

# DT Infrastructure

DT Socialize

17th August 2020

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                  | <b>4</b>  |
| 1.1      | High-Level System Interaction . . . . .              | 4         |
| <b>2</b> | <b>Quared</b>  | <b>5</b>  |
| 2.1      | Private blockchain . . . . .                         | 6         |
| 2.2      | Overview of Quared . . . . .                         | 6         |
| 2.3      | Consensus . . . . .                                  | 8         |
| 2.3.1    | Raft . . . . .                                       | 8         |
| 2.3.2    | Istanbul BFT (Byzantine Fault Tolerance) . . . . .   | 9         |
| 2.3.3    | Comparison of Raft against Istanbul BFT . . . . .    | 12        |
| 2.4      | Data Privacy . . . . .                               | 12        |
| 2.4.1    | Private Transactions and Private Contracts . . . . . | 13        |
| 2.4.2    | Block Validation and State Consensus . . . . .       | 14        |
| 2.5      | A Note on Sharding . . . . .                         | 15        |
| 2.6      | Performance . . . . .                                | 15        |
| 2.6.1    | Latency and Throughput . . . . .                     | 16        |
| 2.6.2    | Public versus Private Contracts . . . . .            | 16        |
| 2.7      | Technical Note . . . . .                             | 17        |
| <b>3</b> | <b>Smart Contracts</b>                               | <b>18</b> |
| 3.1      | OpenZeppelin Smart Contracts . . . . .               | 18        |
| 3.2      | DT Smart Contracts . . . . .                         | 20        |
| 3.3      | Technical Note . . . . .                             | 22        |
| <b>4</b> | <b>Block Indexer</b>                                 | <b>23</b> |
| 4.1      | DT Indexer . . . . .                                 | 23        |
| 4.1.1    | Data Models . . . . .                                | 23        |
| 4.2      | DT Indexer API . . . . .                             | 25        |
| 4.3      | ethstats server . . . . .                            | 26        |
| 4.4      | Technical Note . . . . .                             | 26        |

|           |   |           |
|-----------|---|-----------|
| <b>5</b>  | <b>Public and Private Block Explorers</b> | <b>26</b> |
| 5.1       | Public Block Explorer . . . . .           | 27        |
| 5.2       | Private Block Explorer . . . . .          | 28        |
| 5.3       | Technical Note . . . . .                  | 29        |
| <b>6</b>  | <b>Proxy and Signer Proxy</b>             | <b>30</b> |
| 6.1       | Proxy . . . . .                           | 30        |
| 6.1.1     | Authorisation . . . . .                   | 30        |
| 6.1.2     | Call Rate Limit . . . . .                 | 31        |
| 6.2       | Signer Proxy . . . . .                    | 31        |
| 6.3       | Technical Note . . . . .                  | 31        |
| <b>7</b>  | <b>Web Wallet</b>                         | <b>31</b> |
| 7.1       | Seed Phrases and Mnemonics . . . . .      | 32        |
| 7.2       | Wallet types . . . . .                    | 33        |
| 7.3       | Operations . . . . .                      | 34        |
| 7.3.1     | Wallet creation . . . . .                 | 34        |
| 7.3.2     | Funds . . . . .                           | 35        |
| 7.4       | Technical note . . . . .                  | 35        |
| <b>8</b>  | <b>Admin Panel</b>                        | <b>35</b> |
| 8.1       | Technical note . . . . .                  | 36        |
| <b>9</b>  | <b>Address Verification Service</b>       | <b>37</b> |
| <b>10</b> | <b>Legacy Token Migration Service</b>     | <b>37</b> |

## List of Figures

|    |  |    |
|----|--|----|
| 1  | DT Infrastructure Deployment Diagram. . . . .  | 4  |
| 2  | Quared architecture built with reference to go-ethereum. . . . .   | 7  |
| 3  | State transitions of a Quared node. . . . .  | 11 |
| 4  | Simple Privacy Design of a Quorum network. . . . .   | 13 |
| 5  | RAFT versus IBFT consensus - Throughput and Latency measurements. . . . .  | 17 |
| 6  | Latency and throughput measurements for private contract deployments. . . . .  | 17 |
| 7  | DT Smart Contracts inheritance from OpenZeppelin. The inheritance among OpenZeppelin's smart contracts is omitted for clarity. . . . . | 19 |
| 8  | DT Smart Contracts full class diagram with OpenZeppelin as an external dependency. . . . .   | 21 |
| 9  | Sample screen of the Ethereum Networks Stats web service. . . . .  | 27 |
| 10 | Screenshot of the Public Block Explorer. . . . .   | 28 |
| 11 | Screenshot of the Private Block Explorer. . . . .  | 29 |
| 12 | Hierarchical structure of wallets/keys in BIP 32 . . . . .   | 33 |
| 13 | Web wallet application as seen on a browser. . . . .   | 34 |
| 14 | Admin Panel web application. . . . .   | 36 |

## List of Tables

|   |   |    |
|---|---|----|
| 1 | Comparison between Raft and Istanbul BFT. . . . . | 12 |
|---|---|----|

## Listings

|   |   |    |
|---|---|----|
| 1 | Mongoose schema for transactions. . . . .       | 23 |
| 2 | Mongoose schema for token transactions. . . . . | 24 |
| 3 | Mongoose schema for blocks. . . . .             | 24 |
| 4 | Mongoose schema for uncle blocks. . . . .       | 24 |

# 1 Introduction

In this document, we present a detailed summary of the underlying infrastructure of the DT Network. We will explain each of the components that support the platform, focusing on the technical details of Quared, a private and permissioned blockchain system. This kind of blockchain enables, among others, trust and privacy mechanisms that have not been achieved yet successfully on public smart contract platforms.

## 1.1 High-Level System Interaction

The project's implementation can be more easily understood by splitting the elements of the blockchain network from those of the supporting web applications.

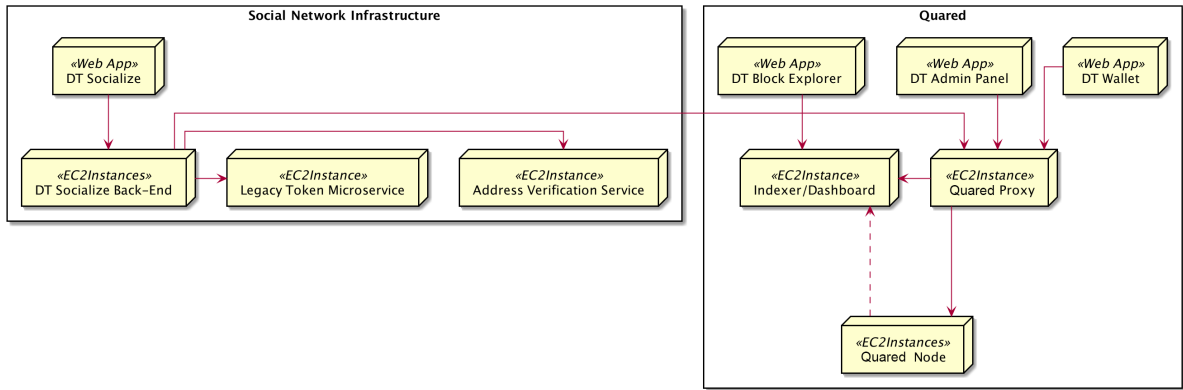


Figure 1: DT Infrastructure Deployment Diagram.

### Social Network Infrastructure

The **DT Circle** platform is meant to be served to the users as a web and mobile application which aims to compensate its users for their activity and data. This includes the opportunity for users to build and enrich their digital profiles, and receive advantages in the ecosystem in return.

Naturally, all the social interactions of the users need to be handled by its corresponding back-end, the **DT Circle Back-End**, in order to save the user-generated data and to handle the content serving.

These applications are a central element of the DT Platform. They are, however, outside the scope of this document which aims to only address components of the blockchain and its interfaces, excluding DT Circle.

For users migrating to the new platform, the **Legacy Token Microservice** provides a way of claiming their legacy funds as new tokens for the new platform. Its task is elaborated in Section 10.

Additionally, as a future-proof extension of the platform, there is a **Address Verification Service** for coming platforms to double-check the existence of a user's account, and tokenise their identity. See Section 9 for further information.

## Quared Infrastructure

The blockchain system designed as the foundation of this part of the infrastructure is Quared, a Quorum-based system for which **Quared nodes** need to be run. Quared is a forked implementation of Ethereum that shares most of the characteristics that have made Ethereum successful, eg smart contract deployments, but addresses shortcomings that are required in enterprise applications, eg privacy. This choice is explained in thorough detail in Section 2 Quared.

Given that Quared supports the deployment of smart contracts in the same fashion as Ethereum, the **DT Smart Contracts** (see Section 3) are a series of Solidity smart contracts based on OpenZeppelin's implementation of various Ethereum standards.

In order to have an always-ready overview of the Quared network, a **Indexer/Dashboard** service is introduced in Section 4. This service monitors the overall status of the Quared network while also listening to events emitted by the smart contracts and indexes them for a prompt retrieval.

So that the information collected and organised by the Indexer can be inspected, two kinds of **DT Block Explorers** are put in place: one for general users and one for the network's administrators. Both explorers and their functionality are explained in Section 5 Public and Private Block Explorers.

The **Quared Proxy** is a pair of services that guard the communication from the outer world into the Quared nodes. They provide a protection against network spamming (of general users) and sign privileged requests (of network administrators). Their advantages are discussed in Section 6 Proxy and Signer Proxy.

The main intended way for the users to transact with their tokens is via the **DT Wallet**, which supports all the use cases that involve the user and their balance. This web application is explored in 7.

The last piece of the blockchain infrastructure is the **Admin Panel** (see Section 8). This web interface supports network administrator tasks such as minting new tokens, freezing individual accounts, and pausing the whole blockchain infrastructure.

## 2 Quared

Quared is the blockchain system designed to be the backbone where we deploy the DT Smart Contracts. Quared is a Quorum-based system whose extension provides the convenience of accessing data in a cryptographically safe manner. It plays a central role in the decentralised execution of the code related to the distribution of the DT Tokens. Quared's key difference in comparison to Ethereum, from which its codebase is derived, resides in it being a private and permissioned blockchain which enables some desirable features, which we elaborate in this section.

## 2.1 Private blockchain

A private blockchain is a replicated, shared ledger that serves as the consensus and reconciliation mechanism on which Smart Contracts are deployed. Smart contracts serve as common business logic that can be accessed by enterprise systems. The main difference with public blockchains, such as Ethereum, is the ability to provide data privacy; transactions and the state data of Smart Contracts are exposed in the clear among and within the replicated shared ledger. Numerous approaches have been tried to tackle this lack of confidentiality with mechanisms such as homomorphic encryption, zero-knowledge proofs, secure multi-party computation, ledger segmentation, or cryptographic protocols, but no clear evidence of their resilience has been established just yet.

