Tagion Technical Paper

Carsten B. Rasmussen and Theis Simonsen

January 21, 2022

Abstract

This paper describes an alternative implementation of a Distributed Ledger Technology (DLT) network compared to a classical network such as Bitcoin. Most other DLT networks use a Proof-of-Work consensus mechanism to secure the Byzantine Fault Tolerance (BFT) of the data storage. In the Tagion network, the BFT is based on the Hashgraph algorithm and data storage on a new type of Distributed Hash Table (DHT) which makes it efficient to maintain a distributed database and guarantee BFT. The Hashgraph algorithm is deterministic and not probabilistic, allowing ordering of transactions. The order of transactions combined with the Lightning Network and the Tagion matching and settlement protocol constitutes a Decentralised Exchange (DEX) protocol on the Tagion network.

To reduce the probability of the network being taken over by an evil group of actors, a new governance model is proposed, which does not rely on a central control or a group of master-nodes.

git@github.com:tagion/docs-paper.git
337cb9d4a70389eb0a34ade61ab57f519added5f
version v1.18-cbr

Contents

1 Introduction

This document describes the Tagion core network.
The Tagion core network includes a byzantine consensus algorithm, a distributed database (DART) with consensus and description of node validators and the basic node selections.

1.1 Network Architecture

The Tagion network architecture consist of a collections of computer nodes which are able to send messages between each other (A ***computer node*** is named ***node*** in this document). Each node can send a message to any other node in the network.

The nodes in the network has different roles and the nodes will change roles over the lifetime depended controlled by the Tagion consensus rules.

The node roles are:

Active Node A Node in this role category takes care of the validations and consensus.

Standby Node A node waiting to be selected as an active node.

Swap Node A node selected to become an active node waiting to be swapped with an ***Active Node***.

Prospect Node A node in this category is waiting to become and 'real' node in the network.

All the nodes communicates via protocol call libp2p [4]. The role of the node actors are selected randomly (***UDR*** section 3.1) according to the node consensus selection rules appendix I.

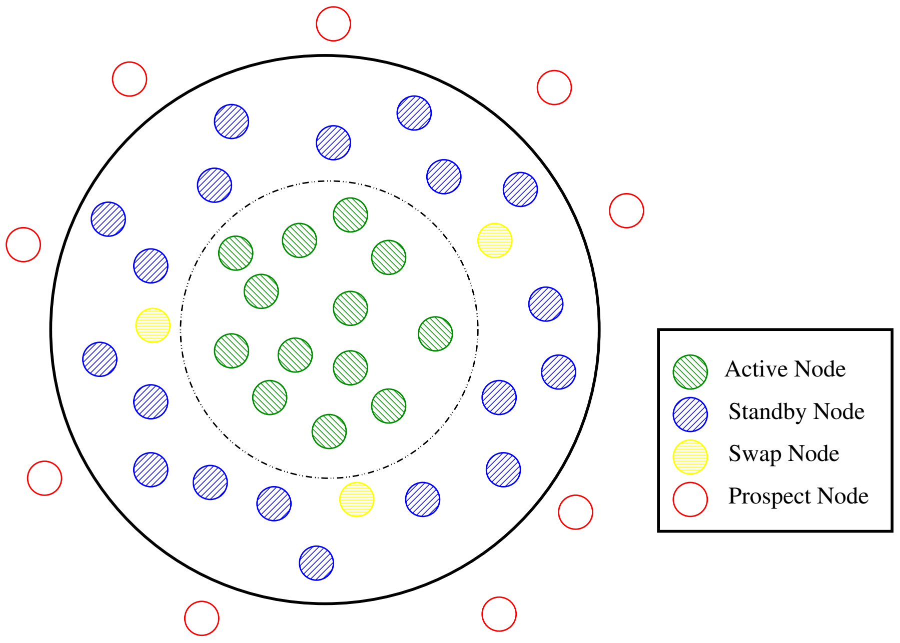

Figure 1: The Network Architecture

The ***Active Node***'s validates a transaction and use the Hashgraph algorithm section 2 to reach consensus. The Hashgraph algorithm is capable of producing a consensus ordered list of

transaction (Defined as ***transaction list***). This means that all the ***Active Node***'s will produces the same order of transactions.
The transaction list is execution in the consensus order appendix C.

1.2 Network security

The Hashgraph algorithm is asynchronous byzantine fault tolerant meaning that if more than 2/3 of the actors are not evil (following protocol) then the network will reach consensus. If more than 1/3 and less than 2/3 of the network actors are not evil, then the network cannot reach consensus. If more than 2/3 then the network has been taken over meaning new protocols has been enforced by the majority potentially destroying data integrity.

The mechanical node swapping described in section 1.1 describes this swapping of nodes, which is ***UDR*** random. Static calculation based on different number of Active and Available Nodes are available. The probability of an coordinated attack to take over the network can be estimated from appendix C.

Different censorious for an attack probability has been estimated in the calculation show in table 1. The probability numbers in column "Halt" represents the probability of the network to halt/slowdown and the column "Take-over" represents the probability for that the Evil nodes can coordinate an attack. From numbers it can be seen that the network get more robust against attack when the ration between total nodes and evil and active nodes.

Active Nodes N	Evil Nodes E	Total Nodes M	Halt $p_{1/3}$	Take-over $p_{2/3}$
31	31	101	0.23	$1.7 \cdot 10^{-7}$
31	31	301	$4.5 \cdot 10^{-5}$	$1.2 \cdot 10^{-17}$
31	31	501	$2.8 \cdot 10^{-7}$	$2.5 \cdot 10^{-17}$
37	31	101	0.2	$1.4 \cdot 10^{-9}$
43	31	101	0.21	$1.4 \cdot 10^{-12}$
43	31	301	$1.1 \cdot 10^{-6}$	$1.5 \cdot 10^{-27}$
43	31	501	$1.1 \cdot 10^{-10}$	$3.5 \cdot 10^{-34}$
31	43	501	$1.2 \cdot 10^{-5}$	$4.5 \cdot 10^{-18}$
31	61	501	$3.9 \cdot 10^{-4}$	$3.8 \cdot 10^{-14}$
31	61	1001	$5.4 \cdot 10^{-7}$	$2.1 \cdot 10^{-20}$
31	61	10001	$1.2 \cdot 10^{-17}$	$2.6 \cdot 10^{-41}$

Table 1: Probability of an attack versus nodes

1.3 Node Architecture

The node core program is implemented in the programming language D with some C and Go libraries for crypto, network and virtual engine functions. It is structured, as shown in the figure below.

HiRPC (HiBON) Dataformat for communication	
NODE	
User API - TLS 1.2	P2P Network
Tagion Virtual Machine	
Consensus mechanism : Hashgraph	
Storage : Distributed Database DART	
Blockchain : Epoch Records	

Table 2: Tagion Node stack

A Tagion Node is divided into units as shown in fig. 2 and each unit handles a service function in the following manner:

A smart-contract is sent to the Transaction-service-unit which are fetching the inputs date form the distributed Data-base DART unit (see section 3) and verifying their signatures of the inputs. The DART-unit connects to other DARTs via the P2P-unit. The transaction-unit forwards the smart-contract to the Coordinator-unit and this information is gossiped to the network via the P2P-unit. When the Coordinator receives an event with a smart-contract, the smart-contract contract is verified and executed via the Tagion Virtual Machine ***(TVM)*** unit, and the results of the outputs are verified.
The Coordinator adds it to an event in the Hashgraph and gossips the informations via the P2P-unit to other nodes in the network. When the Coordinator finds an epoch, it make a list of ordered transactions and forwards this list to the Transcript-service-unit. The transcript-unit executes the smart-contracts in order and produces list called an Recorder. A Recorder contains list of DART instructs where the inputs will be removed and outputs added. The Coordinator sends the Recorder to the DART-uint which executes this list. The DART-unit forwards the Recorder to the Recorder-unit and the Recorder adds this to a block-chain.
In the case where network does not reach consensus, the Coordinator will send an undo-instruction to the Recorder-unit.
If the Recorder-unit receives an undo-instruction the recorder will send the undo-Recorder-list to the DART-unit and the DART-unit will perform this action and put the DART in the previous state before the last Epoch. An undo-Recorder-list is defined a Recorder where the order is reversed and the (remove/add) instructions is inverted to (add/remove).

The Logger-unit and the Monitor-unit used for debugging and monitoring the network.

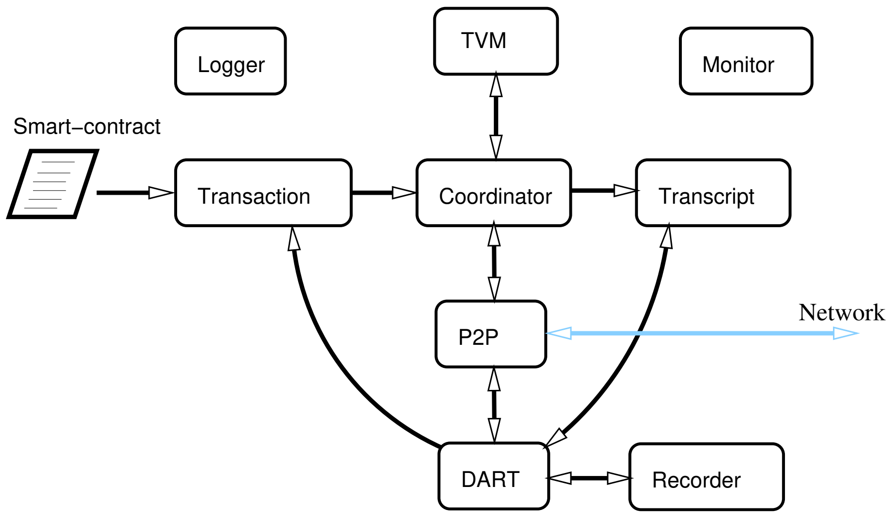

Figure 2: The Tagion Node Architure

Each of the services is running as independent tasks and communication between each-other via commutation channels. The different services modules perform the service as described in the list below.

Coordinator This service manages the hashgraph-consensus and controls other related service for the node. The Coordinator generates and receives events and relays to the network. This service also generates the epoch and sends the information to the TVM services.

Transaction This service receives the incoming transaction script, validates, verifies and fetches the data from the DART and sends the information to the Coordinator.

DART Services to the Distributed-database

P2P This service handles the peer-to-peer communication protocol used to communicate between the nodes

TVM Handles the executions of the smart contracts

Transcript Services the Epoch and orders the smart-contract execution

Recorder This services recorder this history of the DART.

Logger The service handles information logging for the different services

Monitor The Monitor service is used to graphical the activities.

Estimated bandwidth requirement and the average propagation for a transaction the formulas in appendix E.

From simple experimental model result.

Example for a network with nodes $N = 11$ an event size of $E_{size} = 500 bytes$ and network delay of $t_{net} = 300ms$ the estimated epoch propagation delay and the bandwidth of:

$$\begin{aligned} n_{round} &= 2.2 \cdot ln(11) & &\approx 5.27 \\ t_{epoch} &= 3.5 \cdot n_{round} \cdot 300ms & &\approx 5.5s \\ B &= 500bytes \cdot 11^2 & &= 60.5kbytes \\ BW &= 8 \cdot B/(n_{round} \cdot 300ms) & &\approx 28kbit/s \end{aligned}$$

And with $N = 31$:

$$\begin{aligned} n_{round} &= 2.2 \cdot ln(31) & &\approx 8.6 \\ t_{epoch} &= 3.5 \cdot n_{round} \cdot 300ms & &\approx 8s \\ B &= 500bytes \cdot 101^2 & &\approx 481kbytes \\ BW &= 8 \cdot B/(n_{round} \cdot 300ms) & &\approx 152kbit/s \end{aligned}$$

And with $N = 101$:

$$\begin{aligned} n_{round} &= 2.2 \cdot ln(101) & &\approx 10.15 \\ t_{epoch} &= 3.5 \cdot n_{round} \cdot 300ms & &\approx 10.6s \\ B &= 500bytes \cdot 101^2 & &\approx 5.1Mbytes \\ BW &= 8 \cdot B/(n_{round} \cdot 300ms) & &\approx 1.2Mbit/s \end{aligned}$$

And with $N = 1001$:

$$\begin{aligned} n_{round} &= 2.2 \cdot ln(11) & &\approx 15 \\ t_{epoch} &= 3.5 \cdot n_{round} \cdot 300ms & &\approx 16s \\ B &= 500bytes \cdot 1001^2 & &501Mbytes \\ BW &= 8 \cdot B/(n_{round} \cdot 300ms) & &\approx 80Mbit/s \end{aligned}$$

In this example the epoch delay increases round 5s when N is increased by a decade and the bandwidth requirement is increased around 50 times or more.

