

Symbol from NEM

Technical Reference

Version 0.10.0.4

CRYP2NOVA

July 29, 2021

Archived copy · cryp2nova.com

Contents

Preface vi

Typographical Conventions vii

1 Introduction 1
- 1.1 Network Fingerprint . . . 2

2 System 3
- 2.1 Transaction Plugins . . . 3
- 2.2 **Symbol** Extensions . . . 4
- 2.3 Server . . . 5
  - 2.3.1 Cache Database . . . 7
- 2.4 Broker . . . 7
- 2.5 Recovery . . . 8
- 2.6 Common Topologies . . . 9

3 Cryptography 10
- 3.1 Public/Private Key Pair . . . 10
- 3.2 Signing and Verification . . . 11
  - 3.2.1 Batch Verification . . . 11
- 3.3 Verifiable Random Function (VRF) . . . 12
- 3.4 Voting Key List . . . 13
  - 3.4.1 Signature . . . 14

4 Trees 15
- 4.1 Merkle Tree . . . 15
- 4.2 Patricia Tree . . . 16
- 4.3 Merkle Patricia Tree . . . 18
- 4.4 Merkle Patricia Tree Proofs . . . 19

5 Accounts and Addresses 21
- 5.1 Addresses . . . 21
  - 5.1.1 Address Derivation . . . 22

CRYP2NOVA

Archived copy · cryp2nova.com

5.1.2 Address Aliases . . . 24
5.1.3 Intentional Address Collision . . . 24
5.2 Public Keys . . . 24

6 Transactions **26**
6.1 Basic Transaction . . . 26
6.2 Aggregate Transaction . . . 28
6.2.1 Embedded Transaction . . . 29
6.2.2 Cosignature . . . 30
6.2.3 Extended Layout . . . 32
6.3 Transaction Hashes . . . 32

7 Blocks **34**
7.1 Block Fields . . . 34
7.1.1 Importance Block Fields . . . 36
7.2 Receipts . . . 37
7.2.1 Receipt Source . . . 38
7.2.2 Transaction Statement . . . 38
7.2.3 Resolution Statements . . . 39
7.2.4 Receipts Hash . . . 39
7.3 State Hash . . . 40
7.4 Extended Layout . . . 41
7.5 Block Hashes . . . 41

8 Blockchain **43**
8.1 Block Difficulty . . . 43
8.2 Block Score . . . 44
8.3 Block Generation . . . 44
8.4 Block Generation Hash . . . 47
8.5 Block Hit and Target . . . 47
8.6 Automatic Delegated Harvester Detection . . . 49
8.7 Blockchain Synchronization . . . 50
8.8 Blockchain Processing . . . 53

9 Disruptor **55**

CRYP2NOVA

Archived copy · cryp2nova.com

9.1 Consumers . . . 56
9.1.1 Common Consumers . . . 57
9.1.2 Additional Block Consumers . . . 58
9.1.3 Additional Transaction Consumers . . . 60

10 Unconfirmed Transactions **61**
10.1 Unconfirmed Transactions Cache . . . 61
10.2 Spam Throttle . . . 62

11 Partial Transactions **64**
11.1 Partial Transaction Processing . . . 65

12 Network **68**
12.1 Beacon Nodes . . . 68
12.2 Connection Handshake . . . 69
12.3 Packets . . . 70
12.4 Connection Types . . . 70
12.5 Peer Provenance . . . 72
12.6 Node Discovery . . . 73

13 Reputation **74**
13.1 Connection Management . . . 74
13.2 Weight Based Node Selection . . . 75
13.3 Node Banning . . . 76

14 Consensus **79**
14.1 Weighting Algorithm . . . 80
14.2 Sybil Attack . . . 82
14.3 Nothing at Stake Attack . . . 84
14.4 Fee Attack . . . 85

15 Finalization **90**
15.1 High Level Overview . . . 91
15.2 Rounds . . . 93
15.3 Voters . . . 93
15.4 Messages . . . 94

CRYP2NOVA

Archived copy · cryp2nova.com

15.5 Algorithm . . . 95
15.5.1 Prevote . . . 96
15.5.2 Precommit . . . 96
15.5.3 Commit . . . 97
15.6 Proofs . . . 97
15.7 Sybil Attack . . . 99
15.8 Nothing at Stake Attack . . . 99
15.9 Examples . . . 99

16 Time Synchronization **102**
16.1 Gathering Samples . . . 102
16.2 Applying Filters to Remove Bad Data . . . 103
16.3 Calculation of the Effective Offset . . . 104
16.4 Coupling and Threshold . . . 105

17 Messaging **107**
17.1 Message Channels and Topics . . . 107
17.2 Connection and Subscriptions . . . 108
17.3 Block Messages . . . 108
17.4 Transaction Messages . . . 110
17.4.1 Cosignature Message . . . 113

References **115**

Index **116**

CRYP2NOVA

Archived copy · cryp2nova.com

Preface

> You miss 100% of the shots you don't take.
>
> - *Wayne Gretzky*

NEM had its humble beginnings as a "call for participation" on a bitcointalk thread in January 2014. The cryptospace had just experienced a boom at the tail end of 2013 - although nowhere near where it would go a few years later - and there was a lot of enthusiasm in the space. NXT had just launched as one of the first PoS blockchains, and much of the early NEM community was inspired by and had connections to the NXT community. This includes all three of the remaining core developers.

Although there was some initial discussion on what to build, we quickly decided to create something new from scratch. This allowed for more design flexibility as well as the use of high coding standards from the very beginning. This also gave us the opportunity to contribute something new to the blockchain landscape. As a result of lots of effort - mostly nights and weekends - this culminated in the release of NIS1 mainnet in March 2015. We were pleased with what we built, but knew we took a few shortcuts, and continued improving it. Eventually, we came to the realization that the original solution would need a rearchitecture to fix some central performance bottlenecks and allow faster innovation in the future.

We are grateful to Tech Bureau who provided support for us to build a completely new chain from scratch - **Symbol**. We are hopeful that this fixes many of the problems inherent in NIS1 and provides a solid foundation for future improvements and enhancements. Our mandate was to build a high performance *blockchain* with optional support for deterministic finalization. In this, we think, we succeeded.

This has been a long journey for us, but we are excited to see what yet is to come and what novel things **you** use **Symbol** to build. We would like to once again thank the contributors and the many people who have inspired us...

BloodyRookie gimre Jaguar0625

Archived copy · cryp2nova.com

Typographical Conventions

Description	Attributes	Example
Project name	bold, colored	**Symbol**
Classes	fixed-width font	`ProcessBootstrapper`
Fields	fixed-width font	`FeeMultiplier`
File paths	fixed-width font	`commit_step.dat`
Configuration settings (prefixed with configuration filename) in text mode	italic	*network:maxDifficultyBlocks*
Configuration settings (without configuration filename) in equations and formulas	regular	maxDifficultyBlocks
Fields in equations and formulas	regular	T::SignerPublicKey
Function names in equations and formulas	regular	VerifiableDataBuffer()

Table 1: Typographical conventions used in this document

CRYP2NOVA

Archived copy · cryp2nova.com

1 Introduction

> From the ashes a fire shall be woken,
> A light from the shadows shall spring;
> Renewed shall be blade that was broken,
> The crownless again shall be king.
>
> - *J.R.R. Tolkien*

TRUSTLESS, high-performance, layered-architecture, blockchain-based DLT protocol - these are the first principles that influenced the development of **Symbol**. While a handful of DLT protocols were considered, a blockchain protocol was quickly chosen because it was deemed most true to the ideal of trustlessness. Any node can download a complete copy of the chain and can independently verify it at all times. Nodes with sufficient harvesting power can always create blocks and never need to rely on a leader. These choices necessarily sacrifice some throughput relative to other protocols, but they seem most consistent with the philosophical underpinnings of Bitcoin[Nak09].

Symbol supports both probabilistic and deterministic block finalization. Under probabilistic finalization, the probability that any particular block is rolled back decreases as more and more blocks are added - or chained - to it. Although the probability might become vanishingly small, it is always nonzero[1]. In contrast, deterministic finalization includes a mechanism in the protocol that allows checkpoints to be set that can never be rolled back. This can lead to potentially deep rollbacks but provides stronger assurances. In either case, the risk of block rollback and transaction invalidation is assumed by users.

As part of a focus on trustlessness, the following features were added relative to NEM:

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

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

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

Archived copy · cryp2nova.com

three primary configurations (per network), but further customized hybrid configurations are possible by enabling or disabling specific extensions.

The three primary 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 easier querying and can be used in combination with a NodeJS REST server.
3. Dual: These nodes perform the functions of both 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 real needs, should lead to a more globally resource-optimized network.

Basing the core block and transaction pipelines on the disruptor pattern - and using parallel processing wherever possible - allows high rates of transactions per second relative to a typical blockchain protocol.

NIS1 was a worthy entry into the blockchain landscape and **Symbol** is an even worthier evolution of it. This is not the end but a new beginning. There is more work to be done.

1.1 Network Fingerprint

Each network has a unique *fingerprint* that is composed of the following:

1. Network Identifier - One byte identifier that can be shared across networks. All addresses compatible with a network must have this value as their first byte.
2. Generation Hash Seed - Random 32 byte value that must be globally unique across networks. This value is prepended to transaction data prior to hashing or signing in order to prevent cross network replay attacks.

Archived copy · cryp2nova.com

2 System

> A new technology can transform society, but when the technology is in its infancy, very few people can see its full potential.
>
> - *Liu Cixin*

Symbol supports high customization at both the network and individual node levels. Network-wide settings, specified in *network*, must be the same for all nodes in a network. In contrast, node-specific settings can vary across all nodes in the same network, and are located in *node*.

Symbol was designed to use a *plugin / extension* approach instead of supporting Turing complete smart contracts. While the latter can allow for more user flexibility, it's also more error prone from the user perspective. A plugin model limits the operations that can be performed on a blockchain and consequently has a smaller attack surface. Additionally, it's much simplier to optimize the performance of a discrete set of operations than an infinite set. This helps **Symbol** achieve the high throughput 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 manner so that all nodes can agree on the global blockchain state. The network requires each node to load a set of *transaction plugins*, and this set determines the kinds of transactions the network supports. This set is determined by the presence of *network:plugin** sections in the *network* configuration. Any changes, additions or deletions of these plugins must be coordinated and accepted by all network nodes. If only a subset of nodes agree to these modifications, those nodes will be on a fork. All built-in **Symbol** transactions are built using this plugin model in order to validate its extensibility.

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

```
extern "C" PLUGIN_API
void RegisterSubsystem(
        catapult::plugins::PluginManager& manager);
```

`PluginManager` provides access to the subset of configuration that plugins need to initialize themselves. Through this class, a plugin can register zero or more of the

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

Archived copy · cryp2nova.com

following:

1. Transactions - New transaction types and the mapping of those types to parsing rules can be specified. Specifically, the plugin defines rules for translating a transaction into component notifications that are used in further processing. A handful of processing constraints can also be specified, such as indicating a transaction can only appear within an aggregate transaction (see 6.2: Aggregate Transaction).

2. Caches - New cache types and rules for serializing and deserializing model types to and from binary can be specified. Each state-related cache can optionally be included in the calculation of a block's `StateHash` (see 7.3: State Hash) when that feature is enabled.

3. Handlers - These are APIs that are always accessible.

4. Diagnostics - These are APIs and counters that are accessible only when the node is running in diagnostic mode.

5. Validators - Stateless and stateful validators process the notifications produced by block and transaction processing. The registered validators can subscribe to general or plugin-defined notifications and reject disallowed values or state changes.

6. Observers - Observers process the notifications produced by block and transaction processing. The registered observers can subscribe to general or plugin-defined notifications and update blockchain state based on their values. Observers don't 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 Symbol Extensions

Individual nodes within a network are allowed to support a heterogeneous mix of capabilities. For example, some nodes might want to store data in a database or publish events to a message queue. These capabilities are all optional because none of them impact consensus. Such capabilities are determined by the set of *extensions* a node loads as specified in *extensions-{process}:extensions*. Most built-in **Symbol** functionality is built using this extension model in order to validate its extensibility.

Archived copy · cryp2nova.com

An extension is a separate dynamically linked library that exposes a single entry point in the following form [3]:

```
extern "C" PLUGIN_API
void RegisterExtension(
        catapult::extensions::ProcessBootstrapper& bootstrapper
```

`ProcessBootstrapper` provides access to full **Symbol** configuration and services that extensions need to initialize themselves. 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 representing the executable's state 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. It can also create dependent services and tie their lifetimes to that of the hosting executable. There are very few limitations on what a service can do, which allows the potential for significant customizations.
2. Subscriptions - An extension can subscribe to any supported blockchain event. Events are raised when changes are detected. Block, state, unconfirmed transaction and partial transaction change events are supported. Transaction status events are raised when the processing of a transaction completes. Node events are raised when remote nodes are discovered.

In addition to the above, extensions can configure the node in more intricate ways. For example, an extension can register a custom network time supplier. In fact, there is a specialized extension that sets a time supplier based on algorithms described in 16: Time Synchronization. This is an example of the high levels of customization allowed by this extension model. For understanding the full range of extensibility allowed by extensions, please refer to the project code or developer documentation[4].

2.3 Server

The simplest **Symbol** topology is composed of a single server executable. Transaction Plugins required by the network and **Symbol** Extensions desired by the node operator are loaded and initialized by the server.

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

[4] See https://nemtech.github.io/ for details.

Archived copy · cryp2nova.com

Symbol stores all of its data in a `data` directory. The contents of the data directory are as follows:

1. Block Versioned Directories - These directories contain block information in a proprietary format. Each confirmed block's binary data, transactions and associated data are stored in these directories in files with `.dat` and `.stmt` extensions. The statements (see 7.2: Receipts) generated when 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 block hashes.

   When deterministic finalization is enabled, these directories will also contain versioned `.proof` files that contain proof information for each finalization epoch.

2. `audit` - Audit files created by the audit consumer (see 9.1.1: 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 importance recalculation point. These files enable deep rollbacks that require undoing multiple importance calculations. This directory is only created when deterministic finalization 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 typically prefixed with the name of the source process.

5. `spool` - Subscription notifications are written out 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 case of an ungraceful termination.

6. `state` - **Symbol** stores its proprietary storage files in this directory. `supplemental.dat` and files ending with `_summary.dat` store summarized data. Files ending in `Cache.dat` store complete cache data.

7. `statedb` - When *node:enableCacheDatabaseStorage* is set, this directory will contain RocksDB files.

8. `transfer_message` - When *user:enableDelegatedHarvestersAutoDetection* is set, this directory will contain extracted delegated harvesting requests for the current node.

9. `commit_step.dat` - This stores the most recent step of the commit process initiated by the server. This is primarily used for recovery purposes.

Archived copy · cryp2nova.com

10. `index.dat` - This is a counter that contains the number of blocks stored on disk.
11. `proof.index.dat` - This is a counter that contains the number of finalization proofs stored on disk.
12. `voting_status.dat` - This stores information about the last finalization 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 state (see 7.3: State Hash) requires a cache database and most network configurations are expected to have it enabled.

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

1. High rate of transactions per second is desired.
2. Trustless verification of cache state is not important.
3. Servers are configured with a large amount of RAM.

In this mode, all cache entries are always resident in memory. On shutdown, cache data is written to disk across multiple flat files. On startup, this data is read and used to populate the memory caches.

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

2.4 Broker

The broker process allows more complex **Symbol** behaviors to be added without sacrificing parallelization. Transaction Plugins required by the network and **Symbol** Extensions desired by the node operator are loaded and initialized by the broker. Although the broker supports all features of Transaction Plugins, it only supports a subset of **Symbol** Extensions features. For example, overriding the network time supplier in the broker is not supported. Broker extensions are primarily intended to register subscribers and react to events forwarded to those subscribers. Accordingly, it's expected that the server and

Archived copy · cryp2nova.com

broker have different extensions loaded. Please refer to 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 blockchain state.

`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 raises subscription events in the blockchain sync consumer (see 9.1.2: Additional Block Consumers) when it holds an exclusive lock to the blockchain data. These operations are offloaded to the broker to prevent slow database operations when the server has an exclusive lock. The server overhead is minimal because most of the data used by the broker is also required to recover data after an ungraceful server termination.

2.5 Recovery

The recovery process is used to repair the global blockchain state after an ungraceful server and/or broker termination. Transaction Plugins required by the network and **Symbol** 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 process configuration and the value of the `commit_step.dat` file. Generally, if the server exited after state changes were flushed to disk, those changes will be reapplied. The blockchain state will be the same as if the server had applied and committed those changes. Otherwise, if the server exited before state changes were flushed to disk, pending changes will be discarded. The blockchain state will be the same as if the server had never attempted to process those changes.

After the recovery process completes, the blockchain state should be indistinguishable from the state of a node that never terminated ungracefully. `spool` directories are repaired and processed. Block and cache data stored on disk are reconciled and updated. Pending state changes, if applicable, are applied. Other files indicating the presence of an ungraceful termination are updated or removed.

Archived copy · cryp2nova.com

2.6 Common Topologies

Although a network can be composed of a large number of heterogeneous topologies, it is likely that most nodes will fall into one of three categories: Peer, API or Dual. The same server process is used across all of these topologies. The only difference is in what extensions each loads.

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

API nodes are more heavyweight nodes. They can synchronize with other nodes, but cannot harvest new blocks. They support hosting bonded aggregate transactions and collecting cosignatures to complete them. They require a broker process, which is configured to write data into a MongoDB database and propagate changes over public message queues to subscribers. The REST API is dependent on both of these capabilities and is typically co-located with an API node for performance reasons in order to minimize latency.

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

CRYP2NOVA

Archived copy · cryp2nova.com

3 Cryptography

> I understood the importance in principle of public key cryptography but it's all moved much faster than I expected. I did not expect it to be a mainstay of advanced communications technology.
>
> - *Whitfield Diffie*

BLOCKCHAIN technology demands the use of some cryptographic concepts. **Symbol** uses cryptography based on Elliptic Curve Cryptography (ECC). The choice of the underlying curve is important in order to guarantee security and speed.

Symbol uses the Ed25519 digital signature algorithm. This 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 $q = 2^{252}+27742317777372353535851937790883648493$ elements. It was developed by D. J. Bernstein et al. and is one of the safest and fastest digital signature algorithms [Ber+11].

Importantly for **Symbol** purposes, 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 unimportant.

3.1 Public/Private Key Pair

A *private key* is a random 256-bit integer k. To derive the public key*public key* $\underline{\mathrm{A}}$ from it, the following steps are taken:

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

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

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

Since A is a group element, it can be encoded into a 256-bit integer $\underline{\mathrm{A}}$, which serves as the public key.

Archived copy · cryp2nova.com

3.2 Signing and Verification

Given a message M, private key k and its associated public key $\underline{A}$, the following steps are taken to create a signature:

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

Then $(\underline{R}, \underline{S})$ is the *signature* for the message M under the private key k. Note that only signatures where $S < q$ and $S > 0$ are considered as valid **to prevent** the problem of *signature malleability*.

To verify the signature $(\underline{R}, \underline{S})$ for the given message M and public key $\underline{A}$, the verifier checks $S < q$ and $S > 0$ and then calculates

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

and verifies that

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

If S was computed as shown in (7) then

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

so (8) will hold.

3.2.1 Batch Verification

When lots of signatures have to be verified, a batch signature verification can speed up the process by about 80%. **Symbol** uses the algorithm outlined in [Ber+11]. Given a batch of (M_i, A_i, R_i, S_i) where (R_i, S_i) is the signature for the message M_i with public key A_i, let $H_i = \mathrm{H}(R_i, A_i, M_i)$. Additionally, assume a corresponding number of uniform distributed 128-bit independent random integers z_i 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 $P_i = 8R_i + 8H_i A_i - 8S_i B$, then if (9) holds, it implies

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

Archived copy · cryp2nova.com

All P_i are elements of a cyclic group (remember q is a prime). If some P_i is not zero, for example P_2, it means that for given integers $z_0, z_1, z_3, z_4 \ldots$, there is exactly one choice for z_2 to satisfy (10). The chance for that is 2^{-128}. Therefore, if (9) holds, it is a near certainty that $P_i = 0$ for all i. This implies that the signatures are valid.

If (9) does not hold, it means that there is at least one invalid signature. In that case, **Symbol** falls back to single signature verification to identify the invalid signatures.

3.3 Verifiable Random Function (VRF)

A verifiable random function (*VRF*) uses a public/private key pair to generate pseudo-random 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. **Symbol** uses the ECVRF-EDWARDS25519-SHA512-TAI defined in [Gol+20].

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

$$\begin{aligned} H &= \text{map_to_group_element}(alpha, Y) \\ \gamma &= xH \\ k &= \text{generate_nonce}(H) \\ c &= \text{IetfHash}(3, 2, H, \gamma, kB, kH)[0..15] \\ s &= (k + cx) \mod q \\ proof &= (\gamma, c, s) \end{aligned}$$

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

$$\begin{aligned} H &= \text{map_to_group_element}(alpha, Y) \\ U &= sB - cY \\ V &= sH - c\gamma \\ verification_hash &= \text{IetfHash}(3, 2, H, \gamma, U, V)[0..15] \end{aligned}$$

When the calculated *verification_hash* matches the c part of the proof, the verification of the random value is successful.

[5] This is typically called proving, not to be confused with verifying, because the private key owner needs to prove that it generated the random value with its private key.

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

Archived copy · cryp2nova.com

A proof hash, also called a VRF hash output, can be derived from γ of a *validated* proof:

$$proof_hash = \text{IetfHash}(3, 3, 8\gamma) \tag{11}$$

3.4 Voting Key List

Finalization voters are required to specify the range of finalization epochs in which a root voting key is eligible to vote. Prior to announcing a root voting key, voters are required to build a voting key list that contains epoch-pinned voting keys. The list construction is roughly based on a simplified Bellare-Miner [BM99] construction. Each key in the list is bound to a single epoch and is wiped when the finalization process advances to a subsequent epoch. This provides some forward secrecy. Even if an attacker obtains a voting key list, completed epochs cannot be modified or repudiated.

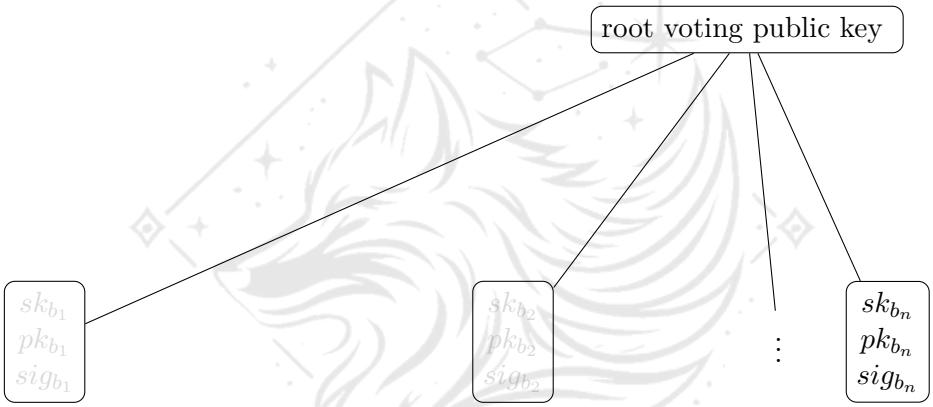


Figure 1: Voting key list

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

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

$$sig_{b_i} = \text{Sign}_{root\ secret\ key}(pk_{b_i} || \text{IntToBin}(i))$$

When signing a voting message for a given epoch, a message signature is created:

$$sig_{message\text{-}i} = \text{Sign}_{sk_{b_i}}(message)$$

Archived copy · cryp2nova.com

3.4.1 Signature

The signature for a vote in a given epoch is composed of two pairs:

- $(\textit{root voting public key}, sig_{b_i})$
- $(pk_{b_i}, sig_{\textit{message-i}})$

A *voting key list signature* is considered verified when:

- *root voting public key* is registered to participate in the given epoch.
- *message* signer key matches epoch-bound key.
- All component signatures are cryptographically verified.

