resolution.

7.2.1 Receipt origin

Each part of a block that is processed is assigned a two-part block scope identifier. The source (0, 0) is always used to identify block-triggered events regardless of the number of transactions in a block.

Font	main id	Secondary ID
Block	0	0
Transaction	1-based index inside the block	0
integrated trans action	1-based index of added content door inside the block	1-based index into the aggregate

Table 3: Receipt source values

7.2.2 Transaction statement

Transaction statements are used to group receipts that have a shared receipt source. Each statement is made up of a receipt source and one or more receipts. Consequently, each receipt source that generates a receipt will have exactly one corresponding transaction statement.

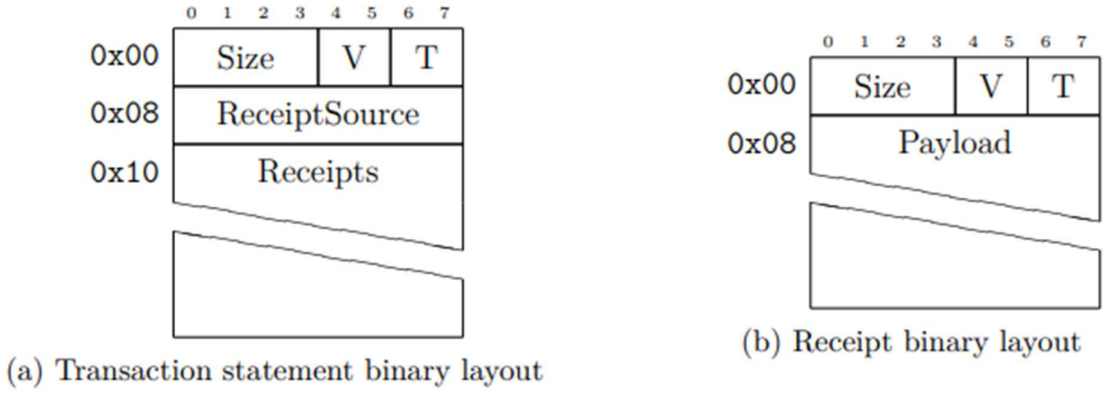


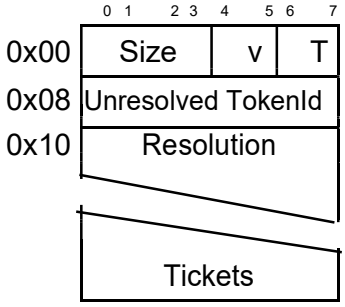
Figure 15: Transaction Statement Layout

Data in transaction statements is not padded because it is only written during processing and is never read, so padding does not provide any benefit to server performance. A transaction statement hash is constructed by concatenating and hashing all the data in the statement, except the Size fields, which are derived from other data.

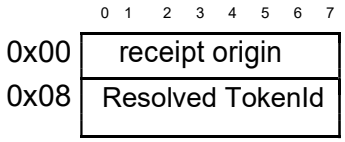
7.2.3 Resolution statements

Resolution declarations are used exclusively to indicate alias resolutions. They allow a client to always resolve an unresolved value even when it changes within a block. Theoretically, two unresolved aliases within the same block could resolve to different values if there is an alias switch between their uses. Each statement is made up of an unresolved value and one or more resolutions to support this.

There are two types of resolution statements - address and token - corresponding to the two types of aliases. DespiteFigure 16 illustrates the layout of a token resolution statement, the layout of an address resolution statement is nearly identical. The only difference is that the resolved and unresolved values are 25-byte addresses instead of 8-byte token IDs.



(a) Binary Token Resolution Statement Layout



(b) Resolving Input Binary Layout

Figure 16: Token resolution statement layout

The resolve statement data is not padded because it is only written during processing and is never read, so padding does not provide any benefit to server performance. A resolve statement hash is constructed by concatenating and hashing all the data in the statement, except for the Size fields, which are derived from other data.

It is important to note that a resolve statement only occurs when a resolve occurs. If an alias is registered or changed in a block, but is not used in that block, a resolve statement will not be generated. However, each block that contains that alias and requires it to be resolved will produce a resolve statement.

7.2.4 Receipt hashes

To compute the receive hash of a block, all statements generated during block processing are first collected. A Merkle tree is then created by adding all the hash values of the statements in the following order:

1. Hashes of transaction statements sorted by receipt source.
2. Hashes of address resolution statements sorted by unresolved address.
3. Hashes of token resolution statements ordered by unresolved token ID.

When a network is configured with network: enable Verifiable Receipts , the root hash of this merkle tree is set as the Receipt Hash of the block. A client can perform a Merkle test to prove that a particular statement occurred during the processing of a specific block.

7.3 State hashes

Bitxor stores the global state of the blockchain in various typed state repositories. For For example, account status is stored in one repository and multiple signature status is stored in another. Each repository is a simple key value store. The specific repositories present on a network are determined by the transaction plugins enabled by that network.

When a network is configured with network: enable Verifiable State, a Patricia tree is created for each repository. This produces a single hash that deterministically fingerprints each repository. Consequently, assuming N repositories, N hashes deterministically identify the global state of the blockchain.

It is possible to store all N hashes directly in each block header, but this is not desirable. Each block header should be as small as possible because all clients, at a minimum, need to synchronize all headers to verify that a chain is rooted in the genesis block. Also, adding or removing functionality could change the number of repositories (N) and the format of the block header.

Instead, all root hashes are concatenated.[19] and hash to calculate the State Hash, which is a single hash that deterministically determines the global state of the blockchain.

$$
\begin{aligned}
RepositoryHashes &= 0 \\
RepositoryHashes &= \sum_{0 \le i \le N} \text{concat}(RepositoryHashes, RepositoryHash_i) \\
StateHash &= \text{H}(RepositoryHashes)
\end{aligned}
$$

[19] The concatenation order is fixed and is determined by the repository id.

7.4 Extended design

The block design described above was correct with a simplificationtwenty. All transactions are padded to completion on 8-byte boundaries. This ensures that all transactions also start on 8-byte boundaries. Padding bytes are never signed or included in any hash.

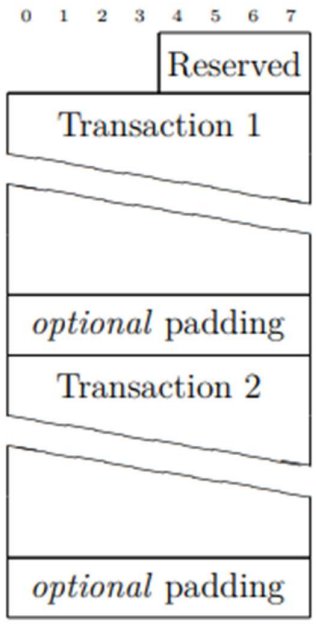


Figure 17: Block transaction footer with padding

7.5 Block hashes

Each block has an associated hash: an entity hash. This hash uniquely identifies a block and is used to prevent the same block from being processed multiple times.

The entity hash of a block is calculated as a hash of the following:

1. block signature
2. PublicKey block signer
3. Block (header) of verifiable data

The entries in an entity hash for a block and a transaction are similar. The only difference is that the Generation Hash Seed network is not an input in the entity's hash calculation.

[20] This is also consistent with the extended design of an aggregate transaction.

for a block Including this field would serve no purpose for a block. Any block that contains at least one transaction cannot be replayed on a different network because the hashes of the transaction entity are different between networks. Consequently, the Transactions Hash of the block would also be different.

8 Blockchain

Bitxor focuses on a public ledger called a blockchain that links blocks together. The entire history of transactions is kept on the blockchain. All blocks and transactions within blocks are deterministically and cryptographically ordered. The maximum number of transactions per block per network can be configured.

8.1 Block difficulty

The genesis block has a default starting difficulty of 10^{14}. All difficulties are capped between a minimum of 10^{13} and a maximum of 10^{15}.

The difficulty of a new block is derived from the difficulties and timestamps of the most recently confirmed blocks. The number of blocks considered is configurable per network.

If there are less than network: max Difficulty Blocks available, only the available ones are taken into account. Otherwise, the difficulty is calculated from the last *network:maxDifficultyBlocks* blocks in the following way:

$$d = \frac{1}{n}\sum_{i=1}^{n}(\text{difficulty of } block_i) \qquad \text{(average difficulty)}$$

$$t = \frac{1}{n}\sum_{i=1}^{n}(\text{time to create } block_i) \qquad \text{(average creation time)}$$

$$difficulty = d\,\frac{\text{blockGenerationTargetTime}}{t} \qquad \text{(new difficulty)}$$

This algorithm produces blocks with an average time close to the desired *network: blockGenerationTargetTime* Network Configuration.

If the new difficulty is more than 5% higher or lower than the difficulty of the last block, the change is capped at 5%. The maximum exchange rate of 5% per block makes it difficult for an attacker with significantly less than 50% importance to secretly create a better chain. Since the difficulty roughly correlates to the total amount of importance currently being harvested, the attacker's secret chain will necessarily have a much lower difficulty. Limiting the maximum decrease in difficulty per block means the

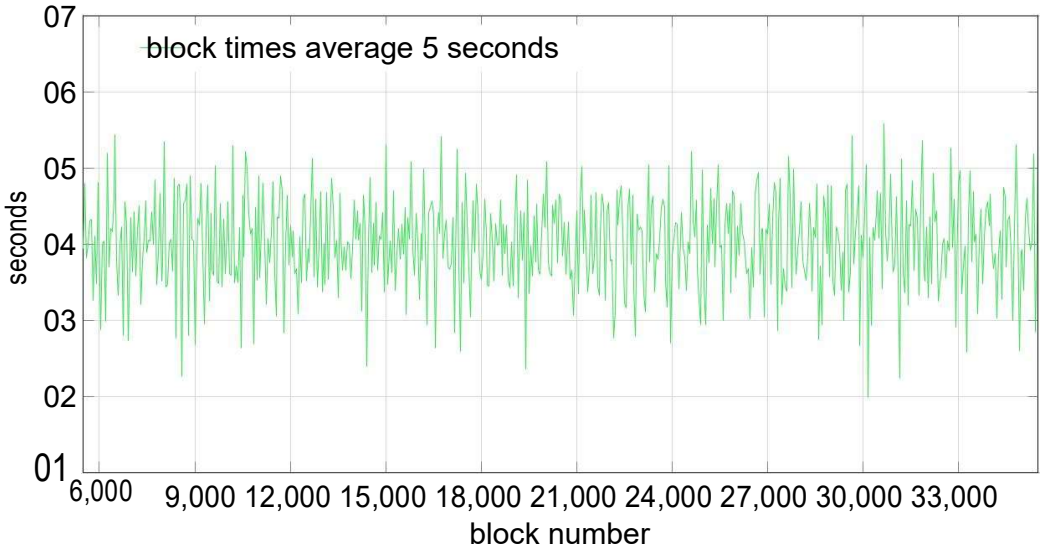


Figure 18: Development network average block times, with target block time = 15 s

the attacker's chain will quickly fall behind the main chain. By the time the difficulty adjusts to something more proportional to the importance of the attacker, the main chain will be way ahead. The block times will be considerably higher than *network:blockGenerationTargetTime* at the beginning of the attacker's secret chain.

8.2 Block Score

A block's score is derived from its difficulty and the time (in seconds) that has elapsed since the last block:

$$score = difficulty - time\ elapsed\ since\ last\ block \quad \text{(block score)}$$

8.3 Block generation

The process of creating new blocks is called *harvesting*. The collection account gets most of the fees from the transactions it includes in a block. This gives the collector an incentive to create a valid block and add as many transactions to it as possible.

The fees paid in each block are divided into one of three buckets. When *network:harvest-BeneficiaryPercentage* is non-zero, that percentage of fees will be credited to the harvester's designated beneficiary. When *network:harvestNetworkPercentage* is nonzero, that percentage

age of the fees will be credited to the specified network of the network: address of the recipient of network fees for mining. This account is intended to allow networks to be self-sufficient and can be used for incentive programs such as rewarding voting nodes. All other fees will be credited to the harvester's account.

An account is eligible to harvest if all of the following are true:

1. The importance score at the last importance recalculation height is non-zero.
2. Balance none other than a network defined by the *network:minHarvesterBalance..*
3. Balance no greater than a network defined by *network:maxHarvesterBalance*[twenty-one].
4. The VRF public key is registered for the account.

The owner of an account can delegate its importance to another account[22] to avoid exposing a funded private key.

The actual reward a harvester receives can be customized based on network settings. If the inflation setting has non-zero values, each harvested block may contain an additional inflation block reward. This makes the reward more profitable. If harvest fee sharing is enabled (*network:harvestBeneficiaryPercentage*), the harvester will lose a portion of the fees for the node that hosts their harvest key. This makes running network nodes more profitable but harvesting less profitable.

Each block must specify a fee multiplier that determines the effective fee to be paid by all transactions included in that block. Usually the node owner sets the *node: minFeeMultiplier* that applies to all blocks collected by the node. Only transactions that meet the following will be allowed to enter that node's cache of unconfirmed transactions and will be eligible for inclusion in blocks collected by that node:

$$\text{transaction max fee} \geqslant \text{minFeeMultiplier} \cdot \text{transaction size (bytes)} \quad (12)$$

Rejected transactions can still be included in blocks collected by other nodes with lower requirements. The specific algorithm used to select transactions for inclusion in harvested blocks is configured using the *node:transactionSelectionStrategy* setting.**Bitxor** offers three built-in selection strategies[23]:

[21] This feature is primarily intended to prevent basic funds and trading accounts from being harvested.

[22] In all cases, all available transactions must already meet the node min Fee Multiplier requirement.

1. oldest:
   This is the strategy that requires the least resources and is recommended for high TPS networks. Transactions are added to a new block in the order they are received. This ensures that the oldest transactions are selected first and tries to minimize transaction timeout. As a consequence, this strategy rarely maximizes profit for the harvester.
2. maximize-rate:
   Transactions are selected in such a way as to maximize the cumulative fee for all transactions in a block. A profit optimization node will choose this strategy. Maximizing the total block fee does not necessarily mean that the number of included transactions is also maximized. In fact, in many cases, the collector will only include a subset of the transactions that are available.
3. minimum fee:
   This strategy first selects the transactions with the lowest maximum fee multipliers. Altruistic nodes will choose this strategy along with a very low node: min Fee Multiplier. If this setting is zero, then the collector will include transactions with zero fees first. This allows users to send transactions that are included in the blockchain for free! In practice, only a few nodes are likely to support this. Even with a subset of nodes running, zero-fee transactions will still have the lowest probability of being included in a block because they will always be supported by the fewest nodes in the network.

Transactions can initiate transfers of static and dynamic amounts. Static quantities are fixed and independent of external factors. For example, the amount specified in a transfer transaction24 it is static. The exact amount specified is always transferred from the sender to the recipient. In contrast, dynamic amounts are variable relative to the average transaction cost. They are typically reserved for fees paid to acquire unique network artifacts such as namespaces or token. For such devices, a flat fee is not desirable because it would not respond to the market. Similarly, the exclusive use of FeeMultiplier of a single block is problematic because harvesters could cheat and receive artifacts for free by including records in self-harvested blocks with zero fees. Instead, a dynamic fee multiplier is used. This multiplier is calculated as the median of the FeeMultiplier values in the last *network:maxDifficultyBlocks* blocks. *network:defaultDynamicFeeMultiplieris* used when there are not enough values and as a replacement for zero values. The last adjustment ensures that the effective quantities are always different from zero. To arrive at the effective amount, the base amount is multiplied by the dynamic fee multiplier.

effective amount= base amount dynamic fee multiplier

8.4 Block generation hash

The generation hash of a block is derived from the previous block generation hash and the VRF proof included in the block:

gh(1) = generationHash (generation hash)
gh(N) = verify vrf_proof(proof(block(N)), gh(N − 1), *VRF public key of account*)

Making the generation hash dependent on a VRF makes it effectively random and unpredictable. This is important because it makes the identity of the next harvester unknown even to an adversary with perfect information. Until a block is pushed onto the network, its generation hash is unknown to all but its collector. Consequently, until the best block is sent to the network, since each generation hash depends on the previous generation hash, the input in the calculation of the generation hash for the next block is unknown until then.

A VRF public key must be registered on the blockchain before it can be used to collect a block. This preregistration requirement prevents an adversary from choosing an arbitrary VRF public key that can maximize its impact on the next block. As a result, the opponent cannot produce many blocks in an attempt to maximize his hit.

8.5 Block hit and target

To check whether an account can create a new block at a specific network time, the following values are compared:

- hit: defines per-block value that needs to be hit.
- target: defines per-harvester power that increases as time since last harvested block increases

An account can create a new block each time it hits <target. Since the goal is proportional to the elapsed time, a new block will be created after a certain amount of time, even if all accounts are unlucky and generate a very high hit.

In the case of delegated collection, the importance of the original account is used instead of the importance of the delegated account.