2 Hashgraph Consensus Mechanism

The Tagion network is based around a Hashgraph consensus algorithm and mathematical proof discovered by Leemon Baird [1]. This algorithm solves the Byzantine Generals' Problem of generating a consensus order list of actions between distributed computer nodes connected in a network. If more than 2/3 of the nodes follows the same consensus rules, all the nodes will, in finite time, reach the same order of events. The network distributes the information via a gossip protocol, sending information about the data received from the other nodes in the network. All the nodes solving the Hashgraph algorithm will come to the same order of transactions. In fig. 3 below shows a Hashgraph of gossip information representing the information flow between network nodes. In finite time all nodes in the network will be able to build the same Hashgraph of gossip information.

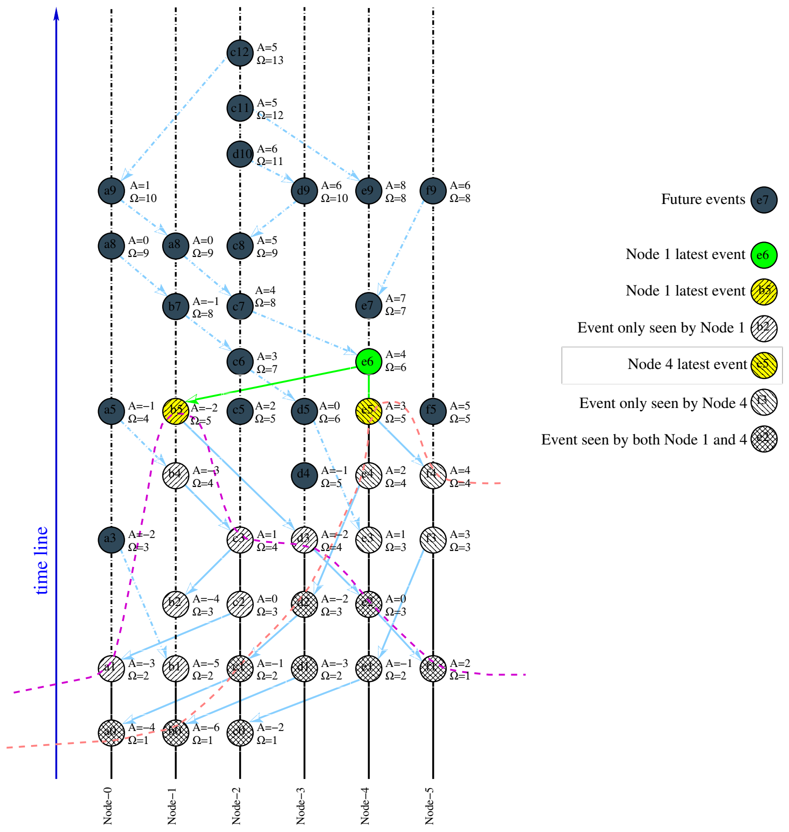

Figure 3: Hashgraph with altitude(A) and order parameter Ω

Each vertical line represents a compute node and each circle an event. The line between the events represents the communication of the events between the nodes. The events coloured red define a witness that is used to divide the consensus into rounds. Each round decides a list of events to be collected. This list of events is called an epoch and must be sorted into the same order as all other nodes. The description of the Hashgraph algorithm can be found in [1].

2.1 Gossip protocol and Wavefront propagation

The Hashgraph algorithm uses a gossip protocol called "gossip about gossip" to propagate information between the nodes. It means node A sends all the information of the communication that it knows to a randomly selected node B. This enables node B to construct the same Hashgraph as node A. (Patent [3])

In the Tagion network, a protocol called Wavefront is used to exchange information between two nodes, ensuring that node A and B only need to communicate three times to share the state of the graph. Each node keeps track of an integer value called Altitude. Altitude is increased by one for each event created by the node. Each node stores its current view of Altitude for each node in the network. By exchanging information about the Altitude between two nodes, both can figure out if their altitude is higher and send a list of events which are in front. The information of altitudes is called an wavefront.

The Wavefront information exchange has four states:

1. Node A selects random Node B and sends a list of all Altitudes. This state is called a tidal-wave.

2. Node B receives a tidal-wave from Node A. If Node B has already sent a tide-wave to Node A, then Node B will send what is called a breaking-wave to Node A. Otherwise Node B will return a list of all events which are in front of the tidal-wave of Node A. This state is called first-wave.

3. If Node A receives a first-wave from Node B, it returns a list of all the events which are in front of Node B. When this state has been reached the wavefront exchange ends.

4. If Node A or Node B receives a breaking-wave, the wavefront communication is dropped. This prevents both nodes from going into an infinity echo where they forever send information back and forth. In the network, a node will often have many simultaneous wavefront connections so it will sometimes receive the same event package from other nodes. Then it will drop any duplicated events it receives.

In fig. 4 the state diagram are shown and which show an communication example of the graph shown in fig. 3 where node number 1 initial the communication.

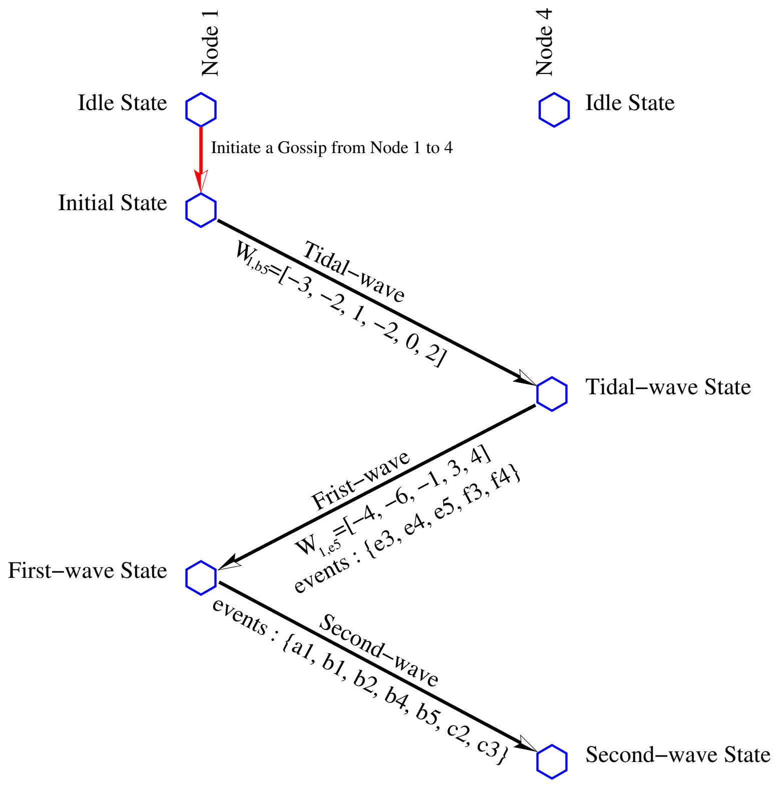

Figure 4: Wavefront communication state

2.2 Consensus Ordering

In the Tagion implementation of the Hashgraph algorithm, an Event is only allowed to point to one or none "other parent" which is called a "father-event" as shown om fig. 5. This strategy aids in solving the graph forking problem and simplifies the consensus ordering. The "self-parent" is defined as a "mother-event" in the Tagion implementation. An event must have a mother-event but doesn't have to have a father-event.

Each event points to the previous event called the mother-event, and some also point to another father-event. The mother-event is defined as the previous event from the same node. The father is an event sent via the gossip network from another node.

The order Ω is calculated as:

$$\Omega_{B,k+1} = max\big(\Omega_{A,k}, \Omega_{B,k}\big) + 1 \tag{1}$$

The events in the epoch list are sorted by the order Ω. If the order of two events is equal, the hash h of the event is used to calculate the order. The following expression is used to order the events:

$$l(A,B) = \begin{cases} l'(A,B) & \text{if } (\Omega_A = \Omega_B) \\ \Omega_A < \Omega_B & \text{otherwise} \end{cases} \tag{2}$$

$$l'(A,B) = \begin{cases} l(A_{mother}, B_{mother}) & \text{if (A has a mother \& B has a mother)} \\ l(A_{father}, B_{father}) & \text{if (A has a father \& B has a father)} \\ 0 & \text{if (A has a no father)} \\ 1 & \text{if (B has a father)} \\ H(h_A \parallel h_B) < H(h_B \parallel h_A) & \text{otherwise} \end{cases} \tag{3}$$

If parameter $l(A,B)$ is $'true'$ if event A is ordered before event B (see appendix G).

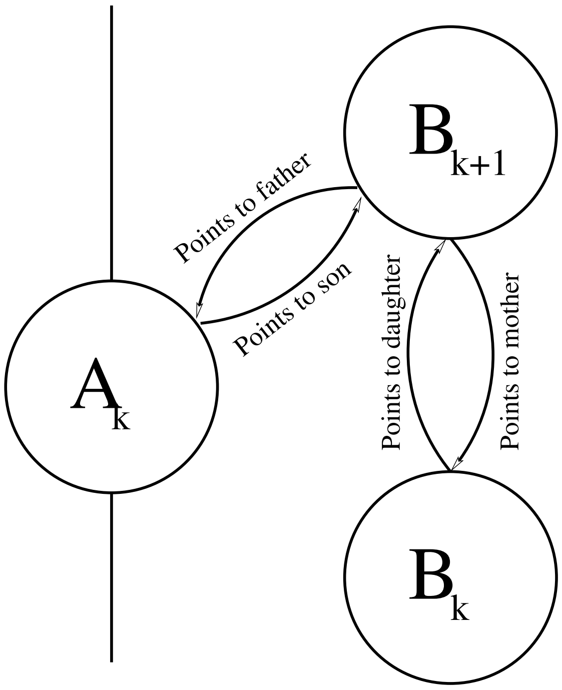

Figure 5: Events and relations

3 Distributed Database (DART)

Distributed Archive of Random Transactions ***(DART)*** [Patent [2]] is built to store and keep track of transactions in the Tagion network. The database efficiently handles the removal and addition of transactions in a secure and distributed manner. Each transaction is stored in a distributed hash-table using a cryptographic hash of the transaction T data. Each transaction is identified by a unique hash value h. The transaction is put into a table ordered by the numerical value of the hash.

$$h = H(T),\ h \in [0 : 2^{N-1} - 1],\ N \in \mathbb{N} \tag{4}$$

$$S_i \in [i \cdot 2^{N-M} : (i+1) \cdot 2^{N-M}],\ i \in [0 : m-1],\ m = 2^M \tag{5}$$

H is the cryptography hash function

N represents the number of bits

M represents the bit witdh of the sector

h hash value

S is sections

m hash-table divided into sections S

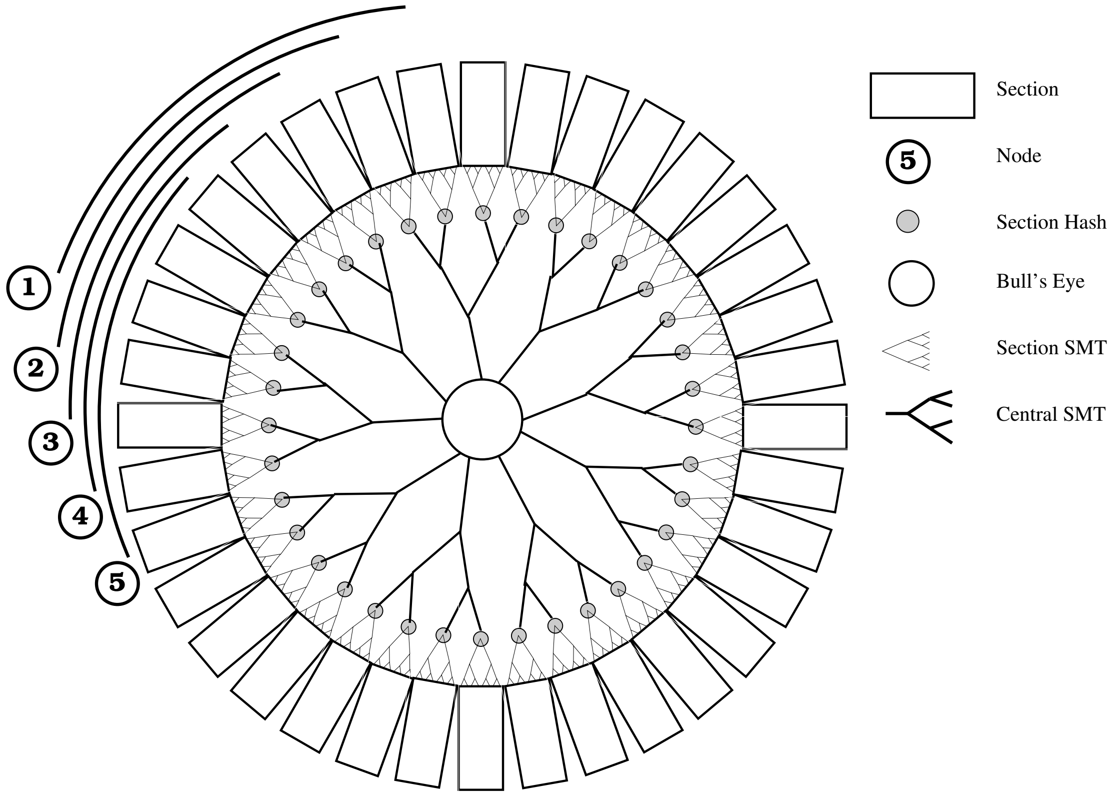

Figure 6: The structure of the DART database

The hash-table is distributed between the nodes in the network, where each node manages a sample of sections. A section must be managed by more than Q nodes to keep redundancy and security of the data. Each node must maintain the database sections within the node's section angle. This means adding and removing the transaction and updating the Merkle-tree root of the section hash. The DART is updated according to the transaction list in an epoch generated by the network. The scripting engine will evaluate actions in the epoch and decide if an archive should be added, removed or selected. The selection of an archive means that the archive is sent back to the network and deleted from the DART. When a node updates a section, it must calculate the Section Merkle-root and sign it and send it to the network. The signed section and the selected archive are distributed to the network via the gossip protocol. Each node will collect all the signed roots of the updated section when the majority has been reached for all updated sections. The node must calculate the Bull's eye (Merkle root) of the DART and sign and distribute the information via the gossip protocol. When the majority of the nodes in the network has reached a consensus of the Bull's eye value, the DART is considered to be updated. If no consensus has been reached for DART, the current transaction in the epoch must be dropped, and the DART must revert to the previous state. The Bull's eye value is stored in a hash-linked chain of blocks where each block points to the previous block's hash. Each block contains the Bull's eye pointer and a block number. It ensures data integrity, the state of the database. The concept is shown in the figure below.