CRYP2NOVA

Archived copy · cryp2nova.com

4 Trees

> “All our wisdom is stored in the trees.”
>
> *- Santosh Kalwar*

Symbol uses tree structures in order to support trustless light clients. Merkle trees allow a client to cryptographically confirm the existence of data stored within them. Patricia trees allow a client to cryptographically confirm the existence or non-existence of data stored within them.

4.1 Merkle Tree

A Merkle tree[Mer88] is a tree of hashes that allows efficient existence proofs. Within **Symbol**, all basic merkle trees are constrained to being balanced and binary. Each leaf node contains a hash of some data. Each non-leaf node is constructed by hashing the hashes stored in child nodes. In the **Symbol** implementation, when any (non-root) layer contains an odd number of hashes, the last hash is doubled when calculating the parent hash.

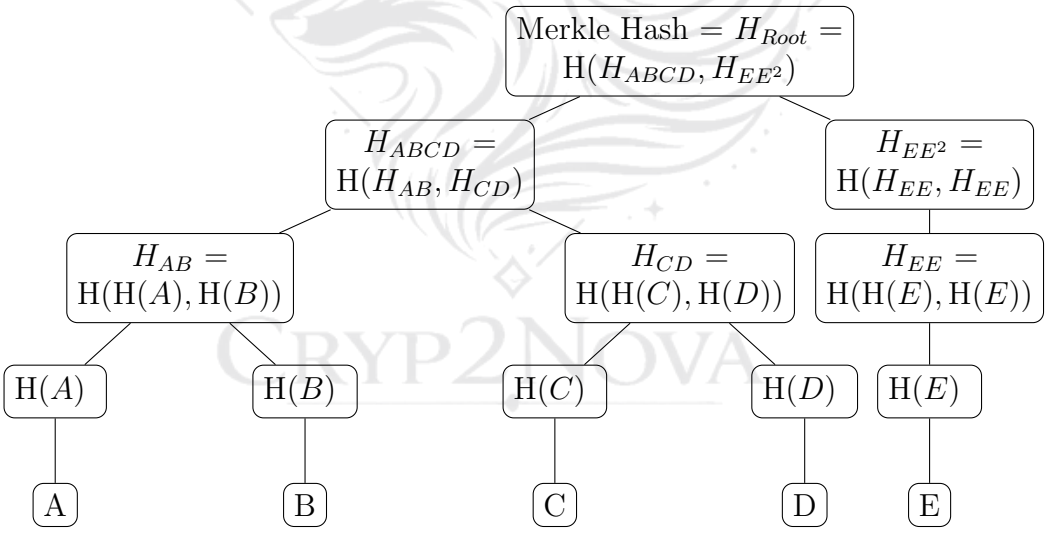


Figure 2: Four level Merkle tree composed of five data items

A benefit of using merkle trees is that the existence of a hash in a tree can be proven with only $\log(N)$ hashes. This allows for existence proofs with relatively low bandwidth requirements.

Archived copy · cryp2nova.com

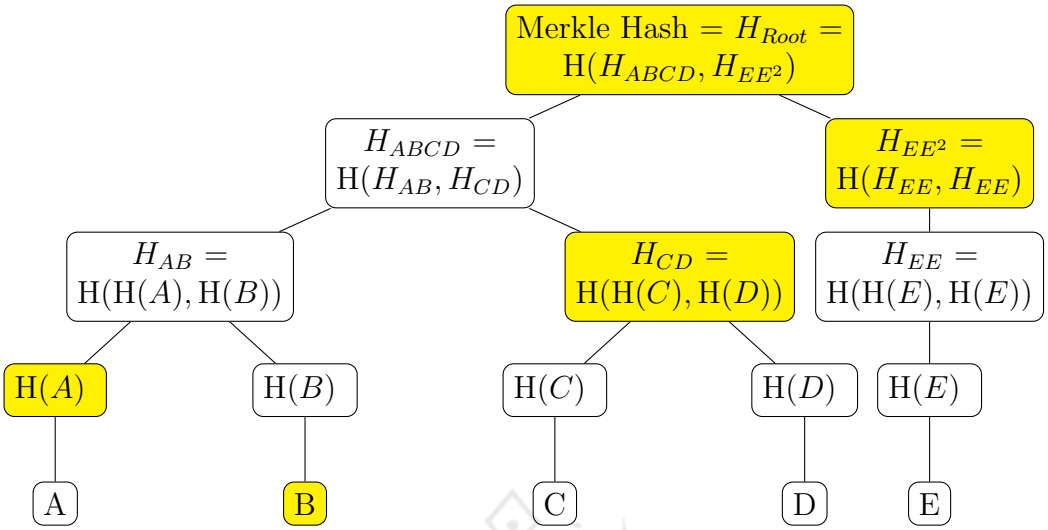


Figure 3: Merkle proof required for proving existence of B in the tree

A merkle proof for existence requires a single hash from each level of the tree. In order to prove the existence of B, a client must:

1. Calculate H(B).
2. Obtain H_{Root}; in **Symbol**, this is stored in the block header.
3. Request H(A), H_{CD}, H_{EE^2}.
4. Calculate $H_{Root'}$ = H(H(H(H(A), H(B)), H_{CD}), H_{EE^2}).
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 constructed from key value pairs, and supports both existence and non-existence proofs requiring only log(N) hashes. Non-existence proofs are possible because this tree is deterministically sorted by keys. The application of the same data, in any order, will always result in the same tree.

When inserting a new key value pair into the tree, the key is decomposed 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 slightly optimized algorithms.

Archived copy · cryp2nova.com

For illustration, consider the following key value pairs in Table 2. Some examples will use ASCII keys to more clearly elucidate concepts, while others will use hex keys to more accurately depict **Symbol** implementations.

Figure 4 depicts a full Patricia tree where each letter is represented by a separate node. Although this tree is logically correct, it is quite expansive and uses a lot of memory. A typical key is a 32 byte hash value, which implies that storing a single value could require up to 64 nodes. In order to work around this limitation, successive empty branch nodes can be collapsed into either a branch node with at least two connections or a leaf node. This leads to a different but more compact tree, as depicted in Figure 5.

key	hex-key	value
do**	`646F0000`	verb
dog*	`646F6700`	puppy
doge	`646F6765`	mascot
hors	`686F7273`	stallion

Table 2: Patricia tree example data

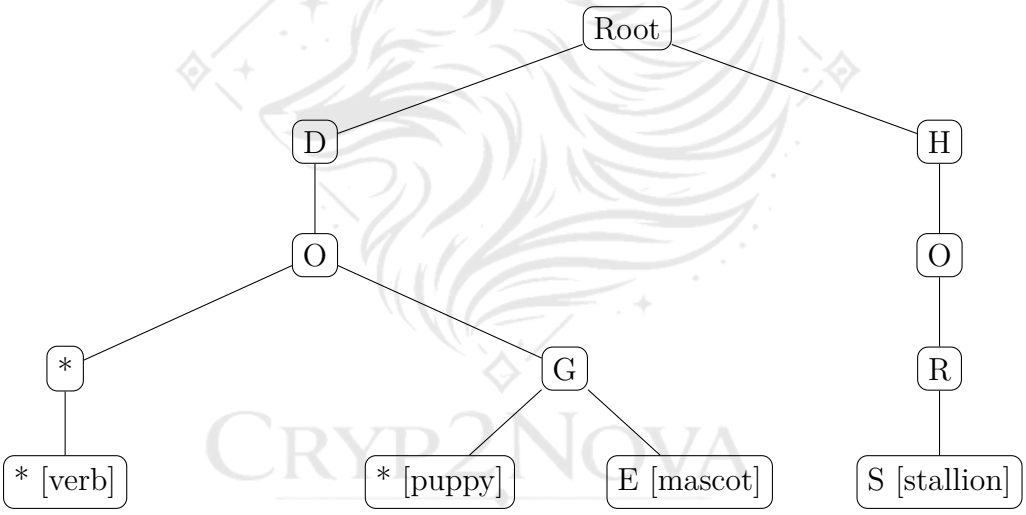


Figure 4: Conceptual (expanded) Patricia tree composed of four data items

Archived copy · cryp2nova.com

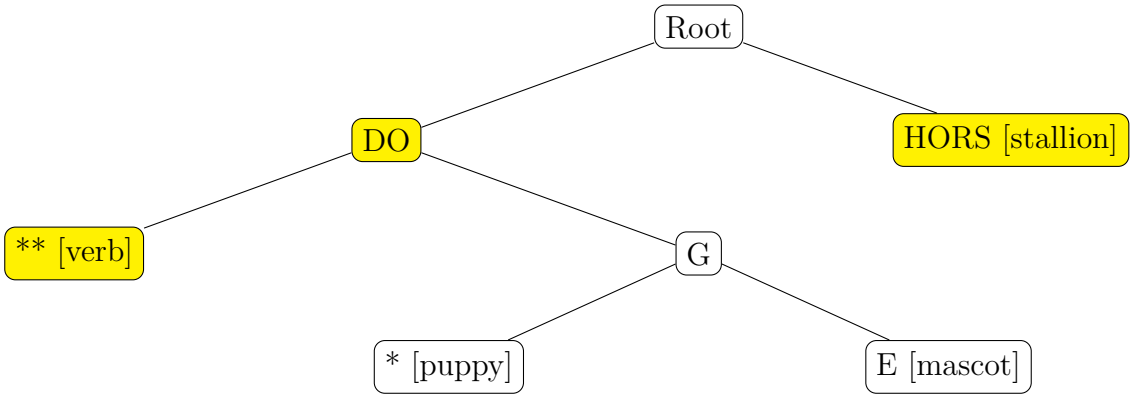


Figure 5: Conceptual (compact) patricia tree composed of four data items

4.3 Merkle Patricia Tree

A Merkle Patricia tree is a combination of Merkle and Patricia trees. The **Symbol** implementation centers around 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.

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

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

A *leaf node* is composed of the following two items:

1. TreeNodePath: Encoded tree node path (with leaf bit set).
2. ValueHash: Hash of the value associated with the key ending at the leaf.

The hash of a leaf node can be calculated by hashing its component parts:

$$\mathrm{H}(\mathit{Leaf}) = \mathrm{H}(\mathit{TreeNodePath}, \mathit{ValueHash})$$

.

A *branch node* is composed of the following items:

Archived copy · cryp2nova.com

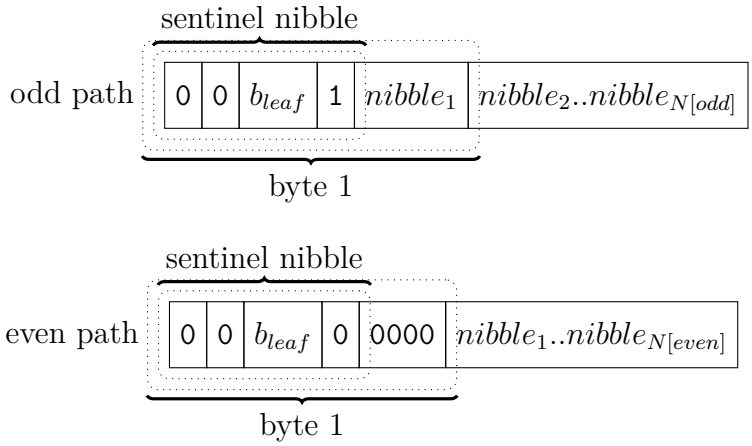


Figure 6: Tree node path encoding

1. TreeNodePath: Encoded tree node path (with leaf bit unset).
2. $LinkHash_0, \ldots, LinkHash_{15}$: Hashes of children where the index is the next nibble part of the path. When no child is present at an index, a zero hash should be used instead.

The hash of a branch node can be calculated by hashing its component parts:

$$\mathrm{H}(Branch) = \mathrm{H}(TreeNodePath, LinkHash_0, \ldots, LinkHash_{15})$$

.

Reconstructing the earlier example with hex keys yields a tree that illustrates a more accurate view of how a **Symbol** tree is constructed. Notice that each branch node index composes a single nibble of the path. This is depicted in Figure 7.

4.4 Merkle Patricia Tree Proofs

A Merkle proof for existence requires a single node from each level of the tree. In order to prove the existence of $\{key = \texttt{646F6765}, value = \mathrm{H}(mascot)\}$, a client must:

1. Calculate $\mathrm{H}(mascot)$ (remember, all leaf values are hashes).
2. Request all nodes on the path `646F6765`: $Node_6$, $Node_{646F}$, $Node_{646F67}$.
3. Verify that $Node_{646F67} :: Link[6]$ is equal to $\mathrm{H}(\mathrm{Leaf}(mascot))$.

Archived copy · cryp2nova.com

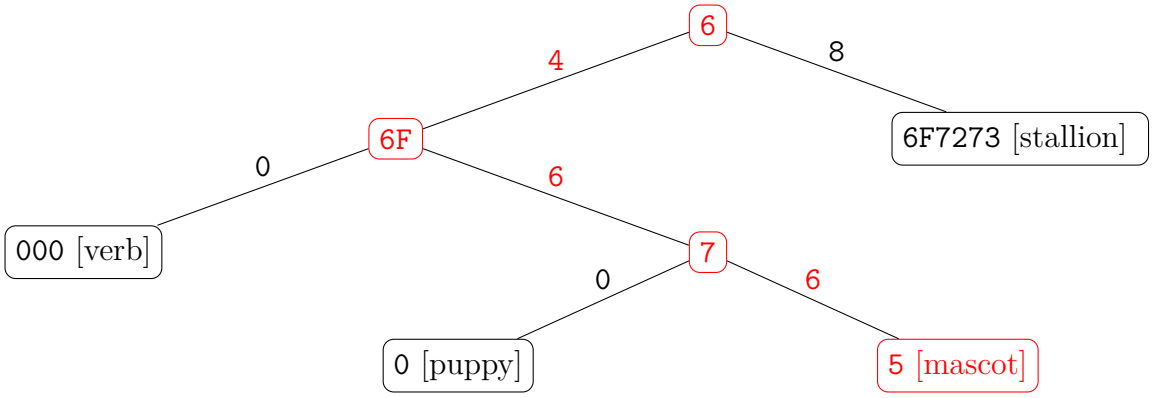


Figure 7: Realistic Patricia tree with branch and leaf nodes and all optimizations. Path to *mascot* [646F6765] is highlighted.

4. Calculate H($Node_{646F67}$) and verify that $Node_{6467} :: Link[6]$ is equal to H($Node_{646F67}$).
5. Calculate H($Node_{6467}$) and verify that $Node_{6} :: Link[4]$ is equal to H($Node_{6467}$).
6. Existence is proven if all calculated and actual hashes match.

A Merkle proof for non-existence requires a single node from each level of the tree. In order to prove the non-existence of $\{key = 646F6764, value = \text{H}(mascot)\}$, a client must:

1. Calculate H(*mascot*) (remember, all leaf values are hashes).
2. Request all nodes on the path 646F6764: $Node_{6}$, $Node_{646F}$, $Node_{646F67}$.
3. Verify that $Node_{646F67} :: Link[5]$ is equal to H(Leaf(*mascot*)). Since $Link[5]$ is unset, this check will fail. If the value being searched for was in the tree, it must be linked to this node because of the determinism of the tree.

CRYP2NOVA

Archived copy · cryp2nova.com

5 Accounts and Addresses

> A beginning is the time for taking the most delicate care that the balances are correct.
>
> - *Frank Herbert*

Symbol uses elliptic curve cryptography to ensure 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 signing public key. Each account is linked to mutable state that is updated when transactions are accepted by the network. This state is globally consistent and may contain zero or more public keys.

5.1 Addresses

A *decoded address* is a 24 byte value composed of the following three parts:

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

The checksum allows for quick recognition of mistyped addresses. It is possible to send mosaics[7] to any valid address even if the address has not previously participated in any transaction. If nobody owns the private key of the account to which the mosaics are sent, the mosaics are most likely lost forever.

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

Notice that the Base32 encoding of binary data has an expansion factor of $\frac{8}{5}$, and a decoded address's size is not evenly divisible by five. Accordingly, any Base32 representation of a decoded address will contain some extra information. While this is not inherently problematic, when encoding a decoded address, the extra input byte is set to zero by convention for consistency across clients. Additionally, the last byte of the resulting 40 character string is dropped.

[7] A mosaic is a digital asset defined on **Symbol**. Other technologies refer to mosaics as tokens.
[8] http://en.wikipedia.org/wiki/Base32

Archived copy · cryp2nova.com

5.1.1 Address Derivation

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

1. Perform 256-bit SHA3 on the public key.
2. Perform 160-bit RIPEMD_160 of hash resulting from step 1.
3. Prepend network version byte to RIPEMD_160 hash.
4. Perform 256-bit SHA3 on the result, take the first three bytes as a checksum.
5. Concatenate output of step 3 and the checksum from step 4.
6. Encode result using Base32.

CRYP2NOVA

Archived copy · cryp2nova.com

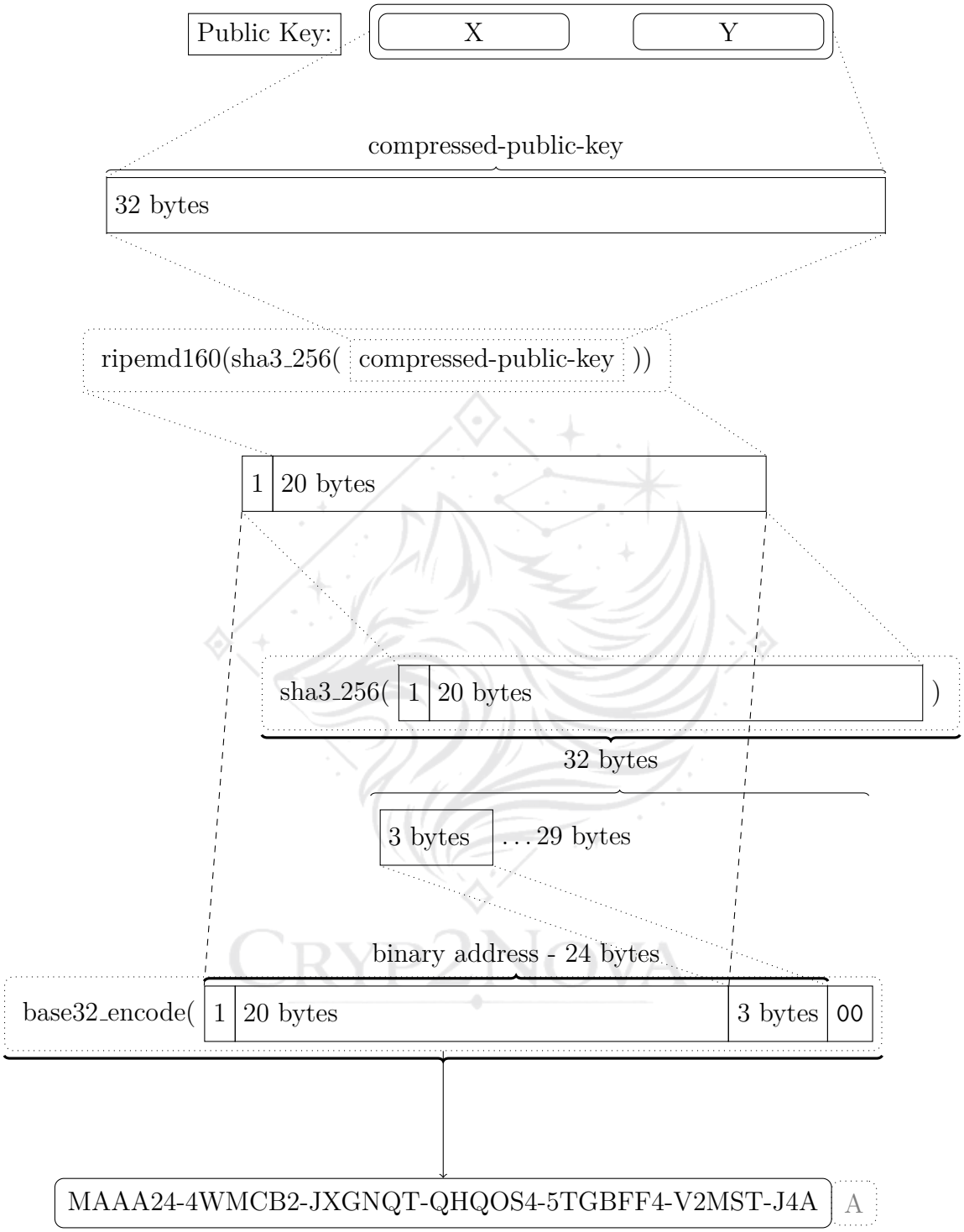


Figure 8: Address generation

Archived copy · cryp2nova.com

5.1.2 Address Aliases

An address can have one or more aliases assigned using an address alias transaction[9]. All transactions accepting addresses support using either a public key derived address or an address alias. In case of such transactions, the format of the address alias field is:

- network byte ORed with value 1
- 8 byte namespace id that is an alias
- 15 zero bytes

5.1.3 Intentional Address Collision

It is possible that two different public signing keys will yield the same address. If such an address contains any valuable assets **AND** has not been associated with a public key earlier (for example, by sending a transaction from the account), it would be possible for an attacker to withdraw funds from such an account.

In order for the attack to succeed, the attacker would need to find a private+public keypair such that the SHA3_256 of the public key would **at the same time** be equal to the RIPEMD_160 preimage of the 160-bit hash mentioned above. Since SHA3_256 offers 128 bits of security, it is mathematically improbable for a single SHA3_256 collision to be found. Due to similarities between **Symbol** addresses and Bitcoin addresses, the probability of causing a **Symbol** address collision is roughly the same as that of causing a Bitcoin address collision.

5.2 Public Keys

An account is associated with zero or more public keys. The supported types of keys are:

1. signing:
   ED25519 public key that is used for signing and verifying 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 into an account's address calculation.

2. linked:
   This public key links a main account with a remote harvester. For a `Main` account,

[9] See https://nemtech.github.io/concepts/namespace.html#addressaliastransaction for details.

Archived copy · cryp2nova.com

this public key specifies the remote harvester account that can sign blocks on its behalf. For a `Remote` account, this public key specifies the main account for which it can sign blocks. These links are bidirectional and always set in pairs.

3. node:
   This public key is set on a `Main` account. It specifies the public key of the node that it can delegate harvest on. Importantly, this does not indicate that the remote is actively harvesting on the node, but only that it has the permission to do so. As long as either the harvesting account or the node owner is honest, the account will be restricted to delegate harvesting on a single node at a time.

   An honest harvester should only send its remote harvester private key to a single node at a time. Changing its remote will invalidate all previous remote harvesting permissions granted to all other nodes (and implies forward security of delegated keys). Any older remote harvester private keys will no longer be valid and unable to be used for harvesting blocks. An honest node owner should only ever remote harvest with a remote harvester private key that is *currently* linked to its node public key

4. VRF:
   ED25519 public key that is used for generating and verifying random values. This public key must be set on a `Main` account in order for the account to be eligible to harvest.

5. voting:
   BLS public key that is used for signing and verifying finalization messages. All voting keys are temporary and must be registered with both a start and end epoch. This public key must be set on a `Main` account in order for the account to be eligible to vote. It is only valid when the current epoch is within its registered range. In order to allow a seamless switchover of keys, an account can have at most *network: maxVotingKeysPerAccount* voting keys registered at once.

CRYP2NOVA

Archived copy · cryp2nova.com

6 Transactions

> In fact, a large part of what we think of as economic activity is designed to accomplish what high transaction costs would otherwise prevent or to reduce transaction costs so that individuals can negotiate freely and we can take advantage of that diffused knowledge of which Friedrich Hayek has told us.
>
> - *Ronald Coase*

TRANSACTIONS are instructions that modify the global chain state. They are processed atomically and grouped into blocks. If any part of a transaction fails processing, the global chain state is reset to the state prior to the transaction application attempt.

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

Aggregate 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 still constraining the global set of operations allowed to a finite set. It does not require the introduction of a Turing-complete language and all of its inherent disadvantages. Developers do not need to learn any new languages nor develop bespoke contract implementations from scratch. The composition of transactions should be less error-prone and lead to fewer bugs than implementing computationally complete programs.

6.1 Basic Transaction

A basic transaction is composed of both cryptographically verifiable and unverifiable data. All verifiable data is contiguous and is signed by the transaction signer. All unverifiable data is either ignored (e.g. padding bytes) or deterministically computable from verifiable data. Each basic transaction requires verification of exactly one signature.

