Proxima: a DAG–based cooperative distributed ledger

Evaldas Drąsutis*

Edit: 2024-04-30 (draft)

Abstract

This paper introduces a novel architecture for a distributed ledger, commonly referred to as a "blockchain", which is organized in the form of directed acyclic graph (DAG) with UTXO transactions as vertices, rather than as a chain of blocks. Consensus on the state of ledger assets, controlled by token holders, is achieved through the profit-driven behavior of token holders themselves, which is viable only when they cooperate by following *biggest ledger coverage* rule. The profit-and-consensus seeking behavior is facilitated by enforcing purposefully designed UTXO transaction validity constraints. Token holders are the sole category of participants authorized to make amendments to the ledger, making participation completely permissionless - without miners, validators, committees or staking.

In the proposed architecture, participants do not need knowledge of the global state of the system or the total order of ledger updates. The setup allows to achieve high throughput and scalability alongside with low transaction costs, while preserving key aspects of decentralization, open participation, and asynchronicity found in Bitcoin and other proof-of-work blockchains, but without insane energy consumption. Sybil protection is achieved similarly to proof-of-stake blockchains, using tokens native to the ledger, yet the architecture operates in a leaderless manner without block proposers and committee selection.

"Large numbers of strangers can cooperate successfully by believing in common myths"[1]

"Blockchains are a tool for coordinating social consensus"[2]

*Contacts: email: evaldas.drasutis@mif.vu.lt, GitHub: lunfardo314, X/Twitter: @lunfardo314

[1]Yuval Noah Harari, Sapiens: A Brief History of Humankind

[2]X/Twitter post by Dankrad Feist, an Ethereum researcher

Contents

1 Preface

This whitepaper presents Proxima design rationale at the "detailed technical vision" level. It is not intended to be neither academical paper nor a specification. Here we follow more engineering approach. To support design decisions presented in the whitepaper, we developed an advanced prototype of the Proxima node[3]. This whitepaper is partially based on findings while working on the prototype. The platform is intended for further experimental research and development.

The ideas presented in the whitepaper are inspired by two main precursors: the famous *Bitcoin whitepaper* by Satoshi Nakamoto[4] and the ideas presented in *The Tangle* whitepaper by Serguei Popov[5]. We adhere to the fundamental principles of the Bitcoin's design, especially the permissionless, stateless and asynchronous nature of the *proof-of-work* (PoW) consensus. The DAG consensus principle, "each transaction confirms other two", which implies partially-ordered structure of the ledger, is entirely taken from *The Tangle* paper. We extend the DAG data structure, the *tangle*, with deterministic UTXO[6] ledger concepts, which are naturally DAG-based. We pivot the entire reasoning around the DAG-based consensus from probabilistic to behavioral by introducing on-ledger incentives and purposefully designed ledger validity constraints, in the manner similar to Bitcoin and other blockchains. The proposed principle we call the **cooperative consensus**.

With Proxima we make no claims about solving all blockchain problems such as trilema or being better that any other solutions. We hope, however, it will attract interest from blockchain system developers and academia. Some aspects, convergence and security in particular, certainly require deeper mathematical modeling and analysis.

Disclaimer. The Proxima effort is independent of activities of the IOTA Foundation. The Proxima node prototype is not a fork nor does it contain any dependencies on implementations of IOTA. Rather, Proxima aims to independently propose a solution to the decentralization problem of the tangle (also known as "coordicide") starting from the original ideas. Proxima takes into account some lessons learned from IOTA 2.0 and offers an alternative approach.

2 Introduction

2.1 Generic principles of the permissionless distributed ledger

We start from a broad perspective to derive the most generic principles of permissionless blockchain design from Bitcoin and then apply these fundamental concepts to the Proxima architecture.

A *ledger update* T (or *ledger mutation*) is defined as a deterministic and atomic prescription that transforms the ledger state S_{prev} into another state, S_{next}. The term *deterministic* implies the transformation always leads to S_{next} for a given S_{prev}; the process is unaffected by other ledger updates that may occur before, after, or in parallel. This transformation from S_{prev} to S_{next} can also be called a *state transition*. An entity that produces state updates is said to be **writing to the ledger**. The term *atomic* indicates that the ledger update is applied in its entirety to the ledger state, never partially.

In Bitcoin and other blockchains, ledger updates take the form of blocks. Each block consists of an ordered set of transactions and a block header with a proof-of-work (PoW) nonce, a Merkle root of transactions, and a hash of the previous block. By design, each block is deterministic and atomic.

Ledger updates are subject to **validity constraints** (or **ledger constraints**). Validity constraints refer to a publicly known algorithm that, when applied to the block, provides a deterministic answer: either *yes* for a valid block or *no* for an invalid one. The expectation is that most participants in the network consider only valid blocks and reject invalid ones immediately. The participants enjoy full freedom of what ledger updates to produce, as long as they do not violate ledger constraints.

The validity constraints of a block typically involve verifying the validity of transactions within the block and checking the validity of the block header. In proof-of-work blockchains, this includes verifying

[3] Proxima repository: https://github.com/lunfardo314/proxima

[4] Bitcoin whitepaper: https://bitcoin.org/bitcoin.pdf

[5] The Tangle: https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf

[6] UTXO stands for **U**nspent **T**ransa**X**tion **O**utputs, a model of the ledger

the PoW nonce, which is easy to check but difficult to produce. The criteria for transaction validity in a block are usually blockchain-specific.

A block typically includes a **reward** for the issuer in the form of native tokens on the ledger, often newly minted (causing supply inflation) or collected as fees. The rewarding rules are subject to the ledger constraints, such as the halving rules in Bitcoin.

In the blockchain, different blocks serve as competing alternatives as ledger updates, requiring each node to decide which of several conflicting blocks to apply to its local copy of the ledger state. Valid ledger updates are freely broadcasted to all participants, and each participant (node) may choose any of them to apply to the ledger they maintain locally.

This setup leads to competition among block producers for their blocks to be accepted as updates to the global ledger state. The **behavioral assumption** is that block producers will seek profit in the form of rewards, incentivizing them to win the competition. Given the high difficulty of proof-of-work and the randomness of the nonce-mining process, the optimal strategy for profit-seeking block producers is to follow the longest chain. If they do not, their chances of catching up with the network diminish exponentially with each block they fall behind, decreasing their chances of receiving rewards. This is the well-known "longest chain wins" rule.

This results in a converging, consensus-seeking behavior (also known as **probabilistic consensus**) among **permissionless, unbounded, and generally unknown set of writers to the ledger**[7].

For us, the important general observation is that the "longest chain wins" consensus rule, followed by nodes, is not a primary fact or axiom but rather derived from basic facts and assumptions as an optimal profit-seeking strategy that leads to a game-theoretical (Nash) equilibrium among participants. These base assumptions include:

- Specific validity constraints of the ledger updates, enforced system-wide
- The liquidity of the native token on the ledger, which has real-world value and incentivizes participants to seek real-world profits

In our opinion, the distinguishing characteristic of the **Nakamoto consensus** is its open participation and permissionless writing to the ledger, rather than a specific consensus rule, which is derived from a particular set of ledger validity constraints.

We can view the validity constraints of Bitcoin blocks, PoW nonce in particular, as purposefully designed by Satoshi Nakamoto to achieve stable profit- and consensus-seeking behavior from otherwise independent and selfish writers to the ledger, in a hostile environment created by the shared interest in the distributed ledger. The Bitcoin protocol enables the **coordination of unwritten, real-world social consensus among free pseudonymous participants**.