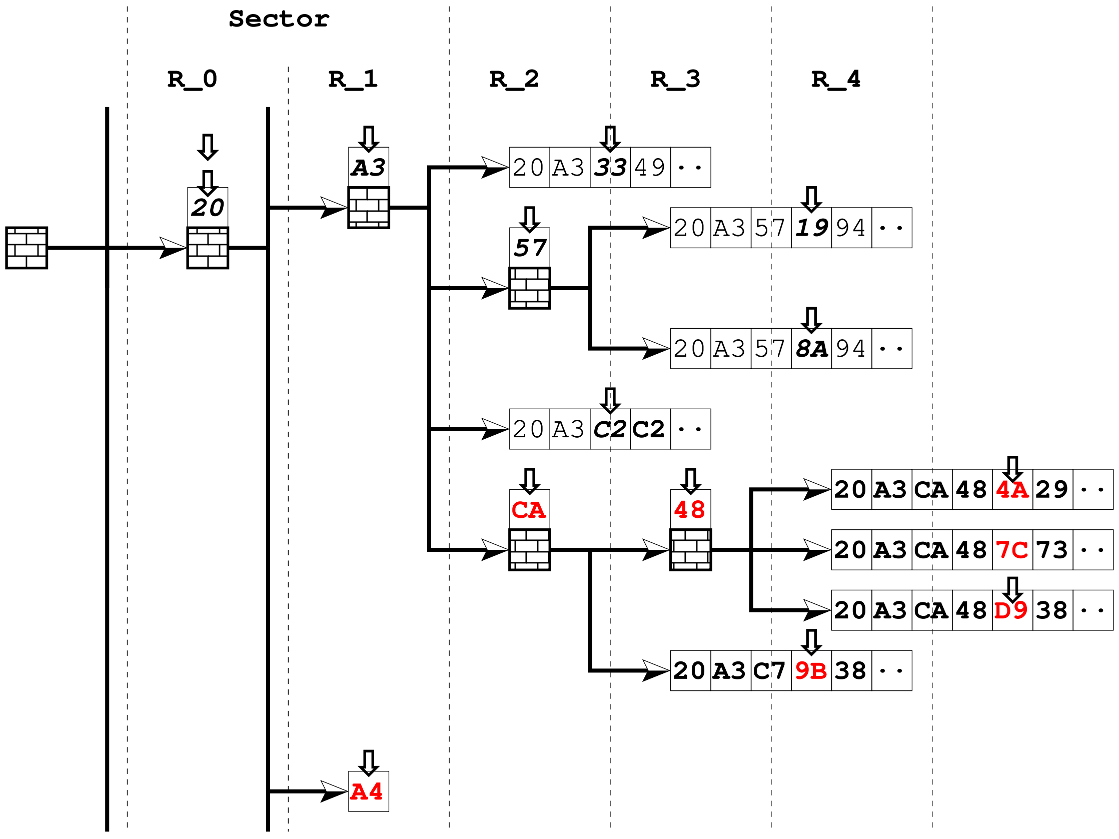

Figure 7: The data structural layout of DART database

3.1 Sparse Merkle Tree

The data in a section is mapped using a Sparse Merkle Tree ***(SMT)*** which makes it efficient to add into and remove archives from the Merkle tree.
The hash point into the DART is divided by rims. Rim zero is the most significant byte (MSB) of the hash fingerprint of the archive. Rim one is the next byte in the hash etc.
In the current implementation of the DART, the first two rims are used as the section index. Thus the DART has a total number of indices $2^{16} = 65536$ which is equivalent to the two bytes unsigned number.
Each section stores the archive in a hash table. An SMT is used as a look-up table, and each rim is sectioned into a sub sparse Merkle tree. This means that rim two is the first SMT and rim three is the second SMT etc.
As a comparison, a traditional Merkle tree with $2^{24} = 8^3 \approx 16 \cdot 10^6$ Archives. Calculating a full Merkle tree requires the calculation of around $32 \cdot 10^6$ hashes. By contrast, using an SMT with $16 \cdot 10^6$ archives mean just around 2000 hashes have to be calculated.

Core protocol updates

The DART will be used for protocol updates by the following consensus. One or more nodes will need to run the new protocol update in a parallel DART containing the same transaction information as the current accepted DART. The new DART will have a new Bull's eye DART Merkle Root ***(Bullseye)***, and this will result in a fork of the Bull's eye chain. When the majority of the nodes run the newly upgraded nodes, they can decide to drop support for the old DART and run the new DART. When enough nodes stop running the old DART, it will not be able to reach a consensus, and the upgrade has completed.

DART garbage collection

A garbage collection script will run every (G) epoch and remove all the bills which are older than a specified date. Bills that have not been used for a long period will be burned, which ensures that the system does not contain dead bills/money. It is the owner's responsibility to recycle their bills before the expiry date.

Unpredictable Deterministic Random

The merkle root ***Bullseye*** is used as seed for the Unpredictable Deterministic Random ***(UDR)***, which is used in the different algorithmic consensus for the network.

4 Special Records

The archives are stored in the DART using the hash-fingerprint as an index-pointer like in Distribute Hash Tabel ***(DHT)*** (See section 3). The hash of an archive is calculated in two ways as follows. If the archive does not contain $\#'param'$ the hash is calculated from the binary data of the HiBON archive and if the $\#'param'$ exists this type of archive is call Parameter Indexed Archive ***(PIA)***. For a PIA the hash-pointer is calculated for the content of the $\#'param'$.

The parameter starting with a dollar sign is reserved for use as system parameters (like $\$'param'$) and should only be used for as such, or else the system will reject it as an error. In particular, $\$type$ is used to set the type of an HiBON object. An PIA-archive must contain a $\$type$ parameter.

Some of the parameters in this special archive has restricted access. The access `ro` means that this parameter is set on the creation of the archive and can not be changed. The `rc` access means that this parameter is controlled and updated by the network and can only be read.

4.1 Name card contract

A Network Name Card ***(NNC)*** is a record which are composed of two archives Name Card Label ***(NCL)*** and Name Card Record ***(NCR)*** both of the archives are stored in the DART.

The NCL label card sets the NNC name and NCR record stores the data related to the name-card (see table 4). The two archives are always updated in pairs in the network.

The hash-pointer of an NCL is calculated for the name-parameter and not for the archives in itself. The NCL can and must only contain the parameter as shown in the table 3.

When an NNC is updated the NCR is updated, the $\$previous$ hash-pointer is set to the previous NCR and the $\$index$ is increased by one. The $\$record$ parameter in NCL is set to the hash-pointer of NCR and $\$sign$ is set to the signature of $\$record$.

The $\$index$ of the first NCR is set to 0, and the $\$previous$ parameter is to hash value of $\$pubkey$ of the NCL archive.

The $\$lang$ sets the type of restricted letters and symbols which is allowed to be used in the NNC name.

Parameter	Description	Type	Access
$\$type$	Set contract type to 'NCL'	`string`	`ro`
$\#name$	Name of the name-card	`string`	`ro`
$\$lang$	Language letter code	`string`	`ro`
$\$time$	Creation date	`utc`	`ro`
$\$pkey$	Public key	`ubyte[]`	`ro`
$\$sign$	Signature of the $\$record$	`ubyte[]`	`rc`
$\$record$	Hash pointer to the NCR archive	`ubyte[]`	`rc`

Table 3: NCL Network Name Card

Parameter	Description	Type	Access
$\$type$	Sets the contract type to 'NCR'	`string`	`ro`
$\$name$	Hash value of the $NCR.\$name$	`ubyte[]`	`ro`
$\$previous$	Hash pointer to the previous NCR	`ubyte[]`	`rc`
$\$index$	Index number	`uint`	`rc`
$\$node$	Optional node record	`#`	`rc`
...	...	...	...

Table 4: NCR Network Name Record

4.2 Node contract

Network Node Record ***(NNR)*** is used to store the node data of the record

Parameter	Description	Type	Access
$\$type$	Set contract type to 'NNR'	`string`	`ro`
$\#node$	Public key for the node	`ubyte[]`	`ro`
$\$name$	Hash value of the $NCR.\$name$	`#`	`ro`
$\$time$	Creation date	`utc`	`ro`
$\$sign$	Signature of $\$name$ by $NCR.\$pkey$	`ubyte[]`	`rc`
$\$state$	The state of the ($\boldsymbol{PN}$, $\boldsymbol{N}$, $\boldsymbol{AN}$)	`uint`	`rc`
$\$gene$	Node gene bit-string	`ubyte[]`	`rc`

Table 5: NNR Network Node Record

4.3 Sub-Network contract

Via a special contract executed on the Tagion Main Network ***(TMN)*** a Tagion Sub Network ***(TSN)*** can be launched. This sub-network will create a new sub-DART which only can be updated from the nodes running the TSN.
A group of nodes can initiate the sub-network by signing this contract and stake an amount in TGS.
The rules for the TSN are set when the network is launched, and the rules can differ from the rules in TMN. The TSN can be assigned to a group of nodes or fully open for all nodes. When a TSN is launched an Tagion Sub Network Funds ***(TSNF)*** is created on the main network to hold the funds for the fees in the main network. The funds are used to pay rewards to the nodes running the TSN and to pay fees to the main. When an epoch is created in the bull's eye for TSN, a contract is automatically sent to the MTN, and a fee is deducted from the TSNF account and burned. If there are not enough funds in the TSNF account, TMN rejects the contract.

TGS used in the TSN must be locked in a Tagion Sub Network Account ***(TSNA)*** these funds can only be transferred between other TSNA and to the TSNF. Funds can not be transferred from TSNF to a TSNA.

The funds in TSNA can be transferred to TGS-bills again via TSNA-contract this contract can take multiple TSNA as input and will transfer all the funds to bills on the output, all the input TSNA will be deleted from DART as is the case with TGS-bills.

Basic rules for a Sub-Network

§1 TSN can have a different rule set then applies for TMN

§2 No nodes can transfer money out of a TSNF account

§3 An TSNF account is used to pay rewards to the TSN nodes and the burning fees to the TMN

§4 An TSNA account funds can be transferred to a TSNF account

§5 An TSNA account can transfer money to other TSNA accounts. The fees burned are a little higher than fees for bills.

§6 The rewards in the TSN are paid from the TSNF account because a TSN can not print TGS as rewards as is the case for TSN.

5 Tagion Virtual Engine

The scripting engine's language is called Funnel. It is based on a stack machine, which is a simple, functional language inspired by the programming language FORTH. The scripting engine executes at different run levels. The lowest level is full Turing equivalent and is only able to make conditional forward jumps; it cannot run loops or functions. The scripting engine is limited by the number of instructions executed, call stack depth, data stack depth and memory.

The limitation is done to prevent a script running into infinite loops. The transaction script can use a library of standard functions which is stored in the DART, and the fingerprint of the script which is stored in the Bull's eye blockchain that is the current state of the script.

