FLETA TECH PAPER

ENG

FLETA™

TABLE OF CONTENTS

CONCEPT

FLETA aims to provide blockchain technology which is able to be serviced and also support integration with existing development environments. Providing sufficient functions for service is made possible through our own lightning-quick speed transactions based on high processing volume and low block time, along with an independent multichain structure that helps each service to operate independently. Moreover, FLETA provides Smart Contract services, which can be used with current development formats such as RDBMS, NoSQL, and Event Sourcing, in order to integrate with existing development environments. This, in turn, lowers the entrance barriers emerging between emerging blockchain-based services and existing services. Additionally, this lessens the challenges of development, supporting more efficient and faster advancement and operation of services.

BLOCKCHAIN STRUCTURE

U.S.P(United States Patent) Application Number : 62717703

FLETA is a Hybrid model, both supporting the UTXO and Account models. UTXO's advantage lies in that it uses various keys to make it hard to pinpoint users, while Account model requires less data and has faster processing speeds. Anyone can perform a UTXO-based transaction simply by creating keys. By using a small portion of FLETA, they can utilize the Account in order to complete the transaction with low execution fees. This allows people to transact without creating the Account, while at the same time imposing low execution fees and processing a relatively small amount of data, thus enhancing the efficiency of the process.

UTXO

In the UTXO model, UTXO refers to one pile of coins. Using keys that one owns, it is possible to, combine or split the coin pile or change the ownership. FLETA uses Public Hash made through Public Keys to show ownership. Public Hash consists of 1 Byte Checksum of Public Key and 32 Bytes of SHA256 Hash, which means it is possible to use it without directly revealing the Public Key. If the user creates and uses a new Key with every transaction, UTXO's strength of providing privacy can be leveraged. Moreover, it is possible to use UTXO without creating the Account.

Account

In the Account model, Account is accessible through certain Public and Private Keys. Coin is inscribed as Balance and sending or receiving coin changes the balance amount. It is only possible after creating Account, and has lower execution fees than UTXO, as it processes a relatively low volume of data. FLETA is designed to support various types of Account, and also provides a great deal of basic Account; SingleAccount, MultiSigAccount, FormulationAccount, and ContractAccount. The type of Account can be expanded by each DApp, and details are elaborated in the Transaction Extension of Architecture section. Account address created in FLETA's mainchain is equally weighted in all subchains, plus all coins and tokens can be both withdrawn or deposited with one address.

Account Address

Account Address uses base 58 to express coordinates (block height, transaction location within block) that encompass the Account creating transaction, coordinates that include the current chain creating transaction, and Bytes consisting of nonce value used in Contract. Elaboration of the structure can be seen below.

Address: Base58({AccountCoordinate(6), ChainCoordinate(6), nonce(8)})

In the case of FLETA's mainchain, ChainCoordinate is (0,0). Other than the Sub-Contract, which is made by codes in Contract like Solidity, nonce value is 0. Thus, the basic address of the mainchain has a 9-digit Address through base 58. In this address system, all Account cannot be overlapped, and the transaction is possible with a relatively short address.

Transaction

Transaction serves as a command to change the current chain status to Account, Account Data, or UTXO. FLETA has been developed in a way that can support various types of Transaction, and also provide various basic Transaction to include; Create Account Transaction, Transfer Transaction, Burn Transaction, Deposit Transaction, and Withdraw Transaction. Through this Transaction, Account is created, coins are transferred, and Contract is operated to change Account Data and UTXO. Additionally, it is possible to withdraw using UTXO or deposit using Account in UTXO. The type of Transaction can be expanded by each DApp, and details are elaborated in the Transaction Extension of Architecture section.

Contract

Contract refers to a process in which it is connected with Account and the designated code is operated with the same input to change AccountData accordingly. FLETA protocol is designed to support various types of Contract and provides a great deal of basic Contract to include; Solidity, RDBMS, NoSQL, and Event Sourcing. AccountData is changed using Solidity, and EVM (Ethereum Virtual Machine) which runs it. Thus, the same result always comes out because every chain that made its way through the same block has the same AccountData. The type of Contract can be expanded by each DApp, and details are elaborated in the Transaction Extension of Architecture section.

