Catalyst Distributed Operating System:

Catalyst Core Protocol

Dr. Pauline Bernat[*], Joseph Kearney[†] and Francesca Sage-Ling[‡]

Version 1.0

Last edited: June 24, 2019

[*]pauline.bernat@atlascity.io
[†]joseph.kearney@atlascity.io
[‡]fran.sl@atlascity.io

Abstract

Distributed Ledger Technology (DLT) is undoubtedly one of the most disruptive new technologies to have emerged over the past decade. When the paper [1] was published in 2008 by an unknown author under the famous pseudonym Satoshi Nakamoto, the world was shaken by a global economical crisis that considerably deteriorated our trust in central authorities. The new protocol embedded in this paper proposed a revolutionary approach to conduct transfers of digital assets, without the need of a third party, thus opening the door to a new economical approach with greater transparency, privacy and prosperity for all. Bitcoin was the first successful public blockchain to demonstrate the potential for this technology to be used as a decentralised yet trusted store of value. Building on this early success, next generation blockchains such as Ethereum and Neo demonstrated the potential for blockchain platforms to provide decentralised computing services, enabling more complex applications and reaching more markets than straight forward storage of value. Subsequent blockchains and distributed ledgers established use-cases in many other areas notably through the use of Internet of Things devices and machine learning techniques [2].

As Bitcoin and other projects grew in popularity it became apparent that real challenges awaited this new technology. Indeed, the early systems were not built to meet the demand for services at scales comparable to those of cloud services. The main challenge of blockchain is to solve the so-called blockchain trilemma, building a system that can process a high throughput of transactions while ensuring the system integrity and accessibility to all. Additional concerns arise notably around the risk of power centralisation that would inevitably lead to the level of wealth disparity we observe in a world governed by centralised systems, and to disastrous environmental impacts. A plethora of projects started with in mind to tackle these challenges (and more).

Building a blockchain or distributed ledger is a complex task and for that reason most existing projects are clones, also known as forks, made from a small number of original blockchains. This allows organisations to benefit from already developed blockchains while modifying the elements relevant to their field. The problem with such an approach is that it restricts truly original thinking about wider technological issues such as how a network can scale or operate in environments for which the original Bitcoin blockchain was never designed. As a result of forking from the past, the fundamental issues restricting present blockchain technologies such as scale, privacy, performance and interoperability remain as much of a challenge today as when these early blockchains were first developed [3].

Atlas City took a very different approach when designing a new core protocol ledger and accompanying distributed computing capability, starting from a set of operational requirements and developing a cohesive system that delivers to those requirements. The code base developed by Atlas City researchers and engineers is original and will be made available as an open-source software. To solve the fundamental issues inhibiting the growth of distributed ledger-based computing, engineers and researchers at Atlas City were and are encouraged to

ask and rethink fundamental questions about the new distributed operating system they envisage.

Learning from popular and new blockchains and distributed ledgers as well as the wider IT industry, Atlas City developed Catalyst, a full-stack Distributed Operating System (DOS) built to fulfil the real-world potential of DLT, to enable the next generation of distributed computing applications and business models. This paper presents the core protocol of Catalyst DOS.

Contents

Glossary

Account (definition and types) – A digital account on Catalyst is a record of KAT tokens held by an entity (individual(s) or device(s)) and defined by a digital address associated to a public key, a token balance and an account inertia charaasterise the acccount activity on the network. There are three types of accounts stored on Catalyst ledger: non-confidential accounts within which the account balance is readable to anyone, confidential accounts within which the amount is obfuscated and held in the form of a commitment, and smart contract-based accounts.

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.

Eclipse Attack - When a almost all of a nodes peers are controlled by one malicious entity. Thereby allowing the malicious entity to control the information that is passed to the honest users node.

KAT tokens – 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.

Ledger Partition - A ledger partition is a database storing accounts of a given type: confidential, non-confidential or smart-contract based accounts.

Ledger Shard - A ledger shard is an ensemble of accounts stored on a ledger partition and defined by an account inertia lying within a certain range of values. The set of accounts defining a shard changes over time as the accounts inertia evolves, based on their activity on the network.

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.

Producers - 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 a blockchain or distributed ledger. Such program can be triggered by transactions or messages generated by other codes, and/or once certain requirements have been fulfilled.

Sybil Attack - A malicious entity spins up alternative identities all under their control. Thereby giving them increase control over a network. Sybil attacks can also be used to perform a 51% attack. It can also allow them the spam the network with messages.

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 – 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

Introduction

This paper presents the core protocol behind Catalyst network. It gives an overview of the database architecture of the distributed 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 Network**: 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 types of account stored on Catalyst. The concepts of current ledger state and Distributed File System (DFS) 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 - Catalyst Consensus Mechanism**: this chapter presents the new consensus mechanism implemented on Catalyst.
- **Chapter 6 - Catalyst Scalability**: this chapter describes how sharding techniques are implemented to Catalyst consensus mechanism to maintain the network performance at very large scale.
- **Chapter 7 - Security Considerations**: this chapter discusses security considerations with regards to the signature scheme and the consensus-based protocol on Catalyst.

Chapter 1

Technical Specifications

This chapter gives an overview of the cryptographic libraries and tools used to generate, sign and verify transactions and the consensus-based protocol on Catalyst ledger that permits the update of the distributed ledger across its peer-to-peer network.

A distributed ledger 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 to 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 updated accordingly across the network.

DLT's relies on the generation of cryptographically secure ledger updates (or blocks in blockchain terms) 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' pseudonyms 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 [4]. 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 [5], is fully documented and rigid enough so that independent verifications can and have been done. This is widely seen as a security advantage, since it prevents the generating party from maliciously manipulating the curve parameters [6]. 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 [7], 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:

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2 \tag{1.1}$$

The order of Curve25519 can be expressed as $N = 2^c l$ 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 integers 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 [5] it is possible to reduce an 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 EC of the form $y^2 = x^3 + ax + b$ 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 itself generating a cyclic subgroup of order $l_G \leq n$. This subgroup is composed of the set of points: $\{0G, 1G, 2G, ..., (l_G - 1)G\}$ with $0G = l_G G$ known as the point at infinity. This subgroup is defined by the relation $xG = (x \ mod \ l_G)G$. 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 EC.

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. 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 the same 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 Blake2b-256 (or simply Blake2b) which produces a 256-bits string and is known to be amongst the fastest hashing algorithms and particularly suitable for mobile applications [8]. 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 enable the transfer of tokens between digital addresses in such a way that 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, sometimes referred to as a blinding factor.

In this paper, PC are used to obfuscate the amount of KAT tokens v_i associated to an account stored on the Catalyst 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 $b_i \in \mathbb{Z}_p$ ($p \gg M$). The generators G and H are 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 cyclical property of an EC, a PC of the form $C = vH + bG$ can be rewritten as:

$$C = (v \ mod \ l_H)H + (b \ mod \ l_G)G \tag{1.2}$$

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: $\sum_{i=1}^{n} C_i = 0H + bG$ where $bG = \sum_{i=1}^{n} b_iG$.

On the Catalyst ledger we take advantage of the cyclical nature of elliptic curve and allow for 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 create a commitment including a negative amount -5 as follows:

$C = (l_H - 5 \ mod \ l_H)H + (b \ mod \ l_G)G$. The use of positive and negative numbers in digital transactions is rather uncommon yet advantageous. Indeed, it offers an improved 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 $[0, L]$ where L represents an upper limit on the number of tokens allowed per account ($L \ll l_H$).

Confidential transactions have a cumbersome feature with respect to non-confidential ones, that is a clear increase of a transaction size as well as of 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] proposing 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 ($L = 2^n$), 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 and Units

The Catalyst native network token is named 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. It 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 0.0000000000001 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 a 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 to 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 (described in section 5.2) is implemented to incentivise users on the network to contribute to the ledger database management, offering them tokens as reward for their work. This reward typically comprises 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 considerations defining the token supply model of KAT tokens are beyond the scope of this document.

Chapter 2

Peer-to-Peer Catalyst Network

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 connected peers, randomly chosen amongst nodes in the network. 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-byte 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 2.5, and to assign a reputation to the peer identifier to track badly performing nodes. The reputation of a node is determined by how its peer identifier is formed. To maintain a good reputation, nodes need to respond correctly to requests from their peers. For example a peer can send the IP:Port number chunk of a node, and the corresponding node is required to respond with the associated public key. The node reputation decreases when providing 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 [11] (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 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 nor required to perform any other 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 KAT tokens.
- **Storage node** - A node able to sell some of its spare storage to allow other users to store their 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 the right 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 work together and follow a consensus-based protocol to build the ledger state updates, as described in section 5.2.

2.5 Nodes Registration

The selection of producers among the worker pool 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 $\mathcal{C}_n$ 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 ($\mathcal{C}_{n-1}$) 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 $\mathcal{C}_{n+1}$ in the following way: for each worker node with an identifier Id_i the quantity $u_i = Id_i \oplus r$ is defined, 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 worker pool) is sorted in ascending order and the first P identifiers in that list are the nodes selected to be producers for the cycle $\mathcal{C}_{n+1}$.

In a large network, it can be anticipated that a large number of nodes have available resources to be used to manage the ledger database and try to join the worker pool (which translates as a high demand for work). The size of the worker pool must however be determined by security as well as economical factors. Indeed, it must be profitable for a node to join the worker pool. Said otherwise, the average number of tokens earned by a producer over a period of time should cover its operational cost. As there might be more nodes willing to work than required for the worker pool, nodes may join a secondary pool, *a.k.a* a work queue, and wait to be called to join the worker pool. In order for these nodes to join the worker pool, there must be a mechanism that limits the time period during which a node can persist in the worker pool. The approach considered is to grant nodes joining the worker pool a worker pass which is valid for a limited period of time.

The list of identifiers of nodes in the worker pool can be maintained in a hash table (DHT_w) distributed across the network. Such table also stores the time of issuance of the node worker pass. 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 work queue.

By providing proof of their available resource to the network [12, 13], nodes can freely apply to become workers: they join the work queue. Every node on the network can store such proof alongside the node identifier in a secondary distributed hash table (DHT_q). As nodes leave the worker pool, 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 worker pool.

Such approach could be seen as facilitating Sybil-identity attack scenarios [14], however expensive, if an entity controls a large number nodes at the top of the work queue (at least equal to half the worker pool 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 worker pool 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 each node in the work queue. The ranking given to a node joining the work queue is not purely chronological-based. It also depends on the volume of nodes joining the queue during same allotted time period. Assume S_t nodes wanting to join the worker queue during a small window of time $[t, t + \Delta t]$. The S_t nodes first register to a tertiary queue DHT_s. At the end of the time window, a fixed and limited number of nodes from DHT_s, $z \leq S_t$, are randomly selected. z is equal to the number of nodes who left the worker pool during the time window $[t - \Delta t, t]$. These z nodes are given a ranking drawn from a normal distribution centred around R_q, which is a predetermined threshold of the worker queue length. This means that some selected nodes may obtain a ranking higher than nodes currently at the bottom of the worker queue. The rest of the nodes in the tertiary queue $(S_t - z)$ are given a ranking drawn from a normal distribution centred around $R_l = R_q - s$, where s is a shift inversely proportional to the volume of nodes in the tertiary queue DHT_s. Figure 2.1 illustrates the process of ranking allocation for nodes joining the worker queue.