2.2 Proxima approach. From tangle to UTXO tangle

In Proxima, we aim to achieve the same goals as Bitcoin: a decentralized distributed ledger with fully permissionless set of writers to the ledger, while applying them to a completely different ledger structure. We aim to design a set of validity constraints for a DAG-based ledger where UTXO transactions act as ledger updates instead of blocks. The profit-seeking behavior of independent producers of UTXO transactions, enforced by the validity constraints, should converge to a consensus on the ledger state. Similar to the longest chain rule, the *biggest ledger coverage rule* (also known as *heaviest state rule*) emerges from the validity constraints.

It's important to note the fundamental difference between blockchain blocks and UTXO transactions as ledger updates: blockchain blocks always conflict with each other, placing block producers in constant **competition**. In contrast, UTXO transactions treated as deterministic ledger updates may or may not conflict, allowing multiple UTXO transactions to update the same ledger state in parallel, creating different yet non-conflicting parallel ledger states. These states can be consolidated and treated as one, facilitating **cooperation**.

[7]note that PoW consensus, strictly speaking, is not even a consensus in a classical (Lamport, The Byzantine Generals Problem and derivatives) sense, which assumes bounded and known set of writers to the ledger. It is rather a "behavior" than a "protocol"

The main idea of cooperative behavior is drawn from the tangle. Let's quote the original *The Tangle* paper:

"The transactions issued by nodes constitute the site set of the tangle graph, which is the ledger for storing transactions. The edge set of the tangle is obtained in the following way: when a new transaction arrives, it must ***approve two previous transactions****. these approvals are represented by directed edges [...]".*

Furthermore:

"The main idea of the tangle is the following: to issue a transaction, users ***must work to approve other transactions****. Therefore,* ***users who issue*** *a transaction are contributing to the network's security. It is assumed that the nodes check if the approved transactions are not conflicting. If a node finds that a transaction is in conflict with the tangle history, the* ***node will not approve*** *the conflicting transaction in either a direct or indirect manner"*

The principle of one transaction approving two previous transactions results in a DAG-like data structure that is incrementally expanded by adding new vertices, called tips. DAG edges represent approval links. The specific number of approved transactions (two) is not crucial; we generalize it to "any number from 1 to a fixed N".

We consider the principle of *one transaction approving two previous transactions* as the foundation for **cooperative behavior**, as each ledger update consolidates several non-competing ledger updates into one history, unlike in the blockchain, which chooses one of many competing paths.

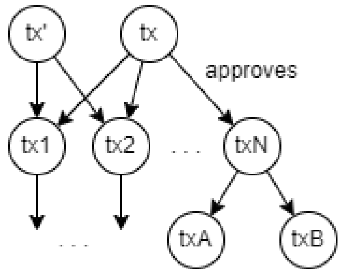


Figure 1: Tangle principle

It's important to note that the original concept of the tangle is not specific about the ledger model it uses, nor how the ledger relates to the tangle DAG. It also is ambiguous about which entities can add new transactions (vertices) to the tangle: nodes and users (token holders) are different entities, with only token holders possessing private keys required to sign a transaction.

We aim to fill these gaps by extending the generic tangle to the **UTXO tangle**. The well known UTXO ledgers (Bitcoin, Cardano) already forms a DAG, with vertices as transactions and edges representing consumed outputs of other transactions. We expand this model by allowing optional **endorsement** links to be added to the UTXO transaction. These *endorsement* links resemble both the original tangle's vertex approval links and blockchain block references to predecessor blocks. *Endorsements* indicate a declared and enforced consistency of the ledger update with referenced ledger updates.

These transactions form a DAG with two types of edges: (a) consuming the output of other transactions, and (b) endorsing other transactions. This data structure is known as the *UTXO tangle.*

Endorsements are an atomic part of the transaction, signed by the same entity that moves tokens. Their position in the DAG is fixed by the token holder; the transaction cannot be "reattached" by anyone else.

Here is our interpretation of the UTXO tangle (see figure 2):

- Each vertex of the UTXO tangle represents a UTXO transaction that **consumes** outputs from other vertices/transactions and may optionally **endorse** a chosen set of other vertices.
- Each vertex represents a separate ledger state resulting from mutating the consolidated state. The consolidated state is consistent among all inputs of the transaction, including endorsed ledger states.
- The UTXO tangle represents multiple ledger states. Any two ledger states (vertices) in the UTXO tangle may or may not conflict. The past cone of each vertex is conflict-free and represents the history of ledger updates, transforming one consolidated ledger state into another.
- By adding a vertex to the UTXO tangle, one creates another version of the ledger, a new "fork" of the state. Thus, the growth of the UTXO tangle involves creating forks and consolidating them.

In the UTXO tangle, we are specific about who can create a new vertex and add it to the tangle: the token holder and no one else (the entity with private keys). The **token holders are the sole writers to the ledger**. Thus, each vertex of the UTXO tangle bears the identity of the issuer in the form of a signature on the vertex data.

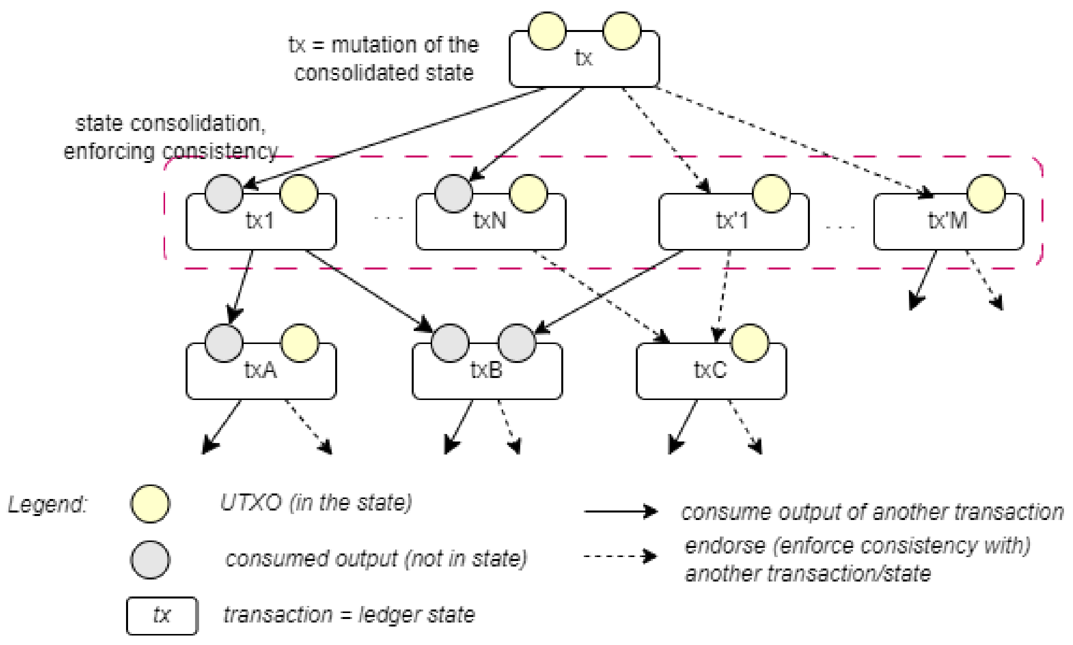

tx = mutation of the consolidated state
tx
state consolidation, enforcing consistency
tx1
txN
tx'1
tx'M
txA
txB
txC
Legend:
UTXO (in the state)
consumed output (not in state)
tx
transaction = ledger state
consume output of another transaction
endorse (enforce consistency with) another transaction/state

Figure 2: UTXO tangle

2.3 Cooperative consensus principle

This section summarizes the main principles behind cooperative consensus on the UTXO tangle and the "biggest coverage rule" (or "heaviest state rule").