None of the unverifiable header fields need to be verified. `Size` is the serialized size of the transaction and can always be derived from verifiable transaction data. `Signature` is an output from signing and an input to verification. `SignerPublicKey` is an input to both signing and verification. In order for a transaction T to pass signature verification, both `Signature` and `SignerPublicKey` must be matched with the verifiable data, which

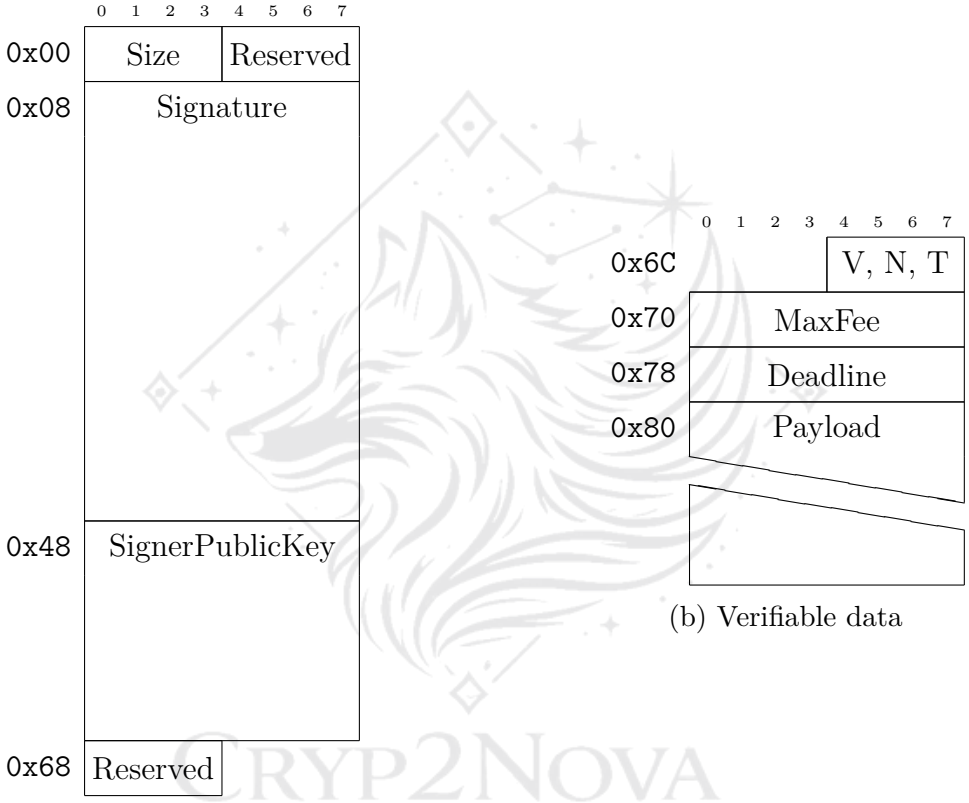
Archived copy · cryp2nova.com

has a length relative to `Size`.

$$\text{verify}(T::\text{Signature}, T::\text{SignerPublicKey}, \text{VerifiableDataBuffer}(T))$$

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

Binary layouts for all transaction types are specified in **Symbol**'s open source schema language [10]. Please refer to the published schemas for the most up to date specifications.



(a) Header (unsigned)

(b) Verifiable data

Figure 9: Basic transaction binary layout

In figures, `(V)ersion`, `(N)etwork` and `(T)ype` are abbreviated due to space concerns.

[10] Schemas can be found at https://github.com/nemtech/catbuffer.

Archived copy · cryp2nova.com

6.2 Aggregate Transaction

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

An aggregate transaction can always be submitted to the network with all requisite cosignatures. In this case, it is said to be *complete* and it is treated like any other transaction without any special processing.

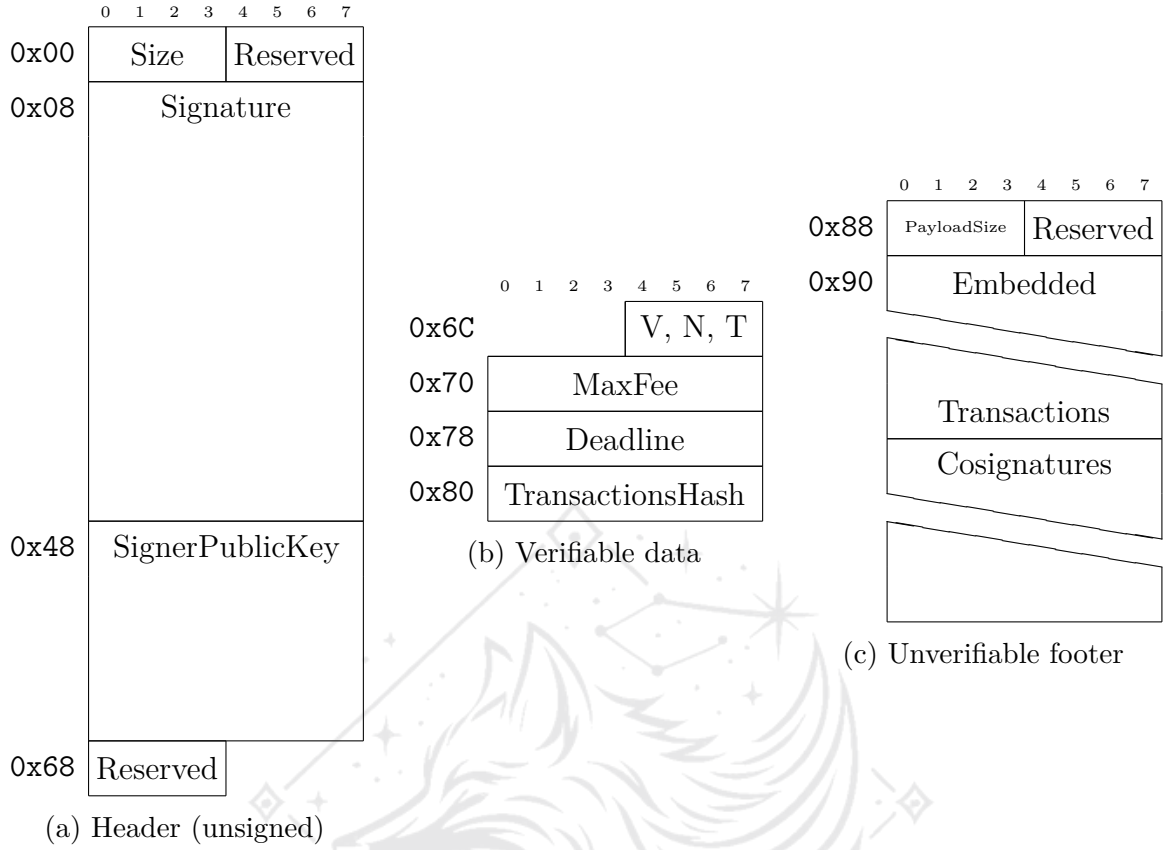
API nodes can also accept *bonded* aggregate transactions that have incomplete cosignatures. The submitter must pay a bond that is returned if and only if all requisite cosignatures are collected before the transaction times out. Assuming this bond is paid upfront, an API node will collect cosignatures associated with this transaction until it either has sufficient signatures or times out.

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

None of the unverifiable footer fields need to be verified. `PayloadSize` is a computed size field that must be correct in order 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.

CRYP2NOVA

Archived copy · cryp2nova.com



(a) Header (unsigned)

(b) Verifiable data

(c) Unverifiable footer

Figure 10: Aggregate transaction header binary layout

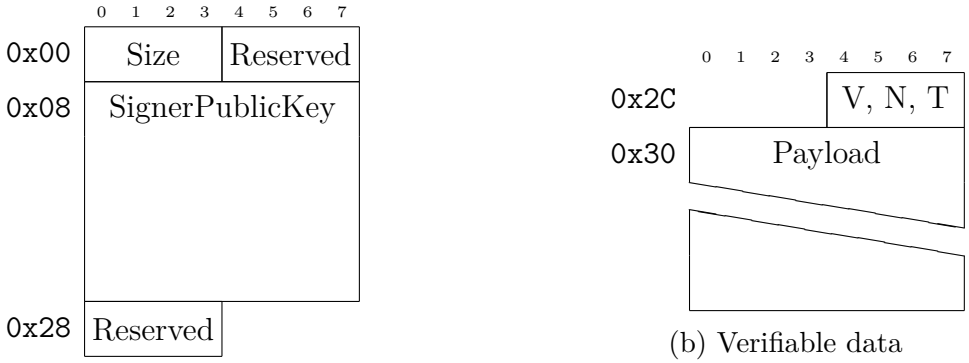
6.2.1 Embedded Transaction

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. `Signature` is removed because all signature information is contained in cosignatures. `MaxFee` and `Deadline` are removed because they are specified by the parent aggregate transaction.

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

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

Archived copy · cryp2nova.com



(a) Header (unsigned)

(b) Verifiable data

Figure 11: Embedded transaction binary layout

6.2.2 Cosignature

A cosignature is composed of a version[11], a public key and its corresponding signature. Zero or more cosignatures are appended at the end of an aggregate transaction. Cosignatures are used to cryptographically verify an aggregate transaction involving multiple parties.

[11] `Version` is reserved for future extensibility. Currently, it is expected to always be zero.

CRYP2NOVA

Archived copy · cryp2nova.com

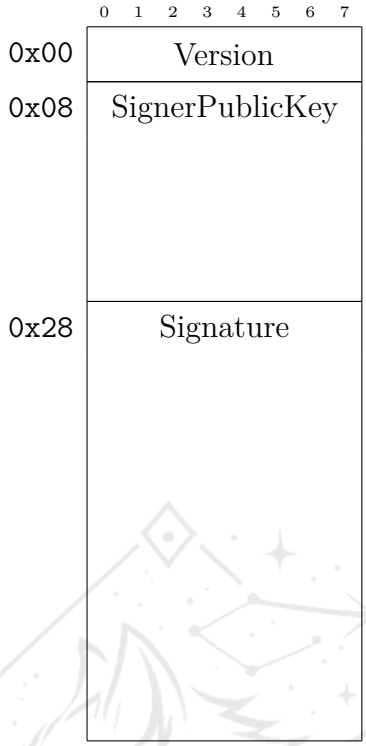


Figure 12: Cosignature binary layout

In order for an aggregate transaction A to pass verification, it must pass basic transaction signature verification and have a cosignature for every embedded transaction signer [12].

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

$$\text{verify}(A\text{::Signature}, A\text{::SignerPublicKey}, \text{VerifiableDataBuffer}(A))$$

Additionally, all cosignatures must pass signature verification. Notice that cosigners sign the hash of an aggregate transaction data, not the data directly.

$$\sum_{0 \leq i \leq N_C} \text{verify}(C\text{::Signature}, C\text{::SignerPublicKey}, \text{H}(\text{VerifiableDataBuffer}(A)))$$

Finally, there must be a cosignature that corresponds to and satisfies each embedded transaction signer.

[12] In the case of multisignature accounts, there must be enough cosignatures to satisfy the multisignature account constraints.

Archived copy · cryp2nova.com

6.2.3 Extended Layout

The aggregate transaction layout described earlier was correct with one simplification. All embedded transactions are padded so that they end on 8-byte boundaries. This ensures that all embedded transactions and cosignatures begin on 8-byte boundaries as well. The padding bytes are never signed nor included in any hashes.

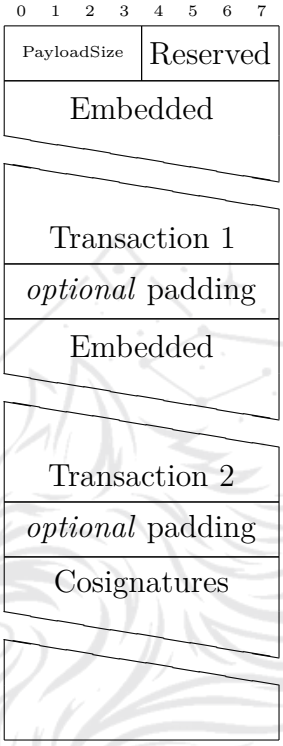


Figure 13: Aggregate transaction footer with padding

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 confirmations of the same transaction. The latter is more specific and is used when calculating `TransactionsHash` (see 6.2: Aggregate Transaction).

A transaction's entity hash is calculated as a hash of the following:

1. Transaction `Signature` - If this field was not included, an adversary could prevent a transaction from being included in the network by frontloading it with a nearly identical transaction that contained a malformed signature.

Archived copy · cryp2nova.com

2. Transaction `SignerPublicKey` - If this field was not included, an adversary could prevent a transaction from being included in the network by frontloading it with a nearly identical transaction that contained a malformed signer public key.

3. Network `GenerationHashSeed` - This field prevents cross network transaction replay attacks [13].

4. Transaction verifiable data

All confirmed transactions must have a unique entity hash. An aggregate transaction's entity hash is independent of its cosignatures. This prevents the same aggregate transaction from being confirmed multiple times with different sets of valid cosignatures.

A regular transaction's Merkle component hash is identical to its entity hash. An aggregate transaction's Merkle component hash is calculated by hashing its entity hash concatenated with all the public keys of its cosignatures [14]. This ensures that `TransactionsHash` reflects all cosignatures that allowed the confirmation of an aggregate transaction.

CRYP2NOVA

[13] Additionally, when signing and verifying transaction data, the network `GenerationHashSeed` is prepended to the data so that transaction signatures will only verify on networks with a matching `GenerationHashSeed`.

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

Archived copy · cryp2nova.com

7 Blocks

> A stumbling block to the pessimist is a stepping-stone to the optimist.
>
> - *Eleanor Roosevelt*

Symbol is, at its core, a blockchain. A blockchain is an ordered collection of blocks. Understanding the parts of a **Symbol** block is fundamental to understanding the platform's capabilities.

The layout of a block is similar to the layout of an aggregate transaction (see 6.2: Aggregate Transaction). A block shares the same unverifiable header[15] as an aggregate transaction, and this data is processed in the same way. Likewise, a block has a footer of unverifiable data followed by transactions. Unlike an aggregate transaction, a block is followed by basic - not embedded - transactions, and each transaction within a block is signed independently of the block signer[16]. This allows any transaction satisfying all conditions to be included in any block.

None of the unverifiable footer fields need to be verified. `Reserved` bytes are used for padding and have no intrinsic meaning.

7.1 Block Fields

`Height` is the block sequence number. The first block, called the *nemesis block*, has a height of one. Each successive block increments the height of the previous block by one.

`Timestamp` is the number of milliseconds that have elapsed since the nemesis block. Each successive block must have a timestamp greater than that of the previous blocks because block time is strictly increasing. Each network tries to keep the average time between blocks close to *target block time*.

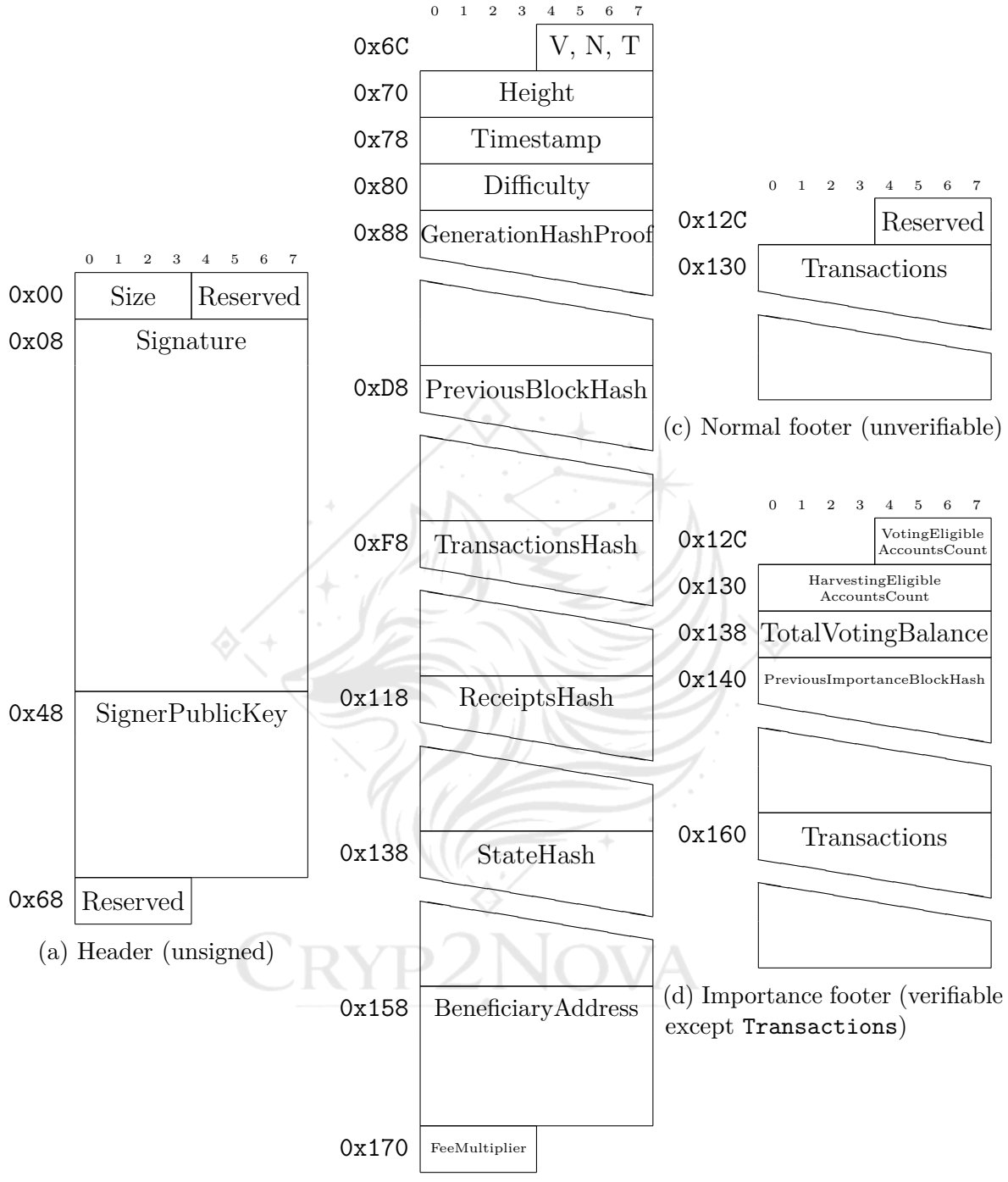
`Difficulty` determines how hard is to harvest a new block, based on previous blocks. Difficulty is described in detail in 8.1: Block Difficulty.

`GenerationHashProof` is the VRF proof generated with the block harvester's VRF private key. It is composed of a 32 byte γ, a 16 byte verification hash (c) and a 32 byte scalar (s) (see 3.3: Verifiable Random Function (VRF)). This is used to guarantee that future block harvesters are unpredictable (see 8.3: Block Generation).

[15] For emphasis, this is not referring to the verifiable block header. It is referring to fields like `Signature` that precede the verifiable block header.

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

Archived copy · cryp2nova.com



(a) Header (unsigned)

(b) Verifiable data

(c) Normal footer (unverifiable)

(d) Importance footer (verifiable except `Transactions`)

Figure 14: Block header binary layout

Archived copy · cryp2nova.com

`PreviousBlockHash` is the hash of the previous block. This is used to guarantee that all blocks within a blockchain are linked and deterministically ordered.

`TransactionsHash` is the Merkle root hash of the hashes of the transactions stored within the block [17]. In order to compute this field, a Merkle tree is constructed by adding each transaction hash in natural order. The resulting root hash is assigned to this field.

`ReceiptsHash` is the Merkle root hash of the hashes of the receipts produced while processing the block. When a network is configured without *network:enableVerifiableReceipts*, this field must be zeroed in all blocks (see 7.2: Receipts).

`StateHash` is the hash of the global blockchain state after processing the block. When a network is configured without *network:enableVerifiableState*, this field must be zeroed in all blocks. Otherwise, it's calculated as described in 7.3: State Hash.

`BeneficiaryAddress` is the account that will be allocated the block's beneficiary share when the network has a nonzero *network:harvestBeneficiaryPercentage*. This field can be set to any account, even one that isn't yet known to the network. This field is set by the node owner when a new block is harvested. If this account is set to one owned by the node owner, the node owner will share in fees paid in all blocks harvested on its node. This, in turn, incentivizes the node owner to run a strong node with many delegated harvesters.

`FeeMultiplier` is a multiplier that is used to calculate the effective fee of each transaction contained within a block. The *node:minFeeMultiplier* is set by the node owner and can be used to accomplish varying goals, including maximization of profits or confirmed transactions. Assuming a block B containing a transaction T, the effective transaction fee can be calculated as:

$$\text{effectiveFee}(T) = B.\text{FeeMultiplier} \cdot \text{sizeof}(T)$$

If the effective fee is greater than the transaction `MaxFee`, the transaction signer keeps the difference. Only the effective fee is deducted from the transaction signer and credited to the harvester. Further information about fee multipliers can be found in 8.3: Block Generation.

7.1.1 Importance Block Fields

Every block at which an importance calculation takes place contains an extended footer with additional verifiable information [18]. This information reflects the global blockchain state after processing the block.

[17] This field has the same purpose as the identically named field in an aggregate transaction.
[18] *network:votingSetGrouping* is expected to be a multiple of *network:importanceGrouping*.

Archived copy · cryp2nova.com

`VotingEligibleAccountsCount` is the number of accounts eligible to vote in the finalization epoch covering the next importance group. This is an exact number and only includes accounts that satisfy all voting requirements. For example, accounts that do not have a voting key registered are excluded.

`HarvestingEligibleAccountsCount` is the number of accounts with a balance of at least *network:minHarvesterBalance*. This is an estimate and a maximum. The actual number of harvesting eligible accounts might be lower because harvesting ineligible accounts are not excluded. For example, accounts that do not have a VRF key registered are not excluded.

`TotalVotingBalance` is the sum of the balance of all accounts eligible to vote in the finalization epoch covering the next importance group. This value allows the trustless verification of finalization proofs. It can be used as the authoritative denominator when calculating the percentage of voting stake accumulated by voters of a finalization proof.

`PreviousImportanceBlockHash` is the hash of the previous importance block. This is a longer duration guarantee, relative to `PreviousBlockHash`, that all blocks within a blockchain are linked and deterministically ordered. It can be used to enable a trustless fast sync protocol, which only needs to download importance block headers instead of all block headers.

7.2 Receipts

During the execution of a block, zero or more receipts are generated. Receipts are primarily used to communicate state changes triggered by side effects to clients. In this way, they allow simpler clients to still be aware of complex state changes.

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

Another example is around harvest rewards. Receipts are produced that indicate the main - not delegated - account that gets credited and any beneficiary splits. 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 transaction, address resolution and mosaic resolution.

Archived copy · cryp2nova.com

7.2.1 Receipt Source

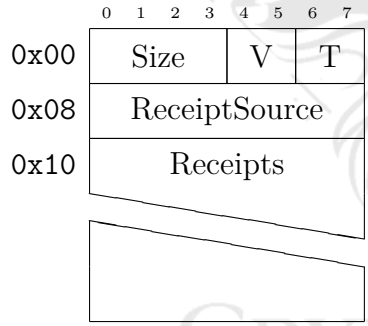
Each part of a block that is processed is assigned a two-part block-scoped identifier. The source (0, 0) is always used to identify block-triggered events irrespective of the number of transactions in a block.

Source	Primary Id	Secondary Id
Block	0	0
Transaction	1-based index within the block	0
Embedded Transaction	1-based index of containing aggregate within the block	1-based index within the aggregate

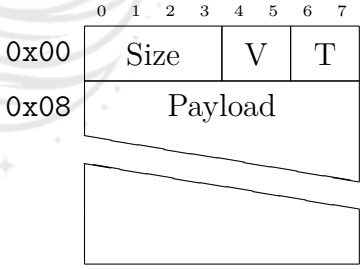
Table 3: Receipt source values

7.2.2 Transaction Statement

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



(a) Transaction statement binary layout



(b) Receipt binary layout

Figure 15: Transaction statement layout

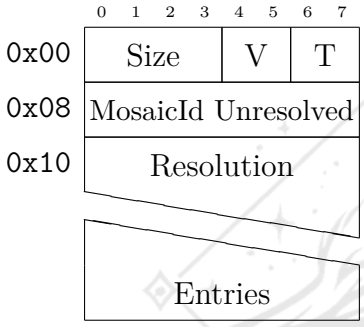
Transaction statement data is not padded because it is only written during processing and never read, so padding yields no server performance benefit. A transaction statement hash is constructed by concatenating and hashing all statement data excepting `Size` fields, which are derivable from other data.