Quared, on the other hand, has a simpler approach to privacy while still built on top of the Ethereum platform. Its design preserves many of the key attributes of Ethereum such as ensuring every node on the network participates in and increases the overall security of the entire network while at the same time only revealing the details of private transactions to those parties involved in the transactions. Furthermore, unlike many other private blockchains, Quared supports the development and deployment of Smart Contracts with most of the robust tools developed for Ethereum.

## 2.2 Overview of Quared

Quared is a private and permissioned blockchain derived from the official Go implementation of the Ethereum protocol<sup>1</sup>. It is based on the open-source project "Quorum," originally developed by JP Morgan<sup>2</sup> and powered now with other strong partnerships such as that with Microsoft.

Quorum implementations can use either its "raft-based" consensus algorithm (a consensus model for faster blocktimes, transaction finality and on-demand block creation) or the Istanbul BFT (Byzantine fault tolerance) consensus mechanism described in the Ethereum Improvement Proposal EIP-650<sup>3</sup>. Furthermore, it achieves Data Privacy through the introduction of a new "private" transaction type. Among the design goals of Quorum is to reuse as much existing technology as possible, minimising the changes from go-ethereum in order to reduce the effort required to keep in sync with future versions of the public Ethereum code base. Most of the logic responsible for the additional privacy functionality resides in a layer that sits on top of the standard Ethereum protocol layer (see Figure 2).

The basic idea behind Quared, as carried on from Quorum, is to *use cryptography to prevent all except those party to the transaction from seeing sensitive data*. The solution involves a single, shared blockchain and a combination of smart contract software architecture as well as some modifications to the Ethereum protocol. The smart contract architecture provides segmentation of private data. These changes to the go-ethereum codebase include modifications to the block proposal and validation processes: the block validation process is modified such that all nodes validate public transactions, and any private transactions they are party to by executing the contract code associated with the

---

<sup>1</sup><https://github.com/ethereum/go-ethereum>

<sup>2</sup><https://www.jpmorgan.com/country/UK/EN/Quorum>

<sup>3</sup><https://github.com/ethereum/EIPs/issues/650>

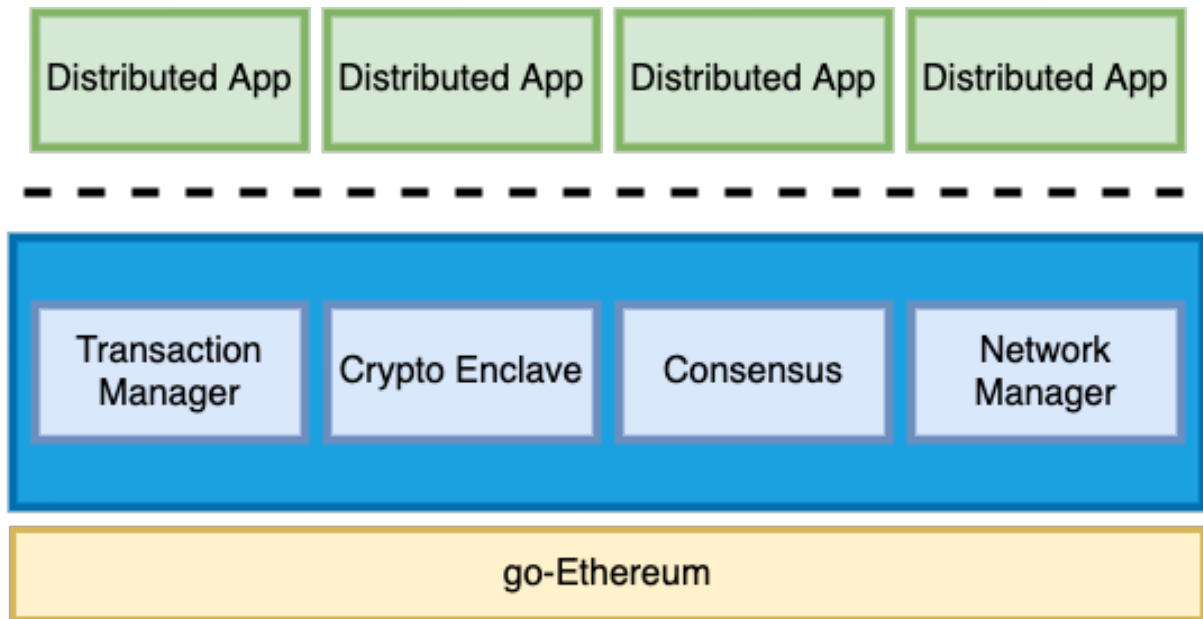


Figure 2: Quared architecture built with reference to go-ethereum.

transactions. For other “private transactions”, a node will simply skip the contract code execution process altogether.

A node will store but not be able to decrypt transactions to which it is not party. This creates a segmentation of the state database, ie the state database is split into a private state database and a public state database. All nodes in the network are in perfect state consensus on their public state whereas the private state databases will differ. Although the client nodes may no longer have access to the entire state database, they do possess a copy of all the transactions in the network to ensure consistency and immutability; they just cannot access the state of transactions they are not party to since they are encrypted with keys they do not possess. This is an important distinction relative to other segmentation strategies based on multiple block chains, and adds to the security and resiliency of the design.

## Highlights of Quared

- **Built on Ethereum**

- Over 50,000+ unit tests, security audits, bounty programmes.
- Largest ecosystem of developers, tools, and Dapps.
- Market cap of USD 45B+, and trading volume of USD 12.5B+.

- **Simple Privacy Design**

- Supports both, public and private transactions, and Smart Contracts.

- **Single Blockchain Architecture**

- All public/private smart contracts and their derived state are from a single, common, complete blockchain of transactions validated by every node in the network.
  - The state of a private smart contract is updated only for those parties involved in it.
  - Best of both worlds: every node validating the list of transactions while only exposing details of private transactions and contracts to relevant parties.
- **High Performance**
    - Able to perform hundreds of transactions per second depending on system profiling.

## Components

- **Transaction Manager** – allows access to encrypted transaction data for private transactions, manages local data stores and communication with other transaction managers.
- **Crypto Enclave** – responsible for private key management and encryption and decryption of private transaction data.
- **Consensus**
  - **Raft-Based** – consensus model for faster block time, on-demand block creation and better transaction finality.
  - **Istanbul BFT** – Three phase consensus, better fault tolerance, self verifiable blocks.
- **Network Manager** – controls access to the network, enabling a permissioned network of nodes to be created.

## 2.3 Consensus

Quorum-based nodes can implement two consensus mechanisms – Raft or Istanbul BFT. Although DT Quorum nodes use the latter, a brief introduction to the first will be given in order to compare them.

### 2.3.1 Raft

This implementation is useful for a closed-membership/consortium setting *where Byzantine fault tolerance is not a requirement*, ie where there is leader/follower model and forking is disallowed ensuring transaction finality. There is a single leader for the entire cluster through which all log entries will flow. There is a one-to-one correspondence between Raft and Quorum nodes, ie a Raft node is also an Quorum node, and the leader (“minter”) is the only node that should mint new blocks. The minter bundles transactions into a block without presenting a PoW (proof of work).



The leader is elected after a period of voting during which all nodes assume the role “candidate”. Once a new leader is chosen, the node which won the election takes the “leader” role and all other nodes take a “follower” role. When the leader/minter creates a block, the block is set to be the new head of the chain only after the block has been verified by the majority of Raft nodes. All nodes will then extend the chain together. This differs from Ethereum where the block is written to the database in any node and immediately considered the new head of the chain.

### 2.3.2 Istanbul BFT (Byzantine Fault Tolerance)

#### Terminology

- **Validator:** Block validation participant.
- **Proposer:** A block validation participant that is chosen to propose block in a consensus round.
- **Round:** Consensus round. A round starts with the proposer creating a block proposal and ends with a block commitment or round change.
- **Proposal:** New block generation proposal which is undergoing consensus processing.
- **Sequence:** Sequence number of a proposal. A sequence number should be greater than all previous sequence numbers. Currently each proposed block height is its associated sequence number.
- **Backlog:** The storage to keep future consensus messages.
- **Round state:** Consensus messages of a specific sequence and round, including pre-prepare message, prepare message, and commit message.
- **Consensus proof:** The commitment signatures of a block that can prove the block has gone through the consensus process.
- **Snapshot:** The validator voting state from last epoch.

#### Consensus

Istanbul BFT is inspired by Castro-Liskov 99 paper. However, the original PBFT (practical Byzantine fault tolerance) needed substantial tweaking to make it work in the context of blockchains. First off, there is no specific “client” which sends out requests and waits for the results. Instead, all of the validators can be seen as clients. Furthermore, to keep the blockchain progressing, a proposer will be continuously selected in each round to create block proposals for consensus. For each consensus result, we expect to generate a verifiable new block rather than a bunch of read/write operations to the file system.

Just as the original Practical BFT, Istanbul BFT uses a 3-phase consensus. These phases are **PRE-PREPARE**, **PREPARE**, and **COMMIT**. The system can tolerate at most  $F$  faulty nodes in a  $N$  validator nodes network, where  $N = 3F + 1$ . Before each round, the validators will pick one among themselves as the proposer, by default, in a round robin fashion. The proposer will then propose a new block proposal and broadcast it along with the

PRE-PREPARE message. Upon receiving the PRE-PREPARE message from the proposer, validators enter the state of PRE-PREPARED and then broadcast a PREPARE message. This step is to make sure all validators are working on the same sequence and the same round. While receiving  $2F + 1$  of PREPARE messages, the validator enters the state of PREPARED and then broadcasts its COMMIT message. This step is to inform its peers that it accepts the proposed block and is going to insert the block to the chain. Lastly, validators wait for  $2F + 1$  of COMMIT messages to enter COMMITTED state and consequently insert the block to the chain, updating their respective states.

Blocks in the Istanbul BFT protocol are final, which means that there are no forks and all valid blocks must be somewhere in the main chain. To prevent a faulty node from generating a totally different chain from the main chain, each validator appends  $2F + 1$  received COMMIT signatures to `extraData` field in the header before inserting it into the chain. Thus blocks are self-verifiable and light client can be supported as well. However, the dynamic `extraData` would cause an issue on block hash calculation. Since the same block from different validators can have different set of COMMIT signatures, the same block can have different block hashes as well. To solve this, we calculate the block hash by excluding the COMMIT signatures part. Therefore, we can still keep the block/block hash consistency as well as put the consensus proof in the block header. Consensus states Istanbul BFT is a state machine replication algorithm. Each validator maintains a state machine replica in order reach block consensus.

States:

- **NEW ROUND:** Proposer to send new block proposal. Validators wait for PRE-PREPARE message.
- **PRE-PREPARED:** A validator has received PRE-PREPARE message and broadcasts PREPARE message. Then it waits for  $2F + 1$  of PREPARE or COMMIT messages.
- **PREPARED:** A validator has received  $2F + 1$  of PREPARE messages and broadcasts COMMIT messages. Then it waits for  $2F + 1$  of COMMIT messages.
- **COMMITTED:** A validator has received  $2F + 1$  of COMMIT messages and is able to insert the proposed block into the blockchain.
- **FINAL COMMITTED:** A new block is successfully inserted into the blockchain and the validator is ready for the next round.
- **ROUND CHANGE:** A validator is waiting for  $2F + 1$  of ROUND CHANGE messages on the same proposed round number.

