Rubix – The Proofchain

Technical Whitepaper
Version 1.1
December 21, 2020

https://github.com/rubixchain/rubixnetwork

Abstract

The Rubix Proofchain protocol is a deterministic state-machine that is designed to address the scale, cost, and privacy shortcomings of blockchain protocols that rely on one sequentially organized chain of all global transactions. The protocol divides the global state-machine into a large, but finite number of state-machines called Proofchains. While each Proofchain maintains one state, together all Proofchains represent a globally accessible singleton state that is immutable. This paper explains various components that make up the protocol.

Blockchain protocols such as Bitcoin[1] & Ethereum[8] in general achieve a globally accessible singleton state by organizing all global transactions sequentially as blocks (each block having a finite number of transactions) and organizing the blocks as a hashchain. Such blockchain protocols require exhaustive mining-based Proof-of-Work (PoW) consensus algorithms to secure the state, which results in high latency, low throughput, and high transaction costs. While the Proof-of-Stake (PoS)[7] consensus protocols may alleviate the energy & throughput issues associated with the PoW consensus, PoS protocols suffer from concentration (nodes with higher existing stakes may continue to gain larger voting power), security ("nothing-at-stake" & impersonation risks). Further, both PoW & PoS protocols require every node to store the entire global state, which results in significant storage inefficiencies.

In contrast to the sequential transaction architecture of blockchains, Rubix ProofChain processes transactions in an asynchronously parallel manner. Each transaction achieves finality on its own without waiting to be pooled with unrelated transactions.

ProofChain

The Rubix global state is made of 50.7 million ProofChains. Each ProofChain is bound by one unique utility token. A ProofChain P_T is made of all transactions that use token T_n to confirm. All transactions within a ProofChain P_T are validated individually and sequentially. However, transactions of different ProofChains are validated asynchronously and parallelly.

Tokens

Every transaction in Rubix network is bound to one or a set of tokens. These tokens are various entities in Rubix platform based upon its design and application. The two main categories are Asset tokens and Utility tokens. There are a total of 50.7 million native utility tokens (named Rubix or RBX tokens). Asset tokens represent both (a) unique digital assets like tickets, coupons, rewards, vouchers or collectibles and (b) the digital form of any real-world assets like land, shares, vehicle etc.. These tokens are Non-Fungible Tokens (NFT) much like Ethereum's ERC721 tokens. They are unique and cannot be interchanged. Such tokens do not add any value on its own to the network and hence are not limited in supply nor are restricted in creation. When compared to its Ethereum model ERC 721, Rubix tokens are more dynamic. Unlike ERC721 which use a parameter "**value**" during the creation itself, Rubix asset token's value can vary during the life, depending on the underlying utility token(s) value. Rubix Asset Tokens are bound to the proofchains with the help of the utility token(s) (RBX).

A land asset token A_i bound by utility tokens worth *x* units holds its price value at *x*. In the next transfer of token A_i, let us assume it is bound to a transaction with utility tokens worth *y* units; the value of the asset token A_i then

changes to *y* units. Thus, in Rubix platform, the current value of any asset token depends upon the earlier transaction it was involved in. If an asset token has not been transacted even once in the network, it effectively holds no value in the network.

Rubix native utility tokens (RBX) are capped at 50.7 million in total. With Rubix's breakthrough ProofChain architecture, the network does not require expensive miners or centralized stakeholders with proof of stake protocols to maintain the network integrity. All the utility tokens are pre-created, but a set number of tokens can be mined by application developers & notaries who store proofs & transaction data. Rubix is built to facilitate decentralized applications that power real world commerce at a significant scale on a decentralized network. Early app developers can earn pre-created tokens based on the velocity of transactions on their applications. Rubix consensus protocol deploys notaries who engage in a consensus process to achieve PBFT[3], but the notaries are not essential to prevention of double spending. The notaries though can enhance the robustness of the network by storing proofs & transaction data. Hence the notaries also can earn utility tokens based on their proof of work to store data.

Rubix generates 50.7 million native utility tokens. The following conditions must be satisfied while pushing value-based tokens into any distributed trustless network. Firstly, the tokens should be finite in supply; No node including the Rubix developers should be able to create additional tokens. Secondly, each of the 50.7 million tokens should be uniquely identifiable and publicly verifiable at any point of time by all nodes.

Rubix generates its utility tokens from a combinatorics design called Latin square[7]. By design, the number of Latin squares that can be generated from a set of inputs of a size is limited. This design meets the need for unique values for each of the Latin squares which is important to satisfy the second condition of tokens in the Rubix Proofchain (all tokens should be uniquely identifiable & publicly verifiable by any node). Rubix platform uses a reduced Latin square of size 7 × 7 that can create 16.9 million unique Latin squares; three reduced Latin Squares are created to generate the required 50.7 million tokens. Each of these Latin square values are hashed (SHA3-256) to obtain Rubix utility tokens. Such representation of unique finite tokens secures against infiltration of more pseudo tokens into Rubix network. While each token is unique from another, they are all equal in their utility to validate transactions. In other words, all tokens are fungible with each other. The utility of a token T_x is also independent of the work done on the ProofChain P_T (number of transactions validated on the chain). For example, ProofChain P_{Tx} could be longer than P_{Ty}, but corresponding tokens T_x and T_y are equal in their utility for validating a new transaction.

To keep integrity of the network, it is important to verify the genuine ownership of each of the 50.7 million Rubix utility tokens. Rubix achieves this by representing its tokens in the form of Latin square hashes and along with its custom version of the Interplanetary File System (IPFS)[2] protocol. Each of the 50.7 million hashes in Rubix chain is pushed into IPFS and committed by any one of the token issuance nodes.

Since IPFS follows content-based addressing, any alteration even on a single bit will result in an entirely different hash value that will not be verified by any of the token chains. Once all the hash values are generated, they are passed into a bloom filter. Bloom filter is a memory efficient probabilistic data structure that tells whether an element is present in a set or not. The set generated by passing all the hash values to bloom filter can be used by any node in the network to perform the first level of verification of the token in an efficient manner.

Nodes

A Node looking to transact must register on the Rubix network. Rubix full nodes can be set up on most laptops & desktops. A Rubix node will automatically be registered as a peer-node on the Inter Planetary File System (IPFS). An IPFS peer node is automatically assigned a Peer ID. Peer ID is a cryptographic hash of the public key of the peer node.

