

# Catalyst Distributed Operating System Protocol

Dr. Pauline Bernat, Joseph Kearney and Francesca Sage-Ling

May 30, 2019



# Contents

<b>Glossary</b>	<b>5</b>
<b>Introduction</b>	<b>7</b>
<b>1 Technical Specifications</b>	<b>9</b>
1.1 Choice of Elliptic Curve . . . . .	10
1.2 Choice of Hashing Algorithm . . . . .	10
1.3 Zero-Knowledge Proofs . . . . .	11
1.4 Tokens . . . . .	12
<b>2 Peer-to-peer Communication Protocol</b>	<b>15</b>
2.1 Peer Identification Protocol . . . . .	15
2.2 Peer Messaging . . . . .	16
2.3 Gossip Protocol . . . . .	16
2.4 Peer Role Types . . . . .	16
2.5 Nodes Registration and Worker Pass . . . . .	17
<b>3 Catalyst Distributed Ledger</b>	<b>21</b>
3.1 Ledger Database Architecture . . . . .	21
3.2 Accounts on Catalyst . . . . .	22
3.3 Ledger Current State and DFS . . . . .	22
<b>4 Transactions</b>	<b>25</b>
4.1 Transaction Types . . . . .	25
4.2 Transaction Structure . . . . .	25
4.3 Transaction Entries . . . . .	26
4.4 Transaction Signature . . . . .	27
4.5 Transaction Validity . . . . .	29
<b>5 Consensus Mechanism</b>	<b>31</b>
5.1 Background . . . . .	31
5.1.1 Motivation . . . . .	31
5.1.2 Naming Convention . . . . .	33
5.2 Democratic Consensus . . . . .	34
5.2.1 Computation Phase . . . . .	34
5.2.2 Voting Phase . . . . .	36
5.2.3 Synchronisation Phase . . . . .	39
<b>6 Catalyst at Large Scale</b>	<b>41</b>
<b>7 Benchmarking</b>	<b>43</b>

<b>8 Security</b>	<b>45</b>
8.1 Security of the Consensus Mechanism I . . . . .	45
8.2 Security of the Consensus Mechanism II . . . . .	49
8.3 Security of the Signature Scheme . . . . .	49
8.3.1 Rogue Key Attack . . . . .	49
8.3.2 Quantum Attack . . . . .	49
<b>Conclusion</b>	<b>51</b>

# Glossary

**Account (definition types)** – There are two types of accounts on the catalyst network. The first is non-confidential accounts, within which the users public key and their account balance is held. This Account balance is readable to anyone. The second type are confidential accounts. Within which the public key is held, along with the amount, which is obfuscated using a blinding factor, it is held in the form of a commitment.

**Account inertia** – Account inertia refers to the activity of an account. An account with low inertia is one that is frequently utilising the network. An account with high inertia is one that is retaining tokens but not creating work for the network.

**Blockchain** – A blockchain is a peer-to-peer immutable decentralised ledger of information. It can be considered a decentralised database. Transactions created on a blockchain are bundled together into blocks, which are linked together using the hash of the previous block. It provides an indefinitely traceable history of all transactions that have taken place on the network.

**Confidential Transaction** – A transaction within which the amounts of KAT tokens transferred are not visible to all through the use of cryptographic techniques. The validity of the transaction can still be checked without revealing the actual amount.

**Consensus Mechanism** – Consensus is a method of reaching agreement on a set of proposed changes submitted by users during a period of time. This changes the state of the ledger to reflect these agreed changes. Consensus on Catalyst takes a different approach, it uses a collaborative approach among nodes to generate a correct update to the ledger.

**dApp** – dApp refers to a Distributed Application and describes an application running on multiple nodes simultaneously with the outputs fed into the consensus mechanism to ensure the network agrees on the result.

**Distributed File System (DFS)** – This is a storage mechanism, within which there is no single point of storage rather an entire network. Allows files to be stored in an efficient and distributed manner. DFS is used to store files as well as historical ledger state updates. DFS is maintained by all nodes on the network.

**Distributed Ledger Technology (DLT)** – All blockchains are distributed ledgers, not all distributed ledgers are blockchains. It can be considered a database where there must be no central source of storage. Catalyst uses a ledger-based system where updates are made at each ledger cycle. These updates are used to change the overall state of the ledger.

**KAT** – A medium of exchange used on Catalyst Network, enabling users to perform actions on the ledger such as accessing services provided on the network or storing and retrieving files.

**Ledger Cycle** – A fixed period of time after which the ledger state is updated using a consensus drawn by the producers. It is comparable to the block time in traditional blockchain.

**Node** – A node is a device connected to the other nodes (its peers) on a peer-to-peer network. A node could be a physical device, like a single-board computer, or running in a virtual machine or containers, such as Docker.

**Pedersen Commitment** - The use of a blinding factor to create a commitment that obfuscates the amount. The use of math on elliptic curves allows the user to prove no coins were created or destroyed.

**Producer node** - The group of peers that have been selected to perform management work on the ledger for a specific ledger cycle. These producers collect new tokens as reward for the work they performed.

**Range Proof** – Range proofs are used to determine the validity of a hidden value. The range proof allows the user to demonstrate unequivocally that the value being declared is within a specified range, without revealing the actual amount.

**Smart Contract** – Smart contracts are computer programs that define sets of rules and requirements and are deployed on the blockchain or ledger. Such program can be triggered by transactions or messages generated by other codes, or once certain requirements have been fulfilled.

**Transaction** – Defined as a message broadcast on the network that represents the transfer of KAT tokens to and from a set of digital addresses. A transaction can be non-confidential (amount being transferred is visible to all) or confidential (amount in an entry is obfuscated using zero-knowledge argument technique).

**Wallet** – Cryptographically secured software used to send receive and store tokens. It holds a users private keys well as the Catalyst address.

**Worker node** – A peer registered for work queue that has been granted a pass for a finite period of time which entitles it to contribute to the ledger database management. This node can be selected at random to become a producer for a ledger cycle.

**Worker Pool** - The group of nodes that have been granted a worker pass for a finite period of time. These nodes for that period of time have a chance of being randomly selected to perform validation work for a particular ledger cycle.

**Worker Queue** - A queue of all the nodes that have declared themselves ready and capable of performing work for the ledger, yet have not been granted a worker pass. The worker pass allows peers to move from the work queue to the worker pool

**51% attack** – 51% attack is the primary attack of concern for all blockchains. It can be described as where a malicious user or group of users collude together to control more than 50% of the nodes selected to produce the correct update of the ledger for a cycle. From there they can perform malicious activities e.g. spend their tokens twice.

# Introduction

This paper presents the core protocol behind Catalyst network. It gives an overview of the database architecture of the ledger and the structure and different types of transactions supported on Catalyst. A new consensus-based protocol based on the collaborative work performed by the network nodes and that uses the computing resources available across the network to reach a consensus is presented. This paper is organised as follows:

- **Chapter 1 - Technical Specifications:** this chapter describes the cryptographic libraries and tools used in Catalyst code base, including the choice of elliptic curve, hashing algorithm and the zero-knowledge proof protocols.
- **Chapter 2 - Peer-to-peer Communication Protocol:** this chapter describes the process followed by nodes joining the network and the process of peer identification. The different roles of nodes on Catalyst are explained as well as the process to register on the network in order to perform work related to the network (and ledger database) management.
- **Chapter 3 - Ledger Database Architecture:** this chapter gives an overview of the ledger database architecture as well as the different type of accounts stored on Catalyst. The concepts of current ledger state and Distributed File system for storing ledger state updates and files on the ledger are introduced.
- **Chapter 4 - Catalyst Transactions:** this chapter introduces the different transaction types supported on Catalyst and describes the transactions structure, including the process followed by users to generate and validate transaction signatures.
- **Chapter 5 - Consensus Mechanism:** this chapter presents the new consensus mechanism implemented on Catalyst.
- **Chapter 6 - Catalyst at Large Scale:** this chapter describes how sharding techniques are implemented on the Catalyst network, how the ledger is partitioned and the consensus mechanism adapted to function on a shard ledger.
- **Chapter 7 - Benchmarking:** this chapter presents results of expected performances of the network for different scaling scenarios.
- **Chapter 8 - Security:** this chapter discusses security considerations with regards to the signature scheme and the consensus protocol on Catalyst.





# Chapter 1

## Technical Specifications

This chapter gives an overview of the cryptographic libraries and tools used to define objects (transactions) and the consensus-based protocol on Catalyst ledger that permits the update of the decentralised ledger across its peer-to-peer network.

Distributed Ledger Technologies (DLT) can be described as a distributed database managed by a peer-to-peer network of computers. Many forms of data, from simple text files to media files or bank accounts can be stored on a database. In centralised network, a database is typically managed by a central computer or server and some part of the database are accessible to users. On the contrary, decentralised database are replicated across the network, each computer holds a local copy of the database. The database is no longer managed by a central authority but instead by a plurality of computers (or nodes) on the network. The replication of the database across multiple nodes removes the vulnerability of single point of failure found with centralised database. Users can exchange digital data stored on a database via exchange requests, referred as transactions. To generate an exchange of data, the transactions are signed by the owners of the data being exchanged. Nodes on the network agree on the validity of the transactions issued by users via a consensus-based protocol thus authorising the transactions to take place and the database to be accordingly updated across the network.

DLT's relies on the generation of cryptographically secure transactions in order to remove the need for a central authority. The ownership and exchange of data is made possible via the use of asymmetric encryption where users hold public / private key pairs. The public key can be made visible to all users and is derived from the private key solely known by the user. Public keys act as users pseudonym on the network. Knowledge of the private key is necessary to successfully sign a transaction, the digital signature proving ownership of the data being transferred to another user. While it is impossible to derive the private key from a public key or digital signature with classic computers, it is easy to verify, given a public key, that a signature could only have been generated by the user in possession of the associated private key.

Two common asymmetric encryption techniques are RSA and Elliptic Curve (EC) cryptography. RSA's hardness relies on the difficulty of integer factorisation of large prime numbers. EC cryptography relies on the hardness of the discrete logarithm problem. In DLTs, EC cryptography is chosen in preference to RSA due to the significantly smaller key size for the same level of security. On Catalyst, EC-based private keys have a 256-bit size which provides a 128-bit security. An equivalent RSA-based key would have a 3072-bit size.

## 1.1 Choice of Elliptic Curve

On the Catalyst ledger, Elliptic Curve (EC) cryptography is used to sign messages and generate proofs of knowledge of information amongst users without having to reveal the said information. Throughout this paper, EC points are used for the creation of public keys and Pedersen Commitments (PC). We use EC cryptography techniques to generate and verify the signature of transaction encompassing the transfer of KAT tokens (see section 1.4) from or to accounts locked by the private keys of Catalyst users. EC cryptography techniques are also used for the generation and verification of range proof that any number (*i.e.* an amount of tokens) hidden in a PC can be provably shown to lie within a range of acceptable values.

There exists many types of EC, some of which are part of NIST curves. While the NIST curves, such as secp256r1 curve, are advertised as being chosen verifiably at random, there is little explanation for the seeds used to generate these. By contrast, the process used to pick non-NIST curves, such as the twisted Edwards Curve25519 used in Monero [11], is fully documented and rigid enough so that independent verification can and has been done. This is widely seen as a security advantage, since it prevents the generating party from maliciously manipulating the curve parameters [12]. Moreover, EC such as Curve25519 are designed to facilitate the production of high-performance constant-time implementations.

On the Catalyst Ledger we opt for the twisted Edwards Curve25519 [13], that is a birational equivalent of the Montgomery curve Curve25519. It is defined over the prime field  $\mathbb{F}_p$  where  $p = 2^{255} - 19$ , by the following equation:

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

The order of Ed25519 can be expressed as  $N = 2^cl$  with  $c$  a positive integer and  $l$  a 253-bit prime number.  $N$  is a 76-digit number equal to:

$$N = 2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$$

Elements in  $\mathbb{F}_p$  are 255-bit integer and can thus be represented in 32 bytes with the most significant bit set to 0. An EC point on the twisted Edwards Curve25519 would therefore be represented with 64 bytes. But given point compression techniques described in [11] (section 2.4) it is possible to reduce a EC point to a 32-byte data where 255 bits represents the  $x$  coordinate of the point and the last bit indicates the  $y$  coordinate.

In general terms, an elliptic curve,  $y^2 = x^3 + ax + b$ , is defined over a prime field  $\mathbb{F}_p$  where  $p$  determines the maximum values of  $x$  and  $y$ , the two coordinates of an elliptic curve point. The elliptic curve has a cyclic group of  $n$  points. A EC generator,  $G$  for instance, is a EC point generating a cyclic subgroup of order  $l_G \leq n$ . This subgroup is composed of the set of points:  $\{0G, 1G, 2G, \dots, (l_G - 1)G\}$  with  $0G = l_G G$  known as the point at infinity. This subgroup is defined by the relation  $xG = (x \bmod l_G)G$ . The order of the subgroup  $l_G$  is a divisor of  $n$ . For instance, if  $n = 100$ , then  $l_G$  can take a value in  $\{2, 5, 10, 25, 50\}$ . Note that  $n$  can not be a prime number. If  $l_G = n$ , the subgroup of  $G$  includes all the points of the elliptic curve.

## 1.2 Choice of Hashing Algorithm

Hashing generally refers to algorithms used to obfuscate data by generating a summary of the data, or hash, in such a way that the original data can not be restored using the hash, *i.e.*

the hashing function is a one-way function, while the hash can be used to prove knowledge or ownership of the original data. A hashing function generates a pseudo-random string of fixed length from a data of arbitrary length. The hashing algorithm is said to be collision-resistant when the probability to generate two hash from two different data is negligible. Furthermore a hashing algorithm has the property that two similar data will lead to very different hashes, that is to say a collection of hashes does not allow an entity unaware of the original data to acquire knowledge about the data.