Archived copy · cryp2nova.com

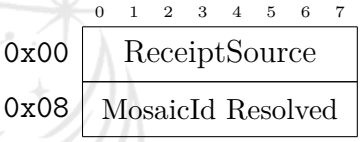
7.2.3 Resolution Statements

Resolution statements are used exclusively for indicating 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 change between their usages. Each statement is composed of an unresolved value and one or more resolutions in order to support this.

There are two types of resolution statements - address and mosaic - corresponding to the two types of aliases. Although Figure 16 illustrates the layout of a mosaic 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 mosaic ids.



(a) Mosaic resolution statement binary layout



(b) Resolution entry binary layout

Figure 16: Mosaic resolution statement layout

Resolution statement data is not padded because it is only written during processing and never read, so padding yields no server performance benefit. A resolution statement hash is constructed by concatenating and hashing all statement data excepting `Size` fields, which are derivable from other data.

It is important to note that a resolution statement is only produced when a resolution occurs. If an alias is registered or changed in a block, but not used in that block, no resolution statement will be produced. However, a resolution statement will be produced by each block that contains that alias and requires it to be resolved.

7.2.4 Receipts Hash

In order to calculate a block's receipt hash, first all statements generated during block processing are collected. Then a Merkle tree is created by adding all statement hashes in the following order:

Archived copy · cryp2nova.com

1. Hashes of transaction statements ordered by receipt source.
2. Hashes of address resolution statements ordered by unresolved address.
3. Hashes of mosaic resolution statements ordered by unresolved mosaic id.

When a network is configured with *network:enableVerifiableReceipts*, the root hash of this merkle tree is set as the block's `ReceiptHash`. A client can perform a Merkle proof to prove a particular statement was produced during the processing of a specific block.

7.3 State Hash

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

When a network is configured with *network:enableVerifiableState*, a Patricia tree is created for each repository. This produces a single hash that deterministically fingerprints each repository. Accordingly, assuming N repositories, N hashes deterministically fingerprint the global blockchain state.

It is possible to store all N hashes directly in each block header, but this is undesirable. Each block header should be as small as possible because all clients, minimally, need to sync all headers to verify a chain is rooted to the nemesis block. Additionally, 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 hashed to calculate the `StateHash`, which is a single hash that deterministically fingerprints the global blockchain state.

$$\begin{aligned} \mathit{RepositoryHashes} &= 0 \\ \mathit{RepositoryHashes} &= \sum_{0 \le i \le N} \mathrm{concat}(\mathit{RepositoryHashes}, \mathit{RepositoryHash}_i) \\ \mathit{StateHash} &= \mathrm{H}(\mathit{RepositoryHashes}) \end{aligned}$$

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

Archived copy · cryp2nova.com

7.4 Extended Layout

The block layout described earlier was correct with one simplification[20]. All transactions are padded so that they end on 8-byte boundaries. This ensures that all transactions begin on 8-byte boundaries as well. The padding bytes are never signed nor included in any hashes.

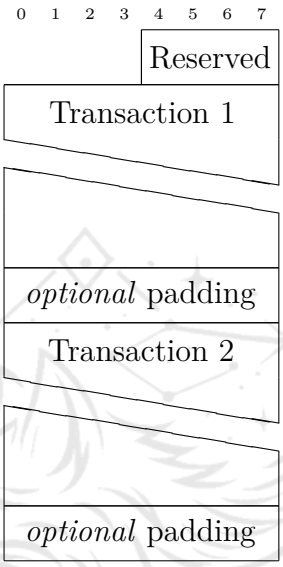


Figure 17: Block transaction footer with padding

7.5 Block Hashes

Each block has one associated hash - an *entity hash*. This hash uniquely identifies a block and is used to prevent processing the same block multiple times.

A block's entity hash is calculated as a hash of the following:

1. Block `Signature`
2. Block `SignerPublicKey`
3. Block (header) verifiable data

The inputs into an entity hash for a block and a transaction are similar. The one difference is that the network `GenerationHashSeed` is not an input into the entity hash calculation

[20]This is consistent with the extended layout of an aggregate transaction as well.

Archived copy · cryp2nova.com

for a block. Including this field would serve no purpose for a block. Any block containing at least one transaction cannot be replayed in a different network because transaction entity hashes are different across networks. Consequently, the block's `TransactionsHash` would also be different.

CRYP2NOVA

Archived copy · cryp2nova.com

8 Blockchain

> It is a mistake to look too far ahead. Only one link of the chain of destiny can be handled at a time.
>
> - *Winston Churchill*

Symbol is centered around a public ledger called the blockchain that links blocks together. The complete transaction history is held in the blockchain. All blocks, and transactions within blocks, are deterministically and cryptographically ordered. The maximum number of transactions per block can be configured per-network.

8.1 Block Difficulty

The nemesis block has a predefined *initial difficulty* of 10^{14}. All difficulties are clamped between a minimum of 10^{13} and a maximum of 10^{15}.

The difficulty for a new block is derived from the difficulties and timestamps of the most recently confirmed blocks. The number of blocks taken into consideration is configurable per-network.

If less than *network:maxDifficultyBlocks* are available, only those available 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 setting.

If the new difficulty is more than 5% larger or smaller than the difficulty of the last block, then the change is capped to 5%. The maximum change rate of 5% per block makes it hard for an attacker with considerably less than 50% importance to create a better chain in secret. Since difficulty is roughly correlated with the total amount of importance currently harvesting, the attacker's secret chain will necessarily have a much lower difficulty. Limiting the maximum difficulty decrease per block means the

Archived copy · cryp2nova.com

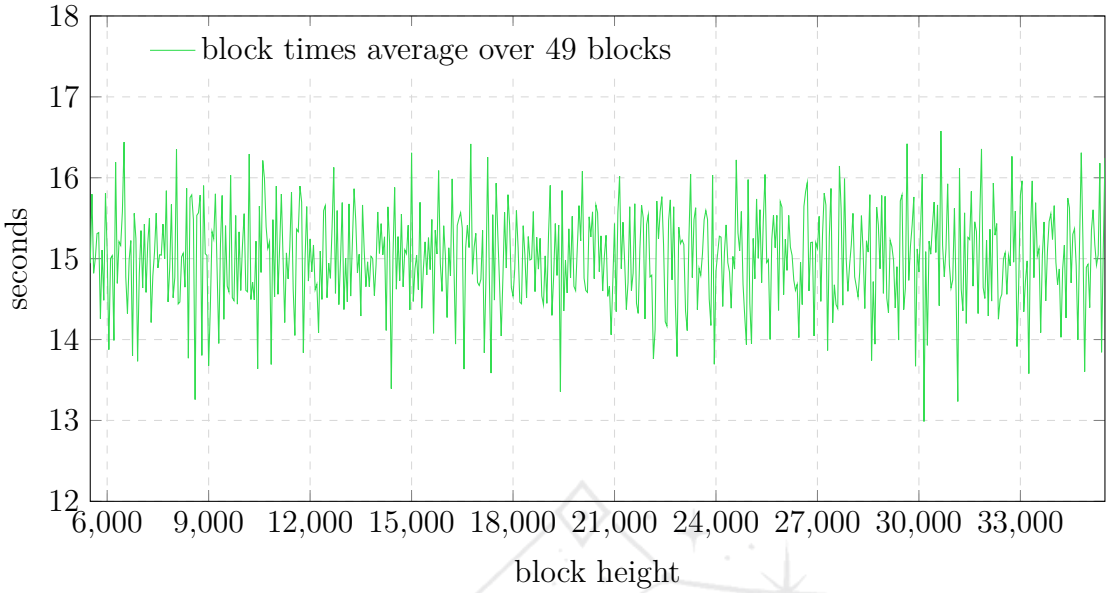


Figure 18: Dev network average block times, with target block time = 15s

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

8.2 Block Score

The score for a block is derived from its difficulty and the time (in seconds) that has elapsed since the last block:

$$score = difficulty - time\ elasped\ since\ last\ block \qquad \text{(block score)}$$

8.3 Block Generation

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

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

Archived copy · cryp2nova.com

age of fees will be credited to the network specified *network:harvestNetworkFeeSinkAddress*. This account is intended to allow networks to be self-sustaining and can be used for incentivization programs such as rewarding voting nodes. The remainder of fees will be credited to the harvester account.

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

1. Importance score at the last importance recalculation height is nonzero.
2. Balance no less than a network defined *network:minHarvesterBalance*.
3. Balance no greater than a network defined *network:maxHarvesterBalance* [21].
4. VRF public key is registered for the account.

An account owner can delegate its importance to some other account[22] in order to avoid exposing a private key with funds.

The actual reward a harvester receives is customizable based on network settings. If *inflation* configuration has nonzero values, each harvested block may contain an additional inflation block reward. This makes harvesting more profitable. If harvesting fee sharing is enabled (via *network:harvestBeneficiaryPercentage*), the harvester will forfeit a share of fees to the node hosting its harvesting key. This makes running network nodes more profitable but harvesting less profitable.

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

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

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

[21] This feature is primarily intended to prevent core funds and exchange accounts from harvesting.

[22] See https://nemtech.github.io/concepts/harvesting.html#accountlinktransaction for details.

[23] In all cases, all available transactions must already fulfill the requirement that *node:minFeeMultiplier* indicates.

Archived copy · cryp2nova.com

1. `oldest`:
   This is the least resource-intensive strategy and is recommended for high TPS networks. Transactions are added to a new block in the order in which they have been received. This ensures that the oldest transactions are selected first and attempts to minimize the number of transactions timing out. As a consequence, this strategy is rarely profit maximizing for the harvester.

2. `maximize-fee`:
   Transactions are selected in such a way that the cumulative fee for all transactions in a block is maximized. A profit-optimizing node will pick this strategy. Maximizing the total block fee does not necessarily mean that the number of transactions included is maximized as well. In fact, in many cases, the harvester will only include a subset of the transactions that are available.

3. `minimize-fee`:
   Transactions with the lowest maximum fee multipliers are selected first by this strategy. Altruistic nodes will pick this strategy together with a very low *node: minFeeMultiplier*. If this setting is zero, then the harvester will include transactions with zero fees first. This allows users to send transactions that get included in the blockchain for free! In practice, it is likely that only a few nodes will support this. Even with such a subset of nodes running, zero fee transactions will still have the lowest probability of getting included in a block because they will always be supported by the least number of nodes in the network.

Transactions can initiate transfers of both static and dynamic amounts. Static amounts are fixed and independent of external factors. For example, the amount specified in a transfer transaction[24] is static. The exact amount specified is always transferred from sender to recipient. In contrast, dynamic amounts are variable relative to the average transaction cost. These are typically reserved for fees paid to acquire unique network artifacts, like namespaces or mosaics. For such artifacts, a flat fee is undesirable because it would be unresponsive to the market. Likewise, solely using a single block's `FeeMultiplier` is problematic because harvesters could cheat and receive artifacts for free by including registrations 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:defaultDynamicFeeMultiplier* is used when there are not enough values and as a replacement for zero values. The latter adjustment ensures that the effective amounts are always nonzero. In order to arrive at the effective amount, the base amount is multiplied by the dynamic fee multiplier.

[24] See https://nemtech.github.io/concepts/transfer-transaction.html for details.

Archived copy · cryp2nova.com

$$\textit{effective amount} = \textit{base amount} \cdot \textit{dynamic fee multiplier}$$

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:

$$\begin{aligned} \mathrm{gh}(1) &= \mathrm{generationHash} && \text{(generation hash)} \\ \mathrm{gh}(N) &= \mathrm{verify_vrf_proof}(\mathrm{proof}(\textit{block}(N)), \mathrm{gh}(N-1), \textit{VRF public key of account}) \end{aligned}$$

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 for an adversary with perfect information. Until a block is pushed to the network, its generation hash is unknown to all but its harvester. Accordingly, until the best block is pushed to the network, since each generation hash is dependent on the preceding generation hash, the input into the generation hash calculation for the next block is unknown until that point.

A VRF public key needs to be registered on the blockchain before it can be used to harvest a block. This preregistration requirement prevents an adversary from choosing an arbitrary VRF public key that can maximize its hit on the next block. As a result, the adversary is unable to produce many blocks in an attempt to maximize its hit.

8.5 Block Hit and Target

To check if an account is allowed to 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 is allowed to create a new block whenever $\textit{hit} < \textit{target}$. Since *target* 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 harvesting, the importance of the original account is used instead of the importance of the delegated account.

Archived copy · cryp2nova.com

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 can be enabled, which results in more stable block times. If enabled, *multiplier* above is calculated in the following way [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 [27].

First, let's rewrite value above 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 $\frac{gh}{2^{256}}$ is always < 1, therefore log will always yield negative value. Now, $\log_2\left(\frac{gh}{2^{256}}\right)$, can be rewritten as $\log_2(gh) - \log_2(2^{256})$.

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

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

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

Archived copy · cryp2nova.com

Dropping 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:

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

The implementation approximates the logarithm using only the first 32 non-zero bits of the new generation hash. There's also additional handling for edge cases.

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

8.6 Automatic Delegated Harvester Detection

When *user:enableDelegatedHarvestersAutoDetection* is set, the server allows other accounts to register as delegated harvesters via special transfer messages. The server inspects all transfer messages sent to the account matching its node certificate public key and pushes matching ones 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 harvest on a node when all of the following links are setup:

- Main account has signed the delegated harvest request.
- Linked remote public key matches the encrypted linked remote harvester private key.
- VRF public key matches the encrypted VRF private key.
- Node public key matches the node's certificate public key (see 12.2: Connection Handshake).

Messages are partially encrypted in order to prevent an adversary from obtaining plain text remote harvester and VRF private keys. AES256 GCM is the (symmetric) encryption

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

Archived copy · cryp2nova.com

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 bytes	No	Magic bytes indicating the message is a harvest request
Ephemeral public key	32 bytes	No	Public key used to derive the symmetric encryption key
AES GCM Tag	16 bytes	No	MAC tag over the encrypted data
AES GCM IV	12 bytes	No	Initialization vector for AES GCM algorithm
Signing private key	32 bytes	Yes	Remote harvester signing private key linked to the main account
VRF private key	32 bytes	Yes	VRF private key linked to the main account

Table 4: Harvest request message format

If possible, announced delegated harvester private keys will be used by the server to harvest blocks. A server can have at most *harvesting:maxUnlockedAccounts* harvesters. Upon reaching that limit, the evaluation of any new delegated harvester is based on the *harvesting:delegatePrioritizationPolicy* setting. When the policy is set to `Age`, accounts announced earlier are preferred. As a result, a new delegated harvester cannot replace any existing delegated harvester. When the policy is set to `Importance`, accounts with higher importances are preferred. As a result, a new delegated harvester can replace an existing delegated harvester with less importance.

Successful announcements are stored in the `harvesters.dat` file. Accepted delegated harvesters are persisted across server reboots. The server does not provide any explicit confirmation that it is or is not currently harvesting with a specific delegated harvester. The blockchain only stores a record of all the links related to delegated harvesting, but not the current activity.

8.7 Blockchain Synchronization

A score can be assigned to any chain of blocks by summing the scores of the component blocks:

$$score = \sum_{block \in blocks} block\ score \qquad \text{(blockchain score)}$$

Archived copy · cryp2nova.com

Blockchain synchronization is crucial to maintaining distributed consensus. Periodically, a local node will ask a remote node about its chain. The remote node is selected from a set of partners based on various factors, including reputation (see 13: Reputation).

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

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

Figure 19 illustrates the process in more detail.

CRYP2NOVA

Archived copy · cryp2nova.com

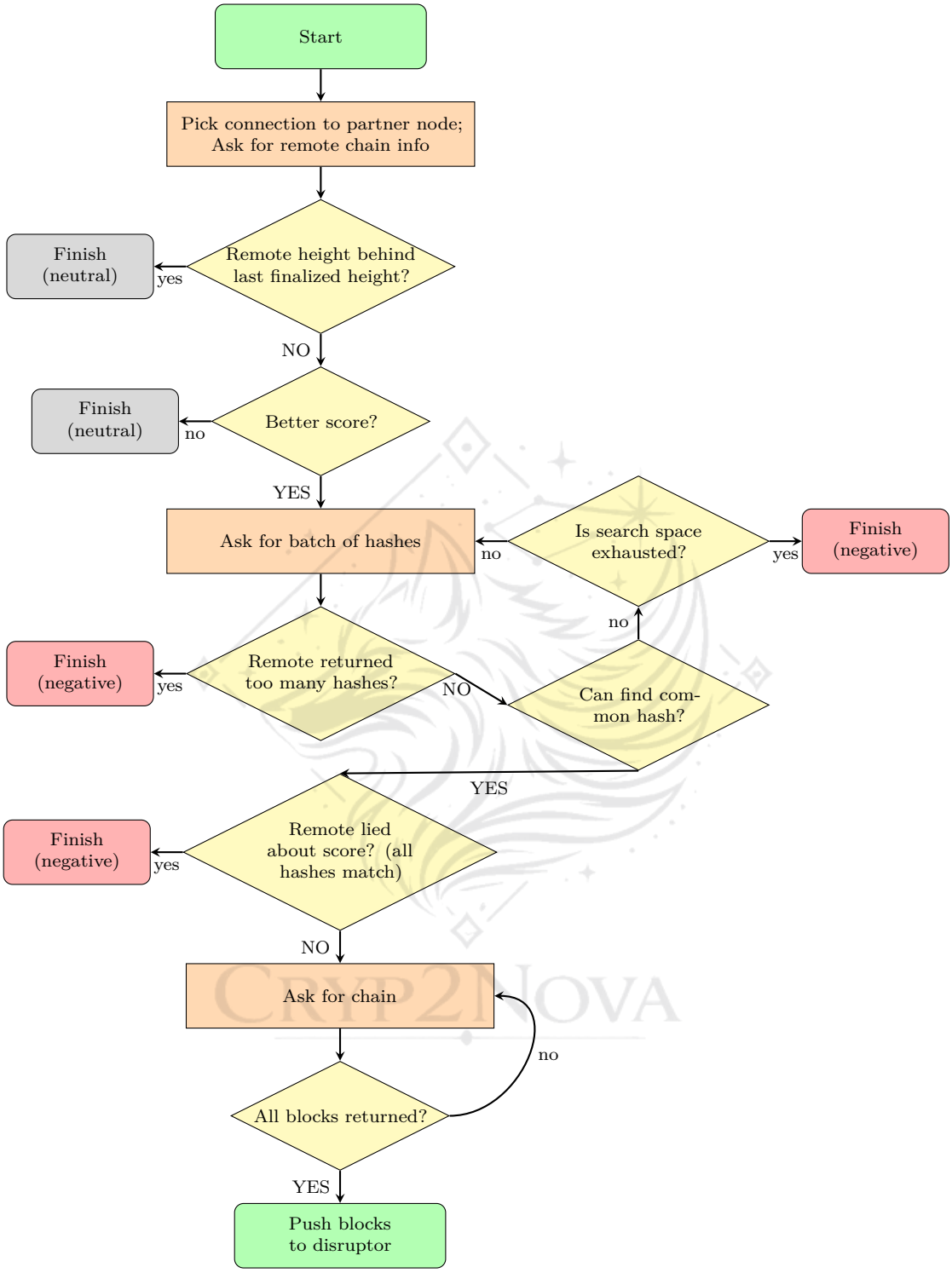


Figure 19: Blockchain synchronization flow chart

Archived copy · cryp2nova.com

8.8 Blockchain Processing

Execution

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

In order 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 appended to the stream. At the end of this process, the notification stream completely describes all state changes specified in the block and its transactions.

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

After all notifications produced by a block are processed, the `ReceiptsHash` (see 7.2.4: Receipts Hash) and `StateHash` (see 7.3: State Hash) fields are calculated and checked for correctness. Importantly, when *network:enableVerifiableState* is enabled, this is the point at which all state Patricia trees get updated.

Rollback

Occasionally, a block that has been previously confirmed needs to be undone. This is required in order to allow fork resolution. For example, to replace a worse block with a better block. In **Symbol**, at most *network:maxRollbackBlocks* can be rolled back at once. Forks larger than this setting are irreconcilable.

When a block is being rolled back, it is decomposed into an ordered stream of notifications. This stream is reversed relative to the stream used during execution. Since transactions in a block may be dependent on changes made by previous transactions in the same block, they need to be undone before their dependencies are undone.

Once the stream of notifications is prepared, each notification is processed individually. No validation is needed because the rollback operation is returning the blockchain to a previous state that is known to be valid. Instead, the changes specified by the notification are simply reverted from the in memory blockchain state and the next notification is

[29] A more detailed description of these stages can be found in 9.1: Consumers.

Archived copy · cryp2nova.com

processed.

After all notifications produced by a *blockchain part* are processed, the previous blockchain state is restored. When *network:enableVerifiableState* 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 prior to the last rolled back block.

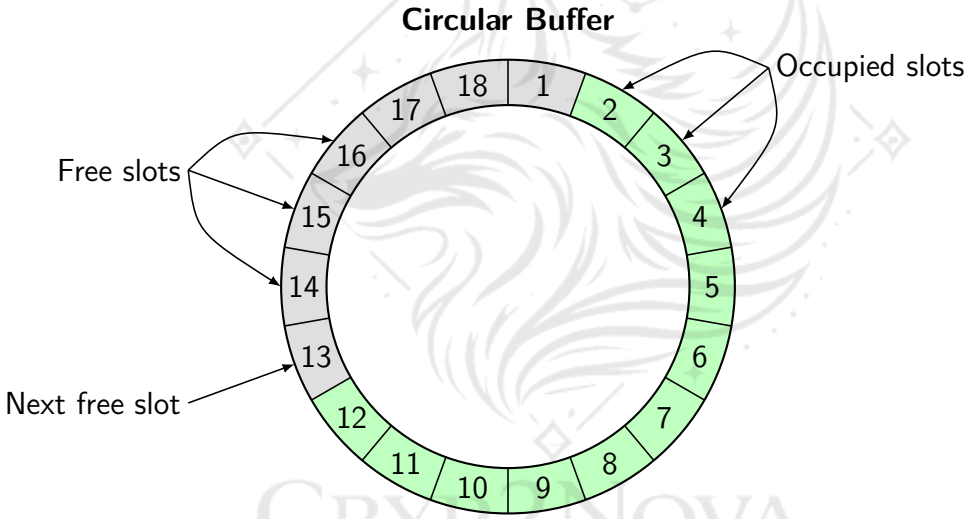
CRYP2NOVA

Archived copy · cryp2nova.com

9 Disruptor

> Death is the great disruptor. It thrusts us opposite life's mirror, invites our truthful exploration, and reveals the naked truth from which rebirth is possible and we are free to reinvent ourselves anew.
>
> - *B.G. Bowers*

One main goal of **Symbol** is to achieve high throughput. In order to help achieve this goal, the disruptor[30] pattern is used to perform most data processing. A disruptor uses a ring buffer data structure to hold all elements in need of processing. New elements are inserted into the next free slot of the ring buffer. Fully processed elements are removed to make space for new elements. Since the ring buffer has a finite number of slots, it can run out of space if processing can't keep up with new insertions. The behavior of **Symbol**, in such a case, can be configured to either exit the server or discard new data until a slot becomes available.