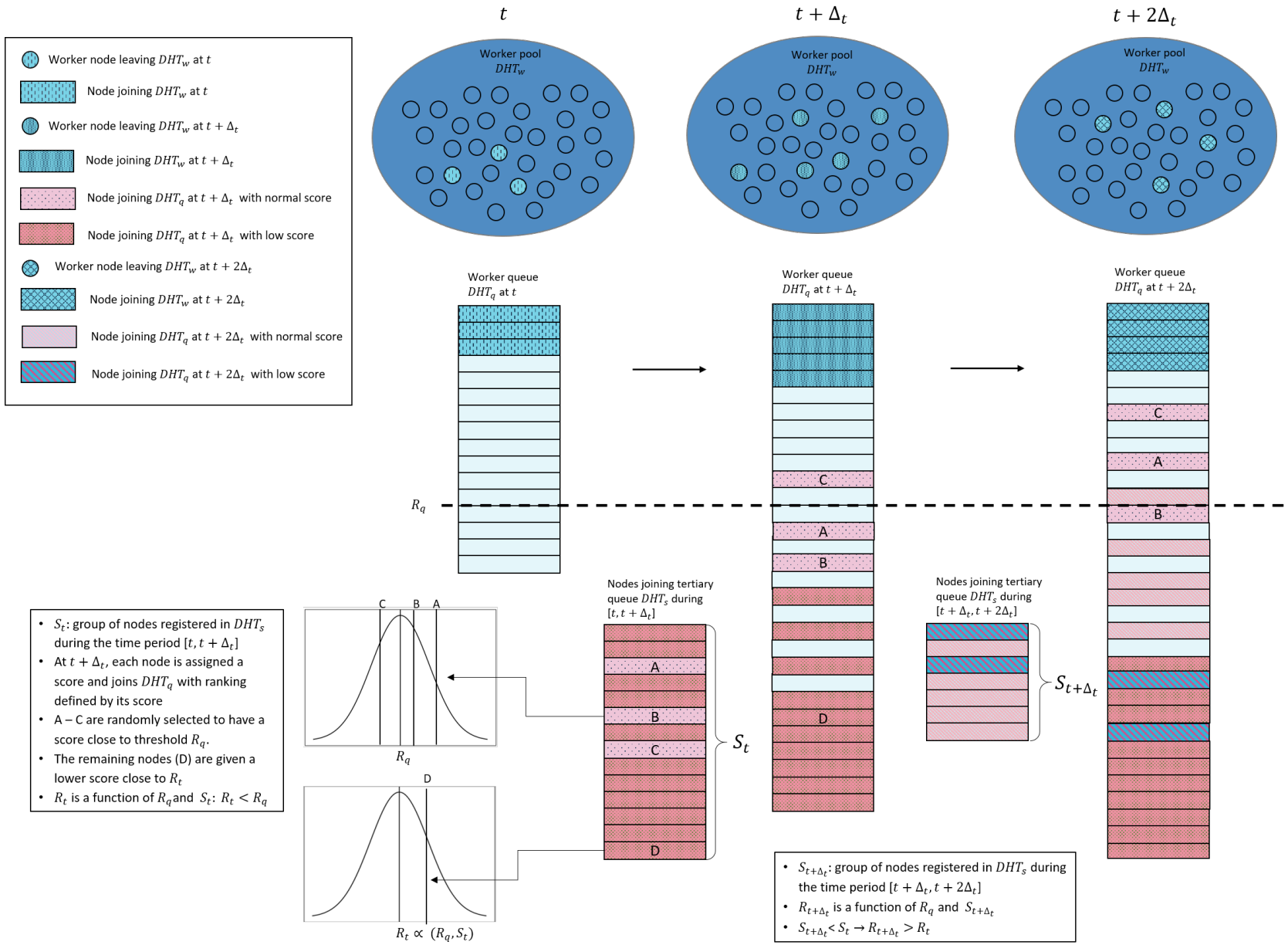


Figure 2.1: Illustration of the process followed by Catalyst network to add nodes to the work queue.

Chapter 3

Catalyst Distributed Ledger

The Catalyst database is designed to ensure Catalyst system 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.

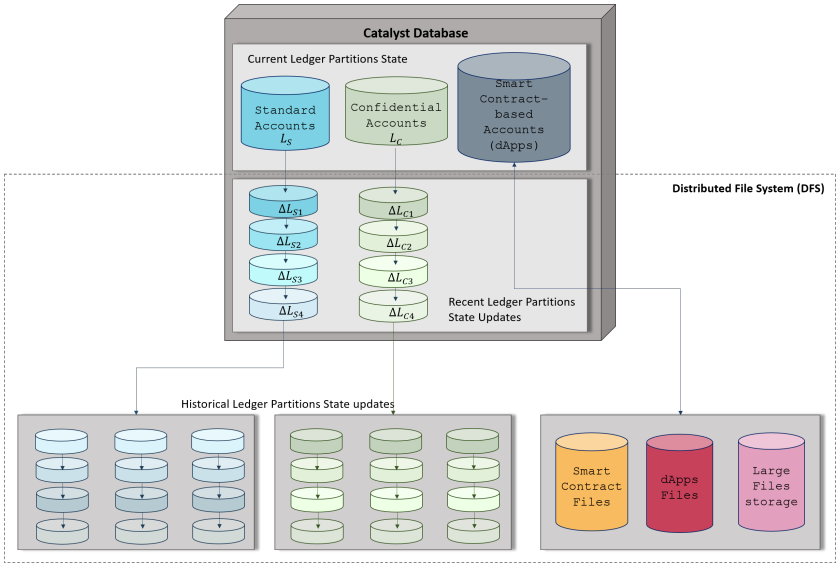


Figure 3.1: Illustration of Catalyst database architecture.

At the top level lies the current state of the ledger, *i.e.* the database containing the current balance of digital accounts recorded on the ledger. It 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 between 30 seconds and 1 minute, a ledger state update is generated by a pool of nodes selected to manage the ledger database, the producers, and distributed to the network users who can then update their local copy of the ledger state. The process followed by producers to generate a ledger update, *i.e.* the consensus-based protocol, is described in section 5.2.

The middle level comprises the recent ledger state updates, that is a set of the last recent ledger state updates accepted by producers and broadcast across the network. Historical data, or old ledger state updates, represent the bottom level. Both middle and bottom levels are maintained by the Catalyst Distributed File System (DFS) module. The top and middle levels

sits 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 shows the smart contracts and dApps stored on a database unit separate from the account balances and communicate with DFS for the access, production and storage of files. Technical specificities around smart contracts and dApps are discussed in a paper soon to be released.

3.2 Accounts on Catalyst

Different types of accounts are stored 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(s).
- Smart contract-based accounts. A smart contract-based account has an associated code that can be triggered by transactions or messages generated by other codes.

An account comprises the following components:

- An address component: a 20-byte address A_i, which is derived from a public/private key pair $\{Q_i, k_i\}$ (where $Q_i = k_i G$, G is an EC generator) using a collision-resistant hash function ($\mathcal{H}$): $A_i = \mathcal{H}(Q_i)$. The last 20 bytes of the hash are used to create the address. A prefix is added to distinguish between the different types of accounts, allowing users to hold accounts of different type yet derived from the same public key.
- An amount component: when non-confidential, the amount is a 8-byte number $v_i \in \mathbb{Z}_M$. This represents the account A_i balance in KAT tokens (with M a threshold on the number of 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 v_i of the account using a blinding factor $b_i \in \mathbb{Z}_p$ ($p \gg M$).
- An inertia component: a 8-byte number $\epsilon_i \in \mathbb{Z}$, which represents the level of activity of the account at a point in time, that is the number of token transfers involving said account. The account inertia is used in the context of a shard-based ledger, as explained in chapter 6.
- A data component: reserved to smart contract-based accounts and used to store data (or a reference to data stored on DFS) that amounts to a maximum of 64 bytes.

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

3.3 CLS Structure

The ledger state thus encompasses different partitions, each of which keep the balance of accounts of a specific type up-to-date. The current state of the ledger state (CLS) 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 section 4.2. 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 in discussed in section 5.2. The ledger state update is a cryptographically secure structured data object that allow 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 network users. Transactions broadcast during a ledger cycle are collected by nodes who then uses 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 row, 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 $-5\ KAT$ and one with Bob's address and a positive amount $5\ KAT$ (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 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 *a la* Bitcoin typically includes:

- a set of inputs where each input comprises the details of the account or digital address being debited, the (positive) amount associated to that address 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).

On the Bitcoin blockchain, valid blocks of transactions get appended to the blockchain in such a way that any new block is cryptographically sealed and linked to the last block appended to the blockchain. A block contains a set of transactions that transfer digital assets from a set of digital addresses to another set, as well as an extra transaction, called a *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) refers to the output of a transaction stored on a previous block (*a.k.a* an 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 store UTXOs. 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. As detailed in sections 4.2 and 5.2, the removal of UTXOs is made possible via the combination of a novel consensus-based protocol and a new transaction structure such that any token transfer embedded in a transaction (whether spending or receiving tokens) is signed and thus authorised by the relevant parties involved in said transfer. User nodes need not access old ledger state updates to be able to transfer tokens from their account stored on Catalyst ledger. They only need a local copy of the current ledger state.

Once a ledger state update is generated by a pool of producers, it is stored on DFS and can be accessed by any node to update their local copy of the current ledger state. DFS is built upon the IPFS protocol [15] and is 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 updates is an approach taken to prevent the bloating of the ledger and allow the network to support services a scale. Indeed, this approach allows Catalyst ledger to remain both lean and cryptographically secure.

Chapter 4

Catalyst 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 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 KAT 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 section 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 $\{E_i\}_{i=1,..,n}$. 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 worker pool.
- The transaction fees paid by the transaction participants
- A data field that can contain up to 60 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. Any other field can be included to the transaction, depending on the nature of the token exchange.

4.3 Transaction Entries

Transaction entries are used on the Catalyst ledger to represent the transfer of tokens into or out of the account referenced in the entries. 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 two 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 (when the transaction is non-confidential) or a Pedersen commitment (when the transaction is confidential) and represents the number of tokens spent from or transferred to the address associated with the public key.

On Catalyst, the amount or number of tokens included in a transaction entry 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 in 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
  - A range proof $\Pi_i(C_i')$ (as discussed in section 1.3) that proves that the token balance of A_i remains within an acceptable range of value (typically greater than 0 and smaller than a threshold M of number of tokens) after the transaction has taken place.

  The construction of these elements is discussed below.

Assume that the balance v_i of the account A_i is initially represented on the ledger by the PC:

$$C_i = (v_i \ mod \ l_H)H + (b_i \ mod \ l_G)G \tag{4.1}$$

Where b_i is a blinding factor chosen by the account holder. The latter wishes to transfer or receive a_i tokens. To obfuscate the amount of tokens transferred in a transaction entry E_i, the account holder creates a PC:

$$C_i^t = (v_i^t \ mod \ l_H)H + (b_i^t \ mod \ l_G)G \tag{4.2}$$

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 for v_i^t. U_i thus creates a second PC:

$$C_i' = (v_i' \ mod \ l_H)H + (b_i' \ mod \ l_G)G \tag{4.3}$$

Defined as:

$$C_i' = C_i + C_i^t \begin{cases} v_i' = v_i + v_i^t \\ b_i' = b_i + b_i^t \end{cases}$$

It represents the commitment of the account balance after the transaction has taken place. If $a_i < 0$, $v_i'H = [v_i + (l_H + a_i) mod \ l_H]H = [v_i + a_i]H$. 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.