The hashing algorithm used on Catalyst ledger is 256Blake2b (or simply Blake-2b) which produces a 256-bits string and is known to be amongst the fastest hashing algorithms and particularly suitable for mobile applications [14]. Throughout the document, the hashing function is referenced by the symbol  $\mathcal{H}$ .

### 1.3 Zero-Knowledge Proofs

Confidential transactions on blockchains were introduced by G. Maxwell [9] as new data structures to allow for the transfer of tokens between digital addresses in such a way that only the amount or number of tokens exchanged is hidden, offering more privacy to the users. The amount obfuscation is achieved via the use of Pedersen commitment (PC). A Pedersen commitment is of the form  $C = vH + bG$  where  $G$  and  $H$  are two distinct generators of the EC,  $v$  is an amount of tokens hidden in  $C$  and  $b$  is the PC mask, often referred as a blinding factor.

In this paper, PC are used to obfuscate the amount of tokens  $v_i$  associated to an account stored on the ledger. The PC has the following form:  $C_i = v_iH + b_iG$  and obfuscates the balance of an account in KAT tokens, represented by an integer  $v_i \in \mathbb{Z}_M$  (with  $M$  the maximum number of tokens defined in Catalyst system). Said balance is hidden using a blinding factor  $b_i \in \mathbb{Z}_p$  ( $p \gg M$ ).  $G$  and  $H$  in  $C_i$  are two distinct generators of the elliptic curve. These two generators are in fact two different base points of the same subgroup of EC points such that the discrete logarithm is preserved, *i.e.* the  $x$  value in the relation  $xG = H$  (or  $xH = G$ ) is unknown. As a result, the two EC points  $v_iH$  and  $b_iG$  are added to form a valid EC point, *e.g* the Pedersen Commitment. Using compression techniques, an EC point on Curve25519 amounts to 32 bytes, leading to a Pedersen Commitment (PC) size of 32 bytes.

Given the cyclic property of an EC, a PC of the form  $C = xH + yG$  can therefore be rewritten as:

$$C = (x \bmod l_H)H + (y \bmod l_G)G$$

Where  $l_G$  is the order of the generator  $G$  (or number of points on the elliptic curve defined over  $G$ ) and  $l_H$  is the order of generator  $H$ .

The use of PC provides a cryptographically secure method to mask the amount spent (or received) in a transaction. Furthermore, the sum of the PCs in a transaction can be used to prove that the sum of the amounts spent and received in a transaction amounts to 0 KAT tokens, *i.e.* the transaction does not create or destroy tokens. Assume a transaction with  $n$  PCs, the sum must verify:  $\sum_{i=1}^n C_i = 0H + xG$  where  $xG = \sum_{i=1}^n b_iG$ .

On the Catalyst ledger we take advantage of the cyclical nature of elliptic curve and allow the use of positive as well as negative numbers of tokens to be contained within a PC. In truth these negative numbers are actually positive and very large numbers. For example a

user sending 5 KATs would make a commitment including a negative amount  $-5$  as follows:  $C = (l_H - 5 \bmod l_H)H + (y \bmod l_G)G$ . As explained in chapters 3 and 4, the use of positive and negative numbers offers a better anonymity solution to users as the nature of a transfer embedded in a commitment (whether it consists in spending or receiving tokens) needs not be specified in a transaction. The group of commitments in a transaction can simply be added together in order to verify no new tokens are created (or tokens destroyed) in the transaction.

Since  $0H = l_H H$ , it is in practise be possible to generate a PC with a very large number of tokens, with the malicious aim to create new tokens while producing a valid PC sum. In order to circumvent this problem, we use range proofs. Range proofs enable a user to prove that an amount lies between a specific range of values without revealing the amount. The range of values chosen for a range proof is  $[0, M]$  where  $M$  represents the maximum number of tokens allowed per account ( $M \ll l_H$ ).

Confidential transactions have a cumbersome feature with respect to non-confidential ones, that is a clear increase of a transaction size as well as the generation and verification times. The range proof associated to a transaction PC is the primary cause for the transaction size increase. This leads to a lack of scalability and a significant increase in transaction fees compared to non-confidential transactions. The Bulletproof protocol is a new zero-knowledge proof protocol [10] which proposes an improved inner product argument algorithm which results in a greatly reduced size of the range proof associated to a PC. While traditional range proof sizes are typically linear in the bit-length  $n$  of the range proof, Bulletproof provides a significant saving by creating range proof where only  $[2 \log_2(n) + 9]$  group and field elements are required. Moreover, Bulletproof protocol allows to generate aggregated range proofs with a size that grows logarithmically in the number of commitments, offering a faster batch verification time. The range proof generated for confidential transactions on Catalyst ledger are produced using the Bulletproof protocol.

## 1.4 Tokens

Catalyst Network native network token is called KAT (in reference to Katal, the unit of catalyst activity). A KAT provides the network with the functionality to pay for network services or receive value for the provision of network services and derives its intrinsic value from the development and use of the network and hence provides utility for both work undertaken by producer nodes and use of the network.

A Fulhame (FUL) is the smallest amount with KAT tokens, representing 0.00000000000001 KAT (a thousand billionth of a KAT token), named in homage to the chemist who invented the concept of catalysis, Elizabeth Fulhame.

KAT is a utility token and as such aims at providing Catalyst network users with access to services supported by dApps and Smart Contracts. The tokens are not designed as an investment although the value of the tokens can vary according to the demands for services on the Network. These tokens are considered medium of exchange as these can be used to facilitate the sale, purchase or trade of services on the network. Such trades take place via the use of transactions created by users and broadcast on the network.

The transfer of KAT tokens between user accounts are embedded in transactions. The transactions are processed by some nodes on the network (as discussed in section 2.4) which are

tasked with verifying said transactions and using these to produce a valid update result of the balance of these accounts stored on the distributed ledger. The ledger database needs to be frequently and securely updated to account for these token transfers. A healthy network thus relies on a robust mechanism to manage the ledger database in a decentralised manner. Catalyst consensus-based protocol (Chapter 5) is implemented to incentivise users on the network to contribute to the ledger database management, offering them tokens as reward for their work. Such reward typically comprises of two components: a) tokens paid by the users issuing transactions and directly debited from their accounts, in the form of transaction fees; b) new tokens injected (or released) into the system. The token supply model adopted for Catalyst base currency (KAT tokens) is inflationary-based: the number of tokens injected into the system (annually) will be a fraction of the total amount of circulating tokens adjusted to ensure a healthy and growing network. The economic consideration defining the token supply model of KAT are beyond the scope of this document.



# Chapter 2

## Peer-to-peer Communication Protocol

In Blockchain and DLT peer-to-peer communication allows messages (including but not limited to transactions) to be propagated across a network. Peer-to-peer networks rely upon information to be passed between nodes in a fast and efficient manner. The protocol to propagate messages needs to be such that the large majority of nodes receive accurate messages in a timely manner. Catalyst implements a gossip protocol to propagate messages amongst peers. Gossip protocols, also known as epidemic protocols are named as such because of how they spread information. During the protocol a node will propagate the message to a number of its randomly chosen connected peers. As nodes receive the message they, propagate the message their peers. This allows the message to spread rapidly with a high level of coverage.

### 2.1 Peer Identification Protocol

Catalyst implements a peer identification protocol. Each node that joins the network must have a peer identifier that describes the node's identity. This allows users to track their connected peers as well as associate a reputation to each node. A peer identifier consists of the following set of parameters:

- **Client ID** - a 2-bytes number to differentiate test networks from live network.
- **Client Version** - a 2-byte version number of code running on peer
- **IP** - a 16-byte IP address, IPv4 or IPv6
- **Port** - a 2 byte port number
- **Public Key** - a 20-byte public key

Each peer identifier on the network should be unique. The peer identification protocol allows the Catalyst network to prevent Sybil attacks, as discussed in ??, and to assign a reputation to the peer identifier to track badly performing nodes. The reputation of a node will be determined by how their peer identifier is formed. To maintain a good reputation, nodes will need to respond correctly to requests from their peers. For example a peer will send the IP:Port number chunk of a node, and the corresponding node will be required to respond with the associated public key chunk associated. The node reputation decreases if provided the wrong public key or not being available while a correct response is rewarded by an increase in reputation. Multiple nodes joining on the same IP space will see their reputation decrease.

Peer discovery on the Catalyst network is performed using a Metropolis-Hastings Random Walk with Delayed Acceptance (MHRWDA). This random walk is designed to cause a high

cost to eclipse attacks from malicious nodes. When a new node joins the network, it connects to a random seed peer. It then performs a random walk through the peer network, recording discovered peers only after an initial burning period. This algorithm reduces the chance of revisiting previously checked nodes, as well as reducing any bias towards nodes which have a large number of peers.

Messages will be relayed by all nodes on the network. Catalyst implements a gossip protocol to spread messages across peers. Gossip protocols propagate messages through a network by relying on each node to communicate with their neighbours. This peer identification protocol allows nodes to check the identity of peers they are connected to, as well as their roles on the network.

## 2.2 Peer Messaging

Work in progress

## 2.3 Gossip Protocol

Work in progress

## 2.4 Peer Role Types

Peers on the Catalyst network can participate in the network and assume a variety of roles. These include:

- **User node** - default state of all nodes on the network. The role of a user node is to receive transactions, check the transactions validity and when valid, forward these to their peers. User nodes can also generate transactions and observe the network. However, they are not entitled or required to perform any work on the network.
- **Reservist node** - A node that has signalled its intent to perform work for the network and provided proofs of its available computing resource dedicated to the network.
- **Worker node** - A node that has been granted a worker pass for a finite period of time.
- **Producer node** - A worker node that has been selected to perform work for a particular ledger cycle . A producer node will be rewarded for performing good quality work by receiving tokens.
- **Storage node** - A node able to sell some of its spare storage to allow other users to store there data in a decentralised manner.

All nodes on the network will receive transactions, validate and forward these transactions to other peers that they are connected with. This is to allow efficient propagation of transactions across the network.

Reservist nodes upon registering to perform work on the network are placed at the back of a node queue (or work queue) from which they must wait to be given a worker pass. This pass grants them to become a worker node and a member of the worker pool for a finite period of time. During this period of time several ledger cycles happen and the worker node has a



chance of being randomly selected to become a producer for any ledger cycle. The producer nodes are the network peers that build the ledger state update, as described in ??.

## 2.5 Nodes Registration and Worker Pass

The selection of producers among the pool of workers can be achieved for each ledger cycle using a randomised approach. Since a producer generates a ledger state update for a ledger cycle based on transactions collected during the previous ledger cycle(s), such assignment to a node should be revealed at least one cycle ahead. Assume that at the beginning of a ledger cycle  $C_n$  at time  $t = t_{n,0}$ , a random number  $r$  is drawn using the Merkel tree of the ledger state update produced one cycle ahead ( $C_{n-1}$ ) as seed to the PRNG. The random number  $r$  is used to define the list of workers selected to become producers for the next cycle  $C_{n+1}$  in the following way: for each worker node with an identifier  $Id_i$  we define the quantity  $u_i = Id_i \oplus r$  where  $\oplus$  is an XOR function (for binary-based modulus addition). The list of new identifiers  $\{u_i\}_{i=1,\dots,N}$  ( $N$  is the total number of nodes in the pool of worker) is sorted in ascending order and the first  $P$  identifiers in that list correspond to the nodes selected to be producers for the cycle  $C_{n+1}$ .

In a large network, it can be anticipated that a large number of nodes have available resource to be used to manage the ledger database and try to join the pool of workers (which we can translate as a high demand for work). The size of the pool of workers must however be determined by security as well as economic factors. Indeed, it must be profitable for a node to join the pool of workers. Said otherwise, the average number of tokens earned by a producer over a period of time should cover its operation cost. As there might be more nodes willing to work than required for the pool of workers, nodes may join a secondary pool, *a.k.a* work queue, a wait to be called to join the pool of workers. In order for these nodes to join the pool of workers, there must be a mechanism that limit the time period during which a node can be a worker. The approach considered is to grant nodes joining the pool of worker a worker pass which is valid for a limited period of time. For security reasons explained in section 8.1, rather than defining a strict expiry time, we associate a decay time to the validity worker pass. Similarly to unstable nuclear elements the work pass has a 50% chance to remain valid after the worker sits in the pool of workers for a period of time equal to the decay time. The decay time is not fixed over time but instead can vary depending on the quality of work performed by the worker when selected to be a producer. It can also vary to account for the demand for work, *i.e.* the length of the work queue, although a threshold is considered to mitigate the risk of a malicious entity (or group of entities) trying to simulate a highest demand for work than reality, as explained below.

The list of identifiers of nodes in the pool of workers can be maintained in a dynamic hash table of fixed length ( $DHT_w$ ) distributed across the network. Such table also stores the decay time of each node worker pass. ADD FORMULA PROBABILITY.

At the end of a ledger cycle, nodes in the network can therefore deduce the list of worker passes which validity has expired. Nodes on the network can update the table, freeing some slots that can be occupied by the nodes sitting in the queue.

By providing proof of their available resource to the network, nodes can spontaneously candidate to become worker: they join the work queue. Every node on the network can store such proof alongside the node identifier in dynamic hash table ( $DHT_q$ ). As nodes leave the pool of workers, nodes from the queue can join it. Assuming the list of identifiers sorted chronologically in  $DHT_q$ , a logic can be implemented such that nodes with identifiers at the top of the list are the first ones to access the pool of workers. However, such approach may facilitate

Sybil-identity attack scenarios, however expensive, if an entity controls a large number nodes at the top of the work queue (at least equal to half the pool of worker size  $N/2$ ) and frequently adds many nodes to the work queue such that the size of the work queue pool is large enough to create an impression of a large demand for work.