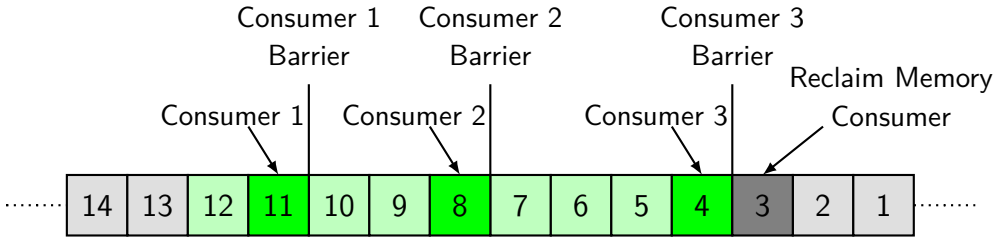


Each element in the ring buffer is processed by one or more consumers. Each consumer takes a single element as input. Some consumers calculate data from the input and attach it to the element, while others validate the element or alter the global chain state using the element's data. Some consumers depend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

[30] https://en.wikipedia.org/wiki/Disruptor_(software)

Archived copy · cryp2nova.com

Consumers can set an element's completion status to `CompletionStatus::Aborted` in case it is already known or invalid for some reason. Subsequent consumers ignore aborted elements.



9.1 Consumers

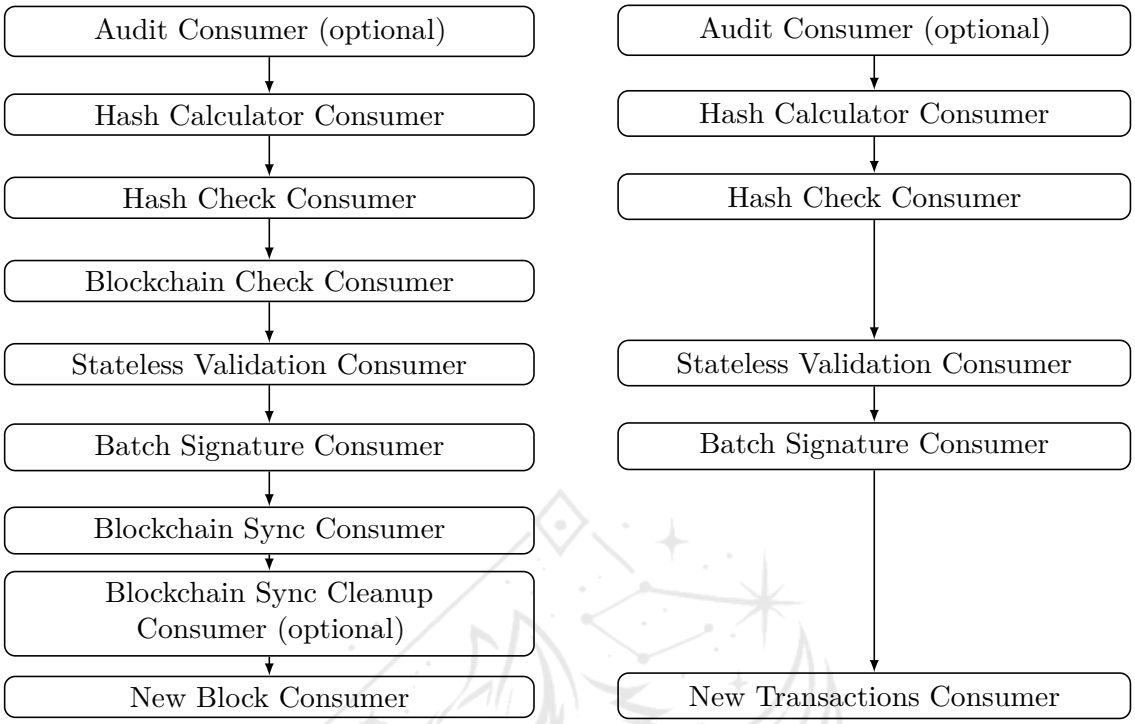
In **Symbol**, a block disruptor is used to process incoming blocks and blockchain parts. A blockchain part is an input element composed of multiple blocks. This disruptor is primarily responsible for validating, reconciling and growing the blockchain.

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

All disruptors are associated with a chain of consumers that perform all processing of their input elements. Different disruptors are customized by using different consumer chains. All consumers can inspect the data being processed and some can modify it.

CRYP2NOVA

Archived copy · cryp2nova.com



(a) Block Consumers (b) Transactions Consumers

Figure 20: **Symbol** consumer chains

9.1.1 Common Consumers

The block and transaction disruptors share a number of consumers.

Audit Consumer

This consumer is optional and can be enabled via node configuration. If enabled, all new elements are written to disk. This makes it possible to replay the incoming network action and is helpful for debugging.

Hash Calculator And Hash Check Consumers

It is very common for a server to receive the same element many times because networks consist of many servers that broadcast elements to several other servers. For performance reasons, it is desirable to detect at an early stage whether an element has already been processed in order to avoid processing it again.

Archived copy · cryp2nova.com

The hash calculator consumer calculates all the hashes associated with an element. The hash check consumer uses the hashes to search the recency cache, which contains the hashes of all recently seen elements. The consumer used by the transaction disruptor will also search the hash cache (containing hashes of confirmed transactions) and the unconfirmed transactions cache. If the hash is found in any cache, the element is marked as `CompletionStatus::Aborted` and further processing is bypassed.

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. Each plugin can add stateless validators.

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

Batch Signature Consumer

This consumer validates all signatures of all entities in an element. This is separate from the *Stateless Validation Consumer* because it uses batch verification. For improved performance, this consumer processes many signatures at once in a batch instead of individually. This can be done in parallel using many threads.

Reclaim Memory Consumer

This consumer completes the processing of an element and frees all memory associated with it. It triggers downstream propagation of the statuses of all transactions that were updated during processing. The overall result of the sync operation is used to update the reputation of - and possibly ban - the sync partner (see 13: Reputation)).

9.1.2 Additional Block Consumers

The block disruptor also uses a few block-specific consumers.

Blockchain Check Consumer

This consumer performs state-independent integrity checks of the chain part contained within an element. It checks that:

Archived copy · cryp2nova.com

- The chain part is not composed 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 chain part are linked.
- There are no duplicate transactions within the chain part.

Blockchain Sync Consumer

This consumer is the most complex one. All tasks that require or alter the local server's chain state are performed in this consumer.

First, it checks that the new chain part can be attached to the existing chain. If the chain part attaches to a block preceding the tail block, all blocks starting with the tail block are undone in reverse order until the common block is reached.

Next, it executes each block by performing stateful validation and then observing changes. Stateless validation is skipped because it was performed by previous consumers. If there are any validation failures, the entire chain part is rejected. Otherwise, all changes are committed to the chain state (both the block and cache storages) and the unconfirmed transactions cache is updated.

Finally, this consumer determines the last finalized block, which is the newest block that cannot be rolled back. When probablistic finalization is enabled, the last finalized block is *network:maxRollbackBlocks* blocks before the last block. When deterministic finalization is enabled, the last finalized block is the block referenced in the last finalization proof. The consumer prunes the global blockchain state of any and all data that is only needed to enable rollbacks to blocks before the last finalized block.

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

Blockchain Sync Cleanup Consumer

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

Archived copy · cryp2nova.com

New Block Consumer

This consumer forwards single blocks, either harvested by the server or pushed from a remote server, to other servers in the network.

9.1.3 Additional Transaction Consumers

The transaction disruptor uses a single transaction-specific consumer.

New Transactions 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're added to the unconfirmed transactions cache. Forwarding is intentionally done before stateful validation because one server might reject transactions that could be accepted by other servers (e.g. if the transaction has too low a fee for the local server). Subsequently, stateful validation is performed on the forwarded transactions, and the valid ones are stored in the unconfirmed transactions cache.

CRYP2NOVA

Archived copy · cryp2nova.com

10 Unconfirmed Transactions

> I don't believe one reads to escape reality. A person reads to confirm a reality he knows is there, but which he has not experienced.
>
> - *Lawrence Durrell*

Any transaction that is not yet included in a block is called an *unconfirmed transaction*. These transactions may be valid or invalid. Valid unconfirmed transactions are eligible for inclusion in a harvested block. Once a transaction is added to a block that is accepted in the blockchain, it is *confirmed*.

Unconfirmed transactions can arrive at a node when:

1. A client sends a new transaction directly to the node.
2. A bonded aggregate transaction is completed with all requisite cosignatures and is promoted from the partial transactions cache.
3. A Peer node broadcasts transactions to the node.
4. A Peer node responds to the node's request for unconfirmed transactions. As an optimization, the requesting node indicates what transactions it already knows in order to avoid receiving redundant transactions. Additionally, it supplies the minimum fee multiplier (see 8.3: Block Generation) it uses 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 arrives at a node, it is added to the transaction disruptor (see 9: Disruptor). All transactions that haven't been previously seen and pass stateless validation will be broadcast to peer nodes. At this point, it is still possible for the node to reject the broadcast transactions because stateful validation is performed after broadcasting. Due to different node settings, it's possible for some nodes to accept a specific unconfirmed transaction and other nodes to reject it. For example, the nodes could have different *node:minFeeMultiplier* settings.

10.1 Unconfirmed Transactions 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 *unconfirmed transactions cache*. This can fail for two reasons:

Archived copy · cryp2nova.com

1. The maximum cache size configured by *node:unconfirmedTransactionsCacheMaxSize* has been reached.

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 throttle.

Whenever new blocks are added to the blockchain, the blockchain state changes and the unconfirmed transactions cache is affected. Although all transactions in the cache are valid at the time they were added, this doesn't guarantee that they'll be valid in perpetuity. For example, a transaction could have already been included in a block harvested by another node or a conflicting transaction could have been added to the blockchain. This means that transactions in the cache that were perfectly valid previously could be invalidated after changes to the blockchain state. Additionally, when blocks with transactions are reverted, it's possible that some of those previously confirmed transactions are no longer included in any block in the new chain. Those reverted transactions should be added to the cache.

As a result of these considerations, the entire unconfirmed transactions cache is completely rebuilt whenever the blockchain changes. Each transaction is rechecked by the stateful validators and purged if it has become invalid or has already been included in a block. Otherwise, it is added back to the cache.

10.2 Spam Throttle

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

Simply limiting the number of unconfirmed transactions that a node accepts is suboptimal 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 are free to create.

Symbol implements a smart throttle that prevents an attacker from filling the cache completely with transactions while still letting honest actors successfully submit new unconfirmed transactions. *node:enableTransactionSpamThrottling* can be used to activate the throttle. Assuming the cache is not full, it works in the following way:

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

Archived copy · cryp2nova.com

2. If the new transaction is a bonded aggregate transaction, throttling is bypassed.
3. Else the Spam throttle is applied.

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

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

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 a transaction's maximum fee increases the number of slots available in the cache. Nonetheless, this mechanism for boosting the effective importance is limited by *node:transactionSpamThrottlingMaxBoostFee*.

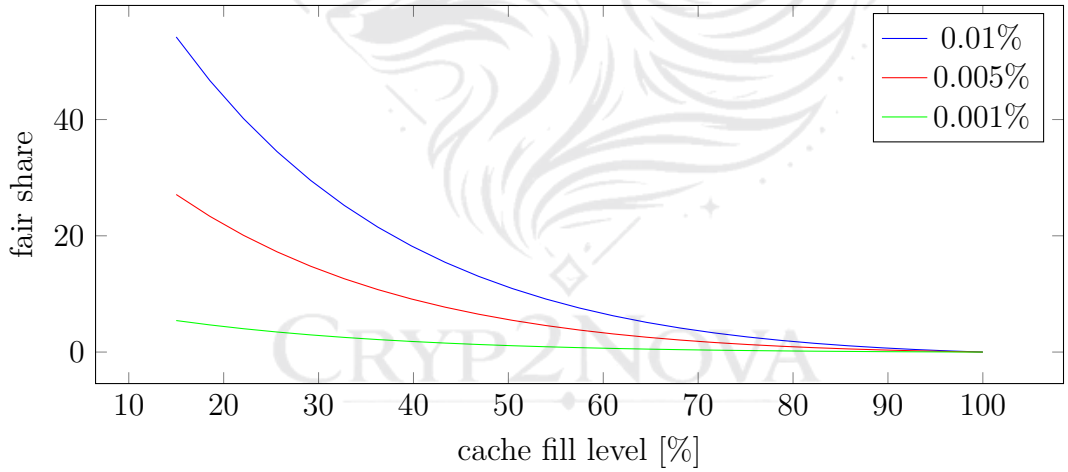


Figure 21: Fair share for various effective importances with max cache size = 10000

Figure 21 shows the fair share of slots relative to the fill level of the cache for various effective importances. An attacker that tries 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 maximum transaction fees but that will be more costly and expend funds at a faster rate.

Archived copy · cryp2nova.com

11 Partial Transactions

> The whole is greater than the sum of its parts.
>
> - *Aristotle*

BONDED aggregate transactions (see 6.2: Aggregate Transaction) are also referred to as partial transactions. The name *partial* is fitting because the transactions have insufficient cosignatures and are unable to pass validation until more cosignatures are collected.

Support for handling partial transactions is provided by the *partial transaction extension*. If a network supports bonded aggregate transactions, this extension should be enabled on all API and Dual nodes.

Partial transactions are synchronized among all nodes in a network that have this extension enabled. A node passively receives partial transactions and cosignatures pushed by remote nodes. It also periodically requests transactions and cosignatures from remote nodes via the *pull partial transactions task*. As an optimization, the requesting node indicates which transactions and cosignatures it already knows in order to avoid receiving redundant information.

When the hash lock plugin is enabled, in order for a partial transaction to be accepted by the network, a hash lock must be created and associated with the transaction. The hash lock is essentially a bond paid in order to use the built-in cosignature collection service. If the associated partial transaction is completed and confirmed in the blockchain before the hash lock expires, the bond is returned to the payer. Otherwise, the bond is forfeited to the harvester of the block at which the lock expires. This feature makes spamming the partial transactions cache more costly 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 pushed to the *partial transaction dispatcher*. Received cosignatures are collated with partial transactions already in the cache and are immediately rejected if there are no matching transactions [31]. When transactions and cosignatures are received together, they are split and processed individually as above.

Compared to the normal transaction dispatcher (see 9.1: Consumers), the partial transaction dispatcher is minimalistic.

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

Archived copy · cryp2nova.com

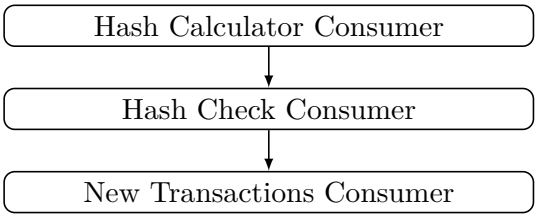


Figure 22: Partial transaction consumers

The hash calculator and hash check consumers work as described for the transaction dispatcher in 9.1.1: Common Consumers. The one difference is that the hash check consumer will additionally search the partial transactions cache for previously seen transactions. The new transaction consumer has a similar purpose to the one described in 9.1.3: Additional Transaction Consumers. The difference is that it broadcasts and processes partial transactions instead of unconfirmed transactions. Specifically, it broadcasts partial transactions to the network and then adds valid ones to the partial transactions cache.

11.1 Partial Transaction Processing

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

The partial transactions cache collates all new cosignatures with the transactions it already contains. In order to be added to the cache, a cosignature must be new, verifiable and associated with an existing partial transaction. It is possible for a previously accepted cosignature to become invalid, in which case it should be purged. For example, a cosignature could become invalid if its signer was removed from a multisignature account participating in the partial transaction. The cosignature collation process is intricate enough to handle such edge cases.

Archived copy · cryp2nova.com

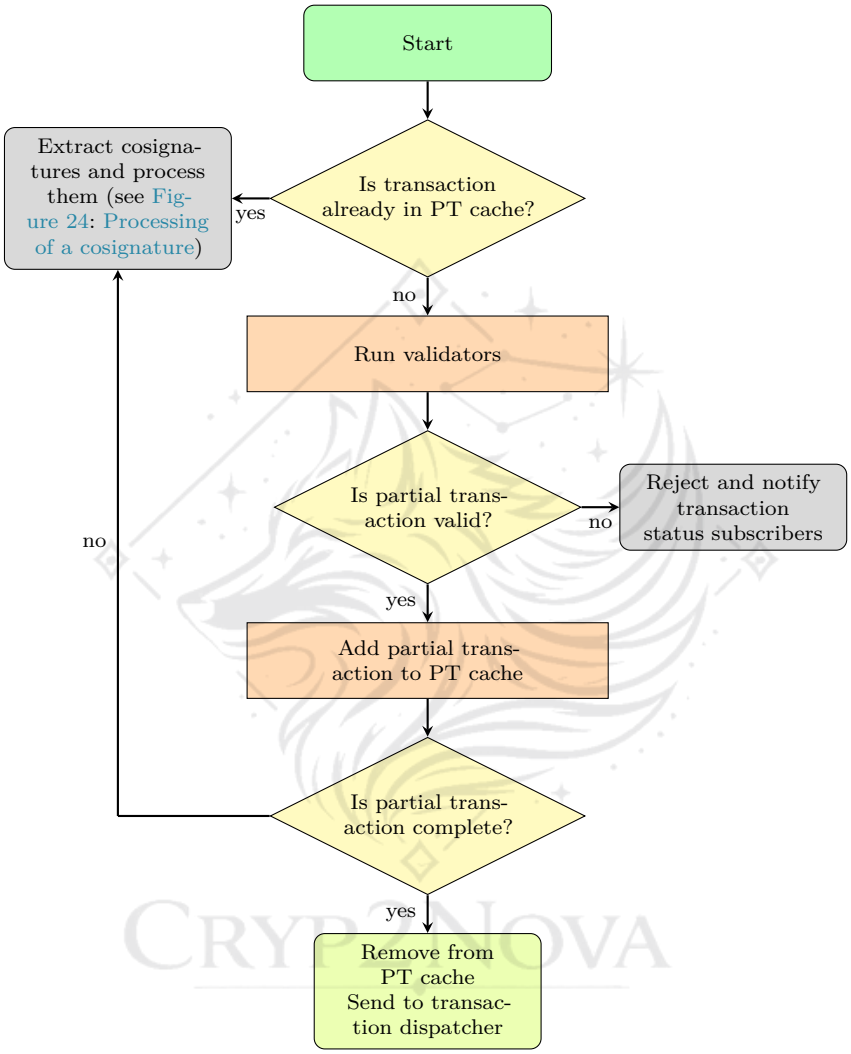


Figure 23: Processing of a partial transaction

Archived copy · cryp2nova.com

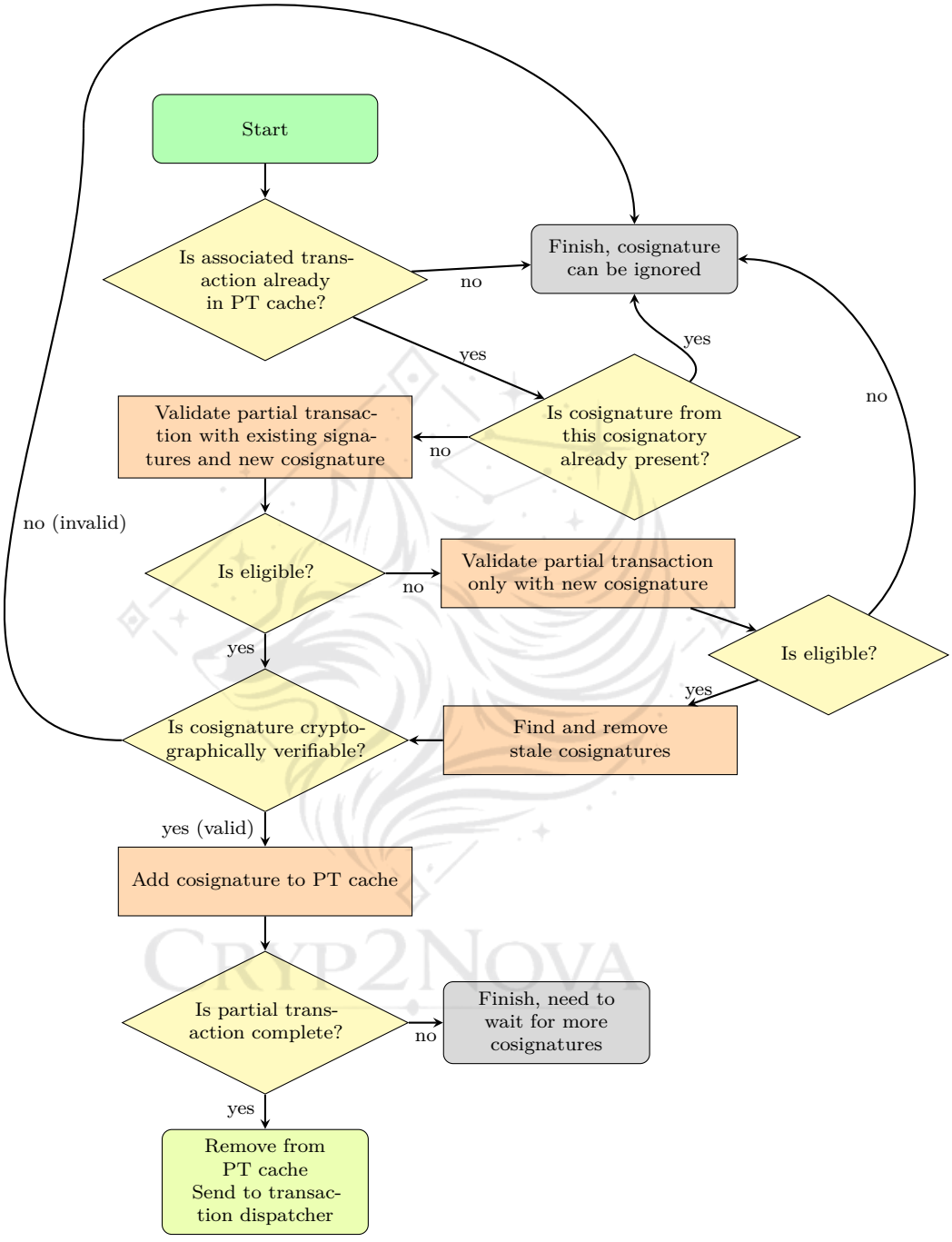


Figure 24: Processing of a cosignature

Archived copy · cryp2nova.com

12 Network

> Pulling a good network together takes effort, sincerity and time.
>
> - *Alan Collins*

DYNAMIC discovery of nodes allows a peer-to-peer network to grow. **Symbol** implements this dynamic discovery in the *node discovery extension*. Public networks are typically open and allow any node to join. Private networks can restrict the nodes that are allowed to join and behave as a federated system [32]. **Symbol** is flexible enough to even allow private networks to specify all node relationships via configuration files.

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

12.1 Beacon Nodes

A freshly booted node is initially isolated and not connected with any peers. It needs to join a network before it can make any meaningful contributions, like validating or harvesting blocks. In **Symbol**, a list of *static* beacon nodes are stored in a peers configuration file. In order to join a network, a new node first connects to these nodes. These files don't need to be identical across all nodes in a network.

A public network is recommended to specify a set of high availability beacon node candidates. Each node's peers 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 every boot. The important thing is that beacon nodes are well distributed. This reduces stress on individual beacon nodes and makes DoS attacks on beacon nodes more difficult. These nodes are given slight preference in node selection (see 13.2: Weight Based Node Selection) relative to non-beacon nodes because they are assumed to have high-availability. They aren't conferred any other special privileges or responsibilities. They can be thought of as doors into 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 peers configuration

[32] By carefully distributing harvesting and currency mosaics, a private network can delegate permissions to different accounts. For example, only accounts owning sufficient harvesting mosaic can create blocks and only accounts with nonzero currency can initiate transactions with nonzero fees.

[33] A node's resolved IP is only broadcast to other nodes when it doesn't specify a hostname. Hostnames are preferentially propagated in order to support nodes with dynamic IPs.

Archived copy · cryp2nova.com

file. Nodes with this extension enabled need to additionally synchronize partial transactions among other nodes that also have this extension enabled (see 11: Partial Transactions).

A node's roles specify the capabilities it supports. Typically, these are used by a connecting node to choose 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 finalization voting procedure when deterministic finalization is enabled. Nodes with the `IPv4` role support IPv4 communication. Nodes with the `IPv6` role support IPv6 communication[34]. Roles are not mutually exclusive. Nodes are allowed to support multiple roles.