IPFS uses a public-key (or asymmetric) cryptographic system to generate a pair of keys: a public key which can be shared, and a private key which needs to be kept secret. With this set of keys, an IPFS peer node can perform authentication, where the public key verifies that the peer with the paired private key actually sent a given message, and encryption, where only the peer with the paired private key can decrypt the message encrypted with the corresponding public key.

Decentralized Identity (DID)

Every node joining Rubix platform creates a unique network Decentralized Identity (DID). DID is a 256*256 PNG image that is self-created but verified decentralized by peers in the network.

DID creation process

DID creation take 2 input parameters, first a 256*256 PNG image of users' choice. Second seed is a SHA3-256 hash H, that is generated from user and machine characteristics. Decentralized Identifier needs to satisfy two properties; Uniqueness and Randomness. Uniqueness is required to distinctly refer each node and randomness property is essential for share generation phase. The PNG image provides randomness since its chosen by user whereas the hash H contributes towards achieving uniqueness.

DID Embedding Process

1. The 256*256 PNG image is converted to bits.
2. First 256 bits performs a XOR operation with the SHA3-256 hash H.
3. A hash of H is created to embed with the next bits 256-511 of the PNG image bits.
4. Subsequently, a hash chain of hashes is created and each hash is embedded to corresponding PNG image bits
5. The embedded bits are finally converted back to a 256*256 image

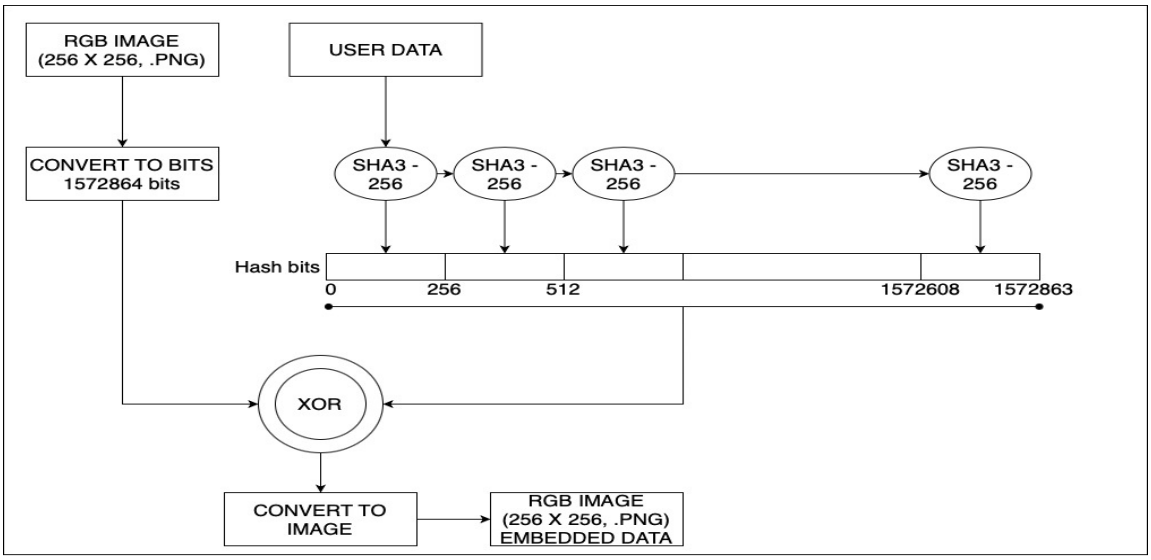


Fig1: DID embedding process

DID Share Generation Process

The DID for node i, K_i, is split into two secret shares using the Non-Linear Secret Sharing (NLSS)[6] cryptographic techniques – (a) a public network share K_{in} and (b) a private share K_{ip}. Upon splitting, the two shares are 8 times the size of the original DID. K_i and K_{in} are stored by the node in the IPFS. The IPFS hashes of K_i and K_{in}, $H(K_i)$ and $H(K_{in})$ respectively, are shared across other peer nodes in the network using gossip protocol. Any peer node, to perform a transaction with node i, fetches the K_i and K_{in} from the IPFS using $H(K_{in})$ and $H(K_{in})$.

K_{ip} is the secret knowledge known only to the node i. To authenticate the node i to the Rubix Network, knowledge of the private share knowledge of the private share K_{ip} is necessary. Since only the node i has the knowledge of the private share K_{ip}, only it can authenticate successfully to the rest of the network.

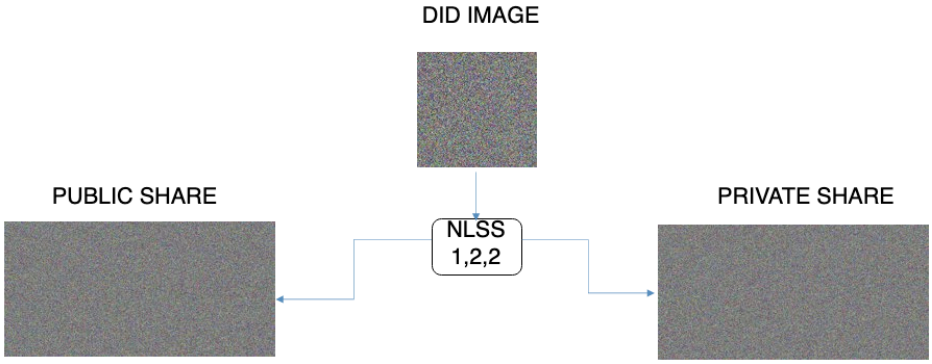


Fig2: DID shares creation using NLSS 1,2,2 scheme

Authentication using DID

The DID Token for node **i**, K_i is split into two secret shares using the Non-Linear Secret Sharing (NLSS) cryptographic techniques – (a) a public network share K_{in} and (b) a private share K_{ip}. Each of the shares expand

into 8 times upon splitting. K_i and K_{in} are stored on the IPFS & are globally accessible as well as verifiable. K_i and K_{in} are inseparable from the Peer ID of the node **i** and hence cannot be faked. The private share K_{ip} is secretly and securely stored by the node **i**. Using the NLSS cryptographic techniques, each pixel of the DID token K is split into two secret shares. Kin and Kip are reconstructed from the split shares of all pixels of K. Knowledge of either K_{in} or K_{ip} does not help in the reconstruction of K_i. To reconstruct K_i, knowledge of both K_{in} and K_{ip} as well as the Non-Linear function F_n used for splitting into the secret shares is needed.

