Tender DHT Specification

July 2020

Decentralized ledgers come with three promises : decentralized, scalable, and secure. Most current platforms come short on at least one of these fronts. For instance, a ledger like Bitcoin gets more and more centralized with time as peers need more resources to keep up with the difficulty of processing each transaction. And yet its transaction throughput is not scalable but fixed. More recent platforms come with a tradeoff between scalability and decentralization: by reserving transaction validation to sophisticated peers, they achieve higher or scalable throughput.

The Tender DHT aims to solve this problem and provide an immutable ledger that is simultaneously decentralized, scalable and secure. And it comes in the form of a common key-management and transaction storage back-end, that can be used by all Tender apps at the same time. These apps include the Tender Identity manager, Tender currencies, and we plan to make its features openly available to build third-party apps on top of this Tender platform.

In order to get past current platforms' limitations and deliver secure decentralized transactions at scale, we introduce a few new elements: dispatch proofs,

1. Goals

The Tender DHT aims to be a both a public ledger for apps to store, retrieve data, and a peer-to-peer network, so that apps can make peers communicate with each other.

It also needs to scale efficiently.

We need it to have the following features:

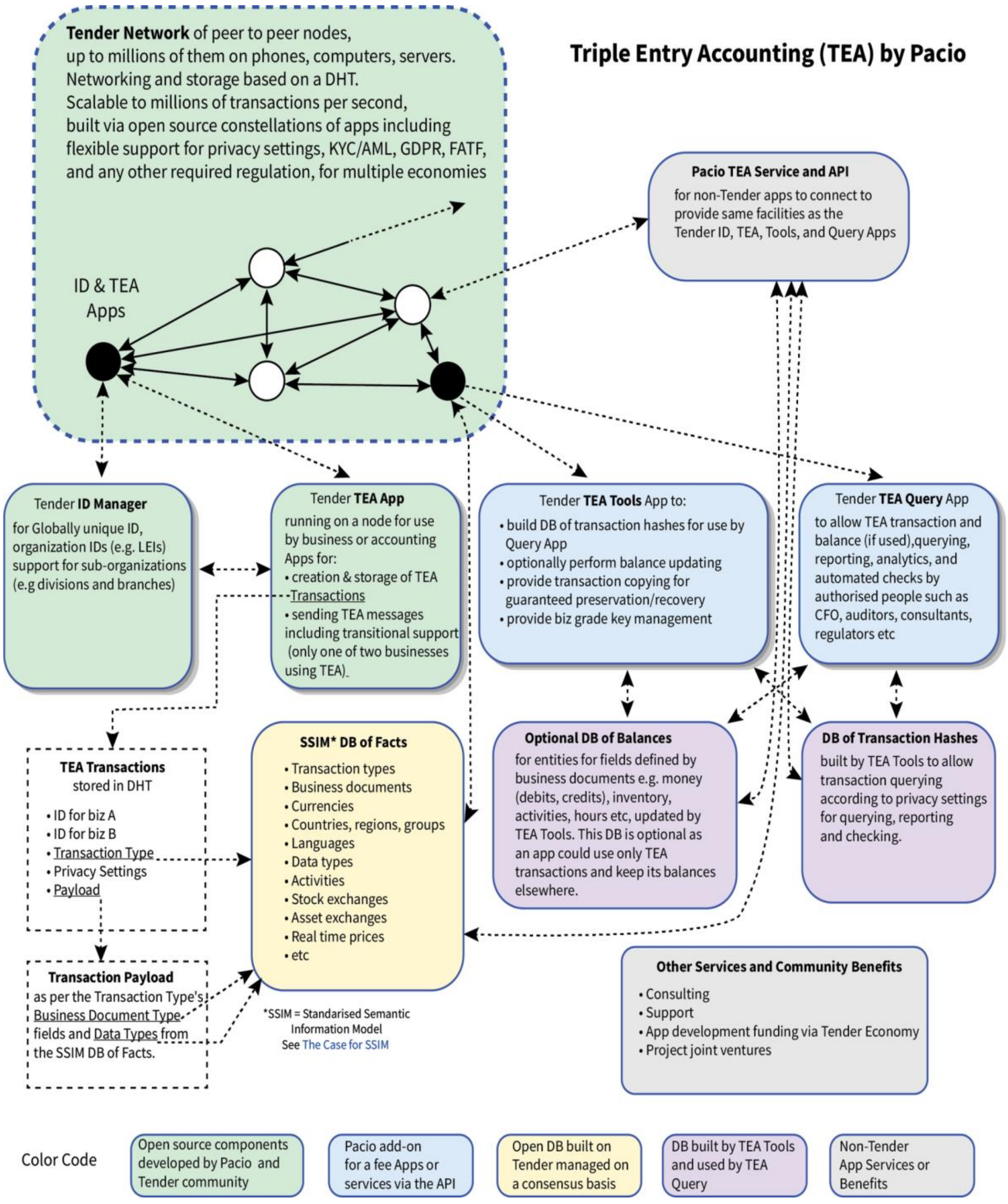
- distributed data storage:
  - in the form of transactions
- peer-to-peer overlay networking:
  - peers keep track of each other, send each other messages
  - this network is built as a layer on top of the IP protocol
- Efficient scaling:
  - the network becomes faster, faster than we add nodes