State Transitions

- **NEW ROUND  $\rightarrow$  PRE-PREPARED:**
  - **Proposer** collects transactions from txpool.
  - **Proposer** generates a block proposal and broadcasts it to **validators**. It then enters the PRE-PREPARED state.
  - Each **validator** enters PRE-PREPARED upon receiving the PRE-PREPARE message with the following conditions:

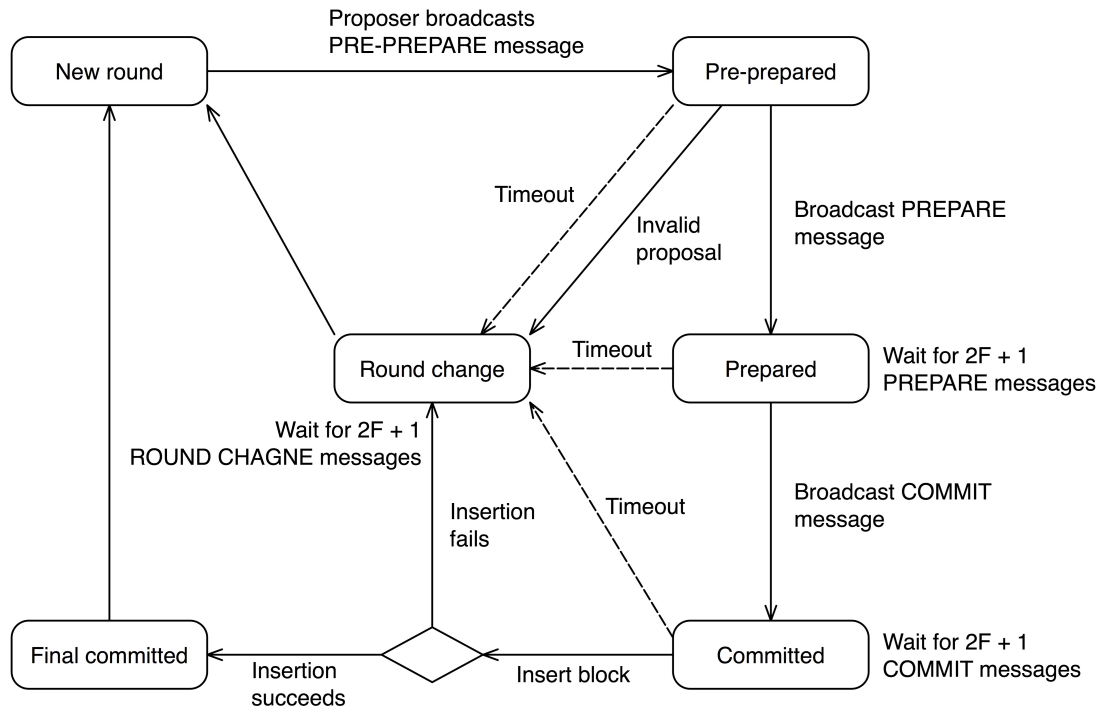


Figure 3: State transitions of a Quared node.

- \* Block proposal is from the valid proposer.
- \* Block header is valid.
- \* Block proposal's sequence and round match the **validator's** state.
- **Validator** broadcasts PREPARE message to other validators.
- PRE-PREPARED → PREPARED:
  - **Validator** receives  $2F + 1$  of valid PREPARE messages to enter PREPARED state. Valid messages conform to the following conditions:
    - \* Matched sequence and round.
    - \* Matched block hash.
    - \* Messages are from known validators.
  - **Validator** broadcasts COMMIT message upon entering PREPARED state.
- PREPARED → COMMITTED:
  - **Validator** receives  $2F + 1$  of valid COMMIT messages to enter COMMITTED state. Valid messages conform to the following conditions:
    - \* Matched sequence and round.
    - \* Matched block hash.
    - \* Messages are from known validators.

| Raft  | IBFT   |
|---|--|
| <u>Pros</u>   |  |
| <ul style="list-style-type: none"> <li>• Fast non-empty block minting (50 ms, configurable)</li> <li>• Transaction finality, ie no forks</li> </ul> | <ul style="list-style-type: none"> <li>• Hard to tamper with the past</li> <li>• Resilient against up to 30 % malicious actors</li> </ul>  |
| <u>Cons</u>   |  |
| <ul style="list-style-type: none"> <li>• No protection from bad actors, ie past can be edited</li> </ul>  | <ul style="list-style-type: none"> <li>• Mints blocks at a constant rate, ie empty blocks are possible</li> <li>• Message overhead get exponentially worse as more validators are added</li> </ul> |

Table 1: Comparison between Raft and Istanbul BFT.

- COMMITTED  $\rightarrow$  FINAL COMMITTED:
  - **Validator** appends  $2F + 1$  commitment signatures to `extraData` and tries to insert the block into the blockchain.
  - Validator enters FINAL COMMITTED state when insertion succeeds.
- FINAL COMMITTED  $\rightarrow$  NEW ROUND:
  - **Validators** pick a new **proposer** and starts a new round timer.

### 2.3.3 Comparison of Raft against Istanbul BFT

As we can see, both consensus mechanisms prevent forks unlike Proof of Work (Bitcoin/Ethereum) and Proof of Elapsed Time (Hyperledger Sawtooth). Nevertheless, they differ on the amount of healthy nodes needed, the space their produced blockchains require, and their speed. Their pros and cons are summarised at a glance in Table 1.

For a quantitative comparison of the capabilities of these algorithms see Subsection 2.6 Performance.

## 2.4 Data Privacy

Data privacy in Quared is achieved by cryptographic means and segmentation. The data of each transaction gets cryptographically signed and these signatures can be verified easily by anyone in possession of the corresponding public key – namely by DT Nodes in the network. Segmentation of the state is applied to each node’s local state database which contains the contract storage and is only accessible to the node. *Only nodes party to private transactions are able to execute the private contract code associated with the transactions*, which results in updating the private contract data storage in the local state database. The result is that each node’s local state database is only populated with public data and private data to which they are party. Figure 4 below outlines the high-level logical design of the Quorum privacy solution.

## Simple Privacy Design

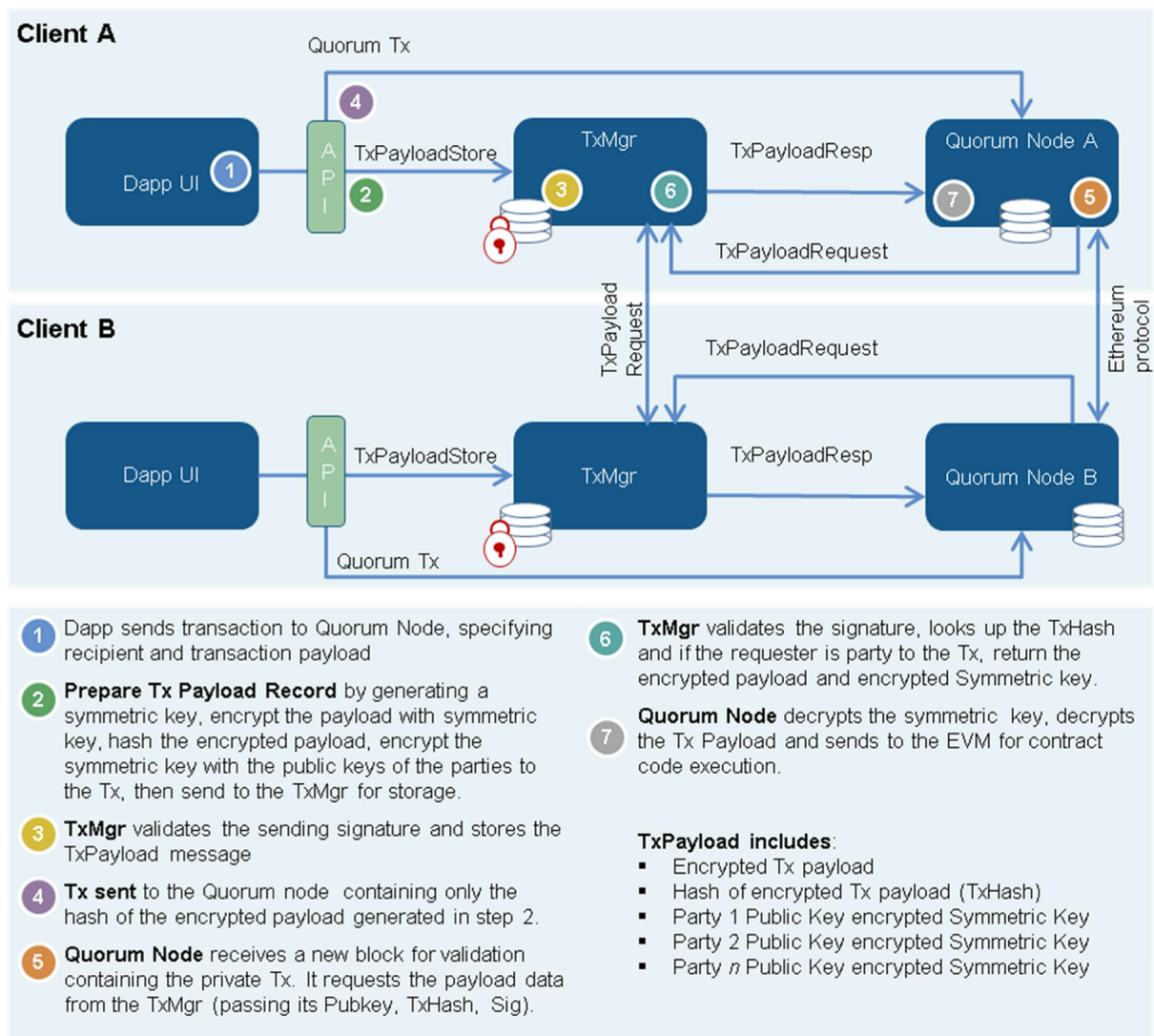


Figure 4: Simple Privacy Design of a Quorum network.

### 2.4.1 Private Transactions and Private Contracts

Private transactions are facilitated through an API exposed to the decentralised application (Dapp) that originates the transaction. A private contract is a regular contract created by a private transaction. During the block construction and validation process the node party to the private transaction will decrypt the transaction data prior to sending it to the Quorum Virtual Machine (QVM). The QVM, therefore, does not need to support encryption/decryption operations. The private state data of a private contract is stored in the clear in the local state database of the nodes party to the transaction.

#### Private Transactions

A Private transaction is a transaction that only carries a 256-bit hash in the data field. It is also identified by the parameter  $v$  of the transaction object being 37 or 38 in contrast to Ethereum's usual 27 or 28.