The target is calculated as follows[25]:

$$
\begin{aligned}
multiplier &= 2^{64} \\
t &= time\ in\ seconds\ since\ last\ block \\
b &= 8999999998 \cdot (account\ importance) \\
i &= total\ chain\ importance \\
d &= new\ block\ difficulty \\
target &= \frac{multiplier \cdot t \cdot b}{i \cdot d}
\end{aligned}
$$

*block time smoothing*s can be enabled, which results in more stable block times. If enabled, the above multiplier is calculated as follows[26]:

$$
\begin{aligned}
factor &= \text{blockTimeSmoothingFactor}/1000.0 \\
tt &= \text{blockGenerationTargetTime} \\
power &= factor \cdot \frac{time\ in\ seconds\ since\ last\ block - tt}{tt} \\
smoothing &= \min\left(e^{power}, 100.0\right) \\
multiplier &= \text{integer}\left(2^{54} \cdot smoothing\right) \cdot 2^{10}
\end{aligned}
$$

Hit is 64-bit approximation of $2^{54} \left|\ln\left(\frac{gh}{2^{256}}\right)\right|$ where gh is a new generation hash

First, let's rewrite the above value using log with base 2:

$$
hit = \frac{2^{54}}{\log_2(\mathrm{e})} \cdot \left|\log_2\left(\frac{gh}{2^{256}}\right)\right|
$$

Note that $\frac{gh}{2^{256}}$ always < 1, therefore log will always return a negative value. now $\log_2\left(\frac{gh}{2^{256}}\right)$ can be rewritten as $\log_2(gh) - \text{record}_2(2^{256})$.

[25] The implementation uses 256-bit integers instead of floating point arithmetic to avoid any problems due to rounding.

[26] The implementation uses fixed-point rather than floating-point arithmetic to avoid any problems due to rounding. Specifically, 128-bit fixed-point numbers are used where the high 112 bits represent the integer part and the low 16 bits represent the decimal part. log2(e) approaches 14426950408/10000000000. If the calculated power is too negative, smoothing will be set to zero.

[27] The implementation uses 128-bit integers instead of floating point arithmetic to avoid any problems due to rounding. log2(e) is approximated as 14426950408889634 / 10000000000000000.

Eliminating the absolute value and rewriting yields:

$$scale = \frac{1}{\log_2(\mathrm{e})}$$

$$hit = scale \cdot 2^{54}(\log_2\left(2^{256}\right) - \log_2(gh))$$

This can be further simplified to:

$$hit = \text{scale} \cdot (2^{54}\ 256 - 2^{54} \cdot \text{Log}_2\ (gh))$$

The implementation approximates the logarithm using only the first 32 nonzero bits of the next generation hash. There is also additional handling for extreme cases.

Also note that *hit* has an exponential distribution. Therefore, the probability of creating a new block does not change if the importance is divided among many accounts.

8.6 Automatic detection of delegated harvester

When *user:enableDelegatedHarvestersAutoDetection* is set, the server allows other accounts to register as delegate harvesters through special transfer messages. The server inspects all transfer messages sent to the account that match its node certificate public key and sends those that match to a file queue. Periodically, a scheduled task inspects all queued messages. Any message that contains unexpected or malformed content is ignored and discarded. Valid messages are decrypted and processed.

A main account is eligible[28] to delegate mining to a node when all of the following links are configured:

- The primary account has signed the proxy collection request.
- The linked remote public key matches the private key of the encrypted linked remote harvester.
- The VRF public key matches the encrypted VRF private key.
- The public key of the node matches the public key of the node certificate (see12.2:Connection Handshake).

Messages are partially encrypted to prevent an adversary from obtaining VRF private keys and remote plaintext harvesters. AES256 GCM is the encryption (symmetric)

[28] Even if all of these conditions are met, the node owner can still decide not to allow the eligible account to delegate collection.

scheme used. The encryption key is derived from the server's node certificate key pair and a random ephemeral key pair generated by the client.

Each message is expected to have the following contents:

Name	Size	Encrypted?	Description
0xE201735761802AFE	8 byte	Nope	Magic bytes indicating that the message is a mining application
ephemeral public key	32 byte	Nope	Public key used to derive the symmetric encryption key
AES-GCM tag	16 bytes	Nope	MAC tag on encrypted data
AES GCM IV	12 byte	Nope	Initialization vector for AES GCM algorithm swing
private key signing	32 byte	Yes	Remote harvester signing private key linked to main account
VRF private key	32 byte	Yes	VRF private key linked to the account

Table 4: Reward request message format

If possible, the server will use the delegated collector's advertised private keys to collect blocks. A server can have at most harvesters: max Unlocked Accounts harvesters. Upon reaching that limit, the evaluation of any new delegated collector is based on the delegate priority policy setting. When the policy is set to Age, previously advertised accounts are preferred. As a result, a new delegated collector cannot replace any existing delegated collector. When the policy is set to Importance, accounts with the highest importance are preferred. As a result, a new delegated collector can replace an existing delegated collector with less importance.

Successful announcements are stored in the harvesters.dat file. Accepted delegate collectors persist across server restarts. The server does not provide any explicit confirmation that it is or is not currently collecting with a specific delegated collector. The blockchain only stores a record of all the links related to the delegate collection, but not the actual activity.

8.7 Blockchain synchronization

A score can be assigned to any blockchain by adding the scores of the component blocks:

(blockchain score)

$$score = \sum_{block \in blocks} block\ score$$

Blockchain synchronization is crucial to maintaining distributed consensus. Periodically, a local node will query a remote node about its chain. The remote node is selected from a pool of partners based on several factors, including reputation (see13:Reputation).

If the remote node promises a chain with a higher score, the local node tries to find the last common block by inspecting the hashes provided by the remote node. When deterministic completion is enabled, a binary search is performed to find the last common block. The search space is all the hashes between the last completed block and the current local height. If successful, the remote node will supply as many blocks as the configuration allows.

If the provided string is valid, the local node will replace its own string with the remote string. If the provided string is invalid, the local node will reject the string and consider the synchronization attempt with the remote node to have failed.

Figure 19illustrates the process in more detail.

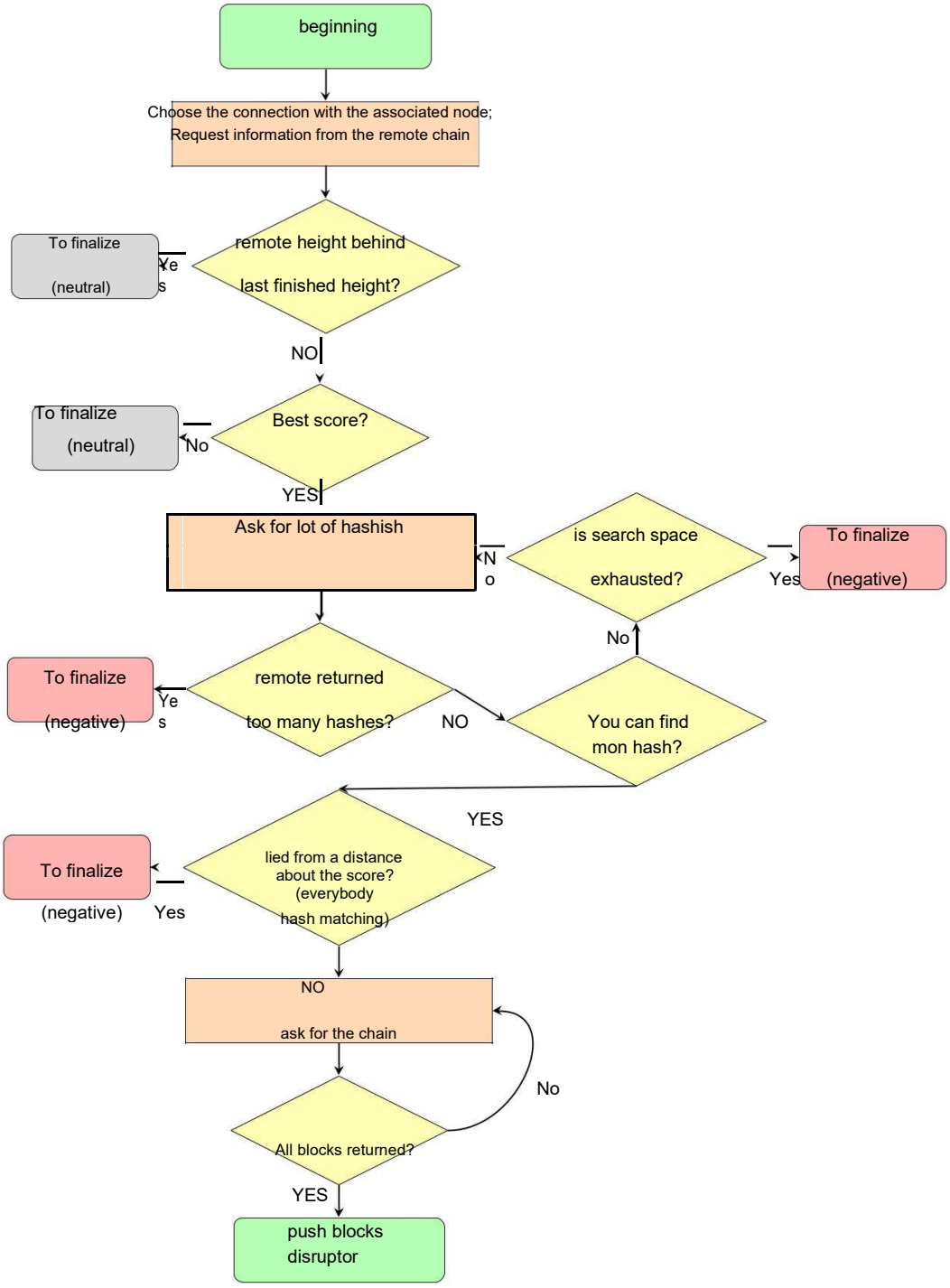


Figure 19: Blockchain Synchronization Flowchart

8.8 Blockchain processing

Execution

Conceptually, when a new block is received, it is processed in a series of stages[29]. Before processing, the block and its transactions are broken down into an ordered stream of notifications. A notification is the fundamental processing unit used in **Bitxor**.

To extract an ordered stream of notifications from a block, its transactions are decomposed in order followed by its block-level data. The notifications produced by each decomposition are added to the sequence. At the end of this process, the notification flow fully describes all state changes specified in the block and its transactions.

Once the notification flow is prepared, each notification is processed individually. First, it is validated regardless of the state of the blockchain. It is then validated against the current state of the blockchain. If any validation fails, the containing block is rejected. Otherwise, the changes specified by the notification are made to the state of the blockchain in memory, and the next notification is processed. This sequence allows transactions in a block to depend on changes made by previous transactions in the same block.

After all notifications produced by a block are processed, Receipts Hash (see7.2.4: receipt hashes) Y Hashed State (watch 7.3: state hashes) the fields are calculated and checked to see if it is correct. It is important to note that when network: enable Verifiable State is enabled, this is the point at which all Patricia state trees are updated.

Back

Occasionally, it is necessary to undo a block that has been previously committed. This is necessary to allow resolution of the fork. For example, to replace a worse block with a better block. In **Bitxor**, at most network: max Rollback Blocks can be rolled back at a time.
Forks larger than this setup are irreconcilable.

When a block is reverted, it breaks down into an ordered stream of notifications. This stream is inverted relative to the stream used during execution. Since transactions in a block can depend on changes made by previous transactions in the same block, they must be rolled back before their dependencies are rolled back.

Once the notification flow is prepared, each notification is processed individually. No validation is needed because the rollback operation returns the blockchain to a previous state that is known to be valid. Instead, the changes specified by the notification are simply rolled back from the blockchain's state in memory and the next notification is

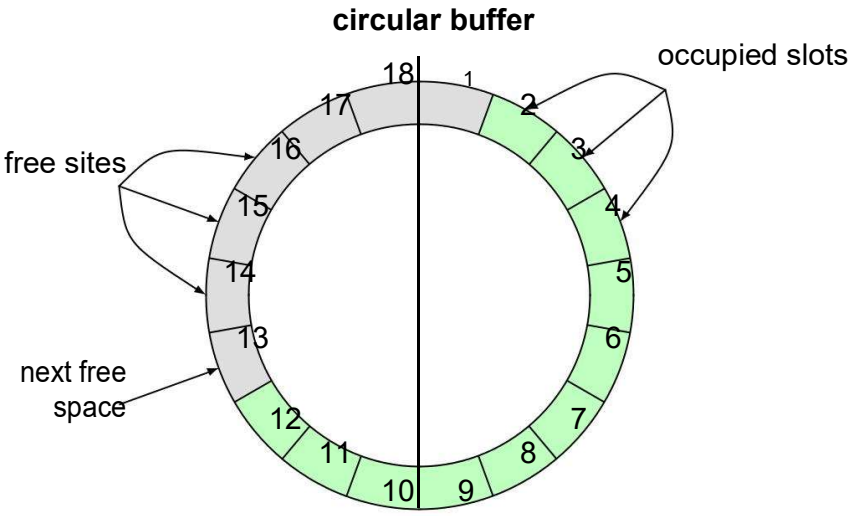
[29] A more detailed description of these stages can be found in9.1:consumers.

processed.

After all notifications produced by a part of the blockchain are processed, the previous state of the blockchain is restored. When network: enable Verifiable State is enabled, the in-memory state hash still needs to be updated. Instead of individually applying all tree changes, the in-memory state hash is forcibly reset to the state hash of the common block before the last reverted block.

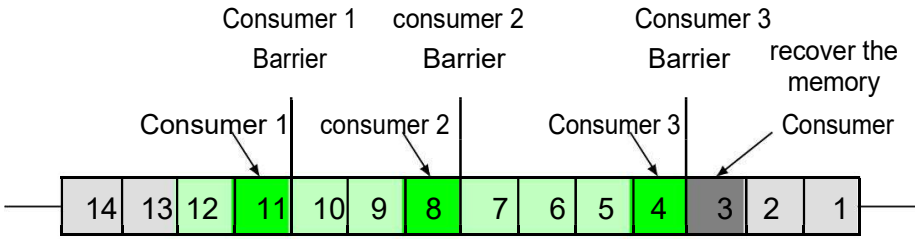
9 Disruptor

ONE main objective of **Bitxor** is to achieve high performance. To help achieve this target, the disrupter[30] pattern is used to do most of the data processing. A disruptor uses a ring buffer data structure to hold all the elements they need processing. New items are inserted into the next free slot in the ring buffer. Fully rendered items are removed to make room for new items. Since the ring buffer has a finite number of slots, it can run out of space if processing cannot keep up with new inserts. the behavior of **Bitxor**, in which case it can be configured to exit the server or discard new data until space is available.



Each item in the circular buffer is processed by one or more consumers. Each consumer takes a single element as input. Some consumers compute the data from the input and attach it to the element, while others validate the element or modify the state of the global chain using the element's data. Some consumers rely on work done by previous consumers. Therefore, consumers must always act on input elements in a predefined order. To guarantee this, each consumer has an associated barrier. The barrier prevents a consumer from processing an item that has not yet been processed by its immediately preceding consumer. The last consumer reclaims all memory that was used during processing.

Consumers can set an item's completion status to *CompletionStatus::Aborted* if it is already known or invalid for some reason. Subsequent consumers ignore the overridden elements.



9.1 Consumers

In **Bitxor**, a block disruptor is used to process incoming blocks and parts of the blockchain. A part of the blockchain is an input element made up of multiple blocks. This disruptor is primarily responsible for validating, reconciling, and growing the blockchain.

A transaction disruptor is used to process unconfirmed incoming transactions. Transactions that are fully processed are added to the uncommitted transaction cache.

All disruptors are associated with a chain of consumers that do all the processing of their input elements. The different disruptors are personalized by using different consumer strings. All consumers can inspect the data being processed, and some can modify it.

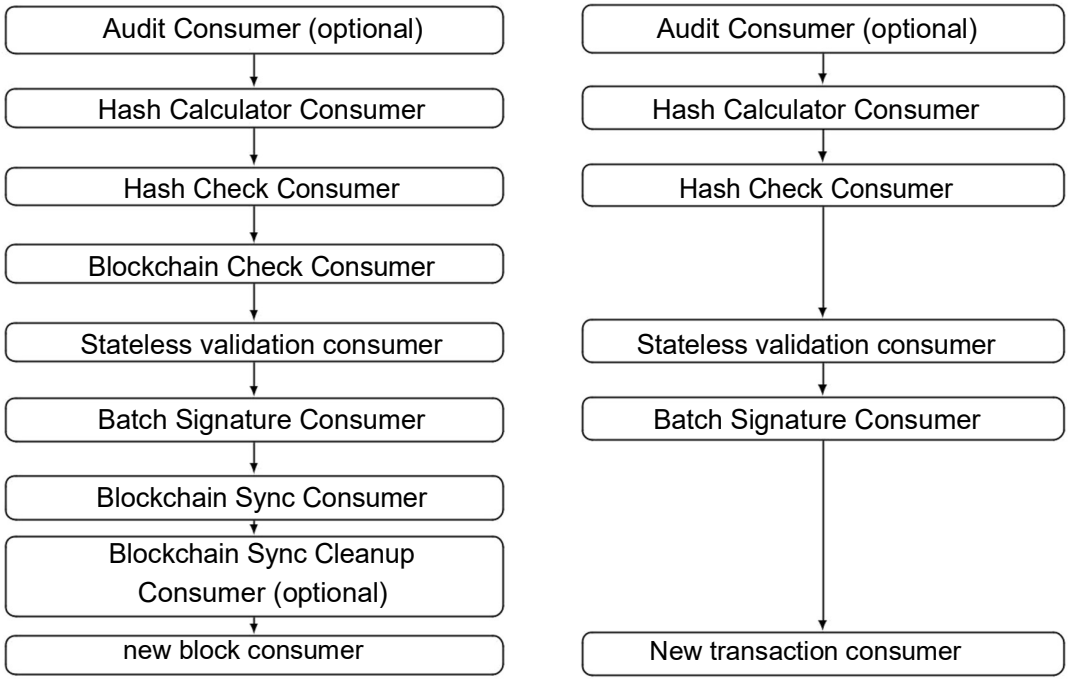


Figure 20: **Bitxor** consumer chains

9.1.1 Common Consumers

Block and transaction disruptors share a number of consumers.