Token holders, the sole entities participating in the protocol, exchange transactions. Each token holder builds their own UTXO tangle based on the received transactions.

Token holders share a common interest in reaching a consensus on the state of on-ledger assets. Each transaction includes a reward in new tokens (inflation) or fees. The liquidity of the native token on the ledger drives the process toward profit-seeking behavior.

The token holder creates a new transaction by consolidating chosen non-conflicting ledger states, represented by vertices in their local UTXO tangle. The new transaction consumes outputs of those transactions (inputs) and/or endorses them, creating a new valid ledger state.

The token holder has a random and asynchronously changing set of possible choices for consumption and endorsement targets, as new transactions from other token holders arrive unpredictably. The token holder must select a non-conflicting subset of all possible inputs to produce a transaction.

The token holder optimizes the selection process to create the best possible outcome for the new transaction. This approach transforms the choice of consolidation targets into a dynamic**optimization problem**, aiming to find a non-conflicting set of inputs that aligns with an objective function. Each token holder continuously optimizes a deterministic objective function that reflects its position relative to the global consensus target, which periodically advances. There must be guarantees against falling into a local optimum.

Even without considering the continually changing set of possible choices, the optimization is an NP-complete problem. The UTXO tangle graph is built in real-time, has unbounded depth, and potentially contains many conflicting vertices. To adhere to real-time constraints, even with relatively low transactions per second (TPS), practical application requires heuristics. Moreover, the token holder has only a limited time to construct the new transaction; falling behind the natural growth of the UTXO tangle and the pace of the majority of other token holders increases the risk of the transaction becoming orphaned.

We introduce a special metric of the transaction (vertex) on the UTXO tangle called **ledger coverage** (or just **coverage**). The value is deterministically calculated from the past cone of the vertex as the sum of amounts of outputs, which are consumed by the past cone of the transaction in the given **baseline ledger state**, an "interim state snapshot", which is always defined for the special kind of transactions called *sequencer transactions*. Details in section 7.2.Branches.

The *ledger coverage* metric represents how close the transaction is to the desired ledger state of the global consensus. We use this metric as the objective function of the optimization problem the token holder will be solving within a fixed time interval called a **slot**. The maximum value of ledger coverage is equal to the $2 \cdot totalSupply$ of the ledger state, i.e., it has an upper bound. Intuitively, several non-conflicting ledger states with maximal ledger coverage (i.e., with all baseline outputs consumed) will be identical. Therefore, a strong growth in ledger coverage suggests convergence of different local ledger states to a consensus ledger state. See below fig. 3 and section 8.Convergence.

The token holder maximizes ledger coverage of the newly issued transaction. This is known as the **biggest ledger coverage rule** (or **heaviest state rule**). It is analogous to the "longest chain rule" in Proof of Work (PoW) consensus.

There is a guarantee that, within the boundaries of one slot and until maximal coverage is reached, a non-empty set of token holders can keep advancing with greater ledger coverage by bringing their controlled tokens to the UTXO tangle with new transactions. This process will continue as long as these token holders are incentivized to create new transactions.

The Nash equilibrium among token holders arises from each token holder wanting their transaction to be settled in the heaviest future ledger state possible (i.e., with the highest ledger coverage possible). Therefore, maximizing ledger coverage is the optimal strategy. Deviating from this strategy increases the chances that a transaction will not be chosen by other token holders and will eventually be orphaned.

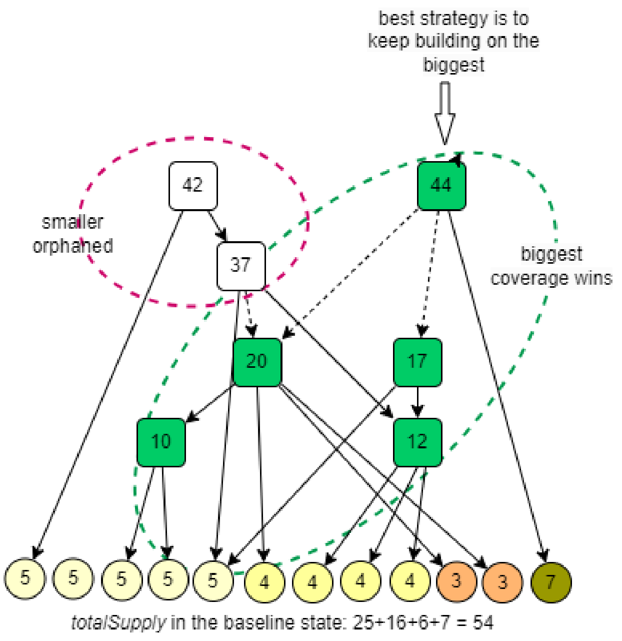

best strategy is to keep building on the biggest
smaller orphaned
biggest coverage wins
42
37
44
20
17
10
12
5
5
5
5
5
4
4
4
4
3
3
7
totalSupply in the baseline state: 25+16+6+7 = 54

Figure 3: Ledger coverage

The process is **cooperative** because only the token holder who controls tokens can bring the corresponding ledger coverage to the UTXO tangle. Thus, each token holder can only ensure growing coverage for their transaction by consolidating the states of other token holders, i.e., by cooperating with them.

The cooperative consensus is a **Nakamoto consensus** because it occurs among an unbounded and completely permissionless set of consensus-seeking writers to the ledger (token holders). These entities make decisions without relying on any global dynamic knowledge, such as a global consensus on weight distribution of voting entities, a committee of voters/quorum, stakes, a coordinator, or similar mechanisms. They adhere to the consensus rule, which is computed from their local copy of the UTXO tangle.

2.4 The big picture

In a practical setup, the cooperative consensus principle, along with the biggest ledger coverage rule, must be enhanced with many details to be applicable in an ever-growing UTXO tangle. Sections below will cover these details, but here is a broad overview.

All token holders are allowed to build chains of transactions on the ledger. While building a chain, the token holder generates inflation proportional to the capital held on-chain. This is the fundamental incentive for passive capital holders to actively contribute to the consensus.

A special class of active token holders, known as **sequencers**, plays a distinct role in the system while remaining permissionless. Each sequencer builds a chain of special transactions called **sequencer transactions**, each transaction consolidating as much ledger coverage as possible. During the slot, all sequencers work together to cover the baseline ledger state of the previous slot as best they can, given time, performance, and communication delay constraints. A sequencer's success depends on its performance characteristics: its connectivity to other sequencers and the amount of on-ledger capital it can bring as ledger coverage to other sequencers.

Non-sequencer token holders have the opportunity to **tag along** their transactions with the sequencer chains of their choice, offering a small or even zero fee (a "bribe") to incentivize the sequencer to consume the **tag-along output**.

Inflation mechanisms and tagging-along make sequencing a profit-seeking activity.

Large token holders can also **delegate** their tokens to participate in chain building by sequencers, receiving inflation rewards in return. "Lazy whales" who do not delegate face the opportunity cost of not generating inflation from their holdings.

At the slot boundary, sequencers may produce a special kind of sequencer transaction called a **branch transaction** or **branch** to enter the next slot and win the **branch inflation bonus**. Each branch transaction must select one branch transaction from the previous slot and consume its special output, known as the **stem output**. As a result, all branches in the same slot are conflicting transactions, and only one can be included in the ledger state. These constraints aim to guide the UTXO tangle towards a single path of branches, with side branches quickly becoming orphaned.

At the slot boundary, sequencers compete rather than cooperate. However, to reach the competition, they must cooperate during the slot.

Ledger states corresponding to branches are committed to the databases maintained by nodes connected to the network of peers.