Role in Pacio's vision and roadmap

The complete Pacio vision is available in the whitepaper at:

https://pacio.io/docs/PacioWhitePaper.pdf

In the complete vision of Pacio's work (seen in the diagram below), the Tender DHT realizes the vision for the Tender network (seen at the top-left of he diagram):



Triple Entry Accounting (TEA) by Pacio

Tender Network of peer to peer nodes,
up to millions of them on phones, computers, servers.
Networking and storage based on a DHT.
Scalable to millions of transactions per second,
built via open source constellations of apps including
flexible support for privacy settings, KYC/AML, GDPR, FATF,
and any other required regulation, for multiple economies

Pacio TEA Service and API
for non-Tender apps to connect to provide same facilities as the Tender ID, TEA, Tools, and Query Apps

Tender **ID Manager**
for Globally unique ID, organization IDs (e.g. LEIs) support for sub-organizations (e.g divisions and branches)

Tender **TEA App**
running on a node for use by business or accounting Apps for:
- creation & storage of TEA Transactions
- sending TEA messages including transitional support (only one of two businesses using TEA).

Tender **TEA Tools** App to:
- build DB of transaction hashes for use by Query App
- optionally perform balance updating
- provide transaction copying for guaranteed preservation/recovery
- provide biz grade key management

Tender **TEA Query** App
to allow TEA transaction and balance (if used),querying, reporting, analytics, and automated checks by authorised people such as CFO, auditors, consultants, regulators etc

TEA Transactions
stored in DHT
- ID for biz A
- ID for biz B
- Transaction Type
- Privacy Settings
- Payload

SSIM* DB of Facts
- Transaction types
- Business documents
- Currencies
- Countries, regions, groups
- Languages
- Data types
- Activities
- Stock exchanges
- Asset exchanges
- Real time prices
- etc

Optional DB of Balances
for entities for fields defined by business documents e.g. money (debits, credits), inventory, activities, hours etc, updated by TEA Tools. This DB is optional as an app could use only TEA transactions and keep its balances elsewhere.

DB of Transaction Hashes
built by TEA Tools to allow transaction querying according to privacy settings for querying, reporting and checking.

Transaction Payload
as per the Transaction Type's Business Document Type fields and Data Types from the SSIM DB of Facts.