Since K_{ip} is the secret knowledge possessed by the node **i**, only node **i** can authenticate itself to another peer node. In other words, to authenticate to another peer node, knowledge of the private share K_{ip} is necessary. The authentication process involves cryptographic Proof of Work from the node **i**. Let us say that the node **i** would like to authenticate itself to node **j**. Node **j** would issue a cryptographic challenge by asking for the values of the 256 pixels from the private share K_i. Node **j** would randomly select the 256 position values that make up the challenge. Note that node **j** would be honest in picking the 256 position values since it is seeking the identity of another node. There is a total of 1,572,864 pixels (each RGB pixel can take 24 values & there are a total of 256x256 or 65,536 pixels; 65,536 x 24 = 1,572,864) in a DID token. Upon receiving the challenge, Node **i** would return the values of the 256 pixels of the private share K_{ip}. Node **j** combines the values of the 256 pixels of K_{ip} with the values of the corresponding 256 pixels of K_{in} to compare the result with the values of the corresponding 32 pixels of K_i. (note that each pixel of K_i is expanded to 8 pixels while splitting into K_{ip} & K_{ip}. A complete match would successfully authenticate Node **i** to Node **j**.

Non-Linear Secret Sharing (NLSS)

In a secret-sharing scheme, a set number of parties share a common secret. The information about the secret a single party owns is called a secret share. In secret-sharing schemes for ***n*** parties the shares t_i for $i = 1,2, \ldots, n$ are often computed according to some arithmetic functions $t_i = f_i(s_1, s_2,\ldots,,s_u)$ where s_1 is the secret and $s_2,\ldots,s_u$ are randomly chosen elements of some fields. We call such a scheme linear if all f_i are linear with respect to the variable $s=(s_1, \ldots, s_u)$, and nonlinear otherwise. Linear secret sharing schemes are not cheating-immune[4]. Nonlinear secret sharing shame is a cheating-immune secret sharing scheme that prevents a cheater, who submits a corrupted share, from gaining an advantage in knowing the secret over the honest participants. A perfect secret sharing scheme realizing the access structure Γ is a method of sharing a secret among a set of participants P, in such a way that the following two properties are satisfied:

i. Every authorized subset B ⊆ P can determine the secret.
ii. Every unauthorized subset B ⊆ P obtains no information about the secret.

An (n, n)-secret sharing scheme can be represented by a defining function $f : F_q^n \rightarrow F_q$ which maps to each vector of shares a secret value in F_q. The defining function of an *(n, n)*-secret sharing scheme over F_2 is a Boolean function.

Theorem 1:*An (n, n)-secret sharing scheme with defining function f is* 1*-cheating-immune if and only if f is* 1*-resilient and satisfies the SAC.*

Theorem 2 : *Let C be a binary* [*n – m,m, d*] *linear code with dual distance* $d^{\perp}$*. Assume that* $d^{\perp} \leq d$*. Let G be a generator matrix of C and let α be a vector in* F_2^{n-m} *which is not in C. Consider the function f (x, y) =* $xGy^T \oplus \alpha y^T$*, where* $x \in F_2^m$ $y \in F_2^{n-m}$ *and* $\oplus$ *is the addition over* F_2*. Then we have the following:*

i. f is t-resilient with $t = \min_{c \in C}\{wt(c \oplus \alpha)\} - 1$

ii. f satisf ies the strengthened propagation of degree l with $l = d^{\perp} - 1$.

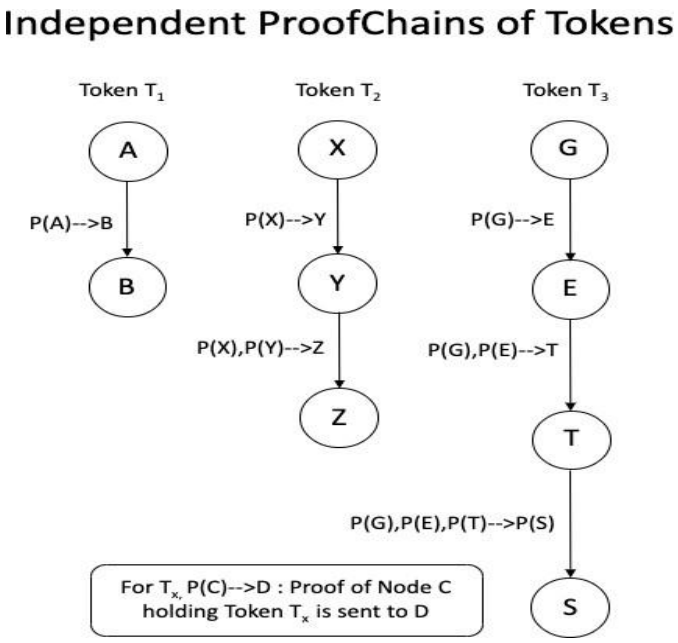
Rubix Transaction

All Rubix nodes join the network to conduct transactions with each other. A transaction can be (a) a digital contract that involves exchange of services or goods in return for exchange of other services or goods (b) a digital contract that involves exchange of services or goods for exchange of any medium of value such as fiat currency or simple transfer of native token.

At any given point of time, several peer-to-peer transactions are submitted to the Rubix network. Transactions are processed parallelly independent of each other unless transactions involve common peers. A transaction is initiated by one peer node. To initiate the transaction, the peer node must use at least one RBX token. Depending on which native token (T_i) is used by the peer node initiating the transaction, the transaction is added to the corresponding ProofChain P_{Ti}. If multiple tokens are used in a transaction, the transaction is added to multiple ProofChains.

Proofchain

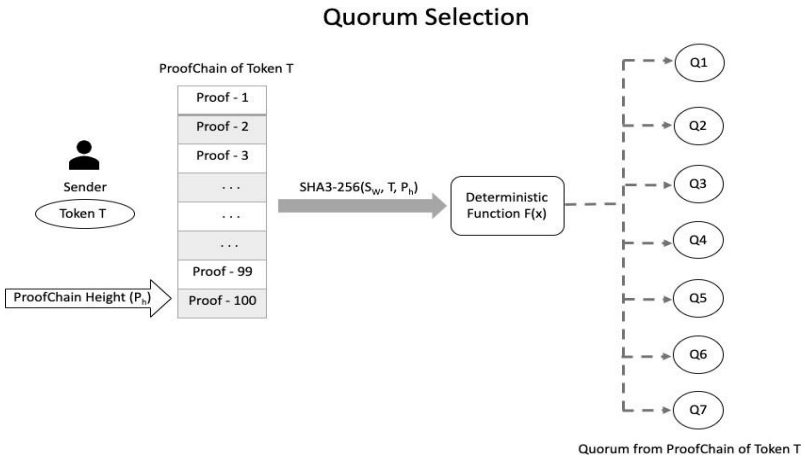
A ProofChain P_{Ti} is a chain of all transactions bound by the utility token T_i (note that there are a total of 50.7 million RBX utility tokens in the Rubix Network). A set number of the 50.7 RBX tokens are committed by the genesis node G. All proofchains with the RBX tokens committed in the genesis node G originate with the genesis node. All such RBX tokens are stored and committed on the IPFS by the genesis node G. The node G's ownership of such tokens is globally verifiable. All RBX tokens that are not pre-committed to the node G are mined by either developers or notaries. In such cases, once the RBX token is mined, the corresponding Proofchain of that token starts with the node that has mined the token.