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

Another type of transaction entry is considered on Catalyst, that is a stand-alone entry. It is not included in a transaction but is added to the ledger state update generated by the producers during the ledger cycle and includes the reward allocated to a specific producer for its contribution in producing a valid update of the ledger state. Such entry, called *ledger*

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

Table 4.1: Structure of confidential and non-confidential transactions on Catalyst and size per transaction component.

compensation entry (or simply compensation entry), is very similar to a non-confidential entry. It includes an 8-byte amount, that is however always positive, and a 32-byte public key from which the address of an account stored on the ledger is derived. However, unlike transaction entry, a compensation entry need not be signed to authorise the transfer of tokens to the account address specified in said entry.

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 can verify the validity of the signature given the public key Q_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* [16] 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 $\{E_i\}_{i=1,..,n}$ and additional information X mentioned in the previous section (see table 4.1), excluding the transaction timestamp:

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

The participant U_i responsible for E_i (holder of the account A_i) creates the following challenge:

$$e_i = \mathcal{H}(m \mid\mid \tilde{Q})\mathcal{H}(L \mid Q_i) \tag{4.4}$$

Where:

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

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

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

U_i then creates the following partial signature:

$$s_i = r_i + e_i \cdot k_i \tag{4.5}$$

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 $(s_i,\ R_i)$ with $R_i = r_i G$ is forwarded to the other transaction participants. Each participant in the transaction U_k $(k \neq i)$ can compute:

$$R_i' = s_i \cdot G - e_i \cdot Q_i \tag{4.6}$$

where $Q_i = k_i \cdot G$ 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 transaction signature of non-confidential transaction is composed of the pair:

$$T = (\underbrace{s_1 + .. + s_n}_{s}, \underbrace{R_1 + .. + R_n}_{R}) \tag{4.7}$$

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

1. Compute the list of n challenges $\{e_i'\}_{i=1,..,n}$ given the public keys embedded in the transaction entries $\{E_i\}_{i=1,..,n}$ included in m

2. Compute the quantity $R' = s \cdot G - \sum_{i=1}^{n} e'_i Q_i$

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 transaction signature of confidential transaction is defined differently although its structure is the same as that of non-confidential transaction . Using the public keys $\{D_i = d_i G\}_{i=1,..,n}$, it is composed of the pair:

$$T = (\underbrace{s_1 + .. + s_n}_{s}, \underbrace{D_1 + .. + D_n}_{D}) \tag{4.8}$$

The verifier can compute:

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

If $R' = s \cdot G - \sum_{i=1}^{n} e'_i Q_i$, 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 adding the transaction fees paid by the different participant, thus ensuring that no tokens are created or lost in the transaction.

Once the transaction signature is valid, the participant can append it to the transaction. The transaction timestamp is defined as the time when the transaction is completed by said participant and ready for broadcast across the network.

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

Catalyst 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) [17][1]. 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.
- Collectively across a network of nodes there is significant distributed computer resources to securely maintain a ledger. Network performance should as a result improve as the network scales up.

[1] This energy consumption allows approximately 445 million transactions for Bitcoin and Ethereum combined per year [18, 19], compared to Switzerland where 820 million debit card transactions are processed per year [20] for an estimated energy consumption of 0.001358 TWh.

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 [21]. 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.

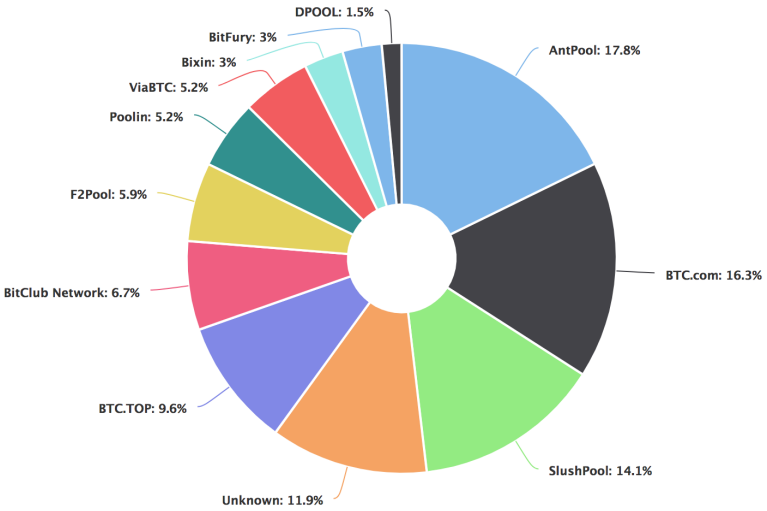


Figure 5.1: Distribution of the Bitcoin hashrate power over a 24h period, as the 30 of October 2018[22]

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 [23] 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 bock 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 remains the risk for centralisation of wealth and subsequently the network management, with the mining work inevitably distributed to a few wealthy nodes [24].

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. The algorithm used by nodes to produce a valid ledger state update does not require the execution of computationally expensive tasks, allowing nodes with limited resources to contribute. 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 the following sections we assume that the ledger is composed of one single partition comprising a fixed set of accounts. The implementation of sharding techniques are discussed in a chapter 6. We therefore consider one single worker pool and one subset of producer nodes selected per ledger cycle.

A ledger cycle $\mathcal{C}_n$ starts at time $t = t_{n,0}$ and lasts for a period of time Δt_{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 $\mathcal{C}_n$. Each producer P_j can be identified by its peers as well as the rest of the network via its unique identifier Id_j (see section 2.1).

During $\mathcal{C}_n$ the P producers collaborate to create a ledger state update ΔL_n based on the set of m_{n-1} transactions broadcast on the network during the previous ledger cycle $\mathcal{C}_{n-1}$. To limit discrepancies in the set of transactions collected by the different producers and processed during $\mathcal{C}_n$ a small time window Δt_{freeze} is considered. The m_{n-1} transactions $\{Tx_j\}_{j=1,..,m_{n-1}}$ are actually collected during the period of time $[t_{n-1,0} - \Delta t_{freeze}, t_{n,0}]$ ($t_{n-1,0} = t_{n,0} - \Delta t_{cycle}$) and must have a timestamp comprised between $t_{n-1,0} - \Delta t_{freeze}$ 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 $\mathcal{C}_n$, 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 process followed by producer nodes during a ledger cycle is described in phases. The first three phases consist in producing the correct ledger state update before its broadcast to the entire network. During each phase, a producer P_i generates a quantity α_i and broadcasts it to the network, while collecting the α_j quantities generated by the producers $\{P_j\}_{j \in P/i}$. The final phase ensures the ledger state is updated across the network.

5.2 Protocol

Section 2.5 describes how user nodes register to become worker nodes and can be selected from the worker pool 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 ΔL_n for the ledger cycle $\mathcal{C}_n$ starts at $t = t_{n,0}$ and last for a period of time Δt_{cycle}. At the end of the ledger cycle, nodes in the network use ΔL_n to update their local copy of the ledger state. This section describes the different phases of a ledger cycle.

5.2.1 Construction Phase

During the first phase (*a.k.a* construction phase) of the ledger cycle $\mathcal{C}_n$, 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 $t = t_p = t_{n,0}$ and lasts for a period of time Δt_p, therefore ending at $t_p + \Delta t_p$.

Local ledger state update generation and broadcast

At $t = t_p$, the producer P_j flushes its mempool from the m_{n-1} transactions $\{Tx_i\}_{i=1,..,m_{n-1}}$ 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 $[t_p, t_p + \Delta t_{p0}]$ ($\Delta t_{c0} < \Delta t_p$). The producer uses a salt s, defined using a pseudo-random number generator that takes for seed the Merkel tree of the previous valid ledger state update ΔL_{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 section 4.3) $\{E_\alpha\}_{\alpha=1,...,n_i}$, 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_α, P_j creates a corresponding hash variable:

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

   Each pair (E_α, O_α) 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 $L_E^f = \{E_\beta\}_{\beta=1,...,M}$

are sorted following a lexicographical order based on their associated hash variable: $O_1 < O_2 < ... < O_\beta < ... < O_M$. 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 transactions entries list L_E^f concatenated with the hash tree of the transactions signature d_n:

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

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

$$\boxed{h_j = h_{\Delta j} \mid\mid Id_j} \tag{5.1}$$

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 section 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.

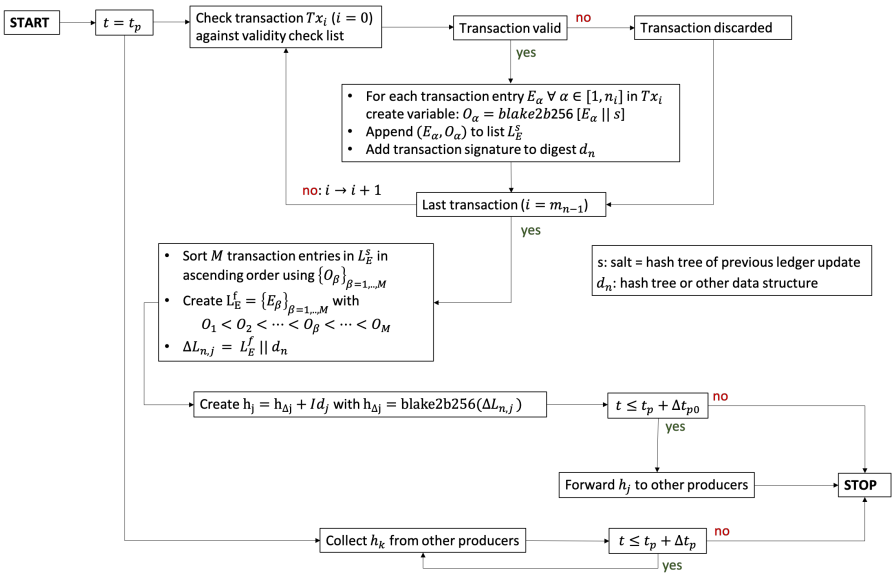


Figure 5.2: Flowchart illustrated the steps followed by a producer P_j node during a period of time Δt_p leading to the broadcast of a local hash h_j.

6. At $t \leq t_p + \Delta t_{p0}$, 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 construction phase ($t = t_p + \Delta t_p$), C_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 $\mathcal{C}_n$, the producer P_j collects at most $P - 1$ contributions ($e.g\ C_j = P$) 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 practice, a producer may not collect exactly P local hashes during $\Delta t_p (e.g\ C_j \leq P)$ 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 Campaigning Phase

During the second phase (*a.k.a* campaigning phase) of a ledger cycle, a producer P_j designates a candidate for the correct ledger state update. At the end of the process, producers forward their proposed candidate to their peers.

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

Local candidate generation and broadcast

Using the C_j contributions stored in its mempool, P_j follows a series of steps during a period of time Δt_{c0} ($\Delta t_{c0} < \Delta t_c$):

1. P_j verifies that the same local hash $h_{\Delta j}^{maj}$ is embedded in a majority C^{maj} of contributions, where $h_{\Delta j}^{maj} = max[unique(h_{\Delta k})\ \forall\ k \in \{C_j\}]$ and $C^{maj} = count[(h_{\Delta k} = h_{\Delta j}^{maj})\ \forall\ k \in \{C_j\}]$. The threshold, $C_{threshold}$, 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 C_{min} and C^{maj}. Threshold considerations are discussed in detail in section 7.2.