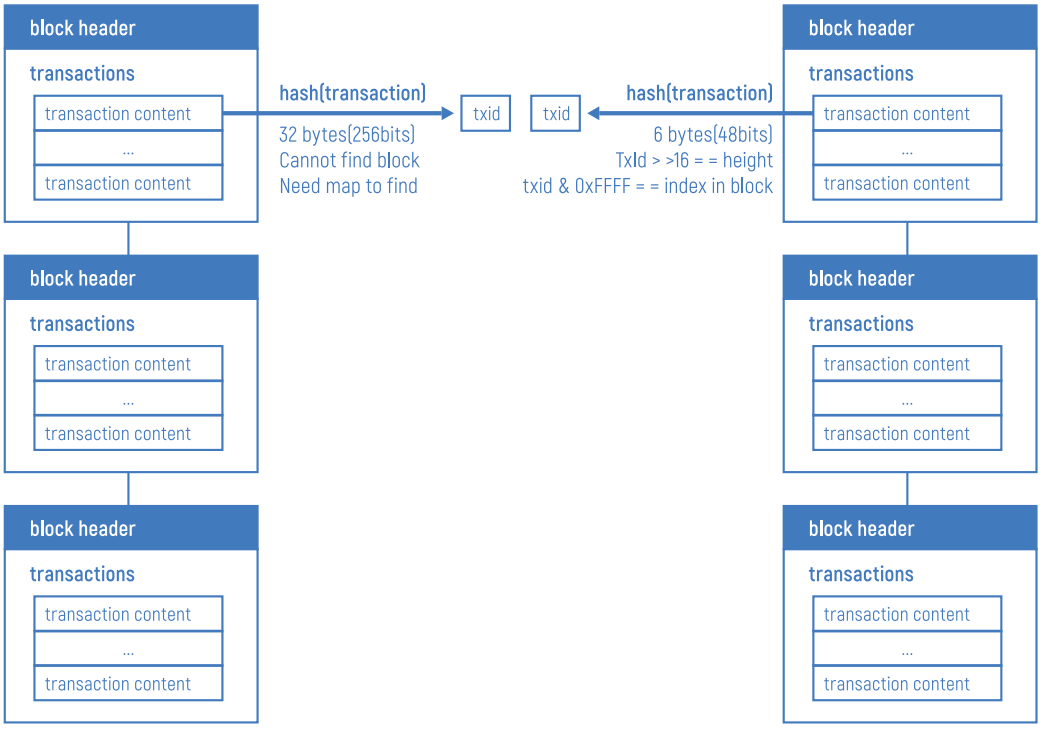
TxId in UTXO

In the UTXO model, a conventional TxId uses Hash regarding transactions, and the signature is processed with this TxId. This method, however, requires a dedicated index because the locations of the block and the transactions are not immediately detected with TxId. This, in turn, leads to the corresponding index becoming larger, and simultaneously 32 bytes is required of Vin entry value.

With the aim of reducing transaction size and enabling immediate tracing of the location of the block and the transaction, the format of TxId in FLETA is composed of 6 Bytes with the following structure:

TxId : {Height(4), Index(2)}

This structure consists of 4 Bytes of Height (block height) and 2 Bytes of Transaction Index (location of transaction within a block). This way, block height can be easily obtained using Shift Operation of 32 bits, and the location of the transaction within a block can be obtained through Shift Operation of 16 bits and Bits Masking. All transactions are referenced using a completed blockchain and therefore, they all end up having the same value even without using transaction hash.



Block

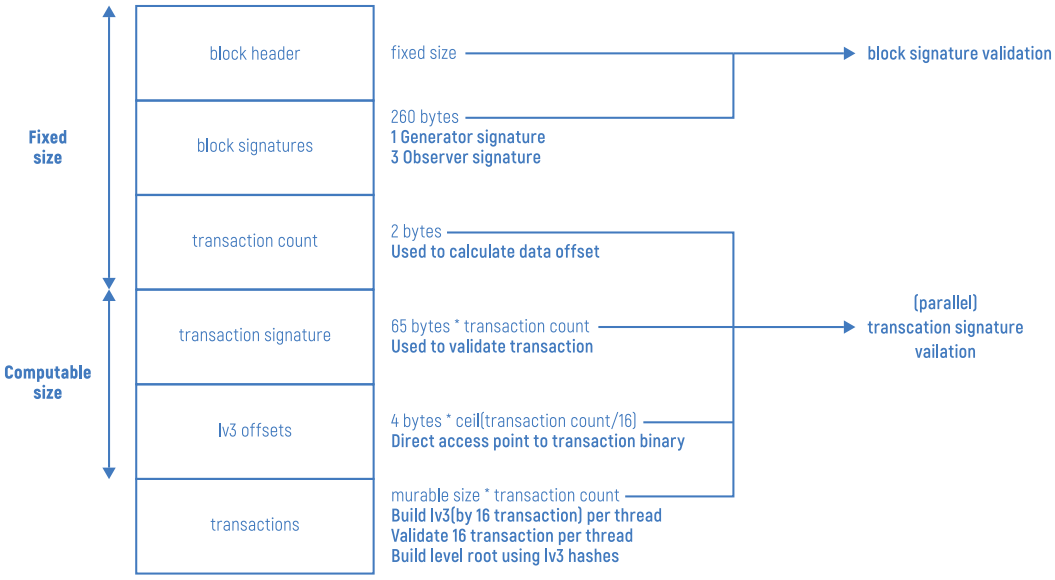
A block consists of a block header, a transaction and a signature. In the traditional header, transactions of the previous block and the Merkle Tree root hash using Txld are included. Merkle Tree, however, has an inefficient computational structure, making it difficult to verify and exchange Light Node data with a simple transaction list. Therefore, we at FLETA, replaced the Merkle Tree validation with Level Tree validation. Merkle Tree is often used in P2P network systems to detect changes when sending data, but it requires the entire tree to function. In reality, the block is received from a single node and the Merkel Tree's size is almost equivalent. It is thereby difficult to use it for partial verification using P2P data transmission, which is the same as simply performing additional SHA 256 from a different angle.

FLETA's block consists of a block header and a transaction list, using a Level structure to support Light Node and parallel processing. The basic block structure is:

Block: {BlockHeader, TransactionSignature[], Transaction[], BlockSignature}

BlockHeader: {Version, HashPrevBlock, HashLevelRoot, Timestamp, Timeout, FormulationAddress}

The Level Tree is a hexadecimal tree structure binding 16 transaction hashes and using the hash of that bind once again. In other words, the Level Tree is a structure in which the maximum number of inscribable transactions per block is 65535, and each level designed to have 16 offspring. Thus, there are levels 1, 2, and 3 (Lv1, Lv2, Lv3). In the block header, HashLevelRoot using 16 Lv1s is inscribed; Lv1 is a Hash value using 16 Lv2s; and Lv3 uses a hashed value occurring from 16 concatenated Hashes. When forming the hash, HashFunction (Hash1 + 8bits + Hash2 + 8bits ... + 8bits + Hash16)—a function that hashes the values concatenated by inserting a designated pattern of 8bits padding between hash values—is used so as to improve speed and reduce the possibility of falsification and tampering.