*SSIM = Standarised Semantic Information Model
See The Case for SSIM

Other Services and Community Benefits
- Consulting
- Support
- App development funding via Tender Economy
- Project joint ventures

Color Code				
🟩 Open source components developed by Pacio and Tender community	🟦 Pacio add-on for a fee Apps or services via the API	🟨 Open DB built on Tender managed on a consensus basis	🟪 DB built by TEA Tools and used by TEA Query	⬜ Non-Tender App Services or Benefits

2. Current platforms and scaling problems

With broadcast-type blockchains

The speed of the complete system is limited by several factors:

- every peer stores the whole dataset
- every peer processes every new transaction

Existing DHT-based scaling systems

Bittorrent

Bittorrent was the most notable project to make use of a DHT to scale distributed storage. With Bittorrent, a lot of peers are interested in the same files.

This makes a content-addressed storage like a DHT very well suited to the problem, as a lot of peers are then able to send a piece of content to a peer who requires it.

Our problem is different: we can use a DHT, but peers are not intrinsically interested in storing another user's content. Instead, it costs them (the price of the storage and network needed) with no tangible benefit, unless we design incentives to do so.

Holochain

The Holochain is notably focused on scaling applications independently. For that, it uses a DHT-based overlay network and content-addressed distributed storage. This eliminates the need for every peer to store every piece of data or transaction, but introduces different challenges: in cryptocurrency applications, for instance, peers validate transactions against the current state, which means they need to know past transactions.

Holochain doesn't bundle consensus rules for such a shared system state, since it doesn't deal with one global shared state. This makes it easier to scale applications which don't need such a feature. But as a side-effect, there is no clear path to create cryptocurrencies on top of holochain.

While removing the consensus for the equations makes it easier for hApps to scale relative to other complexities and variables, it also removes one of the features required for the system we intend to build.

In essence, it scales the storage part, but leaves other parts for applications to figure out. Consensus is one of those parts that are necessary in cryptocurrency applications, but omitted from the holochain framework.

This makes holochain quite opinionated about the types of applications you should build with it, and notably, it excludes cryptocurrencies.

Different types of scaling

Different parts of the system to scale

Network

Consensus

Storage and retrieval

Bootstrap

Different variables to scale against

p number of peers

b number of past transactions (or number of blocks for a blockchain architecture)

Different complexities to scale against

Time complexity C_t : how much time does the process take to complete ?

Communication complexity C_c : How many messages are necessary for the system to “know” the new state / transaction ?

Note: since messages can be sent and processed in parallel, communication complexity is not always greater than time complexity

Memory/Storage complexity C_m : how much memory does the process need to complete ? how much storage does the process need long term ?

Achieving scaling in this context means the system can maintain each complexity under the maximum O(variable), for each variable and each part of the system. We’ll express the complexities as O(f(b,p)) where f(b,p) must scale slower than both b and p.

The metric for performance measurements is the throughput of the network in transactions per second. Achieving scaling means the throughput rises as or more rapidly than any variable (for high enough values of the variable).

Summary of state of the art

insert explanation of how these complexities are obtained before summing up

Networking

Broadcasting (bitcoin), gossiping (ethereum) or flooding (avalanche)

All 3 come with a communication complexity of O(p)

Consensus

Proof-of-work: sacrifices decentralization and efficiency. Resources get spent on calculations of which the results are discarded, and the amount of resources needed limits the amount of participants, leading to centralization.

Proof-of-stake: as currently implemented, peers with higher stakes get more votes on the final consensus, which could potentially lead the network to make decisions that favor the same peers, leading to centralization over time A powerful counterpower to that is that centralization reduces trust, and therefore the value of the network which the voting peers have stake in. This incentivizes these peer to vote in a fair way.

Byzantine fault tolerance (BFT):

pBFT: partial trust in peers. reputation.