   If $C_j > C_{min}$ and $C^{maj} > C_{threshold}$:

   - 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:

     $$\boxed{c_j = h_{\Delta j}^{maj}\ ||\ \#(L_j(prod))\ ||\ Id_j} \tag{5.2}$$

     Where # represents a hash tree or some other compressed data structure of the list $L_j(prod)$. c_j corresponds to P_j's candidate for 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 $t \leq t_c + \Delta t_{c0}$, P_j broadcasts its candidate c_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 c_j.

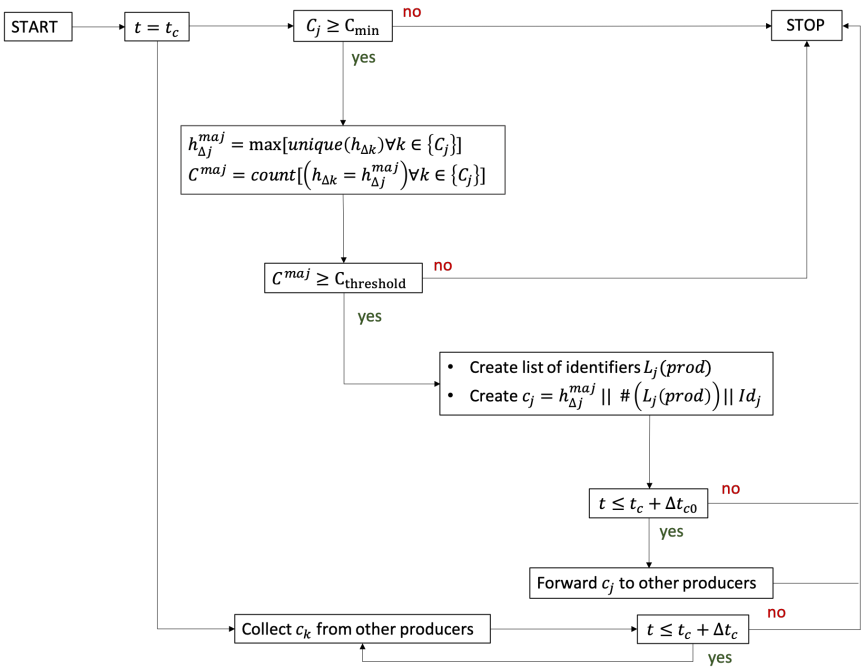


Figure 5.3: Flowchart illustrating the series of steps followed by a producer P_j to issue a candidate c_j.

Local candidates collection

Shortly after the second phase starts (at $t \approx t_c$), P_j starts collecting other candidates c_k generated by other producers $\{P_k\}_{k \in P/j}$ in its mempool. The collection lasts for a period of time Δt_c after which the producer holds V_j candidates in its mempool ($V_{min} \leq V_j \leq P$).

5.2.3 Voting Phase

During the third 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 including a reward to some producers to their peers.

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

Ballot generation and broadcast

At $t = t_v$:

1. P_j verifies that the same local hash h^{maj} is embedded in a majority of candidates. With $h^{maj} = max[unique(h_{\Delta k}^{maj}) \ \forall \ k \in \{V_j\}]$ and $V^{maj} = count[(h_{\Delta k}^{maj} = h^{maj}) \ \forall \ k \in \{V_j\}]$, this condition is met if $V^{maj} > V_{threshold}$ (See section 7.2 for more explanations).

2. The producer P_j can only participate in the following steps 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 producer, any two producers P_j and P_k would build the same list of identifiers $\mathcal{L}_j(prod) = \mathcal{L}_k(prod)$. In practice, 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 compensation entry to the ledger state update, that 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.

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 different hash $h_{\Delta j}$ to $h_{\Delta j}^{maj}$ 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 c_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 candidate c_k satisfying $h_{\Delta k}^{maj} = h^{maj}$ if and only if that identifier is included in at least $P/2$ lists $\{\mathcal{L}_k(prod)\}_{k=1,..,V_j}$ associated to a candidate c_k satisfying $h_{\Delta k}^{maj} = h^{maj}$. 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 number 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.

P_j follows a series of step for a period of time Δt_{v0} ($\Delta t_{v0} < \Delta t_v$):

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 candidate c_k satisfying $h_{\Delta k}^{maj} = h^{maj}$.

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 candidate c_k satisfying $h_{\Delta k}^{maj} = h^{maj}$ if and only if that identifier is included in at least $P/2$ lists $\{\mathcal{L}_k(prod)\}_{k=1,..,V_j}$ associated to a candidate c_k satisfying $h_{\Delta k}^{maj} = h^{maj}$.

3. P_j then creates a list L_{CE} of compensation 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 this remainder goes to the producers who voted correctly on the previous ledger cycle update. Let $\mathcal{L}_{n-1}(vote)$ be the list of the identifiers of producers who voted correctly on the previous ledger cycle update $\mathcal{C}_{n-1}$. We later demonstrate how such a list is derived during a ledger cycle. For now, let's assume that L_{CE} includes compensation entries for producers involved the production of the ledger state update for this ledger cycle $\mathcal{C}_n$ and the producers involved in the voting process of the precedent cycle $\mathcal{C}_{n-1}$.

4. P_j then creates the final ledger state update for $\mathcal{C}_n$ including the reward allocated to the producers for their contribution:

$$\mathbf{LSU_j = L_E^f \;||\; d_n \;||\; L_{CE}}$$

P_j then computes its final vote (or ballot):

$$\boxed{v_j = \mathcal{H}(LSU_j) \;||\; \#(\mathcal{L}_j(vote)) \;||\; Id_j} \tag{5.3}$$

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

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

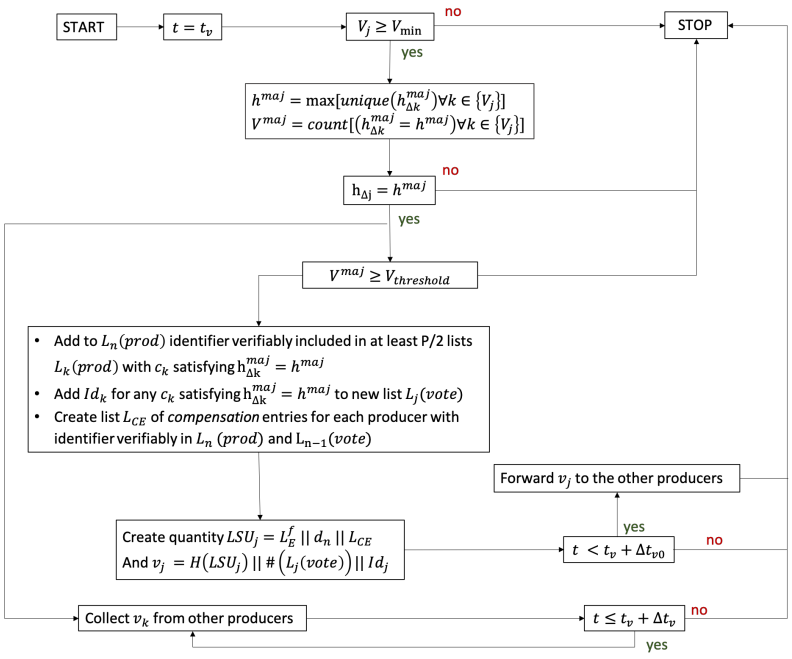


Figure 5.4: Flowchart illustrating the series of steps followed by a producer P_j during the voting phase of the ledger cycle.

Ballots collection

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

5.2.4 Synchronisation Phase

Final ledger state update generation and broadcast

The last phase (*a.k.a* synchronisation phase) of a ledger cycle starts at $t = t_s$, with $t_s = t_{n,0} + \Delta t_p + \Delta t_c + \Delta t_v$, and lasts for a period of time Δt_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 ΔL_n for the cycle $\mathcal{C}_n$ as:
   $\mathcal{H}(\Delta L_n) = max[unique(\mathcal{H}(LSU_k))\ \forall\ k \in \{U_j\}]$ and the associated number of votes collected: $U^{maj} = count[(\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n))\ \forall\ k \in \{U_j\}]$ and verifies that $U^{maj} > U_{threshold}$.

2. P_j creates a new list $\mathcal{L}_n(vote)$ and appends to $\mathcal{L}_n(vote)$ the identifier of a producer included in the list $\mathcal{L}_k(vote)$ of a vote v_k satisfying $\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n)$ if and only if the identifier is included in at least $C_n/2$ lists $\{\mathcal{L}_k(vote)\}$ associated to a vote v_k satisfying $\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n)$. Note that C_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 $\mathcal{L}_n(prod)$.

3. P_j then creates the following quantities:

$$\boxed{\mathbf{H_j} = \mathcal{H}(\mathbf{\Delta L_n}) \mid\mid \#(\mathcal{L}_\mathbf{n}(\mathbf{vote})) \mid\mid \mathbf{Id_j}} \tag{5.4}$$

4. The producer then broadcasts H_j to the network.

Ledger state synchronisation across the network

During the time period $[t_s, t_s + \Delta t_{cycle}]$, worker nodes, as well as DFS nodes collect $\{H_k\}_{\forall k \in P}$ quantities broadcast by the producers. By extracting the identifier Id_k embedded in any collected quantity H_k, a DFS node can easily compile a list of producer identifiers having broadcast the same quantity $\mathcal{H}(\Delta L_n)$ (concatenated with the same list $\mathcal{L}_n(vote)$). Producers having generated the correct ledger state update ΔL_n can forward it to DFS nodes. Once a DFS node has collected $x \geq P/2$ identical hash values, they can accept any ΔL_n value matching $\mathcal{H}(\Delta L_n)$. User nodes can then request the ledger state update ΔL_n from DFS nodes and 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 list $\mathcal{L}_n(vote)$ embedded in each H_k quantity. If selected to be a producer for the next cycle $\mathcal{C}_{n+1}$, a worker can use it to generate the reward allocated to the producers who correctly voted for the accurate ledger state update during the ledger cycle $\mathcal{C}_n$.

Figure 5.5 summarises the different phases of the ledger cycle.

The various parameters and thresholds mentioned in this chapter and their impact on the levels of security and confidence in the successful production of a ledger state update are discussed in section 7.2.