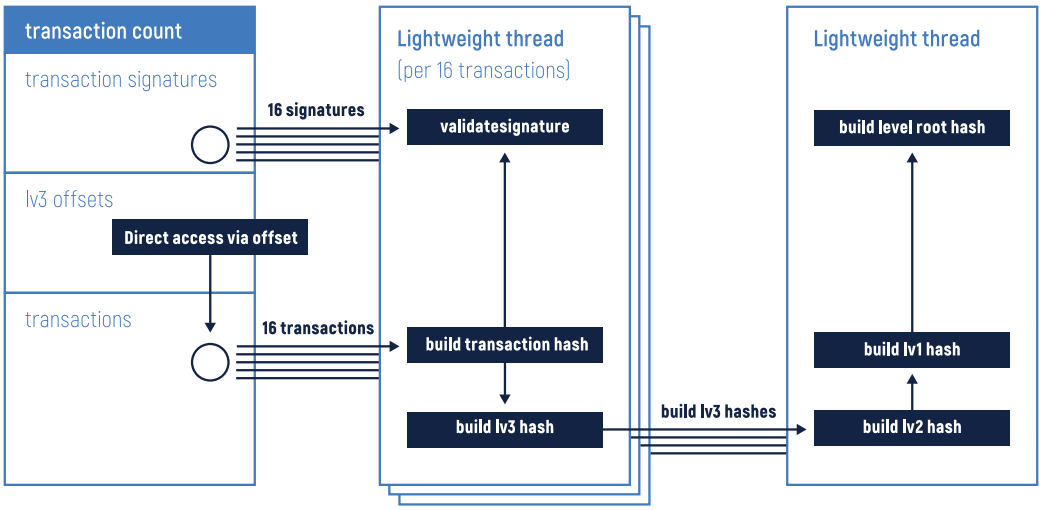
Serialization of the next block is designed to expedite verification of blocks in parallel, with the following structure:

BlockSerialization: {BlockHeader, BlockSignature, TransactionCount, Level3Indexes, TransactionSignatures, Transactions}

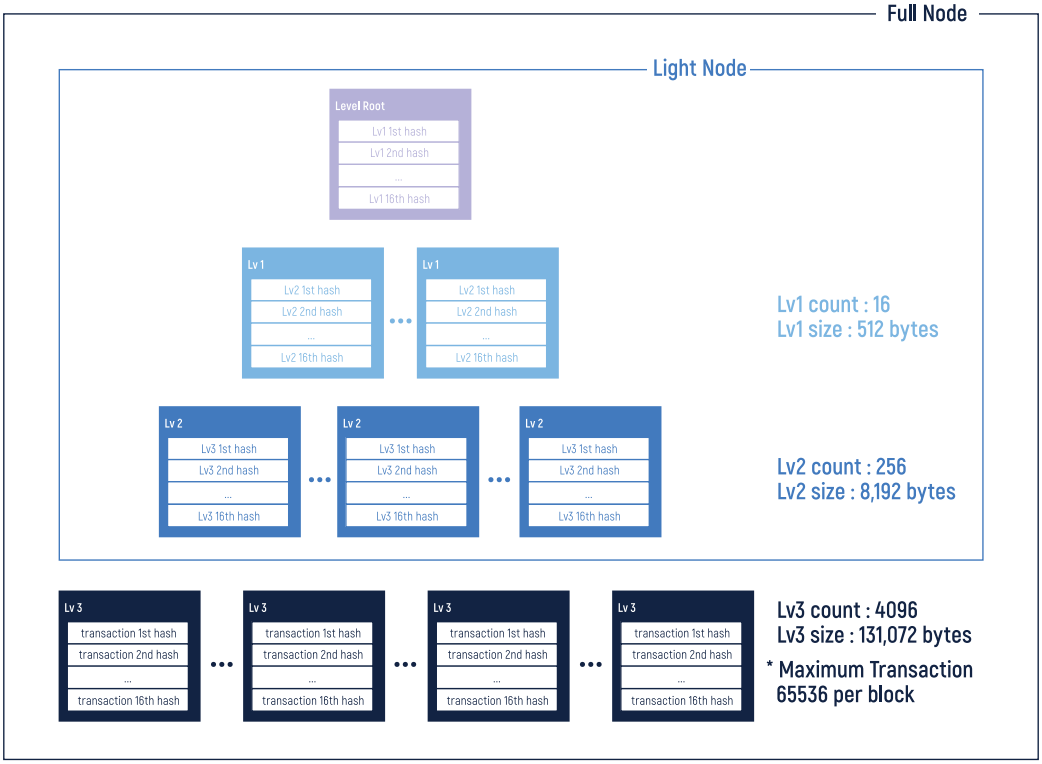
BlockSignature : {CreatorSignature, Signatures[9]}

First off, the BlockHeader and the TransactionCount here are of fixed sizes; Level3Indexes and TransactionSignature are of fixed sizes proportional to the TransactionCount. This allows for immediate checks of binary positions of Transactions, and using Leve3Indexes allows for a quick parallel verification of binary data since hexadecimal-bound transactions can be located immediately. BlockSignature — a signature using hash value for BlockHeader—consists of the signature of the generator, the block generator group, and the observer node. Block generation is performed by the block generator, reward order performed by the generator and the block generator group, and the content is confirmed by the observer node. Here, the blocks with observer node signatures (those that have completed verification) are also transmitted with the TransactionSignatures input; the individual node also verifies both the transaction and the signature, preventing erroneous transactions by design.