BFT with sharding:

Types of decentralized ledgers and their different complexities:

	Communication complexity	Consensus complexity	Storage complexity	peer introduction complexity
Bitcoin	Broadcasting: time complexity O(p)		Whole chain per peer: time complexity O(b)	Chain download: time complexity O(b)
Sharded blockchains	O(p) for cross-shard transactions		best case: O(p/ log b)	best-case: O(p/ log b)
Holochain	O(log p)	No consensus provided. Adding a consensus often degrades the other complexities. But not always !	Store and retrieve by closest peer lookup: O(log p)	Finding neighbours: O(log p)

3. Getting past current platforms' limitations

Solutions for a scalable network

The Tender DHT takes the shape of a blockchain data structure, layered onto a distributed hash table (DHT) for storage and networking.

In order to provide a more scalable solution to the distributed transactions problem (than current solutions), the Tender platform uses:

- Decentralized network storage
  This is to not require every peer storing every transaction and block. Instead, each peer stores its share of the total of transactions.
- Dispatch proofs
  to distribute the validation step equally between peers. This is the key to making the transaction validation scalable.
- Backwards **AND** forward links between blocks
  Backward links are common in blockchains to ensure authenticity of the blocks, forward links help new or often-offline clients skip forward in the block history without having to trust other peers for validation

Scaling properties we aim for

Here's complexities we aim for:

- Bootstraping: Ct=O(1), Cc=O(log p) thanks to introducers and jumps
- Storage: closest-to-hash replication (DHT): storage size: O(b / p)
- Consensus: Ct = O(1), Cc=O(log p)
- Network: Cc=O(log p)

And also:

- Fairness: each peer has an appropriate chance of getting involved in the validation process
- Decentralization
- Efficiency: no resource gets spent on discarded results (e.g. no proof-of-work)

And resistance to:

- Sybil attacks. Where an attacker spawns a large number of peers in order to gain more influence on the network behaviour.

4. Implementation of the Tender DHT

Keys and addresses

Keys

Identifiers on the network are ed25519 keypairs. From a 32-byte seed we get:

- a 32-bytes private key
- a 32-bytes public key

Addresses

Addresses need several representations:

- a bytes representation (for software)
- a string representation (for people, with features like a

Bytes representation

The bytes representation of an address is the public key itself (32-bytes long).

String representation

The addresses are self-describing using multi-formats. The added length due to the multi-format prefix doesn't affect (i.e. make longer) the actual length of the address in bytes representation.

Checksum

The string representation also contains a short checksum, in order to let any third-party applications (e.g. wallets) make the user experience fool-proof. For instance, for a cryptocurrency application, we don't want users to send significant amount of funds to a wrong or inexistent address.

We want to avoid users having to deal with addresses directly in user interfaces (e.g. identity manager and various wallets). But the address representation will still be needed, if only for developers to be able to display addresses and debug their applications.

Storage on disk

The common-denominator strategy to store keys cross-platform is to store the user's keypair encrypted in a file. Encryption is done using the user's OS password, with AES-256 GCM.

Per-platform optimizations

Several platforms have secure enclaves designed for the exact purpose of storing sensitive data. Wherever possible, we use those secure enclaves to store keys, and fallback onto encrypted keyfiles described above.

DHT overlay networking

The P2P functionality of Tender consists of:

- protocol for serializing and broadcasting claims
- Node / client for receiving and storing claims
- DHT network to query and retrieve claims from their hash

Structure of the routing table

A peer uses its public key as an ID on the network.

Each peer stores a routing table that contains contact details for some peers it knows on the network. In order to keep track of a contact we need:

- its ID (its public key)
- its publicly accessible IP address
- a port

Its stores them in a table where we call each row a bucket. A bucket is a set of peers that contain **the same first i bits** as the current peer's address. Each bucket contains maximum K peers.

Let's try to represent that. Row i contains the bucket B_i :

i=0 B_0	10100	?	?	?	?
i=1 B_1	**0**0011	**0**0101	**0**0100	**0**0110	**0**0001
i=2 B_2	**01**001	**01**010	**01**101	**01**110	**01**111
i=3 B_3	**010**01	**010**10	**010**00	n/a	n/a
i=4 B_4	**0101**0	n/a	n/a	n/a	n/a

Let the current peer's ID be **01011**

Let **K=5** for this figure

The current peer's goal is to discover enough peers to maintain these rows full with the most stable peers it knows. Stable meaning peers that have the longest history of staying online and available for validating transactions.