Audit Consumer

This consumer is optional and can be enabled through node configuration. If enabled, all new items are written to disk. This makes it possible to reproduce the incoming network action and is useful for debugging.

Hash calculator and hash check consumers

It is very common for a server to receive the same item many times because networks consist of many servers transmitting items to multiple other servers. For performance reasons, it is desirable to detect at an early stage if an item has already been processed to avoid processing it again.

The hash calculator consumer computes all hashes associated with an item. The hashes are used by the hash check consumer to search the recent cache, which contains hashes of all recently viewed items. The consumer used by the transaction disruptor will also search the hash cache (which contains hashes of confirmed transactions) and the cache of unconfirmed transactions. If the hash is found in any cache, the item is marked as *CompletionStatus::Aborted* and further processing is skipped.

Stateless validation consumer

This consumer handles state-independent validation by validating each entity in an element. This can be done in parallel using many threads. Every plugin can add stateless validators.

An example of stateless validation is the validation that a block does not contain more transactions than the network allows. This verification depends on the network configuration, but not on the global state of the blockchain.

Batch Signature Consumer

This consumer validates all the signatures of all the entities in an element. This is separate from the stateless validation consumer because it uses batch verification. To improve performance, this consumer processes many signatures at once in a batch instead of individually. This can be done in parallel using many threads.

memory recall consumer

This consumer completes the processing of an item and frees all memory associated with it. Triggers downward propagation of the states of all transactions that were updated during processing. The overall result of the sync operation is used to update the reputation of the sync partner (and possibly ban it) (see13:Reputation)).

9.1.2 Additional Block Consumers

The block disruptor also uses some specific block consumers.

Blockchain Check Consumer

This consumer performs integrity checks independent of the state of the portion of the string contained within an element. Check that:

- The chain part is not made up of too many blocks.
- The timestamp of the last block in the chain part is not too far in the future.
- All blocks within the part of the chain are linked.
- There are no duplicate transactions within the chain part.

Blockchain Sync Consumer

This consumer is the most complex. All tasks that require or modify the state of the local server chain are performed on this consumer.

First, it checks that the new part of the chain can be joined to the existing chain. If the part of the chain joins a block that precedes the tail block, all blocks starting with the tail block are rolled back in reverse order until the common block is reached.

It then executes each block by performing stateful validation and then watching for changes. Stateless validation is skipped because it was performed by previous consumers. If there are any validation errors, the entire part of the string is rejected. Otherwise, all changes are committed to the state of the chain (both block storage and caching) and the cache of uncommitted transactions is updated.

Finally, this consumer determines the last completed block, which is the newest block that cannot be rolled back. When probabilistic completion is enabled, the last completed block is *network:maxRollbackBlocks* before the last block. When deterministic completion is enabled, the last completed block is the block referenced in the last completion test. The consumer strips the global state of the blockchain of any and all data that is only needed to allow rollbacks to blocks prior to the last completed block.

*This consumer is the only part of the **Bitxor** system that modifies the state of the chain and needs write access.*

Blockchain sync cleanup consumer

This consumer is optional and can be enabled through node configuration. If enabled, removes all files created by Blockchain Sync Consumer. This consumer should only be enabled when a server is running without a broker.

New block consumer

This consumer forwards individual blocks, either collected by the server or sent from a remote server, to other servers on the network.

9.1.3 Additional Transaction Consumers

The transaction disruptor uses a single transaction-specific consumer.

New transaction consumer

This consumer forwards all transactions that have valid signatures and have passed stateless validation to the network. Stateful validation is not performed on transactions until they are added to the uncommitted transaction cache. Forwarding is done intentionally before stateful validation because a server may reject transactions that could be accepted by other servers (for example, if the transaction has a fee too low for the local server). Subsequently, a stateful validation of the forwarded transactions is performed and the valid ones are stored in the uncommitted transaction cache.

10 Unconfirmed transactions

Any transaction that is not yet included in a block is called a *unconfirmed transaction*.

These transactions may be valid or invalid. Unconfirmed valid transactions are eligible for inclusion in a harvested block. Once a transaction is added to a block that is accepted on the blockchain, it is confirmed.

Unconfirmed transactions can reach a node when:

1. A client sends a new transaction directly to the node.
2. A linked aggregate transaction is completed with all required joint signatures and is promoted from the partial transaction cache.
3. A Peer node transmits transactions to the node.
4. A Peer node responds to the node's request for unconfirmed transactions. As an optimization, the requesting node indicates which transactions it already knows about to avoid receiving redundant transactions. Additionally, it provides the minimum rate multiplier that you use when creating blocks. This prevents the remote node from returning unconfirmed transactions that will be immediately rejected by the requesting node.

When an unconfirmed transaction reaches a node, it is added to the transaction disruptor. All transactions that have not been previously seen and pass stateless validation will be broadcast to the peer nodes. At this point, it is still possible for the node to reject transmit transactions because stateful validation is performed after transmission. Due to different node configurations, it is possible that some nodes will accept a specific unconfirmed transaction and other nodes will reject it. For example, nodes could have different node settings: min Fee Multiplier.

10.1 Unconfirmed transaction cache

When a transaction passes all validation, it is eligible for inclusion in a harvested block. At this point, the node tries to add it to the cache of uncommitted transactions. This can fail for two reasons:

1. The maximum cache size configured by node has been reached: unconfirmed Transactions Cache Max Size.
2. The cache contains at least as many unconfirmed transactions as can be included in a single block, and the new transaction is rejected by the spam regulator.

Every time new blocks are added to the blockchain, the state of the blockchain changes and the cache of unconfirmed transactions is affected. Although all transactions in the cache are valid at the time they are added, this does not guarantee that they will be valid in perpetuity. For example, a transaction may already have been included in a block collected by another node or a conflicting transaction may have been added to the blockchain. This means that transactions in the cache that were previously perfectly valid could become invalid after changes in the state of the blockchain. Also, when blocks with transactions are reversed, some of those previously confirmed transactions may no longer be included in any blocks on the new chain.

As a result of these considerations, the entire cache of unconfirmed transactions is completely rebuilt every time the blockchain changes. Stateful validators recheck each transaction and purge it if it has become invalid or has already been included in a block. Otherwise, it is added back to the cache.

10.2 Spam Accelerator

The initiator of an unconfirmed transaction does not have to pay a fee to the nodes that hold the transaction in the unconfirmed transaction cache. Since the cache uses valuable resources, a node must have some protection against spam with many unconfirmed transactions. This is especially important if the node is generous and accepts zero-fee transactions.

Simply limiting the number of unconfirmed transactions a node accepts is not optimal because normal actors should still be able to send a transaction even when a malicious actor is spamming the network. Limiting the number of unconfirmed transactions per account is also not a good option because accounts can be freely created.

Bitxor implements a smart throttle that prevents an attacker from filling the cache completely with transactions while allowing honest actors to successfully submit new unconfirmed transactions. node: enable Transaction Spam Throttling can be used to activate the throttle. Assuming the cache is not full, it works like this:

1. If the cache contains fewer uncommitted transactions than can be included in a single block, the throttling is ignored.

2. If the new transaction is a linked aggregate transaction, the limitation is ignored.
3. Otherwise, the spam throttle is applied.

Let cur Size be the current number of transactions in the cache and max Size be the maximum configured size of the cache. Also let rel. importance of A is the relative importance of A, that is, a number between 0 and 1. If a new unconfirmed transaction T arrives with signer A, then the fair share of account A is calculated:

$$\begin{aligned} maxBoostFee &= \text{transactionSpamThrottlingMaxBoostFee} \\ maxFee &= \min(maxBoostFee, T::\text{MaxFee}) \\ eff.\ \ importance &= (rel.\ \ importance\ \ of\ A) + 0.01 \cdot \frac{maxFee}{maxBoostFee} \\ fair\ share &= 100 \cdot (eff.\ \ importance) \cdot (maxSize - curSize) \cdot \exp\left(-3\frac{curSize}{maxSize}\right) \end{aligned}$$

If account A already has as many transactions in the cache as its fair share, then the new transaction is rejected. Otherwise, it is accepted. The formula shows that an increase in the maximum fee for a transaction increases the amount of available cache space. However, this mechanism for enhancing effective transcendence is limited by *node: transactionSpamThrottlingMaxBoostFee*.

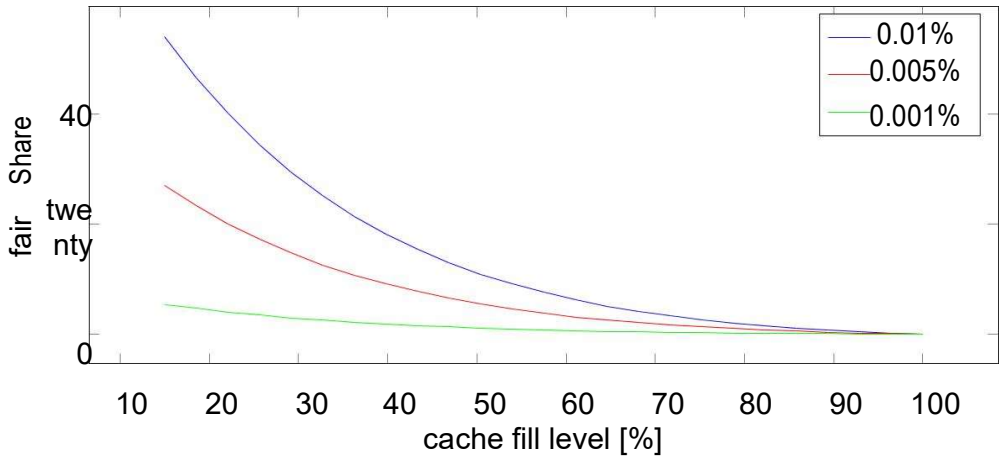


Figure 21: Fair distribution of several effective weights with maximum cache size = 10000

Figure 21 shows the fair share of slots relative to cache fill level for various actual importance. An attacker trying to occupy many slots cannot gain much by using many accounts because the importance of each account will be very low. The attacker can increase the maximum transaction fees, but that will be more expensive and will spend the funds at a faster rate.

11 Partial transactions

Bonded aggregate transactions (see 6.2: Added transaction) are also referred as partial transactions. The partial name is appropriate because the transactions have insufficient joint signatures and cannot pass validation until there are more joint signatures.

They are collected.

The partial transaction extension provides support for handling partial transactions. If a network supports linked aggregate transactions, this extension must be enabled on all API and Dual nodes.

Partial transactions are synchronized between all nodes in a network that have this extension enabled. A node passively receives partial transactions and joint signatures powered by remote nodes. It also periodically requests transactions and joint signatures from remote nodes through the extract partial transactions task. As an optimization, the requesting node indicates which transactions and joint signatures it already knows to avoid receiving redundant information.

When the hash lock plugin is enabled, in order for the network to accept a partial transaction, a hash lock must be created and associated with the transaction. The hash lock is essentially a paid bonus to be able to use the built-in co-signature collection service. If the associated partial transaction is completed and confirmed on the blockchain before the hash lock expires, the bonus is returned to the payer. Otherwise, the bond is forfeited to the harvester of the block on which the lock expires. This feature makes spamming the partial transaction cache more expensive because it requires *node:lockedFundsPerAggregate* to be paid, at least temporarily, for a partial transaction to enter the cache.

When a node receives a new partial transaction, it is sent to the partial transaction dispatcher. Received joint signatures are checked against partial transactions already in the cache and are immediately rejected if there are no matching transactions[31]. When transactions and joint signatures are received together, they are split up and processed individually as noted above.

Compared to normal transaction dispatcher (see9.1:consumers), the partial transaction dispatcher is minimalist.

[31] This implies that a partial transaction must be present in the cache before any of their joint signatures can be accepted.

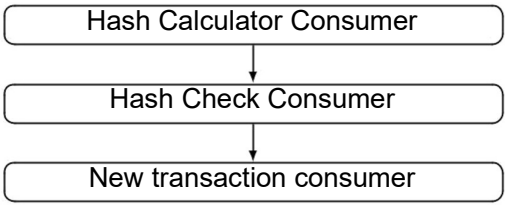


Figure 22: Partial Transaction Consumers

The hash calculator and hash check consumers work as described for the transaction dispatcher in9.1.1:Common Consumers. The only difference is that the hash verification consumer will additionally search the partial transaction cache for previously seen transactions. The new transaction consumer has a similar purpose as described in9.1.3:Additional Transaction consumers. The difference is that it transmits and processes partial transactions instead of unconfirmed transactions. Specifically, it broadcasts partial transactions to the network and then adds valid ones to the cache of partial transactions.

11.1 Partial Transaction Processing

The partial transaction cache contains all partial transactions that are waiting for additional signatures. When a new partial transaction is received that passes all validations, it is added to the cache. It will remain in the cache until a sufficient number of joint signatures are collected or until it becomes invalid. For example, the transaction will be purged if its associated hash lock expires. Whenever a partial transaction is completed by collecting enough co-signatures, it will be immediately sent to the transaction dispatcher and processed as an unconfirmed transaction.

The partial transaction cache collects all new co-signatures with the transactions it already contains. To be added to the cache, a joint signature must be new, verifiable, and associated with an existing partial transaction. It is possible for a previously accepted joint signature to become invalid, in which case it should be removed. For example, a co-signature could become invalid if its signer was removed from a multi-signature account that is involved in the partial transaction. The cosignature collection process is complex enough to handle these edge cases.