Validation



Overall consistency of a transaction can be verified by constructing a level with a transaction's hash and comparing it to the HashLevelRoot. The validation of the signature can be verified by comparing the signature, transaction, and level in parallel by dividing the TransactionSignatures in 16s and dividing the transaction in threads of 16 using Level3indexes. Since all such functions are read operations, they can be carried out simultaneously. Upon receiving a block, verifying transactions by 16 and composing Lv3 from the hash from verification and verifying by Level Tree throughout.

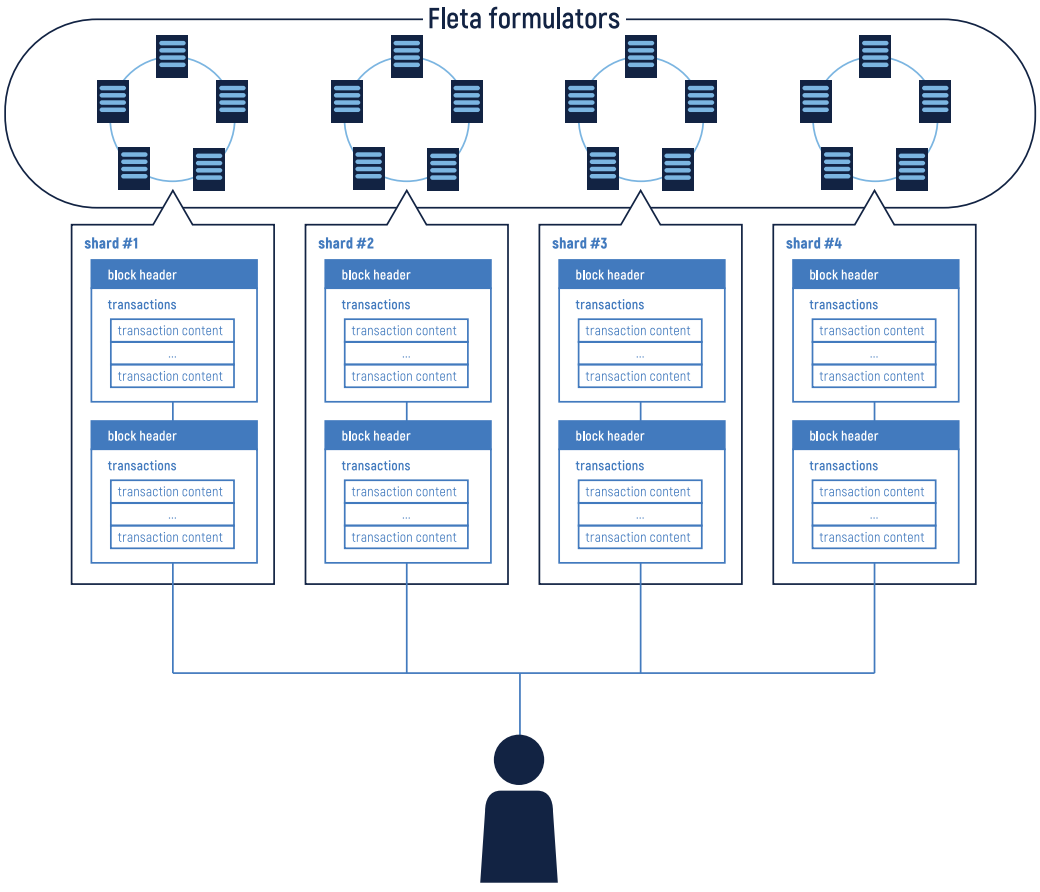


The level structure is also advantageous for the verification of the Light Node. The Light Node has 512 Bytes per Level 1 and 8192 Bytes per Level 2, so the Light Node can contain 8880 Bytes of validation data per block including the block header. If it is necessary to have data on a particular transaction, the Height will be displayed in the TxId, so the block can be immediately known, and since the position is displayed on the Index, knowing which Level Tree node contains a particular transaction is immediately possible. By importing 16 transactions that correspond to a tree in Level3 (512 bytes and 3600 Bytes respectively) the contents can be verified through the tree structure. Therefore, a lightweight node can perform high level transaction verifications with low data reception.

Sharding (Multinode Parallel Transaction Processing)

Currently, two forms of sharding exist in FLETA: **1)** partitioning data to store it and **2)** partitioning a transaction to process it.

The sharding described below refers to processing a transaction in parallel by using multiple nodes as shards.



Multiple shard chain with one address

Sharding

FLETA regards each shard as an independent blockchain, operating in a fully parallel fashion. However, a user's public and private keys can be viewed and used as if they were processed as a single entity, using the same value regardless of the shard. Using this method, we at FLETA came up with a complete parallel shard mechanism that from the beginning, has no chance of double spending.