In addition to the basic data a typical Ethereum `sendTransaction` accepts, the new API takes a list of public keys that identify the parties to the transaction (`privateFor`). With this data a standard Ethereum transaction is generated where the payload is simply the hash of the encrypted private data. This newly formed Ethereum transaction carrying only the cryptographic hash is sent to the Quared node where it is distributed out to all the nodes in the network as a pending transaction.

A private transaction contains:

- The recipient
- Signature identifying the sender
- The amount of ether
- An optional `privateFor` (identifies the transaction as private and lists the parties privy to the transaction)
- An optional data field (a 256-bit hash in the case of a private transaction)

**Private Contracts** As mentioned before, a private contract is a contract that was created by a private transaction. The state of each of private contract is represented as its own separate Patricia-Merkle trie. One cannot create a private contract with a public transaction because the state of the contract created with a public transaction is recorded in the separate public state Patricia-Merkle trie.

#### 2.4.2 Block Validation and State Consensus

Standard Ethereum block validation includes a step to confirm the global state of all contracts match the global state hash included in the block header. This is the cryptographic proof that every node in the network has the exact same state database, ie a provable replica.

The Quared state database is split into two, a private state and a public state. These are represented in memory as two separate Patricia-Merkle trees. Quared block validation only matches on the public state.

Private state consensus is passed up to the application layer and is supported by a new `storageRoot` RPC API. When parties to a private contract require cryptographic state consensus evidence, the application can retrieve the private contract state hash for a specified block and share this value with the parties to the contract either off-chain or through an on-chain transaction. It should be noted that a specific application design may not require or, in fact, expect parties to the same contract to have different state. It should also be noted that if public state consensus is achieved, it should be impossible for parties to a private transaction to be out of consensus. This is because:

1. Validating a block includes global transaction hash consensus and public state consensus and a few other checks.
2. The QVM is deterministic – the same inputs always generate the same outputs.
3. The inputs are the same because we are confirming consensus on the list of transactions.

4. We know we are using the same blockchain of transactions because we are achieving public state consensus.
5. If we have the same blockchain of transactions and the public transactions result in public state consensus and since the QVM is deterministic then it should be impossible to have the QVM produce a different state of a private contract by processing a private transaction.

In public Ethereum, in practice, the only time blocks are invalidated is due to a fundamental configuration problem, for example if a node’s genesis block is misconfigured. Once a node is properly connected to the network and syncing blocks with its peers, it never actually fails blocks because of a global consensus break. Occasionally in the public network a malicious node will attempt to send bad blocks. Of course the threat of a malicious node is more real in the public Ethereum network so it is critically important to validate global consensus but in reality, because everyone knows that a malicious node would be detected and their blocks rejected no one bothers with that attack vector.

In permissioned Quared, the same sort of thing occurs in that if a node has a fundamental configuration problem, it will quickly see blocks not validating and syncing with the network. If it is able to get blocks to sync by only looking at public state —taking into consideration that every block has public transactions for block voting— then there is no fundamental configuration problem and nodes party to the same private transactions and contracts will deterministically arrive at the same private state for those contracts.

## 2.5 A Note on Sharding

It is worth mentioning that the Quared privacy design exhibits several properties of sharding techniques that have been proposed to enhance the performance and throughput of blockchain networks. Sharding basically segments the validation of transactions such that not every node in the network is validating every transaction. In Quared, a node is only processing transactions that are public or private transactions to which they are party, which could be described as selective sharding.

## 2.6 Performance

Tests of Quorum have demonstrated throughput of dozens to hundreds of transactions per second depending on system configuration. No specific code changes were required; the block `gaslimit` parameter enabling the node to pack in many more transactions per block was simply updated. The gating factor on throughput is the rate at which the Quorum node is able to validate blocks in which the processing of transactions must be serial.

The experimental setup in Baliga et al [2018]<sup>4</sup> provides a quantitative insight on the performance of a Quorum network.

---

<sup>4</sup><https://arxiv.org/abs/1809.03421>

### 2.6.1 Latency and Throughput

*Transaction throughput* is defined as the number of transactions per second successfully processed by the blockchain network. A transaction is successfully processed when it is included in a block and committed as part of the ledger. *Transaction latency* is the time elapsed between when a request is sent and the time when the response is received by the client. For read transactions, it is the time taken to receive the response for a read query. For write transactions, it is the time elapsed between the request and an event confirmation as received by the client after the transaction is confirmed on the blockchain.

In the experimental setup three different workloads were examined. A smart contract with pre-loaded key-value pairs was used for each workload. These workloads consisted of:

- *Write-only workload*: The write-only workload comprises of all write transactions that update a value for a randomly selected key in the key-value store of the smart contract. Write generates a transaction on the blockchain that requires the consensus algorithm to execute successfully.
- *Null workload*: The null workload comprises of transactions that call a function within the smart contract that simply returns. The null function skips the processing within the smart contract and therefore represents the baseline cost for the write call.
- *Read workload*: The read workload comprises of read transactions that read the values for randomly selected keys from the key-value store within the smart contract. Read workload is generated by all clients sending their transactions to a single peer. This design is intentional as reads are served locally by the peer by performing lookups within its local data store. Reads do not generate a transaction on the blockchain.
- *Mix workload*: Mix workload has a 50-50 mix of reads and writes.

The throughput and latency metrics of each consensus algorithm can be visualised in Figure 5. In this figure we see that the additional security of Istanbul BFT takes a toll on the transaction latency but remains within the range of a few seconds.

### 2.6.2 Public versus Private Contracts

Private contracts in Quared involve additional encryption/decryption operations and secure communication overhead between peers. This feature was built to support transaction privacy where a subset of parties transacting with each other within a large consortium can do so without others having any knowledge of the private transactions. In this case, the privately transacting parties will have to deploy a private smart contract that will be stored and run only on the peers that are counterparties to the transaction. The goal of this experiment was to measure the throughput and latency of the system when private contracts are deployed instead of public contracts. Our controlled workloads described so far use public smart contracts.

With the same experimental setup, the authors deployed a private contract among all peers and measured the latency and throughput of the network of a write-only workload



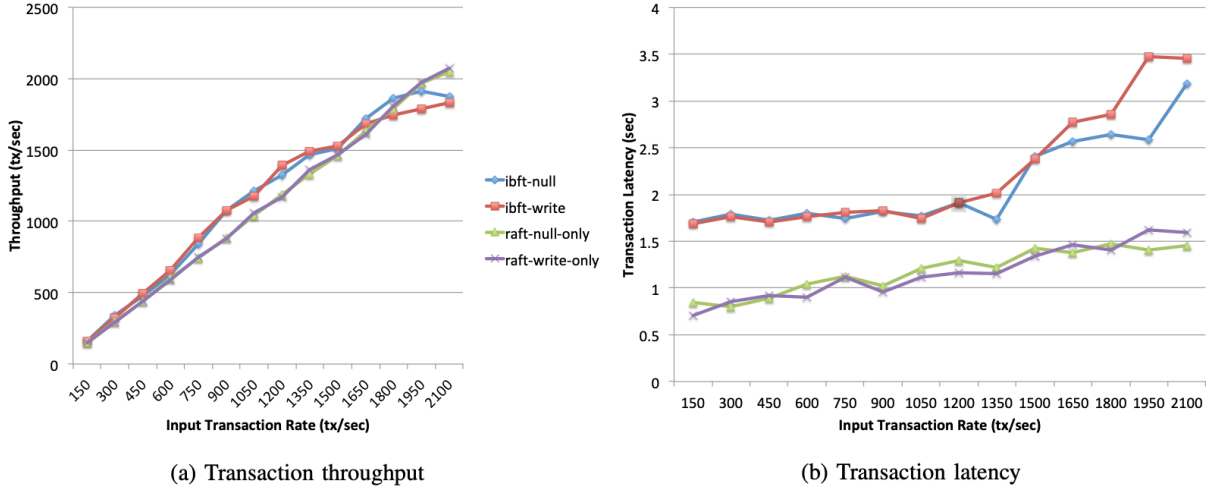


Figure 5: RAFT versus IBFT consensus - Throughput and Latency measurements.

by invoking some methods within the private contract. In Figure 6, we can observe that by varying the input transaction range from 150 transactions/second up to roughly 600 transactions/second the throughput grows linearly as expected while maintaining a flat latency. Over 600 transactions/second, the latency increases linearly for both, IBFT and Raft, reaching almost 4.5 seconds at 900 transactions/second. In either case, the difference of these metrics between the consensus algorithms is negligible.

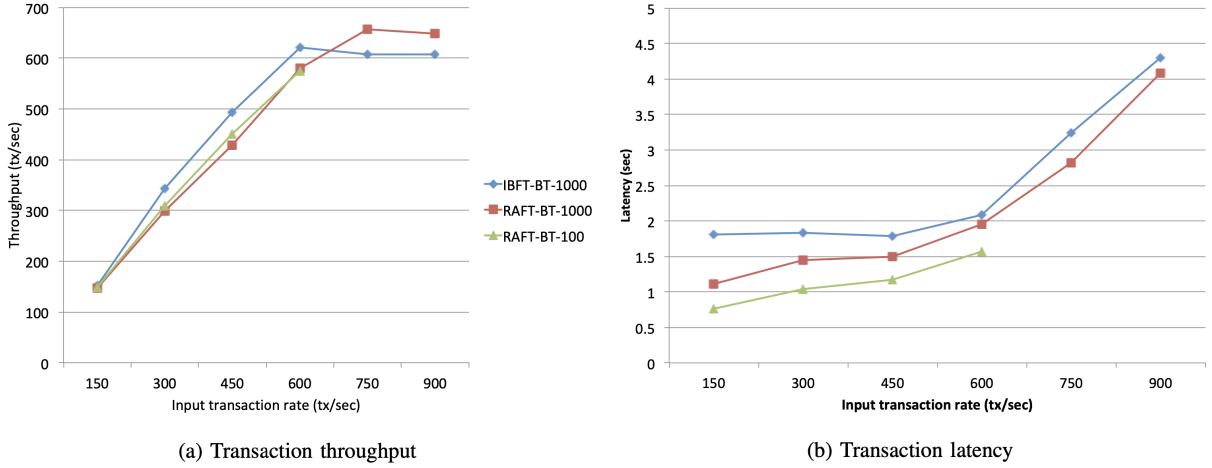


Figure 6: Latency and throughput measurements for private contract deployments.

## 2.7 Technical Note

Quared by design enables convenient and secure access to the data in its nodes participants. This section focused on its base, the Quared Node, for the rest of the functionalities, refer to the further sections of this document.

The chosen consensus algorithm for the Quared Nodes of Quared is Istanbul Byzantine Fault Tolerant (IBFT).

Although IBFT is deemed slower than Raft, the overhead of the transaction latency is not worrisome.

IBFT provides a higher level of security which supports up to 30 % malicious nodes. Furthermore its implementation on the Quared Nodes is deemed resilient, and overall suitable for Quorum’s objective in the DT Infrastructure.

## 3 Smart Contracts

Smart Contracts are pieces of code running on a blockchain rather than on a conventional machine. The most popular blockchain on which smart contracts are run is Ethereum.