The peer identification protocol on Catalyst network is designed to limit the number nodes on the same IP address entering a pool of nodes. In particular the Ipv4 address space is small and limits the ability of an entity to spin up multiple nodes on different IP addresses. There is an exponential cost associated to get the number of distinct Ipv4 addresses needed to have enough nodes in the work queue. We nevertheless consider an alternative approach to define the dynamic of nodes leaving the work queue and joining the pool of workers that both prevents Sybil attack and incentivise nodes to join the work queue during periods of low demand for work. We define a ranking (or score) associated to nodes in the work queue such that the work queue is not concurrent. The ranking is chronological but instead depends on the volume or flux of nodes joining the queue during same allotted time period. Nodes wanting to join the work queue during a small window of time register to a tertiary queue. At the end of the time window, a fixed and limited number of nodes from the tertiary queue are randomly selected. These nodes are given a ranking drawn from a normal distribution centred around the current lower rank of the work queue, which means that some selected nodes may obtain a ranking higher than nodes currently at the bottom of the work queue. The rest of the nodes in the tertiary queue are given a lower ranking drawn from a normal distribution centred around a lower rank value shifted from the current lower rank of the work queue, the shift being inversely proportional to the volume of nodes in the tertiary queue. The rotation rate of nodes in the pool of workers then depends on the size of the work queue but only take into account the number of nodes with a ranking higher than a given threshold.

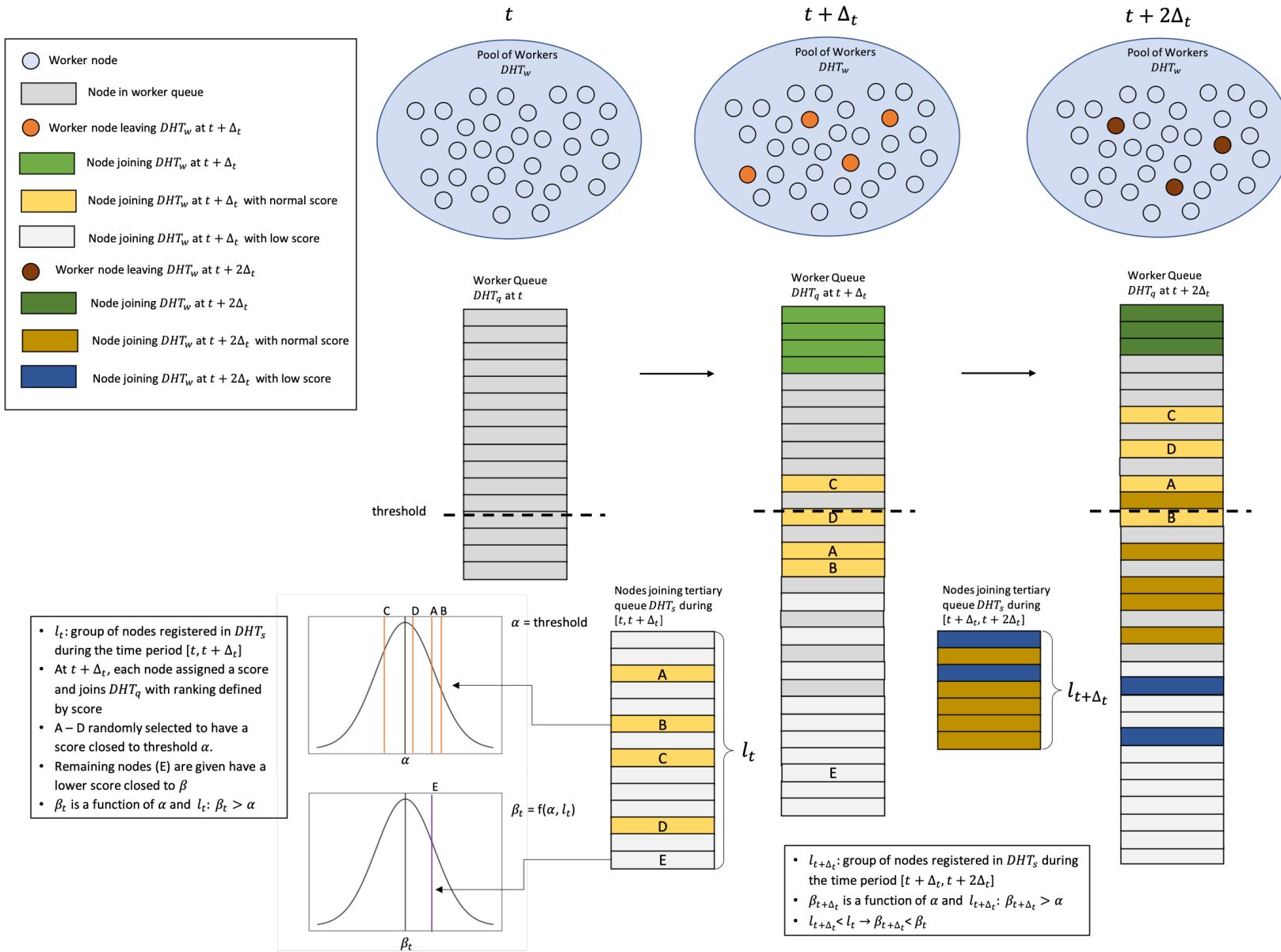


Figure 2.1: Illustration of the process followed by Catalyst network to add nodes to the work queue.



# Chapter 3

## Catalyst Distributed Ledger

The Catalyst database is designed to ensure it can run on low resource devices and fit the different needs of the network users without compromising on data integrity or accessibility.

### 3.1 Ledger Database Architecture

Catalyst has a multi-level data architecture, as illustrated in Figure 3.1.

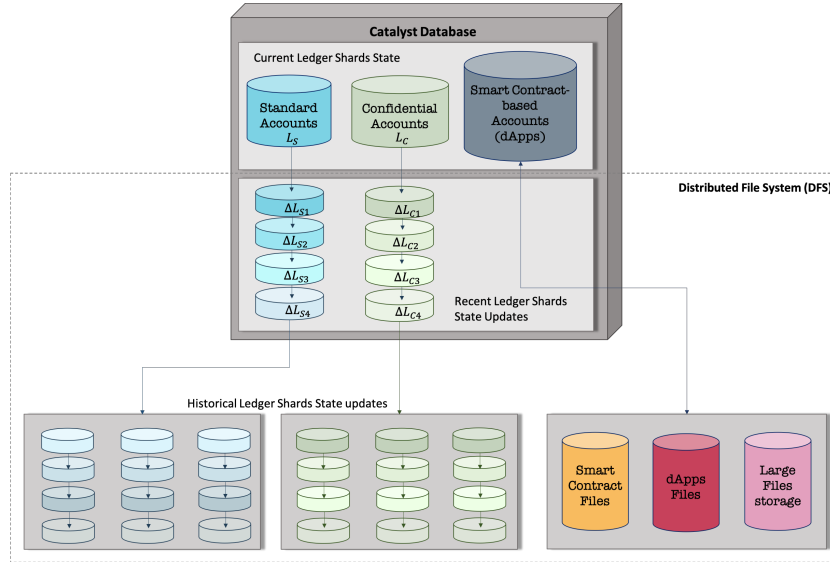


Figure 3.1: Illustration of Catalyst database architecture

At the top level lies the current state of the ledger (CS), *i.e.* the database containing the current balance of digital accounts recorded on the ledger. The CS represents a snapshot of the ledger state, at the present time. It is periodically updated. At the end of a ledger cycle, that lasts for a fixed period of one minute, a ledger state update is generated by a pool of nodes selected to manage the ledger database and distributed to the network users who can then update their local copy of the ledger state. The process followed by nodes to generate a ledger update, *i.e.* the consensus-based protocol, is discussed in chapter 5.

The middle level comprises the recent ledger state updates, that is a set of the last recent ledger state updates accepted by and broadcast across the network. Historical data, or old ledger state updates, represent the bottom level. Both middle and bottom levels are maintained by Catalyst Distributed File System (DFS) module. The top and middle levels sit on every node on the network and are thus immediately accessible. On the other hand the bottom

level is maintained by some but not necessarily all nodes in the network. Long term data is thus available with a short delay which constitutes a small trade-off for a compact ledger database maintained by every node.

Figure 3.1 also illustrates how the smart contracts and dApps are stored on a database unit separate from the account balances and communicate with the DFS for the access, production and storage of files.

## 3.2 Accounts on Catalyst

Different types of accounts are supported on Catalyst ledger. Namely:

- Non-confidential user-based accounts, with a balance in KAT tokens that is updated via the validation of non-confidential transactions. The account balance is visible to all.
- Confidential user-based accounts, with a balance in KAT tokens that is updated through the validation of confidential transactions. The account balance is hidden, only known to the account holder.
- Smart contract-based accounts. A smart contract-based account has an associated code that can be triggered by transactions or messages generated by other code.

An account is defined by the following components:

- An address component: a 32-byte address  $A_i$ , which is derived from a public/private key pair  $\{Q_i, k_i\}$  (where  $Q_i = k_i G$ ) using a collision-resistant hash function ( $\mathcal{H}$ ):  $A_i = \mathcal{H}(Q_i)$ .
- An amount component: when non-confidential, the amount is a 8-byte number  $v_i \in \mathbb{Z}_M$  (with  $M$  the maximum number of tokens defined on Catalyst system). This represents the account  $A_i$  balance in KAT tokens. When confidential the amount component is a 32-byte Pedersen Commitment [9]  $C_i = v_i H + b_i G$  that hides the balance of the account using a blinding factor  $b_i \in \mathbb{Z}_p$ .
- A data component: reserved to smart contract-based accounts and is used to store data that amounts to a maximum of  $x$  bytes.

As such, Catalyst ledger state is naturally split into sub-states, or shards, where each shard stores accounts of a given type. A node on Catalyst network may not maintain a copy of all shards but must remain aware of the possible dependencies among shards (see chapter 6).

## 3.3 Ledger Current State and DFS

The ledger state thus encompasses different sub-states, each of which keep the balance of accounts of a specific type up-to-date. The current state of the ledger state lists the accounts balance at the present time, allowing anyone to access (and comprehend in the case of non-confidential accounts) the available balance in tokens of an account.

When users on the network wish to transfer tokens to other users, they issue transactions that are broadcast to the network. The structure of these transactions are discussed in chapter 4. The transactions are collected by nodes assigned to the management of the ledger database (as

detailed in section 2.4) and used to generate a ledger state update. The production of valid ledger state updates in a trust-less environment is discussed in chapter 5. The ledger state update is a cryptographically secure structured data object that allows users to update their local copy of the ledger.

The ledger state update consists in a summary of the token transfers embedded in the transactions broadcast by the users. Transactions broadcast during a ledger cycle are collected by nodes who then use these to generate a ledger state update during the next ledger cycle. In layman's terms, the ledger state update can be viewed as a structured database with a series of rows, each row having two components: a public key referring to the address of an account stored on the ledger and an amount (positive or negative) that represents a token transfer.

Assume for instance that Alice transfers 5 KAT tokens to Bob. The transfer is represented by a transaction with two entries. The ledger state update would include two rows: one with Alice's address and a negative amount  $-5 \text{ KAT}$  and one with Bob's address and a positive amount  $5 \text{ KAT}$  (transaction fees, discussed in section 4.2 are ignored here). Once a user receives a valid ledger state update, the former can use the latter to update their local copy of the ledger: Alice's account is debited of 5 tokens while Bob's account is credited of 5 tokens. Note that the ledger state update produced for one ledger cycle only includes balance changes of accounts called in the transactions broadcast on the network during the precedent ledger cycle. This allows for a compact ledger state update as there may be many more accounts stored on the ledger that are not used during a ledger cycle.

Transactions in the context of DLT refer to data objects created and cryptographically signed by users and propagated as messages on the peer-to-peer network. A transaction typically includes:

- a set of inputs where each input comprises the details of the account or digital address being debited, the amount being debited and the signature of the account owner, proving the legitimacy or ownership of the tokens as well as the valid balance of the debited account.
- a set of outputs where each output comprises the details of the account being credited. Rather than a signature, a locking program is attached to the output, that effectively locks the tokens sent to this output using the public key of the recipient (the user holding the account being credited).

A digital signature associated to a transaction input is a mathematical scheme that allows the owner of the associated private key to prove that they have authorised the spending of the funds locked in the output of a transaction stored on the ledger. A valid signature further guarantees the non-repudiation of the message (the sender cannot deny having sent the message) and the message integrity (the latter has not and cannot be tampered with).

In the Bitcoin blockchain, valid blocks of transactions get appended to the blockchain in such a way that any block is cryptographically sealed and linked to the previous valid block. A block contains a set of transactions that transfers digital assets from one digital address to another, as well as an extra coinbase transaction that rewards the miner who successfully produced that block with new digital coins. Each transaction input contained in a valid block (except the coinbase transaction input) refers to the output of a transaction stored on a previous block (a.k.a unspent transaction output – UTXO). It can actually be viewed as the second state of that output. First the output is unspent and locked, stored on a valid block. Secondly, the

output is used as input in a new transaction and unlocked by the owner of the unlocking key. Eventually an old block in the blockchain will solely contain spent transaction outputs used as inputs in transactions stored in other later blocks. As such the old block becomes obsolete as it no longer holds any spendable tokens.

The Catalyst ledger operates differently in the sense that it does not have UTXO. The ledger state comprises digital accounts of which the balance changes over time as transactions debiting or crediting these accounts are validated on Catalyst network. Once a ledger state update is used to update the ledger database of accounts, the ledger state update is stored on DFS and can be accessed by any node as it is either locally stored or accessible upon request to special nodes storing historical data (bottom level). User nodes need not access to old ledger state update to be able to transfer tokens from their account stored on Catalyst ledger. They only need a local copy of the current ledger state. DFS is built upon the IPFS protocol which is not novel. It is nevertheless used to store files as well as historical ledger state updates. This removes the burden on user nodes to maintain the full history of the ledger database while allowing for fast retrieval of files as well as old ledger state updates. DFS is maintained by all nodes on the network. However, DFS is made of a multitude of compartments and each node needn't hold all compartments.