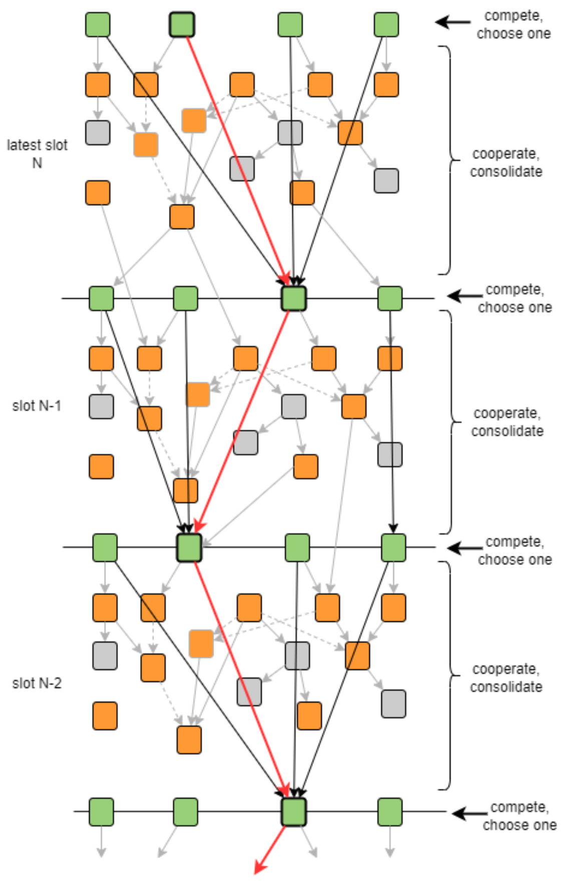


Figure 4: Big picture

2.5 Characteristics of the architecture

To conclude the introduction, we will highlight the main characteristics of the proposed architecture:

- **Fully permissionless** writing to the ledger. Token holders, who are the only ones authorized to write to the ledger, can issue ledger updates without requiring any permitting procedures. There is no need for committees or voting processes.
- **Leaderless determinism**. The system operates without a consensus leader or block proposers, providing a more decentralized approach.
- **Asynchrony**. The architecture relies on only weak assumptions of synchronicity.
- **No need for a global knowledge** of the system state, such as composition of the committee, assumptions about node weights, stakes, or other registries.
- **1-tier trust assumptions**. Only token holders are involved in the process, as opposed to the multiple layers of trust required in other blockchain systems such as PoW (which includes users and miners) and PoS (which includes at least users, block proposers, committee(s)).
- **Absence of a deterministic global state**. Due to the non-determinism of the ledger account (a set of outputs), it cannot be used as a single global liquidity pool.
- **Parallelism at the consensus level**. Assets converge to their final state in parallel.
- **Massive parallelism** at the node level. All transactions are validated in parallel at each node.

3 UTXO tangle

3.1 UTXO transaction model

The Proxima model of the UTXO ledger is fundamentally similar to the well-known UTXO models of Bitcoin, Cardano (EUTXO)[8], and others. However, it introduces *endorsements* as a novel feature, which extends the UTXO model with additional validity constraints without altering its core principles.

The main concept of the UTXO ledger model is **output** or **UTXO**. A set of all outputs constitutes a **ledger state**. Each output O is uniquely identified on the ledger by **output ID**. It has two mandatory properties (among others): **amount** and **lock**. The *amount*, denoted as $amount(O)$, is an amount of native tokens **locked** in the output O. The *lock* specifies conditions which must be satisfied on the transaction in order to **unlock** the output (**spend** or **consume** it). In the most common form, *lock* is a (target) address, a hash of some public key, which corresponds to the secret private key.

The main concept of the UTXO ledger model is the **output** or **UTXO**. The complete set of all outputs constitutes the ledger state. Each output O is uniquely identified on the ledger by its **output ID** and has two mandatory properties: **amount** and **lock**. The amount, denoted as $amount(O)$, represents the quantity of native tokens locked in the output O. The lock specifies the conditions necessary to unlock the output, allowing it to be spent or consumed. The most common lock is target address (hash of public keys corresponding to secret private keys). Other examples of locks in the Proxima ledger include chain locks, stem locks, delegation locks, tag-along locks, and more.

Each transaction T includes:

- A unique identifier on the ledger.
- A set of outputs **produced** by the transaction: $out(T) = \{O_0^T, \ldots, O_{n-1}^T\}$, referred to as **produced outputs** of the transaction T. The producing transaction and output index constitute the unique *output ID.*
- A set of outputs **consumed** by the transaction:: $in(T) = \{O_{i_a}^{Ta}, \ldots, O_{i_z}^{Tz}\}$. These consumed outputs' IDs are also known as *inputs* of the transaction T. The notation $T' \leftarrow T$ denotes that transaction T consumes an output of T'.
- Optionally, up to N endorsements of other transactions: $endorse(T) = (T_0, \ldots, T_k), k < N$. The notation $T' \Leftarrow T$ denotes that transaction T endorses transaction T'.
- a **timestamp**, representing the **ledger time** of the transaction: $timestamp(T)$, roughly equivalent to block height in blockchains. This will be discussed in section 4. Nodes. Ledger time.
- contains **input unlock data** and, optionally, other **auxiliary data**: $unlock(T)$
- contains **signature** (including public key) of all the data above. It is denoted as $senderID(T)$

Therefore, a transaction T is a tuple: $(in(T), out(T), endorse(T), timestamp(T), unlock(T), senderID(T))$.

A transaction is considered valid if it satisfies all validity constraints, including valid inputs, endorsements, produced outputs, transaction-level constraints, and past cone constraints. Equivalently, a transaction T is valid when the predicate $valid(T)$ renders *true*.

3.2 Transaction graph. Past cone

Consider a set of transactions $L = \{T_0, \ldots, T_n\}$. Let L be a vertex set of a graph with edge set consisting of all pairs of transactions from L, that are connected either by a consumption relation $\leftarrow$ or endorsement relation $\Leftarrow$:

$$edges(L) = \{(T_a, T_b) \| T_a \leftarrow T_b \vee T_a \Leftarrow T_b\}$$

. This naturally forms a *directed graph* in the set of transactions L.

Assume the directed transaction graph is also *connected* and *acyclic*, i.e. it is a **transaction DAG** (Directed Acyclic Graph). Then the *past cone* of the transaction $T \in L$ is a subset $past(T) \subset L$, where each transaction $T' \in past(T)$ is reachable through the relations $\leftarrow$ and $\Leftarrow$ in the transaction DAG.

[8]The Extended UTXO Model: https://omelkonian.github.io/data/publications/eutxo.pdf

The set of all outputs produced or consumed by transactions of L is denoted as $UTXO(L)$:

$$UTXO(L) = G_L \cup \bigcup_{T \in L} out(T)$$

where G_L is an assumed *baseline state*. Special cases of *baseline state* include *genesis* or *snapshot* (see section 3.3).

We say that two distinct transactions $T_a, T_b \in L$ engage in **double-spending**, if there's an output $O \in UTXO(L)$, that is consumed by both T_a and T_b, i.e., if their consumed inputs intersect:

$$in(T_a) \cap in(T_b) \neq \emptyset$$

A set of transactions L is *consistent*, if it does not contain a double-spending pair of transactions. Similarly:

- Any two transactions $T_a, T_b \in L$ are *conflicting*, if union of their past cones contains a double-spending pair.
- A set of transactions $\{T_a, T_b, \dots\} \subset L$ is *consistent*, if union of their past cones does not contain a conflicting pair of transactions
- A transaction $T \in L$ has *valid past cone*, if its past cone $past(T)$ is *consistent*

3.3 Ledger. Baseline state

A consistent set of transactions $L = \{T_1, \dots, T_n\}$ is called a *ledger*. Each ledger L has a corresponding set of outputs denoted as $S_L = \{O_1, \dots, O_m\}$, known as a **ledger state**.