INDEPENDENT MULTICHAIN STRUCTURE

FLETA is a blockchain platform with an independent multichain structure. Each DApp operates its own chain. Only when additional issue of token for ICO takes place or interchain function is used, does it operate in conjunction with the main chain. That is, other than such functions, each DApp's chain is maintained independently. By independently composing a formulator using the DApp token, the DApp's chain continues to operate even if the main chain ceases to operate.

Token Chain

Token Chain

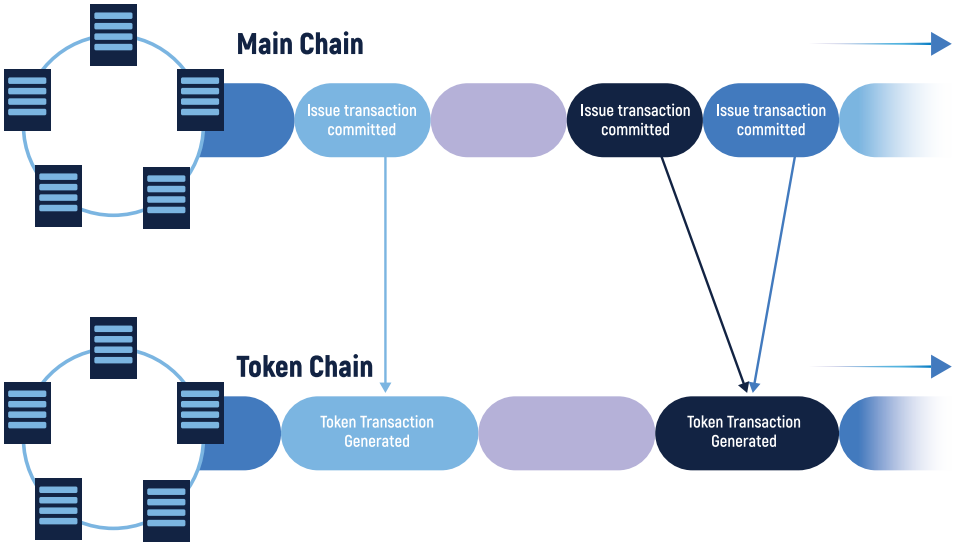
FLETA's DApp has to issue each token, which enables an independent Token Chain. By setting the Genesis information, such as total quantity of token, account composition, observer node public key, seed node, IP, Lockup, selling of token, and performing TokenCreation Transaction, the Token Account is created. Such information can be used to designate Token Chain node, comprising a network. Until this point is the early stages of the composition and following this, that chain separates itself from the mainchain, with its blocks operating. Smart Contract of DApp operates in each of the Token Chain, resolving the execution fee or overload problems in the prior mechanisms where different DApps overlapped. Token Chain takes the block from mainchain in order to issue the token and also for the sake of interchain function. During this process, TokenIssueTransaction that needs to be done in Token Chain is processed, recording the point processed in block header and issuing tokens.

Token Issue

Token Issue

TokenIssueTransaction can be used when token sale information is in the initial value of the created token. When the user deposits token in Token Account through TokenIssueTransaction, it is verified and authenticated via that sale information. Token Chain processes the information that needs to be processed within the Token Chain, out of all the mainchain transactions. This is the point when real tokens are issued.

In summary, the mechanism by which FLETA exchanges token, is the token goes into the token administrator's address, concurrently leading to the issuing of TokenIssueTranscation. Token Chain verifies the Issue Transaction and provides the amount in the address to which deposit is made, by creating designated tokens.



Token Issue

Interchain Communication

Interchain Communication

Basically, DApps of FLETA are operated as independent blockchains. Thus, in order to support interchain communication among DApps, interchain technology is required. Interchain technology is operated in the process by which each DApp chain regularly reports its block header to the mainchain, leading to the inscription of that information. This allows tokens to be transferred from one DApp to another. The transferred token is completely deleted in the chain that sent it, and the chain that receives the token takes the newest block header information from the mainchain. It also approaches the chain with Light Node, receiving tokens and finalizing the process to create tokens. This means that a chain could be in possession of different types of token and that Smart Contract can be operated via such different types of tokens. Only tokens that are authenticated can be transferred to DApps that authenticate the act of receiving tokens from other DApps. Then, the execution fee and token to be paid, is set. Such authentication is done by the founder of Token Account, as the founder issues TokenAllowanceTransaction in the very TokenAccount.

CONSENSUS : Proof-of-Formulation

U.S.P(United States Patent) Application Number : 62717695

Consensus refers to a common understanding on block generation, in particular it signifies who generates the next block or who chooses the blocks out of the generated blocks in the chain process. The prior consensus used a method that disseminated blocks throughout the network for the arbitrary users to mine. However, this requires high recovery of confirmation or block time, as miners are able to generate subsequent blocks only when the new blocks are disseminated throughout the whole network. As a way to deal with this problem, only a select number of miners were picked in order to achieve lower block time.

FLETA has come up with a PoF (Proof-of-Formulator), allowing fast generation and dissemination of blocks by using Formulator reward sequence to designate the mining target and narrow down the dissemination range. Additionally, the existence of observer node allows immediate authentication and prevents fork of blocks. Anyone can make the formulator, so the door is open to all. Low block time can be achieved as the mining sequence of the formulator is fixed, making the dissemination range of new blocks very small.

Rank Table

Rank Table

RankTable calculates the score on all FormulationAccount and ranks the scores. All node has a RankTable and because the score is calculated through transaction and chain height, the list is the same. The authority to generate new blocks is given to the Formulator with the highest rank. When the block is generated and thus included in the score, the sequence changes and the authority can go to another.