The design of a ledger compartment dedicated to the storage of files and historical ledger state update (DFS) is an approach taken to prevent bloating of the ledger and allow the network to hold services a scale. Indeed, this approach allows Catalyst ledger to remain lean and cryptographically secure. As detailed in chapter 4, the removal of UTXO is made possible via a new structure of transaction objects in which any token transfer (wherever spending or receiving tokens) is signed and thus authorised by the relevant participants involved in said transfer.



# Chapter 4

## Transactions

Transactions are the integral element to any blockchain or DLT. These are messages broadcast by users on the network that encompass the transfer of KAT tokens and data to and from a digital accounts stored on the ledger. Catalyst network strives to offer users a variety of services accessible on the network and as such support a plurality of transaction types. This includes the choice of opting for hidden or visible accounts via the support of both non-confidential and confidential transactions, thus offering different levels of anonymity to the network users. This section describes how different transaction structures and thus types are supported on Catalyst and details the processes behind the generation and verification of transaction signatures.

### 4.1 Transaction Types

On the Catalyst ledger, a transaction is a message or object used to transfer tokens or data from and to a set of digital accounts. Such transaction can include different types of transfer depending on the nature of the accounts embedded in said transaction. As mentioned in chapter 3.2, Catalyst supports the transfer of confidential and non-confidential assets. Catalyst also supports the transfer of assets and data linked to smart contracts and data storage. These different type of transfers are defined by specific transaction components that allow any node on the network to differentiate between the nature of exchanges embedded in different transactions. In this section we give an overview of the transaction structure and the different components considered for each type of token and data exchange.

### 4.2 Transaction Structure

In traditional blockchains (such as Bitcoin) a transaction is composed of a set of inputs and outputs. An input refers to the output of a transaction stored on a valid block of the blockchain, effectively spending that output (also referred as UTXO). In broad terms, an input thus spends tokens, while an output receives some. The output is locked and can later be spent in an input of a future transaction. On the Catalyst ledger, we opt for a new terminology, defining as *transaction entry* a transaction component that spends or receives tokens.

A transaction object on Catalyst is made of the following components:

- A transaction type specifying the type of exchange embedded in the transaction entries (non-confidential or confidential asset transfer, data storage request and retrieve, smart contracts-related token and/or data transfer).

- A timestamp corresponding to the point in time the transaction is complete and ready to be broadcast on the network.
- A set of  $n$  transaction entries  $\{E_i\}_{i=1,\dots,n}$ . Transaction entries are specific to the nature of the token and data exchange. These are described in 4.3.
- An aggregated signature  $T$  proving ownership of the set of accounts called in the transaction entries.
- A locking time corresponding to a point in time after which the transaction can be processed by a pool of workers.
- the transaction fees paid the transaction participants
- A data field that can contain up to  $x$  Bytes of data transferred in data storage or smart contract-related transactions.

Any valid transaction must contain a type, a timestamp and locking time (the later is set to 0 is there is no waiting period prior to processing a token exchange embedded in the transaction), a list of transaction entries and an associated signature. Additional fields can be included to the transaction object, depending on the nature of the token exchange.

### 4.3 Transaction Entries

Transaction entries are used on the Catalyst ledger to represent some change to the account associated with the entry. This generally takes the form of debit or credit of an account. We use the term entry to replace the traditional input and output, as on Catalyst there is no differentiation as to how debit or credit of an account is formed. Whether spending or receiving tokens, a user must sign their transaction entry and a transaction is complete if and only if all transaction entries have been signed.

A transaction entry typically consists of three components: a public key, from which the address of an account stored on the ledger is derived, an amount component that can be a number or a more complex structure and represents the amount of tokens spent from or transferred to the address, and a fee component that is a number representing the amount of tokens paid as transaction fees. On Catalyst, the number or amount of tokens can be positive (when receiving) or negative (when spending). This choice allows for a) keeping a simple transaction entry structure (there is no need for an extra field to specify the type of transfer embedded in an entry) and b), in the case of confidential asset transfer, an improved anonymity as an observer will be unable to differentiate between a sender and a recipient of a token transfer. The public key  $Q_i$  in a transaction entry  $E_i$  is always a 32-byte element from which one address  $A_i$  stored on the ledger can be derived.

The amount component of an entry however differs depending on the nature of the token exchange:

- for non-confidential asset transfer, it is a 8-byte (positive or negative) number  $v_i^t$  that represents the number of tokens spent from or transferred to the address  $A_i$ , communicated in clear text.
- for confidential asset transfer, it is made of two elements: a 32-byte Pedersen commitment  $C_i^t$  that represents the commitment of tokens spent from or transferred to the address  $A_i$  and a range proof  $\Pi_i(C_i')$  (as discussed in section 1.3) that prove that the token balance

of  $A_i$  remains within an acceptable range of value (typically greater than 0 and smaller than the total number  $M$  of tokens in circulation) after the transaction has taken place. The construction of these elements is discussed below.

Assume that the balance of the account  $A_i$ ,  $v_i$  is initially represented on the ledger by the PC  $C_i = (v_i \bmod l_H)H + (b_i \bmod l_G)G$  where  $b_i$  is a blinding factor chosen by the account holder  $U_i$ . The latter wishes to transfer or receive  $a_i$  tokens. To obfuscate the amount of tokens transferred in the transaction entry  $E_i$ , the account holder creates a PC  $C_i^t = (v_i^t \bmod l_H)H + (b_i^t \bmod l_G)G$  where  $v_i^t = a_i$  if receiving the tokens ( $a_i > 0$ ) and  $v_i^t = l_H + a_i$  if spending the tokens ( $a_i < 0$ ). Given the properties of the Elliptic curve,  $l_H$  is much greater than  $M$  in the second case. As a result, it would not be possible to construct a valid range proof on  $v_i^t$ .  $U_i$  thus creates a second PC,  $C_i' = (v_i' \bmod l_H)H + (b_i' \bmod l_G)G$  defined as  $C_i' = C_i + C_i^t$ . As a result,  $v_i' = v_i + v_i^t$  and  $b_i' = b_i + b_i^t$ . It represents the commitment of the account balance after the transaction has taken place. If  $a_i < 0$ ,  $v_i'H = [v_i + (l_H + a_i) \bmod l_H]H = [v_i + a_i]H$ . It is only possible to generate a valid range proof associated to  $C_i'$  if  $v_i > a_i$ . Note that  $v_i$  is necessarily smaller than  $M$  as the balance of the account would have been determined by a previous transaction entry, itself including a range proof that  $v_i \in [0, M]$ . As discussed in section 1.3, a range proof generated using the bulletproof protocol amounts to 672 Bytes.

Table 4.1 summarises the different components of a transaction and their respective size for the two types of transfer aforementioned.

Transaction component			Size
Type			4 Bytes
Timestamp			4 Bytes
Entries ( $n > 1$ )	non-confidential entry	32-byte public keys	$n \cdot 40$ Bytes
		8-byte amounts	
	or confidential entry	32-byte public keys	$n \cdot 736$ Bytes
		32-byte PCs	
		672-byte range proofs	
Aggregated Signature			64 Bytes
Transaction fees			8 Bytes
Locking Time			4 Bytes

Table 4.1: Structure of transactions on Catalyst and size per transaction components.

Each entry in a transaction needs to be signed to authorise the transfer of tokens. This can be achieved through the use of an aggregated signature scheme as described in section 4.4.

## 4.4 Transaction Signature

On the Catalyst ledger, all the transaction entries are signed to authorise the transfer of tokens which means that all the participants in the transaction need to sign their respective entry for a transaction to be considered complete. When signing an entry  $E_i$  a participant needs to prove ownership of the account  $A_i$  referred in said transaction entry. Said otherwise, the user needs to prove knowledge of the private key  $k_i$  paired the public key  $Q_i$  from which the account address  $A_i$  is derived. A verifier (or producer in Catalyst jargon) can verify the validity of the signature given the public key  $P_i$  specified in an entry  $E_i$ .

Signatures for transactions on the Catalyst network are formed in a highly similar way regardless of whether the asset transfer embedded in said transaction is confidential or non-confidential. The signature scheme describes in this section therefore applies to both transaction types unless explicitly stated.

In blockchains such as Bitcoin and Ethereum, transaction inputs are signed using ECDSA scheme, where the public key is recovered from the signature and used to retrieve the account or UTXO address, thus ensuring that the rightful owner of the tokens is authorised to spend these. The use of a second temporary, often called ephemeral, public/private key pair in the signature adds a layer of protection against malicious attempt to retrieve the private key of a user when signing multiple transactions spending tokens from the same address.

Public key recovery is however incompatible with batch validation, *i.e.* it is not possible to recover a set of public keys from an aggregated signature on multiple transaction inputs. As a result the choice of ECDSA-based scheme for Catalyst transaction would not be optimal as a transaction contain a minimum of two entries. A Schnorr-based signature scheme is preferred to enable user to jointly produce a signature using their private keys. A solution recently proposed by Y. Seurin *et al* [5] also accounts for protection against key-rogue attacks, preventing key malleability to create validate signatures on transaction without knowing the users' private key. We propose a Schnorr-based signature scheme inspired from this recently published work.

We define the transaction core message  $m$  as a set of  $n$  entries  $\{E_i\}_{i=1,\dots,n}$  and additional information  $X$  mentioned in the previous section (see table 4.1):

$$m = \{E_i\}_{i=1,\dots,n} + X$$

The participant  $U_i$  responsible for  $E_i$  creates the following challenge:

$$e_i = \mathcal{H}(m \parallel \tilde{Q})\mathcal{H}(L \mid Q_i) \quad (4.1)$$

Where:

- $\mathcal{H}$  is a hashing function
- $m$  is the transaction core message
- $\parallel$  denotes concatenation between strings
- $\tilde{Q}$  is the aggregated public key such that:

$$\tilde{Q} = \mathcal{H}(L \parallel Q_1) \cdot Q_1 + \dots + \mathcal{H}(L \parallel Q_n) \cdot Q_n$$

- $L$  is the hash of all the public keys used in the transaction expressed as  $L = \mathcal{H}(Q_1 \parallel \dots \parallel Q_n)$

$U_i$  then creates the following partial signature:

$$s_i = r_i + e_i \cdot k_i \quad (4.2)$$

Where  $r_i$  is a pseudo-random number chosen by  $U_i$  and kept secret. Recall that each entry  $E_i$  in a confidential transaction includes a PC obfuscating the amount  $v_i^t$  defined by:  $C_i^t = v_i^t H + b_i^t G$ . For confidential transaction we define  $r_i = b_i^t + d_i$  where  $d_i$  is a pseudo-random number chosen by  $U_i$  and kept secret. The partial signature  $s_i$  and  $R_i = r_i G$  are exchanged amongst the transaction participants. Each participant in the transaction  $U_k$  ( $k \neq i$ ) can compute:

$$R'_i = s_i \cdot G - e_i \cdot P_i \quad (4.3)$$

where  $P_i = k_i \cdot G$  and verify that  $R'_i = R_i$ , proving the validity of the partial signature.

The last participant to receive the full set of partial signatures builds the transaction signature. The signature of non-confidential transaction is composed of the pair:

$$T = (\underbrace{s_1 + \dots + s_n}_s, \underbrace{R_1 + \dots + R_n}_R) \quad (4.4)$$

At the verification phase, a producer can check that the total signature is as follows:

1. compute the list of  $n$  challenges  $\{e'_i\}_{i=1,\dots,n}$  given the public keys embedded in the transaction entries  $\{E_i\}_{i=1,\dots,n}$  included in  $m$
2. compute the quantity  $R' = s \cdot G - \sum_{i=1}^n e'_i P_i$
3. verify that  $R' = R$

If so, the signature  $T = (s, R)$  is valid. The signature is composed of a 32-byte integer and a 32-byte EC point, leading to a compact 64-byte signature for the entire transaction.

The signature of confidential transaction is defined differently. Using the public keys  $\{D_i = d_i G\}_{i=1,\dots,n}$ , it is composed of the pair:

$$T = (\underbrace{s_1 + \dots + s_n}_s, \underbrace{D_1 + \dots + D_n}_D) \quad (4.5)$$

The verifier can compute:

$$R' = \sum_{i=1}^n C_i^t - v^f H + D$$

If  $R' = s \cdot G - \sum_{i=1}^n e'_i P_i$ , the signature  $T = (s, D)$  is valid. The validity of  $T$  proves a verifier that the sum of the commitments in the transaction entries results in a commitment to 0 after subtracting the transaction fees paid by the different participant, thus ensuring that no tokens are created or lost in the transaction.

## 4.5 Transaction Validity

Nodes in the network receive and forward transactions to their peers. In order to prevent spamming attack over the network, nodes only forward transactions that are considered valid against a validity check list.

Before forwarding it to its peers, a peer verifies the transaction against the following list of criteria:

- The transaction syntax (aforementioned in 4.2) and data structure must be correct.
- The transaction size in bytes is greater than or equal to 160 Bytes (defined by the parameter MIN\_STD\_SIZE for non-confidential transaction) or 800 Bytes (defined by the parameter MIN\_CON\_SIZE for confidential transaction).
- The transaction size in bytes is less than 1 Mbyte (defined by the parameter MAX\_STD\_SIZE or MAX\_CON\_SIZE depending on the transaction type).