12.2 Connection Handshake

All connections among **Symbol** nodes are made over TLS v1.3 with a custom verification procedure. Each node is expected to have a two level deep X509 certificate chain composed of a *root* certificate and a *node* certificate. All certificates must be X25519 certificates. **Symbol** doesn't support any other certificate types.

The root certificate is expected to be self-signed with an account's *signing private key*. This account is assumed to be a node's unique owner. Cryptographically linking a node and an account allows selection algorithms to perform node weighting based on the node owner's importance. Additionally, partner nodes use this verified identity to collate reputation (see 13: Reputation) information[35]. Importantly, this certificate is only used for signing the node certificate. For security, its private key should not be kept 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 for authenticating TLS sessions and deriving shared encryption keys for encrypting optional data[36]. It can be rotated as often as desired.

This authentication procedure is performed independently by each partner node. If either node fails the handshake, the connection is immediately terminated.

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

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

[36] Currently, the only encryption key derived is the one used to encrypt and decrypt messages related to automatic delegated harvester detection (see 8.6: Automatic Delegated Harvester Detection).

Archived copy · cryp2nova.com

12.3 Packets

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

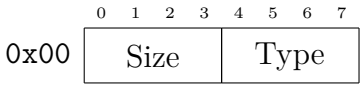


Figure 25: Packet header binary layout

A mix of long lived and short lived connections are used. Long lived connections are used for repetitive activities like syncing blocks or transactions. They support both push and request/response semantics. The connections are allowed to last for *node.maxConnectionAge* selection rounds (see 13.1: 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 (see 12.6: Node Discovery) and time synchronization (see 16: Time Synchronization). Short lived connections help prevent sync starvation, 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 specified packet type. When a complete packet is ready for processing, it is dispatched to the handler registered with its type. All handlers must accept matching packets for processing. Some handlers can also write response packets in order to allow request/response protocols.

12.4 Connection Types

In **Symbol**, long lived 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 identifier* [37]. This allows connections to be selected by capability and more granular logging.

[37] Although the terminology is similar, these are unrelated to services described in 2.2: **Symbol** Extensions.

Archived copy · cryp2nova.com

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

The *node:maxIncomingConnectionsPerIdentity* limit is applied across all services and long and short lived connections. Any incoming connections above this limit will be immediately closed. This limit can be hit when multiple short lived connections are initiated with the same remote node for different operations. This is more likely when connection tasks are more aggressively scheduled immediately after a node boots up. These errors are typically transient and can be safely ignored if they don't persist.

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

Service identifiers are only assigned to long lived connections. Sync service is used to manage outgoing connections to nodes with `Peer` role. API partial service is used to manage outgoing connections to nodes with `API` role. Readers service is used to manage incoming connections. API writers service is experimental and allows incoming connections on port *node:apiPort* to register as writers.

Identifier	Name	Direction
0x50415254	pt.writers	outgoing
0x52454144	readers	incoming
0x53594E43	sync	outgoing

Figure 26: Service Identifiers

For the purposes of node selection described in 13.2: Weight Based Node Selection, node aging and selection are both scoped per service. Reputational information is aggregated across all services. Specifically, assume a node has made both a Sync and API partial connection to another node. Each connection can have a different age because age is scoped per service. Interaction results, from any connection, are always attributed to the node, not the service.

Archived copy · cryp2nova.com

12.5 Peer Provenance

A node collects data about all nodes in its network. The reputability of the data is dependent on its provenance. Possible provenances, ranked from best to worst are:

1. Local - Node is specified in *node:localnode*.
2. Static - Node is in one of the peers configuration files.
3. Dynamic - Node was discovered and supports connections.
4. Dynamic Incoming - Node has made a connection but does not support connections.

It is important to note that the distinguishing characteristic of a *static* node is that it is listed in at least one local peers configuration file. For emphasis, it is possible for one partner to view a node as static while another partner views it as dynamic. Excepting the *local* node, all other nodes are *dynamic*. A subset of dynamic nodes are *incoming*. These nodes have only been seen in incoming but not outgoing connections. As a result, their preferred port is unknown and they can't be connected with.

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

The above is a slight simplification due to how connections are actually managed. When a node disconnects completely from a remote node and reconnects, the update can be a two step process. Consider a dynamic node that attempts 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 a worse provenance than dynamic. As a result, the remote will not update the node's information. Instead, it will set a flag indicating a possible identity update in progress. Later on, when the remote directly connects to the node, it will get the same updated information as before. At this point, the remote will update the information even though an active (dynamic incoming) connection is present because an in progress identity update was detected previously. Without this flag, the active connection with worse provenance would block updates, which is undesirable.

When *network:nodeEqualityStrategy* is `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 reputational migration.

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

Archived copy · cryp2nova.com

The resolved IP primary identity component can also be changed when there are no active connections assuming the secondary identity component is unchanged. In this case, all reputation data associated with the original host is migrated to the new host. When there is an ambiguous match, data with the matching primary identity component is migrated and data with the matching secondary identity component is discarded.

12.6 Node Discovery

After starting up, a node attempts to make short lived connections to all static nodes 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 hostnames to be used in the peers configuration files and simplifies node management. As long as the node is running, this procedure is periodically repeated with a linear backoff.

Periodically, a node will broadcast identifying information about itself to its remote partner nodes. The remote will process the received payload and check it for validity and compatibility. In order to be valid, the identity public key specified by the node must match the public key of its root X509 certificate. In order to be compatible, both the broadcasting and receiving nodes must target the same network. If no hostname is provided, the node's resolved IP will be used in place. If all checks succeed, 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. The remote nodes will respond with all of their active static and dynamic peers. To the requesting node, these will all be treated as dynamic nodes. The original node will request the identifying information from each of these nodes directly. This direct communication is required to prevent a malicious actor from relaying false information about other nodes and to ensure a connection can be established with each new node. The original node will process the received payload and check it for validity and compatibility as above. If all checks succeed, the new node will be added as a new potential partner and will be eligible for selection in the next sync round.

Archived copy · cryp2nova.com

13 Reputation

> It takes many good deeds to build a good reputation, and only one bad one to lose it.
>
> - *Benjamin Franklin*

Symbol uses a peer-to-peer (P2P) network. P2P networks have the great advantage of being robust because they cannot be shut down by eliminating a single node. Nevertheless, a public network comes with its own challenges. The participants of the network are anonymous and anyone can join. This makes it very easy to inject hostile nodes into the network that spread invalid information or try to disturb the network in some way.

There is a need to identify hostile nodes and reduce communication with them. There have been many approaches to achieve this. One of the most successful is building a reputation system for nodes. **Symbol** follows this approach by implementing a simple reputation system. This system attempts to prioritize connections to well-behaved nodes over those to misbehaving nodes. Importantly, reputation doesn't affect the blockchain consensus at all. It only influences the network graph. This chapter will outline the heuristics used.

13.1 Connection Management

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

When determining the nodes from which to disconnect, a node inspects the ages of all of its connections. In order 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 done, these connections are dropped and replaced with new connections to other nodes. This guarantees that over time each node will make connections to many different nodes in the network.

Archived copy · cryp2nova.com

13.2 Weight Based Node Selection

Nodes primarily communicate with each other via the current persistent connections they have established. A node can query another node for new transactions or blocks, or ask for a list of other nodes with which the partner node has interacted. Nodes can also voluntarily send data to other nodes. Each communication between nodes is considered an interaction and each interaction is scored as either 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 outcomes of its own interactions with other nodes. These outcomes are only used locally and not shared with other nodes. A node's interactions with other nodes influence the partner nodes it selects. Interaction results are stored for at most one week but reset on node restart. These results are time-limited to allow nodes that are having transient failures to reestablish themselves as good partners.

When selecting partner nodes, a node first determines a set of candidate nodes. Each candidate node is assigned a raw weight between 500 and 10000 according to the following criteria:

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

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

This formula guarantees that failed interactions rapidly decrease the weight of a remote node and its likelihood of getting selected. The presence of a minimum score still gives a node with many failures a slight chance for 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 (see 13.3: Node Banning), the multiplier is decreased by 1. This ensures that a node does not connect to dynamic banned nodes. The chance of connecting to static banned nodes is reduced by half.

Archived copy · cryp2nova.com

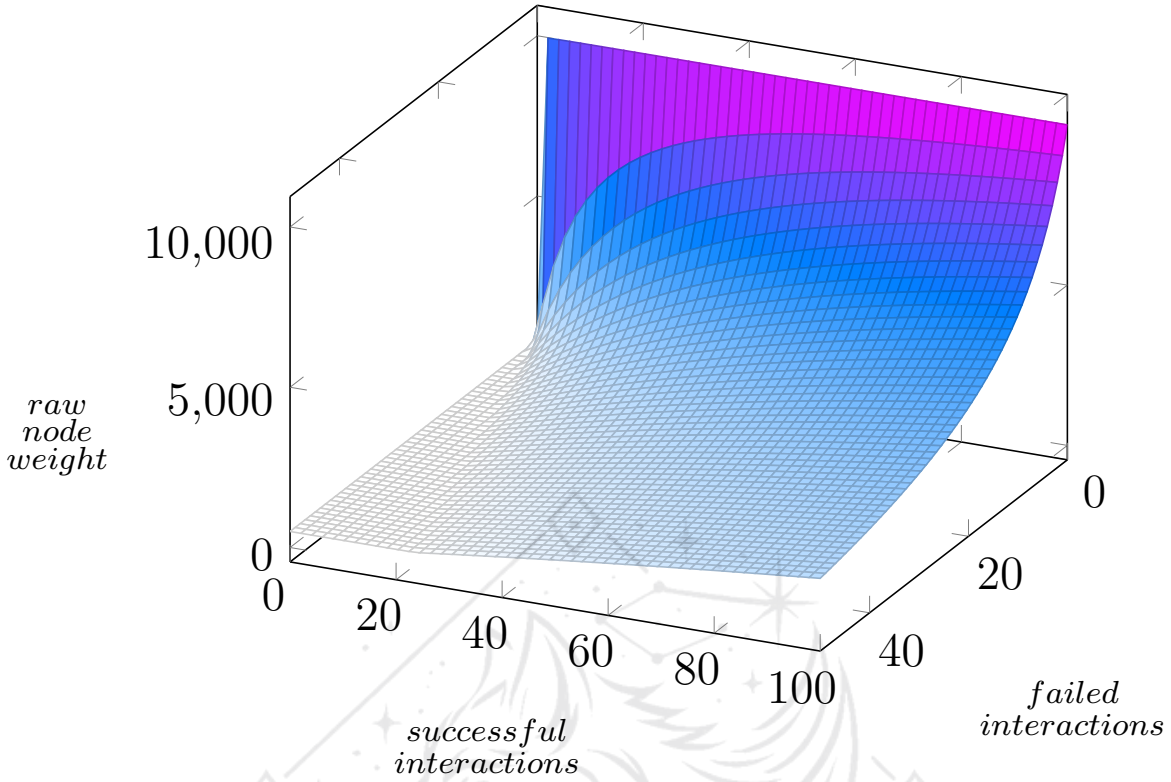


Figure 27: Raw Node Weight

Removal candidates are determined based on their connection age. Each removal candidate that will 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 slot, a candidate node has a chance of getting selected given by:

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

13.3 Node Banning

In a public network there could be potentially malicious nodes that try to disturb normal processing of the network. Therefore, if a node considers a remote node malicious, it will prevent connecting to that node and will not accept incoming connections from it.

Banning is applied at node level and is attached to a node's network scoped identifier

Archived copy · cryp2nova.com

(see 12.6: 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 behaving badly and treat repeat violations more severely by banning the node for longer periods up to *node:maxBanDuration*. During banning, no connections with the banned node will be established. After banning has expired, the node is treated like a normal interaction partner again. There are various scenarios where a remote node will get banned. The penalties vary based on the cause.

	Connection closed	Remote can reconnect	Remote can be selected	Remote can send data
Consecutive interaction failures	No	-	Yes Static No Dynamic	Yes
Invalid data	Yes - All[38]	No	No	No
Exceeded read rate	Yes - All	No	No	No
Unexpected data	Yes	Yes	Yes (after reconnect)	Yes (after reconnect)

Figure 28: Banning Rules

Consecutive Interaction Failures

If interactions with the same node fail for too many consecutive times due to networking or stateful failures, it is better to suspend all interactions with that node for some time, hoping the node will behave better in the future. The number of consecutive interaction failures before the node gets banned as an interaction partner can be configured. The amount of time the node gets banned is measured in selection rounds and can also be configured. While the node is banned, it will not be actively selected as an interaction partner, but it still can 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 triggering the violation.

Archived copy · cryp2nova.com

Invalid Data

Data can be invalid in many ways. For example, if a remote node is on a fork, it might send a new block that does not fit into the local node's chain. Little forks with a depth of one or two blocks happen frequently. Though the sent data is invalid, it is not considered malicious because the remote node's internal state 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 is true for other verification failures that do not depend on the state of a node. In all those cases, the remote node gets banned.

Exceeded Read Data Rate

Each node monitors the read rates of all sockets accepting data from 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.

Unexpectedly Receiving Data

There are situations during node communication where the local node is not expecting 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 immediately closed but there is no persistent banning of the node.

CRYP2NOVA

Archived copy · cryp2nova.com

14 Consensus

> You know what, sometimes it seems to me we're living in a world that we fabricate for ourselves. We decide what's good and what isn't, we draw maps of meanings for ourselves... And then we spend our whole lives struggling with what we have invented for ourselves. The problem is that each of us has our own version of it, so people find it hard to understand each other.
>
> - *Olga Tokarczuk*

Byzantine consensus is a key problem faced by all decentralized systems. Essentially, the crux of the problem is finding a way to get independent actors to cooperate without cheating. Bitcoin's key innovation was a solution to this problem that is based on Proof of Work (PoW). After each new block is accepted into Bitcoin's main chain, all miners begin a competition to find the next block. All miners are incentivized to extend the main chain instead of forks because the chain with the greatest cumulative hashing power is the reference chain. Miners calculate hashes as quickly as possible until one produces a candidate block with a hash below the current network difficulty target. A miner's probability of mining a block is proportional to the miner's hash rate relative to the network's total hash rate. This necessarily leads to a computational arms race and uses a lot of electricity.

Proof of Stake (PoS)[KN12][BCN13] blockchains were introduced after Bitcoin. They presented an alternative solution to the Byzantine consensus problem that did not require significant power consumption. Fundamentally, 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 stake in the network. Since richer accounts are able to produce more blocks than poorer accounts, this scheme tends to allow the rich to get richer.

Symbol uses a modified version of PoS that attempts to award *users* preferentially relative 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 else equal, accounts with larger stakes making more transactions and running nodes have more skin in the game and should be rewarded accordingly. Firstly, accounts with larger balances have larger stakes in the network and have greater incentives to see the ecosystem as a whole succeed. The amount of the currency an account owns is a measure of its stake. Secondly, accounts should be encouraged to use the network by making transactions. Network usage can be approximated by the total amount of transaction fees paid by an account. Thirdly, accounts should be encouraged to run nodes to strengthen the network. This can be

Archived copy · cryp2nova.com

approximated by the number of times an account is the beneficiary of a block [39]. Since the node owner has complete control over defining its beneficiary, any benevolent node owner can alternatively boost this measure for a third-party.

Importances are recalculated every *network:importanceGrouping* blocks. This reduces the pressure on the blockchain because the importance calculation is relatively expensive and processing it every block would be prohibitive. Additionally, recalculating importances periodically allows for automatic state aging. Overall, it is beneficial to calculate importances periodically rather than every block.

In order to encourage good behavior, accounts active in an older time period should not obtain an eternal advantage due to their previous virtuous behavior. Instead, importance boosts granted by transaction and node scores are time limited. The boost lasts for five *network:importanceGrouping* 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*. Notice that this set of accounts is a superset of the set of accounts eligible for block generation (see 8.3: Block Generation). In other words, a nonzero 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: stake, transaction and node.

The stake score, S_A, for an account A is the percentage of currency it owns relative to the total currency owned by all high value accounts. This percentage is no less than the percentage of currency the account owns relative to all outstanding currency. Let B_A represent the amount of currency owned by account A. The stake score for account A is calculated for each eligible account as follows:

$$S_A = \frac{B_A}{\sum_{a \in high\ value\ accounts} B_a} \tag{13}$$

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 represent the amount of fees paid by A in the time period P. The transaction score for account A is calculated for each eligible account as follows:

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

Archived copy · cryp2nova.com

$$T_A = \frac{\text{FeesPaid}(A)}{\sum_{a \in high\ value\ accounts} \text{FeesPaid}(a)} \tag{14}$$

The node score, N_A, 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 represent the number of times A has been specified as a beneficiary in the time period P. The node score for account A is calculated for each eligible account as follows:

$$N_A = \frac{\text{BeneficiaryCount}(A)}{\sum_{a \in high\ value\ accounts} \text{BeneficiaryCount}(a)} \tag{15}$$

Together, the transaction and node scores are called the *activity score* because they are both dynamic and derived from an account's activity as opposed to its stake. The transaction score is weighted at 80% and the node score at 20%. Additionally, the combined score is scaled relative to an account's balance so that there is a dampening effect of activity on importance as stake increases [40]. This effectively allows active smaller accounts to gain an outsized boost relative to active larger accounts. This partially redistributes importance away from rich accounts towards poorer accounts and somewhat counteracts the rich getting richer phenomenon inherent in PoS. The prominence of activity relative to stake can be configured by *network:importanceActivityPercentage*. When this value is zero, **Symbol** behaves like a pure PoS blockchain. Setting this to too a high value could weaken blockchain security by lowering the cost for an attacker to obtain 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 importance recalculations, new data is stored in a working bucket. At each importance recalculation, existing buckets are shifted, the working bucket is finalized and a new working bucket is created. Each bucket influences at most five importance recalculations. As a result, activity information quickly expires.

The *network:totalChainImportance* setting specifies the total importance that is distributed among all accounts in a network. Given that, the spot importance of the account

[40] The activity score is rescaled after dampening so that it contributes the desired *network:importanceActivityPercentage* to the importance calculation.

Archived copy · cryp2nova.com

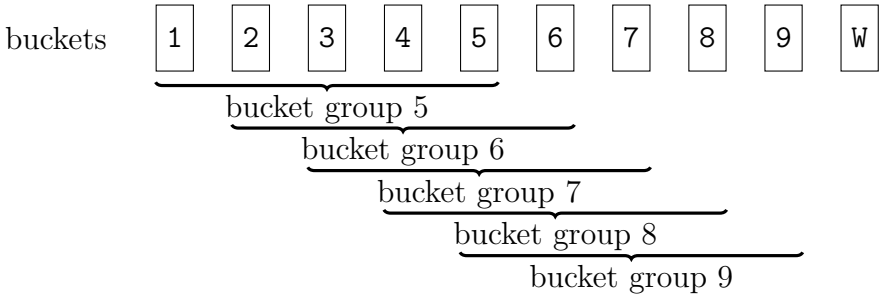


Figure 29: Activity buckets

A, I'_A, can be calculated as follows [41]:

$$\begin{aligned}
\gamma &= \text{importanceActivityPercentage} \\
\mathit{ActivityScore}'_A &= \frac{\text{minHarvesterBalance}}{B_A} \cdot (0.8 \cdot T_A + 0.2 \cdot N_A) \\
\mathit{ActivityScore}_A &= \frac{\mathit{ActivityScore}'_A}{\sum_{a \in \mathit{high\ value\ accounts}} \mathit{ActivityScore}'_a} \\
I'_A &= \text{totalChainImportance} \cdot ((1-\gamma) \cdot S_A + \gamma \cdot \mathit{ActivityScore}_A)
\end{aligned}$$

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

14.2 Sybil Attack

A *Sybil attack*[42] on a peer-to-peer network occurs when an attacker creates multiple identities in order to gain a disproportionately large influence over the network or some other advantage. In **Symbol**, an attacker might attempt such an attack to boost importance. Each component of the importance score needs to be robust against such attacks.

As described in 14.1: Weighting Algorithm, an account's activity score is dampened relative to its balance. Accordingly, splitting an account's balance among multiple accounts

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

[42] See additional discussion in 15: Finalization.

Archived copy · cryp2nova.com

will lower the average dampening factor applied. Assuming a constant level of activity is *sustained* before and after redistribution, the cumulative importance will be higher post split [43]. This effect is by design and encourages virtuous behavior because the importance boost is only realized if activity is sustained. Preservation of the transaction score encourages transacting and paying fees from multiple accounts. Preservation of the node score encourages running additional nodes and connecting them to the network.

Assume $\mu :=$ minHarvesterBalance and an attacker that owns $N \cdot \mu$ total currency. Consider two extremes:

1. The attacker has a single account with $N \cdot \mu$ currency.
2. The attacker has N accounts with μ currency.

Boosting Stake Score

In both extremes, the total currency owned by the attacker is the same. Accordingly, the stake score is the same and there is no benefit gained from splitting accounts. For emphasis:

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

Boosting Node Score

Symbol allows a node owner to specify a beneficiary for every block harvested on their node. Each time an account is specified as a beneficiary, assuming it is already a *high value account*, it will get a slight boost in its node score.

In both extremes, the total beneficiary count for the attacker is the same. Accordingly, the node score is the same and there is no undeserved benefit gained from splitting accounts. For emphasis:

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

The attacker might obtain a higher node score if running more nodes allows the attacker's nodes to host more delegated harvesters. This is not a bad outcome and by design. It

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

Archived copy · cryp2nova.com

encourages more nodes in the network, which is a good thing that strengthens the network.

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 be interacted with 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.

Boosting Transaction Score

The transaction score is solely based on fees. There is no difference between one huge account spending X on fees and N smaller accounts, each spending $\frac{X}{N}$ on fees. For emphasis:

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

The only possibility to boost transaction score is a *fee attack*, which is discussed in detail in 14.4: Fee Attack.

14.3 Nothing at Stake Attack

A general criticism of PoS consensus is the *nothing at stake attack*[44]. 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 submit a payment to one branch and immediately start harvesting on the other branch. Assuming the attacker has sufficient importance to harvest blocks, eventually the branch without the attacker's payment will become the reference chain because it will have a higher score [45]. The attacker's payment is not included in this branch, so the attacker's funds are effectively returned.

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

[44] See additional discussion in 15: Finalization.

[45] This assumes that there is only a single attacker or all attackers collude to withhold harvesting from the same branch.

Archived copy · cryp2nova.com

blocks can be rolled back. If the merchant waits to render services until at least this many blocks are confirmed, the attack is impossible. Second, in order to execute a successful nothing at stake attack, the attacker must own a significant importance in the network [46]. Third, successful execution of this attack against the network will likely have a negative influence on the currency value. Since other harvesters, by harvesting on all forks, enable this attack, profit-maximizing harvesters should only harvest on a single chain to preclude it.

In the second variation, a single attacker harvests on all forks and attempts to capture all fees irrespective of which chain becomes the reference chain. An attacker could harvest on all forks starting from the second block searching for the chain in which the attacker has harvested the most fees. Since block acceptance is probabilistic, in theory, an attacker could spend infinite time building the perfect chain in which the attacker has harvested all blocks.

Most theoretical nothing at stake attacks imagine an idealized blockchain and ignore protocol-level safeguards that protect against such attacks. In practice, this type of attack is impractical if the attacker owns a minority of currency. The aforementioned two defenses are also applicable here. In addition, changes in block difficulty (see 8.1: Block Difficulty) are capped at 5%. It will take some time for the difficulty of the attacker's chain to adjust downward, which will cause the block times at the beginning of the secret chain to significantly lag those of the main chain. These large time differences will make it unlikely for the attacker to produce a chain with a better score (see 8.2: Block Score).

A small amount of stake aging also decreases the likelihood of this second variation. Requiring accounts to have nonzero importances for two consecutive importance recalculations as a precondition for harvesting makes generation hash grinding[47] attacks nonviable. In order to exploit this, the attacker would need to move all currency to a specific account more than *network:importanceGrouping* blocks before the attack could be carried out. Since the attacker can't know all the blocks that will be confirmed in the intervening period, such movement cannot result in any benefit.

CRYP2NOVA

14.4 Fee Attack

A *fee attack* is an attempt by an attacker to exploit the transaction score by paying large fees in order to boost its own importance. The attack is considered effective if it yields a

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

[47] This is an attempt to brute force the block hit (see 8.3: Block Generation), which is dependent on generation hash.