Score of RankTable consists of Phase and Hash. Phase is a value related to time, showing how many times the RankTable has turned. The new formulator always participates in the RankTable with a LargestPhase+1 value. After the generation of blocks, the formulator's phase is increased so that reasonable sequence is secured. The details are as follows:

Score : uint64(Phase) << 32 + uint64(binary.LittenEndian.Uint32(hash[:4]))

This signifies that every Formulator is guaranteed with one mining opportunity in every phase and different formulator sequence is provided in different phrases, in order to prevent potential attack or collusion of and against Formulators.

Connectivity

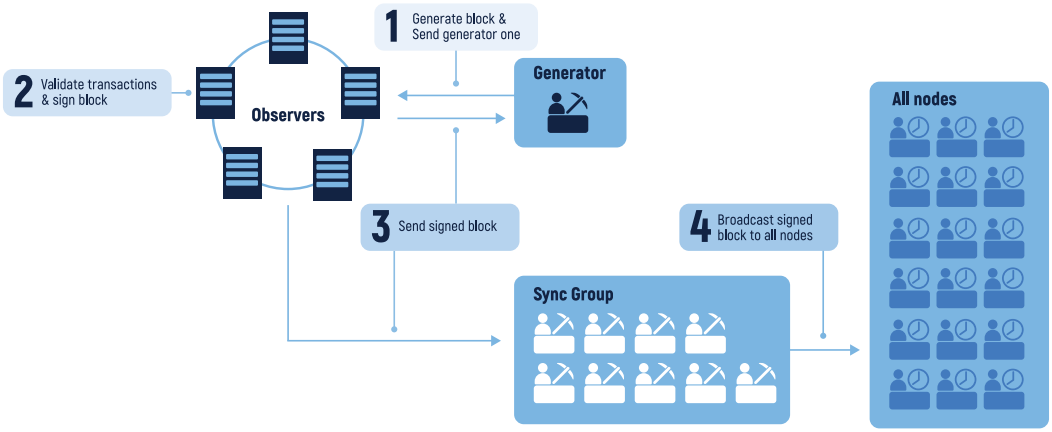
Connectivity

In order to prepare for DDoS attack, every formulator accesses the observer node, therefore hiding the IP of Formulators and maintaining systematic sequence and process. Thus, observer node assumes the responsibility of all costs of protecting DDoS and security, Observer node is able to provide protection with high efficiency and with less cost, as it consists of relatively smaller amount. This, in turn, enables observer node to receive real-time information about the Formulator's activity. The observer node can increase transparency by revealing node status and structure information to formulators and users. If the turn comes for the unconnected node, TimeoutCount can play its role to continue the mining process by excluding such unconnected nodes. Formulators whose turn is skipped, are aware of such, making users able to monitor with ease.

Block Generation

Block Generation

The block generation is performed according to the agreed generation order among the formulators, and the block reward goes to the formulator that generated it. The block generation order is synchronized using the aforementioned formulator synchronization. Within a balanced network the algorithm is managed and directly (or via multiple peers) assesses connections to make agreements of block generation order correspond across the network. Block generation is only possible by the first ranked node here and since a signature is required, only the first ranked node can create forked blocks. This means that with the observer nodes confirming in real time, forks will never occur.



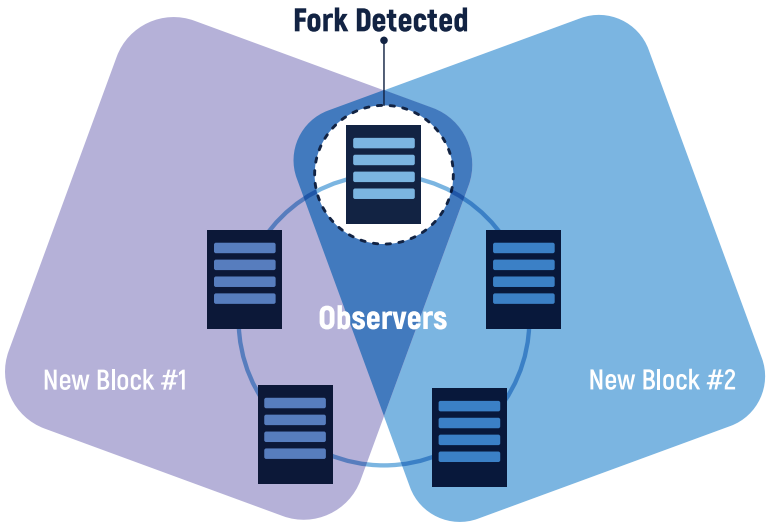
The mining group consists of **A)** the 1st place generator group, **B)** a synchronization group consisted of 2nd to the 10th place, and **C)** a standby group consisted of 11th to 20th place.
The synchronization group agrees on the compensation order, and the observer node performs the content verifications. In other words, the block generator generates a block, sends the generated block to the synchronization group and the observer node, and the synchronization node confirms the generator sequence and header, and proceeds to sign it, and sends it to the observer node.

The observer node receives 6 signatures from the synchronization group and reviews all transaction signatures in the block and exchanges signatures between observer nodes. If three signatures from the five observer nodes are collected, the block is complete and the observer node sends the completed signature part to the synchronization group. The synchronization group creates a completed block by attaching the signature to the previously received block and sends it to the standby group, which then distributes the block to the network. The block generator in this fashion can quickly generate a block, and since 3/5 of the observer nodes signed it, a fork is not possible as at least one observer node will detect the fork before it forms. And since synchronization group proceeds to sign the sequence for verification purposes, a biased observer node signature is prevented. And the role division of synchronization and standby group divides the transmission traffic while ensuring that the block propagates as fast as possible throughout the network.