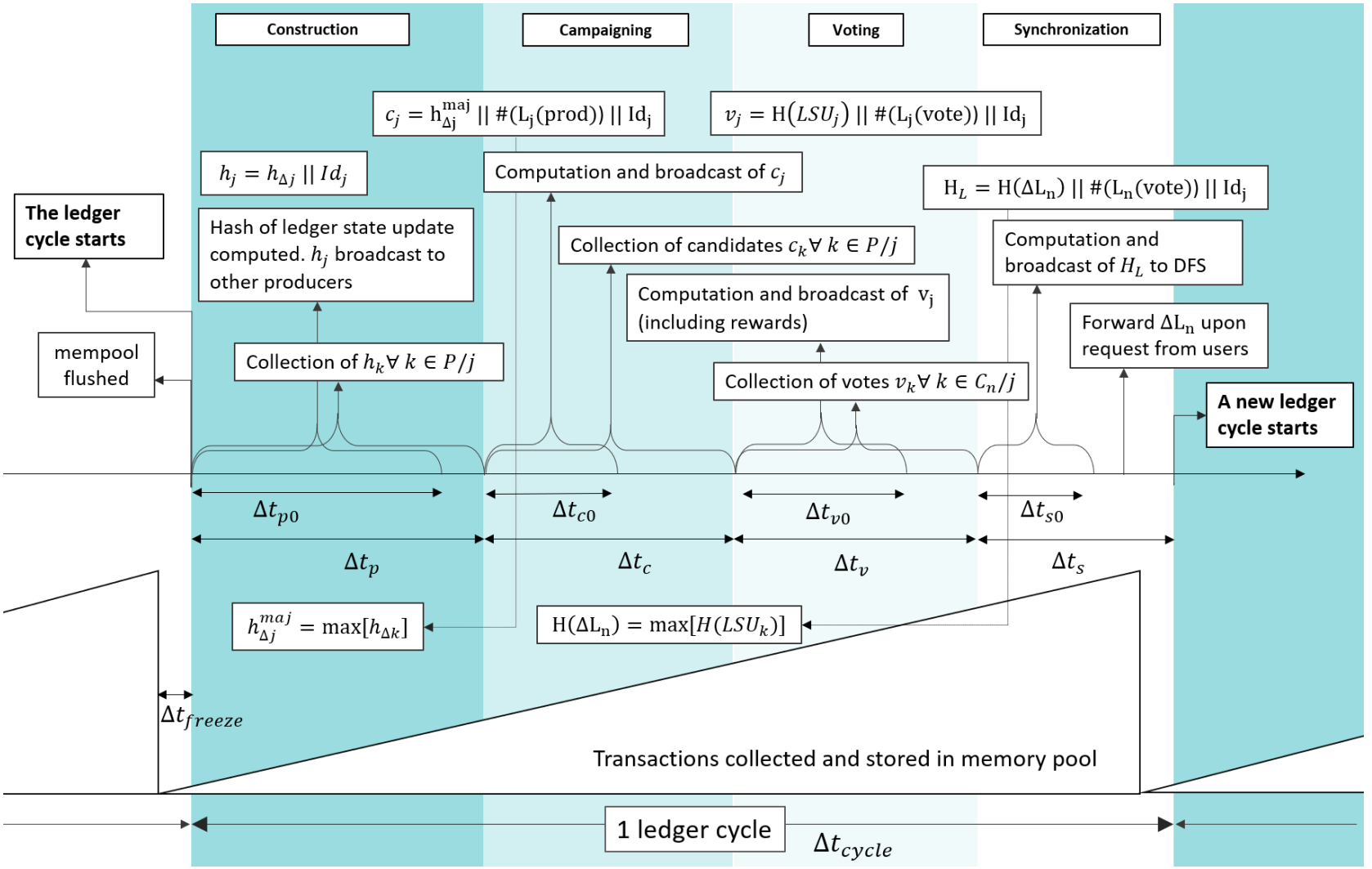


Figure 5.5: Illustration of the different phases followed by a producer during a ledger cycle.

Chapter 6

Scaling Catalyst

I'm writing this section

As Catalyst network scales up, the ledger partitions (partitions storing non-confidential accounts, confidential accounts, smart contract-based accounts) are expected to grow as new and existing users create new digital accounts stored on the ledger and generate more activity on the network, issuing an increasing number of token transfers, notably by using the services available on the network. Even though the current ledger state as well as the ledger state update of a partition may grow less rapidly than traditional blockchains which maintain every single transaction ever created, scaling techniques are considered to ensure the network performance not only remains stable but also improves as the network popularity grows. Indeed, as mentioned in section 5.1.1, the approach followed to design Catalyst protocol took in consideration the reality that the global resource available across a network would genuinely increase as the network grows and should be seen as an advantage rather than a constraint. Said differently, Catalyst protocol was developed with the idea that a distributed ledger should naturally remain decentralised and secure as the ledger database and network scale up. Moreover, Catalyst ledger was inherently conceived to support various partitions and allow these partitions to communicate or exchange tokens in ways that wouldn't compromise the rules inherent to each of them.

This chapter continues in the lines of the chapter 5 dedicated to Catalyst consensus-based protocol and comprises two parts: the first part dedicated to the use of sharding techniques applied to each partition for an efficient yet secure management of the partition state, and the second part devoted to partitions interoperability.

6.1 Dynamic Account Ordering

Having the ability to prune a ledger (or blockchain) of obsolete data without compromising the integrity of the system offers a direct way to regulate the size of the ledger database. Furthermore, the state of a ledger partition comprises accounts that have different levels of activity, *i.e.* are involved in a varying number of token transfer, and therefore require different level of resource to be maintained. Being able to distinguish between accounts that are active from those resulting in fewer transfer of tokens offer the possibility to better allocate resource available across the network.

As explained in section 3.2, each partition on Catalyst ledger lists digital accounts address and their balance in KAT tokens, whether the balance is represented by a number in non-confidential account or a Pedersen Commitment in confidential account. In this section, we

considered adding an extra field to each account that represents the account activity, called the account inertia (mentioned in section 3.2).The inertia of an account is defined by the account activity relative to the overall activity of a partition and is updated at the end of each ledger cycle. The inertia of accounts involved in one or more token transfers during a ledger cycle decreases while the inertia of accounts inactive during the same period of time increases. The ordering of accounts in a ledger partition is then based on a Least Recently Used (LRU) algorithm that is ran at the end of each ledger cycle. Active accounts which have a low inertia sits at the top of the ledger partition state. Accounts with no activity gain inertia and are progressively moved towards the bottom of the ledger state. Accounts that have reached a critical inertia are eventually moved to DFS where they can still have accessed and re-activated for a period of time, after which they are removed permanently unless an explicit agreement is made between the account owner and the nodes responsible for maintaining DFS.

The ordering of accounts in a ledger partition based on the accounts inertia results in a dynamic multi-levelled structure regulated by the network activity. The accounts are split in sub-states, or shards, where each shard is characterised by a range of account inertia values. Assume a ensemble $\mathcal{M}$ of accounts stored on a ledger partition with each account $\mathcal{A}_i$ $\forall i \in \mathcal{M}$ defined by the set (A_i, v_i, ϵ_i), respectively the account address, amount and inertia. A partition $\mathcal{P}$ is split in s shards $(\mathcal{P}_1, .., \mathcal{P}_s)$ where each shard $\mathcal{P}_w$ $\forall w \in [1, s]$ comprises a subset $\mathcal{M}_w$ of accounts such that an account $\mathcal{A}_i$ in $\mathcal{M}_w$ is characterised by an inertia $\epsilon_i \in [\epsilon_w, \epsilon_{w+1})$ (ϵ_w is the inertia threshold separating shards $\mathcal{P}_{w-1}$ and $\mathcal{P}_w$).

Each shard in a partition thus comprises accounts that have similar level of activity on the network. Shards with active accounts will require more resource to be maintained and as such can be managed by heavy nodes. At the contrary, shards of accounts involved in fewer token transfers will result in more compact ledger partition state update and can thus rely on smaller nodes to generate these updates. The shards can thus be managed by separate worker pools with each worker pool characterised by a specific range of computing resources, such as the CPU [12] per node.

The global ledger state update generated during a ledger cycle thus comprise a set of ledger shard state update. Producers selected in the worker pool of a shard to generate the ledger shard state update can reach consensus following the consensus-based protocol described in the chapter 5. However, given that a transaction may embed a transfer of tokens between accounts stored on different shards, there needs to be a mechanism to ensure that any transfer, say from account A to B, partially included the state update of the shard storing A is also included in the state update of the shard storing B. Although each producer pool may reach consensus on the correct shard state update, a safeguard needs to be put in place to ensure that the global ledger state update does not destroy existing tokens nor create tokens outside the scope of the token injection mechanism. To that end, this section describes a new phase followed by the producers during a ledger cycle that take place during the synchronisation phase and ensure that any transaction is properly included in the state update of the different shards storing the accounts involved in said transaction. Additional steps taken during the earlier ledger cycle phases are also described.

6.2 Protocol in a Shard-based Ledger

During the construction phase described in chapter 5, a producer P_j computes a quantity L_E^f that lists the transaction entries included in the transactions collected by producer during the

period of time $[t_{n-1,0} - \Delta t_{freeze}, t_{n,0}]$. In a shard-based ledger, the producer P_j is now a member of the producer pool tasked with updating the state of a ledger shard $\mathcal{P}_w$ defined by a subset $\mathcal{M}_w$ of accounts. As such, P_j does not store all m_{n-1} transactions issued during the ledger cycle $\mathcal{C}_{n-1}$ in its mempool but instead only collects the subset $m_{n-1,w}$ of transactions that contain at least one entry involving an account included in the subset $\mathcal{M}_w$. Some of these transactions will include entries that involve accounts stored on other shards (of the same ledger partition). Let $\mathcal{E}_w^{in}$ be the set of transaction entries involving an account included in $\mathcal{M}_w$ and $\mathcal{E}_w^{out}$ the set of transaction entries involving an account not included in $\mathcal{M}_w$ such that $\mathcal{E}_w^{in} \cup \mathcal{E}_w^{out}$ comprises the n_i entries $\{E_\alpha\}_{\alpha=1,..,n_i}$ of each transaction Tx_i $\forall i \in [1, m_{n-1,w}]$.

In chapter 5, a simple pool of producers agreed on the correct ledger state update ΔL_n built during the cycle $\mathcal{C}_n$. In the context of a shard-based ledger, the producers tasked with updating the state of a ledger shard $\mathcal{P}_w$ work collaboratively to generate the ledger shard state update ΔL_n^w. The set of shard state updates constitutes the global ledger state update: $\Delta L_n = \Delta L_n^1 + ... + \Delta L_n^k + ... + \Delta L_n^s$.

The P producers responsible for the ledger shard $\mathcal{P}_w$ follow a protocol similar to that described in 5. The differences, explained below.....

The producer P_j executes the following steps to create L_E^f:

- For each transaction entry E_α in $\mathcal{E}_w^{in}$, P_j creates a corresponding hash variable:

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

  Each pair (E_α, O_α) is added to a list L_E^s. Steps (1) and (2) are repeated until all transactions have been processed.

- P_j then creates a new list L_E^f using the $M_w = \sum_{i=1}^{\mathcal{E}_w^{in}} n_i$ transaction entries listed in L_E^s such that the transaction entries in $L_E^f = \{E_\beta\}_{\beta=1,...,M_w}$ are sorted following a lexicographical order based on their associated hash variable: $O_1 < O_2 < ... < O_\beta < ... < O_{M_w}$.

- P_j also creates a hash table HT_j using the entries in $\mathcal{E}_w^{in}$.

- The producer then computes a new local hash:

  $$\boxed{h_j = h_{\Delta j} \mid\mid r(HT_j) \mid\mid Id_j} \tag{6.1}$$

  h_j differs from the one defined in equation 5.1 as it includes $r(HT_j)$, the root of HT_j hash table.

P_j then broadcasts h_j to the other producers in the network and collects other $\{h_k\}_{k \in P/j}$ contributions generated by its producer peers $\{P_k\}_{k \in P/j}$ as described in section 5.2.1.