Archived copy · cryp2nova.com

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:e*qual to 10000 currency. Additionally, *network:importanceActivityPercentage* is 5, so the cumulative transaction score (see Equation 14.1: Weighting Algorithm) accounts for 4% of importance.

Large Account

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

The account might try to make a profit by adding a transaction with a high fee to one of its own harvested blocks each recalculation interval. This would boost the account's importance and allow it to harvest more blocks in the future and, consequently, collect more fees. However, this activity is not risk free. The account risks paying the high fee if a better block replaces its block. When the original block is unwound, the high fee transaction will enter the unconfirmed transactions cache and be eligible for inclusion in a new block created by a different harvester. This scenario is a net loss because the account will have to pay the high fee.

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, can be approximated as follows:

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

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

The expected value is positive for small values of P. As P or F increases, it quickly becomes negative. Using the recommended public network settings, P needs to be less than 0.0001 for the expected value to be positive. This implies a fork resulting in a loss occurs less than once every 10000 blocks. Given the mechanism of distributed consensus, this is a near impossibility. Small one or two block forks occur quite frequently.

Archived copy · cryp2nova.com

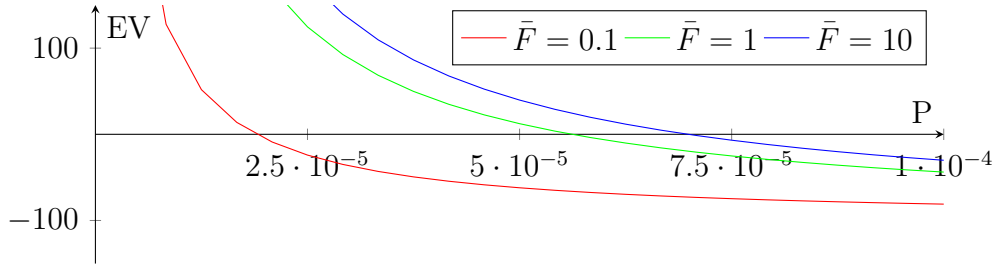


Figure 30: Fee attack large account analysis (F = 100)

Small Account

Consider an account that has a balance equal to *network:minHarvesterBalance*. Assume the account makes one transaction with high fees in two consecutive recalculation intervals. These fees are lost to other harvesters because the probability of the account harvesting a block is quite small. The high fees paid boost the account's importance enough so that it is able to harvest at least one block per importance recalculation interval. From this point forward, the account behaves like the large account in the previous section. It will also add a transaction with a high fee to one of its own harvested blocks each recalculation interval. The account owner hopes that due to the increased probability of harvesting a block, its additional collected fees will exceed its costs.

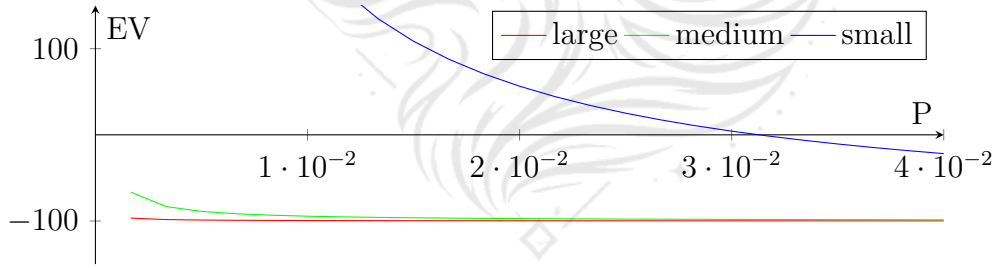


Figure 31: Fee attack balance sensitivity (F = 100, $\bar{F} = 1$)

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[48]:

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

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

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

Archived copy · cryp2nova.com

The expected value is positive for larger values of P than in the large account scenario. A fee attack confers an outsized benefit to a small account relative to a large account because the activity scores of the latter are dampened more aggressively than those of the former. Specifically, a damping factor of $\frac{1}{557}$ is applied to the large account's activity score, but no damping factor is applied to the small account's 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.

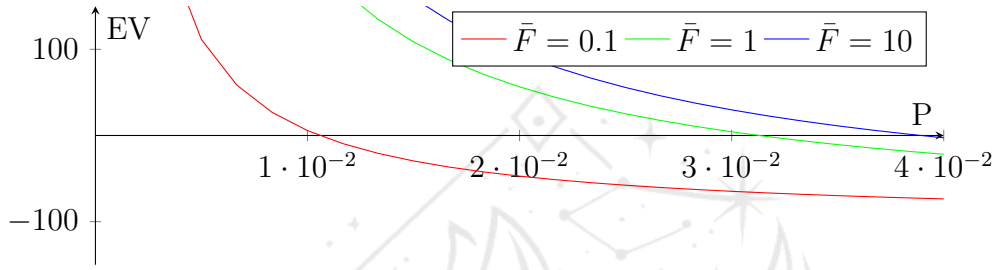


Figure 32: Fee attack small account analysis (F = 100)

Further Discussion

Although a single small account can obtain a positive expected value by executing this attack, the payoff decreases as multiple accounts attempt it simultaneously. Since there is a positive expected value, profit maximizing actors should all attempt this attack. As more accounts attempt it, the importance boost obtained by each individual account decreases rapidly and, consequently, the expected value also decreases.[49].

Additionally, there is an upper limit on the number of small accounts that can execute this attack simultaneously. In order for this attack to be successful, an account needs to be able to harvest at least one block per importance recalculation interval. This presupposes the small account can boost its importance score by exploiting the transaction score component. There is a theoretical limit on the number of accounts that can achieve a significant enough boost because both the importance allotted to the transaction score and the recalculation interval are finite. Considering the recommended public network settings, this limit is approximately $0.04 \div \frac{1}{359} \approx 14.36$ accounts.

[49] As more accounts produce high fee transactions to attempt this attack, $\bar{F}$ increases. For large numbers of attackers, if F is not raised in proportion, the expected value of the attack can increase even though the importance boost per account decreases. This is an expected outcome since the value of blocks also increases significantly.

Archived copy · cryp2nova.com

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:

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

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

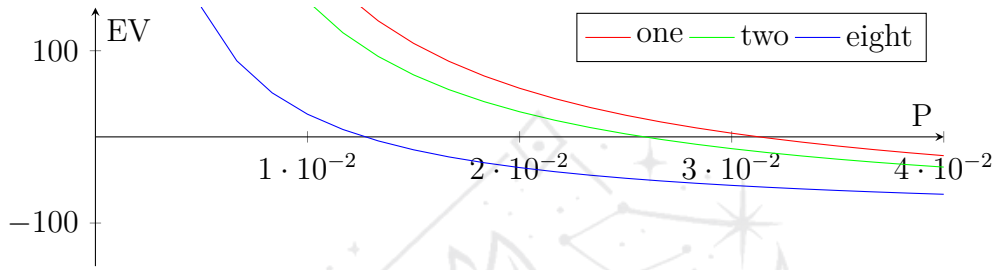


Figure 33: Fee attack declining with more attackers (F = 100, $\bar{F} = 1$)

CRYP2NOVA

Archived copy · cryp2nova.com

15 Finalization

> “And it feels like, finally.”
>
> - *Patrick Ness*

The CAP Theorem posits that, in the presence of a network failure or partition, a distributed system must choose either *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 stalls. A naive consistent system could require all stakeholders to vote on each block and only allow the next block when $\frac{2}{3}$ of stakeholders approve. If stakeholders fail to 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 returned values can be different, so different clients could have different views of the system. PoW and NXT-style PoS systems typically chose availability and eventual consensus. They risk propagating different (potentially unresolvable) views of the system. Available PoW-like systems, like Bitcoin, have a simple fork rule that chooses the chain with the most work. If the network is split into partitions with equal hash power, all partitions will proceed independently without knowing they're partitioned. When they are eventually reconnected, there will be an expensive (and potentially deep) fork resolution.

Symbol always prefers availability over consistency, but it optionally supports the use of a finality gadget on top of its native consensus (see 14: Consensus). This gadget introduces a BFT-inspired voting system that is orthogonal to block production and block consensus. As a result of its optionality, **Symbol** is able to support networks that use either deterministic finalization (when the gadget is enabled) or probabilistic finalization (when the gadget is disabled). The gadget is automatically enabled when *node:maxRollbackBlocks* is zero[50].

The gadget approach is modeled after GRANDPA [SK20] used by Polkadot. GRANDPA, in turn, in influenced by CASPER [BG17], which itself is influenced by PBFT [CL99]. Traditionally, PBFT uses three types of messages: pre-prepare, prepare and commit. Pre-prepare messages, which are used to start rounds, are not used in **Symbol** . Instead, elapsed network time (see 16: Time Synchronization) is used to start rounds. Prepare and commit messages in PBFT roughly correspond to prevote and precommit messages in **Symbol** .

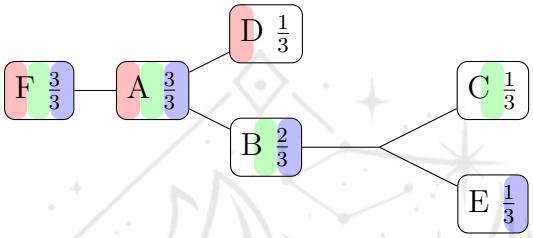
[50] The finalization extension must also be enabled.

Archived copy · cryp2nova.com

15.1 High Level Overview

Block finalization is a complicated process that involves two types of messages: prevotes and precommits. At the beginning of a round, each voter only knows the blocks stored in its local chain. No voter knows the blocks stored in the chains of any other voter. Consequently, it is impossible for a voter to know which of its blocks will receive a supermajority of votes and be finalized ahead of time.

To illustrate the general procedure of finalization, consider a network composed of three equally weighted voters. A supermajority requires at least two of the three voters to vote for the same hash. F refers to the last finalized block.



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

Finalization is greedy, and it attempts to finalize as many blocks as possible each round. In the first *prevote stage*, each voter constructs and publishes a hash chain representing its local chain starting with the hash of the last finalized block (F). In this example, the prevote chains 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)

Assume that `red`, due to bad network connections, only receives the prevote from `green` but not `blue`. Assume that `green` and `blue` receive prevotes from all voters. At this point, each voter has the following prevote chains:

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

Archived copy · cryp2nova.com

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 prevote hash chains to calculate the best block that **might** be finalized this round. `red` only sees a supermajority for A but `green` and `blue` see a supermajority for B. In the next *precommit stage*, each voter publishes the hash of its calculated best block:

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

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

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. Importantly, a precommit for a block is also a precommit for all of the block's ancestors. `green` only sees a supermajority for A but `red` and `blue` see a supermajority 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 finalization algorithm in more detail.

Archived copy · cryp2nova.com

15.2 Rounds

A *finalization round* represents a step in the finalization process and is composed of a *finalization epoch* and a *finalization point*.

An epoch is a group of blocks. All blocks within an epoch are finalized by a single voting set, which is recalculated every *network:votingSetGrouping* blocks. The first epoch is defined as exclusively containing the nemesis block and is inherently considered to be finalized. Subsequent epochs must end at a block with a height that is a multiple of *network:votingSetGrouping*. An epoch is considered *finalized* when all blocks within it are finalized. There will always be a finalization proof available for the last block within an epoch.

A point is a fine-grained step within an epoch that represents progress towards finalization of that epoch. Each point can finalize zero or more blocks. A point can only finalize blocks within its parent epoch. This prevents any block from being potentially finalized by multiple voting sets, which could lead to a safety violation. There is no limit on the number of points associated with an epoch. There will be as many points as necessary to finalize all blocks within an epoch. The last point within an epoch will always finalize that epoch's last block. A block is considered *finalized* when a point finalizes that block or any of its descendants.

When a network is partitioned, it is possible for a point to finalize zero additional blocks. This happens when only the last finalized block receives a supermajority of votes. This is not a fatal error and can occur naturally in the presence of network partitions. Afterwards, the finalization procedure will continue and advance to the next point based on the conditions described in 15.5: Algorithm.

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

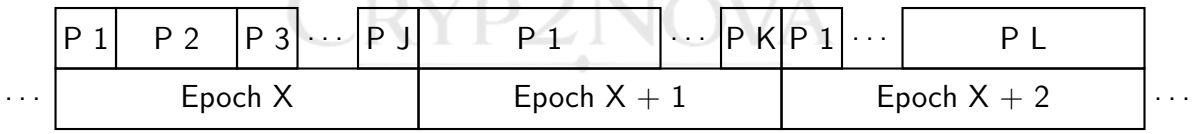


Figure 34: Epoch and (P)oint relationship

15.3 Voters

An account is eligible to vote in a epoch if all of the following are true:

Archived copy · cryp2nova.com

1. Harvesting balance at the last finalized block of the previous epoch is no less than the network defined *network:minVoterBalance*.

2. Voting key is registered such that `StartEpoch` $\leq$ epoch $\leq$ `EndEpoch`.

A voting set for an epoch is the set of all accounts that satisfy the previous conditions. Importantly, only balance, not importance, is used when weighting votes. This prevents a safety violation that could occur when using importances[51].

All accounts eligible to vote are expected to vote. A well-behaved voter is expected to vote in all rounds where it is eligible and not send multiple votes per round. It is implicitly assumed that all voters will run high-availability nodes and turn on *finalization: EnableVoting*. Voters that violate these expectations are considered Byzantine and might be punished in the future. Votes are weighted proportionally to balance such that the votes from voters with higher balances have more impact.

15.4 Messages

Prevote and precommit messages share a common layout.

[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's theoretically possible for multiple partitions to have supermajorities of importance and finalize conflicting blocks.

CRYP2NOVA

Archived copy · cryp2nova.com

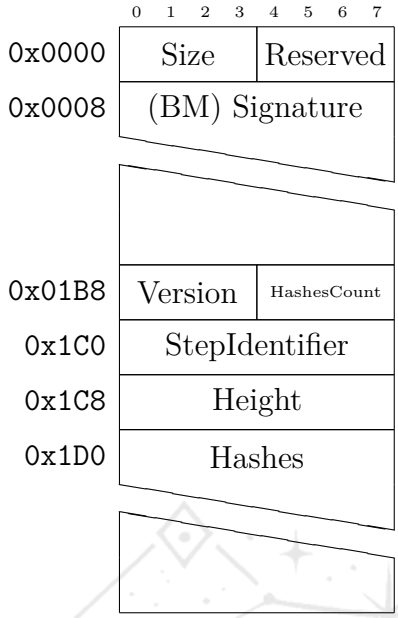


Figure 35: Message layout

`Signature` is the *BM tree signature* (see 3.4: Voting Key List) of the message. The root public key must match the voting key registered for the message's epoch. The bottom public key must match the public key in the tree corresponding to the epoch.

`Version` is the message version.

`Hashes` contains `HashesCount` hashes. A prevote message will contain at least one hash and up to *finalization:maxHashesPerPoint*. A precommit message will always contain exactly one hash.

`StepIdentifier` indicates the finalization round of the message. The high bit of the point is reserved to indicate the type of message where 0 indicates prevote and 1 indicates precommit.

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

15.5 Algorithm

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

Archived copy · cryp2nova.com

estimate and a greedy one at that. $E_{r,v}$ must be the latest block on the chain containing $g(V_{r,v})$ that can receive a supermajority of $C_{r,v}$. A round is *completable* when either:

1. $E_{r,v} < g(V_{r,v})$
2. It is impossible for any child of $g(V_{r,v})$ to have a supermajority of $C_{r,v}$

A voter v can begin a round r when the previous round is completable, and it has cast votes in all previous rounds where it was eligible.

15.5.1 Prevote

When either $1 * \text{stepDuration}$ has elapsed or r is completable, v sends a prevote.

A voter determines the best block that can potentially be finalized. It creates a prevote message composed of all hashes starting with the hash of the last (local) finalized block. The prevote message hash chain will contain at most *finalization:maxHashesPerPoint* hashes. The last hash in the chain will correspond to a block with a height that is a multiple of *finalization:prevoteBlocksMultiple*[52]. This increases the probability that nodes will send prevote messages with identical chains that can be aggregated more aggressively.

Hashes corresponding to unfinalized blocks are prohibited from spanning epochs. This guarantees that there is exactly one voting set that can finalize a block at any height. This property enables dynamic voting sets.

Prevote messages include hash chains, instead of single hashes, because each **Symbol** node stores a single block chain instead of a block tree. The aggregation of hash chains allows **Symbol** to reconstruct a virtual block tree and apply votes to both seen and unseen branches.

Conceptually, a voter has one vote per height. Effectively, it votes with its weight on every hash in the prevote hash chain.

15.5.2 Precommit

Next, a voter waits until $g(V_{r,v}) \geq E_{r-1,v}$. When either $2 * \text{stepDuration}$ (relative to the start of the round) has elapsed or r is completable, v sends a precommit.

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

Archived copy · cryp2nova.com

A voter determines the best block that can potentially be finalized. It creates a precommit message with a single hash corresponding to $g(V_{r,v})$.

Conceptually, a voter has one vote per height. Effectively, it votes with its weight on every hash between the last finalized block and the precommit hash.

15.5.3 Commit

Asynchronously, a voter collects prevote and precommit messages for round r. In practice, these messages will be associated with either the current round or the previous round. When $g(C_{r,v})$ changes, that block as well as all blocks between that block and the local finalized block are finalized.

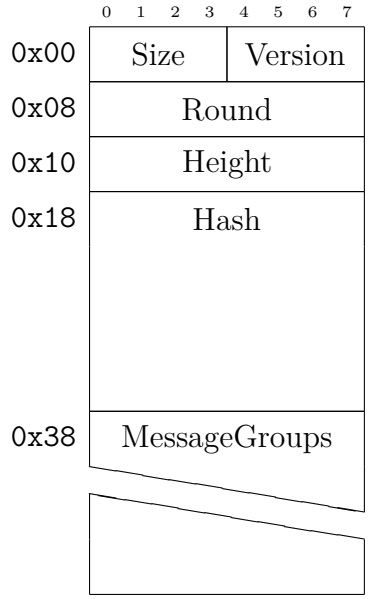
Given a finalization round with epoch e and point p, in most cases, the next round will be $(e, p+1)$. The one exception to this is when the last block of epoch e is finalized. In that case, the next round is $(e+1, 0)$. Notice that it is possible for both $(e, p+1)$ and $(e+1, 0)$ to be started. In that scenario, $(e+1, 0)$ will eventually dominate and $(e, p+1)$ will not complete.

15.6 Proofs

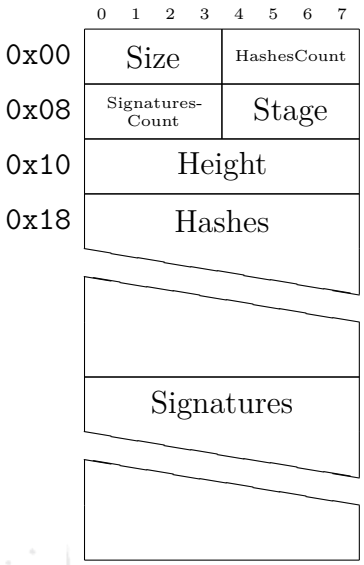
In order to minimize network traffic, non-voters do not send or receive individual prevote or precommit messages. Instead, these nodes pull and individually verify finalization proofs from the network periodically based on *finalization:unfinalizedBlocksDuration*. When this setting is zero, proofs are only pulled at the end of each epoch. When this setting is nonzero, proofs are optimistically pulled whenever the last finalized block is at least *finalization:unfinalizedBlocksDuration* behind the last unfinalized block. Upon verification, these non-voting nodes finalize all epochs up to and including the proof epoch.

CRYP2NOVA

Archived copy · cryp2nova.com



(a) Proof header



(b) Message group

Figure 36: Finalization proof layouts

A finalization proof is composed of a header and a collection of message groups. The header indicates last block finalized by the proof[53]. This block is uniquely identified by its height and hash. The header also indicates the finalization round at which the block was finalized.

A message group is an aggregation of finalization messages that cryptographically verify the proof. All finalization messages that differ only in `Signature` are grouped together into a single message group with all signatures appended. There will always be at least two message groups in a proof - one for prevote and one for precommit - but there can 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 $g(C_{r,v})$ and $g(V_{r,v}) \geq g(C_{r,v})$.

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

Archived copy · cryp2nova.com

15.7 Sybil Attack

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

15.8 Nothing at Stake Attack

When deterministic finalization is enabled, the *nothing at stake attack* can be prevented if a merchant waits to render services until a block is finalized. If a merchant does not wait, there are additional defenses against this attack.

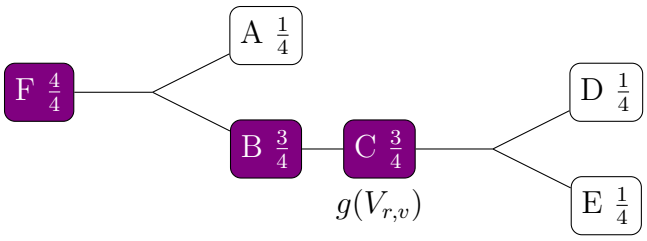
Most significantly, the attacker has a limited amount of time to produce a better chain. The attacker must generate a better chain before the network finalizes the block that the attacker wants to rollback. Additionally, the time required for a significant drop in difficulty will likely be longer than the time it takes to finalize a given chain part.

15.9 Examples

In all the examples, consider a network with four equally weighted voters. A supermajority requires at least three of the four voters to vote for the same hash. F always refers to the last finalized block. A prevote message cast for a block B implies a prevote hash chain composed of hashes from F to B, inclusive.

Example 1

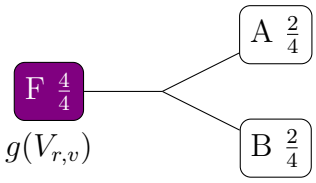
Prevote messages are cast for A, C, D, E. Since C is on the branch of D and E, it has a supermajority of votes even though only one voter voted for it explicitly.



Archived copy · cryp2nova.com

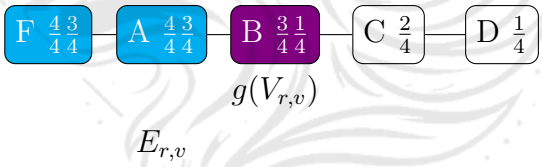
Example 2

Consider a network split into two equally sized partitions. Prevote messages are cast for A, A, B, B. Since F is on the branch of A and B, it has a supermajority of votes even though no voter voted for it explicitly. Importantly, notice that no new blocks are finalized because F is already finalized.



Example 3a

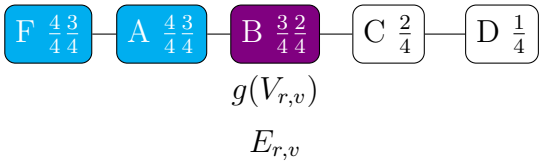
Prevote messages are cast for A, B, C, D so that B has a supermajority of prevotes. Assume two voters see prevotes A, B, C and send precommits for A. Assume one voter sees prevotes B, C, D and sends a precommit for B. A voter can determine that the weight of unknown precommits (25%) cannot cause a supermajority of precommits for B, which only has 25% of precommits. This satisfies condition 1 in 15.5: Algorithm.



Example 3b

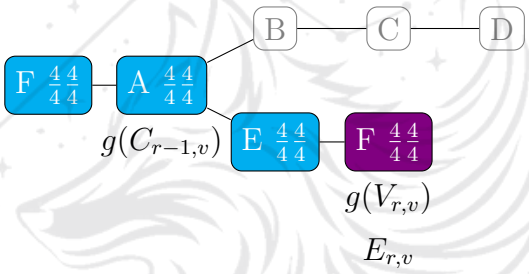
Consider a slight modification of the previous example where prevotes are received in a different order. As above, prevote messages are cast for A, B, C, D so that B has a supermajority of prevotes. Assume two voters see prevotes B, C, D and send precommits for B. Assume one voter sees prevotes A, B, C and sends a precommit for A. A voter can determine that the weight of unknown precommits (25%) cannot cause a supermajority of precommits for C, which only has 0% of precommits. This satisfies condition 2 in 15.5: Algorithm.