- The transaction list of entries must have at least two elements, each element must have a correct syntax. For non-confidential transaction  $E_i$  must have a total size of 48-byte and 3 components:
  - A public key  $Q_i$  with a corresponding account address  $A_i$  stored on the ledger where the account have a visible balance (8-byte field).
  - An amount  $v_i^t$  that once added to the balance  $v_i$  of the account mapped with the public key leaves the account balance positive ( $v_i + v_i^t > 0$ )

For confidential transaction  $E_i$  must have a total size of 742-byte and 4 components:

- A public key  $Q_i$  with a corresponding account address  $A_i$  stored on the ledger where the account has an hidden balance (32-byte field).
  - A 32-byte PC  $C_i^t$
  - A 672-byte range proof must validate against a new PC built out of the sum of  $C_i^t$  and the account balance  $C_i$ .
- The transaction fee amount  $v^f$  is greater than a (positive) minimum fee values MIN\_TX\_FEE
  - The relation  $\sum_{i=1}^n v_i^t - v^f = 0$  must be verified for non-confidential transaction.
  - The transaction signature must validate against the public key built out of the public keys stored in the transaction entries

The verification of the range proofs in the transaction is costly in computer resources, and is therefore only performed by the producers.

# Chapter 5

## Consensus Mechanism

Proof-of-work (PoW) and derivate algorithms are commonly used to manage blockchain and distributed ledger in a distributed manner. Consensus-based protocols based on such algorithms rely on a plurality of nodes, called miners, competing to generate at fairly regular interval of time a valid block of transactions to append to the blockchain. Part of the competition consists in solving a cryptographic puzzle that ensures the validity of the content of a block.

However, this competition amongst nodes wastes a tremendous amount of energy as all miner nodes expend computational power to solve the same problem, yet only the work performed by one node is used to update the blockchain. The energy consumption per year for Ethereum and Bitcoin combined is 66.6 TWh per year which is comparable to yearly energy consumption of Switzerland (61.6 TWh per year) [15]<sup>1</sup>. It is clear that this is not sustainable or environmentally friendly. Moreover, as the difficulty associated with the cryptographic puzzle increases over time, miners are forced to invest in more computer resources to have a chance of earning miner rewards. Such consensus protocols have a clear negative environmental impact and indicate counteractive economic implications with high risk of mining centralisation.

This chapter presents a new consensus-based protocol that can be applied to a peer-to-peer network in order to manage a distributed ledger in a fair and secure manner without wasting unnecessary amount of energy.

### 5.1 Background

#### 5.1.1 Motivation

The consensus algorithm designed by the engineers and researchers at Atlas City rests on the principle that every node participating in the network can contribute to the ledger state update and should be rewarded accordingly. Indeed, the consensus mechanism was conceived based on the observations that:

- In reality, not every node needs to validate every transaction for a network to be secure and a ledger fully decentralised.

---

<sup>1</sup>This energy consumption allows approximately 445 million transactions for Bitcoin and Ethereum combined per year [18][17], compared to Switzerland where 820 million debit card transactions are processed per year [16] for an estimated energy consumption of 0.001358 TWh.

- Collectively across a network of nodes there is significant distributed computer resources to build a trusted ledger. Network performance should as a result improve as the network scales up.

The PoW algorithm was introduced to solve the General Byzantine Problem among participants in the peer-to-peer network, allowing them to reach consensus without trusting one another [1]. In the PoW algorithm or any derivatives, mining nodes collect and validate all transactions broadcast to the network and form a block with these new transactions. The miners compete to solve a computationally hard problem, the solution of which is used to prove that a block is valid and can therefore be appended to the blockchain. The level of difficulty attached to the cryptographic problem solved by the miners is set by the network to ensure that blocks are produced on a regular time interval (roughly 10 minutes in the case of Bitcoin, and approximatively 17s for Ethereum). Under this scenario, one mining node is rewarded for producing the correct next block of the blockchain (which in the analogy of Catalyst corresponds to the last valid ledger state update). Although the solution to the cryptographic puzzle is hard to find, it is very easy to verify which allow for a fast and secure update of the blockchain.

While this approach provides a secure way to maintain a distributed ledger, it leads to a tremendous amount of wasted computational and electrical energy with high risk of mining centralisation. In the example of Bitcoin, the early blocks were mined by individuals with modest computer resource.

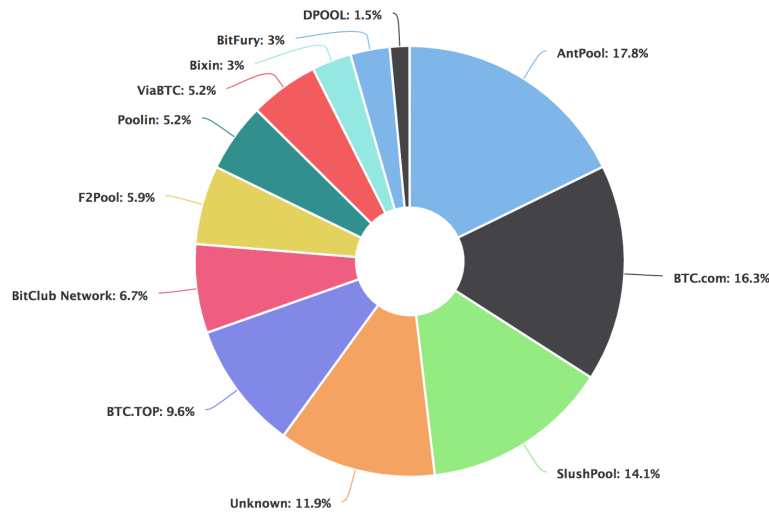


Figure 5.1: Distribution of the Bitcoin hashrate power over a 24h period, as the 30 of October 2018[2]

As illustrated in Figure 5.1, the situation is rather different nowadays. Few miners work independently (represented as the “unknown” 11.9%) while the remaining join mining pools such as Slush Pool (which was the first mining pool created for Bitcoin mining) to share their computer resources and the collected rewards, usually against the payment of a fee (2% of the mining reward with Slush Pool). Some mining pools are private pool, such as BTC.top. It also worth noting that around 80% of the mining pools are located in China [3] where the electricity is considerably cheaper than in other parts of the world.

A popular alternative to the PoW algorithm currently considered by several blockchain projects is the Proof-of-Stake algorithm (PoS). This approach addresses the footprint concerns from the



former by assigning the task of producing the next valid block to a subset of miners. The miner nodes can be selected randomly or based on criteria such as the miner's wealth (stake). The main concern with a PoS-based consensus mechanism is the risk for centralisation of wealth and subsequently the network management, with the mining work inevitably distributed to a few wealthy nodes [4].

### *Unity is strength*

Catalyst consensus mechanism is not based on a competitive process. Instead, the nodes in the network collaborate to collectively build the correct update of the ledger state. At the end of a ledger cycle, new tokens are injected into the system and all the nodes that contributed to producing the correct ledger state update receive a share of that reward.

## 5.1.2 Naming Convention

Nodes who contribute to maintaining the ledger state are called producers, rather than miners. Indeed, producers do not solve a computationally hard problem, but instead validate the transactions broadcast to the network and use these to collaboratively build (produce) a ledger state update.

In this chapter we assume that the ledger is composed of one single shard comprising a fixed set of accounts. The implementation of sharding techniques are discussed in chapter ???. We therefore consider one single pool of workers and one subset of producer nodes selected per ledger cycle.

A ledger cycle  $C_n$  starts at time  $t = t_{n,0}$  and lasts for a period of time  $\Delta t_{cycle}$ , therefore ending at  $t = t_{n,0} + \Delta t_{cycle}$ .  $P$  producers  $\{P_j\}_{j \in P}$  are selected to build the ledger state update during the ledger cycle  $C_n$ . Each producer  $P_j$  can be identified by its peers as well as the rest of the network via its unique identifier  $Id_j$ .

During  $C_n$  the  $P$  producers collaborate to create a ledger state update  $\Delta L_n$  based on the set of  $m_{n-1}$  transactions broadcast on the network during the previous ledger cycle  $C_{n-1}$ . To limit discrepancies in the set of transactions collected by the different producers and processed during  $C_n$  a small time window  $\Delta t_{freeze}$  is considered. The  $m_{n-1}$  transactions  $\{Tx_j\}_{j=1,\dots,m_{n-1}}$  are actually collected during the period of time  $[t_{0,n-1} - \Delta t_{freeze}, t_{n,0}]$  ( $t_{n-1,0} = t_{n,0} - \Delta t_{cycle}$ ) and must have a timestamp comprised between  $t_{0,n-1} - \Delta t_{freeze}$  and  $t_{n,0} - \Delta t_{freeze}$ .

Each producer compiles a ledger state update and interacts with its peers to vote on the most popular ledger state update produced by the set of producers. Each producer is thus tasked with two responsibilities: compiling a local ledger state update and voting on the correct (most popular) ledger state update. Each task entitles the producer to receive part of the reward allocated to producers for maintaining the ledger state. The amount of reward individually collected depends on the quality of work performed by a producer. During a ledger cycle  $C_n$  two lists of producer identifiers are created,  $\mathcal{L}_n(prod)$  and  $\mathcal{L}_n(vote)$ . The first one lists the identifiers of producers who correctly built the ledger state update while the second one lists the identifiers of producers who correctly voted on the correct ledger state update built by the producers included in the first list.

Throughout the different phases of the ledger cycle, the producers exchange quantities that are hashes, notably of ledger state updates, (using the *Blake – 2b* hashing function) to which they append their identifiers. The exchange of hashes allows for fast and efficient communication

rounds amongst the peers as these are smaller pieces of data than the actual ledger state updates.

The ledger cycle is composed of three phases. Namely, a computation phase during which the producers locally generate a ledger state update, a voting phase during which the producers reach consensus on the correct set of changes to distribute to the rest of the network and a synchronisation phase during which the correct update including the reward to producers is distributed and the ledger state is updated by all nodes in the network.

## 5.2 Democratic Consensus

Section 2.5 describes how user nodes register to become worker nodes and can be selected from the pool of workers to become a producer for a ledger cycle. This section describes the work performed by producer nodes in order to maintain the ledger state. The work performed by producers in order to generate a ledger state update  $\Delta L_n$  for the ledger cycle  $C_n$  starts at  $t = t_{n,0}$  and last for a period of time  $\Delta t_{cycle}$ . At the end of the ledger cycle, nodes in the network use  $\Delta L_n$  to update their local copy of the ledger state. This section describes the different phases of a ledger cycle.

### 5.2.1 Computation Phase

During the first phase (*a.k.a* computation phase) of the ledger cycle  $C_n$ , a producer  $P_j \forall j \in P$  creates a local ledger state update and exchanges it with its peers.

The first phase starts at  $t = t_c = t_{n,0}$  and lasts for a period of time  $\Delta t_c$ , therefore ending at  $t_c + \Delta t_c$ .

#### Local ledger state update generation

At  $t = t_c$ , the producer  $P_j$  flushes its mempool from the  $m_{n-1}$  transactions  $\{Tx_i\}_{i=1,\dots,m_{n-1}}$  collected during the period of time  $[t_{n-1,0} - \Delta t_{freeze}, t_{n,0}]$  and uses these transactions to create a local ledger state update  $\Delta L_{n,j}$ . The production of  $\Delta L_{n,j}$  lasts for a period of time  $[t_c, t_c + \Delta t_{c0}]$  ( $\Delta t_{c0} < \Delta t_c$ ). The producer uses a salt  $s$ , defined using a pseudo-random number generator that takes for seed the Merkle tree of the previous valid ledger state update  $\Delta L_{n-1}$ . The producer also creates a new hash tree  $d_n$ , to store the aggregated signature embedded in each of the  $m_{n-1}$  transactions.  $P_j$  then follows a series of steps:

1. For each transaction  $Tx_i \forall i \in [1, m_{n-1}]$ ,  $P_j$  verifies that the transaction is valid (see section 4.5) and if so, extracts the  $n^i$  transaction entries (described in 4.3)  $\{E_\alpha\}_{\alpha=1,\dots,n_i}$ , included in  $Tx_i$ . The producer also extracts the transaction signature and adds it to the hash tree  $d_n$ . Note that the transactions signature in  $d_n$  are sorted in alphabetic order, as to ensure that two same sets of transaction signature result in the same hash tree.
2. For each transaction entry  $E_\alpha$ ,  $P_j$  creates a corresponding hash variable:

$$O_\alpha = \mathcal{H}[E_\alpha \parallel s]$$

Each pair  $(E_\alpha, O_\alpha)$  is added to a list  $L_E^s$ . Steps (1) and (2) are repeated until all transactions have been processed.

3.  $P_j$  then creates a new list  $L_E^f$  using the  $M = \sum_{i=1}^{m_n-1} n_i$  transaction entries listed in  $L_E^s$  (assuming all transactions are valid) such that the transaction entries in  $L_E^f = \{E_\beta\}_{\beta=1,\dots,M}$  are sorted following a lexicographical order based on their associated hash variable:  $O_1 < O_2 < \dots < O_\beta < \dots < O_M$ . This approach blurs the links between the token flows embedded in the transactions for a better anonymity of the users involved in said transactions.
4.  $P_j$  also extracts the transaction fees  $v_i^f$  paid in each transaction  $Tx_i$  and creates the following sum:

$$x_f = \sum_{i=1}^{m_n-1} v_i^f$$

5.  $P_j$  computes the local ledger state update as the transaction entry list  $L_E^f$  concatenated with the hash tree of the transactions signature  $d_n$ :

$$\Delta L_{n,j} = L_E^f \parallel d_n$$

The producer then computes a second quantity that represents its individual contribution:

$$h_j = h_{\Delta j} \parallel Id_j$$

with  $h_{\Delta j}$  referred as the producer *local hash*:

$$h_{\Delta j} = \mathcal{H}(\Delta L_{n,j})$$

$h_j$  includes the producer unique identifier  $Id_j$  (described in 2.1), used to verify that  $P_j$  is a producer node selected for the ledger cycle and later evaluate the quality of work performed by  $P_j$ .