Consensus Protocol

Rubix consensus is a lightweight Practical Byzantine Fault Tolerance (PBFT) algorithm that require $2n + 1$ out of $3n + 1$ members to reach an agreement. Rubix PBFT algorithm is designed to work in practical, asynchronous environments.

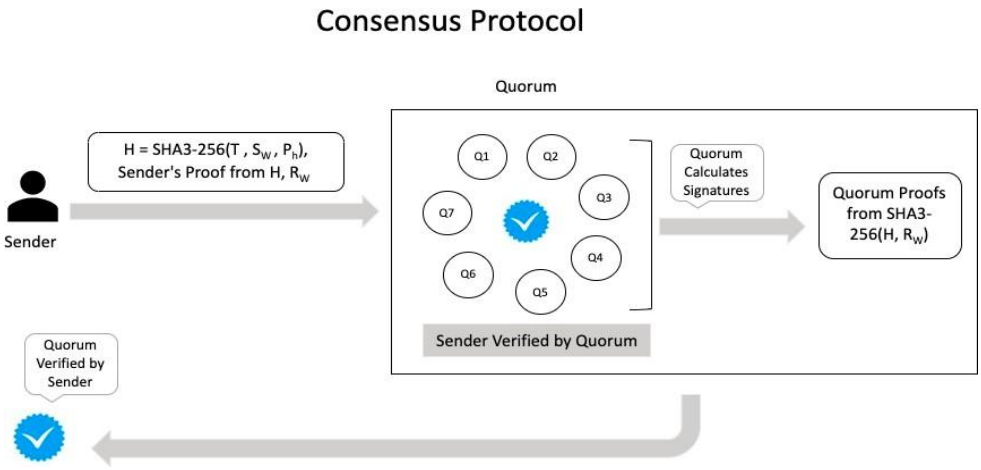
For each transaction, a set of 7 (the minimum number of nodes required for a PBFT consensus) nodes are selected by the initiator. The 7 nodes are deterministically chosen from the ProofChain(s) of the token(s) to be transferred. These nodes, called quorum members perform a Multi-Party Computation (MPC) based consensus. If 5 out of the 7 nodes in the quorum agree to the transaction, the consensus is achieved.



NLSS based consensus protocol

Consensus involves agreement between multiple parties in a distributed network. This can be achieved by running a single chain with blocks created one after the other as an agreement to the previous blocks on the chain. However, such chains never reach a state of finality. Plus, the scalability factors for the all nodes to synchronize with other peers in the network makes these models inoperable. In Rubix platform consensus protocol is run for each transaction independently. This way each transaction can be independently verified reducing forks.

Rubix Consensus involves an Initiator (I), Seven Nodes (Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7) – Quorum Initiator creates the Transaction Data in the following way:
Transaction Data = SHA3-256(Initiator's WalletID + Token + ProofChain Height)



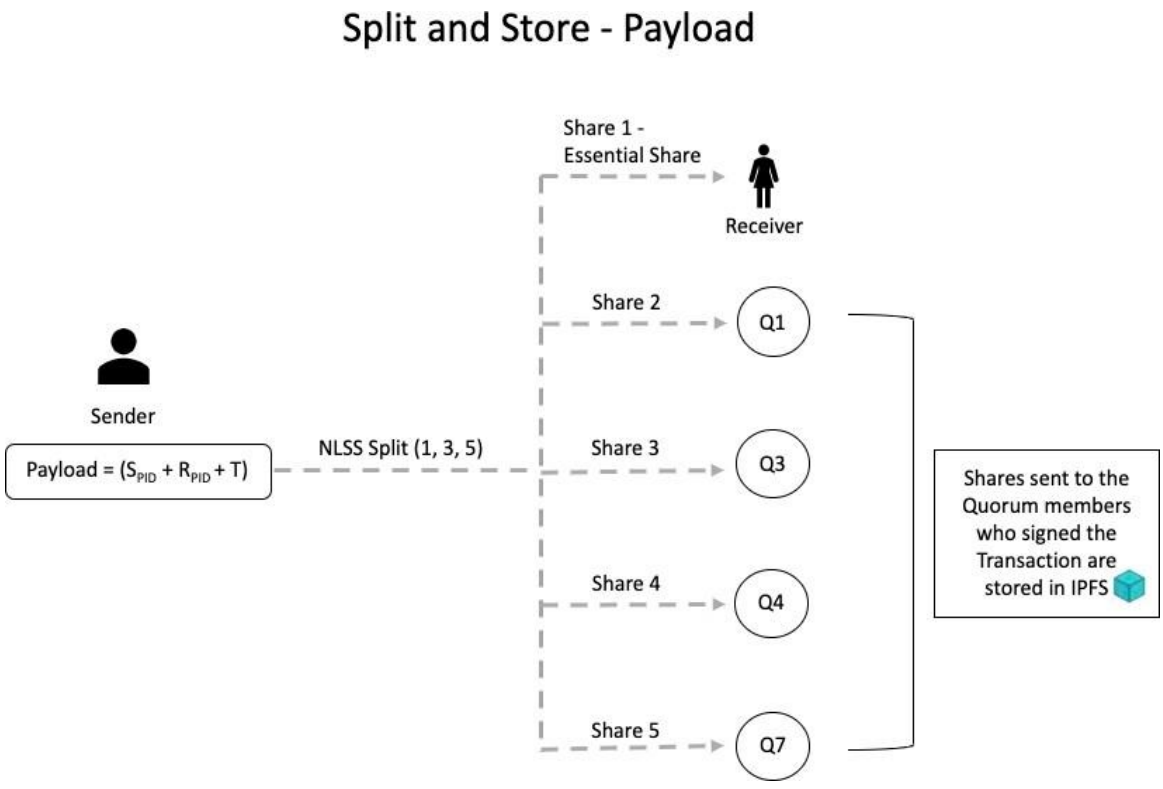
Post-quantum Cheating Immune Algorithm that facilitates Multi-Party Computation (MPC) based consensus process. To recreate the transaction data image, each quorum member should have minimum of two shares of which one should be the essential share. The value 7 is chosen based on Practical Byzantine Fault Tolerance (PBFT) algorithm. Hence, 5 out of 7 votes from the Quorum nodes is required for a successful consensus.