K is arbitrary. In practice, we choose **K = 20**. The range of i depends on the size of addresses. Addresses are 32 bytes long (256 bits), so we make i vary from 0 to 255. We call 256-i the **XOR distance** between two IDs, and represent it as one uint32.

In the figure above, the rows filled with **n/a** indicate that there is no possible address that can fill those cells. For instance, in the last row, only one possible ID has all but 1 (XOR distance) bits in common with the current peer's ID.

Per-server implementation of the routing table

It is ultimately up to the server implementation to choose which peers fill a given bucket. Pacio implements one server that matches the DHT specification, but other third-party servers can also be developed, which would increase the network's resilience to errors in one of the server implementations.

During the release of the Tender currencies, we expect to make a change to how the Pacio server chooses the peers in its routing table, and give a higher priority to peers which have committed to validating transactions, remain online the most often, and keep their replicas of the data available. Peers commit to that by staking a portion of the currencies they hold, to be taken from if they misbehave.

Per-application routing meta-data

While the routing table is common to all applications, each application can store metadata related to peers if it needs.

Data serialization over the network

Data (e.g. claims) that gets broadcasted over the network gets serialized to a byte schema. The serialization of the payload is done via protocol buffers.

To allow for evolution of the schema, its version is a fingerprint of the .proto schema file (e.g. short hash).

NAT traversal

While highly available peers with public IPs are preferred by other peers (as in more likely to be kept around in the routing table), NAT traversal functionality is useful for all the other peers to maintain connections to the network and get a decent level of service.

The roadmap is to provide progressive support of it, in chronological order:

- Public IPs only
- Using a public STUN server (provided by Twilio)
- Peer-to-peer (using the rust implementation of libp2p)

Initial discovery / bootstrap

Through the Tender tracker (bootstrap / discovery service in the cloud). Known URL bootstrap node, certificate is shipped with the application.
In a later iteration of the application, this known node would be entirely optional if another one is provided by the user as configuration.

Risks with the DHT implementation

Risks are mostly associated with the DHT, routing and scaling. A DHT implementation can work correctly, and yet be plagued by some problems once it runs on a network made from real peer nodes, with real-world conditions:

1. Peer availability types: as long as IPv4 is predominant, some nodes will be behind NAT, and some others will be publicly reachable. The DHT implementation should take that in account, and not try to treat those two types of nodes the same. Else the query time will be wasted trying to reach nodes that cannot be reached.
2. Routing table maintenance: Nodes behind NAT should be garbage-collected from the routing tables, since you'd need them to coordinate in order to dial them. That makes them less attractive next-hop-nodes for a query than publicly accessible nodes.
3. Query interruption: Addressing risk 1 and 2 (Peer availability and routing table maintenance) will help terminating queries sooner since there's less of a redundancy need in case next-hop nodes do not answer (because of living behind a NAT).

The simplicity of the DHT concept still allows for a reasonably performant first implementation despite the risks.

Transactions and blocks

Transaction structure is:

- owner: (public Tender ID)
- prev: stateRoot (hash of the state onto which that transaction applies) [3]
- changes: freeform document, standardized per application [1]
- dispatch proof
- hash
- signatures (owner's signature on hash, plus validators') [2] : array of:
    - signer ID (public Tender ID)
    - signature

[1] e.g. for a cryptocurrency, must contain the amount and recipient_address

[2] potentially 1 unique signature aggregate (needs further specification)

[3] hash pointer to the latest known DHT state, serves as a timestamp without relying on machine clocks

The hash of the transaction is:

$$\text{hash} = H(\text{prev} \,||\, \text{owner} \,||\, \text{changes} \,||\, \text{dispatch\_proof})$$

Similarly, a block's structure is:

- owner: address
- prev: stateRoot (hash of the state onto which that block applies) [3]
- sum_changes: freeform document, standardized per application [1]
- dispatch proof
- hash
- signatures (owner's signature on hash, plus validators') [2] : array of:
    - signer ID (public Tender ID)
    - signature

The hash of the block is:

$$\text{hash} = H(\text{prev} \,||\, \text{owner} \,||\, \text{sum\_changes} \,||\, \text{dispatch\_proof})$$

Deterministic validator selection

Generation of a new transaction

Then the tx needs to get validated by at least **T** validators out of **A**.