Run level	Description	Limitation
0	Consensus script	No limits, full Turing equivalent
1	Debug script function (read-only)	Limit resources, read-only call function to level 0
2	Transaction function	Limit resources and call function to levels 0 and 1
3	Transaction script	Limit instruction and call function to level 2

Table 6: Runlevels for the Scripting engine

In contrast to standard FORTH, Funnel is a strictly typed language which supports the types shown in table 7.
Converting from one type to another must be explicitly instructed via a type casting function. If the casting fails, the scripting engine generates an error and the script stops. The scripting engine stops on overflow/underflow/divide-by-zero errors and if an operator is operating on invalid types.

name	Description	D-Type
TEXT	UTF-8 text	`string`
INTEGER	signend 64-bits number	`long`
CARDINAL	Unsignend 64-bits number	`ulong`
BIG	Unsigend big integer number	`BigUint`
HiBON	HiBON Read/Write-only	`HiBON`
DOCUMENT	HiBON Read only	`Document`
BIN	Byte arrays, used to hold keys and hash value	`ubyte[]`

Table 7: Scripting types supported

Funnel Sample code for a test contract

```

bool run(Document )
variable trans_obj
variable trans_scrip_obj
variable signatures
variable hash_trans_scrp_obj
variable payees
variable payers
variable no_payers
variable no_signatures
variable scrip_eng_obj
variable bills
variable no_bills

: loadtransactionobject
    trans_obj !
    trans_obj @ 'transaction_scripting_object' doc@
    trans_scrip_obj !
    trans_obj @ 'signatures' doc@
    signatures !
    trans_scrip_obj @ hash256
    hash_trans_scrp_obj !
    trans_scrip_obj @ 'payees' doc@
    payees !
    trans_scrip_obj @ 'payers' doc@
    payers !
    payers @ length@ no_payers !
    signatures @ length@ no_signatures !
;

: loadscriptingengineobject
    scrip_eng_obj !
    scrip_eng_obj @ 'bills' doc@
    bills !
    bills @ length@ no_bills !
    scrip_eng_obj @ 'transaction_object' doc@
    loadtransactionobject
;

: get_payee_ownerkey
    local _payee
    local _payees
    local _index
    _index !
    _payees !
    _payees @ _index @ doc@ _payee !
    _payee @ 'ownerkey' doc@
;
```

6 Transaction Scripts

When the network receives a transaction request, it is added in an epoch and executed by the scripting engine. A transaction request includes a transaction object which is a data package in HiBON format. The HiBON object contains input bill numbers and the transaction script including a list of digital signatures which signs the transaction script object. The signatures can be verified via the public keys represented in the input bills.

Parameter	Description	Type	Access
$\$type$	Set contract type to 'B0'	`string`	`ro`
$\$V$	Value	`ulong`	`ro`
$\$k$	Epoch number	`uint`	`ro`
$\$T$	Bill type	`string`	`ro`
$\$Y$	Doubled hashed Owner key	`ubyte[]`	`ro`
...	...	...	...

Table 8: Standard archived Bill object

Parameter	Description	Type	Access
$\$in$	Array of Bill numbers and public keys	`[]`	`ro`
$\$read$	Array of Bill numbers and public keys	`[]`	`ro`
$\$out$	Array of public key hashes	`[]`	`ro`
$\$params$	Parameters used by the script	{}	`ro`
$\$script$	Transaction script	{}	`ro`

Table 9: Transaction scripting object

Parameter	Description	Type	Access
$\$record$	Scripting object	{}	`ro`
$\$signs$	Array of input signatures	`[]`	`ro`

Table 10: Transaction object

Transaction Epoch consensus rules:

1. If one or more script objects is found with the same input bill number, the first transaction object in the epoch is kept in the epoch list. Any other object flows in the list are removed.

Transaction object initial consensus rules:

1. The size of the inputs array in the script record must be one or more.
2. The size of the inputs array and the signature arrays must be the same size.
3. The bill type of the first type input must be a Tagion type.
4. Duplicate bill numbers are not allowed.

5. All the inputs must be in the current state of the DART.

If a transaction object violates one of the initial consensus rules, it is handled by a violation script function.

Transaction scripting execution: Because the epoch list is guaranteed to prevent inputs with same bill number, a node can choose to execute the scripts in the epoch in parallel.

First execution procedure and rules:

1. The bills within the node's DART angle are read from the DART.
2. The read bills are gossiped to the network.
3. If the script object has only one input, the script is immediately executed.
4. If all the bills in the inputs are covered in the local DART, the script is executed immediately.

Second execution procedure and rules:

1. The script is executed if all the inputs are received or read for a transaction object and the signatures are correct.
2. The script must finish with a burn function which burns the transaction fee.
3. If the sum of all outputs of the bill type Tagions (bill type can be Tagions or external contracts of, e.g. Euros) is greater than the sum of the input minus the transaction fee, the first input bill is scheduled to be removed, and the transaction is ignored.
4. If the sum of all outputs of types other than Tagion is greater than the input, the first input bill is scheduled to be removed, and the transaction is ignored.

DART execution procedure:

1. When all scripts have been executed, the process of updating the DART begins.
2. All inputs of successfully executed scripts must be removed from sections covered by the node.
3. All outputs of the successfully executed script must be added to the sections covered by the node.
4. All the Merkle roots within the section angle must be calculated and signed and gossiped to the network. Note: From this point, the node can start executing the next epoch.
5. When the node has received the majority for all the sections' Merkle, it calculates the Bull's eye of the DART, which is signed and gossiped to the network.
6. When the majority of consistent Bull's eyes has been received, the node decides that the DART has been updated and changes states. Note: A transaction has been completed at the new state.
7. If one of the above rules fails, the DART is reverted to the previous state.

Note: When a node receives a transaction object, it can send a request to the DART to collect the inputs of the script. By doing the execution in parallel, it improves the transaction time instead of starting to collect inputs when the epoch has been completed.

7 Business Model

The business model consists of two parts, namely incentives and fee payments. The incentives are given to the nodes for their work and fees are paid by the users for using the system.

Money printing - incentives New money is added to the system when an epoch has been completed, and the DART has reached a consensus. The newly printed money is rewarded to one of the active nodes if it has successfully executed the epoch.
The reward winning node is selected via a UDR Lottery, which is seeded from the bull's eye hash of the DART where the epoch was generated.
The amount is calculated by an economic protocol controlled by the economic governance, see **??**.

Money burning - payment When a transaction is performed in the network, more fees are paid by the user initiating the transaction. The fees depend on storage, the transaction amount and the script execution load. The fees paid to the network are burned; thus, the amount is taken out of the money supply. A storage fee is paid per bytes of the total sum of bytes of all outputs stored in the DART.
A transaction fee is paid as a fraction of the total Tagion amount of the input of the transaction script.
The execution fee is calculated per script instructions where each instruction is priced.
If the total Tagion amount of the output transaction script is less than a specified limit, the whole amount is burned and the transaction is not valid. Fees for decentralised exchanges are described in section 10.

8 Parallelism

Transactions with independent bills can run in parallel, enabling scalability and performance. Independent bills mean that inputs and outputs of transactions are not the same bills. It can run in parallel because the overall design of the data, DART and the scripting engine makes it possible.
The scripting engine is an event-driven engine that executes functions in parallel with inputs and produces outputs locally on each node. Inputs which must be used are read from the DART, and the outputs are stored in the DART. When the transaction successfully completes, the inputs are deleted.

The database is distributed, thus nodes only maintain and keep a copy of the part of the database they are subscribed to, see section 3. Because transactions' inputs and outputs are independent and each node only executes a part of the transactions, they can be executed in parallel and the database updated in parallel as well.
It is not the transaction instructions, which are stored in the database, but the actual bills, which are used as inputs and outputs. Then all nodes do not need to execute all data to verify the integrity of the database as in typical blockchain structures. The consensus event and consensus data are thus merely an intermediate calculation, where the output is stored.

9 Privacy

The current banking system achieves a level of privacy by keeping key information hidden from the public. Under this regime, all identities are known by the trusted third party, i.e. the bank.

In the Tagion system, all transactions and bills are public, but physical identities are separated from transactions and bills. The system has full transparency regarding how many bills exist. A public key is bound to a bill and not an account, and the private key is for signing and spending the bill.

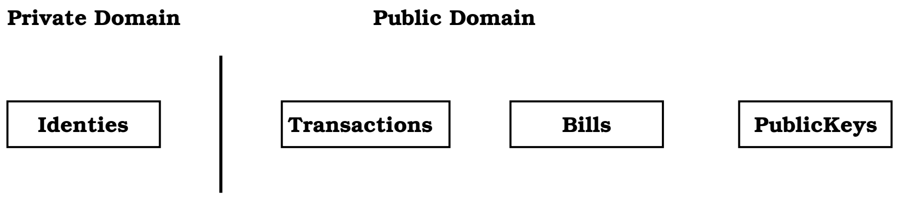

Figure 8: Private and Public domain

Tagion bills are not linked in a chain because each time a bill is spent, a transaction is recorded in the database, deleting the old bill and creating a new one. A full trace of the network will, however, reveal the inputs and outputs of transactions, thus linking the bills. Over time, the bills split and re-combine as they become part of multiple in and out transactions. Therefore, it is not feasible to search back through the linking of bills for a pattern, because it is not a 1:1 trace of bills and would cause an NP (non-polynomial) problem, which cannot be solved in finite time.

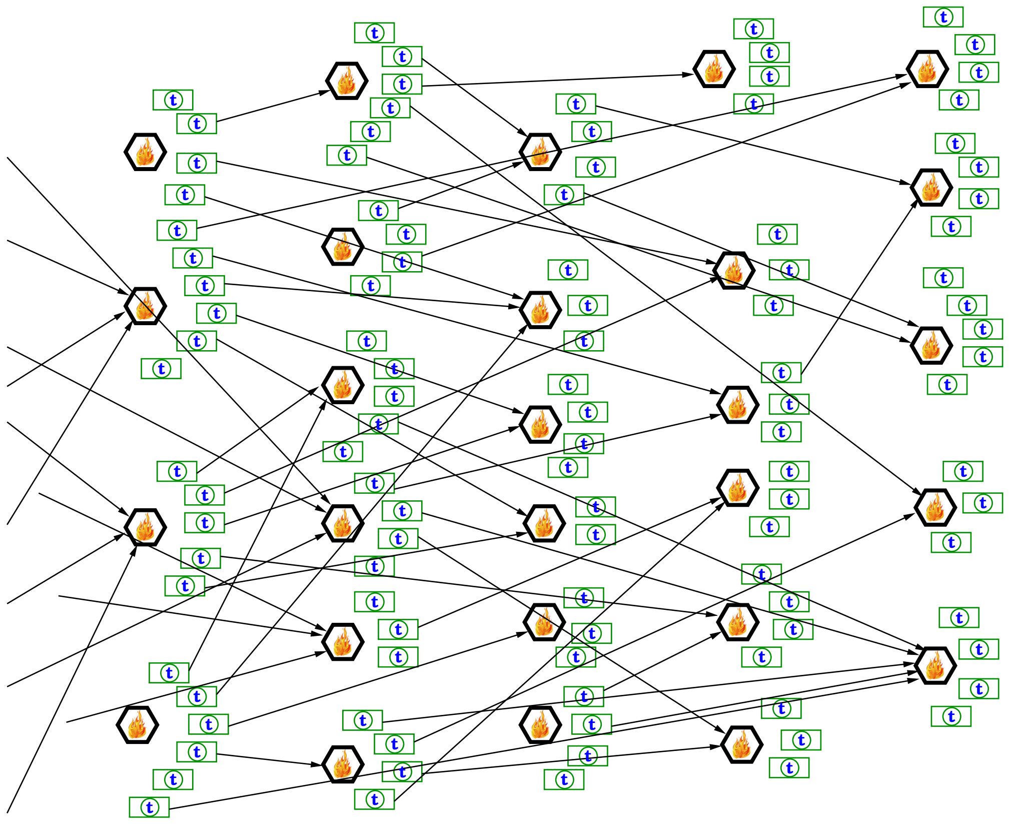
Figure 9: A transaction is represent as a hexagon with a flame the small-bank-note with a **t** represent bills

A user can determine if the same public key should be the owner of all his/her bills or a different, derived, public key. They can hold a different public key for each owned bill, and these keys may not correlate with each other. By using a different public key for each node, a user can make transactions in full privacy, i.e. anonymously.
A node is a public servant and therefore needs to reveal public information. A node in the Tagion system needs to use a fixed public key to ensure the governance of the node. The public key is the identifier for the node that can be perceived as an account, and it is the account for receiving rewards.