Steps:

1. Sender picks quorum members Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7 from ProofChain using a deterministic well dispersed function.
2. Sender calculates the SHA3-256 hash H of T, S_w and the ProofChain height P_h and sends H, Sender's Signature from H and Receiver ID R_w to all quorum members.
3. The quorum members provide 32 × 64 bits of their private share from positions derived from H and R_w. This way the quorum agrees to this particular transaction of sender and token. (This can be Light-weight PoW quorum performs)
4. When 5 or more signatures are received by sender, he performs Peer-Peer authenticated token transfer and IPFS Committing & Uncommitting.
5. Once the BFT count is reached, the Rubix consensus is successful.

Payload split & store

Visual Secret Sharing scheme is used during this storage module at the Sender after successful Consensus. The payload is split into 5 shares using **(1,3,5)** scheme. The essential share is stored/retained with Receiver of the token. The shares related to each transaction are stored in different nodes (Recipient, Notaries, Backup Notaries) to prove provenance of the transaction. In case the notaries (who were part of the consensus and hold shares) fail, the two backup notaries have rest of shares stored.



Simple token transfer transaction

Let us say node A would like to enter a transaction with node B. To validate the transaction, A needs at least one token (more

than one token may be needed in certain transactions). For the current illustration, let us say one token is needed. A will enter a transaction to procure the required token T_x from the genesis node G. Nodes G and A enter a Peer-Peer transaction. Before entering the transaction, A requests the IPFS hash of the token T_x from G. A verifies if node G owns the token by checking if G and only G has committed T_x on the IPFS. If true, A proceeds to complete the transaction with G. If A finds that node(s) other than G found committing T_x, A will request for additional proofs of ownership from G and other nodes who are found committing T_x.

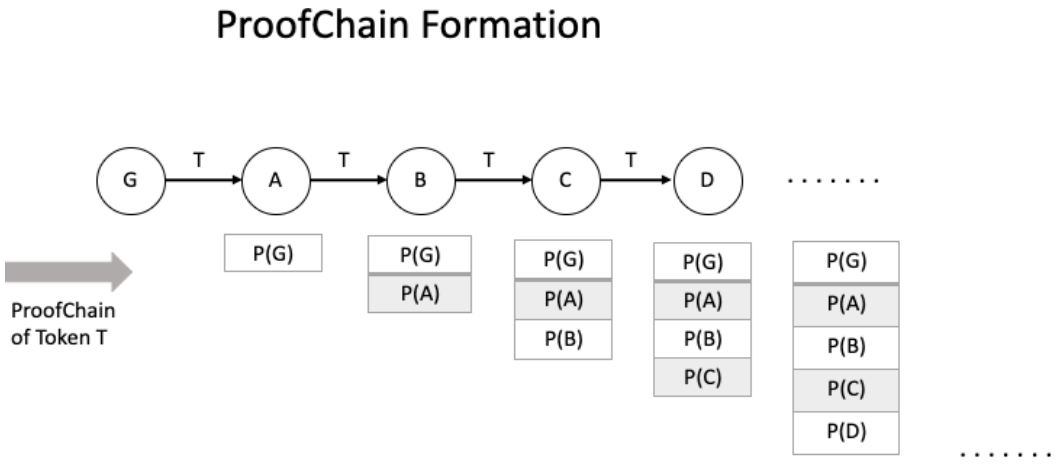
If no fork is found and G is the sole committer of T_x, then G proceeds to complete the transaction with G.

1. For a user in the Rubix network, after successfully completing the DID registration and share generation, a gossip protocol broadcasts the Public share and DID token across the network.

2. Initially the sender sends the token's IPFS hash to the receiver and the receiver acknowledges the availability of the token

3. The token's latest proof (ProofChain) is known only to the sender and therefore, after the acknowledgement the latest ProofChain is sent to the receiver for picking challenge-response positions.

4. The ProofChain represents the universal proven state of each token, which is publicly verifiable.



5. If token T is purchased from Rubix genesis node G by user A. Node A gets empty token chain from G creates a new chain with proof P(G)

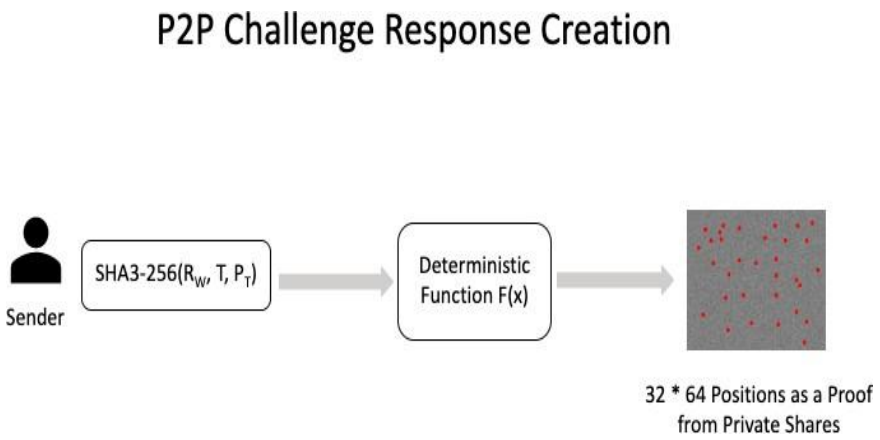
6. Now, when user A transfer token to user B, the proof for this transaction P(A) is appended to the existing ProofChain. Hence the current ProofChain contains two proofs, P(G) > P(A)

7. The sender calculates SHA-256 hash of receiver's wallet share (R_w), token (T) and the ProofChain (P_T).

8. A Deterministic Function F(x) is used on the hash to select 32 challenge - response positions

F(x) = (2402 + hashCharacters[k]) * 2709 + k + 2709 + hashCharacters[k])%2048

9. The challenge response positions are picked from a hash which is calculated using R_w, T and P_T



10. R_w and T are used in hash calculation to ensure that the agreed token is being transferred to the recipient

11. For instance, A is transferring a token, T to B [A (T)—> B]. The inputs for the hash calculation are B_w, T and P_T.

12. Any other user out of this transaction, C cannot claim token T has been transferred to C by A

    [A (T)—> C].