During the campaigning phase, P_j verifies that the same local hash $h_{\Delta j}^{maj}$ is embedded in a majority C^{maj} of contributions, where:

- $h_{\Delta j}^{maj} \mid\mid r(HT^{maj}) = max[unique(h_{\Delta k} \mid\mid r(HT_k)) \; \forall \; k \in \{C_j\}]$
- $C^{maj} = count[(h_{\Delta k} = h_{\Delta j}^{maj} \& r(HT_k) = r(HT^{maj})) \; \forall \; k \in \{C_j\}]$.

The remaining steps in this phase are as described in section 5.2.2.

During the voting phase, a producer P_j verifying $h_{\Delta j} = h^{maj}$ creates its ballot as:

$$\boxed{v_j = \mathcal{H}(LSU_j) \mid\mid r(HT^{maj}) \mid\mid \#(\mathcal{L}_j(vote)) \mid\mid Id_j} \tag{6.2}$$

P_j then broadcasts v_j to the other producers in the network and collects other $\{v_k\}_{k \in P/j}$ votes generated by its producer peers $\{P_k\}_{k \in C_n/j}$ as described in section 5.2.1.

During the synchronisation phase, a producer P_j computes a quantity $\mathbf{H_j}$ (equation (5.4)) that embeds the ledger state update generated a majority of producers within the producer pool. In the context of a shard-based ledger, the quantity $\mathbf{H_j}$ is defined as:

$$\boxed{\mathbf{H_j} = \mathcal{H}(\mathbf{\Delta L_n^w}) \mid\mid \mathcal{H}(\mathcal{L}_\mathbf{n}^\mathbf{w}(\mathbf{vote})) \mid\mid \mathbf{HT^{maj}} \mid\mid \mathbf{Id_j}} \tag{6.3}$$

It includes the hash of the identifier list $\mathcal{L}_n^w(vote)$ of producers who voted correctly on the ledger shard state update $\mathcal{P}_w$. It also includes the hast tree HT^{maj} corresponding to the set of transaction entries $\mathcal{E}_w^{in}$ included in ΔL_n^w.

6.3 Partitions Interdependencies

Ledger partitions (partitions storing non-confidential accounts, confidential accounts,...)

Chapter 7

Security Considerations

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. 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.

7.1 Selection of Worker and Producer Nodes

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. The probability P_{51} depends on the following parameters:

- N : the total number of nodes in the worker pool.
- 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 worker pool ($0 \leq O \leq N$). 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. ($0 \leq p \leq P$).

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 worker pool, 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 $N \gg P$, P_{51} can there be expressed by the discrete sum:

$$P_{51} = \sum_{p=p_0}^{P} P_A(p) \tag{7.1}$$

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:

$$P_A(p) = \frac{\overbrace{\binom{O}{p}}^{\text{A}}\overbrace{\binom{N-O}{P-p}}^{\text{B}}}{\underbrace{\binom{N}{P}}_{\text{C}}} \tag{7.2}$$

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 worker pool. Finally, C corresponds to the number of available combinations for choosing P nodes from the pool of N nodes.

In equation 7.2, $P_A(p)$ 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 worker pool 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 worker pool, nodes are part of a work queue. Nodes in the worker pool are granted a work pass valid for finite period time. As a result, a varying number of nodes leaves the worker pool at each ledger cycle. Although the size of the worker pool might be constant (N nodes), the selection of nodes actually forming the worker pool 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 the probability over a series of ledger cycles.

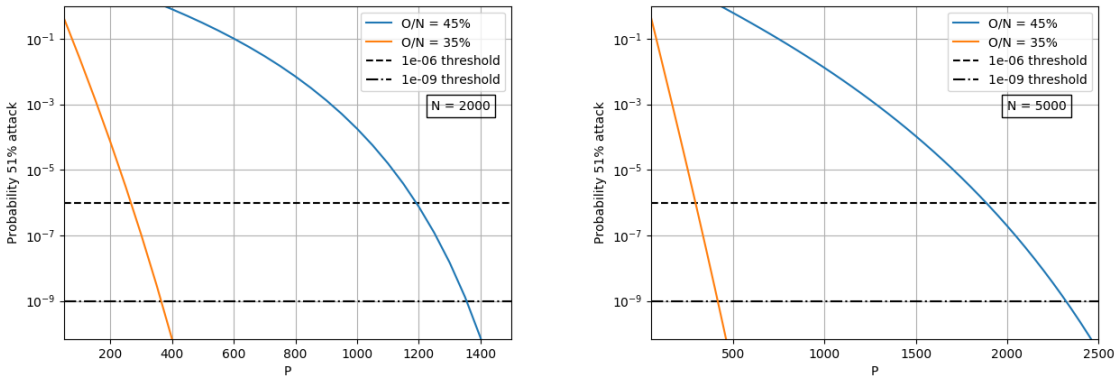
When $N \gg P$ the probability of a successful attack can therefore be estimated using the cumulative hypergeometric distribution function (CDF) for $p \in [p_0, P]$. In this paper, we provide probability estimate obtained using *scipy.stats* Python library. The graphs presented are obtained using *matplotlib.pyplot* 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[1].

As an example, assume a rather large number of nodes in the worker pool, $N = 20,000$, 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 worker pool 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 worker pool.

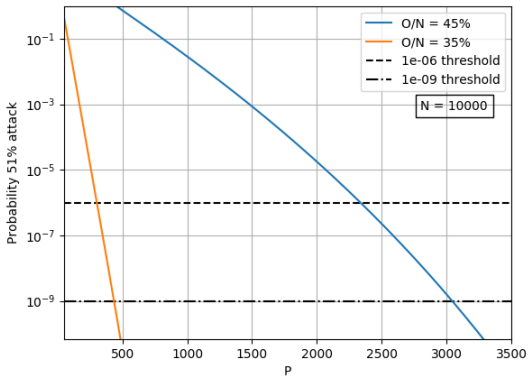
Figure 7.1 shows the probability of a successful control of more than 50% of the producers as a function of the number of producers for four different worker pool sizes and two attack scenarios: when a malicious entity controls $O/N = 45\%$ of the worker nodes in blue, and in orange when a malicious 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 worker pool 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.

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

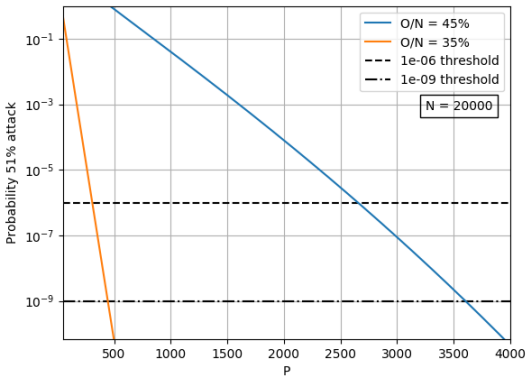


(a) $N = 2000$

(b) $N = 5000$



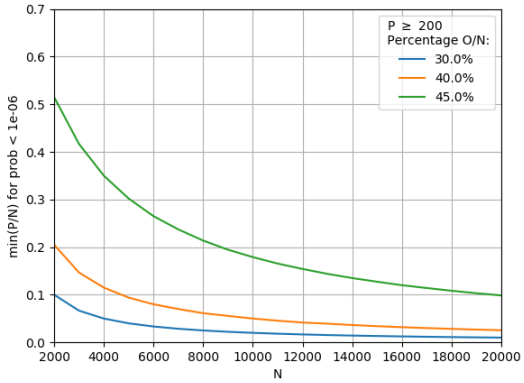
(c) $N = 10000$



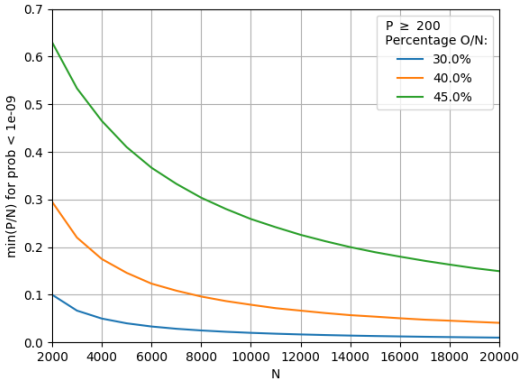
(d) $N = 20000$

Figure 7.1: Probability of 51% attack as a function of P for various worker pool 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 7.2 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 7.2: 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 gives 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 worker pool, 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 worker pool. 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.

7.2 Production of a Ledger State Update

The previous section discusses the level of security against 51% attack when a malicious entity controlling more than half the producer nodes can attempt to tamper with the ledger update. Specifically, the security of the consensus mechanism is considered as a function choice of parameters (P, N). As N becomes large and the ratio P/N is low, it becomes very unlikeky for a malicious entity to gain control of the worker pool, notwithstanding an increasingly expensive cost of attack.

In this section, we explore the confidence level associated with the production of a ledger state update. During the last phase of the consensus mechanism, each node on the network updates their local copy of the ledger with what they perceive as being the correct ledger state update generated by the producers. Each user node must collect $x > P/2$ identical hashes from the producers to safely conclude that a consensus was reached amongst the producers. Recall that a producer P_j broadcasts $H_j = \mathcal{H}(\Delta L_n) \,||\, \#(\mathcal{L}_n(vote)) \,||\, Id_j$ to the network. The producer identifier Id_j is used by a user node to distinguish between the hash generated by two producers. The correct $\mathcal{H}^c(\Delta L_n)$ is thus defined by $\mathcal{H}^c(\Delta L_n) = max[unique(H_k \;/|\; Id_k) \;\forall\; k \in \{P\}]$ with $/|$ a reverse concatenate, and $x = count[(H_k \;/|\; Id_k = \mathcal{H}^c(\Delta L_n)) \;\forall\; k \in \{P\}]$. If $x = P$, all producers agree on the correct ledger state update for cycle $\mathcal{C}_n$.

As laid out in section 5.2, a producer executes a series of steps in each phase of the consensus mechanism in a consecutive manner. The producer can only move to a phase if a set of conditions are fulfilled in the previous phase. For a producer P_j, the first three phases consist of generating a quantity α_j that obeys certain criteria, and then broadcasting it to its producer peers while collecting the quantities α_k produced and broadcast by other producers $\{P_k\}_{k \in P/j}$:

1. **Construction phase:** $\alpha_j = h_j$
   h_j is the hash of the ledger state update (excluding any compensation entry) generated by P_j, using the set of transactions stored in its mempool, concatenated with its identifier Id_j (see equation 5.1): $h_j = h_{\Delta j} \,||\, Id_j$.

   Participation All producers $\{P_j\}_{\forall j \in P}$ participate in the construction phase.

   Time h_j must be broadcast before $t_p + \Delta t_{p0}$. Other local hashes are collected during the time period $[t_p, t_p + \Delta t_p]$.

   Quality Each transaction included in the ledger state update must verify a list of validity checks (see section 4.5).

2. **Campaigning phase**: $\alpha_j = c_j$
   c_j is the ledger state update candidate generated by P_j (see equation 5.2):
   $c_j = h^{maj}_{\Delta j} \,||\, \#(L_j(prod)) \,||\, Id_j$ with $h^{maj}_{\Delta j}$ the hash of the most common ledger state update found by P_j given the set of local hashes collected during the construction phase. $L_j(prod)$ is the partial list of identifiers compiled by P_j which includes the identifier of

any producer having broadcast a local hash corresponding to the most common ledger state update.