10 Decentralised Exchange using Lightning Network

Via a TSN, interfacing other alien Distributed Ledger Technology ***(DLT)*** for exchanges is possible. Most of the current DLTs are based on Prof Of Work ***(POW)*** which secures the immutability of the ledger and has been proven to be very robust. The downside for those types of DLT is the long confirmation time.
A suggested solution to decreasing the confirmation time is to use a second layer solution using a network of payment channels like the Lightning Network ***(LN)***.
Tagion Network ***(TN)*** and LN support each other to enable a Decentralised Exchange ***(DEX)*** for DLT networks which support full-duplex payment channels, Hashed Time Lock Contract ***(HTLC)*** and Multi Signature ***(MultSig)***. The advantage of using the TN as a support system to store intermediate data for the payment channels is that the LN-nodes can share data even if some of the LN nodes go offline. In the current LN use in Bitcoin and other similar networks, the routing between the payment channels is a challenge. One of the reasons for this is that the routing tables are difficult to share and maintain between the nodes. In Tagion, because data is stored in a DART this makes sharing data feasible.
Because funds in an alien DLT can be locked via an HTLC, the alien-currency ***(ALC)*** the funds can be swapped with the native tagion-currency in an atomic manner. This feature enables the Tagion network to support exchange functionality in a decentralised manner providing full liquidity because all exchange pairs have Tagions as the counterpart.

10.1 Trading flow using Lightning and Tagion Network

The Tagion network can order the bids/asks in a Byzantine Fault Tolerant ***(BFT)*** manner. This means that the network is able to come to a consensus on the order of transactions and this solves the matching and prices discovery in a fair manner and solves front-running the order-book in a decentralised manner.

The idea is one STN handles only one trading pair between TGS and ALC. By only using one pair, the matching and routing problem is reduced significantly in comparison to a full multi-currency-DEX with more than two currencies.
First of all, the price discovery is more straightforward, and the amount of data to process is much less if only one pair must be matched and discovered. The second advantage of the one pair DEX is that sub-networks only need to handle one alien smart contract format.

10.1.1 Price discovery and matching

The DEX are able to handle two types of orders as shown in table 12 and table 11. The orders are sent to the TN and at each epoch the trade-order-queue is sorted according to the hashgraph consensus ordering.

Exchange order pairs

ATO Order to buy TGS for ALC

BTO Order to buy ALC for TGS

The matching-engine will maintain two sales-lists of Ask Trade Orders ***(ATO)*** and a Bid Trade Orders ***(BTO)***, those sales-lists are sorted according to the exchange rate with the lowest exchange ratio at to top of the list. A detailed example can be found in appendix F.

The ***ask*** exchange rate is defined as:

$$E_{ask} = \frac{Q}{P} \text{ in unit } [ACL/TGS] \tag{6}$$

The ***bid*** exchange rate is defined as:

$$E_{bid} = \frac{P}{Q} \text{ in unit } [TGS/ACL] \tag{7}$$

P is the price in TGS

Q is the price in ACL

The trade-order-queue is maintained with the orders that are not executed. A new trade-order-queue is generated in each epoch and added in the end of the current trade-order-queue. The matching executes the first in trade-order-queue, i.e. the oldest order is searched first for a match.

The ATO and BTO order in the trade-order-queue are defined as buyers. A match is defined as found, when the buyer's exchange-rate is higher than or equal to the seller's exchange-rate from the corresponding sales-list. The price of the settlement will be set at the seller's exchange-rate.

- When an ATO buys from BTO-sales-list at $E_{ask,BTO}$ price when: $E_{ask,ATO} \geq E_{ask,BTO}$ or the same as $\frac{Q_{ATO}}{P_{ATO}} \geq \frac{Q_{BTO}}{P_{BTO}}$.
- When an BTO buys from ATO-sales-list at $E_{bid,ATO}$ price when: $E_{bid,BTO} \geq E_{bid,ATO}$ or the same as $\frac{P_{BTO}}{Q_{BTO}} \geq \frac{P_{ATO}}{Q_{ATO}}$.

Summarising, a buyer with an ATO-order from the order-queue matches a seller with a BTO-order from the BTO-sales-list and vice versa. Then the size is calculated, and a trading-contract with the corresponding pair is generated and stored in the TN.

If the ATO or the BTO has sold the whole size, then the order is removed from the lists. If an order includes a valid time period of t and if the epoch consensus time is greater than this valid time period t then the order is removed and not executed.

Parameter	Description	Type	Access
$*type*	Set the contract type to 'ATO'	`string`	`ro`
P	Price unit TGS	`ulong`	`ro`
Q	Price unit ACL	`ulong`	`ro`
size	Size of ACL	`ulong`	`ro`
lock	Random Hash-lock key	`bin`	`ro`
time	Valid time period	`utc`	`ro`

Table 11: ATO HiBON to buy TGS for ALC

Parameter	Description	Unit	Access
$\$type$	Sets the contract type to 'BTO'	`string`	`ro`
P	Price unit TGS	`ulong`	`ro`
Q	Price unit ACL	`ulong`	`ro`
$size$	Size og TGS	`ulong`	`ro`
$lock$	Random Hash-lock key	`bin`	`ro`
$time$	Valid time period	`utc`	`ro`

Table 12: BTO HiBON to buy ALC for TGS

10.1.2 Exchange execution rules

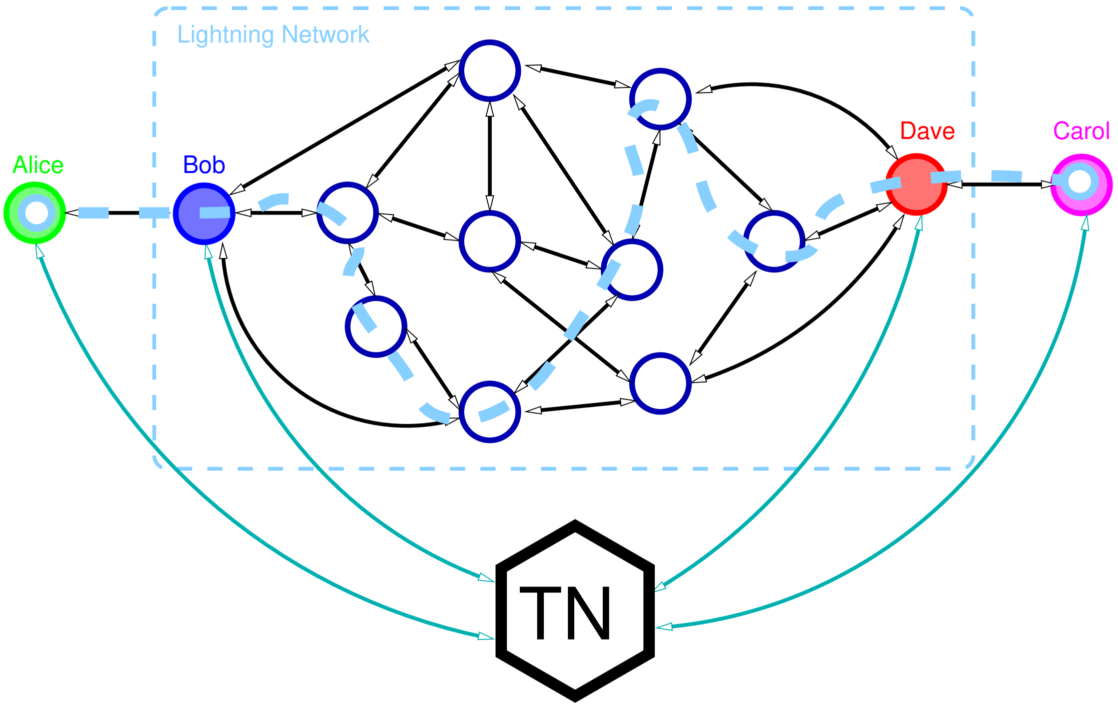

Figure 10: Tagion Decentralised Exchange based on Lightning Network

In the following example, the execution flow of the DEX is described.

Alice wants to trade TGS for ALC (ATO). It can be done as follows.

A.1 **Entry:** Alice opens an LN channel with Bob.

A.2 Alice requests a trading channel from Bob with a guarantee of $\omega_{alice,S}$ in ALC

A.3 Bob locks up a guarantee of an amount $\tau_{bob,S}$ in TGS which matching Alice's $\omega_{alice,S}$ amount. Bob creates an HTLC lock it with R_{bob} and send this information to the TN.

A.4 Alice pays ω_{alice} to the contract lock with R_{bob}.

A.5 **Order:** Alice sends an order to the TN including an HTLC contract to Bob locked with R_{alice} and the amount α_{alice} in ALC. The information includes the bid/ask prices and HTLC contract which is sent to the TN.
Note: *Carol has previously locked funds with R_{calor} in TGS to buy ALC*

A.6 When the TN discoveries a trading pair matching Alice and Carol the network generated a TN-HTLC trading contract locked with both R_{alice} and R_{carol}.

A.7 When Bob verifies that, Carol has to reveal R_{calor}. Bob initials a route between Alice to Carol according to the trading bill. Bob also makes an HTLC return the rest of Alice's funds locked with R_{alice}.

A.8 Alice reveals R_{alice} and the funds can be transferred.

A.9 **Exit:** Alice can ask Bob to reveal R_{bob} and exit the trading channel. Bob's locked funds are returned, and Alice can claim her funds.

Carol wants to trade ALC for TGS (BTO). It can be done as follows

B.1 **Entry:** Carol opens an LN channel with Dave.

B.2 Carol requests a trade channel with Dave, both Carol and Dave guarantee $\tau_{carol,S}$ and $\tau_{dave,S}$ in TGS and hash-locked with R_{carol} in the TN.

B.3 **Order:** Carol sends an order to Dave via the TN.

B.4 Dave receives a confirmation from the network about the order from Carol.

B.5 Dave and creates an HTLC contract to Carol locked with R_{dave} at the amount of $\alpha_{carol,T}$ in ALC.

B.6 When the TN discovers a trading pair matching Alice and Carol, the network generates a TN-HTLC trading contract locked with both R_{alice} and R_{carol}.

B.7 Dave reveals the R_{dave} the Alice can execute the trade.

B.8 **Exit:** Carol can ask Dave to reveal R_{dave} and exit the trading channel. Dave's and Carol's locked funds are returned.

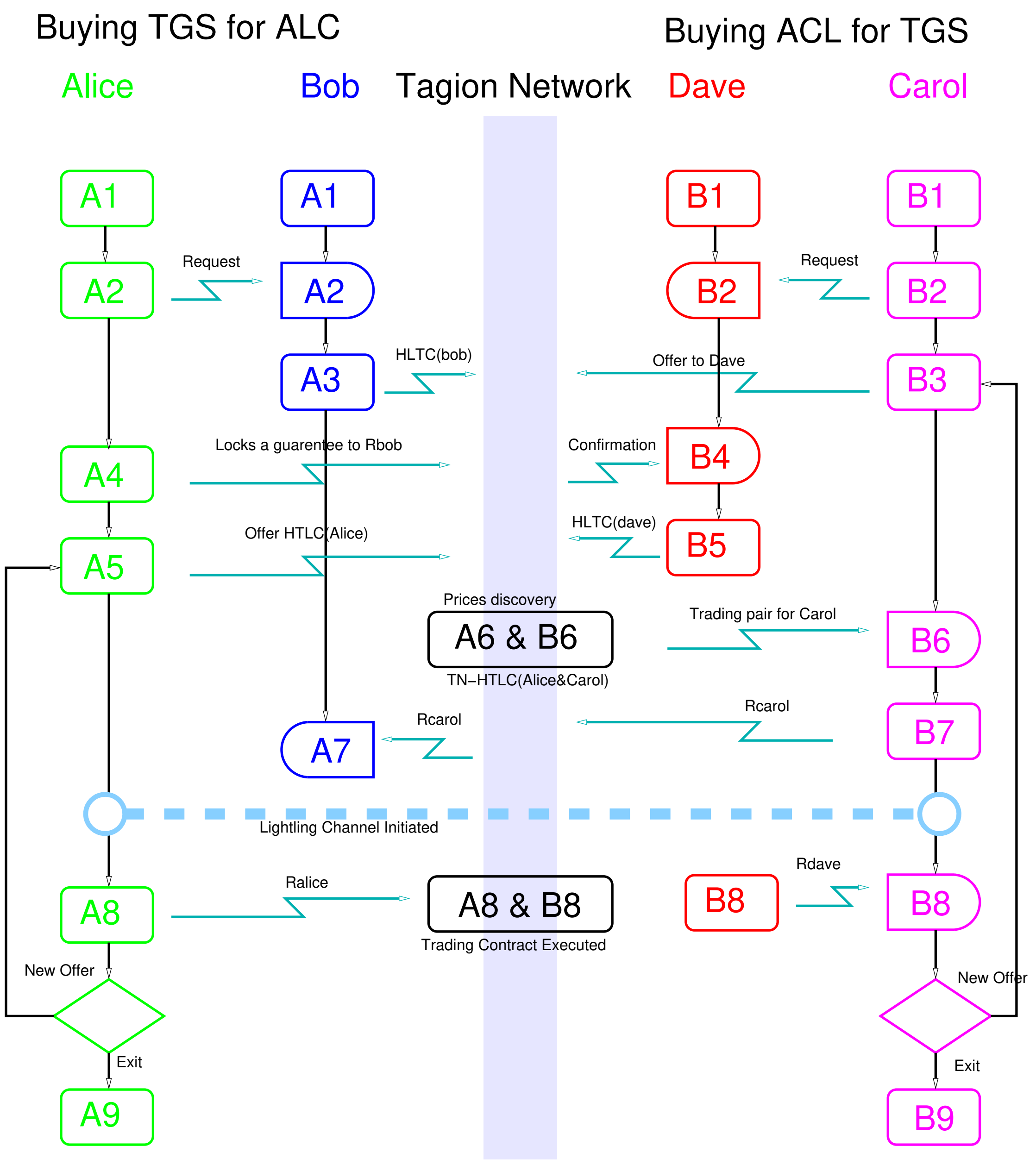

Figure 11: DEX transaction flow

Incentives and Penalty If one or more of the 4 participants (Alice, Bob, Carol and Dave) fails to execute the trade, the flowing penalty rules will be performed by the TN consensus.