The validators are chosen deterministically with:

for i in [1 .. A],

v_i = H(i || tx.prev || tx.owner || tx.changes)

A minimum of **T** of these **A** validators needs to sign the transaction for it to become validated.

Forward and backward links

One of the additions of the Tender DHT compared to existing trustless ledger implementations resides in forward links. They enable peers that come back online to directly retrieve the latest state of another peer's assets, without having to, either:

- sequentially verify all transaction since it went offline
- stay online to follow all transactions as they happen
- trust another peer for the validations

Instead, peers can:

- sequentially follow and validate all blocks and transactions that get added to the chain
- fast-forward to the latest shared state using forward links

Implementation of forward links

Backward links are embedded in the blocks (and transactions) at commit time. They are the cryptographic hash that proves the blockchain is authentic (the prev link of the next block is the hash of the previous block).

Forward links, by contrast, are added as meta-data to a block after the fact. Once the successor to a given block is committed, the validators responsible for the previous block creates and collectively signs a forward link.

The forward link cannot be taken in account in the hash of the block. Instead, its security resides in a group signature by the validators of the previous block.

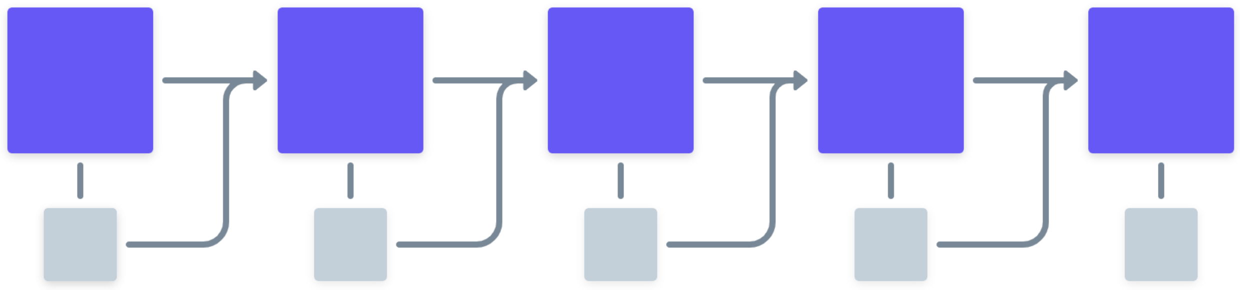


Figure: Forward links

These forward links let a newly online peer forward efficiently through all the blocks produced since it was last seen online. Indeed, it just has to verify the validators' signatures of the forward link.

The signature is a threshold signature, which means at least **T** out of **A** validators are required to sign it. This leaves a bit of margin for any previous validators to go offline in the meantime.

Modeling under high churn

Under high churn (i.e. when too many validators go offline at the same time), we might hit the lower limit of potential signers being online. In that case, peers performs a new authenticated search through the DHT, in order to know which peers are now the new closest ones, as well as which peers are now along the search path.

The new authenticated search is performed for each block that needs to be included in the forward link, and for which not enough of the previous validators are offline.

Fast-forward with longer-distance forward links, or jumps

The structure we have so far is akin to a doubly linked list: we have authenticated backward links through the *prev* field of each block, and forward links signed and added by validators after the next block is chosen.

We can speed up the peer catch-up by adding similar links we call **jumps**, which, instead of pointing to the next block, point several blocks further. Each successive jump points 4 times further as the previous one, leading to distances increasing logarithmically: 1,4,16,64, ...