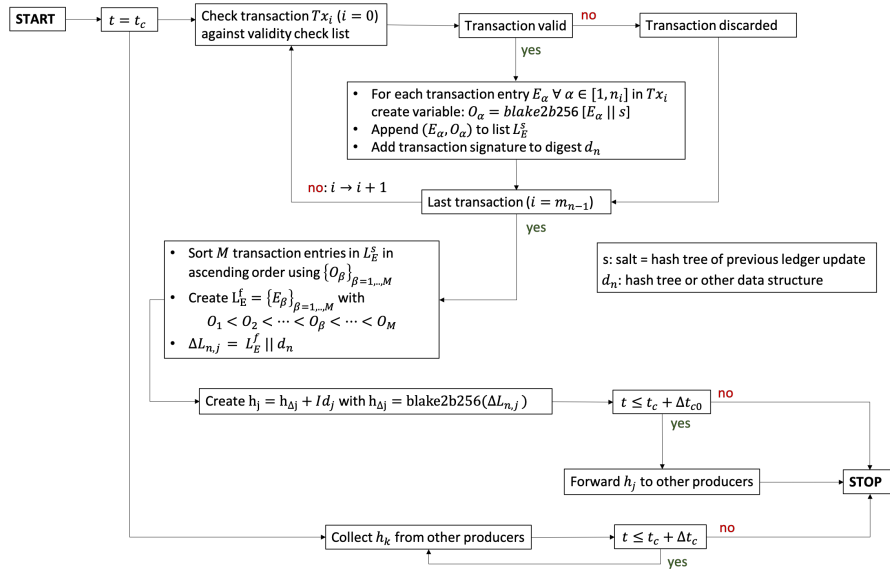


Figure 5.2: Flowchart illustrated the steps followed by a producer  $P_j$  node during a period of time  $\Delta t_c$  leading to the broadcast of a local hash  $h_j$ .

6. At  $t \leq t_c + \Delta t_{c0}$ ,  $P_j$  broadcasts  $h_j$  to the other producers in the network. Figure 5.2 describes the process followed by  $P_j$  to produce and broadcast  $h_j$ .

### Local ledger state updates collection

During the first ledger cycle phase,  $P_j$  collects other  $\{h_k\}_{k \in P/j}$  contributions generated by its producer peers  $\{P_k\}_{k \in P/j}$  in its mempool.

At the end of the computation phase ( $t = t_c + \Delta t_c$ ),  $K_j$  local hashes are stored in  $P_j$ 's mempool (including the local hash producer by  $P_j$ ). Given the set of  $P$  producers selected for the ledger cycle  $C_n$ , the producer  $P_j$  collects at most  $P - 1$  contributions (*e.g.*  $K_j = P$ ) with each contribution made of a local hash and a unique identifier. In an ideal world, two producers  $P_j$  and  $P_k$  would use the same set of transactions and as a result compute the same quantity ledger state update, leading to  $\Delta L_{n,j} = \Delta L_{n,k}$ . In practise, a producer may not collect exactly  $P$  local hashes during  $\Delta t_c$  (*e.g.*  $K_j \leq P$ ) and may not process the exact same set of transactions as its peers. The following steps describe how each producer can verify that a ledger state update has been generated by a majority of producers and generate the reward allocated to the producers.

### 5.2.2 Voting Phase

During the second phase (*a.k.a* voting phase) of a ledger cycle, a producer  $P_j$  votes on the correct ledger state update. At the end of the process, producers forward their proposed ledger state update to their peers.

The second phase starts at  $t = t_v$  where  $t_v = t_c + \Delta t_c$  and lasts for a period of time  $\Delta t_v$ , therefore ending at  $t_v + \Delta t_v$ .

#### Local vote

Using the  $K_j$  contributions stored in its mempool,  $P_j$  follows a series of steps during a period of time  $\Delta t_{v0}$  ( $\Delta t_{v0} < \Delta t_v$ ):

1.  $P_j$  verifies that the same local hash  $h_{\Delta j}^{maj}$  is embedded in a majority  $K^{maj}$  of contributions, where  $h_{\Delta j}^{maj} = \max[\text{unique}(h_{\Delta k}) \mid \forall k \in \{K_j\}]$  and  $K^{maj} = \text{count}[(h_{\Delta k} = h_{\Delta j}^{maj}) \mid \forall k \in \{K_j\}]$ . The threshold,  $K_{threshold}$ , to decide if a majority of producers agrees on the same ledger state update, should be strictly greater than 50%, due to statistical consideration. The relevant variables for a producer to decide if the same ledger state update is found by a majority of producers are  $K_{min}$  and  $K^{maj}$ . Threshold considerations are discussed in detail in chapter 8.

If  $K_j > K_{min}$  and  $K^{maj} > K_{threshold}$  :

- $P_j$  creates a list  $\mathcal{L}_j(prod)$  and appends to said list the identifiers of any producer  $P_k$  who forwarded a contribution  $h_k$  satisfying  $h_{\Delta k} = h_{\Delta j}^{maj}$ .  
If  $h_{\Delta j} = h_{\Delta j}^{maj}$ ,  $P_j$  also appends its identifier to the list  $\mathcal{L}_j(prod)$ .
- The producer  $P_j$  then computes the following quantity:

$$v_j = h_{\Delta j}^{maj} \parallel \#(\mathcal{L}_j(prod)) \parallel Id_j$$

Where  $\#$  represents a hash tree or some other compressed data structure of the list  $\mathcal{L}_j(prod)$ .  $v_j$  corresponds to  $P_j$ 's vote on the most popular ledger state update. A hash tree of a list is useful to quickly verify that an object (an identifier) is included in the list.  $\#(\mathcal{L}_j(prod))$  is a witness of the list of producers who correctly generated the most popular ledger state update according to  $P_j$ .

- At  $t \leq t_v + \Delta t_{v0}$ ,  $P_j$  broadcasts its vote  $v_j$  to the other producers in the network. Figure 5.3 displays a flowchart describing the steps followed by  $P_j$  to create and broadcast  $v_j$ .

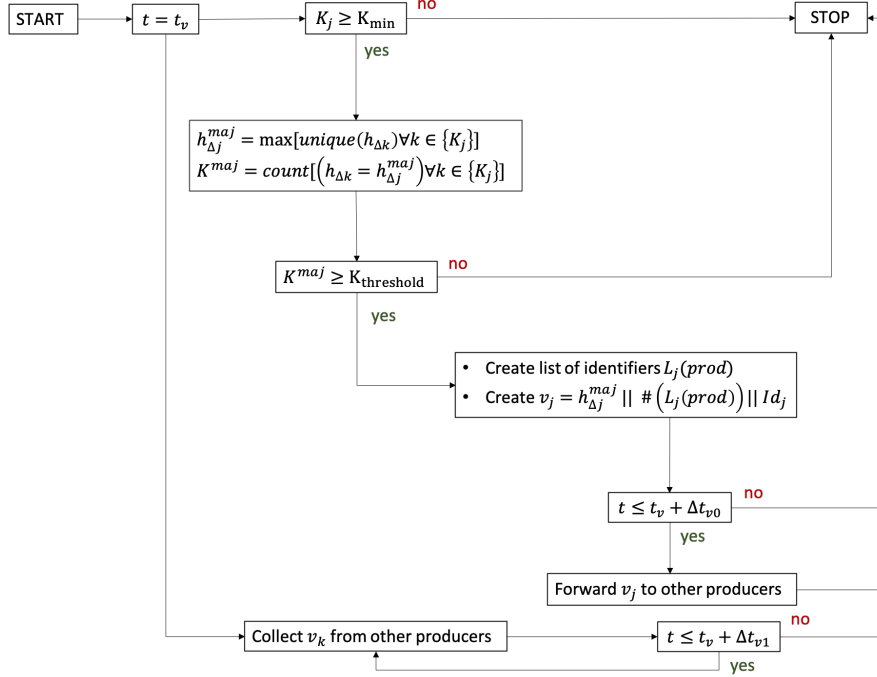


Figure 5.3: Flowchart illustrating the series of steps followed by a producer  $P_j$  to issue a local vote  $v_j$ .

### Production reward allocation

Shortly after the second phase starts ( $t \approx t_v$ ),  $P_j$  starts collecting votes  $v_k$  generated by other producers  $\{P_k\}_{k \in P/j}$  in its mempool. The collection lasts for a period of time  $\Delta t_{v1}$  ( $\Delta t_{v0} < \Delta t_{v1} < \Delta t_v$ ) after which the producer holds  $V_j$  votes in its mempool ( $V_{min} \leq V_j \leq P$ ). At this point:

1.  $P_j$  verifies that the same local hash  $h^{maj}$  is embedded in a majority of votes. With  $h^{maj} = \max[\text{unique}(h_{\Delta k}^{maj}) \mid k \in \{V_j\}]$  and  $V^{maj} = \text{count}[(h_{\Delta k}^{maj} = h^{maj}) \mid k \in \{V_j\}]$ , this condition is met if  $V^{maj} > V_{threshold}$  (See chapter 8 for more explanations).
2. The producer  $P_j$  can only participate in the next phase if the local hash computed during the computation phase,  $h_{\Delta j}$ , is equal to  $h^{maj}$ . Indeed,  $P_j$  needs to have knowledge of the ledger state update of which the hash was used to vote in order to proceed.

If each producer collects the local hash generated by every producers, any two producers  $P_j$  and  $P_k$  would build the same list of identifiers  $\mathcal{L}_j(prod) = \mathcal{L}_k(prod)$ . In practise, a producer may not have collected all local hashes and as a result have an incomplete list of identifiers, yet have collected enough data to be able to issue a correct vote. We mentioned how the identifier of a producer can be appended to a local hash to verify that  $P_j$  is a producer node selected for the ledger cycle and evaluate the quality of work performed by  $P_j$ . Indeed,  $Id_j$  can be used to create and add a special transaction entry to the ledger state update, called coinbase entry, that effectively rewards the producer for its work performed during the ledger cycle. The correct (complete) list of producers who successfully built the correct (dominant) ledger state update for that cycle,  $\mathcal{L}_n(prod)$ , is used to create these new transaction entries and append these to

the final ledger state update generated for that cycle. It is therefore crucial that a majority of producers succeed in generating that list in order to generate the same final ledger state update.

This second part of the voting process thus consists in creating the final list of identifiers involved in the production of the ledger state update. As explained below the final list  $\mathcal{L}_n(prod)$  is obtained by merging the partial lists included in the producers' vote. A producer  $P_j$  could have produced a hash  $h_{\Delta_j}$  different than  $h_{\Delta_j}^{maj}$  yet added his identifier to  $\mathcal{L}_j(prod)$  in the attempt to get some token reward allocated to producers who correctly produced the next ledger state update. In such scenario  $Id_j$  would be an element of the list included in  $v_j$  (or any other producer node controlled by  $P_j$ ), but it wouldn't be included in any other list  $\{\mathcal{L}_k(prod)\} \forall k \in P/j$ . To prevent such malicious behaviour, a rule imposes that  $P_j$  only append to the final list  $\mathcal{L}_n(prod)$  the identifier of a producer included in the list  $\mathcal{L}_k(prod)$  of a vote  $v_k$  satisfying  $h_{\Delta_k}^{maj} = h_{\Delta_k}^{maj}$  if and only if that identifier is included in at least  $P/2$  lists  $\{\mathcal{L}_k(prod)\}_{k=1,\dots,V_j}$  associated to a vote  $v_k$  satisfying  $h_{\Delta_k}^{maj} = h_{\Delta_k}^{maj}$ . Although this eliminates the risk of unethical behaviour from the producer, this also means that there would be little incentive for a producer to broadcast its vote if its identifier was not included in  $\mathcal{L}_n(prod)$ . However, the probability that a producer compiles the correct final list  $\mathcal{L}_n(prod)$  strongly depends on the amount of votes collected. The more votes collected by a producer, the greater the probability that said producer will compile the complete final list. Although a producer may not have produced the correct local ledger state update, participating in the voting process is therefore an important contribution to the overall consensus protocol and should entitle the producer to some reward. To that end a producer  $P_j$  can use the identifier of other producers included in their vote and create a second list  $\mathcal{L}_j(vote)$  to account for their participation in the voting process.

At  $t = t_v + \Delta t_{v1}$ ,  $P_j$  follows a series of step for a period of time  $\Delta t_{v2}$  ( $\Delta t_{v1} + \Delta t_{v2} < \Delta t_v$ ):

1.  $P_j$  creates a new list  $\mathcal{L}_j(vote)$  and appends to said list the identifier of any producer  $P_k$  who forwarded a vote  $v_k$  satisfying  $h_{\Delta_k}^{maj} = h_{\Delta_k}^{maj}$ .
2.  $P_j$  creates the final list  $\mathcal{L}_n(prod)$  and appends to said list the identifier of a producer included in the list  $\mathcal{L}_k(prod)$  of a vote  $v_k$  satisfying  $h_{\Delta_k}^{maj} = h_{\Delta_k}^{maj}$  if and only if that identifier is included in at least  $P/2$  lists  $\{\mathcal{L}_k(prod)\}_{k=1,\dots,V_j}$  associated to a vote  $v_k$  satisfying  $h_{\Delta_k}^{maj} = h_{\Delta_k}^{maj}$ .
3.  $P_j$  then creates a list  $L_{CE}$  of coinbase entries for each producer whose identifier is included in  $\mathcal{L}_n(prod)$ . Each producer receives  $x_h$  tokens. Assume that  $Y_h \leq P$  identifiers are included in  $\mathcal{L}_n(prod)$  and  $X$  is the total number of tokens injected per cycle for the pool of  $P$  producers, the quantity  $x_h$  is defined such that  $Y_h x_h = f_{prod} X + x_f$  where  $x_f$  represents the total number of fees collected from the  $m_{n-1}$  transactions and  $f_{prod}$  represents the fraction of new tokens injected per cycle and distributed to the producers who built the correct ledger state update. The remaining  $(1 - f_{prod})X$  tokens are distributed to other contributing nodes in the network. A part of it goes to the producers who voted correctly on the previous ledger cycle update. Let  $\mathcal{L}_{n-1}(vote)$  be the list of identifier of the producers who voted correctly on the previous ledger cycle update  $C_{n-1}$ . We demonstrate later how such list is derived during a ledger cycle. For now let's assume that  $L_{CE}$  includes coinbase entries for producers involved the production of the ledger state update for this ledger cycle  $C_n$  and the producers involved in the voting process of the precedent cycle  $C_{n-1}$ .
4.  $P_j$  then creates the final ledger state update for  $C_n$  including the reward allocated to the producers for their contribution:

$$\mathbf{LSU}_j = \mathbf{L}_E^f \parallel \mathbf{d}_n \parallel \mathbf{L}_{CE}$$

$P_j$  then computes its final vote:

$$w_j = \mathcal{H}(\mathbf{LSU}_j) \parallel \#(\mathcal{L}_j(\text{vote})) \parallel Id_j$$

which includes a partial list of identifiers of producers who voted for the correct ledger state update  $h^{maj}$ .