§1 Alice doesn't claim the transaction

Incident

- If Alice does not reveal R_{alice} within the timeout limit.

Action

- After a timeout period less than the Alice HLTC time lock.
- The funds will be reverted to Carol.
- The trade is deleted.
- Bob keeps Alice's stacked fund's.
- If the price of Alice's funds is less than Bob's guaranteed funds, Bob gets some of his funds back, which corresponds to the current trading prices.
- The rest of Bob funds is burned.
- If Alice reveals the R_{alice} after the timeout, Alice loses her funds to Carol.

§2 **Bob doesn't initial the routing**

Incident

- If Bob is offline or choses not establish a connection within the a timeout period.

Action

- Bob loses his funds and the trade bill is deleted.
- Carol's funds are returned.
- Alice will get her funds back after the HTLC time lock runs out.

§3 **Carol doesn't claim the transaction**

Incident

- Carol does not reveal the R_{carol} within the timeout limit.

Action

- If Bob makes creates a contract which returns Alice's funds with a time limit Bob gets his funds back.
- Carol's transaction stake is burned and the rest of the funds is returned to Carol.
- The transactions bill is deleted.

§4 **Dave doesn't accept the routing**

Incident

- If Dave is offline or choses not establish connection within the a timeout period.

Action

- After the timeout Carol can reclaim the stack ω_{carol} and open a new channel with Eric.
- Dave's stack ω_{dave} is burned
- Carol can initial the trade by revealing R_{carol}.
- The transaction's bill is deleted after execution.

A HiBON Data format

All data exchanged and stored in the network is structured using a data format called Hash-invariant Binary Object Notation ***(HiBON)*** which is inspired by Binary JSON ***(BSON)***, but the two formats are not compatible. In HiBON the keys are sorted according to the ordering rules described below (in D-lang). By ordering the keys, the data is hash invariant for the same collection.

```
/**
This function decides the order of the HiBON keys
Returns:
true if the value of key a is less than the value of key b
*/
@safe @nogc bool less_than(string a, string b) pure nothrow
    in {
        assert(a.length > 0);
        assert(b.length > 0);
    }
    do {
        uint a_index;
        uint b_index;
        if (is_index(a, a_index) && is_index(b, b_index)) {
            return a_index < b_index;
        }
    return a < b;
    }

/**
Converts from a text to a index
Params:
a = the string to be converted to an index
result = index value
Returns:
true if a is an index
*/
@safe @nogc bool is_index(const(char[]) a, out uint result) pure nothrow {
    import std.conv: to;

    enum MAX_UINT_SIZE = to!string(uint.max).length;
    @nogc @safe static ulong to_ulong(const(char[]) a) pure nothrow {
        ulong result;
        foreach (c; a) {
            result *= 10;
            result += (c - '0');
        }
        return result;
    }

    if (a.length <= MAX_UINT_SIZE) {
        if ((a[0] is '0') && (a.length > 1)) {
            return false;
        }
        foreach (c; a) {
            if ((c < '0') || (c > '9')) {
                return false;
            }
        }
        immutable number = to_ulong(a);
        if (number <= uint.max) {
            result = cast(uint) number;
            return true;
        }
    }
    return false;
}
```

Only printable ASCII keys are allowed to be used as keys in the HiBON; this means no control characters or special characters allowed. The key is validated accordingly to the function described below.

```
/**
Checks if the keys in the range is ordred
Returns:
ture if all keys in the range is ordered
*/
@safe bool is_key_ordered(R)(R range) if (isInputRange!R) {
    string prev_key;
    while (!range.empty) {
        if ((prev_key.length == 0) || (less_than(prev_key, range.front))) {
            prev_key = range.front;
            range.popFront;
        }
        else {
            return false;
        }
    }
    return true;
}
```

Data type	Code	D-Type	Description
float64	0x01	`double`	64bit floating point
string	0x02	`string`	UTF-8 string
Embedded document	0x03	{}	HiBON object
Embedded array	0x04	`[]`	HiBON Array object (Only index numbers allowed)
Boolean	0x08	`bool`	Boolean false=0, true=1
64bits UTC Time	0x09	`utc`	UTC datetime 64bits signed integer
int32 number	0x10	`int`	32bit usigned number
int64 number	0x12	`long`	64bits signed integer
float128	0x13	`decimal`	128bits floating point
Big integer	0x18	`bigint`	Signed big integer
uint32	0x20	`uint`	32bit unsigned number
float32	0x21	`float`	32bit floating point
uint64	0x22	`ulong`	64bit unsigned number
Big integer	0x28	`ubigint`	Unsigned big integer
Native Document	0x43	`Document`	Reserved for internal use only
Defines Array	0x80	`void`	Reserved type for internal use only
Array of float64	0x81	`double[]`	Array of unsigned 64bits integer (size is multiple of 8bytes)
Binary string	0x85	`ubyte[]`	Array of bytes (size is multiple of 1bytes)
Array of int32	0x90	`int[]`	Array of 32bits signed integers (size is multiple of 4bytes)
Array of int64	0x92	`long[]`	Array of 64bits signed integer (size is multiple of 8bytes)
Array of int32	0x90	`int[]`	Array of 32bits integer (size is multiple of 4bytes)
Array of uint64	0x92	`long[]`	Array of 64bits integer (size is multiple of 8bytes)
Array of float128	0x93	`decimal[]`	Array of 128bits floating point (size is multiple of 16bytes)
Array of uint32	0xA0	`uint[]`	Array of unsigned 32bits integer (size is multiple of 4bytes)
Array of float32	0xA1	`float[]`	Array of 32bits floating point (size is multiple of 4bytes)
Array of int64	0xA2	`ulong[]`	Array of unsigned 64bits integer (size is multiple of 8bytes)
Defines string arrays	0x83	`string[]`	Reserved type for internal use only
Defines Document arrays	0xC3	`Document[`	Reserved type for internal use only
Defines HiBON arrays	0x82	`HiBON[]`	Reserved type for internal use only

Table 13: HiBON Basic data-types

Any data types which are not defined in table 13 are illegal and must be rejected by the network. The types used in the table are primarily the types used in D except for a few as {} and `[]`.

A.1 HiRPC – HiBON Remote Procedure Call

Hash invariant Remote Procedure Call ***(HiRPC)*** works like JSON-RPC just with signed binary data, the above-defined HiBON format. It means the data is hash-invariant enabling hash- and signature functions to be executed quickly and unambiguously.

Parameter	Description	Type	Access
$\$type$	Set contract type to 'HiRPC'	`string`	`ro`
$\$pkey$	Public key	`bin`	`ro`
$\$sign$	Signature of $\$msg$	`bin`	`ro`
$\$msg$	Message object table 15, table 16, table 17	{}	`ro`

Table 14: HiRPC format

Parameter	Description	Type	Access
$\$id$	Message id	`uint`	`ro`
$\$method$	Name of remote call function	`string`	`ro`
$\$params$	Params for the $method function (optional)	{}	`ro`

Table 15: HiRPC method message object

Parameter	Description	Type	Access
$\$id$	Message id	`uint`	`ro`
$\$result$	Result of the $\$method$ call	{}	`ro`

Table 16: HiRPC success message object

Parameter	Description	Type	Access
$\$id$	Message id	`string`	`ro`
$\$msg$	Error object table 18	{}	`ro`

Table 17: HiPPC error response object

Parameter	Description	Type	Access
$\$code$	Set contract type to 'HiRPC'	`uint`	`ro`
$\$msg$	Error message	`string`	`ro`
$\$data$	Data object (optional)	`[]`	`ro`

Table 18: HiRPC error object

B Crypto “Bank” bill

The bill has a value V, public/private key (y, x) and the bank bill number B, which is the hash of the bill.
V can have the value of a natural number: $V \in N$
This is a newly printed bill:

$$Y_{alice} = H(L_{k+1} \parallel y_{alice}) \tag{8}$$
$$B_{k+1} = H(V \parallel L_{k+1} \parallel t_{k+1} \parallel T \parallel Y_{alice}) \tag{9}$$

(H is a hash function, is the hash of the previous confirmed Bull's eye, is the consensus timestamp of the current Epoch, is the epoch number and T is the contact type)

The total value of all bills of type T must be accounted for.

$$V_{total,k+1} = V_{total,k+1} + V_k \tag{10}$$

Simple transaction Ownership of the bill can be transferred to Bob, if:

- Bob reveals his public key to Alice
- Alice generates a new bill and signs it with her private key

Because of the network fee, the value will be reduced by ΔV.

$$V_{k+1} = V_k - \Delta V \tag{11}$$

If V_{k+1} is negative or zero, the transaction is eliminated and will not generate a new bill.

$$V_{total,k+1} = \begin{cases} V_{total,k} - \Delta V & \text{if } (V_k - \Delta V \geq 0) \\ V_{total,k} - V_k & \text{otherwise} \end{cases} \tag{12}$$

Resulting in:

$$Y_{bob} = H(B_k \parallel y_{bob})$$
$$B_{k+1} = H(V \parallel B_k \parallel t_k \parallel k \parallel T \parallel Y_{bob})$$

The new bill is now written to the DART with key B_{k+1}:

$$V_{k+1}, B_k, t_k, k, T, Y_{bob}$$

The old bill B_k is removed from the DART.

The Split of a crypto bill
A bill can be split into a number of other bills if the combined value of the new nodes matches the original:

$$V_k = \left[\sum_{i=0}^{I-1} V_{k+1,i}\right] + \Delta V \tag{13}$$

Each new bill is generated as:

$$Y_{k+1,i} = H(B_k \parallel y_{k+1,i})$$
$$B_{k+1,i} = H(V_{k+1} \parallel B_k \parallel t_k \parallel k \parallel T \parallel Y_{k+1}),\ y_{k+1,i} \neq y_{k+1,j} \text{ for } (i \neq j)$$

All new bills marked B_{k+1} are stored in the DART as before, and the old bills are removed.
Join or collect bills into one bill.
A number of bills can be collated into one bill if the value adds up as follows:

$$V_a = \left[\sum_{i=0}^{I-1} V_{b,i}\right] + \Delta V \tag{14}$$

The new common bill number is generated by hashing a sorted list of the joined bill numbers.

$$B'_k = H(B_{k,0} \parallel B_{k,1} ... \parallel B_{k,I-1}) \tag{15}$$

The new bill number will be generated:

$$\begin{aligned} Y'_{k+1} &= H(B'_k \parallel y_{k+1}) \\ B_{k+1} &= H(V_a \parallel B'_k \parallel t_k \parallel k \parallel T \parallel Y'_{k+1}) \end{aligned}$$

These new consolidated bills are stored in the DART.

C Network

The following steps are executed in the network for a standard transaction:

1. The transaction object is sent to one of the active nodes (an inactive node should relay the transaction object to an active node).
2. When a node receives a transaction object, its format and the signatures of all the inputs are checked.
3. If the transaction object is valid, it is added to the payload of an event.
4. The event is gossiped to the network.
5. The payload is put into an epoch list in order.
6. The epoch list is processed in the epoch order.
7. All inputs to the transaction are collected from the DART database.
8. The transaction script is executed when all inputs are read from the DART.
9. The output of the transaction scripts is gossiped to the network.
10. When the network reaches consensus on all outputs of the transactions, the DART is updated.
11. The new Merkle root (Bull's eye) of the DART is calculated, and the Bull's eye is gossiped to the network.
12. When the majority of the nodes reach consensus on the Bull's eye, it is added to the DART blockchain. The transaction is now approved.

D Network Security

Network attack surface

In the following, the probability of an evil attack on the network is estimated via a simple model.
The network participants are given by the flowing parameters.

M is the total number of nodes which are available for the network (This includes active and passive nodes, not prospect nodes).

N is the number of active nodes running the networks.

E is the number of nodes controlled by the evil attacker.

n_E is the number of evil nodes among the N active nodes.

The attack scenario is divided into two categories. The first category prevents the network from reaching a consensus, and in the second category, the attacker is able to take over the network and decide the faith what is going which transactions going it to a block.