13. B cannot claim A has transferred some other token by A [A (~T)—>C].

14. ProofChain is used because, apart from the sender no other user in the network possesses the latest token chain of the token. Therefore, no party, not even the receiver cannot calculate the hash and pre-compute the 32 challenge - response positions for the transaction of token T.

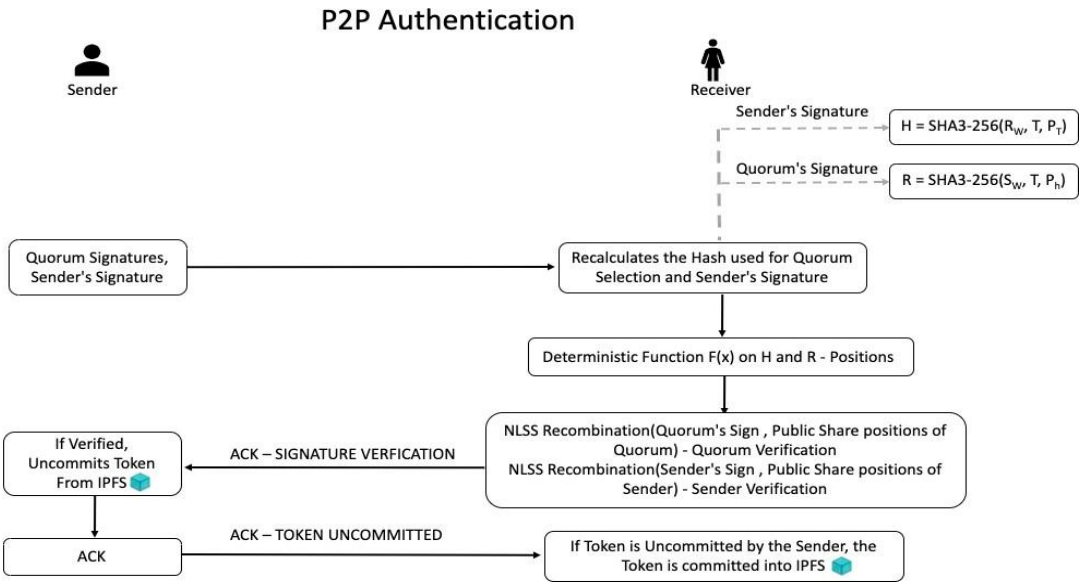
15. The sender picks the corresponding 2048 (32 positions pre-calculated along with 63 trailing bits of each 32 bits is chosen from the total 20, 97, 152 of the shares. Making it a total of 32 * 64 = 2048 bits) positions values from each of the 3 private shares and is shared with the receiver.

16. On the other side, the receiver also calculates the SHA3-256 hash of W_R, T and P_T and subsequently determines the 2048 positions by applying the same deterministic function on the hash

17. Since the wallet shares and DID of all users are publicly available, corresponding 2048 bits from W_S are chosen and is recombined using NLSS recombination with private positions shared by the sender. The recombined result yields 32 bits. The receiver picks the corresponding 32 positions from the DID sender and performs a comparison check.

18. If the 32 bits obtained from the recombination and the 32 bits picked from the DID sender matches, the sender is authenticated.

19. Receiver checks whether the token is held by any other node in the network and then acknowledges the sender



20. The token to be transferred is uncommitted by the sender and committed by the receiver thereby claiming the ownership of the token

The transaction proof generated in the process detailed from Step 1 to Step 20 is deterministic and highly secure. The proof confirms that A has received T_x from G. A can only claim that it received T_x from G and not any other token T_y. This is because A will not be able to produce the knowledge of the 256 bits of G that correspond to token T_y.
Furthermore, a third node B cannot claim the receipt of T_x from G since node B will not be able to produce the proof that G has signed the transfer of token T_x. Note that G has signed the transfer to node A by hiding the knowledge of G's private share with A's network share.

For node B to prove the transfer, it would have needed G to sign the transaction with B's network share.

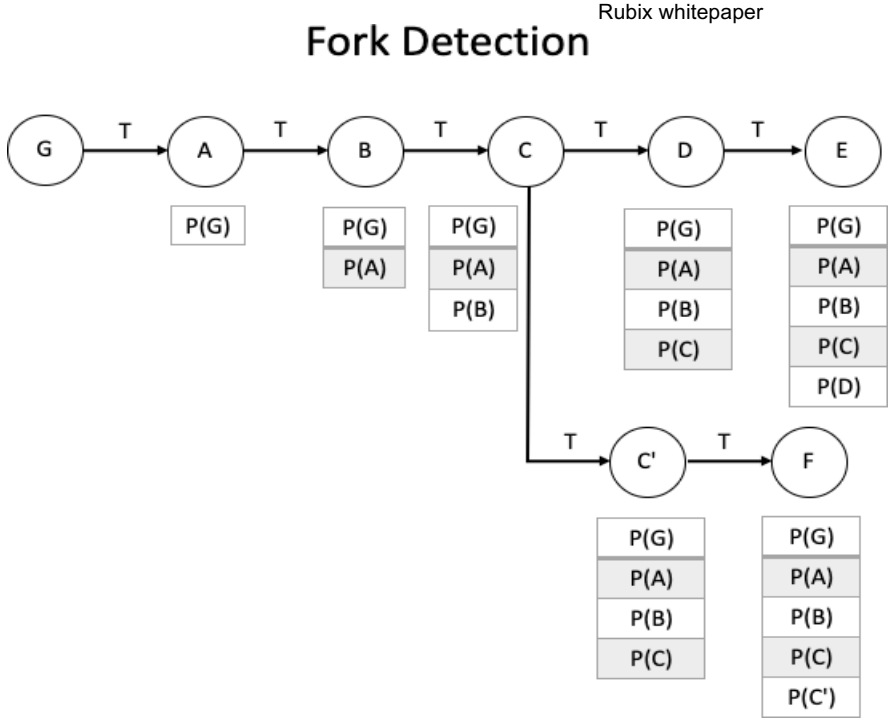
The probability that same 32 bits will be selected to sign the transaction for two different tokens and/or for the same token to a different receiver is $(1/262144)^{32}$ or $6.44e^{-94}$. User authentication as described in steps 1 to 21 has a brute force resistance of 2^{196} .