Participation All producers $\{P_j\}_{\forall j \in P}$ participate in the campaigning phase.

Time c_j must be broadcast before $t_c + \Delta t_{c0}$. Other candidates are collected during the time period $[t_c, t_c + \Delta t_c]$.

Quality
- The number C_j of local hash collected by P_j must verify $C_j \geq C_{min}$.
- The number of identical local hashes $C^{maj} = count[(h_{\Delta k} = h_{\Delta j}^{maj}) \ \forall \ k \in \{C_j\}]$ must verify $C^{maj} \geq C_{threshold}$.

3. **Voting phase**: $\alpha_j = v_j$
v_j is the vote generated by P_j (see equation 5.3): $v_j = \mathcal{H}(LSU_j) \ || \ \#(\mathcal{L}_j(vote)) \ || \ Id_j$ with $LSU_j = L_E^f \ || \ d_n \ || \ L_{CE}$ the candidate of the ledger state update locally elected by P_j which includes the compensation entries for the producers $\{P_k\}$ would generate a correct ledger state update verifying $h_j = \mathcal{H}(L_E^f \ || \ d_n)$. $L_j(vote)$ is the partial list of identifiers compiled by P_j which includes the identifier of any producer having broadcast a candidate corresponding to the most common ledger state update. L_{CE} is the list of compensation entries created using the identifiers included in $L_n(prod)$, the complete and final list of C_n producers having broadcast a local hash corresponding to the most common ledger state update.

Participation Only producers finding a $h^{maj} = max[unique(h_{\Delta k}^{maj}) \ \forall \ k \in \{V_j\}]$ satisfying $h^{maj} = h_j$ participate.

Time v_j must be broadcast before $t_v + \Delta t_{v0}$. Other votes are collected during the time period $[t_v, t_v + \Delta t_v]$.

Quality
- The number V_j of candidate collected by P_j must verify $V_j \geq V_{min}$.
- The number of identical local hash $V^{maj} = count[(h_{\Delta k}^{maj} = h^{maj}) \ \forall \ k \in \{V_j\}]$ must verify $V^{maj} \geq V_{threshold}$.
- $L_n(prod)$ includes the identifier of producers included in at least $P/2$ lists $\{\mathcal{L}_k(prod)\}_{k=1,..,V_j}$ associated to a candidate v_k satisfying $h_{\Delta k}^{maj} = h^{maj}$.

4. **Synchronisation phase**: $\alpha_j = H_j$
H_j is the hash of the elected ledger state update ΔL_n generated and broadcast by P_j (see equation 5.4): $H_j = \mathcal{H}(\Delta L_n) \ || \ \#(\mathcal{L}_n(vote)) \ || \ Id_j$.

Participation All producers $\{P_j\}_{\forall j \in P}$ participate in the synchronisation phase.

Time H_j must be broadcast before $t_s + \Delta t_{s0}$. User nodes must collect at least x identical $\mathcal{H}(\Delta L_n)$ during the time period $[t_s, t_s + \Delta t_s]$ and request the corresponding ledger state update to synchronise their local copy of the ledger.

Quality
- The number U_j of votes collected by P_j must verify $U_j \geq U_{min}$.
- The number of identical hashes $U^{maj} = count[(\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n)) \ \forall \ k \in \{U_j\}]$ must verify $U^{maj} > U_{threshold}$.
- $L_n(vote)$ includes the identifier of producers included in at least $C_n/2$ lists $\{\mathcal{L}_k(vote)\}_{k=1,..,C_n}$ associated to a vote w_k satisfying $\mathcal{H}(LSU_k) = \mathcal{H}(\Delta L_n)$. C_n corresponds to the number of identifier of producers who correctly computed the ledger state update and are therefore included in $\mathcal{L}_n(prod)$.

The probability $\mathcal{P}(x > P/2)$ that $x > P/2$ at the synchronisation phase depends on a series of criteria:

1. $(C_{min}, C_{threshold})$: a producer needs to collect enough individual local hashes (at least C_{min}) and find a majority (at least $C_{threshold}$) of identical ledger state update hashes to be able to issue a candidate. C_{min} is typically defined as a fraction of P: $C_{min} = f_C P$ with $0 < f_C < 1$. On the other hand, the definition of $C_{threshold}$ is more complex and depends on C_j.

   Although in theory $C_{threshold}$ could be set at $C_j/2$, a higher threshold must be chosen to allow a producer to decide on a candidate in good confidence. Indeed, one must account for the statistical uncertainty associated to the ratio C^{maj}/C_j due to the size of the data sample used to compute this ratio. Moreover, there should be no ambiguity on the choice of a candidate, should for instance a second set of identical hashes of size close to $C_j/2$ be found in an attempt to tamper with the ledger state by a malicious entity controlling a large number of worker nodes. The confidence interval on a ratio $r = C^{maj}/C_j$ is defined as:

$$r \pm z\sqrt{\frac{r(1-r)}{C_j}} \tag{7.3}$$

   Where z is a score associated to the confidence level in r ($z = 4.22$ for a 99.999% confidence level) and the remaining expression is the standard error of the ratio estimate.

   In an scenario where only two types of hash are collected by a producer P_j, $h_1 = h_{\Delta j}^{maj}$ and $h_2 \neq h_{\Delta j}^{maj}$, P_j compiles the two ratios $r_1 = h_1/C_j$ and $r_2 = h_2/C_j$. Since $h_2 = C_j - h_1$, the two ratios have the same margin error: $\Delta r_1 = \Delta r_2$. As illustrated in Figure 7.3, if the margin error associated to the two ratios are such that $r_1 - \Delta r_1 < r_2 + \Delta r_2$, P_j cannot say with certainty that a majority of nodes agrees, even if $r_1 > 50\%$. A decision can only really be made if $r_1 > 50\% + \Delta r_1$. Figure 7.3(left) shows that for a $r_1 = 0.7$ the producer must collect at least $C_j = 110$ data in order to remove any ambiguity with a confidence level at 99.999%. Indeed, if $r_1 = 70\%$ and $C_j = 2000$, the second ratio r_2 would represent at best 30% of the data collected by the producer, the statistical uncertainty on these two ratios would leave a significant gap between 34.3% and 65.7%. For $V = 1000$, that gap would be reduced to [36.1%,63.9%], still large enough to give enough confidence to a producer that a clear majority of nodes agree on a common data. This is illustrated in Figure 7.3(right) when $R_1 = 0.6$. It can be seen that when $C_j = 200$ that there would be an overlap between the margin errors around r_1 and r_2, while when $C_j = 2000$ the producer can conclude with a confidence level of 99.999% that $r_1 > r_2$.

   $C_{threshold}$ is therefore defined for confidence level (CL) as:

$$C_{threshold}(CL) = \left(0.5 + z(CL)\sqrt{\frac{C^{maj}(C_j - C^{maj})}{C_j^3}}\right) \times C_j \tag{7.4}$$

   For a confidence level at 99.999%, $C_{threshold}$ can be expressed as:

$$C_{threshold}(99.999\%) = \left(0.5 + 4.22\sqrt{\frac{C^{maj}(C_j - C^{maj})}{C_j^3}}\right) \times C_j \tag{7.5}$$

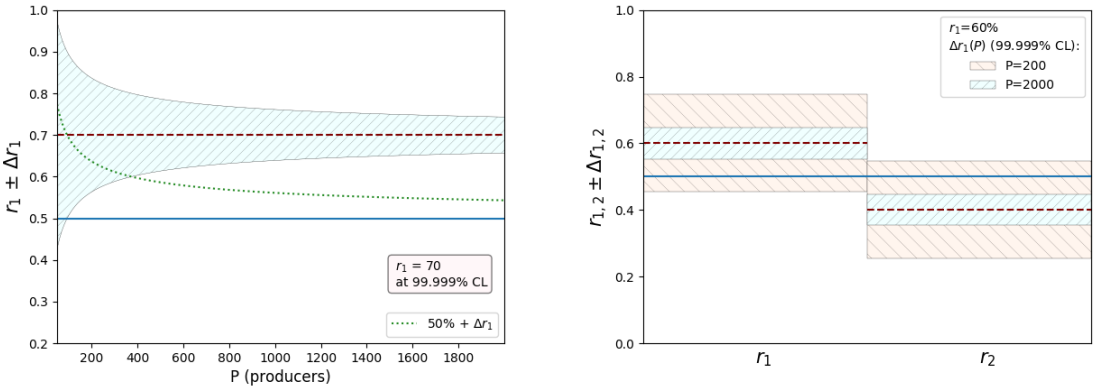


Figure 7.3: Left: $ri \pm \Delta r_i$ ($i = 1$ *or* 2) as a function of P, the size of the producers pool, when $r_1 = 60\%$. Right: $r \pm \Delta r$ at 99.999% confidence level, for two values of P (200, 2000) when only two types of hash are collected by a producer, when $r_1 = 70\%$.

2. $(V_{min}, V_{threshold})$: a producer needs to collect enough individual candidates (at least V_{min}) and find a majority (at least $V_{threshold}$) of identical hash embedded in the candidate to be able to issue a vote. V_{min} is typically defined as a fraction of P: $V_{min} = f_V P$ with $0 < f_V < 1$. $V_{threshold}$ is defined following the same approach considered for $C_{threshold}$:

$$V_{threshold}(99.999\%) = \left(0.5 + 4.22\sqrt{\frac{V^{maj}(V_j - V^{maj})}{V_j^3}}\right) \times V_j \tag{7.6}$$

3. $(U_{min}, U_{threshold}, C_n)$: a producer needs to collect enough individual votes (at least U_{min}) and find a majority (at least $U_{threshold}$) of votes with identical hash embedded in these to be able to confidently broadcast the hash of the next ledger state update across the network. Two votes are considered identical if they include the same hash of a complete ledger state update including the compensation entries that reward the producers for their work. Two complete ledger state updates are therefore identical notably if the lists $\mathcal{L}_n(prod)$ used to create the compensation entries are identical. The list $\mathcal{L}_n(prod)$ comprises the identifiers of the C_n producers that produced the right ledger state update (without compensation entries) during the production phase. C_n is typically defined as a fraction of P: $C_n = f_{prod} P$ with $0 < f_{prod} < 1$. U_{min} is defined as a fraction of C_n: $U_{min} = f_U C_n$ with $0 < f_U < 1$. Indeed U_{min} is a fraction of producers among these that had produced a candidate such that the local ledger update embedded in said candidate corresponding to ΔL_n /| L_{CE}. $U_{threshold}$ is defined following the same approach considered for $C_{threshold}$:

$$U_{threshold}(99.999\%) = \left(0.5 + 4.22\sqrt{\frac{U^{maj}(U_j - U^{maj})}{U_j^3}}\right) \times U_j \tag{7.7}$$

In summary the probability $\mathcal{P}(x > P/2)$ that $x > P/2$ can be expressed as a function of $P, C_{min}, C_n, V_{min}, U_{min}$:

$$\mathcal{P}(x > P/2) = f(f_{prod}, f_C, f_V, f_U) \begin{cases} f_{prod} = C_n/P, f_C = C_{min}/P \\ f_V = V_{min}/P, f_U = U_{min}/C_n \end{cases}$$

The tests conducted on the gossip protocol implemented on Catalyst suggest that a high percentage of nodes $(90-95\%)$ in a large network will successfully collect data from all their peers. Furthermore tests on a smaller networks such as the sub-networks of workers and producers in charge of producing the ledger state update during a ledger cycle gives the percentage of nodes collecting the data from all their peers as close to 100%. As a result the numbers (C_j, V_j) of data collected by a producer P_j are naturally expected to be close to P and U_j close to C_n. A simulation analysis was done to determine the optimal set of parameters $(C_n, C_{min}, V_{min}, U_{min})$ to ensure a probability $\mathcal{P}(x > P/2)$ greater than 99.999% for various sizes of the producers pool P. Figure 7.4 displays a *n-ary* tree $(n = 4)$ illustrating the minimum sets of parameters $(f_{prod}, f_C, f_V, f_U)$ found for a pool of producers made of (a) 200 nodes and (b) 500 nodes when varying the parameters $(f_{prod}, f_C, f_V, f_U)$ between 0.75 and 0.95 with a step of 0.05. We observe that for $P = 200$, when $f_{prod} = 80\%$ of producers generate the correct ledger state update, $\mathcal{P}(x > P/2) > 99.999\%$ when all thresholds are set at 80%. The values represented in the tree branches show how the thresholds naturally decrease as the number P of nodes in the pool of producers increases.