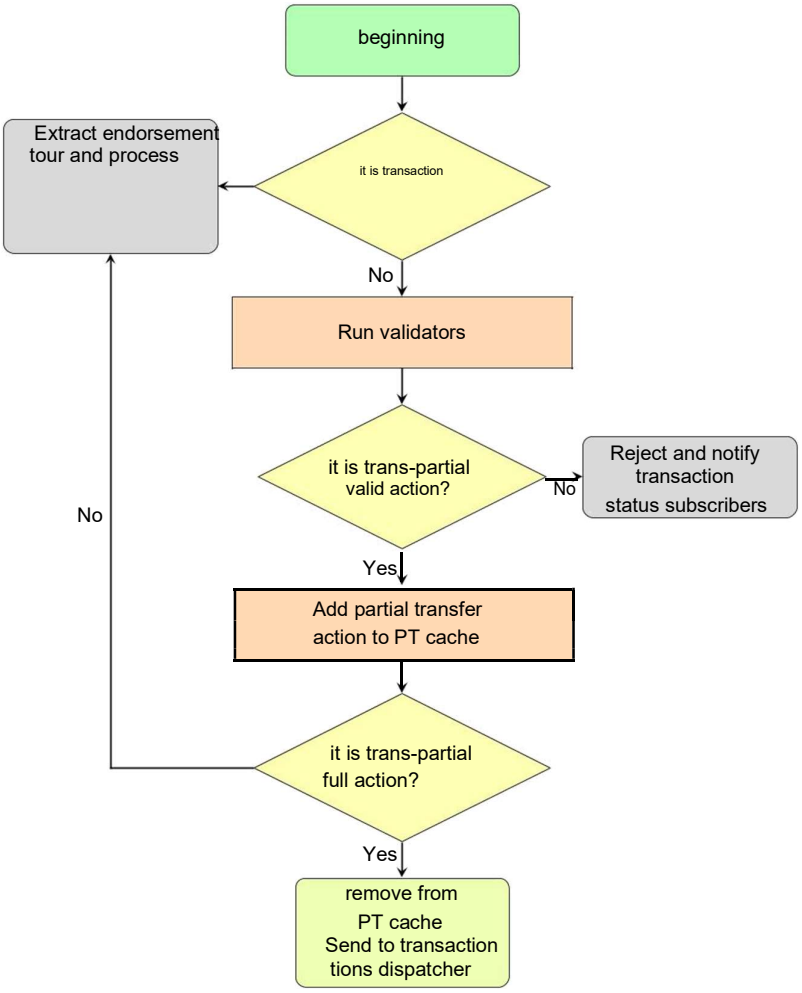


Figure 23: Processing a partial transaction

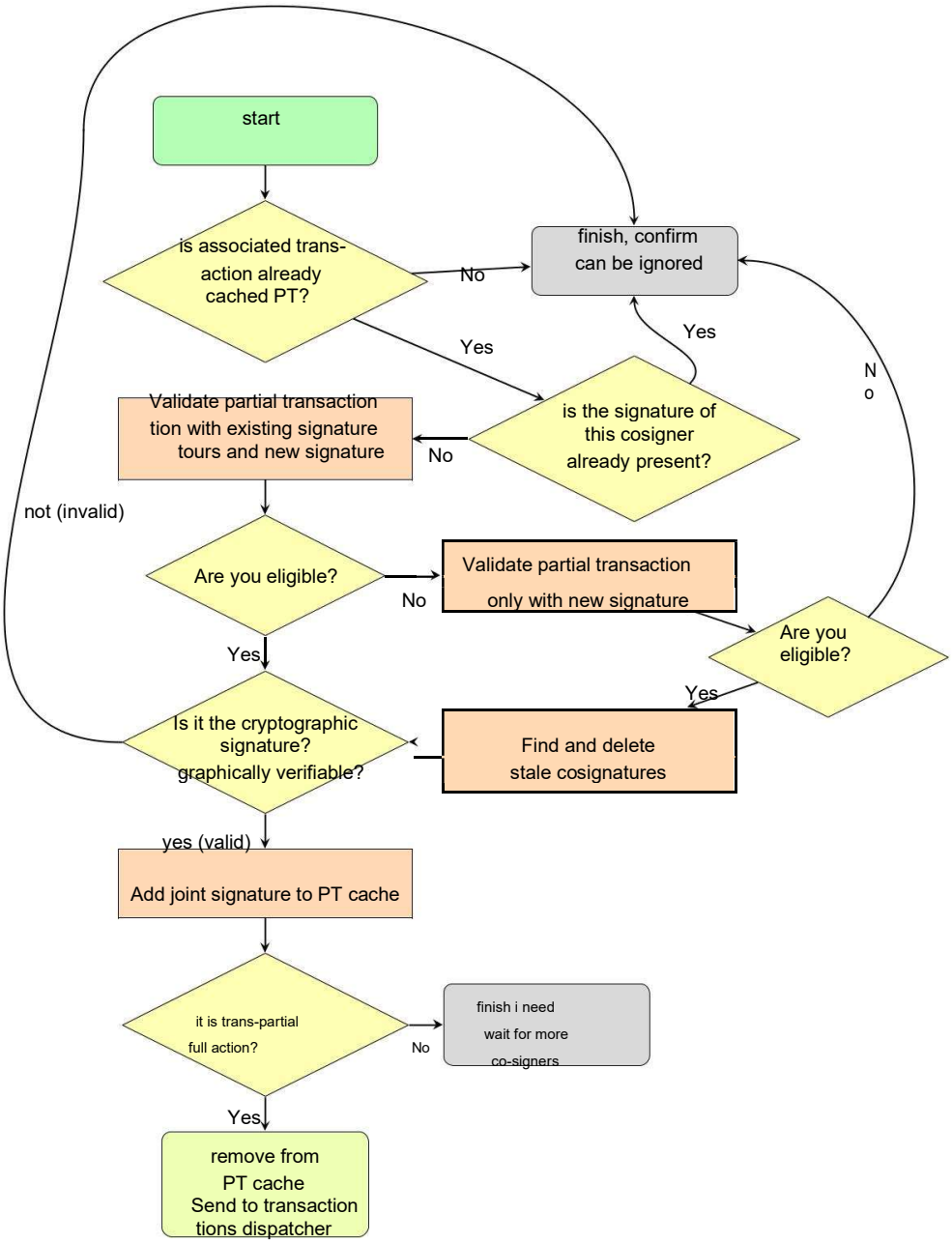


Figure 24: Processing of a joint signature

12 The net

Dinamic node discovery enables the growth of a peer-to-peer network. **Bitxor** implement this dynamic discovery in the node discovery extension. Public networks are usually open and allow any node to join. Private networks can restrict the nodes that can join and behave as a federated system[32]. **Bitxor** it is flexible enough to allow even private networks to specify all node relationships via configuration files.

Bitxor supports a configurable node identification policy configured by *the* network: nodeEqualityStrategy. Valid policies allow a node to be identified by its resolved IP (host)[33] or public boot key (public-key). The former is preferred for public networks.

12.1 Beacon nodes

A newly started node is initially isolated and not connected to any peer. You need to join a network before you can make meaningful contributions, like validating or collecting blocks. In **Bitxor**, a list of static beacon nodes is stored in a peer configuration file. To join a network, a new node first connects to these nodes. These files do not need to be identical on every node on a network.

A public network is recommended for specifying a set of HA beacon node candidates. Each node's peer configuration file should contain a random subset of these nodes. The random subset can be selected once before connecting to the network for the first time or more frequently before each boot. The important thing is that the beacon nodes are well distributed. This reduces stress on individual beacon nodes and makes DoS attacks on beacon nodes more difficult. These nodes have a slight preference in node selection (see13.2:Weight based node selection) relative to non-beacon nodes because they are assumed to be highly available. No other special privileges or responsibilities are conferred on them. They can be considered as doors to the network.

Certain extensions may require their own set of beacon nodes. For example, the partial transaction extension stores its own set of beacon nodes in a separate peer configuration

[32] By careful distribution of tokens and reward, a private network can delegate permissions to different accounts. For example, only accounts that own enough mining tokens can create blocks, and only accounts with a non-zero currency can initiate transactions with different fees.

[33] A node's resolved IP is only broadcast to other nodes when you don't specify a hostname. Hostnames are preferentially propagated to support nodes with dynamic IP addresses.

proceedings. Nodes with this extension enabled need to additionally synchronize partial transactions between other nodes that also have this extension.

A node's roles specify the capabilities that it supports. These are typically used by a connecting node to choose the appropriate partners. Nodes with the Peer role support basic synchronization. Nodes with the API role support partial transaction synchronization. Nodes with the voting role participate in the completion voting procedure when deterministic completion is enabled. Nodes with the IPv4 feature support IPv4 communication. Nodes with the IPv6 role support IPv6 communication3. 4. The roles are not mutually exclusive. Nodes can support multiple roles.

12.2 connection handshake

All connections between **Bitxor** nodes are built on top of TLS v1.3 with a custom verification procedure. Each node is expected to have a deep two-tier X509 certificate chain made up of a root certificate and a node certificate. All certificates must be X25519 certificates. **Bitxor** it does not support any other type of certificate.

The root certificate is expected to be self-signed with an account's signing private key. This account is assumed to be the sole owner of a node. The cryptographic linking of a node and an account allows the selection algorithms to perform a weighting of nodes based on the importance of the node owner. In addition, partner nodes use this verified identity for reputation checking (see13:Reputation) information35 . It is important to note that this certificate is only used to sign the node certificate. For security, your private key should not be stored on a running server.

The node certificate is signed by the root certificate. It can contain a random public/private key pair. This certificate is used to authenticate TLS sessions and obtain shared encryption keys to encrypt optional data.36. It can be rotated as many times as desired.

This authentication procedure is performed by each associated node independently. If any of the nodes fails the handshake, the connection is terminated immediately.

[34] If a node does not explicitly specify an IPv4 and/or IPv6 role, it is assumed that it only supports IPv4 communication.

[35] In the public network, nodes are primarily identified by their resolved IP.

36 Currently, the only derived encryption key is the one used to encrypt and decrypt messages related to the automatic detection of delegated collectors (see8.6:Automatic detection of delegated harvester).

12.3 packages

Bitxor uses TCP for network communication on the port specified by *node: port*. The communication is centered around a higher level packet model in addition to and is distinct from TCP packets. All packets begin with an 8-byte header that specifies the size and type of each packet. Once a complete package is received, it is ready for further processing.

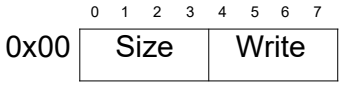


Figure 25: Packet Header Binary Layout

A combination of long-term and short-term connections are used. Long-lived connections are used for repetitive activities like synchronizing blocks or transactions. They support both push and request/response semantics. Connections are allowed to last for *node:maxConnectionAge* selection rounds (see13.1:Connection Management) before they are eligible for recycling. Connections older than this setting are recycled primarily to allow direct interactions with other partner nodes, and secondarily as a precaution against zombie connections.

Short-lived connections are used for more complex multistage interactions between nodes. For example, they are used for node discovery (see12.6:Node discovery) and time synchronization. Short-lived connections help prevent out-of-sync, which can occur when all long-lived connections are in use and no sync partners are available.

Handlers are used to process packets. Each handler is registered to accept all packets with a specific packet type. When a complete packet is ready for processing, it is sent to the controller registered with its type. All controllers must accept matching packets for processing. Some drivers can also write response packets to allow request/response protocols.

12.4 connection types

In **Bitxor**, long-term connections are primarily identified as readers or writers. This is orthogonal to whether they are incoming or outgoing. They are secondarily identified by purpose, or service identifier37. This allows connections to be selected by capacity and more granular logging.

[37] Although the terminology is similar, these are not related to the services described in2.2:**Bitxor** Extensions.

Reader connections are mostly passive and are used to receive data from other nodes. Each server reads asynchronously from each reader connection. Every time a new packet is received in its entirety, it is sent to an appropriate controller. If no matching driver is available, the connection is closed immediately.

The *node:maxIncomingConnectionsPerIdentity* limit applies to all long-lived and short-lived connections and services. Any incoming connection above this limit will be closed immediately. This limit can be reached when multiple short-lived connections are initiated to the same remote node for different operations. This is most likely when connection tasks are scheduled more aggressively immediately after a node starts. These errors are usually transient and can be safely ignored if they do not persist.

Writer connections are more active and are used to send data to other nodes. Broadcast operations send data to all active and available writers. Additionally, writers can be individually selected and used for request/response protocols. To simplify recipient processing, writers participating in an ongoing request/response protocol do not receive broadcast packets.

Service IDs are only assigned to long-lived connections. The synchronization service is used to manage outgoing connections to nodes with the Peer role. The API Partial Service is used to manage outgoing connections to API role nodes. The readers service is used to manage incoming connections. The API writers service is experimental and allows incoming connections on the port node: apiPort to be registered as writers.

identifier	Name	Address
0x50415254	pt writers	outgoing
0x52454144	readers	coming
0x53594E43	sync	oytgoing

Figure 26: Service Identifiers

For the purposes of node selection described in13.2:Weight based node selection, age, and node selection are scoped by service. Reputation information is aggregated across all services. Specifically, suppose that one node has made a partial sync and API connection to another node. Each connection can have a different age because the age is defined per service. Interaction results, from any connection, are always attributed to the node, not the service.

12.5 Origin of peers

A node collects data about all the nodes on its network. The reputation of the data depends on its origin. The possible sources, ordered from best to worst, are:

1. Local: The node is specified in *node:localnode.*
2. Static – The node is in one of the peer configuration files.
3. Dynamic: The node has been discovered and is supporting connections.
4. Dynamic input: the node has made a connection but does not support connections.

It is important to note that the distinguishing feature of a static node is that it appears in at least one local peer configuration file. For emphasis, it is possible for one partner to see a node as static while another partner sees it as dynamic. Except for the local node, all other nodes are dynamic. You are entering a subset of dynamic nodes. These nodes have only been seen on incoming connections but not outgoing. As a result, their preferred port is unknown and they cannot connect.

Existing node data can only be updated if the new data is not from a worse provenance than the existing data. For example, updated information about a static node sourced dynamically is discarded, but updated information about a dynamic node sourced dynamically or statically is allowed.

The above is a slight simplification due to how connections are actually handled. When a node completely disconnects from a remote node and reconnects, the upgrade can be a two-step process. Consider a dynamic node trying to reconnect to a remote with a different identity public key. When the node initiates a connection, the remote will classify the connection as dynamic incoming, which has worse provenance than dynamic. As a result, the remote will not update the node information. Instead, it will set a flag indicating a possible identity update in progress. Later, when the remote connects directly to the node, it will get the same updated information as before. In this point, the remote will update the information even if there is an active connection (dynamic incoming) because an identity update in progress was previously detected. Without this flag, the active connection from the worst source would block updates, which is not desirable.

When *network:nodeEqualityStrategy* is a public key, the secondary identity component is the resolved IP. When there are no active connections, this is allowed to change. This strategy does not support reputation migration.

When *network:nodeEqualityStrategy* is host, the secondary identity component is the identity public key of the node. When there are no active connections, this is allowed to change.

The IP Resolved parent identity component can also be changed when there are no active connections, assuming the child identity component is not changed. In this case, all reputation data associated with the original host is migrated to the new host. When there is an ambiguous match, the data with the matching parent identity component is migrated and the data with the matching child identity component is discarded.

12.6 Node discovery

After starting, a node attempts to establish short-lived connections to all static nodes that it has loaded from its peers' configuration files. These connections are primarily intended to retrieve the resolved IP addresses of all static nodes. This allows host names to be used in peer configuration files and simplifies node management. As long as the node is running, this procedure is repeated periodically with a linear backtracking.

Periodically, a node will transmit identifying information about itself to its remote partner nodes. The remote will process the received payload and check its validity and compatibility. To be valid, the identity public key specified by the node must match the public key of its X509 root certificate. To be compatible, both transmitting and receiving nodes must point to the same network. If no hostname is provided, the resolved IP of the node will be used. If all checks are successful, the node will be added as a new potential partner and will be eligible for selection in the next sync round.

Periodically, a node will request all known peers from its remote partner nodes. Remote nodes will respond with all of their static and dynamic peers active. For the requesting node, all of these will be treated as dynamic nodes. The original node will request the identification information of each of these nodes directly. This direct communication is necessary to prevent a malicious actor from passing false information about other nodes and to ensure that a connection can be established with each new node. The originating node will process the received payload and verify its validity and compatibility as stated above. If all checks are successful, the new node will be added as a potential new partner and will be eligible for selection in the next sync round.

13 Reputation

Bitxor uses a peer-to-peer (P2P) network. P2P networks have the great advantage to be robust because they cannot be turned off by removing a single node. However, a public network comes with its own challenges. Network participants are anonymous and anyone can join. This makes it very easy to inject hostoken nodes on the network that spread invalid information or try to disrupt the network in some way.

There is a need to identify hostoken nodes and reduce communication with them. There have been many approaches to accomplish this. One of the most successful is building a reputation system for nodes. Bitxor follows this approach by implementing a simple reputation system. This system attempts to prioritize connections to nodes that behave well over those to nodes that behave poorly. Importantly, reputation does not affect blockchain consensus at all. It only influences the network graph. This chapter describes the heuristics used.

13.1 Connection Management

Each node can establish at most *node:maxConnections* persistent connections at a time. This limit is expected to be much smaller than the hundreds of thousands of nodes that make up the network as a whole. To prevent groups of isolated nodes from forming, a node will periodically drop existing connections to make room for new connections to different nodes.

In determining which nodes to disconnect from, a node inspects the ages of all its connections. To minimize connection overhead, only connections that have been established for at least *node:maxConnectionAge* rounds are eligible for removal. The next time a node selection round is performed, these connections are discarded and replaced with new connections to other nodes. This ensures that over time each node will establish connections to many different nodes on the network.

13.2 Weight based node selection

Nodes communicate with each other primarily through the current persistent connections they have established. A node can query another node for new transactions or blocks, or request a list of other nodes that the associated node has interacted with. Nodes can also voluntarily send data to other nodes. Each communication between nodes is considered an interaction, and each interaction is scored as successful, neutral, or failed. For example, when a remote node sends new valid data, the interaction is considered successful because it has contributed to the synchronization of the two nodes. If the remote node has no new data, the interaction is neutral. Otherwise, the interaction is considered failed.

Each node keeps track of the results of its own interactions with other nodes. These results are only used locally and are not shared with other nodes. A node's interactions with other nodes influence which partner nodes it selects. Interaction results are stored for a maximum of one week, but are reset when the node is rebooted. These results are time-bound to allow nodes that have transient failures to reestablish themselves as good partners.

When selecting associated nodes, a node first determines a set of candidate nodes. Each candidate node is assigned a raw weight between 500 and 10,000 according to the following criteria:

- If there were 3 or fewer non-neutral interactions with the remote node, it is given a mean raw weight of 5000. This gives new nodes a good chance of being selected.
- Otherwise, let s be the number of successful interactions and f the number of failed interactions. Then the gross weight is calculated by the following formula:

$$rawWeight = \max\left(500, \frac{s \cdot 10000}{s + 9 \cdot f}\right)$$

This formula ensures that failed interactions rapidly decrease the weight of a remote node and its probability of being selected. The presence of a minimum score still gives a heavily failed node a small chance of being selected and possibly improving its score with more interactions.

The raw weight is multiplied with a weight multiplier to give the final weight of a node. For static nodes, the multiplier is 2. For dynamic nodes, it is 1. If a node is banned due to consecutive interaction failures (see13.3:node ban), the multiplier is reduced by 1. This ensures that a node does not connect to dynamic forbidden nodes. The chance to connect to static forbidden nodes is halved.

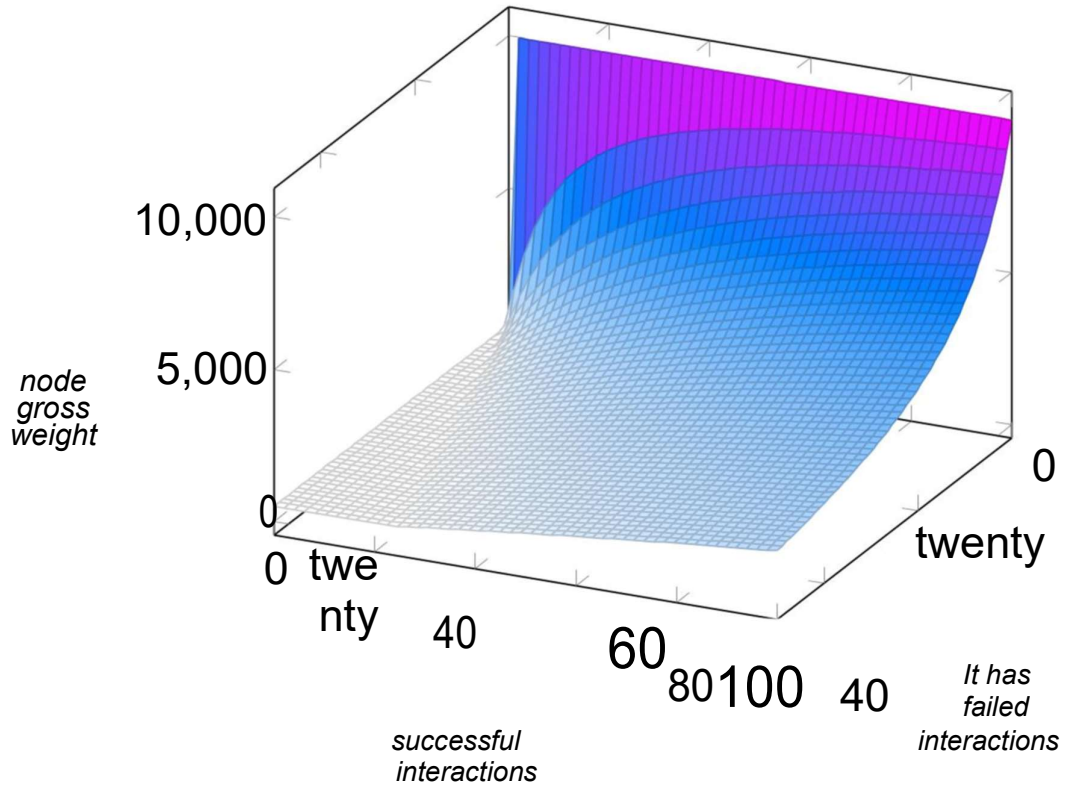


Figure 27: Raw Node Weight

Elimination candidates are determined based on their connection age. Each kill candidate to be closed is replaced with a connection to a new node so that the node maintains the desired level of connections. Finally, for each free space, a candidate node has the possibility of being selected given by:

$$P(\text{node is getting selected}) = \frac{node\ weight}{\sum\limits_{\substack{candidates\\nodes}} candidate\ node\ weight}$$

13.3 Node Banning

In a public network there could be potentially malicious nodes trying to disrupt normal network processing. Therefore, if a node deems a remote node to be malicious, it will prevent connecting to that node and will not accept incoming connections from it.