Archived copy · cryp2nova.com



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 to the next point r. Four prevote messages and four precommit messages are cast for F. Blocks B, C, D have all been pruned from the main chain even though precommits for B were used to commit A at $r-1$. In order to verify the proof at $r-1$, there needs to be a record that B is a descendant of A. This information is stored in the prevote message hash chains but not in the precommit messages. This is why the verification of finalization proofs requires prevotes.



CRYP2NOVA

Archived copy · cryp2nova.com

16 Time Synchronization

> Time and Tide wait for no man.
>
> - *Geoffrey Chaucer*

Like most other blockchains, **Symbol** relies on timestamps for the ordering of transactions and blocks. Ideally, all nodes in the network should be synchronized with respect to time. Even though most modern operating systems have time synchronization integrated, 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, which makes it impossible for them to synchronize with the network.

It is therefore needed to have a synchronization mechanism to ensure 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 near real time. This has the disadvantage that the network relies on servers outside the network.

Using a custom protocol that only relies on the P2P network itself solves this problem, but there is a trade off. It is impossible to guarantee that the network time is always near real time. For an overview of different custom protocols see [Sci09]. **Symbol** uses a custom protocol based on Chapter 3 of this thesis in order to be completely independent from any outside entity. The protocol is implemented in the timesync extension.

16.1 Gathering Samples

Each node in the network manages an integer *offset* that is set to 0 at start. 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 (hereafter called *local node*) selects partner nodes for performing a time synchronization round.

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

[54] https://en.wikipedia.org/wiki/Network_Time_Protocol

Archived copy · cryp2nova.com

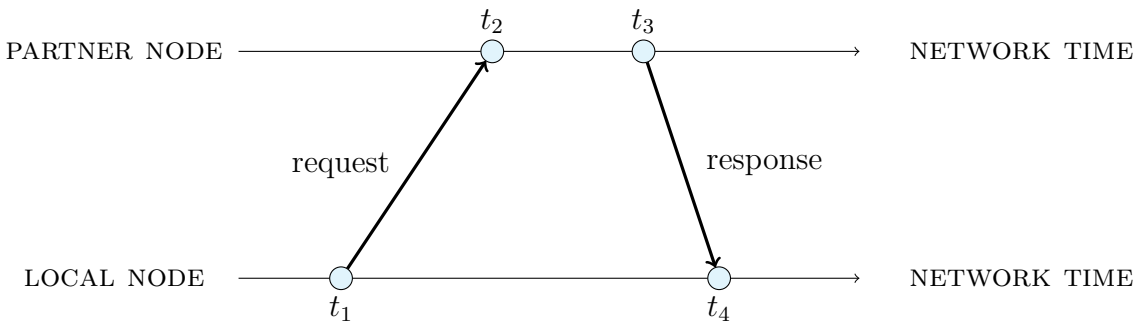


Figure 37: Communication between local and partner node.

request was sent and when each response was received. Each partner node responds with a sample that contains 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 37 illustrates the communication between the 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 every time synchronization partner until the local node has a list of offset estimations.

16.2 Applying Filters to Remove Bad Data

There could be bad samples due to various reasons:

- A malicious node supplies 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 highly asymmetric 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 try to remove the bad samples. The filtering is done in three steps:

Archived copy · cryp2nova.com

1. If the response from a partner is not received within an expected time frame (i.e. if $t_4 - t_1 > 1000ms$) the sample is discarded.

2. If the calculated offset is not within certain bounds, the sample is discarded. The allowable bounds decrease as a node's uptime increases. When a node first joins the network, it tolerates a high offset in order to adjust to the existing consensus of network time within the network. As time passes, the node gets less tolerant with respect to reported offsets. This ensures that malicious nodes reporting huge offsets are ignored after some time.

3. The remaining samples are ordered by their offset and then alpha trimmed on both ends. In other words, on both sides a certain portion of the samples is discarded.

16.3 Calculation of the Effective Offset

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 offset o

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

Archived copy · cryp2nova.com

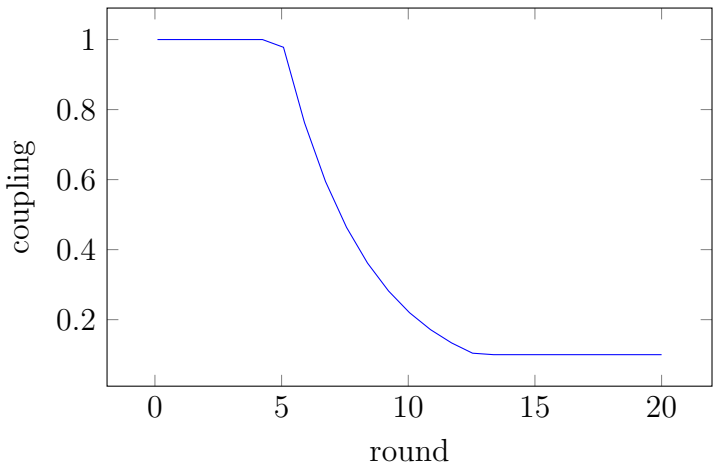


Figure 38: Coupling factor

Note that the influence of an account with large importance is artificially limited because the term $\frac{n}{s}$ caps the scale. Such an account can raise its influence on a macro level by splitting its balance into accounts that are not capped. But, doing so will likely decrease its influence on individual partners because the probability that all of its split accounts are chosen as time-sync partners for any single node is low.

16.4 Coupling and Threshold

New nodes that just joined the network need to quickly adjust their offset to the already established network time. In contrast, old nodes should behave a lot more rigid in order to not get influenced by malicious nodes or newcomers too much.

In order to enable this, nodes only adjust a portion of the reported effective offset. Nodes multiply the effective offset with a coupling factor to build the final offset.

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

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 calculated final offset to its internal offset if the absolute value is above a given threshold (currently set to 75ms). This is effective in preventing

Archived copy · cryp2nova.com

slow drifts of the network time due to the communication between nodes having channel asymmetry.

Archived copy · cryp2nova.com

17 Messaging

> A good message will always find a messenger.
>
> - *Amelia Barr*

Blockchain client applications retrieve blockchain data and present it to their users. In order for these clients to be most useful, they should always present the most up to date blockchain data and refresh their user interfaces whenever the displayed data changes. A naive client could periodically poll a REST server or a local database for blockchain data. This is inefficient because it requires using more network bandwidth and other resources than necessary. Instead, **Symbol** allows clients to subscribe to data changes via a single message queue.

17.1 Message Channels and Topics

The **Symbol** message queue exposed to clients supports multiple channels. Each channel has a unique *topic*. A topic always starts with a topic marker that indicates the kind of messages that will be received. In some cases, the marker is followed by an unresolved address that is used for additional filtering. Since a client is usually not interested in every type of blockchain state change, it can subscribe to a subset of available topics. Figure 39 lists all supported topic markers.

Topic marker name	Topic marker
Block	`0x9FF2D8E480CA6A49`
Drop blocks	`0x5C20D68AEE25B0B0`
Finalized block	`0x4D4832A031CE7954`
Transaction	`0x61`
Unconfirmed transaction add	`0x75`
Unconfirmed transaction remove	`0x72`
Partial transaction add	`0x70`
Partial transaction remove	`0x71`
Transaction status	`0x73`
Cosignature	`0x63`

Figure 39: Topic Markers

Archived copy · cryp2nova.com

17.2 Connection and Subscriptions

Support for messaging is added by the zeromq extension. If a node wants to support messaging, this extension must be enabled in the broker process. The extension registers subscribers for block and transaction related events (see 2.2: **Symbol** Extensions) and maps those events to message queue messages. When enabled, the broker listens on port *messaging:subscriberPort* for new subscribers. Clients can connect and subscribe to the message queue for one or more topics.

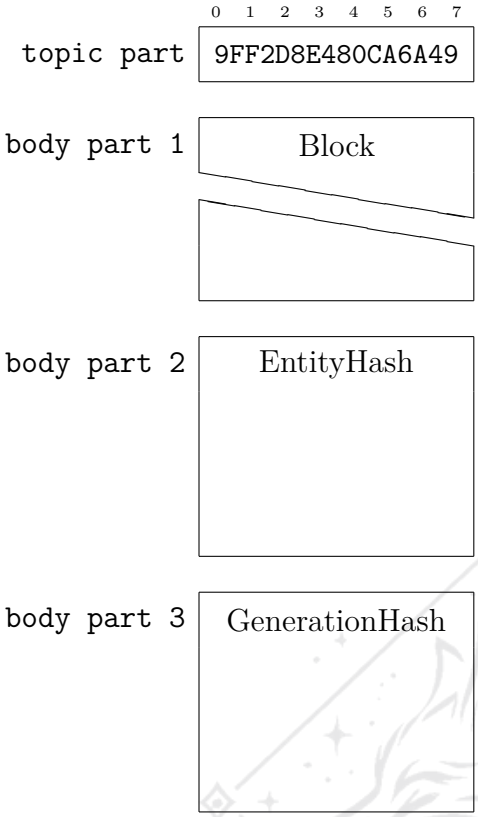
17.3 Block Messages

The topics for block messages only consist of a topic marker. The layouts for all block messages are displayed in Figure 40. The following block messages are supported:

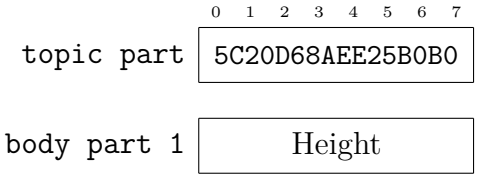
- Block: A new block was added to the chain.
- Drop blocks: Blocks after a given height were dropped.
- Finalized block: Block has been finalized. Published only when deterministic finalization is enabled.

CRYP2NOVA

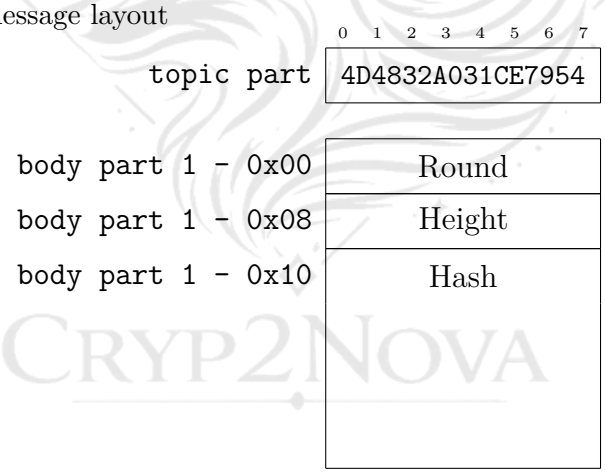
Archived copy · cryp2nova.com



(a) Block message layout



(b) Drop blocks message layout



(c) Finalized block message layout

Figure 40: Block related messages

Archived copy · cryp2nova.com

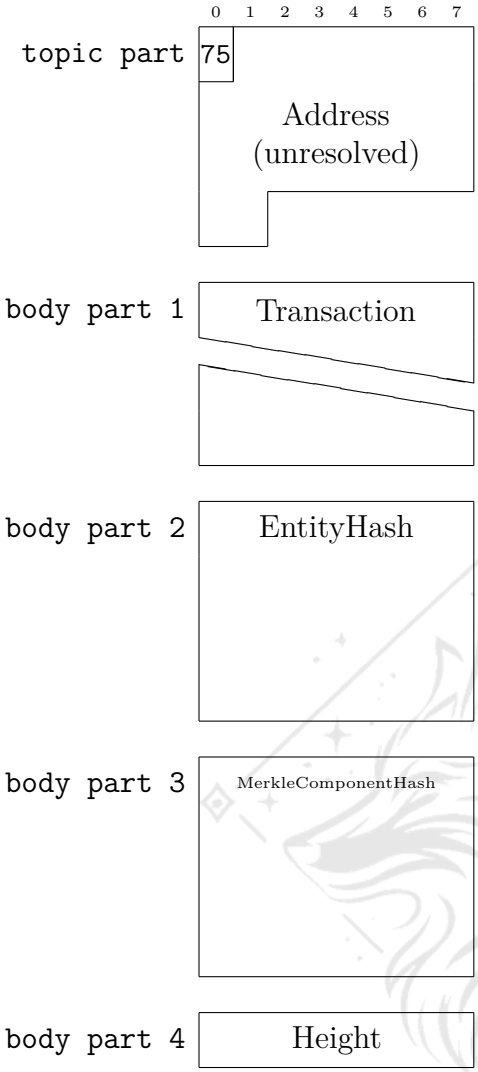
17.4 Transaction Messages

The topics for transaction messages consist of both a topic marker and an optional unresolved address filter. When an unresolved address filter is supplied, only messages that involve the specified unresolved address will be raised. For example, a message will be raised for a transfer transaction only if the specified unresolved address is the sender or the recipient of the transfer. When no unresolved address filter is supplied, messages will be raised for all transactions. The layouts for all transaction messages are displayed in Figure 41 , Figure 42 and Figure 43. The following transaction messages are supported:

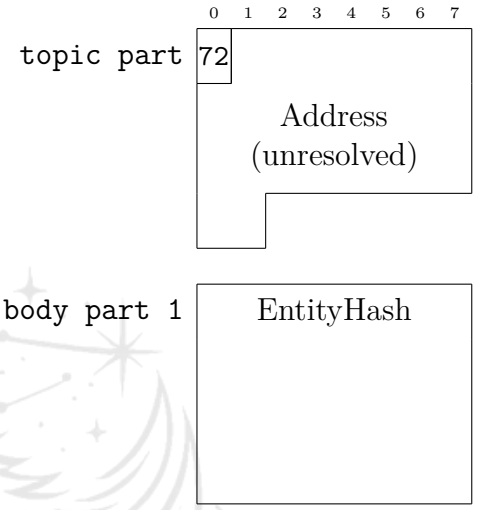
- Transaction: A transaction was confirmed, i.e. is part of a block.
- Unconfirmed transaction add: An unconfirmed transaction was added to the unconfirmed transactions cache.
- Unconfirmed transaction remove: An unconfirmed transaction was removed from the unconfirmed transactions cache.
- Partial transaction add: A partial transaction was added to the partial transactions cache.
- Partial transaction remove: A partial transaction was removed from the partial transactions cache.
- Transaction status: The status of a transaction changed.

CRYP2NOVA

Archived copy · cryp2nova.com



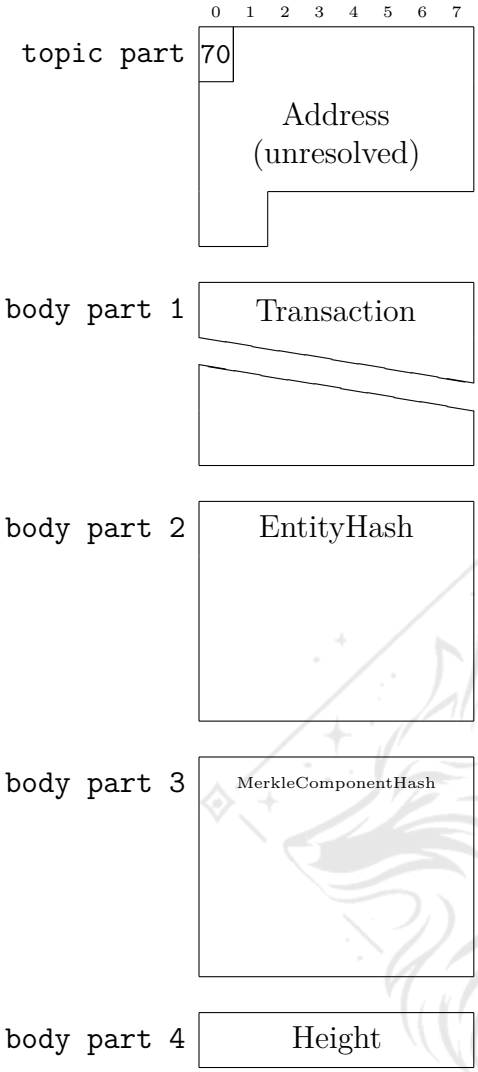
(a) Add message



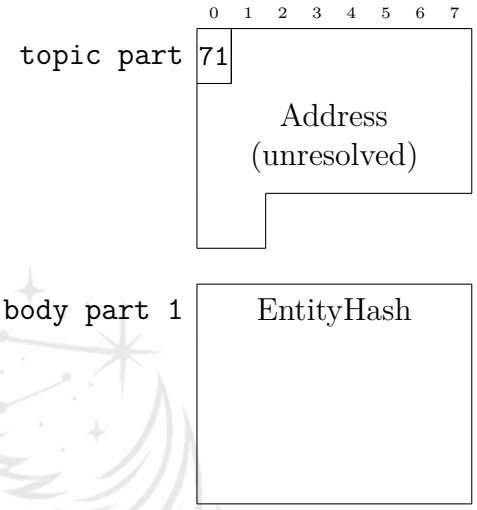
(b) Remove message

Figure 41: Unconfirmed transactions related messages

Archived copy · cryp2nova.com



(a) Add message



(b) Remove message

Figure 42: Partial transactions related messages

Archived copy · cryp2nova.com

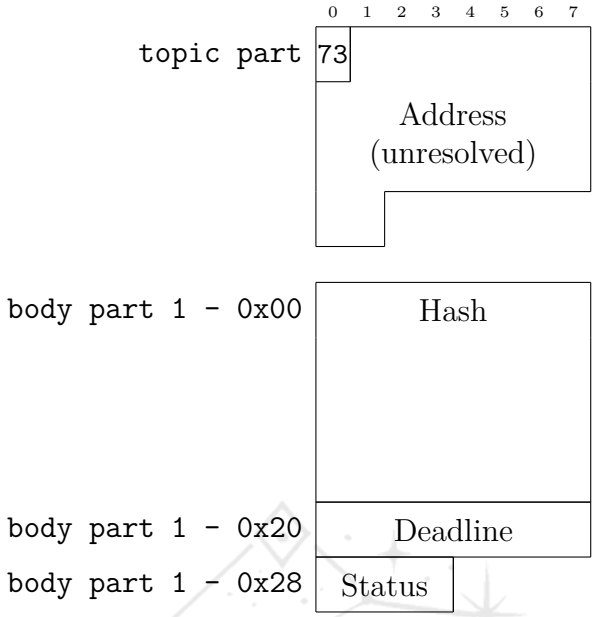


Figure 43: Transaction status message

17.4.1 Cosignature Message

The topic for a cosignature message consists of both a topic marker and an optional unresolved address filter. The message is emitted to the subscribed clients when a new cosignature for an aggregate transaction is added to the partial transactions cache. When an unresolved address filter is supplied, messages will only be raised for aggregate transactions that involve the specified address. Otherwise, messages will be raised for all changes. The layout for the cosignature message is displayed in Figure 44.

CRYP2NOVA

Archived copy · cryp2nova.com

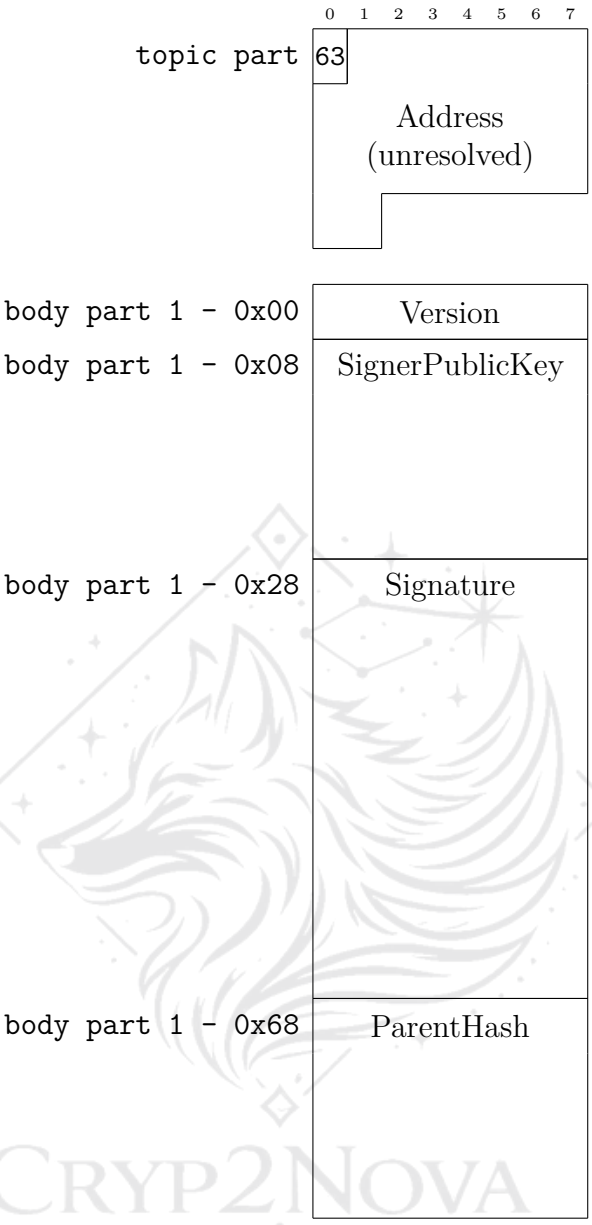


Figure 44: Cosignature message

Archived copy · cryp2nova.com

References

[BCN13] BCNext. *Whitepaper:Nxt*. Web document. 2013. URL: https://nxtwiki.org/wiki/Whitepaper:Nxt#Blocks.

[Ber+11] Daniel J. Bernstein et al. "High-Speed High-Security Signatures". In: *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. 2011, pp. 124–142. DOI: 10.1007/978-3-642-23951-9_9. URL: http://dx.doi.org/10.1007/978-3-642-23951-9_9.

[BG17] Vitalik Buterin and Virgil Griffith. *Casper the Friendly Finality Gadget*. 2017. eprint: 1710.09437. URL: http://arxiv.org/abs/1710.09437.

[BM99] Mihir Bellare and Sara K. Miner. "A Forward-Secure Digital Signature Scheme". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Springer, 1999, pp. 431–448. DOI: 10.1007/3-540-48405-1_28.

[CL99] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 173–186. ISBN: 1880446391. DOI: 10.5555/296806.296824.

[Gol+20] Sharon Goldberg et al. *Verifiable Random Functions (VRFs)*. Internet-Draft draft-irtf-cfrg-vrf-07. Work in Progress. Internet Engineering Task Force, June 2020. 39 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-07.

[KN12] Sunny King and Scott Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. Web document. Aug. 2012. URL: https://decred.org/research/king2012.pdf.

[Mer88] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. 1988, pp. 369–378.

[Mor68] Donald R. Morrison. "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM, 15(4)*. 1968, pp. 514–534.

[Nak09] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[Sci09] Sirio Scipioni. "Algorithms and Services for Peer-to-Peer Internal Clock Synchronization". PhD thesis. Universit'a degli Studi di Roma „La Sapienza", 2009.

[SK20] Alistair Stewart and Eleftherios Kokoris-Kogia. *GRANDPA: a Byzantine Finality Gadget*. July 2020. arXiv: 2007.01560 [cs.DC].

Archived copy · cryp2nova.com

Index

address
  decoded, 21
  encoded, 21
alias address, 24

block
  difficulty, 43
  fee multiplier, 45
  generation, 44
  generation hash, 47
  hit, 47
  nemesis, 34
  score, 44
  synchronization, 52
  target, 47
  time smoothing, 48

consumers, 56
  block, 58
  common, 57
  transaction, 60

Ed25519, 10
extensions, 4

finalization epoch, 93
finalization point, 93
finalization round, 93
fingerprint, 2

generation hash, 47

harvesting, 44

initial difficulty, 43

key
  private, 10
  public, 10

messaging
  channels, 107
  messages, 108

nemesis block, 34
network time, 102
node, 68
  banning, 76
  dynamic, 72
  selection, 75
  static, 68

plugins, 3

reputation
  connection management, 74
  node banning, *see* node, banning
  node selection, *see* node, selection

score
  importance, 80
  node, 81
  stake, 80
  transaction, 80
signature, 11
  BM tree, 95
  malleability, 11
  voting, 14
sybil attack, 82
synchronization, 52

target block time, 34
time offset, 102
TLS, 69
tree
  branch, 18
  leaf, 18

unconfirmed transactions
  cache, 61
  spam throttle, 62

VRF, 12

Archived copy · cryp2nova.com