After the successful completion of the transaction $T_{ØA}$, A creates ProofChain P(G) and both G and A store the ProofChain. When A further transfers token T_x to node B, the ProofChain expands to P(G) + P(A) and nodes A and B store the updated ProofChain. When B further transfers to C, ProofChain expands to P(G) + P(A) + P(B) and nodes B and C store the updated ProofChain. The ProofChain continues to expand perpetually. Note that the updated ProofChain is not propagated back to all the preceding nodes in the ProofChain.

The ProofChain for token T_y is constructed like that of T_x explained so far. The ProofChain represents the universal proven state of each token, which is publicly verifiable.

Double spending

Double spending is not possible since a node wishing to enter a transaction to acquire a token can verify if the token is committed by only one node or by multiple nodes at any point of time. If the token is also committed by another node which is not part of the transfer, acquiring node can verify easily and get alerted of a possible attempt to double spend.



Say C is the owner of a token (sole committer on IPFS). A transfer's a token to D (C uncommits and D commits). This makes D new owner of the token. If C were to make a digital copy of the same token and

commit again, there will be two committers of the same token C and D. Now when a third node E wishes to acquire the token from the legitimate owner D, E finds out that two nodes C and D are committed to the token. Node E will not proceed with the transaction unless C and D provide further proof. Hence double spending is not possible.

There is no economic benefit for C to fraudulently commit the token again as C is not able to transfer the same token twice.

While double spending is easily prevented in the Rubix ProofChain protocol, forking can be used for denial of service attacks. Denial of service attacks are possible if user create a fake node and transfer same token to fake node. Say node C create fake nodes C' just to prevent F from transferring token further. In such cases, node F will challenge node C' to provide it's ProofChain. The ProofChain of node C' will be P(G) > P(A) > P(B) > P(C)

By traversing back, the token chain, it is easy to determine where the fork(s) occurred. The nodes causing fork can easily be determined. While forks can easily be determined, how does the Rubix protocol resolve the forks?
To resolve a fork, each transaction goes through a consensus protocol and the quorum members sign on

the transaction hash. In case of a fork, the signature of the consensus quorum members is verified to decide the correct chain.

Asset tokens

Asset-backed tokens or security tokens carry an actual value as they are associated with the value of an existing real-world asset. Asset-backed and security tokens provide secure, fast and minimal cost trading of standard assets via blockchain technology, and boost liquidity for conventional securities.

Peer-to-Peer Authentication of Asset Transfer

Consider a scenario when User A (K_A, W_A, P_A) want to transfer an asset AT with hash H_{AT} to User B (K_B, W_B, P_B).

User A calculates the SHA3-256 hash of token T, wallet ID of B (W_B), and Asset hash H_{AT}.

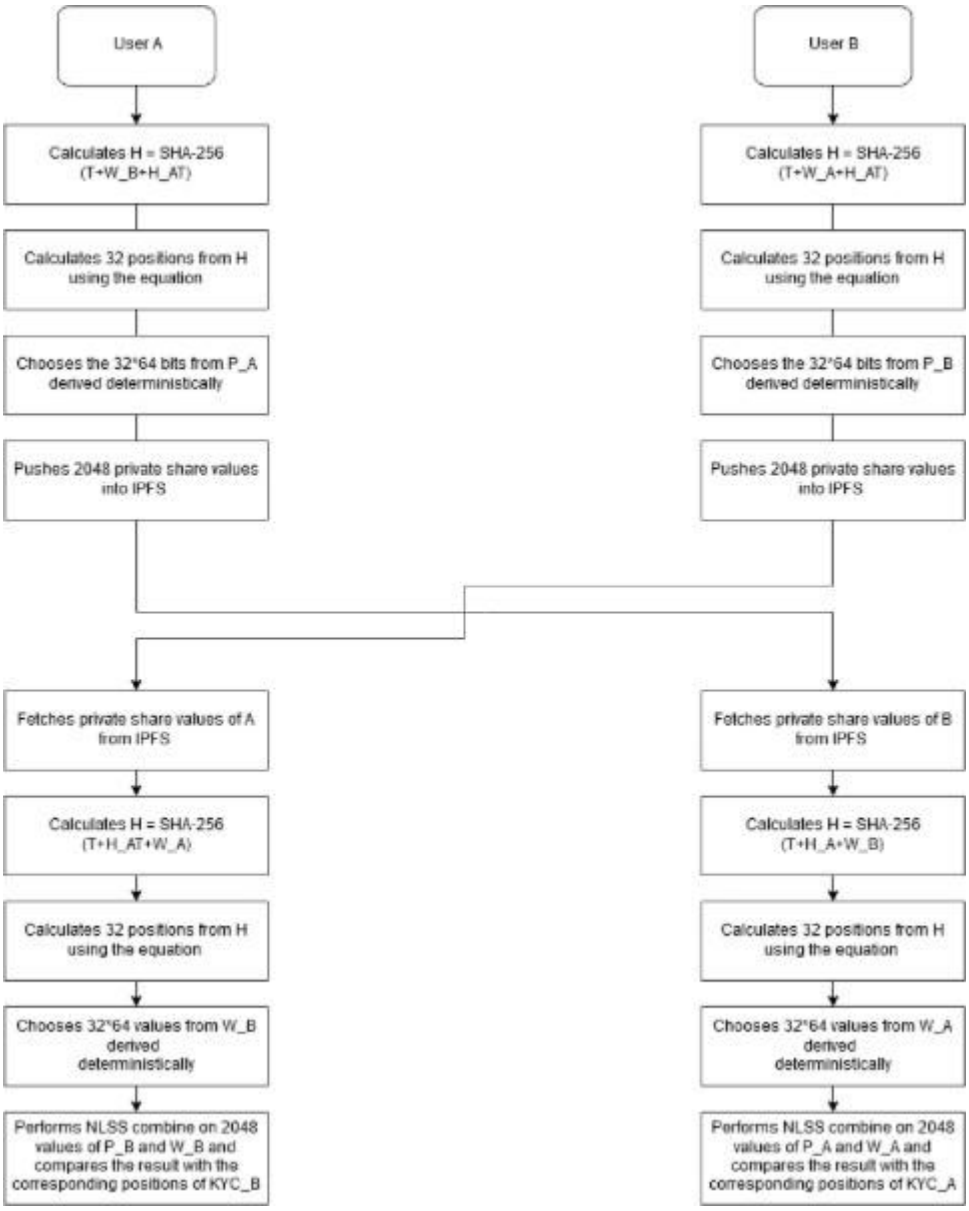
From the calculated hash H, User A deterministically find out 32 positions using a deterministic well-dispersed function.