The ban is applied at the node level and is attached to a node's network scope identifier Page

(watch12.6:Node discovery). A misbehaving node will be immediately banned for a period of *node:defaultBanDuration*. Even after a node is no longer actively banned, the local node will remember for some time (*node:keepAliveDuration*) that the node was misbehaving and will treat repeated violations more severely by banning the node for longer periods up to *node:maxBanDuration* . During the ban, no connections will be established with the banned node. After the ban expires, the node is again treated as a normal interaction partner. There are several scenarios where a remote node will be banned. Penalties vary depending on the cause.

	Connection closed	Remote they can reconnect	Remote can be selected	Remote able to send data
Consecutive Interaction failures	Nope	-	yes static no dynamic	Yes
Invalid data	If all[38]	Nope	Nope	Nope
Exceeded read rate	If all	Nope	Nope	Nope
Unexpected data	Yes	Yes	Yes (after reconnect)	Yes (after reconnect)

Figure 28: Ban Rules

Consecutive interaction errors

If interactions with the same node fail too many times in a row due to network or state failures, it's best to suspend all interactions with that node for a while, in the hope that the node will behave better in the future. The number of consecutive interaction failures before the node is banned as an interaction partner can be configured. The amount of time the node is locked is measured in selection rounds and can also be configured. As long as the node is banned, it will not be actively selected as an interaction partner, but it can still send new data. This violation therefore only results in a partial ban.

[38] All active connections associated with the misbehaving node are closed immediately, not just the connection that triggered the violation.

invalid data

The data may be invalid in many ways. For example, if a remote node is on a fork, it might send a new block that doesn't fit into the local node's chain. Small forks one or two blocks deep occur frequently. Although the data sent is invalid, it is not considered malicious because the internal state of the remote node was understandably different. On the other hand, sending data with invalid signatures clearly indicates that the remote node is malicious because signature verification is independent of a node's state. The same goes for other verification failures that do not depend on the state of a node. In all such cases, the remote node is prohibited.

Data read rate exceeded

Each node monitors the read speeds of all sockets that accept data from its peers. This allows a node to detect when a faulty peer is producing an unexpected amount of data. If the data read during a configured time interval exceeds a maximum, the socket is closed and the node is banned. The maximum read rate is configurable.

Receive data unexpectedly

There are situations during node communication where the local node does not expect to receive any data from the remote. If the remote still sends data in such a situation, it is violating the protocol and the connection is closed. In this case, the connection is closed immediately but there is no persistent node ban.

14 Consensus

Byzantine consensus is a key problem they face all decentralized systems. Essentially, the crux of the problem is finding a way to get independent actors to cooperate without cheating. The innovation Bitcoin key was a solution to this issue that is based on Proof of Work (PoW). After each new is accepted block on the main Bitcoin chain, all the miners start a competition to find the next block. All miners have incentives to extend the chain instead of the forks because the chain with the greatest cumulative power of hashing is the reference string. Miners calculate hashes as fast as possible until one produces a candidate block with a hash below the current target of network difficulty. The probability that a miner will mine a block is proportional to the hash rate of the miner relative to the total hash rate of the network. this leads necessarily to a computational arms race and consumes a lot of electricity.

Proof of Stake (PoS) Blockchains were introduced after Bitcoin. They presented an alternative solution to the Byzantine consensus problem that did not require significant power consumption. Basically, these chains behaved similarly to Bitcoin with one important difference. Instead of predicating the probability of creating a block on a node's relative hash rate, the probability is based on a node's relative share of the network. Since richer accounts can produce more blocks than poorer accounts, this scheme tends to allow the rich to get richer.

Bitxor uses a modified version of PoS that attempts to grant *users* preferably in relation to hoarders. It strives to calculate a holistic score of an account's importance without sacrificing performance and scalability.

There are multiple factors that contribute to a healthy ecosystem. All things being equal, accounts with larger stakes that transact more and run nodes have more stake in the game and should be rewarded accordingly. First, accounts with larger balances have larger shares in the network and have greater incentives for the ecosystem as a whole to succeed. The amount of currency an account holds is a measure of its holding. Second, accounts should be encouraged to use the network by conducting transactions. Network usage can be approximated by the total amount of transaction fees paid for an account. Third, accounts should be encouraged to run nodes to strengthen the network. This could be

approximated by the number of times an account is a beneficiary of a block[39]. Since the node owner has full control over their beneficiary definition, any benevolent node owner can alternatively push this measure for a third party.

Importances are recalculated every network: importance grouping blocks. This reduces the pressure on the blockchain because calculating importance is relatively expensive and processing each block would be prohibitively expensive. Additionally, recalculating weights periodically allows for automatic aging of the state. In general, it is beneficial to calculate the weights periodically rather than every block.

To encourage good behavior, accounts active in an older time period should not gain an eternal advantage due to previous virtuous behavior. Instead, importance boosts granted by transactions and node scores are time-limited. The impulse lasts five *network:groupingimportance*intervals

14.1 Weighting algorithm

All accounts that have a balance of at least *network:minHarvesterBalance* participate in the importance calculation and are called high value accounts. Note that this set of accounts is a superset of the set of accounts eligible for block generation (see8.3:Block generation). In other words, a non-zero importance at the most recent importance recalculation is a necessary but not sufficient condition for block generation.

An account's importance score is calculated by combining three component scores: engagement, transaction, and node.

The stake score, SA, for an account A is the percentage of currency it holds relative to the total currency held by all high value accounts. This percentage is not less than the percentage of currency held by the account in relation to all currency in circulation. Let BArepresent the amount of currency owned by account A. The wagering score for account A is calculated for each eligible account as follows:

$$S_A = \frac{B_A}{\sum_{a \in high\ value\ accounts} B_a}$$

The transaction score, T_A, for an account A is the percentage of transaction fees it has paid relative to all fees paid by high-value accounts within a time period P. Let FeesPaid A be the amount of fees paid by A in time period P . The transaction score for account A is calculated for each eligible account as follows:

[39] This measure is strongly correlated with engagement when all accounts are actively running nodes. Its intent is to differentiate accounts running nodes from idle accounts.

$$T_A = \frac{\text{FeesPaid}(A)}{\sum_{a \in high\ value\ accounts} \text{FeesPaid}(a)}$$

The node score, NA, for an account A is the percentage of times it has been specified as a beneficiary relative to the total number of high-value account beneficiaries within a time period P . Let BeneficiaryCount A be the number of times A has been specified as a beneficiary in time period P . The node score for account A is calculated for each eligible account as follows:

$$N_A = \frac{\text{BeneficiaryCount}(A)}{\sum_{a \in high\ value\ accounts} \text{BeneficiaryCount}(a)}$$

Together, transaction and node scores are called activity scores because they are both dynamic and derived from an account's activity rather than its participation. The transaction score is weighted at 80% and the node score at 20%. Additionally, the combined score is scaled relative to an account balance, so there is a moderating effect of activity on importance as engagement increases.[40]. This allows smaller active accounts to get a big boost relative to larger active accounts. This partially redistributes the importance of rich accounts towards poorer accounts and somewhat counteracts the rich getting richer phenomenon inherent in PoS. The prominence of the activity relative to participation can be set using *network:importanceActivityPercentage*. When this value is zero, **Bitxor** it behaves like a pure PoS blockchain. Setting this too high could weaken blockchain security by reducing the cost for an attacker to gain majority importance and execute a 51% attack.

As a performance optimization, activity information is only collected for accounts that are high value at the time of the most recent importance calculation. Between important recalculations, the new data is stored in a working cube. On each importance recalculation, existing cubes are moved, the working cube is finalized, and a new working cube is created. Each bucket influences at most five important recalculations. As a result, the activity information expires quickly.

The *network:totalChainImportance* setting specifies the total importance that is distributed across all accounts in a network. Since, the punctual importance of the account

[40]The activity score is rescaled after buffering to contribute the desired *network:importance-ActivityPercentage* to the importance calculation.

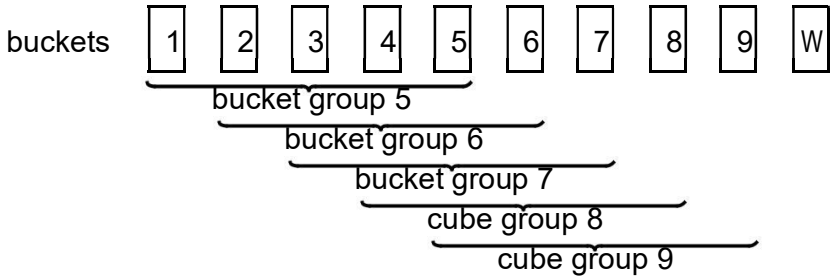


Figure 29: Activity Cubes

A, $ME_{A'}$, can be calculated as follows:

$$\gamma = \text{importanceActivityPercentage}$$

$$ActivityScore'_A = \frac{\text{minHarvesterBalance}}{B_A} \cdot (0.8 \cdot T_A + 0.2 \cdot N_A)$$

$$ActivityScore_A = \frac{ActivityScore'_A}{\sum_{a \in high\ value\ accounts} ActivityScore'_a}$$

$$I'_A = \text{totalChainImportance} \cdot ((1-\gamma) \cdot S_A + \gamma \cdot ActivityScore_A)$$

The final importance score, I_A for account A, is calculated as the minimum of I'_A in the current and previous importance calculations. This serves as a precaution against a stake crushing attack and a general incentive to minimize unnecessary movement of stakes. There is no scaling, so the AI sum for all high value accounts can be less than the network: *totalChainImportance*.

14.2 sibyl attack

Sybil's attack[42] in a peer-to-peer network it occurs when an attacker creates multiple identities to gain disproportionately large influence over the network or some other advantage. In **Bitxor**, an attacker might attempt such an attack to increase prominence. Each component of the importance score must be robust against these types of attacks.

As described in14.1:Weighting algorithm, an account's activity score is reduced relative to its balance. Consequently, dividing the balance of one account among several accounts

[41]There is additional edge case handling that is not reflected in the equation for how zero component scores are handled. If the transaction or node score is zero, the other will be scaled up and serve as the fully weighted activity score. If both are zero, the bet score will be extended and used exclusively.

will reduce the applied average damping factor. Assuming a constant level of activity is maintained before and after the redistribution, the cumulative importance will be greater after the split.[43]. This effect is by design and encourages virtuous behavior because the prominence drive is only realized if the activity is sustained. Preservation of transaction score encourages transaction and payment of fees from multiple accounts. The preservation of the node score encourages the execution of additional nodes and their connection to the network.

Suppose μ := minHarvesterBalance and an attacker who owns N · μ total currency. Consider two extremes:

1. The attacker has a single account with N · μ currency.
2. The attacker has N accounts with μ currency.

Bet Score Increase

At both ends, the total currency held by the attacker is the same. Consequently, the bet score is the same and no profit is made by splitting the accounts. For emphasis:

$$B_A = \sum_{a \in \{1,\dots,N\}} E \tag{16}$$

Boost node score

Bitxor allows the owner of a node to specify a beneficiary for each block harvested on their node. Every time an account is specified as a beneficiary, assuming it's already a high value account, you'll get a slight increase in your node score.

At both ends, the total payee count for the attacker is the same. Consequently, the node's score is the same and there is no unearned benefit from splitting the accounts. For emphasis:

$$\text{BeneficiaryCount}(A) = \sum_{a \in \{1,\dots,N\}} \text{BeneficiaryCount}(a) \tag{17}$$

The attacker could get a higher node score if running more nodes allows the attacker's nodes to host more proxy collectors. This is not a bad result and by design. It's

[43] This assumes that only one account is split. The effect is reduced when multiple accounts are split because activity scores are relative.

it encourages more nodes in the network, which is a good thing that makes the network stronger.

The attacker could try to cheat by setting up N virtual nodes pointing to a single physical machine. Each of these virtual nodes would be treated by the rest of the network as a normal node, and the underlying physical node would interact with it N times more often than a normal node in the network. This implies that the virtual nodes are running on a strong physical server, which is still beneficial to the network relative to a weaker physical server.

Increased transaction score

The transaction score is based on fees only. There is no difference between a large account spending X on fees and N smaller accounts each spending X on fees. For emphasis:

$$\text{FeesPaid}(A) = \sum_{a \in \{1,\dots,N\}} \text{FeesPaid}(a) \tag{18}$$

The only chance to increase the transaction score is a fee attack, which is discussed in detail in14.4:Tariff attack.

14.3 Nothing at stake attack

A general criticism of the PoS consensus is the nothing at stake attack44. This attack theoretically exists when the opportunity cost of creating a block is negligible. There are two variations of this attack.

In the first variation, all harvesters except the attacker harvest on all forks. Simplifying the description to assume a binary fork, the attacker would send a payment to one branch and immediately begin harvesting at the other branch. Assuming the attacker is important enough to collect blocks, eventually the branch without the attacker's payment will become the referral chain because it will have a higher score.Four. Five. The attacker's payment is not included in this branch, so the attacker's funds are effectively returned.

There are three main defenses against this attack. First, the attacker has a limited amount of time to produce a better chain because, at most, *network:maxRollbackBlocks*

[44] This assumes there is only one attacker or all attackers collude to withhold the reward from the same branch.

blocks can be reversed. If the merchant waits to provide services until at least this number of blocks is confirmed, the attack is impossible. Second, to execute a successful nothing-in-play attack, the attacker must possess significant importance in the network.46. Third, the successful execution of this attack against the network will likely have a negative influence on the value of the coin. Since other harvesters, by harvesting on all forks, allow this attack, profit-maximizing harvesters should only harvest on a single chain to avoid it.

In the second variation, a single attacker collects on all forks and attempts to capture all fees, regardless of which chain becomes the referring chain. An attacker could collect on all forks starting from the second block looking for the chain where the attacker collected the most fees. Since block acceptance is probabilistic, in theory an attacker could spend infinite time building the perfect chain where the attacker has harvested all the blocks.

Most theoretical nothing-stakes attacks envision an idealized blockchain and ignore the protocol-level safeguards that protect against such attacks. In practice, this type of attack is not practical if the attacker has a minority of currencies. The two aforementioned defenses are also applicable here. In addition, changes in the difficulty of the blocks (see8.1:block difficulty) are capped at 5%. It will take some time for the attacker chain difficulty to adjust downwards, which will make lock times at the start of the secret chain significantly slower than the main chain. These large time differences will make it unlikely that the attacker will produce a chain with a better score (see8.2:Block Score).

A small amount of betting age also decreases the probability of this second variation. Requiring accounts to have non-zero importance for two consecutive importance recalculations as a precondition for harvesting makes hash generation annoying47 non-viable attacks. To exploit this, the attacker would need to move all the currency to a specific account plus *network:importanceGrouping* blocks before the attack could be carried out. Since the attacker cannot know all the locks that will be confirmed in the interim period, such a move cannot result in any benefit.

14.4 Tariff attack

A fee attack is an attempt by an attacker to exploit the transaction score by paying high fees to increase their own importance. The attack is considered effective if it produces a

[46] Theoretically, an attacker would need only *network:minHarvesterBalance* to execute this attack. In practice, to guarantee a successful execution, the attacker would need a importance large enough to always harvest a block within the rollback interval.

positive expected value.

The analysis in this section will be performed using the recommended public network settings. These include *network:totalChainImportance* equal to 9 billion, *network:importanceGrouping* equal to 359 blocks, and *minHarvesterBalance*:equal to 10000 coins. Also, *network:importanceActivityPercentage* is 5, so the cumulative transaction score (seeEquation 14.1:Weighting algorithm) represents 4% of importance.

big account

Consider an account that is large enough to collect one block per importance recalculation interval without any activity pulse. Assuming only 2 of 9 billion currency is being actively collected, the account will need at least 5.57 million currency to collect this frequently.