First category:

If the evil attacker wants to prevent the networks from reaching a consensus, the evil attacker needs more than 1/3 of the active nodes.

$$\frac{n_E}{N} > \frac{1}{3} \tag{16}$$

Second category:

If the evil attacker wants to take over the network, the attacker needs more than 2/3 of the active nodes.

$$\frac{n_E}{N} > \frac{2}{3} \tag{17}$$

The calculated scenario is based on all the N nodes being changed at every epoch. In the real network, this is not the case; only one node is swapped out and in at every 100 epoch. Thus the probability of an evil takeover is significantly lower than this calculation. The model is chosen because it is easy to express mathematically. The active nodes are selected randomly from M, and the probability that the evil attacker controls the first selected node is:

Definition of permutation formula:

$$P(n, r) = \frac{n!}{(n-r)!} \tag{18}$$

Definition of combination formula:

$$C(n, r) = \frac{n!}{(n-r)! \cdot r!} = \frac{P(n, r)}{r!} \tag{19}$$

The probability of an evil node being selected is:

$$p_{E_0} = \frac{E}{M} \tag{20}$$

The probability of selecting an evil node after selecting an evil node at the nth time is:

$$p_{E_n} = \frac{E-n}{M-n} \tag{21}$$

$$p_{G_{n,e}} = 1 - \frac{E-e}{M-n} \tag{22}$$

The probability of constructing an evil network
In this section, the probability of constructing an evil network is calculated.
The network is randomly constructed by selecting N nodes out of M nodes where E nodes are evil.
A network is defined to be evil if the network contains n_E or more evil nodes out of the N active nodes according to formula first and second category formula above.

The probability that n_E nodes out of N nodes are:

$$p_{n_E} \leq \prod_{i=0}^{n_E-1} p_{E_i} \cdot \prod_{i=n_E}^{N_E} p_{G_{i,n_E}} \cdot C(N, n_E) \Bigg|_{N_E=min(E,N)} \tag{23}$$

If $M \gg n_E$ and $E \gg n_E$ the probability can be approximated to:

$$p_{n_E} \approx p_{E_0}^{n_E} \cdot (1-p_{E_0})^{N-n_E} \cdot C(N, n_E) \text{ for } \frac{E-n_E}{M} \approx \frac{E-n_E}{M-n_E} \tag{24}$$

The probability that n_E nodes or more are:

$$p_{n>n_E} = \sum_{i=n_E}^{N_E} p_{iE} \Bigg|_{N_E=min(E,N)} \tag{25}$$

Example if $N = 100$ and $M = 1000$ and the attacker has E nodes, the probability that the attacker can prevent the network from reaching a consensus is:

$M = 31$	$M = 301$	$M = 1001$
$N = 11$	$N = 31$	$N = 101$
$E = 11$	$E = 31$	$E = 101$
$n_E = 4$	$n_E = 11$	$n_E = 34$
$p_{n\geq 4} \leq 0.42$	$p_{n\geq 11} \leq 1.86 \cdot 10^{-5}$	$p_{n\geq 34} \leq 2.86 \cdot 10^{-12}$

For an attacker to take over the network:

$M = 31$	$M = 301$	$M = 1001$
$N = 11$	$N = 31$	$N = 101$
$E = 11$	$E = 31$	$E = 101$
$n_E = 4$	$n_E = 21$	$n_E = 68$
$p_{n\geq 4} \leq 2.0 \cdot 10^{-3}$	$p_{n\geq 21} \leq 1.2 \cdot 10^{-17}$	$p_{n\geq 68} \leq 1.33 \cdot 10^{-54}$

If we have an epoch time of 10 seconds and the probability is 10^{-53} then the evil attacker can take over the network every 10^{46} years or around 10^{36} the current age of the universe.

Note. For a very large number of M and N the probability can be expressed as a logarithm formula to prevent numerical overflow.
Combination expressed as a logarithm formula:

$$\Phi(n,r) = \sum_{k=r+1}^{n} ln(k) - \sum_{k=1}^{r} ln(k)$$
$$C(n,r) = e^{\Phi(n,r)}$$

The probability expresses as a logarithm:

$$\Pi_{E_n} = \left.\left(\sum_{i=0}^{n_E-1} ln(p_{E_i}) - \sum_{i=n_E}^{N_E} ln(1-p_{E_i})\right)\right|_{N_E=min(E,N)}$$
$$p_{E_n} = e^{(\Pi_{E_n}+\Phi(N,n_E))}$$

If $M \gg n_E$ and $E \gg n_E$ the probability can be approximated to:

$$p_{E_n} \approx e^{(n_E \cdot ln(p_{E_0})-(N_E-n_E)\cdot(1-ln(P_{E_0})+\Phi(N,n_E)))}$$
$$\approx e^{((2\cdot n_E-N_E)\cdot ln(p_{E_0})+n_E-N_E+\Phi(N,n_E))} \text{ for } \frac{E-n_E}{M} \approx \frac{E-n_E}{M-n_E}$$

Security conclusion
By having a volume of, e.g. 1000 nodes and 100 active nodes, which could be a possible amount for a network or shard, then the probability is so low that it will probably never occur in practice. Thus, the actual security is that the nodes are decentralised. Therefore, the node governance protocol is the actual security mechanism, because it regulates the uptake of nodes aiming for it to be democratic, meaning both decentralised and one physical person having only one node.

$$\frac{E-n}{M-n} \tag{26}$$
$$\frac{M-(E-e)-n}{M-n} = \frac{M-n}{M-n} - \frac{E-e}{M-n} \tag{27}$$

E Gossip Model

Epoch consensus

In the following the probability for the Hashgraph-algorithm to reach an Epoch consensus within H events consecutive is estimated in a simple model as follows.

To estimate the bandwidth and the propagation delay simple simulation model has been implemented. The network participants are given by the flowing parameters.

H is the number of event for a node to reach Epoch-consensus.

M is the total number of nodes which are available for the network (This includes active and passive nodes, not prospect nodes).

n is the number of nodes which has not yet been seen by the current node.

Nodes N	Round Factor n_{round}	Bandwidth-Scale B
11	5	171
31	8	1396
101	12	14918
1001	16	1498538
10001	20	149868334
100001	24	14996214212

Table 19: Gossip simulation model

From the simulation the following expression can be driven. The average events per round:

$$n_{round} = 2.2 \cdot ln(N) \tag{28}$$

The average estimated time between epochs:

$$t_{epoch} = K \cdot n_{round} \cdot t_{net} \tag{29}$$

Whe the following is defined as:

N is the number of active nodes running the networks.

K is the epoch scaling factor (Typical value of K is typical a little larger than 3)

n_{round} is the average events for between round.

t_{net} is the network delay.

t_{epoch} is the average propagation delay per transaction.

Estimated bandwidth scale-factor:

$$B = E_{size} \cdot N^2 \tag{30}$$

Estimated bandwidth requirement:

$$BW = \frac{B}{n_{round \cdot t_{net}}} \tag{31}$$

B	is the average bandwidth scale factor used by a node.
E_{size}	is the average memory size of an event including transaction data.
BW	is the average bandwidth used by a node.

F DEX Trading Example

An example of the DEX matching and prices-discovery algorithm described in section 10 is shown in the following tables. The trading-order-queue in table 20 and the two sorted sales list are generated and shown in table 21 and table 22.

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
0	ATO	83TGS	147	10	0.0680	14.7000			
1	ATO	6TGS	138	12	0.0870	11.5000			
2	BTO	785ACL	116	10	0.0862	11.6000			
3	ATO	79TGS	149	10	0.0671	14.9000			
4	ATO	4TGS	113	10	0.0885	11.3000			
5	BTO	217ACL	145	11	0.0759	13.1818			
6	BTO	936ACL	115	13	0.1130	8.8462			
7	BTO	205ACL	145	11	0.0759	13.1818			
8	BTO	949ACL	146	10	0.0685	14.6000			
9	BTO	888ACL	117	10	0.0855	11.7000			
10	BTO	587ACL	112	10	0.0893	11.2000			
11	BTO	314ACL	126	13	0.1032	9.6923			
12	BTO	503ACL	118	12	0.1017	9.8333			
13	ATO	72TGS	106	12	0.1132	8.8333			
14	ATO	57TGS	108	13	0.1204	8.3077			
15	BTO	341ACL	131	12	0.0916	10.9167			
16	ATO	15TGS	127	10	0.0787	12.7000			
17	ATO	43TGS	111	11	0.0991	10.0909			
18	BTO	85ACL	136	12	0.0882	11.3333			
19	ATO	29TGS	144	10	0.0694	14.4000			
20	ATO	63TGS	144	14	0.0972	10.2857			
21	BTO	716ACL	134	11	0.0821	12.1818			
22	ATO	42TGS	139	12	0.0863	11.5833			
23	ATO	37TGS	114	13	0.1140	8.7692			
24	ATO	45TGS	136	10	0.0735	13.6000			
25	ATO	44TGS	131	12	0.0916	10.9167			
26	BTO	87ACL	134	10	0.0746	13.4000			
27	BTO	739ACL	146	13	0.0890	11.2308			
28	ATO	16TGS	138	14	0.1014	9.8571			
29	ATO	42TGS	104	10	0.0962	10.4000			
30	ATO	79TGS	101	12	0.1188	8.4167			
31	BTO	725ACL	117	13	0.1111	9.0000			
32	ATO	3TGS	144	13	0.0903	11.0769			
33	BTO	226ACL	144	12	0.0833	12.0000			
34	ATO	30TGS	140	13	0.0929	10.7692			
35	BTO	526ACL	131	14	0.1069	9.3571			

Table 20: DEX Trading-order-queue

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
14	ATO	57TGS	108	13	0.1204	8.3077			
30	ATO	79TGS	101	12	0.1188	8.4167			
23	ATO	37TGS	114	13	0.1140	8.7692			
13	ATO	72TGS	106	12	0.1132	8.8333			
28	ATO	16TGS	138	14	0.1014	9.8571			
17	ATO	43TGS	111	11	0.0991	10.0909			
20	ATO	63TGS	144	14	0.0972	10.2857			
29	ATO	42TGS	104	10	0.0962	10.4000			
34	ATO	30TGS	140	13	0.0929	10.7692			
25	ATO	44TGS	131	12	0.0916	10.9167			
32	ATO	3TGS	144	13	0.0903	11.0769			
4	ATO	4TGS	113	10	0.0885	11.3000			
1	ATO	6TGS	138	12	0.0870	11.5000			
22	ATO	42TGS	139	12	0.0863	11.5833			
16	ATO	15TGS	127	10	0.0787	12.7000			
24	ATO	45TGS	136	10	0.0735	13.6000			
19	ATO	29TGS	144	10	0.0694	14.4000			
0	ATO	83TGS	147	10	0.0680	14.7000			
3	ATO	79TGS	149	10	0.0671	14.9000			

Table 21: Sort list of ATO or the ask-sales list

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
8	BTO	949ACL	146	10	0.0685	14.6000			
26	BTO	87ACL	134	10	0.0746	13.4000			
7	BTO	205ACL	145	11	0.0759	13.1818			
5	BTO	217ACL	145	11	0.0759	13.1818			
21	BTO	716ACL	134	11	0.0821	12.1818			
33	BTO	226ACL	144	12	0.0833	12.0000			
9	BTO	888ACL	117	10	0.0855	11.7000			
2	BTO	785ACL	116	10	0.0862	11.6000			
18	BTO	85ACL	136	12	0.0882	11.3333			
27	BTO	739ACL	146	13	0.0890	11.2308			
10	BTO	587ACL	112	10	0.0893	11.2000			
15	BTO	341ACL	131	12	0.0916	10.9167			
12	BTO	503ACL	118	12	0.1017	9.8333			
11	BTO	314ACL	126	13	0.1032	9.6923			
35	BTO	526ACL	131	14	0.1069	9.3571			
31	BTO	725ACL	117	13	0.1111	9.0000			
6	BTO	936ACL	115	13	0.1130	8.8462			

Table 22: Sort list of BTO or the bid-sales list

F.1 DEX After the Trading Match

The trade is executed from the first order in the queue which is the top of the table 20 and the matching pairs shown in table 23. A BTO order from the table 20 is searched in table 21 to see if $E_{bid,BTO} \geq E_{bid,ATO}$ and if the order is an ATO the table 22 is searched and if $E_{ask,ATO} \geq E_{ask,BTO}$ a match is found. The executed trading-orders are shown in table 24 and the orders which remain are shown in table 27. The parameter ***Id*** shown in the tables, represents the execution order and matching ***Id*** of the trading pairs and ***No*** is the priority order in the trading-order-queue.