The distance of jump i is:

for i from 0,

$$\mathbf{D = 2^{2i}}$$

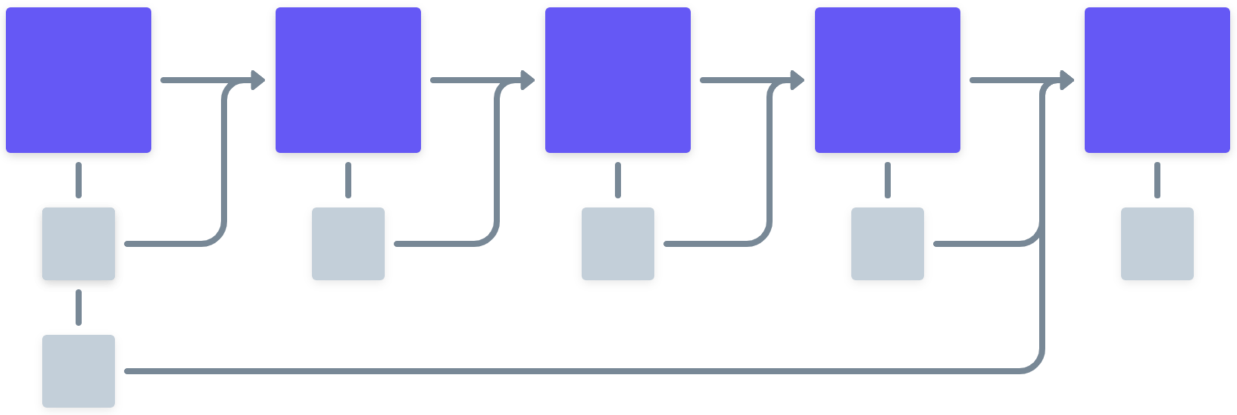


Figure: longer-distance forward links, or jumps

For each new jump, we increase both the signature threshold and the pool of validators to the union of all validators involved in the individual blocks' signatures. This means jump *i* must be signed by at least:

$$\mathbf{t_i} = T * 2^{2i}$$

out of

$$\mathbf{a_i} = A * 2^{2i} \text{ potential signers}$$

Dispatch proofs

Each peer on the search path signs a dispatch proof and passes the payload to the next peer on its routing table.

The validators can then validate all signatures, and validate another fact: due to the structure of the DHT's routing table, each next contact must have at least one more common bit with the owner peer (than the current contact).

Security model

In section 4, we model the security features if dispatch proofs:

- modeling under network churn
- resistance to sybil attacks

Deterministic branch selection

Transaction propagation

When a peer creates a transaction, it is propagated to the validators along the search path (described in the networking section). The validator checks the transaction, signs it if it approves it, and returns the signature to all the peers along the search path, including the owner of the transaction.

Transactions are accumulated into blocks by validators and peers along the search path. As soon as one peer has enough transactions (more than a network constant **MIN_BLOCK_TXS**), it proposes a block to the network. The block is then validated by the deterministically-chosen validators,

Block structure and creation

The block structure is a merkle trie. By inserting into this structure, the block's transaction bytes are naturally sorted from lowest to highest hash. This avoids a pain point: if the block was a list of transactions, peers would be incentivized to spend time and energy re-ordering the transactions inside the block to obtain the lowest hash possible in order to maximize their chances to collect block rewards. By making the block structure deterministic, bl

It incentivizes the block creators to re-order transactions inside of blocks to get the lowest hash ??

=> transactions inside a block are sorted by lowest to highest hash value. The block's data structure is a merkle trie, so this property is obtained naturally when collecting blocks. The structure of the merkle trie is checked

Block creation conflict resolution

It can happen that two block creators race to propose different blocks. In that case, the network simply chooses the block with the lowest hash as the one to integrate.