The account could try to make a profit by adding a transaction with a high fee to one of its own collected blocks at each recalculation interval. This would increase the importance of the account and allow you to collect more blocks in the future and consequently charge more fees. However, this activity is not without risk. The account is at risk of paying the high fee if a better block replaces its block. When the original block is undone, the high-fee transaction will enter the cache of unconfirmed transactions and will be eligible for inclusion in a new block created by a different collector. This scenario is a net loss because the account will have to pay the high fee.

$$\beta = 0.04 \cdot \frac{1}{557} \cdot \frac{F}{359 \cdot \bar{F} + F} \qquad \text{(importance boost)}$$

$$EV = \beta \cdot \frac{359}{P} \cdot \bar{F} - F \qquad \text{(expected value)}$$

The expected value is positive for small values of P . As P or F increases, it quickly becomes negative. With the recommended public network configuration, P must be less than 0.0001 for the expected value to be positive. This implies that a fork resulting in a loss occurs less than once every 10,000 blocks. Given the distributed consensus mechanism, this is almost impossible. Small forks of one or two blocks occur quite often.

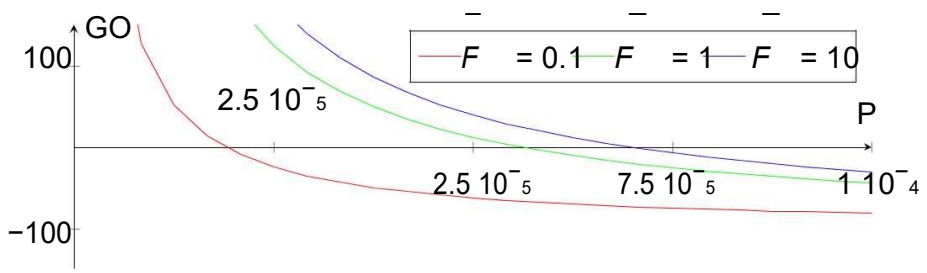


Figure 30: Fee Attack Large Account Analysis (F = 100)

small account

Consider an account that has a balance equal to network:minHarvesterBalance. Suppose the account makes a transaction with high fees in two consecutive recalculation intervals. These fees are lost to other collectors because the probability of the account collecting a block is quite small. The high fees paid increase the importance of the account enough that you can collect at least one block per importance recalculation interval. From this point on, the account behaves like the large account from the previous section. It will also add a transaction with a high fee to one of its own collected blocks at each recalculation interval. The account owner expects that, due to the increased probability of collecting a block, the additional fees charged will exceed their costs.

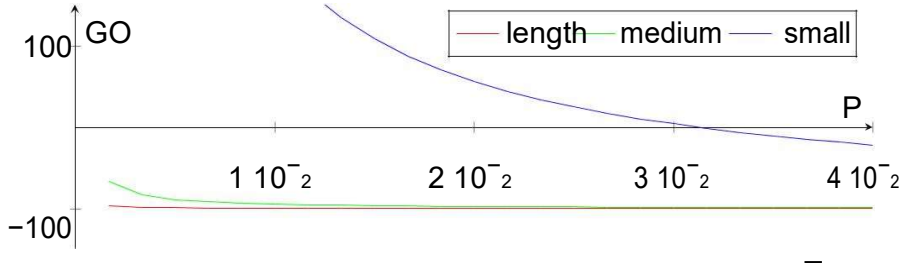


Figure 31: Fee Attack Balance Sensitivity (F = 100, $\bar{F}$ = 1)

Let P be the probability of a fork resulting in a loss, F be the high fee in a block, and $\bar{F}$ be the average fee in a block. The expected value, EV , excluding the initial transaction fees, can be approximated as follows:

$$\beta = 0.04 \cdot \frac{F}{359 \cdot \bar{F} + F} \qquad \text{(importance boost)}$$

$$EV = \beta \cdot \frac{359}{P} \cdot \bar{F} - F \qquad \text{(expected value)}$$

[48]The difference from the large account example is that the damping factor is completely removed.either

The expected value is positive for larger values of P than in the large accounts scenario. A fee attack confers an outsized benefit to a small account relative to a large account because the latter's activity scores are lowered more aggressively than the former's. Specifically, a damping factor of 5571 is applied to the large account activity score, but no damping factor is applied to the small account activity score.

The expected value increases as $\bar{F}$ increases. As P or F increases, it quickly becomes negative. Using the recommended public network settings, P needs to be less than 0.05 for the expected value to be positive. This implies a fork resulting in a loss occurs less than once every 20 blocks. Given the mechanism of distributed consensus, this is possible.

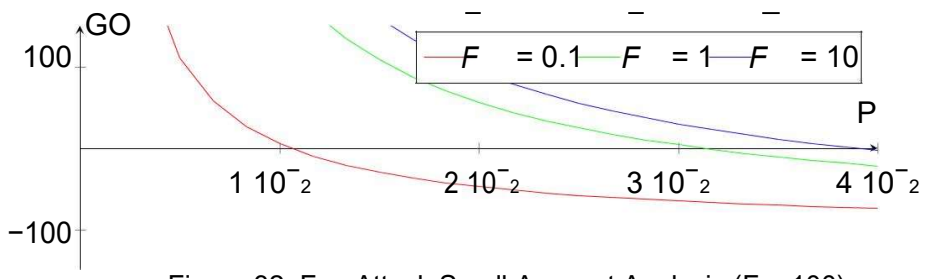


Figure 32: Fee Attack Small Account Analysis (F = 100)

more discussion

Although a single small account can gain a positive expected value from executing this attack, the reward decreases as multiple accounts attempt it simultaneously. Since there is a positive expected value, all profit-maximizing actors should attempt this attack. As more accounts try, the boost in importance earned by each individual account decreases rapidly, and consequently the expected value decreases as well.[49].

Also, there is an upper limit on the number of small accounts that can run this attack simultaneously. For this attack to be successful, an account must be able to collect at least one block per importance recalculation interval. This assumes that the small account can increase its importance score by exploiting the transaction score component. There is a theoretical limit to the number of accounts that can achieve a significant enough boost because both the importance assigned to the transaction score and the recalculation interval are finite. Considering the recommended public network configuration, this limit is approximately 0.04 ÷3591≈ 14.36 accounts.

[49] As more accounts produce high-fee transactions to attempt this attack, F increases. for big numbers For attackers, if F does not increase proportionally, the expected value of the attack may increase even though the importance boost per account decreases. This is an expected result as the value of the blocks also increases significantly.

Let N be the number of small accounts attempting the attack, P be the probability of a fork resulting in a loss, F be the high fee in a block, and $\bar{F}$ be the average fee in a block. The expected value, EV , can be approximated as follows:

$$\beta = 0.04 \cdot \frac{F}{359 \cdot \bar{F} + N \cdot F} \qquad \text{(importance boost)}$$

$$EV = \beta \cdot \frac{359}{P} \cdot \left(\bar{F} + \frac{(N-1) \cdot F \cdot P}{359} \right) - F \qquad \text{(expected value)}$$

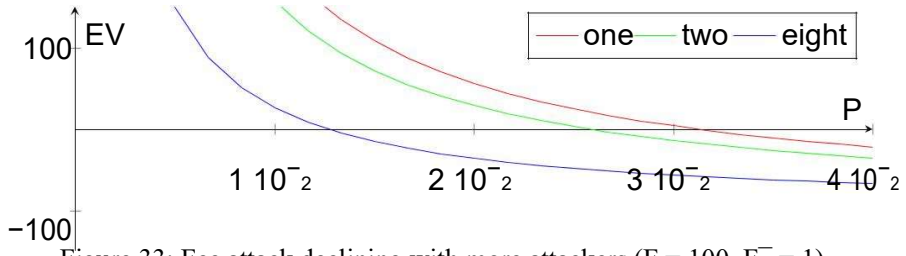


Figure 33: Fee attack declining with more attackers (F = 100, $\bar{F}$ = 1)

15 Ending

The CAP theorem posits that, in the presence of a network failure or partition, a distributed system must choose between consistency or availability.

In a consistent system, requests sent to two nodes will always return the same value. If the value is not globally agreed upon, the request will fail. This ensures that all clients have a uniform view of the system. BFT systems typically choose consistency and risk network positions. A naive consistent system might require all interested parties to vote on each block and only allow the next block when 2 of the interested parties approve. If stakeholders do not vote promptly, there could be a delay in block production.

In an available system, requests sent to two nodes will always return a value immediately. The values returned may be different, so different clients may have different views of the system. PoW and NXT style PoS systems generally choose availability and eventual consensus. They run the risk of propagating different (potentially unsolvable) views of the system. The PoW-like systems available, like Bitcoin, have a simple forking rule that picks the chain with the most work. If the network is divided into partitions with the same hashing power, all partitions will proceed independently without knowing that they are partitioned. When they do eventually reconnect, there will be an expensive (and potentially deep) fork resolution.

Bitxor always prefers availability to consistency, but optionally supports use of a purpose device in addition to its native consensus (see14:Consensus). This device features a BFT-inspired voting system that is orthogonal to block production and block consensus. As a result of its optionality, **Bitxor** supports networks that use deterministic termination (when the device is enabled) or probabilistic termination (when the device is disabled). The gadget is automatically enabled when *node:maxRollbackBlocks* is zero[50].

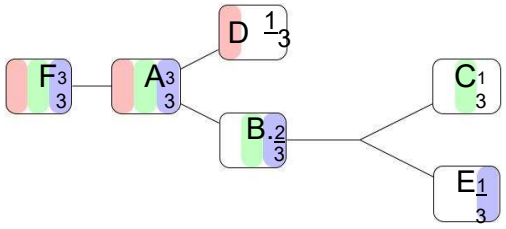
The gadget approach follows the model of GRANDPA [SK20] used by Polkadot. GRANDPA, in turn, is influenced by CASPER [BG17], which in turn is influenced by PBFT [LC99]. Traditionally, PBFT uses three types of messages: prep, setup, and commit. Prep messages, which are used to start rounds, are not used in **Bitxor**. Instead, the elapsed network time (seesixteen:time synchronization) is used to start rounds. The preparation and confirmation messages in PBFT roughly correspond to the pre-voting and pre-confirmation messages in **Bitxor**.

[fifty] The completion extension must also be enabled.

15.1 High level overview

Block completion is a complicated process that involves two types of messages: pre-votes and pre-commits. At the beginning of a round, each voter only knows the blocks stored on the chain local to her. No voter knows the blocks stored on the chains of any other voter. Consequently, it is impossible for a voter to know which block of it will receive a supermajority vote and will be finalized ahead of time.

To illustrate the general termination procedure, consider a network made up of three voters with the same weight. A qualified majority requires that at least two of the three voters vote for the same hash. F refers to the last completed block.



Subsequently, voters will be referenced by their colors in the red, green and blue figure.

Completion is greedy and tries to finish as many blocks as possible each round. In the first stage of prevoting, each voter builds and publishes a hash chain that represents their local chain starting with the hash of the last finished block (F). In this example, the pre-vote strings will be:

1. red: H(F), H(A), H(D)
2. green: H(F), H(A), H(B), H(C)
3. blue: H(F), H(A), H(B), H(E)

Suppose that red, due to bad network connections, only receives the previous vote from green but not from blue. Assume that green and blue receive early votes from all voters. At this point, each voter has the following pre-vote strings:

1. red:
   (a) H(F), H(A), H(D)
   (b) H(F), H(A), H(B), H(C)

2. green:
    (a) H(F), H(A), H(D)
    (b) H(F), H(A), H(B), H(C)
    (c) H(F), H(A), H(B), H(E)
3. blue:
    (a) H(F), H(A), H(D)
    (b) H(F), H(A), H(B), H(C)
    (c) H(F), H(A), H(B), H(E)

Each voter inspects all pre-vote hash chains to calculate the best block that could end up in this round. red only sees a large majority for A, but green and blue see a large majority for B. In the next pre-commit stage, each voter posts the hash of their best computed block:

1. red: H(A)
2. green: H(B)
3. blue: H(B)

Suppose that green, due to bad network connections, only receives the prior acknowledgment from red but not from blue. Assume that red and blue receive prior commitments from all voters. At this point, each voter has the following prior commitments:

1. red: H(A), H(B), H(B)
2. green: H(A), H(B)
3. blue: H(A), H(B), H(B)

Each voter inspects all precommit hashes to calculate the best block that can be finalized. It is important to note that a precommit for one block is also a precommit for all ancestors of the block. green only sees a large majority for A, but red and blue see a large majority for B. In the final commit stage, each voter finalizes the following blocks:

1. red: H(A), H(B)
2. green: H(A)
3. blue: H(A), H(B)

The following sections discuss the completion algorithm in more detail.

15.2 rounds

An end round represents a step in the end process and is made up of an end epoch and an end point.

An epoch is a group of blocks. All blocks within an epoch are terminated with a single voting set, which is recomputed every *network:votingSetGrouping* block. The first epoch is defined as containing exclusively the genesis block and is considered intrinsically finished. Subsequent epochs must end in a block with a height that is a multiple of *network:votingSetGrouping*. An epoch is considered finished when all the blocks within it are finished. There will always be a completion test available for the last block within an epoch.

A point is a fine-grained step within an epoch that represents progress toward the completion of that epoch. Each point can end zero or more blocks. A point can only end blocks within its parent epoch. This prevents any block from potentially being finalized by multiple voting sets, which could lead to a security breach. There is no limit to the number of points associated with an epoch. There will be as many points as it takes to finish all the blocks within an epoch. The last point within an epoch will always end the last block of that epoch. A block is considered finished when a point ends that block or any of its descendants.

When a network is partitioned, it is possible for a point to end up with zero additional blocks. This happens when only the last completed block receives a qualified majority of votes. This is not a fatal error and can occur naturally in the presence of network partitions. Afterwards, the termination procedure will continue and advance to the next point based on the conditions described in15.5:Algorithm.

The important difference between an epoch and a point is their relationship to the voting sets. Different epochs may have different voting sets. All different points within the same epoch must use the same voting set associated with the epoch.

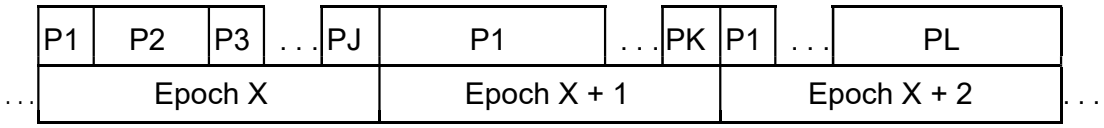


Figure 34: Relationship of epoch and (P)point

15.3 Voters

An account is eligible to vote at one time if all of the following are true:

1. The harvest balance in the last completed block of the previous epoch is not less than the network-defined *network:minVoterBalance*.
2. The voting key is recorded such that Start Epoch ≤ epoch ≤ End Epoch.

A set of votes for an epoch is the set of all accounts that meet the above conditions. It is important to note that only balance, not importance, is used when weighting votes. This prevents a security breach that could occur when using important[51].

All accounts eligible to vote are expected to vote. A well-behaved voter is expected to vote in all rounds for which they are eligible and not submit multiple votes per round. It is implicitly assumed that all voters will be running HA nodes and trigger completion: Enable Voting . Voters who violate these expectations are considered Byzantines and could be punished in the future. Votes are weighted proportionally to balance, so votes from voters with higher balances have more impact.

15.4 Messages

The pre-voting and pre-confirmation messages share a common design.

[51]When there are multiple network partitions, each partition would independently recalculate importances with potentially different activity scores. Since importances are scaled and not absolute, it is theoretically possible for multiple partitions to have supermajorities of importance and end up in conflicting blocks.

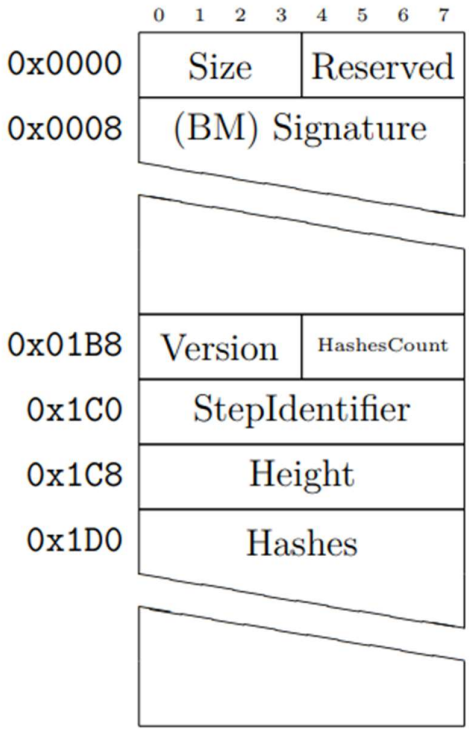


Figure 35: Message layout

Signature is the signature of the BM tree (see3.4:Voting Key List) of the message. The root public key must match the voting key on record for the epoch of the message. The lower public key must match the public key in the tree corresponding to the epoch.

The version is the version of the message.

Hashes contains hashes Hashes Count. A pre-vote message will contain at least one hash and until completion: max Hashes Per Point. A precommit message will always contain exactly one hash.

Step Identifier indicates the completion round of the message. The high bit of the dot is reserved to indicate the type of message where 0 indicates pre-vote and 1 indicates pre-commit.

Height is the block height of the first hash contained in Hashes.

15.5 Algorithm

Define function g(. . .) to select the last block that has cumulative weighted votes of at least a supermajority of available voting weight. Define Vr,v as a prevote at round r by voter v. Define Cr,v as a precommit at round r by voter v. Define Er,v as the estimate by voter v of what might have been finalized in the round r. Notice that this is only an

estimate and a greedy one at that. Er,v must be the latest block on the chain containing g(Vr,v) that can receive a supermajority of Cr,v. A round is completable when either:

- *Er,v < g(Vr,v)*
- *It is impossible for any child of g(Vr,v) to have a supermajority of Cr,v*

A voter v can start a round r when the previous round is compable and he has cast votes in all previous rounds in which he was eligible.