Values from the private share corresponding to 32 positions pre-calculated along with 63 trailing bits of each 32 bits is chosen from the total 1,572, 864, making it a total of ($32 * 64 = 2048$ values). The 2048 positions and token T values are shared to the receiver B. B also calculates the SHA-3 256 hash token T, Asset hash H_{AT}, wallet ID of B, W_B and subsequently determines the 2048 values of A's private share.

B performs Non-Linear Secret Sharing recombine of the 2048-private shares with corresponding wallet share of A and compare the result with 32 positions of DID_A. Only a perfect match results in authentication of user A. Once user B successfully authenticates user A, A tries to authenticate B. A calculates the SHA3-256 hash of token T, wallet ID of B, W_B, and Asset hash H_{AT}.

By following the same steps mentioned earlier B sends his 2048 private share positions to
A. User A also calculates the SHA-3 256 hash of token T, Asset hash H_{AT}, wallet ID of A, W_A
and subsequently determines the 2048 positions of B's private share.

A performs NLSS recombine of the 2048-private share with corresponding wallet share of B and compare the result with 32 positions of DID_B. Only a perfect match result in authentication of user B. Once the authentication of both A and B is completed the transfer of asset will occur.



A Proofchain can be used to process, validate and store peer-to-peer digital contracts. A digital contract is a contract denoting exchange of goods or services in return for other goods or services or in exchange of other means of value like fiat currencies or commodifies. Instead of sequentially pooling all global transactions into a block as a traditional blockchain would do, each transaction is processed and validated on its own with ProofChains. To include a transaction into Proofchain, the transacting parties need a token for provenance.

IPFS in the Rubix Network

IPFS plays a key role in Rubix network due to below properties of IPFS:

1. Immutable objects represent files
2. Objects are content addressed by cryptographic hash
3. DHT to help locate data requested by any node and to announce added data to the network
4. IPFS removes redundant files in the network and does version control
5. Content addressing of the data stored in IPFS, every file name is unique if there is even a single character change in the content of the file
6. Private IPFS that can only be accessed by certain entities

7. Committing of the data - avoids double spending

Rubix Network and Private IPFS

Every node who joins the Rubix network will be part of private IPFS and therefore they are not connected to the external IPFS network and communicate only to those nodes connected to this private IPFS Swarm. All data in the private network will only be accessible to the known peers on the private network.

LIBP2P Protocol

LIBP2P is a modular network stack of protocols, libraries and specifications that enable development of peer-to-peer network applications. A peer-to-peer network in which the participants of the network communicate with each other directly without involvement of a privileged set of servers as in the case of a client-server model. LibP2P uses public key cryptography as the basis of peer identity, serving two complementary purposes. Each peer is given a globally unique name (PeerID) and the PeerID allows anyone to retrieve the public key of the identified peer enabling secure communication between the peers where in no third party can read or alter the conversation in-flight. To route the data to the correct peer, we need their PeerID and the way to locate them in the network which is done in LibP2P using Peer Routing (discover peer addresses by leveraging the knowledge of other peers). Peer routing in LibP2P is done using Distributed Hash Table (DHT) (iteratively route requests to the desired peer ID using Kademlia routing algorithm).

LibP2P module is used in the Rubix network for communication of data from one end to another. The Rubix network traffic is tunneled through LIBP2P stream. We add all nodes who are part of the Rubix network, to a single private swarm network of IPFS. The communication over the internet from initiator to the notaries and participants during the consensus and from initiator to the receiver during the token transfer are performed using the IPFS listen and forward which is part of the libp2p library.

LIBP2P

1. **Listen for Incoming Streams**

   > ipfs p2p listen /x/<applicationName>/1.0 /ip4/127.0.0.1/tcp/<port>
2. **Forward**

   > ipfs p2p forward /x/<applicationName>/1.0 /ip4/127.0.0.1/tcp/<port>/p2p/<peerid>

SwarmConnect – Connection to Peer node either directly or through Realy service

1. **Relay**

   > ipfs swarm connect /p2p/<BootstrapNode>/p2p-circuit/p2p/<peerid>
   BootstrapNode – multiaddress of the bootstrap for relaying
   Peerid - ID of the node to connect
2. **Direct Connection**

   > ipfs swarm connect <Node>
   Node - multiaddress of the node to connect
3. **Swarm Peers**

   > ipfs swarm peers
   List of the peers a node can connect (fetched from DHT)

Token Access Operations and Commands

Add
> ipfs add <tokenname>
Returns Multihash of the file for reference
Get
> ipfs get <tokenhash>
Fetches the file from IPFS referenced using Multihash
Pin
> ipfs pin add <tokenname>
Pins the token– will not be removed during garbage collection
UnPin
> ipfs pin rm <tokenname>
Unpins the token– removed during garbage collection

References

1. Nakamoto, S. (2008) Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

2. Benet, Juan. "Ipfs-content addressed, versioned, p2p file system." *arXiv preprint arXiv:1407.3561* (2014).

3. Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." *OSDI*. Vol. 99. No. 1999. 1999.

4. M. Tompa and H. Woll, How to share a secret with cheaters, Journal of Cryptology, Vol 1, Issue 2, Aug. 1988 https://dl.acm.org/citation.cfm?id=56181

5. Goldreich, Oded, and Yair Oren. "Definitions and properties of zero-knowledge proof systems." *Journal of Cryptology* 7.1 (1994): 1-32.

6. US Patent 009800408: *Method of generating secure tokens and transmission based on (TRNG) generated Tokens and split into shares and the system thereof.*

7. https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ

8. Keedwell, A. Donald, and József Dénes. *Latin squares and their applications*. Elsevier, 2015.

9. Wood, Gavin. "Ethereum: A secure decentralized generalized transaction ledger." *Ethereum project yellow paper* 151.2014 (2014): 1-32.

10. Renvall, Ari, and Cunsheng Ding. "A nonlinear secret sharing scheme." Australasian Conference on Information Security and Privacy. Springer, Berlin, Heidelberg, 1996.

11. Moni Naor and Adi Shamir. Visual cryptography; Workshop on the Theory and Application of Cryptographic Techniques. Springer. 1994, pages 1–12; and Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized Computation Platform with Guaranteed Privacy; preprint arXiv: 1506.03471, 2015.