Smart contracts are deployed onto the Ethereum blockchain by means of broadcasting them and having them mined by a miner. Once a miner verifies a contract and adds it to a block, it is then replicated by all other nodes that synchronise with the Ethereum blockchain. A smart contract deployed on the Ethereum blockchain can be accessed by its address. Interested parties can interact with this smart contract by sending the arguments that a function described in the smart contract requires.

Among the varieties of smart contracts we highlight that of token contracts. These contracts are public ledgers on which all transactions related to a specific contract are stored. Each token holder is uniquely identified by their address which are usually referred to as “accounts.” Note that accounts can also be associated to other features within a contract such as roles that will be described later.

Note that Quorum follows the same smart contracts principles of Ethereum. By transitivity, this means the same applies to our Quared.

### 3.1 OpenZeppelin Smart Contracts

OpenZeppelin is a company that builds developers tools and performs security audits for distributed systems. Founded in 2015, they are the main reference for secure industry standards, especially for Ethereum smart contracts.

Their battle-tested smart contracts provide us with a secure and reliable range of applications that vary from preventing integer overflows to the implementations of ERC token standards. In order to leverage this security guarantee, all DT smart contracts inherit properties from OpenZeppelin’s rich libraries (see the mentioned inheritance in Figure 7).

#### **ECDSA Contract**

Provides a library whose functions can be used to verify that a message was signed by the holder of the private keys of a given address using the Elliptic Curve Digital Signature Algorithm (ECDSA).

#### **Roles contract**

This contract module allows its children to implement role-based access control mechanisms. For instance, roles can be used to restrict access to a certain function call. Addi-

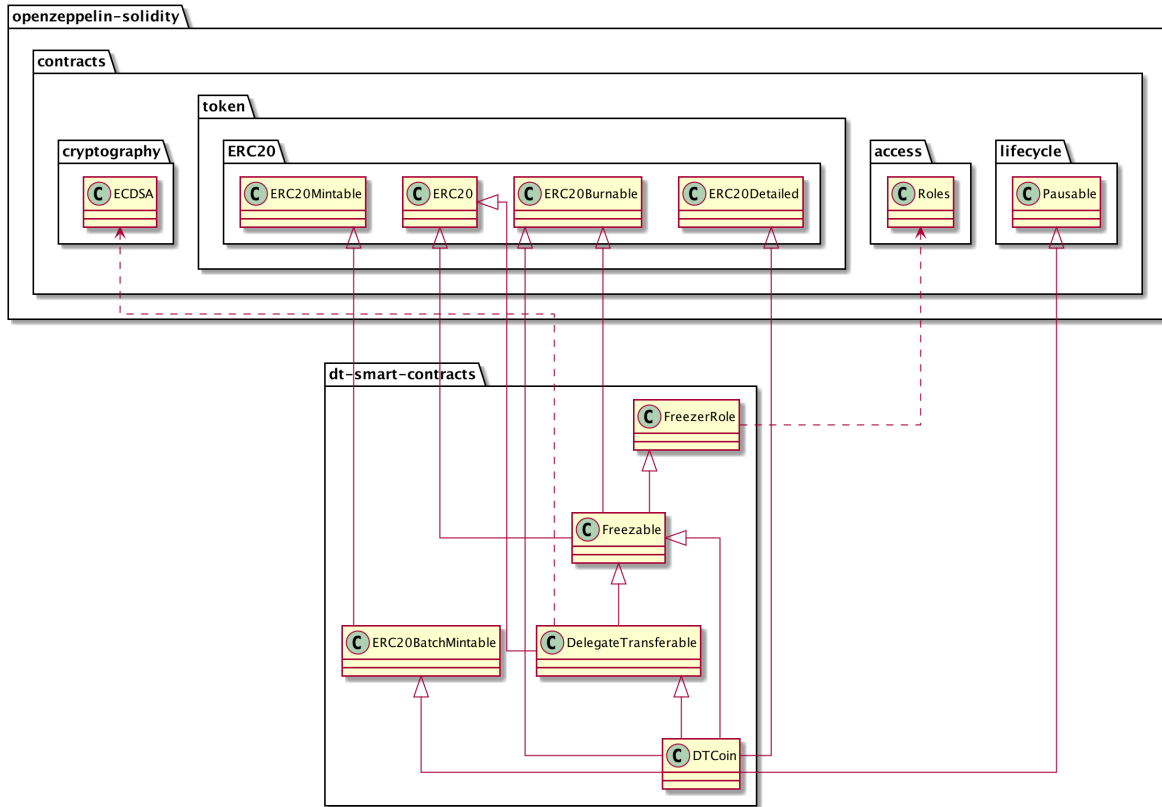


Figure 7: DT Smart Contracts inheritance from OpenZeppelin. The inheritance among OpenZeppelin’s smart contracts is omitted for clarity.

tionally, roles can be granted and revoked dynamically by accounts that have themselves the admin role.

### Pausable Contract

This contract is meant to be use as a base for other contracts and implements an emergency stop mechanism.

### ERC20 Contract

This is the implementation of the first standardised token on Ethereum. The interface is described in the Ethereum Improvement Proposal (EIP) 20 which made it to the status Ethereum Request for Comments (ERC) number 20<sup>5</sup>. Note that this implementation is agnostic of the way tokens are created; in other words it has no preconditions on how tokens are minted.

This token type is the most popular and ubiquitously supported by online services. Among its capabilities are allowing the implementation of minting and burning, transferring and allowing transfers.

<sup>5</sup><https://eips.ethereum.org/EIPS/eip-20>

### **ERC20Mintable Contract**

A canonical implementation of a minting function compatible with the ERC20 contract. This is allowed with some limits such as a hard cap on the newly minted DTCoins, and must be accepted by consensus of the the Quared Nodes.

### **ERC20Brunable Contract**

A canonical implementation of the burning (destroying) function compatible with the ERC20 contract.

### **ERC20Detailed Contract**

This contract is meant to be used as the base for the ERC20 contract, and it specifies additional detailed information: name, symbol, and (number of) decimals.

## **3.2 DT Smart Contracts**

The set of DT smart contracts compose a system to transfer tokens under certain desirable conditions:

- They allow for the transfer of previously signed funds (from the legacy token).
- Certain accounts can freeze other accounts.
- Certain accounts can pause the execution of function calls related to the token.
- The transfer of tokens and pre-signed funds can be stopped completely.
- The accounts involved in a transaction may be individually frozen whilst still allowing the rest of the accounts to operate normally.

For a detailed overview of how these contracts relate among themselves and with OpenZeppelin's, see the class diagram in Figure 8.

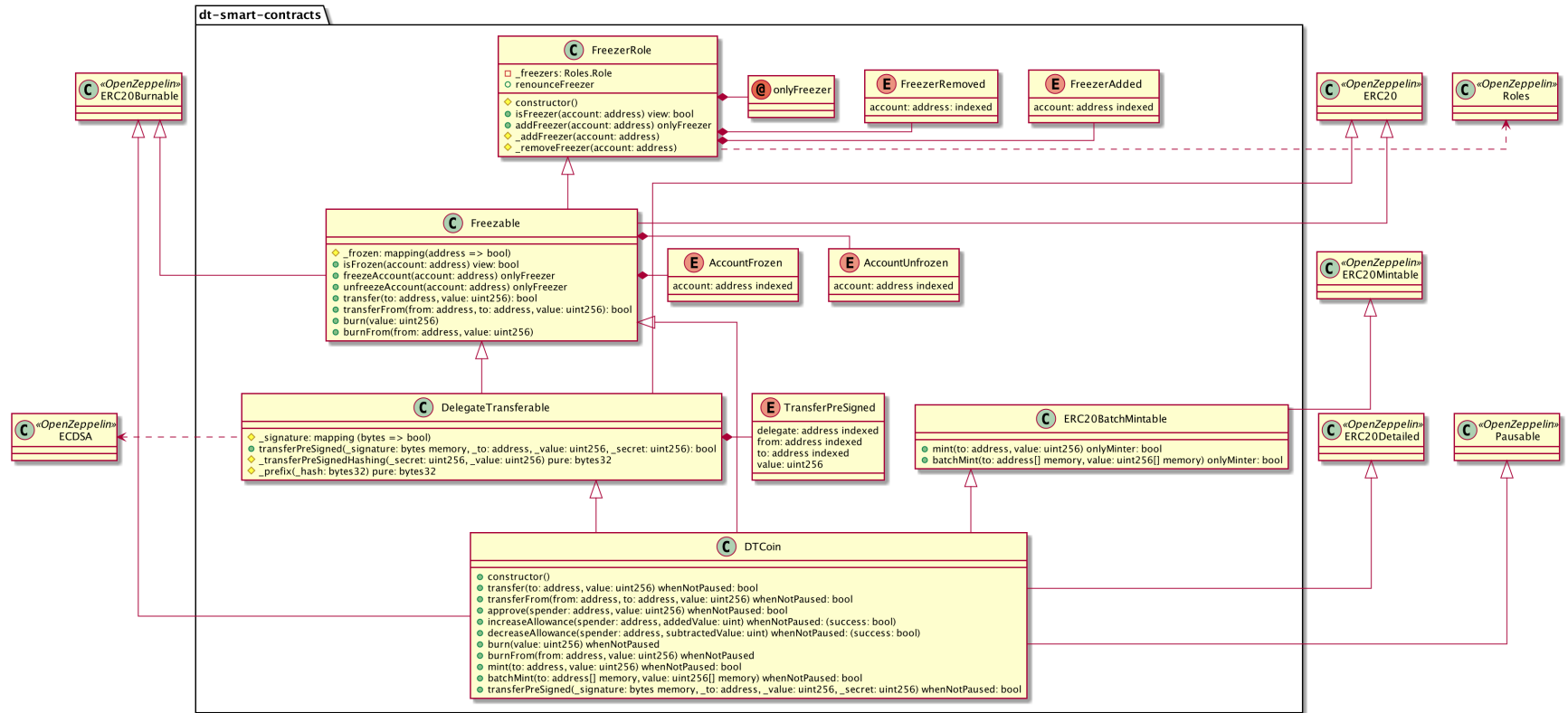


Figure 8: DT Smart Contracts full class diagram with OpenZeppelin as an external dependency.

### FreezerRole Contract

This contract uses the OpenZeppelin's Roles implementation to describe a Freezer role. An account with said role, can freeze operations by other accounts as well as adding and removing the same role to other accounts.

Additionally, this contract emits the events related to having frozen or unfrozen an account.

### Freezable Contract

This contract is simply the realisation of the FreezerRole for the transfer and burn functions.

### DelegateTransferable Contract

A Freezable implementation of an ERC20 contract that allows the transfer of pre-signed funds by verifying the signature of the claimant of said funds.

### ERC20BatchMintable Contract

As its name suggests, this is an extension of an ERC20Mintable contract that allows minting tokens to several accounts at once as a convenience function.

### DTCoin Contract

This is the ledger that holds the transactions related to the ERC20 DT Coin (symbol DTC with 18 decimals).