15.5.1 Prevote

when 1∗step Duration has elapsed or is completionable, v submits a previous vote.

A voter determines the best block that can potentially be finalized. Creates a pre-vote message composed of all hashes starting with the hash of the last completed (local) block. The hash string of the pre-vote message will contain at most the completion: hashes max Hashes Per Point. The last hash of the chain will correspond to a block with a height multiple of completion: prevoteBlocksMultiple[52]. This increases the probability that nodes will send pre-vote messages with identical strings that can be aggregated more aggressively.

Hashes for unfinished blocks are prohibited from spanning epochs. This ensures that there is exactly one voting set that can end a block at any height. This property enables dynamic voting sets.

The pre-vote messages include hash strings, rather than individual hashes, because each **Bitxor** node stores a single chain of blocks instead of a tree of blocks. Hash string aggregation allows **Bitxor** to rebuild a virtual block tree and apply votes to visible and non-visible branches.

Conceptually, a voter has one vote per height. Effectively, he votes with his weight on each hash in the pre-vote hash chain.

15.5.2 Pre-committed

Next, a voter waits until g(Vrv) ≥ mir−1,v. when 2∗step Duration (relative to the start of the round) has elapsed or is completionable, v sends a prior confirmation.

[52] In practice, *finalization:maxHashesPerPoint* is expected to be much larger than *finalization:prevoteBlocksMultiple*. Also, *network:votingSetGrouping* must be a multiple of *finalization:prevoteBlocksMultiple*.

A voter determines the best block that can potentially be finalized. Creates a precommit message with a single hash corresponding to g(Vrv).

Conceptually, a voter has one vote per height. Effectively, it votes its weight on each hash between the last completed block and the previous commit hash.

15.5.3 Engage

Asynchronously, a voter collects pre-vote and pre-commit messages for round r. In practice, these messages will be associated with the current round or the previous round. When g(Crv) changes, that block is terminated, as well as all blocks between that block and the local terminated block.

Given a completion round with epoch e and point p, in most cases the next round will be (e, p + 1). The only exception to this is when the last block of epoch e is finished. In that case, the next round is (e + 1, 0). Note that it is possible to initialize both (e, p + 1) and (e + 1, 0). In that scenario, (e + 1, 0) will eventually dominate and (e, p + 1) will not complete.

15.6 Proofs

To minimize network traffic, nonvoters do not send or receive individual prevoting or precommitting messages. Instead, these nodes individually pull and verify completion tests from the network periodically based on *finalization:unfinalizedBlocksDuration*. When this setting is zero, tests are only extracted at the end of each epoch. When this setting is nonzero, tests are optimistically fetched as long as the last completed block is at least *inalization:unfinalizedBlocksDuration* after the last unfinished block. Upon verification, these non-voting nodes end all epochs up to and including the test epoch.

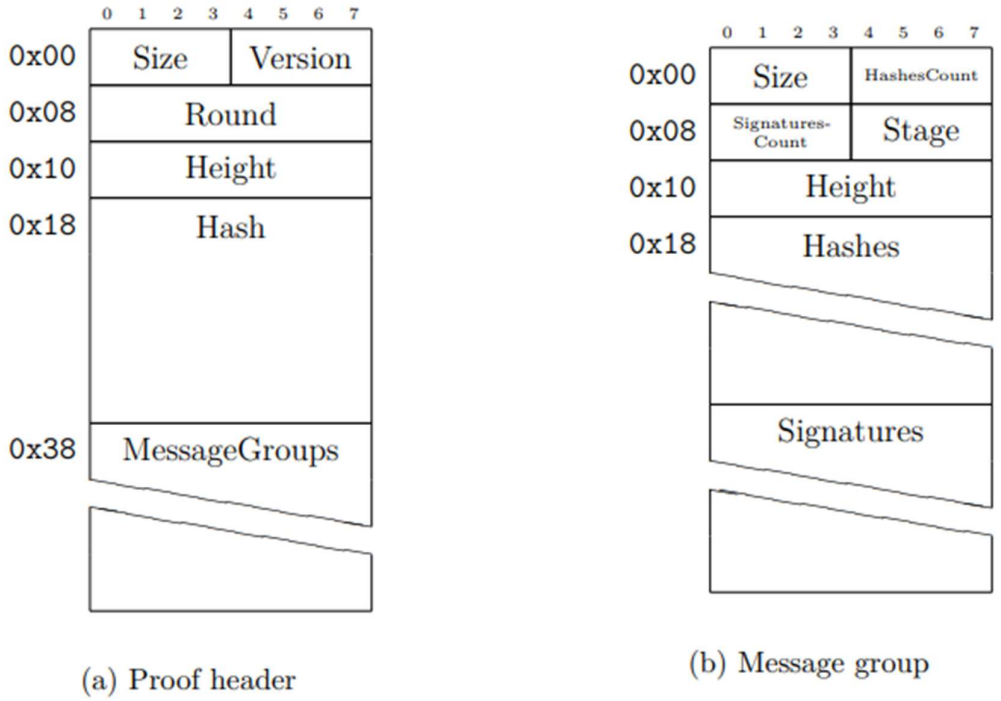


Figure 36: Finalization proof layouts

A completion test is made up of a header and a collection of message groups. The header indicates the last block completed by the test.[53]. This block is uniquely identified by its height and hash. The header also indicates the completion round in which the block was completed.

A message group is an aggregation of completion messages that cryptographically verify the proof. All completion messages that differ only in signature are grouped into a single message group with all signatures attached. There will always be at least two groups of messages in a test, one for pre-voting and one for pre-confirmation, but there may be more due to the way votes are counted.

Verification of a finalization proof requires knowledge of the voting set associated with its epoch. Importantly, the eligible voters and their weights need to be known. In order to be verified, a proof must only contain valid supporting messages from eligible voters. Additionally, the messages must indicate that the block specified in the proof header is g(Cr,v) and g(Vr,v) ≥ g(Cr,v)

[53] Technically, a test terminates the last block and all its ancestors, which are guaranteed to form a single chain.

15.7 Sybil Attack

When deterministic completion is enabled, a Sybil attack among voters is prevented by weighting all votes by account balance. The only way for an attacker to get more voting power is to get more share. Splitting an account balance across multiple accounts will not change the overall voting power.

15.8 Nothing at stake attack

When deterministic completion is enabled, the nothing at stake attack can be avoided if a merchant waits to serve until a block completes. If a merchant does not wait, there are additional defenses against this attack.

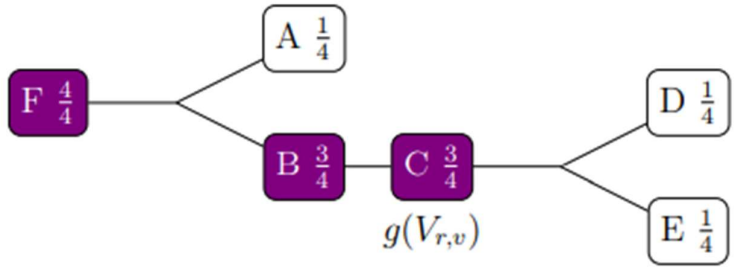
Most significantly, the attacker has a limited amount of time to produce a better chain. The attacker must generate a better chain before the network finalizes the block the attacker wants to undo. Also, the time required for a significant drop in difficulty is probably longer than the time it takes to finish a given part of the chain.

15.9 Examples

In all examples, consider a network with four voters of equal weight. A qualified majority requires that at least three of the four voters vote for the same hash. F always refers to the last completed block. A pre-vote message issued for a block B implies a pre-vote hash chain composed of hashes from F to B, inclusive.

Example 1

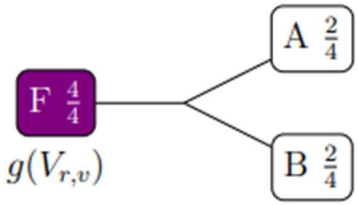
Pre-vote messages are issued for A, C, D, E. Since C is on the branch of D and E, he has a supermajority vote even though only one voter explicitly voted for him



.

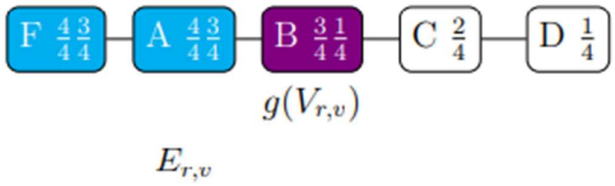
Example 2

Consider a network divided into two partitions of equal size. Pre-vote messages are issued for A, A, B, B. Since F is on the branch of A and B , it has a large majority of votes even though no voter explicitly voted for it. It is important to note that no new blocks are finished because F is already finished.



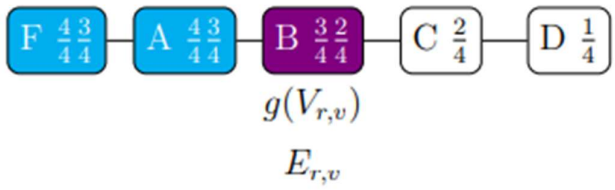
Example 3a

Pre-vote messages are issued for A, B, C, D so that B has a large majority of pre-votes. Suppose two voters see previous votes A, B, C and send pre-commitments for A. Suppose one voter sees previous votes B, C, D and send a previous commitment for B. A voter can determine that the weight of the unknown previous commits (25%) cannot cause a super majority of previous commits for B, which only has 25% previous commits. This satisfies condition 1 on15.5:Algorithm.



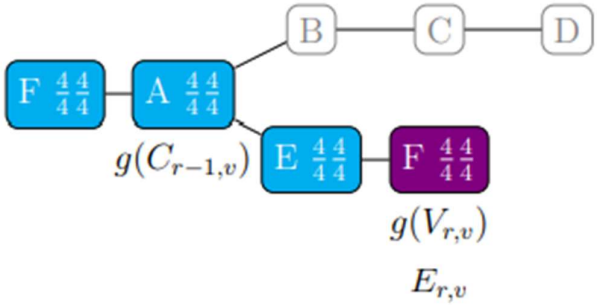
Example 3b

Consider a slight modification of the previous example where the previous votes are received in a different order. As stated above, pre-vote messages are issued to A, B, C, D so that B has a large majority of previous votes. Suppose two voters see previous votes B, C, D and send pre-commitments for B. Suppose one voter sees previous votes A, B, C and send a previous commitment for A. A voter can determine that the weight of the unknown precommits (25%) cannot cause a supermajority of precommits for C, which only has 0% precommits. This satisfies condition 2 in15.5: Algorithm.



Example 4

Consider an extension of the previous example where A is committed at point r − 1 but B is not. Now, the voters have moved on to the next point r. Four pre-vote messages and four pre-confirmation messages are issued for F . Blocks B, C, D have been removed from the main chain, although previous commits for B were used to confirm A at r−1. To check the proof on r − 1, there needs to be a record that B is a descendant of A. This information is stored in the hash strings of the pre-vote message, but not in the pre-commit messages. This is why the verification of the completion tests requires previous votes.



16 time synchronization

Like most other blockchains, Bitxor is based on timestamps for the order of transactions and blocks. Ideally, all nodes on the network should be synchronized with respect to time. Although most operating systems modern have time
Integrated synchronization, nodes can still have local clocks that deviate from real time by more than a minute. This causes those nodes to reject valid transactions or blocks, making it impossible to sync with the network.

Therefore, it is necessary to have a synchronization mechanism to ensure that all nodes agree on time. There are basically two ways to do this:

1. Use an existing protocol, such as Network Time Protocol (NTP)54.
2. Use a custom protocol.

The advantage of using an existing protocol like NTP is that it is easy to implement and the network time will always be close to real. This has the disadvantage that the network depends on servers outside the network.

Using a custom protocol that only relies on the P2P network solves this problem, but there is a tradeoff. It is impossible to guarantee that the network time is always near real time. For an overview of the different custom protocols, Bitxor uses a custom protocol based on Chapter 3 of this thesis to be completely independent of any external entity. The protocol is implemented in the timesync extension.

16.1 Sample Collection

Each node on the network manages an integer offset that is set to 0 at startup. The local system time in milliseconds adjusted by the offset (which can be negative) is the network time (again in milliseconds) of the node.

After the start-up of a node is completed, the node (hereinafter referred to as the local node) selects partner nodes to perform a time synchronization round.

For each selected partner, the local node sends a request asking the partner for its current network time. The local node remembers network timestamps when each

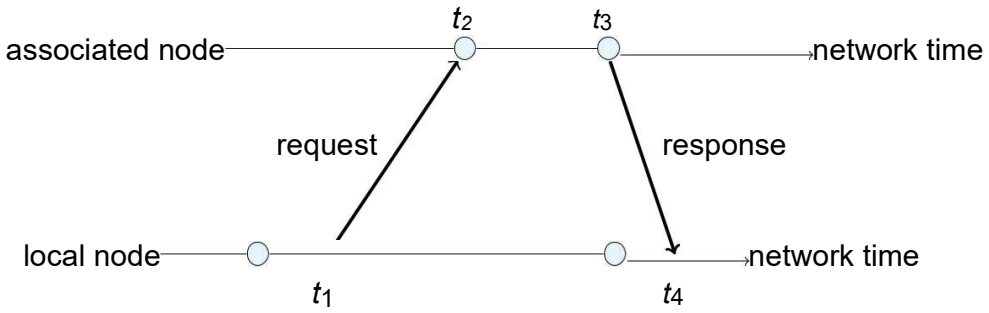


Figure 37: Communication between local and associated node.

the request was sent and when each response was received. Each partner node responds with a token containing the timestamp of the arrival of the request and the timestamp of the response. The partner node uses its own network time to create the timestamps.Figure 37illustrates communication between nodes.

Using the timestamps, the local node can calculate the round trip time

$$rtt = (t_4 - t_1) - (t_3 - t_2)$$

and then estimate the offset o between the network time used by the two nodes as

$$o = t_2 - t_1 - \frac{rtt}{2}$$

This is repeated for each time synchronization partner until the local node has a list of compensation estimates.

16.2 Apply filters to remove bad data

There may be bad samples due to various reasons:

- A malicious node provides incorrect timestamps.
- An honest node has a clock far from real time without knowing it and without having synchronized yet.
- The round trip time is very skewed due to Internet problems or one of the nodes being very busy. This is known as channel asymmetry and cannot be avoided.

Filters are applied that attempt to remove bad samples. Filtering is done in three steps:

1. If a response from a partner is not received within the expected time (ie if $t_4 - t_1 > 1000ms$) the sample is discarded.
2. If the calculated compensation is not within certain limits, the sample is discarded. The allowed limits decrease as a node's uptime increases. When a node first joins the network, it tolerates a high offset to match the existing consensus of network time within the network. As time passes, the node becomes less tolerant of reported tradeoffs. This ensures that malicious nodes reporting large offsets are ignored after some time.
3. The remaining samples are sorted by their offset and then alpha clipped at both ends. In other words, on both sides a certain portion of the samples is discarded.

16.3 Calculation of the Effective Compensation

The reported offset is weighted with the importance of the node reporting the offset. Only nodes that expose a minimum importance are considered as partners in order to avoid solely picking nodes with nearly zero importance. This is done to prevent Sybil attacks.

An attacker that tries to influence the calculated offset by running many nodes with low importances reporting offsets close to the tolerated bound will therefore not have a bigger influence than a single node having the same cumulative importance reporting the same offset. The influence of the attacker will be equal to the influence of the single node on a macro level.

Also, the numbers of samples that are available and the cumulative importance of all partner nodes should be incorporated. Each offset is therefore multiplied with a scaling factor.

Let I_j be the importance of the node reporting the *j*-th offset o_j , n be the number of nodes that were eligible for the last importance calculation and s be the number of samples. Then the scaling factor used is

$$scale = \min\left(\frac{1}{\sum_j I_j}, \frac{1}{\frac{s}{n}}\right)$$

This gives the formula for the effective compensation o

$$o = scale \sum_j I_j \, o_j$$

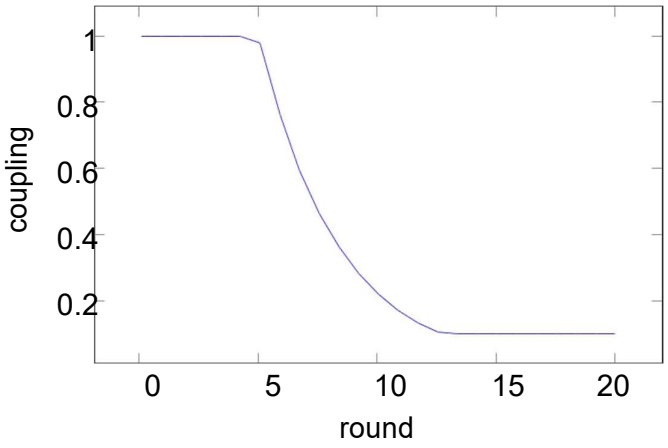


Figure 38: Coupling factor

Note that the influence of an account with great importance is artificially limited because the north term caps the scale. Such an account can increase your influence on a macro level by dividing your balance into accounts that do not have a cap. However, doing so will likely decrease your influence on individual partners because the probability of all your split accounts being chosen as time synchronization partners for a single node is low.

16.4 Coupling and Threshold

New nodes that have just joined the network need to quickly adjust their offset to the already established network time. On the contrary, old nodes should behave much more rigid so as not to be too influenced by malicious nodes or newcomers.

To enable this, nodes only adjust a portion of the reported effective compensation.
The nodes multiply the effective displacement with a coupling factor to generate the final displacement.