Usually, each output in the ledger state is associated with a transaction that produced it. As an exception, we identify a set of outputs in L, called the **baseline state**, which does not have corresponding producing transactions in the ledger. This set of outputs is denoted as G or G_L. A valid ledger can be constructed by incrementally building it from a baseline state G by adding transactions.

Any consistent ledger state may serve as a baseline to develop a ledger on top of it. We can define a valid ledger recursively:

- The empty set of transactions is a valid ledger with a corresponding ledger state equal to the baseline state G: $S_\emptyset = G$.
- Suppose L is a valid ledger with the ledger state S_L. We can append new transaction T to the ledger if:
  - All consumed inputs exist in the ledger state, i.e. $in(T) \subset S_L$
  - All endorsed transaction are on the ledger, i.e. $endorse(T) \subset L$
  - The transaction T is valid: $valid(T)$ renders *true*
  - The resulting set of transactions $L \cap \{T\}$ is consistent (meaning the consolidated past cone is free of conflicts/double spends)
- The new ledger $L' = L \cap \{T\}$ is a valid ledger by definition. The corresponding ledger state $S_{L'}$ of the updated ledger L' is constructed by removing consumed outputs from S_L and adding all produced outputs to it: $S_{L'} = S_L \setminus in(T) \cup out(T)$

The valid ledger L and its state S_L is the one which can incrementally be constructed from the baseline ledger state G by adding valid transactions, that are consistent with the existing ones. This construction ensures the ledger L is always connected and acyclic, i.e. a transaction DAG. Note that each valid ledger is free of double consumption of outputs. i.e. it is consistent.

By the definition, outputs of the genesis/snapshot contain all tokens defined in the ledger, and the sum of the amounts equals to the total supply of tokens:

$$totalSupply = \sum_{O \in G} amount(O)$$

3.4 Multi-ledger

Let's say we have a set of transactions $U = \{T_0, \dots, T_N\}$ as a vertex set with the same genesis G. We define U as an **UTXO tangle** if the past cone of each transaction $T \in U$ is a valid (consistent) ledger L_T.

A UTXO tangle may contain conflicting transactions, which contain *double-spends* in their past cones. In such cases, the corresponding ledger states would be conflicting. Thus, a UTXO tangle is a data structure that may contain multiple versions of ledgers, both conflicting and non-conflicting. This characteristic is known as *multi-ledger*.

3.5 Transaction validity constraints

Each transaction is subject to enforced validity rules. The validity of the transaction T is determined by the predicate $valid(T)$. The validity of the transaction is equivalent to the validity of the ledger L_T and the corresponding ledger state S_{L_T}. In general, the predicate $valid(T)$ is a function of T and its past cone.

3.5.1 Validity of the past cone

Validity of a transaction (vertex in the UTXO tangle) includes ensuring that its past cone is both (a) conflict-free (refer to Transaction graph. Past cone) and (b) forms a *connected graph* (also known as *solid*). To determine whether a transaction's past cone is conflict-free and connected, we must conduct a search with an *unbounded depth*. This means there is no theoretical limit to how far back in the UTXO tangle a double spend between two past cones can occur. The UTXO tangle is a potentially unlimited data structure, akin to the tape of a Turing machine.

In practical design, we work only with bounded data structures, limiting the scope of transactions, outputs, and baseline ledger states. This approach avoids the need for an unbounded traversal when calculating the validity predicate.

3.5.2 Transaction level validity constraints

By *transaction level constraints*, we mean constraints that do not depend on the entire past cone but cannot be solely attributed to individual outputs. These constraints include:

- Checking syntactical transaction composition.
- Enforcing token balance between inputs and outputs, including inflation constraints.
- Verifying validity of *timestamp* and ledger time constraints.
- Checking signature validity.
- Each output must have mandatory constraint scripts, such as specifying the token amount and including lock.
- Validating output-level constraints: this includes checking the scripts of consumed and produced outputs for correctness.
- Enforcing other static constraints: these may include ensuring valid references to optional constraint libraries and other checks not explicitly discussed here.

3.5.3 Validity of outputs

There are only a few constraints enforced at the transaction level. The absolute majority of ledger validity constraints are expressed through the use of **validation scripts**.

Each output O is composed of **constraints** (also known as **constraint script** or **validation script**). These scripts form a vector $O = (c_1, \dots c_n)$. Each script or constraint c_i, when evaluated, results in *true*, if the constraint is satisfied in the given validation context, or *false* if it is not satisfied (violated)[9].

[9]The *validation script* is computationally equivalent to Bitcoin Script or similar engine. It is intentionally non-Turing complete. We won't delve on this deep and bread topic here

The output O is considered **valid in the validation context**, if all of its validation scripts $\{c_i\}$ renders *true* when run in that context.

Each output and each of its validation scripts can be evaluated in two different contexts: *as produced output* and *as consumed output*.

- When evaluated as a *produced output*, the script c_i of output O is run in the context of the producing transaction $T_{prod} : O \in out(T_{prod})$. In other words, the script is provided with the data of the producing transaction T_{prod} as parameter, along with the output itself.
- When evaluated as a *consumed output*, the script c_i of output O is run in the context of the consuming transaction $T_{cons} : O \in in(T_{cons})$. This means that, along with the output itself, the script is provided with the data of the consuming transaction T_{cons} as parameter.

  A consumed output can be validated without requiring the corresponding transaction that produced it. This means the outputs of the transaction can be validated independently of its past cone.

  Validation scripts of the consumed outputs typically involve checking their output unlock data, which must be provided by the consuming transaction. For instance, this includes verifying whether the address data in the lock constraint matches the signature $senderID(T)$ in the consuming transaction T.

The ledger definition contains a static library of standard scripts that are included upon ledger genesis. For example, the standard *chain constraint* is a library script, *addressED25519* lock script is a standard script, and there are many others.[10].

The scripting capabilities and the ability to compose different scripts as "Lego blocks" in the outputs make Proxima UTXO ledger a programmable UTXO ledger[11].

4 Nodes. Ledger time

We envision a system where the only participating entities are token holders. Token holders produce transactions, exchange them with each other, and maintain converging consensus on the UTXO tangle. In our narrative, the Proxima distributed ledger is maintained by token holders rather than nodes.

Of course, real token holders (humans or organizations) require infrastructure support, such as nodes connected to the network. However, in Proxima, nodes are not independent entities; they do not produce transactions, express opinions about transactions, or maintain consensus on the ledger state.

The primary function of the network of nodes is to provide distributed infrastructure for token holders, who are responsible for achieving consensus on the ledger. Proxima nodes do not serve the roles of *miners* (as in proof-of-work systems) or *validator* (as in proof-of-stake systems).

The main functions of a Proxima node are as follows:

- **Maintain Connections with Peers**: nodes establish and sustain connections with other nodes in the network.
- **Gossip**: nodes broadcast (relay) newly received transactions to their peers in the network.
- **Synchronize and maintain local UTXO tangle**: nodes synchronize with the network to maintain a valid local copy of the UTXO tangle. They verify the validity of each transaction before adding it to their local copy.
- **Commit ledger states** for each branch transaction (see section 7.2) into the local multi-ledger database and provide read access to the ledger state for each branch. This function may include database cleanup and state pruning.