5.  $P_j$  then forwards  $w_j$  to the other producers and collect the final votes issued by its peers. Figure 5.4 illustrates the different steps followed by  $P_j$  during the voting phase.

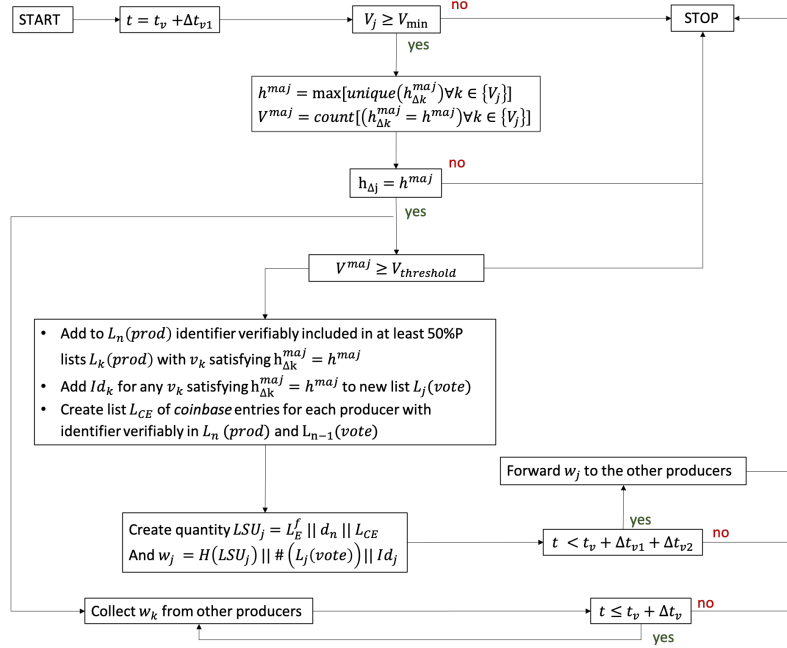


Figure 5.4: Flowchart illustrating the series of steps followed by a producer  $P_j$  during the voting phase of the ledger cycle.

At the end of the voting phase ( $t = t_v + \Delta t_v$ ), the producer  $P_j$  holds  $W_j$  final votes in its mempool with  $W_j \leq K_n$  where  $K_n \leq P$  is the actual total number of producers who correctly computed  $h^{maj}$ .

### 5.2.3 Synchronisation Phase

The third and final phase (*a.k.a* synchronisation phase) of a ledger cycle starts at  $t = t_s$ , with  $t_s = t_{n,0} + \Delta t_c + \Delta t_v$ , and lasts for a period of time  $\Delta t_s$ , therefore ending at  $t_s + \Delta t_s = t_{n,0} + \Delta t_{cycle}$ .

During a period of time  $\Delta t_{s0} < \Delta t_s$ ,  $P_j$  executes the following steps:

1.  $P_j$  defines the ledger state update  $\Delta L_n$  for the cycle  $C_n$  as:  
 $\mathcal{H}(\Delta L_n) = \max[\text{unique}(\mathcal{H}(\mathbf{LSU}_k)) \mid k \in \{W_j\}]$  and the associated number of final votes collected:  $P_n = \text{count}[(\mathcal{H}(\mathbf{LSU}_k) = \mathcal{H}(\Delta L_n)) \mid k \in \{W_j\}]$  and verifies that  $P_n > P_{threshold}$ .

2.  $P_j$  creates a new list  $\mathcal{L}_n(\text{vote})$  and appends to  $\mathcal{L}_n(\text{vote})$  the identifier of a producer included in the list  $\mathcal{L}_k(\text{vote})$  of a vote  $w_k$  satisfying  $\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n)$  if and only if the identifier is included in at least  $K_n/2$  lists  $\{\mathcal{L}_k(\text{vote})\}$  associated to a vote  $w_k$  satisfying  $\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n)$ . Note that  $K_n$  can be easily computed as it corresponds to the number of identifier of producers who correctly computed the ledger state update and are therefore included in  $\mathcal{L}_n(\text{prod})$ .
3.  $P_j$  then creates the following quantities:

$$\begin{aligned} \mathbf{H}_j &= \mathcal{H}(\Delta L_n) \parallel \text{Id}_j \\ \mathbf{H}_{j'} &= \mathcal{H}(\Delta L_n) \parallel \#(\mathcal{L}_n(\text{vote})) \parallel \text{Id}_j \end{aligned}$$

4. The producer then broadcasts  $H_j$  and  $H_{j'}$  to the entire network.  $H_j$  is destined to the user nodes while  $H_{j'}$  is sent to the workers who store  $\#(\mathcal{L}_n(\text{vote}))$  in their mempool.

During the time period  $[t_s, t_s + \Delta t_{\text{cycle}}]$ , user nodes in the network collect  $H_k \forall k \in P$  quantities from producers. Extracting the identifier  $\text{Id}_k$  in  $H_k$  a user node can easily compile a list of identical  $\mathcal{H}(\Delta L_n)$  hash values. Once a user node has collected more than  $P/2$  identical hash values, they can send a request to producers to receive  $\Delta L_n$  and verifies the validity if it against any  $H_k$  quantity previously stored. When valid, the user can safely synchronise their local copy to the ledger. The balance of accounts stored on the ledger are updated and the producers effectively collect their rewards.

Worker nodes also store the lists  $\mathcal{L}_n(\text{vote})$  embedded in each  $H_k$  quantity. If selected to be a producer for the next cycle  $C_{n+1}$ , the worker can use these to generate the reward allocated to the producers who correctly voted for the ledger state update during  $C_{n+1}$ .

Figure 5.5 summarises the different phases of the ledger cycle.

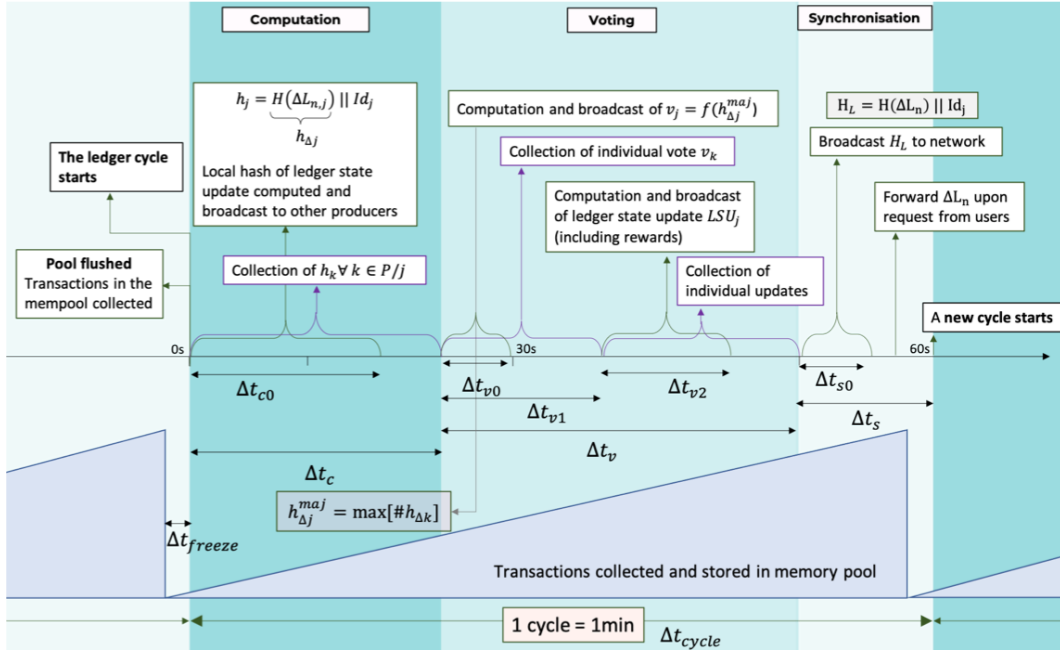


Figure 5.5: Illustration of the different phases of a ledger cycle

The various parameters and thresholds mentioned in this chapter and their impact on the network security are discussed in Chapter 8.



# Chapter 6

## Catalyst at Large Scale

Work in progress



# Chapter 7

## Benchmarking

Work in progress



# Chapter 8

## Security

Whenever financial value is stored on a distributed system, there will be greater incentive to attack the system in the attempt to take control of financial assets or simply disrupt the system to create or destroy existing assets. This is why security against attack on DLT must be of the most grave importance. With no centralised entity to control access and check validity it is up to the peers on the network to ensure its security. Through consensus and the underlying protocols of the network a secure environment must be created to allow transactions to take place in a trust-less environment.

### 8.1 Security of the Consensus Mechanism I

The primary attack of concern for all blockchain and DLT platforms is the subversion of their consensus protocol and is generally referred as a 51% attack. Such attack is made possible when an entity or group of entities collude to have enough influence on the network to produce a block or ledger state update with invalid transactions, in the attempt to alter the ledger integrity. Depending on the protocol, the influence can be in computing power or number of nodes and exceeds 50% of the relevant resource.

An attack could be performed for many reasons aside attempting to steal money from a network, including to discredit or shake trust in a network. A consequence of a successful attack would likely be to reduce token prices. Although there is no tangible proof of this, it could explain why 51% attacks are not too common. Nevertheless, it remains important to prevent and mitigate the risk of an attack as much as possible.

The probability of a 51% attack ( $P_{51}$ ) typically depends on the algorithm used to produce a valid block or ledger update. When considering PoW-based algorithms,  $P_{51}$  can be expressed as a function of the hash rate of network nodes. Since the consensus-based protocol on the Catalyst network as laid out in section 5 does not rely on solving a cryptographic puzzle, the concept of hash rate of nodes involved in the ledger state update is not relevant to quantify the probability or the cost of an attack on Catalyst network. The number of nodes involved in the production of a ledger state update is however relevant, as explained in this section.

The probability of a successful 51% attack on Catalyst network implies that a malicious entity (or group of entities) succeeds in controlling more than half the producer nodes selected to produce the ledger state update during a ledger cycle, giving that entity the power to tamper with the ledger state.

An attack can be considered successful for any value  $p \in [p_0, P]$  where  $p_0 = P/2 + 1$  which is equivalent to  $p > 50\%P$ . When  $P \approx N$ , *i.e.* the number of producers selected during a

ledger cycle is very close to the total number of nodes in the pool of workers, the absence of a randomness element in the selection of  $P$  producers makes it easy to compute the probability of a successful attack on the network:  $P_{51} \approx O/N$ . A malicious entity would know exactly when an attack can successfully be performed, that is when  $O > N/2$ .

When  $N \gg P$ , the probability  $P_{51}$  depends on the following parameters:

- $N$  : the total number of nodes in the pool of workers.
- $P$  : the subset of worker nodes selected to perform work for one ledger cycle ( $P \leq N$ ).
- $O$  : the number of malicious nodes in the pool of workers ( $0 \leq O \leq N$ ). This is a total subset of malicious nodes colluding to perform an attack on the network.
- $p$  : the number of malicious nodes in the subset  $P$  of producers. ( $0 \leq p \leq P$ ).

$P_{51}$  can there be expressed by the discrete sum:

$$P_{51} = \sum_{p=p_0}^P P_A(p) \quad (8.1)$$

where  $P_A(p)$  represents the probability of having  $p$  malicious nodes in the set  $P$ . When the ratio between the total number of nodes  $N$  and the number of nodes  $P$  is large ( $N > 20 \times P$ ) it can be expressed as follows:

$$P_A(p) = \frac{\overbrace{\binom{O}{p}}^A \overbrace{\binom{N-O}{P-p}}^B}{\underbrace{\binom{N}{P}}_C} \quad (8.2)$$

$A$  represents the number of possible combinations for choosing  $p$  nodes from  $O$  malicious nodes.  $B$  represents the number of possible combinations for choosing good (non-malicious) nodes for the remaining  $N - O$  nodes in the pool of workers. Finally,  $C$  corresponds to the number of available combinations for choosing  $P$  nodes from the pool of  $N$  nodes.

In equation 8.2,  $P_A(p)$  is the probability mass function of a hypergeometric distribution over the set of parameters  $\{N, O, P\}$ . Note that such expression is valid for  $\max(0, O + P - N) \leq p \leq \min(O, P)$ .

There are two main arguments behind having a large number of  $N$  nodes:

- To account for the fact that most nodes with sufficient resources may want to join the pool of workers and receive tokens as reward for their contribution to the ledger state management
- To make it increasingly costly for any malicious entity to control more than half the nodes.