Each node keeps track of the number of time synchronization rounds it has performed.
This is called the age of the node.

The formula for this coupling factor c is:

$$c = \max\left(\mathrm{e}^{-0.3a},\ 0.1\right) \text{ where } a = \max(nodeAge - 5,\ 0)$$

This ensures that the coupling factor will be 1 for 5 rounds and then decay exponentially to 0.1.

Finally, a node only adds any final computed offset to its internal offset if the absolute value is above a certain threshold (currently set to 75ms). This is effective in preventing

Slow network time drifts due to communication between nodes that have channel asymmetry.

17 Messenger service

blockchain client applications retrieve data from blockchain and present them to their users To make these clients more useful, always they must present the most up-to-date blockchain data and update their communication interfaces.
user when needed the displayed data changes. A naive customer might periodically poll a REST server or local database for data from blockchain. This is inefficient because it requires using more network bandwidth and other resources than necessary. Instead, **Bitxor** allows customers to subscribe to data changes through a single message queue.

17.1 Message channels and topics

The message queue of **Bitxor** exposed to customers supports multiple channels. Each channel has a unique theme. A topic always begins with a topic marker indicating the type of messages to be received. In some cases, the marker is followed by an unresolved address that is used for additional filtering. Since a client is generally not interested in all types of blockchain state changes, they can subscribe to a subset of available topics. Figure 39 lists all supported theme markers.

Subject marker name	subject marker
Block	0x9FF2D8E480CA6A49
drop blocks	0x5C20D68AEE25B0B0
completed block	0x4D4832A031CE7954
Transaction	0x61
Add unconfirmed transaction	0x75
Delete unconfirmed transaction	0x72
Add partial transaction	0x70
Delete partial transaction	0x71
transaction status	0x73
joint signature	0x63

Figure 39: Topic Markers

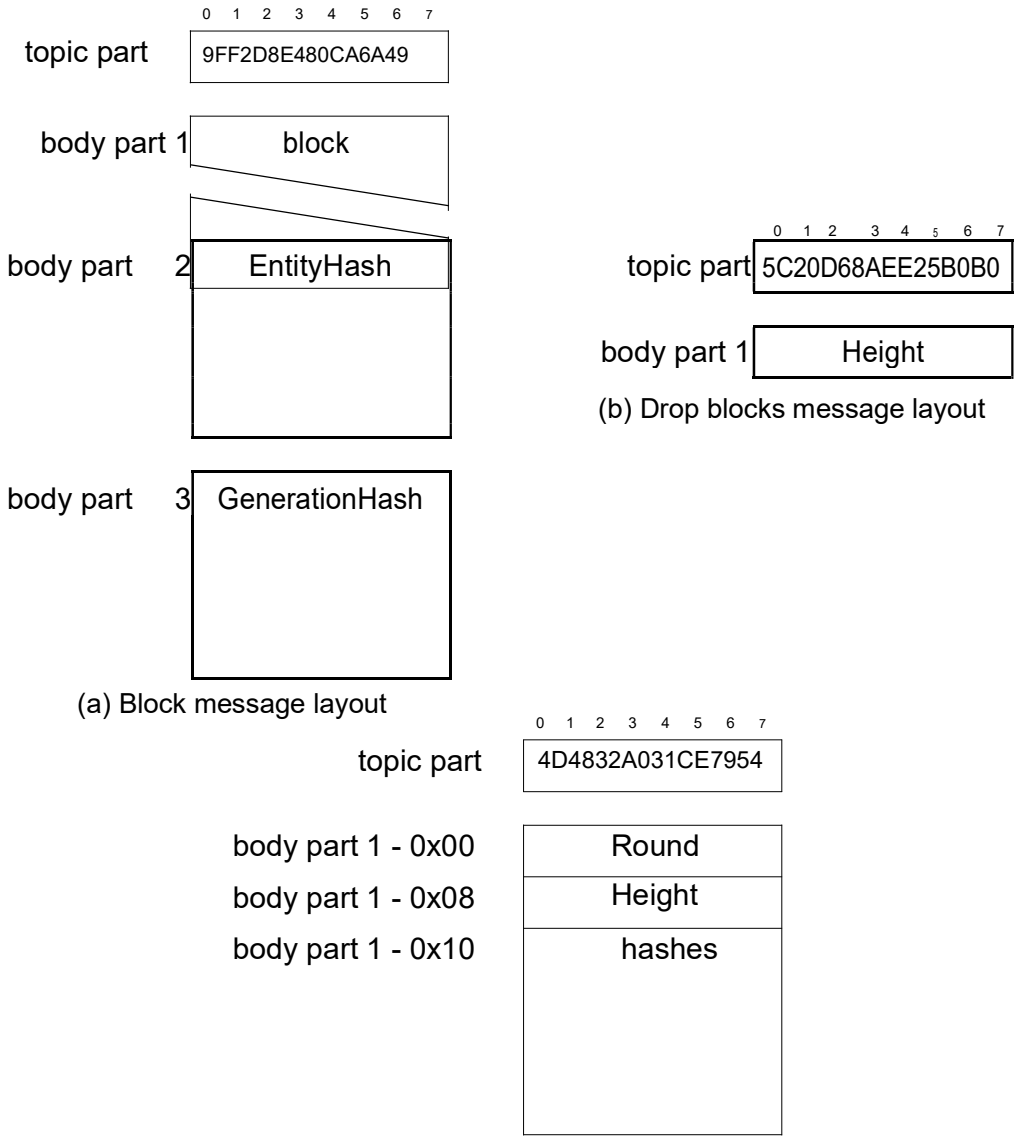
17.2 Connection and Subscriptions

The zeromq extension adds support for messaging. If a node wants to support messaging, this extension must be enabled in the broker process. The extension registers subscribers for events related to blocks and transactions (see2.2:**Bitxor** Extensions) and maps those events to messages in the message queue. When enabled, the agent listens on port messaging: subscriber port for new subscribers. Clients can connect to and subscribe to the message queue for one or more topics.

17.3 block messages

Topics for locked messages only consist of a topic marker. Layouts for all blocking messages are shown inFigure 40. The following block messages are supported:

- Block: A new block has been added to the chain.
- Drop Blocks: Blocks were dropped after a certain height.
- Block completed: The block has been completed. Posted only when deterministic completion is enabled.



(a) Block message layout

(b) Drop blocks message layout

(c) Finalized block message layout

Figure 40: Block-related messages

17.4 transaction messages

Transaction message topics consist of a topic marker and an optional unresolved address filter. When an unresolved address filter is provided, only messages involving the specified unresolved address will be generated. For example, a message will be generated for a transfer transaction only if the specified unresolved address is the sender or recipient of the transfer. When no unresolved address filter is provided, messages will be generated for all transactions. Layouts for all transaction messages are shown inFigure 41, Figure 42Y Figure 43. The following transaction messages are supported:

- Transaction: A transaction was confirmed, that is, it is part of a block.
- Add uncommitted transaction – An uncommitted transaction has been added to the uncommitted transaction cache.
- Uncommitted Transaction Delete – An uncommitted transaction was deleted from the uncommitted transaction cache.
- Add Partial Transaction – A partial transaction has been added to the partial transaction cache.
- Delete Partial Transactions: A partial transaction was deleted from the partial transaction cache.
- Transaction Status: The status of a transaction has changed.

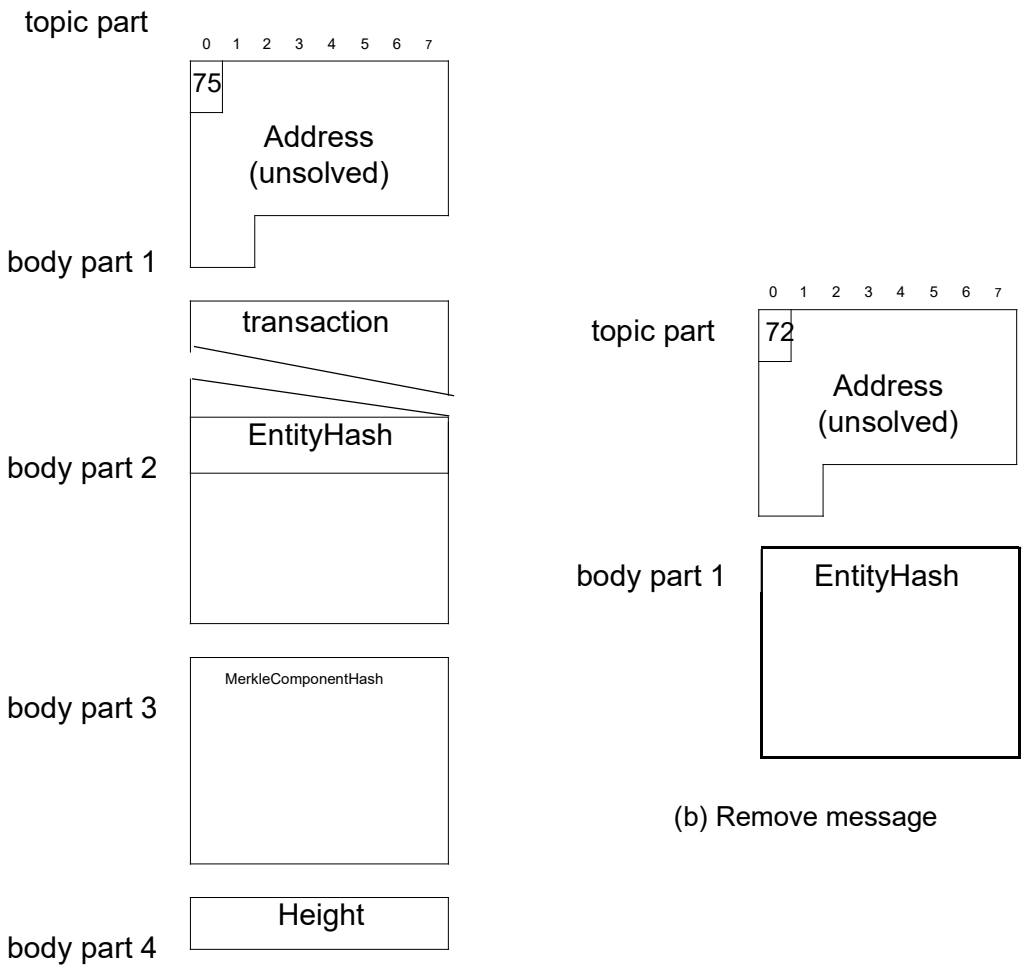
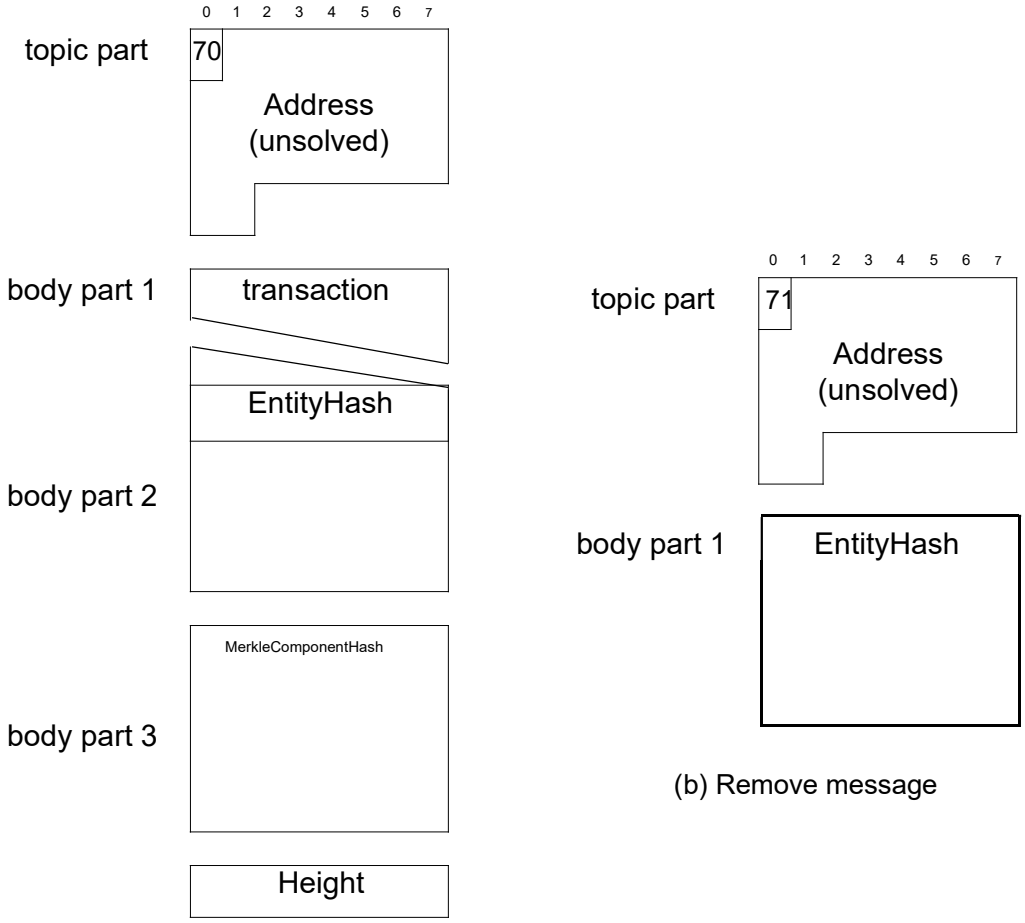


Figure 41: Unconfirmed transactions related messages



(a) Add message

(b) Remove message

Figure 42: Partial transactions related messages

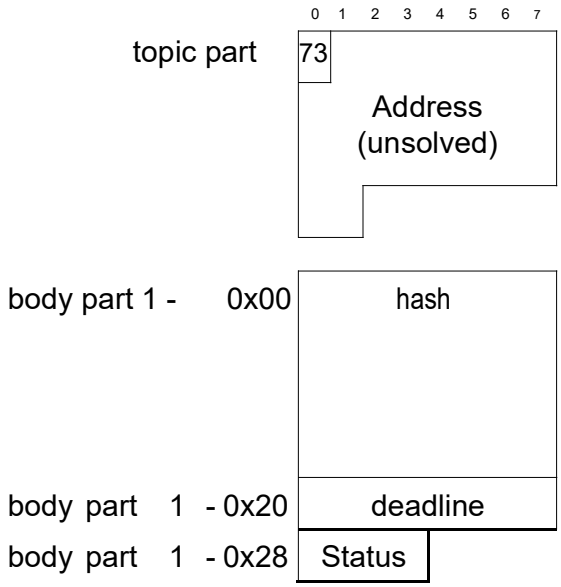


Figure 43: Transaction status message

17.4.1 Joint signature message

The subject of a co-signed message consists of a subject marker and an optional unresolved address filter. The message is issued to subscribed clients when a new co-signature is added for a transaction added to the partial transaction cache. When an unresolved address filter is provided, messages will only be generated for aggregated transactions involving the specified address. Otherwise, messages will be generated for all changes. The joint signature message design is shown inFigure 44.

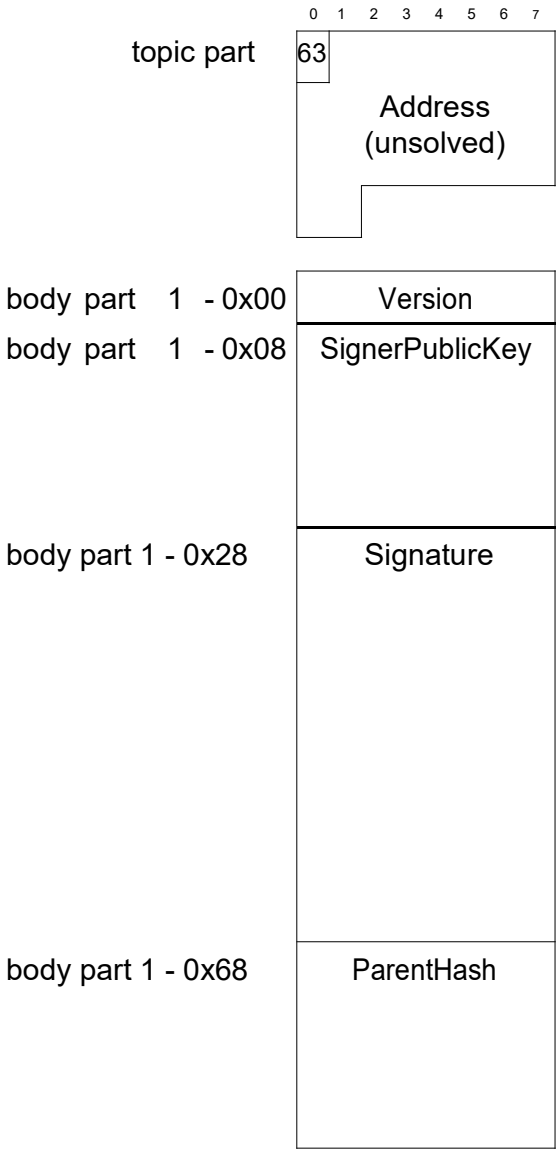


Figure 44: Cosignature message

18 Control-Stake

Decentralized Control

mechanism whose objective is to protect the Bitxor Community from attack and manipulation by large financial groups.

Process that is carried out through a control committee established by the Super Voting Nodes

17.1 Technical Information

The Control-Stake is configured in the configuration of the core of Bitxor, this must set a percentage in *network::harvestControlStakePercentage*, this is dedicated for Control-Stake link to reward harvest system.

Welcome to
Bitxor Blockchain