This contract is fundamentally an ERC20 token with the following characteristics:

- **Pausability:** its function calls are restricted to when the contract is not paused.
- **Freezability:** accounts that are subject to any operation may not be frozen.
- **Burnability:** allows tokens to be burnt.
- **Batch Mintable:** implements the convenience function to mint funds to multiple accounts at once.
- **Delegate Transferable:** allows the previously described transfer of pre-signed funds.

## 3.3 Technical Note

OpenZeppelin's smart contracts are the *de facto* standard for building smart contracts. Their security has been audited and, more importantly, corroborated by their long application history in the systems of multiple noteworthy parties. For example, these standards play central roles in well-known public smart contracts deployed on Ethereum.

Having chosen them as base for the DT Smart Contracts consequently brings along the same resilience and ease of use. Note, for instance, that the DTCoin contract is a collection of inherited properties which were deemed desirable for our use case, and that had been designed by OpenZeppelin as one can see in Figure 8.

## 4 Block Indexer

The Block Indexer is a set of services of the Quared system that enable the retrieval of information from the Quared network. Each service addresses a different concern.

Since these services have access to everything that happens in the Quared network, its unrestricted access is limited to only certain members of the team. For the general public, the Public Block Explorer, which communicates with this service, is available (see Section 5 Public and Private Block Explorers).

### 4.1 DT Indexer

This containerised service is the one responsible for listening to the events emitted by the Quared nodes. It achieves this task by initiating a web3 Crawler that stores all relevant events in a Mongo database.

#### MongoDB

MongoDB is one of the most popular databases management systems tagged under the category of NoSQL. It is a general purpose database with rich scalability features. Instead of using entries and tables, MongoDB stores documents grouped in collections.

The documents that MongoDB handles are stored in BSON (Binary JSON) and are always retrieved to the consumer as regular JSON objects. This feature makes it a suitable choice for storing the data received by the HTTP-JSON API of the Quared nodes.

MongoDB is a schemaless database; in other words, a single grouping or query may refer to several elements that do not share the same structure. In this particular case, not all retrieved JSON objects have the same properties and even if they do their values may not have the same type. Nonetheless, there are means to enforce such schema consistencies. DT Indexer leverages the popular `mongoose` library for the job, ie this library acts as an intermediary between the service and the database to enforce schemas.

#### 4.1.1 Data Models

The data stored by the DT Indexer is stored in MongoDB collections with a (mongoose) schema. Although the amount of data produced by the Quared network is vast, we are interested in indexing only those of most relevance and requested the most by other subsystems. These data points correspond to:

- Transactions (see Listing 1)
- Token transactions (see Listing 2)
- Blocks (see Listing 3)
- Uncle blocks (see Listing 4)

```
1 const transactionSchemaDefinition = {  
2   blockHash: { type: String, lowercase: true },  
3   blockNumber: { type: Number, index: true },  
4   hash: { type: String, lowercase: true, unique: true },
```

```

5     timestamp: Number,
6     input: String,
7     value: String,
8     gas: Number,
9     gasUsed: Number,
10    gasPrice: String,
11    nonce: Number,
12    transactionIndex: Number,
13    from: { type: String, lowercase: true, index: true },
14    to: { type: String, lowercase: true, index: true },
15    contractAddress: { type: String, lowercase: true, index: true },
16    logs: Array,
17    status: { type: Boolean },
18 };

```

Listing 1: Mongoose schema for transactions.

```

1  const tokenTransactionSchemaDefinition = {
2    blockNumber: { type: Number, index: true },
3    hash: { type: String, lowercase: true, unique: true },
4    timestamp: Number,
5    from: { type: String, lowercase: true, index: true },
6    to: { type: String, lowercase: true, index: true },
7    value: String,
8    contract: { type: String, lowercase: true, index: true },
9    method: { type: String, lowercase: true },
10   status: { type: Boolean },
11   logs: Array,
12 };

```

Listing 2: Mongoose schema for token transactions.

```

1  const blockSchemaDefinition = {
2    number: { type: Number, unique: true },
3    timestamp: Number,
4    transactions: Number,
5    hash: { type: String, lowercase: true, unique: true },
6    parentHash: { type: String, lowercase: true },
7    sha3Uncles: { type: String, lowercase: true },
8    miner: { type: String, lowercase: true },
9    difficulty: String,
10   totalDifficulty: String,
11   size: Number,
12   gasUsed: Number,
13   gasLimit: Number,
14   nonce: String,
15   uncles: Number,
16   blockReward: String,
17   unclesReward: String,
18   avgGasPrice: String,
19   txFees: String,
20   extraData: String,
21 };

```

Listing 3: Mongoose schema for blocks.

```

1  const uncleSchemaDefinition = {

```



```

2   number: Number,
3   position: Number,
4   blockNumber: Number,
5   hash: { type: String, lowercase: true, unique: true },
6   parentHash: { type: String, lowercase: true },
7   sha3Uncles: { type: String, lowercase: true },
8   miner: { type: String, lowercase: true },
9   difficulty: String,
10  gasLimit: Number,
11  gasUsed: Number,
12  timestamp: Number,
13  reward: String,
14 };

```

Listing 4: Mongoose schema for uncle blocks.

## 4.2 DT Indexer API

This HTTP-JSON API consists of read-only endpoints that expose the data retrieved by the indexer and then stored in the MongoDB. Its endpoints grouped by the resource type they return are as follows:

### General

- GET /status  
Retrieves the current status of the network plus the last block.

### Transactions

*See Listing 1 Mongoose schema for transactions*

- GET /transaction/:hash
- GET CENSORED
- GET CENSORED
- GET CENSORED

### Token transactions

*See Listing 2 Mongoose schema for token transactions*

- GET CENSORED
- GET CENSORED
- GET CENSORED
- GET CENSORED

## Blocks

*See Listing 3 Mongoose schema for blocks*

- GET /latest
- GET /block/:block
- GET /blockbyhash/:hash
- GET CENSORED
- GET /latestsblocks/:limit

## Uncle blocks

*See Listing 4 Mongoose schema for uncle blocks*

- GET /uncle/:hash
- GET /latestuncles/:limit

### 4.3 ethstats server

The Ethereum Network Stats (ethstats) is popular web service to consult the status of the Ethereum network. It connects to voluntary nodes that are in synchrony with the Ethereum blockchain. The code of this service is publicly available and open-source<sup>6</sup>.

Since Quared shares most of the principles of Ethereum, a simple configuration change is enough for this service to listen to our Quared nodes.

### 4.4 Technical Note

Although each of the Quared nodes possesses an entire copy of the blockchain, retrieving it may be cumbersome for some use cases and inconvenient since there is no graphical representation. The DT Indexer and related services are a practical way of having the most relevant blockchain data of the DT Infrastructure always available and at a glance. The largest added value of this set of services resides in its indexing power (as in database indexes) for a prompt access to data of interest such as transactions that involve the DT Coin.

## 5 Public and Private Block Explorers

We discussed in Section 4 Block Indexer how we obtain the most relevant information from the Quared network regarding the DT Smart Contracts. The main advantage of this approach is the database indexing for fast retrievals.

---

<sup>6</sup><https://github.com/cubedro/eth-netstats>



Figure 9: Sample screen of the Ethereum Networks Stats web service.

We also mentioned briefly the implementation of ethstats server, a web service to consult the status of the Ethereum network — in this case, our Quared network. Nonetheless, ethstats also presents information that is not relevant to a Quared network, eg difficulty, gas price, and gas limit, and does not present the information on a per-block or per-transaction basis. Lastly, it is available only to network administrators due to its privileged communication with the Quared nodes.

In this section we introduce the Public Block Explorer, meant to be open to all interested users, and the Private Block Explorer for the usage of the network administrators. Both applications serve their corresponding users with information of their interest without the need of having the privileged communication to the Quared nodes previously mentioned.

## 5.1 Public Block Explorer

The Public Block Explorer is a React progressive web application served by a lightweight server. This server manages the request that go to the DT Indexer in order to retrieve general aggregated block information and transaction information. See Figure 10 for a visual reference of its interface. This application presents a dashboard with graphs whose data comes updated directly from the DT Indexer.

The information presented to users of this application is the following:

- **Price and Market Capitalisation.** These monetary values are presented on a single graph with a double axis for an easy reference and examination of their correlation.
- **Average block time and total blocks.** As an indirect indication of the health

of the network.

- **Total transactions.** The number of DTC transactions on the network since its inception.
- **Total users and wallet signups.** The number of account holders in the DTCoin smart contract. In a time-scaled graph the user can also examine the evolution of the creation trend.
- **Transactions per second.** Also another indirect health measurement of the network and a trend indicator of activity by of the overall transactions.

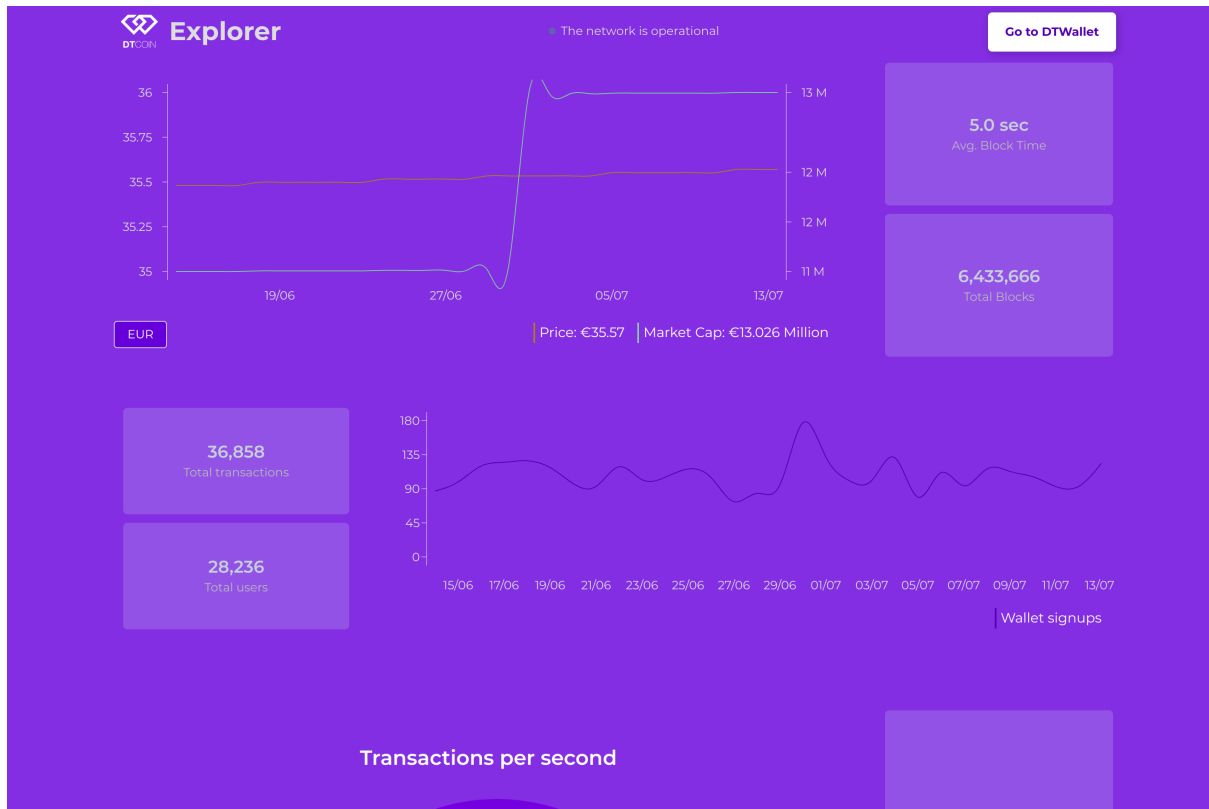


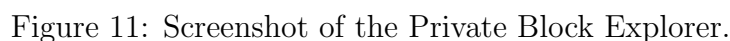
Figure 10: Screenshot of the Public Block Explorer.