To expedite mining, the 1st rank node can send both the 2nd rank and the observer nodes the generated block so that the 2nd rank can be prepared in advance. Of course, if the block sent by the first rank has a problem or fails to sign, the recipient node will discard the flawed block and prepare for a new one. This acts as a catalyst for expedited signing if the generated block is without problems. If the first rank node fails to generate a normal block within 1 second, the second order node generates a new block on its own to be safe. If afterwards the first rank node still fails to create a regular node for over 3 seconds, the 2nd rank immediately propagates its created block and continues with block generation. The observer node confirms that the first rank did not create for more than 3 seconds and proceeds with the signing process.

Fork Prevention

When the highest rank formulator generates a block and receives the signatures of the observer nodes, the observer nodes sign and store the block. When the signature is signed by the synchronization group, it receives the block and the blockchain progresses so that if a fork block occurs, it cannot go past the observer node, preventing a fork from happening by design. The concept is that when the formulator order is correctly configured, the 1st rank node only has the right to generate and sign the block, at which phase making two or more blocks to fork the blockchain will be stopped by the observer nodes. Therefore, if the formulator rank order is synchronized, it is possible to only receive blocks that are not forked, simply by verifying the block generator and observer node signatures. The generated block therefore is decisive, and all transactions approved by the observer node are immediately confirmed.



3/5 sign prevent chain from forking

Through the implementation of observer nodes, the attacker cannot create fork blocks to induce double payments. Furthermore, since the subject of block generation is a formulator, blockchain maintenance is also done by individuals who created the formulators, and since the observer nodes requires no compensation, the reward is solely given to the individuals in possession of the formulators.

ARCHITECTURE : Microkernal System

FLETA's system architecture has a Microkernal system. Kernel constitutes blockchain such as Consensus, Store, Generator and takes care of their operation, so that each chain supports their own account, transaction and contract. If DApps intend to add new account, transaction and contract, that type is added when employing adequate method of generation, verification, operation to the designated interface like accounter.Register Account. It is separated so that it can be added in plugin, not source code, and is developed in a way that each chain is given the option of selecting necessary types out of all the registered types.

The fundamentals of Microkernal approach at the heart of FLETA's perspective on blockchain. Blockchain is consisted of the following processes: exchanging each other's data into P2P, all storing the same history and authenticating through Hash Chaining, managing authority of generation/change/deletion of data through Digital Signature, and changing data by proceeding history through consensus. Microkernal has a structure of abstracting **1)** transaction, which is the really changed formula, **2)** account, which stores and manages transactions, **3)** contract, which carries it out with codes. Other procedures are all processed in Microkernal. Transaction modifies Account, AccountData, UTXO through Context, thereby operating Account and Contract in accordance with that modification.

Context based Approach

Transactor, which processes Transaction, enables Transaction to define Validate and Execute. Also, using the functions of Account, AccountData through Context and the functions of Snapshot, Revert, Commit through UTXO, the process of checking, changing and restoring is made, allowing that Context to be reflected in Store. This renews the status of the chain. In this approach, Context only operates in memory and inscribes as FileSync Mode in Store, ultimately securing the process and preventing data to be stopped or corrupt during the process.

Context based Approach is not a structure where only one Transaction type is applied and interpreted as the occasion demands. Rather, its structure processes various types of transactions and through it, Store can be updated. What it signifies is that DApps of FLETA can easily develop new functions in the blockchain environment, simply by delegating structure processing to Microkernel and attaining additional functions to control Context. Such blockchain structure lays the groundwork for FLETA and FLETA DApps to develop and expand services with ease.

Account Extension

Account is managed by Accounter, who stores and manages data. It is subdivided into two processes. The first is registering the new type in Global. In order to use the first process, the second needs to be taken care of, which is to create Accounter Instance of that chain, and to inscribe the type registered in Global with details, and execution fees. FormulationAccount for Formulators, SingleAccount and MultiSigAccount which carry out basic transactions, are added this way.

To make an Account not for managing the Amount of Coins, but also for storing and exchanging data, three additions are required. The addition of Account type to constitute Data Map, the addition

of DataCreationTransaction and the addition of DataTransferTransaction. If Account is to be processed in the composition like SNS, addition of new Account type and or Transaction types like FriendRequestTransaction, FriendResponseTransaction, SubmitPostTransaction can be developed.

Transaction Extension

Transaction Extension

Transaction is a command used to change data, which is managed through Transactor. It is subdivided into two processes. The first is registering new type in Global. In order to use the first process, the second needs to be taken care of, which is to create Transactor Instance of that chain, and to inscribe the type registered in Global with details, execution fees. FormulationTransaction and RevokeFormulationTransaction which creates and deletes FormulationAccount, TransferTransaction that conducts the basic transaction and BurnTransaction that incinerates, are all added this way.
If Transaction need to be processed over AccountData, which is the Key/Value Store, new Account types can be added and thus processed.