A block is considered final once:

- the validators of the previous block have generated a forward link
- the validators of the $D = 2^{2i}$ previous blocks have generated forward jumps to it
- one next block has been added
- the validators of this block have generated a forward link to the next block

Once a peer sees these elements when querying a block's ID, it can be sure the block is final.

Once a block is final, transaction owners which have had their assets changed in the block must rebase any new transactions onto this block (i.e. their *prev* field should be set to the hash of the newest block).

This ensures that one ID can do at most one transaction per block. Which makes double spending impossible as a side effect.

Misbehavior detection and correction

Every validator has to verify every transaction and signature in the dispatch proof. However, there exists a possibility of malicious actors

Any peer who detects a misbehaviour from another peer can report it via a misbehavior message that gets gossiped to the whole network.

Ban in non-monetary application

In non-monetary transactions (e.g. Tender organizations, which are the first use case of the Tender DHT), the disincentive to cheat is banning.

Stake and penalty in monetary applications

If we consider monetary value applications like cryptocurrencies, a more effective deterrent against misbehaviour is having a stake to lose. In these applications, it is necessary to sign off a stake in order to be considered for transaction validation. In case of provable misbehavior, a penalty is deduced from the stake and transferred to the peer which detected the misbehavior.

This also incentivizes peers to report misbehaving peers and check not only the signatures

Cost of running the network

At launch there will be no incentives to run the network besides the ability to use your Tender identity.

That also means there is no financial disincentive to propagate an invalid transaction.

Data availability

Incentives to stay online.

Tender currencies as a “special” app ? Then money gets printed by the network to reward available nodes and validators. Problem : hidden “inflation fee” instead of tx fees. But inflation is good for the economy (pushes people to spend the money rather than keep it) where tx fees favors the opposite.

In non-monetary applications

No incentive between the launch of the Tender Identity manager and the launch of the Tender currencies. Pacio will runs nodes and cover the cost anyway.

In monetary applications

Stake to validate. Stake only if server with publicly accessible IP. Staking signature gets dispatched to peers on the network in order to get included in peers’ routing tables. The signature is stored in the routing table alongside the contact info (IP, port, peer ID).

5. Security modeling

Dispatch proofs

Modeling under churn

Peer churn doesn't impact the past signatures. If a dispatch proof is signed by the peers along the search path, it means that peers have been online with the appropriate keys to sign this path in the past.

Resistance to sybil attacks ?

What about an attacker that spawns enough peers to have keys along sign along all paths ? Is that possible ? needs to be tried in simulation

6. Scalable DHT transactions as an application platform

One network, different applications

Common parts:

- one user identity on the network
- one routing table and networking component

Per-application diverging parts:

- transaction content (one common general structure, but some parts of it are specified per application). The generic transaction structure is specialized into application-specific transaction structures by specifying the structure of the *changes* field.
- transaction validation rules

Applications

Two examples created by Tender:

- Identity manager
- Scalable currency transfers

Pacio's vision is to build these apps first, and then expose the building blocks used for those so that third-party app developers can start building any applications that interact with the Tender network.

7. Further optimizations

Porting current blockchains’ improvements onto Tender

The advantage with this architecture is that a lot of the improvements that apply to current systems are also applicable to a blockchain built on top of a DHT.

These include performance improvements:

- Sharding
- Zk rollups for compacting blocks

And also feature improvements:

- Merkle commitments privacy (forward-provable commitments without revealing the object of the commitment)
- Zk-rollups for privacy (i.e. coin-mixing)

Depending on applications, some security and performance improvements that have been applied to other decentralized platforms can also work for Tender applications. For instance:

- staking (for cryptocurrency applications only, since you need an asset of monetary value to stake)

While these possible optimizations don’t necessarily decrease any of the complexities of our DHT design, they can still have an impact on performance up to orders of magnitude, by:

- reducing the constant factor
- offloading a lot of the work to sub-networks (similarly to how side-chain networks function). Each “transaction” on the DHT network can be an aggregate of many individual transactions