[10]Note, that each script is taking its parameter data from the context only. For example running the script $addressED25519(0x97459a12301950535caa5115af5edbc04ead272c25880c967ee522dc11bc8193)$ in the consumed output means checking if the output which is locked in this particular target address is unlocked in the context of the consuming transaction

[11]In the Proxima prototype we use simple functional language *EasyFL*: https://gitub.com/lunfardo314/easyfl for output constraint scripting. Almost all transaction validity constraints, needed for the Proxima consensus on the UTXO tangle, are expressed as *EasyFL* formulas

- **Provide stored raw transactions** to other nodes upon request[12].
- **Provide read access** to the UTXO tangle, ledger states and transactions store for sequencers, other token holders and users.
- **Synchronize local clocks**: nodes synchronize their local clocks with the **ledger time** to ensure approximate synchronicity of local clocks across the network.
- **Protect against spamming attacks**: nodes are responsible for protecting themselves and their peers from potential spamming attacks to maintain network integrity (see Spamming prevention).

Nodes communicate by exchanging raw transaction data. **Transactions are the only type of message shared between nodes**.

Token holders access the network by examining their local copy of the UTXO tangle, either from their own nodes or through public services (typically for-profit). Token holders review their local UTXO tangle to make decisions about producing and sending transactions to the network. Nodes themselves do not have any intention or ability to issue transactions.

There is an assumed global world clock (such as an atomic clock) that serves as the time reference shared by all nodes in the network. This assumption is practical as long as the network remains on the planet Earth. We cannot guarantee perfect synchronization of local clocks across all nodes, but nodes are expected to keep their clocks reasonably close to the global reference time.

Node operators (token holders or those who profit from running public nodes) are incentivized to keep their nodes' clocks synchronized as closely as possible: significant deviations between local clock and the *ledger time* on the transaction will lead to higher risk of transactions to be orphaned.

Deviation of local clocks from the global time reference is expected to be normally distributed around zero with a small standard deviation. Network can tolerate significant differences between local clocks of some nodes, as long as the clocks of the largest token holders' nodes are roughly in sync.

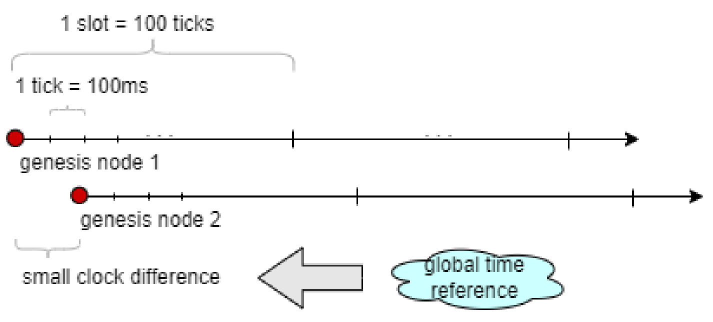
The ledger uses a logical clock called **ledger time**. The axis of ledger time is divided into **slots**, beginning with slot zero at genesis. Each slot is further divided into **ticks**, with the number of ticks per slot being a static parameter of the ledger set at genesis[13].

Each transaction T includes a $timestamp(T)$, which is a ledger time value comprised of a pair::

$$timestamp(T) = (slot(T), ticks(T))$$

where $slot(T)$ represents the slot number on the ledger time axis that the transaction T belongs to, and $ticks(T)$ is the number of ticks in that slot. The timestamp of the transaction is subject to transaction level validity constraints: the **timestamps of inputs and endorsed transactions must be strictly earlier than the transaction timestamp**[14] by at least a specified number of ticks, known as the **transaction pace** (which is a static ledger parameter). The *ledger time* is a rough equivalent of the block height in blockchains.

The relationship between the ledger time scale and real time scale is established by another static ledger parameter: the **tick duration**[15]. The real time duration of a slot is derived from these parameters as $durationOfTheSlot = tickDuration \cdot ticksPerSlot$. In a practical setup, slot duration is expected to be around 10 to 30 seconds, though this may be subject to experimentation and fine-tuning.



The absolute **genesis time** (expressed as Unix time) is another static ledger constant

[12]Note that in the prototype implementation, the function of the transaction store is architecturally separated from the node

[13]in the prototype node one slot has 100 ticks

[14]This implies that sorting transactions by timestamp results in a topological sorting of the DAG

[15]In the prototype, tick duration is set to 100 milliseconds, making the slot duration 10 seconds

that is set when the ledger starts. The timestamp of each transaction can be translated to its real-time value using the three static ledger constants described above: *genesis time*, *tick duration*, and the *number of ticks per slot*.

The node's responsibility is to synchronize ledger time of incoming transactions with its local clock. This is achieved by **delaying** the processing of each transaction with a timestamp that translates to a value that precedes the node's current local clock time. The transaction is delayed until the local clock reaches the timestamp, ensuring the node remains synchronized with the ledger time.

If a node with a local clock running ahead of schedule sends a transaction to a peer node with a local clock running behind, the receiving node will perceive the transaction as arriving from the future. Consequently, the transaction will be held in a "calm down room" until it reaches its scheduled time according to the receiving node's local clock. This behavior is expected of the majority of nodes, ensuring an approximate synchronicity of local clocks throughout the network.

This behavioral assumption is considered realistic because producing a transaction either ahead of or behind the network's average perceived clock can increase the chances that the transaction will be orphaned. This potential for orphaning incentivizes token holders to keep their transactions in line with the network's time standards, ensuring better compatibility and synchronization across the network.

There's no strict need for the lower bound of ledger time of incoming transactions, except the situation of preventing certain type of spamming attacks. For this reason it may be practical to introduce certain past time horizon for too old transactions to be rejected immediately.

5 Chains

The chain constraint allows us to construct non-forkable, unlimited-length sequences of transactions on each ledger in the UTXO tangle, called *chains*[16].

Chains play a crucial role in Proxima ledger and consensus. Outputs in the UTXO ledger are transient, one-time assets with a limited lifetime until they are consumed by a transaction. The *chain* transforms a sequence of outputs into a permanent, non-fungible asset with unlimited lifetime. This provides an important mechanism for continuity and permanence in the ledger.

The **chain constraint** is a standard validation script that can be attached to any output, thereby transforming it into a **chain-constrained output**. Each *chain constraint* is tagged with a unique **chain ID**. A chain is a sequence of chain-constrained outputs that share the same *chain ID*.

When the chain constraint script evaluates the consumption of an output, it deems a consuming transaction invalid unless it produces a continuation of the chain with the same *chain ID*. A chain-constrained output O with a specific *chain ID* can be consumed in a transaction T only if one of the following conditions is met:

- The transaction produces exactly one chain-constrained output, known as a **successor**, with the same *chain ID*. In this case, the consumed output is referred to as a **predecessor**.

- The chain is explicitly terminated, in which case the transaction does not produce any **successor** output.

Thus, the chain constraint enforces a chain of outputs on the UTXO tangle, identified by the chain ID as a unique and

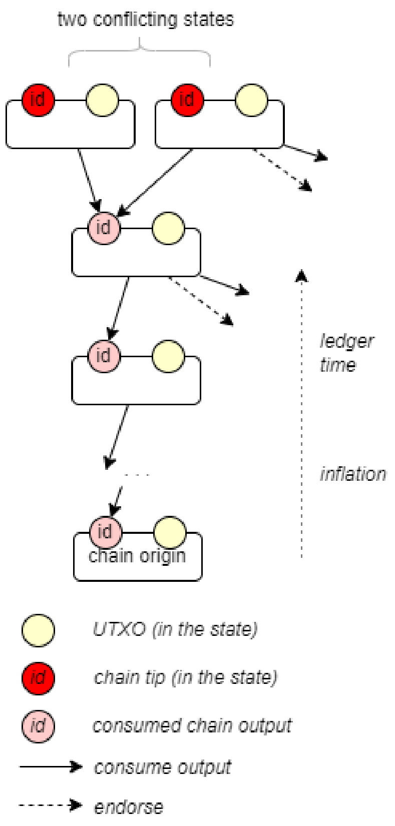


Figure 5: Chain

[16]The concept, known as the **chain** on the UTXO ledger, was first introduced in the IOTA's Stardust ledger specification as **Alias output**

persistent identifier for the chain's lifetime[17].