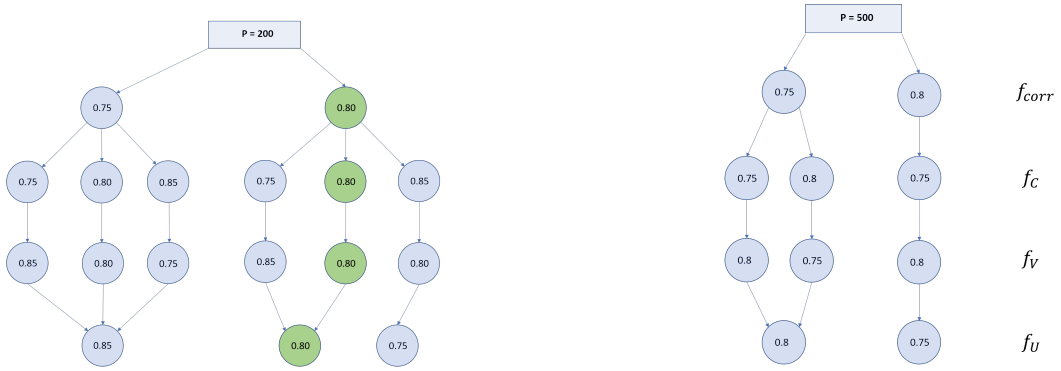


Figure 7.4: Minimum sets of parameters $(f_{prod}, f_C, f_V, f_U)$ found for $\mathcal{P}(x > P/2) > 99.999\%$ for P = 200 (left) and P = 500 (right).

Figure 7.5 shows the minimum common threshold found for (f_C, f_V, f_U) as a function P when $f_{prod} = 75\%$ and $f_{prod} = 95\%$.

At $P = 1000$, the two curves converge to a common threshold $f_C = f_V = f_U = 76\% \pm 1\%$. When $f_{prod} = 75\%$, we find that in approximately 95% of the conducted tests, $x = P$, all producers broadcast the same quantity $\mathcal{H}(\Delta L_n) \parallel \#(\mathcal{L}_n(vote))$. When $P = 200$ and $f_{prod} = 75\%$, a higher common threshold $f_C = f_V = f_U = 85\% \pm 1\%$ is found to satisfy $\mathcal{P}(x > P/2) > 99.999\%$ and $x = P$ in roughly 80% of the tests.

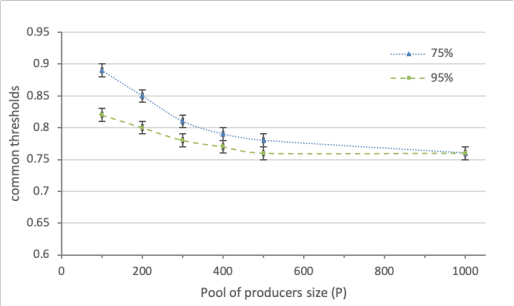


Figure 7.5: Minimum common set of parameters (f_C, f_V, f_U) as a function of P when $f_{prod} = 75\%$ (blue) and $f_{prod} = 95\%$ (green).

The decision regarding the thresholds chosen for generating the ledger state update thus depends on P. As explained in section 7.1, the choice of P as well as that of the worker pool size N also influence the resistance to 51% attack. Figure 7.1 shows that in order to keep the probability of a successful 51% attack below 1 in a billion when $N = 2000$, the size of producers must comprise at least $P = 1200$ producers. The simulation conducted in this section shows that when $P \geq 1000$ a common threshold value $f_C = f_V = f_U = 80\%$ ensures that a consensus

can be reached at 99.999% confidence level when $f_{prod} \geq 75\%$ for any x value greater or equal to $P/2$.

7.3 Signature Scheme

7.3.1 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 posses the public/private key pair for that signature. In the Schnorr signature scheme, the public key of participants are aggregated and the sum represent the public key associated to the signature. Assume that an honest participant use 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 have access to the transaction as they will hold the private key for Q_b. This is because when the keys are aggregated i.e. $Q_m + Q_b$ 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 [25] 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.

7.3.2 Quantum Attack

Quantum computers pose a very real threat to the encryption techniques used in blockchains in the medium to long term [26] [27]. 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 the 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[28], 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

The individual technical components underpinning Distributed Ledger and Blockchain Technologies have existed for decades. As the 1^{st} generation blockchain, Bitcoin managed to recombine these previously established elements in a unique fashion so as to instil and enable trust in a trust-less system, thus achieving decentralisation and eliminating the need for a centralised authority. Whilst completely revolutionary at the time, the implementation and expansion of this new approach uncovered limitations and hurdles to both expanded use and ultimately mainstream adoption. 2^{nd} generation DLTs and blockchains built and improved upon this original foundation but fall short of resolving all associated issues.

Atlas City has developed a distributed operating system, Catalyst, to solve the issues of previous DLTs and blockchains, improving upon those which came before, resolving such challenges and enabling an equitable and proportionate compensation to participants on the network. Catalyst was designed around the notion that a democratic and ethical network can exist which is secure, decentralised, scalable and private.

Catalyst code base does not fork from a previously existing projects and includes original and innovating work, including a new collaborative and environment-friendly consensus-based protocol, the possibility to process both confidential and non-confidential transactions as well as smart contracts, an efficient peer-to-peer communication layer and a multi-level data architecture for a lean ledger database storing a variety of data.

This paper gives an overview of Catalyst core protocol. Another paper, currently under preparation, presents network performances measurement.

Bibliography

[1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* https://bitcoin.org/bitcoin.pdf. Oct. 2008.

[2] UK Government Chief Scientific Adviser. *Distributed Ledger Techonology: beyond block chain.* https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf. 2016.

[3] D. Roe. *10 Obstacles To Enterprise Blockchain Adoption.* https://www.cmswire.com/information-management/10-obstacles-to-enterprise-blockchain-adoption/. June 2018.

[4] D. Hankerson and A. Menezes. "NIST Elliptic Curves". In: *Encyclopedia of Cryptography and Security.* Ed. by H. C. A. van Tilborg and S. Jajodia. Boston, MA: Springer US, 2011, pp. 843–844. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_255. URL: https://doi.org/10.1007/978-1-4419-5906-5_255.

[5] Monero Project. *Monero: the secure, private, untraceable cryptocurrency.* https://github.com/monero-project/monero/tree/bc208be63d880871f4f1d6b51a7b35abd5e9b8f0. 2014.

[6] D. J. Bernstein and T. Lange. *Failures in NIST's ECC standards.* https://cr.yp.to/newelliptic/nistecc-20160106.pdf. Jan. 2016.

[7] D. J. Bernstein. *Ed25519: high-speed high-security signatures.* https://ed25519.cr.yp.to/. 2017.

[8] coinguides. *Blake2b Algorithm – List of Blake (2b) coins, miners and its hashrate.* https://coinguides.org/blake2b/. 2017.

[9] G. Maxwell. *Confidential Transactions.* https://people.xiph.org/~greg/confidential_values.txt. 2015.

[10] B. Bunz et al. *Bulletproofs: Short Proofs for Confidential Transactions and More.* https://eprint.iacr.org/2017/1066.pdf. 2019.

[11] C. Sherlock et al. "Efficiency of delayed-acceptance random walk Metropolis algorithms". In: *arXiv:1506.08155v1* (June 2015).

[12] S. Gal-On et al. *Exploring CoreMarkTM – A Benchmark Maximizing Simplicity and Efficacy.* https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf.

[13] S. Dziembowski et al. *Proofs of Space.* https://eprint.iacr.org/2013/796.pdf. Aug. 2015.

[14] Ameya. *Sybil Attack and Byzantine Generals Problem.* https://medium.com/coinmonks/sybil-attack-and-byzantine-generals-problem-2b2366b7146b. July 2018.

[15] IPFS. https://ipfs.io/.

[16] G. Maxwell et al. *Simple Schnorr Multi-Signatures with Applications to Bitcoin*. https://pdfs.semanticscholar.org/a4da/14a4329d7bf28e2ecbf9a3e42bf1faba523e.pdf?_ga=2.193608301.1119480105.1549621665-1692475185.1549621665. Feb. 2019.

[17] digiconomist. *Ethereum Energy Consumption Index (beta)*. https://digiconomist.net/ethereum-energy-consumption. 2019.

[18] Blockchain. *Blockchain Charts*. https://www.blockchain.com/en/charts. May 2019.

[19] Etherscan. *Ethereum Transaction Chart*. https://etherscan.io/chart. May 2019.

[20] B. Gehring. *Swiss Payment Monitor 2018, p18*. swisspaymentmonitor.ch. 2018.

[21] G. Konstantopoulos. *Understanding Blockchain Fundamentals, Part 1: Byzantine Fault Tolerance*. https://medium.com/loom-network/understanding-blockchain-fundamentals-part-1-byzantine-fault-tolerance-245f46fe8419. Accessed on 2012-11-11. Dec. 2017.

[22] Blockchain.com. *Hashrate Distribution*. https://www.blockchain.com/pools?timespan=24hours. Accessed on 2018-10-31.

[23] J. Tuwiner. *Bitcoin Mining Pools*. https://www.buybitcoinworldwide.com/mining/pools. Jan. 2019.

[24] C. Lacina. *The Inevitable Failure of Proof-of-Stake Blockchains and Why a New Algorithm is Needed (Op-Ed)*. https://cointelegraph.com/news/the-inevitable-failure-of-proof-of-stake-blockchains-and-why-a-new-algorithm-is-needed. May 2015.

[25] G. Maxwell et al. *Simple Schnorr Multi-Signatures with Applications to Bitcoin*. Cryptology ePrint Archive, Report 2018/068. https://eprint.iacr.org/2018/068. 2018.

[26] D. Aggarwal et al. "Quantum attacks on Bitcoin, and how to protect against them. Quantum Physics". In: *arXiv:1710.10377* (Oct. 2017).

[27] J. J. Kearney and C. A. Perez-Delgado. *Blockchain Technologies Vulnerability to Quantum Attacks*. unknown. 2019.

[28] A. Koundinya et al. "Performance Analysis of Parallel Pollard's Rho Factoring Algorithm". In: *arXiv:1305.4365* (May 2013).