## 5.2 Private Block Explorer

The counterpart to the Public Block Explorer is the Private Block Explorer which is meant for the administrators of the Quared Network.

The Quared Network is such that transactions can be decrypted only by those accounts that play a role in them, or those which have admin rights. Practically, this application has a login mechanism in place which requires the user to authenticate as an admin. The user is required to provide their wallet keystore file or mnemonic, and the passphrase to decrypt it. The application can therefore be open to the world wide web, however access

By clicking on any of the transaction, address, or block links, the user is taken to a detailed view of the data that the application can decrypt as the wallet used for authentication is a DT Network administrator's.



The display of network information on a general basis via the Public Block Explorer inspires confidence to the general users as they can inspect the aggregated blockchain data that concerns them. Given that it only presents aggregated data, it is safe to be inspected by anyone interested.

29

## 6 Proxy and Signer Proxy

The Quared network is immutable and eventually consistent by design just as the Ethereum network is. In other words, once the nodes have reached consensus on the state of the network, the events that lead to this state cannot be altered retrospectively. Additionally, it provides privacy mechanisms for obfuscating information to third parties. This leaves us with only one remaining concern: to guarantee easy and stable access to the Quorum Network's state.

The following two services address that concern for general users of the DT Infrastructure, as well as for other kinds of users such as administrators.

### 6.1 Proxy

The Quared network is private by design (given that it is Quorum-based), ie no one can see the state or contracts to which they are not a privy. However, there are still ways of *polluting* the network for accounts that are parties to a valid and trusted contract on the network. For instance:

- **Broadcasting irrelevant transactions or contracts.** Potentially, an account can deploy several contracts private to that account that couldn't be inspected by the client nodes since they don't have the keys to inspect the transactions that created them. This would result in a large increase of the blockchain size without any added value to the network and consume the throughput of transactions relevant to DT Infrastructure.
- **Requesting information excessively.** It could be the case that either maliciously or unintentionally, an account could request the nodes information about a certain state in the network several times in a short time span. This would lead to an increase of the overall latency of the network since other accounts' requests would be introduced deeper in the response queue.

For the reasons explained above, a Proxy is put in place between the DT Quared Network and the user interfaces to it, namely the DT Wallet and the DT Admin Panel, with the tasks of checking that an account is authorised to perform a certain action and that a request has not exceeded its allowance set by the rate of the requests in a certain period of time.

#### 6.1.1 Authorisation

For a request to reach a Quared node, it must have a valid cryptographic **nonce** and **nonceSignature**. The first is stored to have a one-month validity and the latter is used to verify the message meant to be passed along to a Quared node.

**Cryptographic Nonce** In order to avoid a replay attack, an arbitrary number that should be used only once in a cryptographic communication system is employed. Its purpose is to ensure that a certain request is only processed once. This means that even if a message is intercepted by a malicious actor, say a transfer request, the receiving system denies any subsequent attempt of executing a request with the same nonce.

**Nonce Signature** This cryptographic signature is made taking into account both the message and the nonce (usually by concatenating). Signatures cannot be recycled as they change drastically even with a small change in the input. This is the means to prevent a request forgery attack.

### 6.1.2 Call Rate Limit

As seen in Subsection 2.6 Performance, an increased number of transactions per second processed by the network, can contribute to a higher latency in the network. Although this increased latency is not worrisome, it is undesirable to forward requests that may very well be malicious and hinder the network's performance for all its users. For this reason, whenever a request is received, its allowance rate per minute is checked. If it has not been exceeded, the requests can proceed forward towards the Quared node. In the contrary case, the caller gets informed about having reached the limit.

**Redis** An in-memory data structure store, commonly used as a key-value pair database. It's high availability and speedy response makes it suitable for increasing the counter associated to the rate limit of an account. Redis is also highly tolerant to partition which makes it suitable for this distributed system.

## 6.2 Signer Proxy

For those users whose accounts can perform more operations, such as deploying contracts or asking private states, a proxy signer microservice is put in place. This service is meant to be run individually and privately by said users.

The service requires either the keystore or the mnemonic of their wallet along with their corresponding password to be started. Once up, it supports the standard web3 libraries as a network Provider. Clients may therefore send requests to the service instance with ease, which the Signer Proxy will sign and forward to the Quared Proxy.

## 6.3 Technical Note

The Proxy to the Quared nodes provides a means to protect the network from pollution in order to secure a top performance. Additionally, the **nonce** and **nonceSignature** checks defend against replay and forgery attacks.

Having split the signing ability for users with additional permissions allows tackling improvements to the general users's application in a more agile fashion while preserving cryptographic integrity and keeping vital administrative systems online.

## 7 Web Wallet

The DT Web Wallet is a React single-page application that enables users to interact with the funds of their account. It is the most convenient way for a DT Coin holder to inspect their balance and perform operations with their wallets by communicating with the DT Quared Proxy. Rather than a login, it requires the users to provide access to their wallet file or the mnemonic that derives it, as well as the passphrase to decrypt these. This

information is never transmitted but rather kept locally on the local storage module of the browser.

Although abstracted for user-experience purposes, it is crucial to understand what random seeds and mnemonics are, as well as the different types of cryptographic wallets that can be employed.

## 7.1 Seed Phrases and Mnemonics

In a decentralised system, there is no authority that assigns or generates accounts to a user. Any user can create an account by simply generating a **private key**, from which a public key is derived, and from the latter an address is derived. Since Ethereum private keys are 256-bit long, this implies that there are potentially  $2^{256}$  private keys. This is roughly equivalent to 115 quattuorvigintillion (a 78 digit number), which is in the range of the number of atoms in the observable universe. Therefore, gaining access to someone else's funds by guessing their key randomly, is equivalent to two different people having chosen the same atom in the sky.

Nonetheless, computers cannot generate *pure* random numbers; instead they generate pseudo-random numbers with generating functions. This kind of functions starts from a base number that is called **seed**. By feeding the algorithm the same seed, it will generate the same sequence of numbers every time. In our case, by feeding a pseudo-random algorithm with the same seed, it will always generate the same sequence of 256 ones and zeroes.

Storing this 256-bit long key is usually done in a **keystore** file. Due to the nature of electronic files, they are usually stored in media that it is not easily guaranteed to be out of the reach of malicious users. As an example, consider storing this keystore file in a computer with access to the internet and that this computer gets penetrated. Naturally, the keystore can and should be symmetrically encrypted with a password as a measure of preventing usage, should it become compromised.

An alternative method of storing the seed is by employing a **mnemonic**, a sequence of words from which the seed can be reconstructed. For the sake of demonstration, consider an ordered list of 4 words numbered with binary digits:

1. 00 book
2. 01 wardrobe
3. 10 car
4. 11 floor

Now take a fictitious 12-bit long key: 011100101001. If we split this key in chunks of 2 bits, ie 01 11 00 10 10 01, we can immediately see a one-to-one correspondence with the word sequence “wardrobe floor book car car wardrobe”. A similar method with a dictionary of 2048 words and a some checksum digits is employed to generate mnemonics of Ethereum wallets. As it is the case with keystores, a password protection can be added to a wallet mnemonic, ie instead of splitting the true key, we split the password protected version of it.

In either case, a keystore or its equivalent seed allow a user to easily back up and restore a wallet.



## BIP 32 - Hierarchical Deterministic Wallets

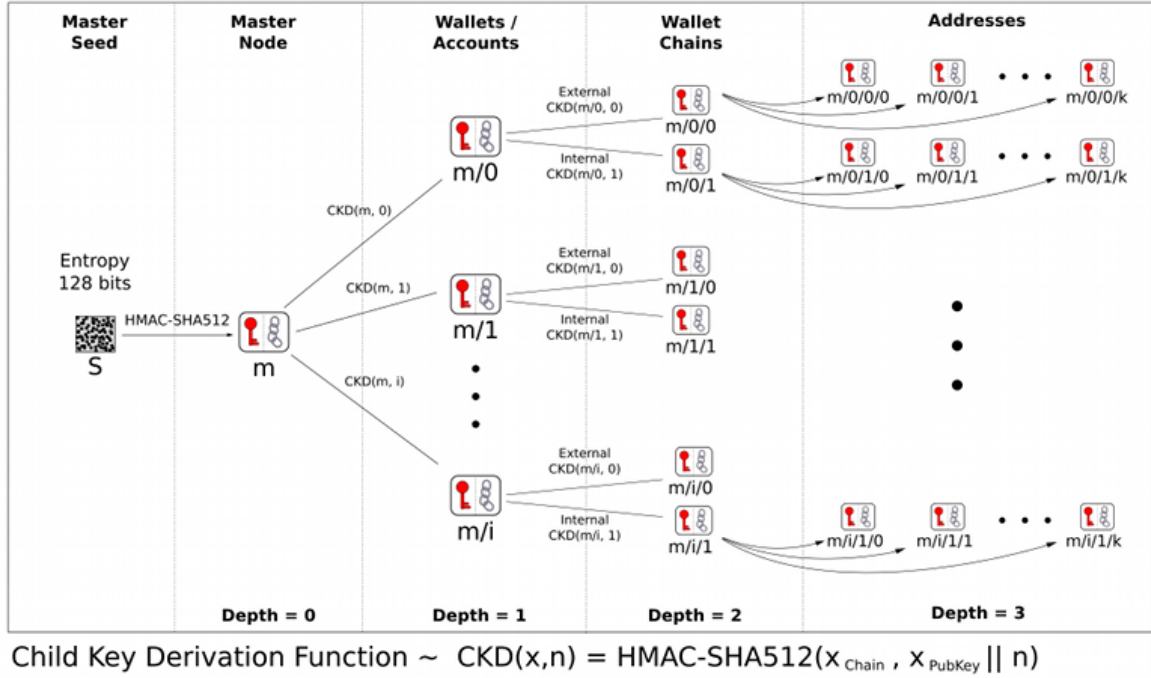


Figure 12: Hierarchical structure of wallets/keys in BIP 32

## 7.2 Wallet types

A **cryptocurrency wallet** is (private) key that allows the signing of transactions to be broadcasted to the Quared Nodes.