If the chain with a given *chain ID* is present in the ledger state S_T, then S_T will contain exactly one chain-constrained output for this *chain ID*, called **chain's tip**. We will use the following notation for chain's tip:

$$tip_{S_T}(id)$$

is the unique output, which represents chain with chain ID id in the ledger state S_T.

The corresponding chain-constrained output, the tip of the chain, is immediately retrievable in the state using the chain ID. This makes each chain a *non-fungible asset*, which can be utilized for various purposes, such as implementing NFTs. In Proxima, chains are primarily used to implement **sequencer milestones** and **branches** (as discussed later).

Note that if two different transactions attempt to *extend* the same chain-constrained output into their respective successors, this will result in double-spending. In other words, these transactions will represent two conflicting ledger states, creating a fork in the ledger. **Within one consistent ledger, chain forks are not possible**.

6 Inflation. Incentives

In Proxima, inflation serves as the basis for an incentive mechanism and is closely tied to the concept of chains. This design aims to reward participants in the network while maintaining the ledger's integrity and promoting healthy economic dynamics within the ecosystem. Proxima enables the controlled generation of new tokens over time, offering a clear and predictable structure for incentives.

By incorporating inflation within the framework of chains, Proxima incentivizes participants to engage in network activities that contribute to the overall security and reliability of the ledger. This approach encourages token holders to participate actively in consensus and transaction validation, as they can benefit from the rewards associated with the inflationary process.

Let's define the maximal amount of inflation allowed on the successor output O' in the chain:

Given the predecessor output O with a timestamp t, and its successor output O' with a timestamp $t + \Delta t$, the maximal inflation on the output O' is proportional to the amount of capital locked in the output O and the time interval Δt, known as chain's pace. It is expressed as the following formula:

$$inflation(O') = \begin{cases} \Delta t \cdot amount(O) \cdot I_t & \text{if } ticks(t+\Delta t) \neq 0 \text{ and } \Delta t \leq W \\ BB & \text{if } ticks(t+\Delta t) = 0 \text{ and } \Delta t \leq W \\ 0 & \text{if } \Delta t > W \end{cases}$$

Here:
- W is a ledger constant called **inflation opportunity window**
- I_t is a ledger constant for the **inflation rate**, adjusted for the halving epoch of the predecessor's ledger time[18]
- BB is **branch inflation bonus**. It is a **randomized value** calculated for branch transactions only (transactions which are on the slot edge) (see section 8.2.Rich chain bias. Meta-stability. Random branch inflation).

The chain constraint script enforces this rule, ensuring that the maximal inflation on the output does not exceed the calculated amount.

The input-output balance validity check for a transaction T ensures that the sum of the produced output amounts does not exceed the sum of the input amounts plus the total inflation from all chain-constrained produced outputs. This check is essential to maintain the integrity of the ledger and to enforce the inflation limits set by the chain constraints. Given a transaction T, the validity check can be

[17] For readers familiar with the structure of the Stardust UTXO ledger implemented in IOTA: the *chain constraint* serves the same function as the *Alias output type* in Stardust. The difference is that the Alias output type is hard-coded (not a script) and implements many other functions, while in Proxima, it is implemented purely as a chain constraint, which can later be combined with other scripts as a "Lego blocks" in the same output

[18] in the prototype defaults *inflation opportunity window* is 6 slots (1 minute). The inflation rate roughly corresponds to annual 60% on the total supply the first year, then halving each year next 5 years

expressed as follows:

$$\sum_{o \in out(T)} amount(o) \leq \sum_{o \in in(T)} amount(o) + \sum_{o \in out(T)} inflation(o)$$

This inflation mechanism creates incentives for token holders to participate in chain creation and maintenance, as they can potentially gain additional tokens over time as long as they to issue at least one chain-constrained output per *inflation opportunity window W*.

Total inflation on the Proxima ledger is directly related to the activity of token holders, incentivizing participation and engagement in the network. This incentives mechanism functions through three main channels:

- token holders can choose to **run a sequencer** while seeking consensus. By doing so, they build sequencer chains with inflation and potentially benefit from transaction fees.
- token holders may **delegate their capital to a sequencer**, effectively lending their resources to support the sequencer's operations. In return, they may receive a share of the inflation generated by the sequencer's activities.
- token holders can **tag-along** their transactions to a sequencer's chain, aligning their activities with the sequencer's operations. This allows them to benefit from the sequencer's efficiency and potential inflation.

The inflation is proportional to the amount of capital being moved around in the ledger and is subject to halving rules that limit inflation over time. This model mirrors traditional financial concepts such as *return on capital* and *time-is-money*, where more active and engaged token holders are rewarded. Conversely, passive holders or "lazy whales" who do not actively participate in the network's operations do not receive the same level of rewards. This creates an equitable system that encourages active participation and contribution to the network's growth and stability.

7 Sequencing

The chain-constrained output that contains a special *sequencer constraint script* is referred to as a **sequencer milestone**. The transaction that produces a sequencer milestone is called a **sequencer transaction**. A **sequencer chain** is a chain of sequencer milestones or transactions. **Sequencing** is the process of building sequencer chains, where transactions in these chains endorse sequencer transactions on other sequencer chains.

Token holders engage in creating transactions and maintaining sequencer chains with the goals of (a) generating inflation, which can yield profits, (b) earning tag-along fees from transactions that attach themselves to the sequencer's chain and (c) contributing to the overall consensus on the ledger.

Sequencing is essential for keeping the network active and functional. It plays a role similar to the block production performed by miners and validators in other distributed ledger systems[19].

7.1 Sequencers

Life and progress on the Proxima ledger are entirely dependent on the activity and behavior of its token holders, making it a very "democratic" system.

Nonetheless, there is substantial evidence that distributed ledgers tend to converge towards a high concentration of capital, where a small number of entities control the vast majority of assets on the ledger[20]. Prominent examples include major exchanges and custodians of tokens that manage large amounts of on-chain assets. This appears to be a fundamental and unavoidable trend.

[19]Sequencing has strong similarity with the sequencing for L2 chains/rollups and in IOTA Smart Contracts (https://files.iota.org/papers/ISC_WP_Nov_10_2021.pdf), hence the name. The Proxima sequencer, as an option, can be a *centralized or distributed sequencer* in the sense of L2 chains, so the name is not misleading

[20]see for example *How centralized is decentralized? Comparison of wealth distribution in coins and tokens*: https://arxiv.org/abs/2207.01340

With Proxima, instead of worrying about "fairness" of the capital distribution or attempting to create a more equitable world, the design embraces practicalities. It assumes that a significant concentration of tokens in the hands of a few is normal. Consequently, major token holders (often "whales" but primarily large exchanges and other custodians) will eventually command a substantial share of the capital, which entails a substantial amount of *skin-in-the-game*, *commitment*, and *trust* in the distributed ledger (more about this in 9.3.General security. Social consensus).

Proxima intentionally distinguishes a particular class of token holders known as **sequencers**, who are capable and willing to take on a significant portion of the shared ledger's responsibility in exchange for certain privileges compared to their non-sequencers. A sequencer can be any entity that controls more than a minimum amount of capital. This minimum amount required for sequencing is defined as a constant at the genesis of the ledger[21]

Any token holder with sufficient capital can become a sequencer by issuing sequencer transactions/milestones and maintaining persistent sequencer chains. There are no registration, on-boarding, staking, or other preliminary steps (even "permissionless") as seen in many PoS networks. This allows for a direct and straightforward approach to becoming a sequencer.