Buyer → Seller	No → No	E_{buyer}	E_{seller}	Bought	Sold	Id
ATO→BTO	1→8	0.0870	0.0685	6.00TGS	87.60ACL	1
BTO→ATO	2→14	11.6000	8.3077	473.54ACL	57.00TGS	2
ATO→BTO	4→26	0.0885	0.0746	4.00TGS	53.60ACL	3
BTO→ATO	5→30	13.1818	8.4167	217.00ACL	25.78TGS	4
BTO→ATO	6→23	8.8462	8.7692	324.46ACL	37.00TGS	5
BTO→ATO	7→13	13.1818	8.8333	205.00ACL	23.21TGS	6
BTO→ATO	9→28	11.7000	9.8571	157.71ACL	16.00TGS	7
BTO→ATO	10→17	11.2000	10.0909	433.91ACL	43.00TGS	8
BTO→ATO	15→20	10.9167	10.2857	341.00ACL	33.15TGS	9
BTO→ATO	18→29	11.3333	10.4000	85.00ACL	8.17TGS	10
BTO→ATO	21→34	12.1818	10.7692	323.08ACL	30.00TGS	11
ATO→BTO	22→33	0.0863	0.0833	18.83TGS	226.00ACL	12
ATO→BTO	25→27	0.0916	0.0890	44.00TGS	494.15ACL	13

Table 23: List of the matching pairs

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
1	ATO	6TGS	138	12	0.0870	11.5000	6.00TGS		1
2	BTO	785ACL	116	10	0.0862	11.6000	473.54ACL		2
4	ATO	4TGS	113	10	0.0885	11.3000	4.00TGS		3
5	BTO	217ACL	145	11	0.0759	13.1818	217.00ACL		4
6	BTO	936ACL	115	13	0.1130	8.8462	324.46ACL		5
7	BTO	205ACL	145	11	0.0759	13.1818	205.00ACL		6
8	BTO	949ACL	146	10	0.0685	14.6000		87.60ACL	1
9	BTO	888ACL	117	10	0.0855	11.7000	157.71ACL		7
10	BTO	587ACL	112	10	0.0893	11.2000	433.91ACL		8
13	ATO	72TGS	106	12	0.1132	8.8333		23.21TGS	6
14	ATO	57TGS	108	13	0.1204	8.3077		57.00TGS	2
15	BTO	341ACL	131	12	0.0916	10.9167	341.00ACL		9
17	ATO	43TGS	111	11	0.0991	10.0909		43.00TGS	8
18	BTO	85ACL	136	12	0.0882	11.3333	85.00ACL		10
20	ATO	63TGS	144	14	0.0972	10.2857		33.15TGS	9
21	BTO	716ACL	134	11	0.0821	12.1818	323.08ACL		11
22	ATO	42TGS	139	12	0.0863	11.5833	18.83TGS		12
23	ATO	37TGS	114	13	0.1140	8.7692		37.00TGS	5
25	ATO	44TGS	131	12	0.0916	10.9167	44.00TGS		13
26	BTO	87ACL	134	10	0.0746	13.4000		53.60ACL	3
27	BTO	739ACL	146	13	0.0890	11.2308		494.15ACL	13
28	ATO	16TGS	138	14	0.1014	9.8571		16.00TGS	7
29	ATO	42TGS	104	10	0.0962	10.4000		8.17TGS	10
30	ATO	79TGS	101	12	0.1188	8.4167		25.78TGS	4
33	BTO	226ACL	144	12	0.0833	12.0000		226.00ACL	12
34	ATO	30TGS	140	13	0.0929	10.7692		30.00TGS	11

Table 24: Orders which are matched and executed

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
14	ATO	57TGS	108	13	0.1204	8.3077		57.00TGS	2
30	ATO	79TGS	101	12	0.1188	8.4167		25.78TGS	4
23	ATO	37TGS	114	13	0.1140	8.7692		37.00TGS	5
13	ATO	72TGS	106	12	0.1132	8.8333		23.21TGS	6
28	ATO	16TGS	138	14	0.1014	9.8571		16.00TGS	7
17	ATO	43TGS	111	11	0.0991	10.0909		43.00TGS	8
20	ATO	63TGS	144	14	0.0972	10.2857		33.15TGS	9
29	ATO	42TGS	104	10	0.0962	10.4000		8.17TGS	10
34	ATO	30TGS	140	13	0.0929	10.7692		30.00TGS	11
25	ATO	44TGS	131	12	0.0916	10.9167	44.00TGS		13
4	ATO	4TGS	113	10	0.0885	11.3000	4.00TGS		3
1	ATO	6TGS	138	12	0.0870	11.5000	6.00TGS		1
22	ATO	42TGS	139	12	0.0863	11.5833	18.83TGS		12

Table 25: Sort list of ATO which are executed

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
8	BTO	949ACL	146	10	0.0685	14.6000		87.60ACL	1
26	BTO	87ACL	134	10	0.0746	13.4000		53.60ACL	3
7	BTO	205ACL	145	11	0.0759	13.1818	205.00ACL		6
5	BTO	217ACL	145	11	0.0759	13.1818	217.00ACL		4
21	BTO	716ACL	134	11	0.0821	12.1818	323.08ACL		11
33	BTO	226ACL	144	12	0.0833	12.0000		226.00ACL	12
9	BTO	888ACL	117	10	0.0855	11.7000	157.71ACL		7
2	BTO	785ACL	116	10	0.0862	11.6000	473.54ACL		2
18	BTO	85ACL	136	12	0.0882	11.3333	85.00ACL		10
27	BTO	739ACL	146	13	0.0890	11.2308		494.15ACL	13
10	BTO	587ACL	112	10	0.0893	11.2000	433.91ACL		8
15	BTO	341ACL	131	12	0.0916	10.9167	341.00ACL		9
6	BTO	936ACL	115	13	0.1130	8.8462	324.46ACL		5

Table 26: Sort list of BTO which are executed

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
0	ATO	83TGS	147	10	0.0680	14.7000			
3	ATO	79TGS	149	10	0.0671	14.9000			
11	BTO	314ACL	126	13	0.1032	9.6923			
12	BTO	503ACL	118	12	0.1017	9.8333			
16	ATO	15TGS	127	10	0.0787	12.7000			
19	ATO	29TGS	144	10	0.0694	14.4000			
24	ATO	45TGS	136	10	0.0735	13.6000			
31	BTO	725ACL	117	13	0.1111	9.0000			
32	ATO	3TGS	144	13	0.0903	11.0769			
35	BTO	526ACL	131	14	0.1069	9.3571			

Table 27: DEX Trading order queue of the orders which are not yet executed

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
32	ATO	3TGS	144	13	0.0903	11.0769			
16	ATO	15TGS	127	10	0.0787	12.7000			
24	ATO	45TGS	136	10	0.0735	13.6000			
19	ATO	29TGS	144	10	0.0694	14.4000			
0	ATO	83TGS	147	10	0.0680	14.7000			
3	ATO	79TGS	149	10	0.0671	14.9000			

Table 28: Sort list of ATO which are not yet executed

No	Type	Size	P	Q	E_{ask}	E_{bid}	Bought	Sold	Id
12	BTO	503ACL	118	12	0.1017	9.8333			
11	BTO	314ACL	126	13	0.1032	9.6923			
35	BTO	526ACL	131	14	0.1069	9.3571			
31	BTO	725ACL	117	13	0.1111	9.0000			

Table 29: Sort list of BTO which are not yet executed

G Consensus order function

The code sample below shows the implementation of the consensus order function.

```
bool order_less(const Event a, const Event b) @safe {
    order_compare_iteration_count++;
    if (a.received_order is b.received_order) {
        if (a._mother && b._mother) {
            return order_less(a._mother, b._mother);
        }
        if (a._father && b._father) {
            return order_less(a._father, b._father);
        }
        if (!a._father) {
            return false;
        }
        if (b._father) {
            return true;
        }

        bool rare_less(Buffer a, Buffer b) {
            const ab = hashgraph.hirpc.net.calcHash(a ~ b);
            const ba = hashgraph.hirpc.net.calcHash(b ~ a);
            const A = (BitMask(ab).count);
            const B = (BitMask(ba).count);
            if (A is B) {
                return rare_less(ab, ba);
            }
            return A < B;
        }

        return rare_less(a.fingerprint, b.fingerprint);
    }
    return a.received_order < b.received_order;
}
```

H Gene Distance

Each node has a gene string, which is used to calculate the gene-score of the node. This node-gene is represented as a binary string of bits.

$$\gamma = [b_0, b_1..b_{N-1}] \; b_i \in \mathbb{B} \tag{32}$$

The gene distance between two nodes A and B is calculated as the number of counted '1' of the ***exclusive-or*** between the two bits vectors.

$$\Lambda(\gamma_A, \gamma_B) = \sum_{i=0}^{N-1} (\gamma_{A,i} \otimes \gamma_{B,i}) \tag{33}$$

The total gene score from a node A to all active nodes can be calculated as:

$$\Lambda_{network} = \frac{1}{M} \cdot \sum_{j=0}^{M-1} \Lambda(\gamma_j, \gamma_A) \tag{34}$$

Where M is the number of active nodes in the network.

The gene of the active node is mutated for each epoch via a UDR random number. A random bit select from the N bits is randomly set to '0' or '1'.

Over time the gene-score between the active nodes is reduced, and this will statistically reduce the score compared to the inactive nodes, thereby increasing the probability of an inactive node to be swapped in as an active node.

I Mutation rules

In this sections the algorithm of bill gene mutations is described.

I.1 Mutation base

A mutation base vector R is generated as an UDR bit-vector

$$R = [\mu_0, \mu_1..\mu_{N-1}]\ \mu_i \in \mathbb{B} \tag{35}$$

I.2 Population gene mutation

From a number M of gene vectors T_j a population mutation gene B is defined.

$$B = [\beta_0, \beta_1..\beta_{N-1}]\ \beta_i \in \mathbb{B} \tag{36}$$

For all 1's for each vector is summed, as follows.

$$s_i = \sum_{j=0}^{M-1} t_{j,i},\ t_{j,i} \in \mathbb{B} \tag{37}$$

Where s_i is the sums of 1's for bit i for all vectors T_j and $t_{j,i}$ is the bits in the T_j vectors.
The bits in the population gene is defined as follows.

$$\beta_i = \begin{cases} 1 & \text{if } (2 \cdot s_i > M) \\ \mu_i & \text{if } (2 \cdot s_i = M) \\ 0 & \text{otherwise} \end{cases} \tag{38}$$

Where mu_i is the mutation base for the population M.

I.3 Production gene mutation

From a gene pair a and b the production gene is defined as:

$$\gamma_i = \begin{cases} a_i & \text{if } (\mu_i = 0) \\ b_i & \text{otherwise} \end{cases} \tag{39}$$

And μ_i is the mutation base of the production mutation.

I.4 Transaction mutation

The bill mutation rules is as follows.

B.1 A population gene B is calculated for all inputs

B.2 The genes of the outputs is production mutated with the epoch gene

The epoch gene is generated for all the outputs as follows:

P.1 A population gene P is calculated for all the transaction output genes

P.2 The previous epoch gene E is produced with P to generate a new E gene

The transaction rewards lottery is selected based on the gene distance between the output gene and the current epoch gene.

List of Tables

List of Figures

List of Abbreviations

ALC	alien-currency
ATO	Ask Trade Orders
BFT	Byzantine Fault Tolerant
BSON	Binary JSON
BTO	Bid Trade Orders
Bullseye	DART Merkle Root
DART	Distributed Archive of Random Transactions
DEX	Decentralised Exchange
DHT	Distribute Hash Tabel
DLT	Distributed Ledger Technology
HiBON	Hash-invariant Binary Object Notation
HiRPC	Hash invariant Remote Procedure Call
HTLC	Hashed Time Lock Contract
LN	Lightning Network
MultSig	Multi Signature
NCL	Name Card Label
NCR	Name Card Record
NNC	Network Name Card
NNR	Network Node Record
PIA	Parameter Indexed Archive
POW	Prof Of Work
SMT	Sparse Merkle Tree
TMN	Tagion Main Network
TN	Tagion Network
TSN	Tagion Sub Network
TSNA	Tagion Sub Network Account
TSNF	Tagion Sub Network Funds
TVM	Tagion Virtual Machine
UDR	Unpredictable Deterministic Random

References

[1] LEEMON BAIRD. “THE SWIRLDS HASHGRAPH CONSENSUS ALGORITHM: FAIR, FAST, BYZANTINE FAULT TOLERANCE”. In: (). Accessed: Swirlds web side.

[2] Bleser Rasmussen Carsten. “Sparsed merkle tree method and system for processing sets of data for storing and keeping track of the same in a specific network”. In: *US20210067330A1* (2020). Owner: i25s AG.

[3] Bleser Rasmussen Carsten. “System and a method implementing a directed acyclic graph (dag) consensus algorithm via a gossip protocol”. In: *US20210227027A1* (2020). Owner: i25s AG.

[4] githib team libp2p githib team. “Modular peer-to-peer networking stack”. In: (). URL: `https://github.com/libp2p`.