Contract Extension

Contract Extension

Contract is a processor that processes user defined code in the form of Virtual Machine managed through Contractor. It is subdivided into two processes. The first is registering new type in Global. In order to use the first process, the second needs to be taken care of, which is to create Contractor Instance of that instance, and to inscribe the type registered in Global with details, execution fees. SolidityContract that operates Solidity, and RelationDatabaseContract are added this way. This can be used to support new language or process linked to outer software. Outer software has to support managing functions like Snapshot, Revert, Commit, and should only be changed through access in that Contract.

SUPPORTED CONTRACT

FLETA basically provides Solidity, Relational Database, Event Sourcing type Contract. Future development of technology and progress in research could enable new types of Contract.

Solidity

Solidity

Solidity is provided through EVM (Ethereum Virtual Machine). Its composition is changed from Trie based Store in EVM to context-based access, along with the change of providing new codes, deleting the undermined codes by hard fork. Some special commands can be deleted or added, but most Contract is applied without changing codes.

Relational Database

Relational Database

Relational Database is provided by utilizing Database Transaction. First, it manages operation status through Transaction functions, namely Begin, Rollback, and Commit. Only when it is reflected on the chain, Commit is operated and Nested Transaction is used to divide each status and process. Through this, if the code to change such database composes contract, the change will be completed

by blockchain. In order to check, directly connecting to database same form as the existing system development is required.

Event Sourcing

Each smart contract can determine the event to be reviewed and when the event passes the smart contract verification, it is committed and recorded in the blockchain.

Event recorded in the blockchain spreads to each node, and the front-end server that provides the DApp service will subscribe to the relevant event stream and will construct an appropriate form of storage to provide service and data. If an event occurs due to a specific operation, it is transferred to the smart contract, which will then review and commit it. This is the same as performing single point write verification and execution in a distributed environment, so that eventually consistency can be maintained without conflict. When processing a new event, all you have to do is deploy a new smart contract, and if you update an existing event handler, you can update it through migration. Smart contracts can support interfaces regardless of languages as long as they go through a common interface and get verified, so we are currently considering support for various languages.

NETWORK

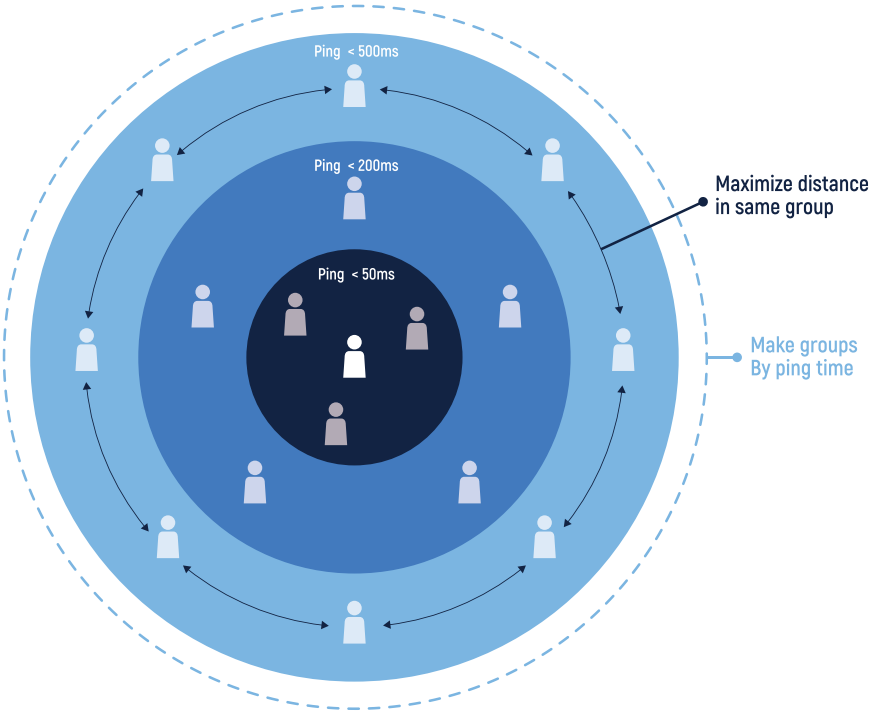
Peer Algorithm : Geologically Balanced Peer Group

The network is maintained in a way that the peer list is shared by the nodes in a P2P network format. Without the existence of peer list, it connects to the seed node to updates the peer list. Keeping the regular node peers and the formulator peers separately, the network maintains the neighboring node peers and the remote node peers evenly to minimize the possibility of a lopsided network. For optimization, it will continuously search for new peers and select a peer within allowable distance to make the network as evenly distributed as possible.

The solution for network lopsidedness is to make a distance estimation using ping and when receiving a new peer also makes the ping reception from the sender so that the overall ping distance on the network spreads evenly. The method of spreading the ping distance evenly forms the group according to the ping value and maximizes the difference in distance between the groups. This process will evenly distribute neighboring and remote nodes in order to reduce lopsidedness in distance, and within a peer group of similar distance, it will try to select the farthest peer to prevent angular lopsidedness.

Peer-to-peer communication method uses TCP/IP communication with TLS, and the packet protocol is as follows.

Packet Protocol : {MagicWord(8), Compression(8), Size(32), Payload, Integrity(*)}



Geological balanced peer group

The above protocol allows performing communication with minimal overhead while maintaining security through TLS. Payload can be compressed by gzip using length and other attributes. Size means the Payload size, and if compressed, it means the compressed size. CRC, Parity Sign or Hash can be used, which can verify the contents of Payload. Payload is an optional value, used in data exchange of less physically credible networks.

THANK YOU

FLETA.io

Update 2018.10.17_v2.0