A **deterministic wallet** is a system of deriving keys from a single seed. Seeds are typically serialised into human-readable words in a seed phrase such as the aforementioned mnemonics. Using a deterministic wallet can enhance the privacy of a user by allowing them to receive funds in several addresses that can be *controlled* by a single seed.

**Hierarchical deterministic wallets** (HD Wallets) provide also a structure in which keys can be derived from other keys ultimately derived from the seed phrase. These derived keys are known as child keys. Any key can be use to receive and sign unspent transactions (funds) and since child keys are derived from a parent key, the parent key has potential access to funds of the child keys. This is feature was first fully described in the Bitcoin Improvement Proposal 32<sup>7</sup> and has been ported to Ethereum. It is proven most useful to emulate the accounting hierarchy of an organisation for the purposes of freezing and auditing spending. A diagram that depicts this tree-like hierarchy can be found in Figure 12.

The DT Web Wallet is of the nature of the last, ie a DT Wallet is a HD Wallet for the DT Quared Network.

<sup>7</sup><https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>

## 7.3 Operations

The capabilities of this web application enable the users to operate with their accounts in a simple interface. For reference, consider the layout presented in Figure 13

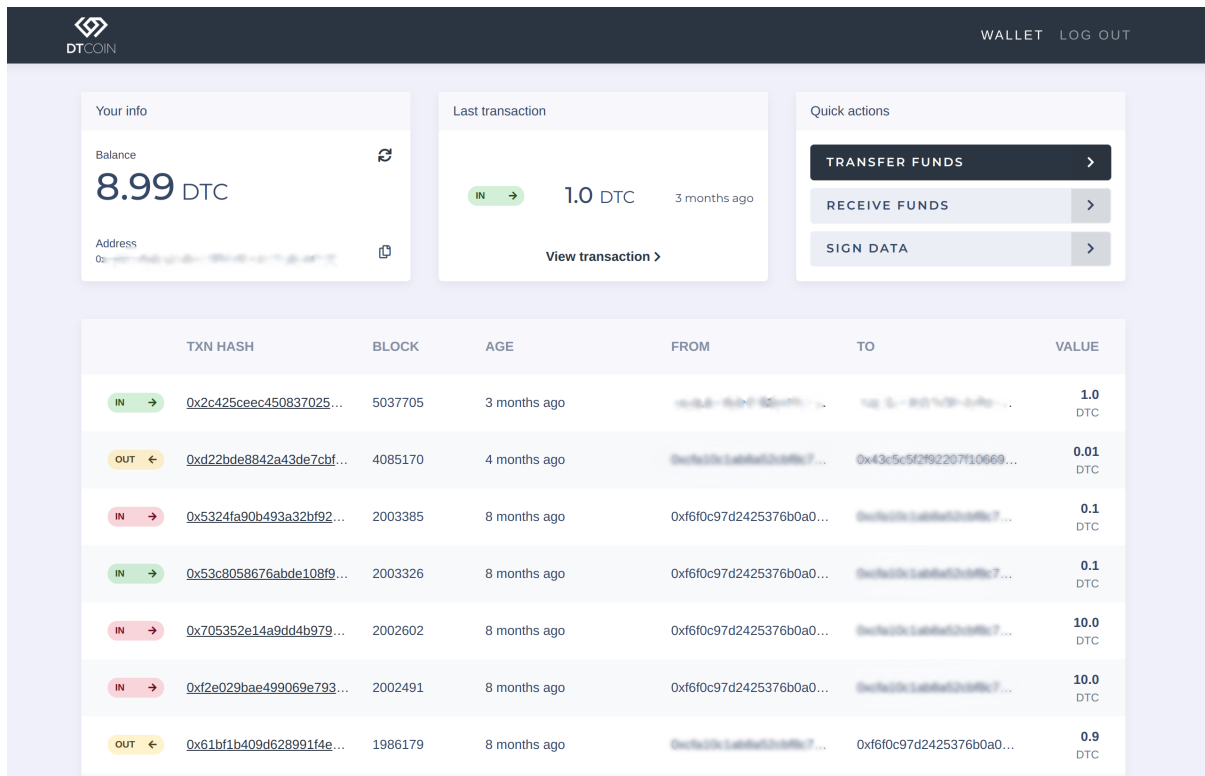


Figure 13: Web wallet application as seen on a browser.

### 7.3.1 Wallet creation

For those new users who do not have a wallet yet, there's a *wizard*-like subsystem that guides user through the process of obtaining a new wallet. Note that all these steps are undertaken without communicating to any other system, ie it all happens on the browser and can be carried out potentially offline.

**Password creation** To start, any user needs to enter a password to encrypt symmetrically the authentication method chosen in the next step.

**Chose of authentication method** The user creating a wallet for the first time has the option of choosing one of the two equivalent authentication methods: a keystore file or a mnemonic.

**Verification** Lastly, the user is asked to either upload their keystore file or to click the words in the mnemonic in the order they were given. This changes according to the authentication method used in the wallet creation process. The Verification step is in place to confirm that users have indeed safely stored their authentication method.

### 7.3.2 Funds

Once a user has signed in with their wallet, they can perform a variety of actions on the provided graphical interface:

- **See balance.** The balance of DTC corresponding to the account's address.
- **Examine latest transaction.** The latest transaction can be inspected via its quick access that leads to the transaction explorer.
- **See all transactions.** All operations in which this account was involved, are shown in inverse chronological order. By clicking on its transaction hash, the on-chain details of transaction are shown.
- **Transfer funds.** Account funds can be transferred to other users specifying the amount and the receiver's address.
- **Receive funds.** This convenience functionality allows the creation of a custom URL with a fixed amount of DT Coin for the intended sender. By entering this URL into a browser, the intended sender of funds goes directly to the fund transfer functionality.
- **Claim legacy tokens.** With this action, users that have legacy funds can claim them in the new system, and receive the equivalent DTC amount in their wallets. See Section 10 Legacy Token Migration Service.
- **Sign data.** Creating a cryptographic signature and appending it to a certain text message. Usually employed to prove ownership of an account or funds.

Note that these operation are conditional to the account not being frozen on the DTCoin contract. In this case, a notification is shown informing the user about the status of the account.

## 7.4 Technical note

The Web Wallet provides any user with a familiar way to interact with their funds. It works basically as any other web wallet with an elegant user interface. It also handles gracefully the request for funds so that intended senders, that which to be careful not to enter a wrong recipient address, can instead open a URL with pre-loaded data concerning the transfer they wish to perform. By using HD Wallets, the Web Wallet leaves the possibility open to generate multiple hierarchical wallets for a more organised accounting of the funds.

## 8 Admin Panel

As previously described in Subsection 3.2 DT Smart Contracts, some accounts perform actions other than just fund transfers as a regular user through the Web Wallet. Namely, those actions are:

- Granting and revoking the permission to pause the DTCoin smart contract, to freeze individual addresses, and to mint new DTC
- Pausing and resuming the execution of the DTCoin smart contract's functions
- Freezing and unfreezing individual addresses, ie forbidding them to interact with the DTCoin smart contract
- Minting new DTC to one or many given addresses
- Adding new admins

This dashboard (see Figure 14) was developed to simplify the execution of these administrative functions for the DT Network staff. To log in to this application, they just need to enter their authentication method, ie their keystore or mnemonic along with the corresponding password.

These actions must be mandated and subsequently approved by a consensus of DT Nodes to be enforced. While Administrators can perform these functions, it is the Nodes which either accept or reject the state changes. Administrative functions which are not approved by the DT Nodes will not be implemented at the DT Network level.

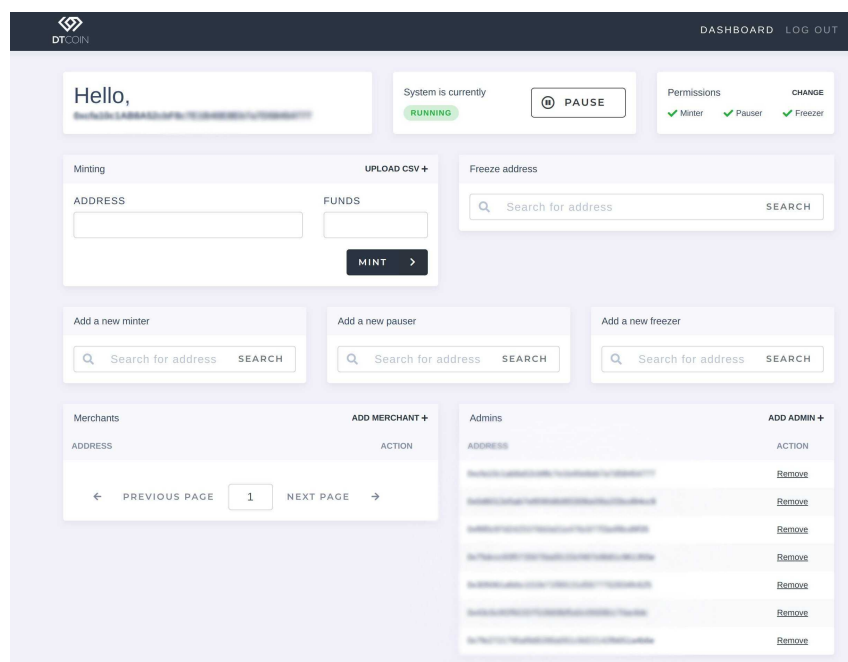


Figure 14: Admin Panel web application.

## 8.1 Technical note

As it was the case with the public and private block explorers, having split the dashboards of regular users from that of more privileged users enables for quicker response times to either application in case of an urgent improvement or fix without sabotaging the integrity of the DT Platform in general.

## 9 Address Verification Service

This component aims to provide an alternative, cryptography-based authentication system for affiliated digital platforms. It would allow users to manage their digital identities across independent platforms without revealing sensible details, and while still preserving data reliability and guaranteeing proof of ownership. Such a feature will eventually be leveraged to manage access to applications throughout the DT Ecosystem and beyond, as well as certifying data ownership in the DT Circle.

## 10 Legacy Token Migration Service

The DT Legacy Token migration service provides utility tools for conducting the process of migrating legacy token balances to the new blockchain. The process is conducted by the social network and involves minting an appropriate amount of tokens on the Quared blockchain and generating and distributing signatures for delegated transfers.

**Legacy Token** This token is a database-managed credit system. In this system, the initial users of the DT Social Network have funds that need to be migrated in order to use the new system. This migration is done by means of creating a delegated transfer that the DTCoin smart contract accepts.

**Delegated Transfer** This framework was born in the open Ethereum network as the response to a common problem in the token economy: the use of two currencies, Ether and the token itself. In our particular case, since no Ether is involved, the two currencies in question are the legacy token and DTC.

This microservice accepts an authorised request in order to create a delegated transfer with its amount, nonce, and signature. Upon receiving this object as a response, the user—or rather the DT Wallet—will request the legacy token funds to the DTCoin smart contract.