As explained in section 2.5, prior to joining the pool of workers, nodes are part of a work queue. Nodes in the pool of workers are granted a work pass valid for finite period time. When a node  $W_j$  joins the pool of workers, it is granted a work pass defined by a decay time  $\tau_j$ . Similarly

to unstable nuclear elements the work pass has a 50% chance to remain valid after  $W_j$  sits in the pool of workers for a period of time  $\tau_j$ . The decay time is not fixed over time but instead can vary depending on the quality of work produced by  $W_j$  when selected to be a producer. The quality of work performed by a producer during a ledger cycle  $C_n$  can be estimated by verifying if the producer identifier is found in the lists  $\mathcal{L}_n(prod)$  and  $\mathcal{L}_n(vote)$ . If the identifier is found in one of the two lists or none, the decay time increases.  $\tau_j$  can also vary to account for the demand of work, *i.e.* the length of the work queue, although a threshold is considered to mitigate the risk of a malicious entity (or group of entities) trying to simulate a highest demand for work than reality.

As a result, a varying number of nodes leaves the pool of workers at each ledger cycle. Although the size of the pool of workers might be constant ( $N$  nodes), the selection of nodes actually forming the pool of worker changes over time. The mechanism to define a score for nodes in the work queue is designed to prevent malicious nodes from gaining control of a large fraction of worker nodes. Nevertheless, as we derive the probability  $P_{51}$  in this section, we must stress that the fraction  $O/N$  may change (increase or decrease) over time and should be taken into account if computing such probability over a series of ledger cycle.

When  $N \gg P$  the probability of a successful attack can therefore be estimated using the cumulative hypergeometric distribution function (CDF) for  $p \in [p_0, P]$ . In this paper, we provide probability estimate obtained using *scipy.stats* Python library. The graphs presented are obtained using *matplotlib.pyplot* library. Rather than computing the CDF, the probability measurements are obtained using the survival probability (SDF), which is the inverse of CDF but is known to provide more accurate results<sup>1</sup>.

As an example, assume a rather large number of nodes in the pool of workers,  $N = 20,000$ , out of which 5% are selected as producers for a given cycle ( $P = 1,000$ ). Further assume a ratio  $O/N = 20\%$ , *e.g.* 1 every 5 nodes in the pool of workers is controlled by a malicious entity ( $O = 4000$ ). The probability of a successful attack is calculated using the SDF of an hypergeometric distribution using these set of parameters and amounts to:  $P_{51} = 1 - SDF(20000, 4000, 1000) \approx 10^{-9}\%$ . For the same set  $(N, P)$ , the probability of a successful attack reaches 0.04% for  $S = 45\%$  of malicious nodes in the pool of workers.

Figure 8.1 shows the probability of a successful control of more than 50% of the producers as a function of the number of producers for four different pool of workers sizes and two attack scenarios: when a malicious entity controls  $O/N = 45\%$  of the worker nodes in blue, and in orange when a malicious controls  $O/N = 35\%$  of the worker nodes in blue. For  $N = 20000$ , the probability remains below  $10^{-9}$  if  $P < \approx 4000$  while for a smaller pool of worker size ( $N = 5000$ ), the ratio  $P/N$  must be at close to 50% to prevent a successful control of more than 50% of the producers.

---

<sup>1</sup>See <https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.hypergeom.html> for more details.

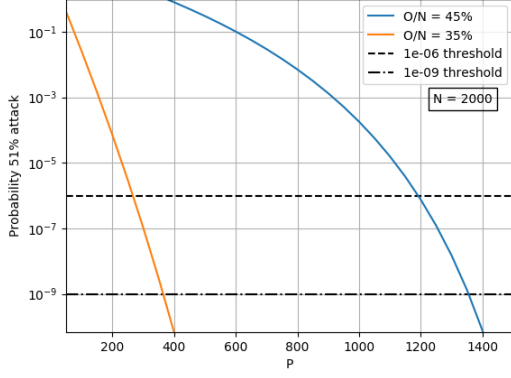
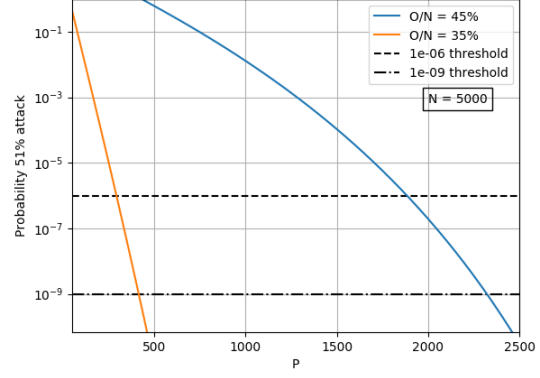
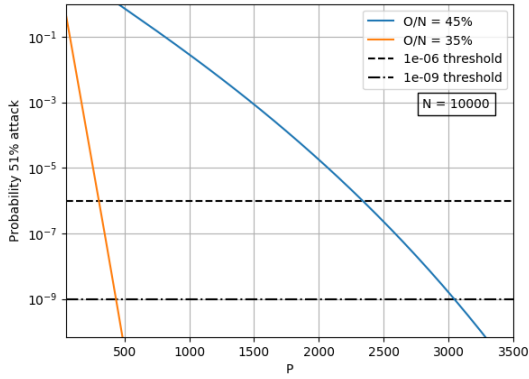
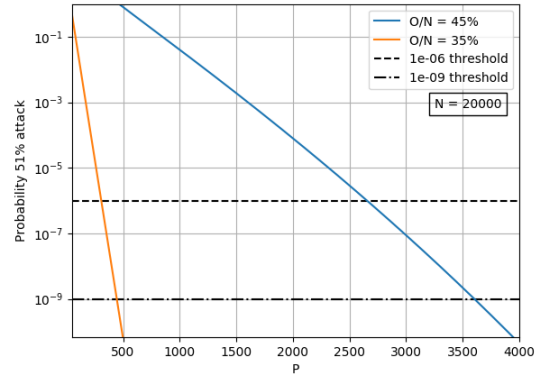
(a)  $N = 2000$ (b)  $N = 5000$ (c)  $N = 10000$ (d)  $N = 20000$ 

Figure 8.1: Probability of 51% attack as a function of  $P$  for various pool of worker size ( $N = \{2000, 5000, 10000, 20000\}$ ) when a malicious entity controls  $O/N = 45\%$  of the worker nodes in blue, and in orange when a malicious controls  $O/N = 35\%$

Figure 8.2 displays the minimum ratio  $P/N$  required to maintain a probability below  $10^{-6}$  and  $10^{-9}$  for various malicious scenario ( $O/N$  ratio between 30% and 45%). This shows that as  $N$  increases the required  $P/N$  ratio required for the same security level decreases.

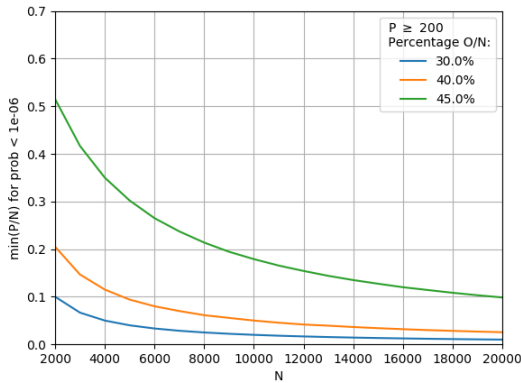
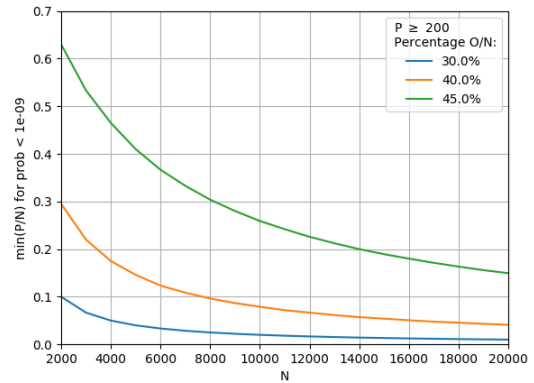
(a) Prob. Attack =  $10^{-6}$ (b) Prob. Attack =  $10^{-9}$ 

Figure 8.2: This graph shows the  $P/N$  ratio required for a probability of a 51% attack of less than  $10^{-6}$  on a validation cycle versus  $N$ . This is shown against various  $O/N$  ratio.

This series of graphs give us a good indication on what pair of parameters ( $N, P$ ) to consider for a high resilience to 51% attack. Given a number of nodes in the pool of workers, we can



deduce the number of producer nodes to select during one ledger cycle. Inversely, given a selected number of producers for a ledger cycle, we can define a minimum size for the pool of workers. As detailed in the next section, the number of producers selected for a ledger cycle is important to ensure that a consensus can be reached on the correct ledger state update to distribute to the rest of the network.

In section 5.2.1, we consider the fact that a producer may not collect exactly  $P$  local hashes during the first ledger cycle phase. We argue that a threshold  $P_{min}$  should apply to decide whether a majority of producers build the same local hash.

## 8.2 Security of the Consensus Mechanism II

Work in progress

## 8.3 Security of the Signature Scheme

### 8.3.1 Rogue Key Attack

When Schnorr signatures used to generate aggregated signature of a transaction are vulnerable to an attack known as Rogue Key attack. Rogue Key attacks performed by a malicious entity consists of generating an aggregated signature in such a way that they possess the public/private key pair for that signature. In the Schnorr signature scheme, the public keys of participants are aggregated and the sum represents the public key associated to the signature. Assume that an honest participant uses its public key  $Q_a$  in the transaction and a malicious participant possesses  $Q_b$ . By sending the public key  $Q_m = Q_b - Q_a$  to the honest participant, the malicious entity has access to the transaction as they will hold the private key for  $Q_b$ . This is because when the keys are aggregated i.e.  $Q_m + Q_a$  the aggregated signature would be  $Q_b$ , for which the malicious user holds the private key (the honest user would not). For an aggregated public key, there should be no user that has a private key equivalent as it should be used to create a signature that can be verified that all users in the transaction participated.

The aggregation of public keys used in Catalyst which is based on Mu-Sig [8] signature scheme that is not vulnerable to this form of attack. Mu-Sig is protected from this form of attack as the scheme does not require a user to demonstrate each public key, only the sum of all the public keys. By not verifying individual public keys, a key rogue attack is not possible. Only one public key is needed for the verification (the aggregated key) for which there will not be an equivalent private key.

### 8.3.2 Quantum Attack

Quantum computers pose a very real threat to the encryption techniques used in blockchains in the medium to long term [6] [19]. The threat is through the use of Shor's algorithm. A quantum attacker using Shor's algorithm on a quantum computer can gain an exponential speed-up in solving the discrete logarithmic problem. The assumption of security the discrete logarithmic functions is the primary basis as to which all elliptic curve cryptography is based. This means that even the schema demonstrated here will be vulnerable to attack. The use of aggregated signatures would provide some resistance, however this resistance would be negligible.

It must be impressed that this is not an issue for the near term and thereby, these schema are highly secure and efficient currently. The most efficient algorithm for classical computers to

solve the discrete logarithm problem is the Pollard's rho[7], this does not run in polynomial time. While Catalyst is not currently resistant to quantum attack, this is a challenge that will be faced by all major distributed ledgers over time.

# Conclusion

Work in progress



# Bibliography

- [1] G. Konstantopoulos, 2017. <https://medium.com/loom-network/understanding-blockchain-fundamentals-part-1-byzantine-fault-tolerance-245f46fe8419>
- [2] <https://www.blockchain.com/pools?timespan=24hours>
- [3] <https://www.buybitcoinworldwide.com/mining/pools>
- [4] C. Lacina, 2015. <https://cointelegraph.com/news/the-inevitable-failure-of-proof-of-stake-blockchains-and-why-a-new-algorithm-is-needed>
- [5] G. Maxwell *et al.* *Simple Schnorr Multi-Signatures with Applications to Bitcoin* 20 May 2018 Last Accessed 08/02/19: [https://pdfs.semanticscholar.org/a4da/14a4329d7bf28e2ecbf9a3e42bf1faba523e.pdf?\\_ga=2.193608301.1119480105.1549621665-1692475185.1549621665](https://pdfs.semanticscholar.org/a4da/14a4329d7bf28e2ecbf9a3e42bf1faba523e.pdf?_ga=2.193608301.1119480105.1549621665-1692475185.1549621665)
- [6] D. Aggarwal, 2018. *Quantum attacks on Bitcoin, and how to protect against them. Quantum Physics*. arXiv:1710.10377
- [7] A. Koundinya, 2013. *Performance Analysis of Parallel Pollard's Rho Factoring Algorithm*. International Journal of Computer Science & Information Technology (IJCSIT)
- [8] G. Maxwell, 2018. *Simple Schnorr Multi-Signatures with Applications to Bitcoin*. <https://eprint.iacr.org/2018/068>
- [9] G. Maxwell, 2015 *Confidential Transactions*. [https://people.xiph.org/~greg/confidential\\_values.txt](https://people.xiph.org/~greg/confidential_values.txt)
- [10] B. B'ünz *et al.*, 2019. *Bulletproofs: Short Proofs for Confidential Transactions and More*. <https://eprint.iacr.org/2017/1066.pdf>
- [11] Monero, 2014-2019. *Monero: the secure, private, untraceable cryptocurrency* <https://github.com/monero-project/monero/tree/bc208be63d880871f4f1d6b51a7b35abd5e9b8f0>
- [12] D. J. Bernstein and T. Lange, 2016. *Failures in NIST's ECC standards*. <https://cr.yp.to/newelliptic/nistecc-20160106.pdf>
- [13] D. J. Bernstein, 2017. *Ed25519: high-speed high-security signatures*. <https://ed25519.cr.yp.to/>
- [14] <https://coinguides.org/blake2b/>
- [15] <https://digiconomist.net/ethereum-energy-consumption>
- [16] B. Gehring, 2018. *Swiss Payment Monitor 2018*, p18. [swisspaymentmonitor.ch](http://swisspaymentmonitor.ch)

- [17] <https://etherscan.io/chart>, 21/05/2019
- [18] <https://www.blockchain.com/en/charts>, 21/05/2019
- [19] J. J. Kearney and C. A. Pérez-Delgado, 2019. *Blockchain Technologies Vulnerability to Quantum Attacks*