Sequencers, by running sequencer chains, will generate returns on their own and leveraged capital in the form of inflation, similar to any other token holder running a chain. In addition to these returns, sequencers also have the opportunity to *receive branch inflation bonuses, collect tag-along fees, and earn delegation margins* in exchange for their services to the network. These incentives provide sequencers with additional income streams beyond standard inflation, rewarding their active participation in and support of the network.

Issuing sequencer transaction is **completely permissionless and pseudonymous**, allowing any token holder with enough capital to create and maintain sequencer chains without any registration or on-boarding. By distinguishing sequencer transactions from other transactions, the system incentivizes specific behaviors from large token holders that are beneficial for all participants in the network. This approach aligns the interests of sequencers with the overall health and stability of the ledger, fostering a more robust and efficient ecosystem.

Creating a sequencer milestone involves adding a standard *sequencer validity constraint script* to the chain-constrained output. This sequencer constraint enforces specific validity rules on the chain output and chain-constrained transactions:

- Each chain-constrained output must have a minimum amount specified by the script.
- Only sequencer milestones are allowed to endorse other transactions. Ordinary users cannot endorse other transactions with their transactions.
- Sequencer transactions are only allowed to endorse other sequencer transactions within the same slot. Cross-slot endorsements are prohibited.
- Only sequencers can create **branches**. A sequencer transaction becomes a **branch transaction** when its timestamp is on a slot boundary (i.e. when $ticks(T) = 0$). Branch transactions have additional constraints and represent a baseline ledger state. All participants eventually agree on a single sequence of baseline ledger states by orphaning alternatives (see section 7.2). The incentive for the sequencer to issue branch transactions is branch inflation bonus.
- Each sequencer transaction must have a **deterministically defined and explicitly or implicitly referenced baseline state**. A sequencer milestone either has a sequencer predecessor on the same slot or endorses another sequencer milestone on the same slot (see section 7.3 for details).

These constraints are designed to ensure a collaborative and converging process among sequencers, promoting a consistent and coherent ledger state.

7.2 Branches

[21] For example, in the prototype implementation, the minimum amount on the sequencer chain is set to 1/1000 of the initial total supply, allowing for a maximum of 1000 sequencer chains. The optimal value for this constant is an open question, particularly in the context of other ways of contributing to the capital of the sequencer, such as *delegation*

A sequencer transaction T_b that is on the slot edge (i.e., with $ticks(T_b) = 0$) is known as a **branch transaction** or **branch**.

Branch transactions are restricted from endorsing other sequencer transactions, because cross-slot endorsements are invalidated in such cases.

Branches, as a special form of sequencer transactions, serve several important purposes.

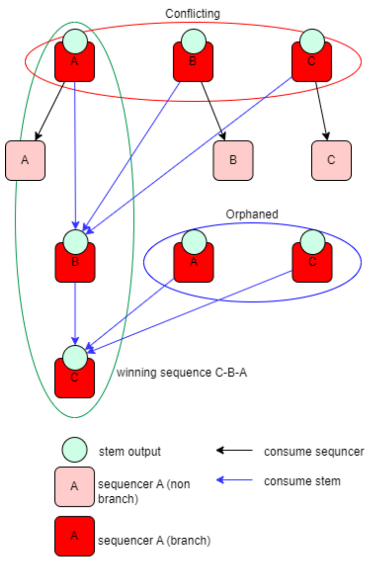


Figure 6: Tree of branches

7.2.1 Purpose: committed ledger states

The primary purpose of branch transactions is to enable nodes to commit the ledger state corresponding to each branch transaction into their local multi-ledger databases (as discussed in Nodes. Ledger time). This way, each node maintains the ledger states for all branches it has encountered.

Each branch transaction serves as a checkpoint with the Merkle root for the ledger state, so it establishes a clear point of reference for the committed ledger state at the beginning of each slot.

Most of the ledger states eventually become orphaned along with the branches. This creates a need for cleanup and pruning processes to manage the size and complexity of the multi-ledger database effectively.

The *multi-state database* is a database with multiple overlapping sparse Merkle trees (tries).

- Each branch transaction in the network is associated with a Merkle root, which is a cryptographic hash representing an immutable ledger state of the branch.
- The database structure enables rapid checking of whether a specific output is present in the state of a branch transaction. If present, the output can be quickly retrieved as needed.
- The structure of the multi-state database can be likened to the Patricia trie used in Ethereum. While the specifics may vary, the underlying principle is the same: overlapping ledger states with different roots can share a majority of trie nodes, allowing for efficient management and retrieval of large volumes of baseline states in the database

7.2.2 Purpose: enforce specific baseline state for each sequencer transaction

The sequencer constraint validity script on the sequencer transaction ensures that each sequencer transaction T directly or indirectly chooses a particular branch transaction T_{branch} on the same slot, as its baseline. We will elaborate in section 7.3.Sequencer baseline. Stem output).

7.2.3 Purpose: make network to choose one state version to follow

All branch transactions on the same slot will inherently conflict due to their design.

A branch transaction can only contain exactly two produced outputs: the chain-constrained sequencer output and, adjacent to it, the **stem output**. The stem output is locked with the *stem lock script*, which enforces additional constraints specific to branches (see Sequencer baseline. Stem output).

The *stem lock* allows the stem output to be consumed (unlocked) by any other branch transaction without the need for signatures or other unlock data. By creating a branch, the sequencer must choose a particular stem output from the branch on the previous slot to consume.

Therefore, multiple branches on the same slot will necessarily double-spend the stem output of the previous branch, causing branch transactions on the same slot to conflict intentionally. This design ensures that only one branch transaction and one stem output can be recorded in the ledger state. It forces sequencers to select one (preferred) branch among several competing ones to continue to the next slot. This results in a **tree of branches**, where the longest branch represents the broadest consensus.

7.2.4 Purpose: branch inflation bonus

A branch transaction rewards the sequencer with a branch inflation bonus (see the section 6). Sequencers compete to have their branch included in the future chain of ledger states to receive this bonus.

The branch inflation bonus is a randomized (yet capped) value that sequencers can mine to increase its amount. See section 8.2.

7.2.5 Tree of branches. The winning branch

Each branch transaction references a single predecessor branch by consuming the stem output, creating a tree of baseline states that mirrors the structure of tree of blocks in the blockchain.

Branches within the same slot are conflicting, ensuring each sequencer chain forms a single non-forking chain of branches—a winning branch. Adhering to the winning branch and abandoning (orphaning) other branches is the essence of the converging consensus on the ledger state.

7.3 Sequencer baseline. Stem output

The sequencer validity constraint mandates that each valid sequencer transaction has a deterministically defined **baseline state** (refer to 7.2.2). For each sequencer transaction T, exactly one of the following conditions must be met:

1. T is itself a branch transaction and therefore defines its own baseline state.
2. The chain predecessor T_{pred} of T is on the same slot as T. In this case T will inherit its baseline state from T' recursively
3. T endorses other milestone T' on the same slot as T: $T' \Leftarrow T$. In this case, T inherits baseline state from T' recursively

The enforced validity constraints outlined above ensure that each non-branch sequencer milestone T has a chain of branch transactions in its past cone. The tip of the chain belongs to the same slot as T and is known as the **baseline branch**. The ledger state that the baseline branch represents is referred to as the baseline state of transaction T. The baseline branch is denoted as $baseline(T)$, and the baseline state of T is denoted as G_{L_T}.

As per above (see Branches), different versions of the ledger state make a tree of branch transactions. Each ledger state have exactly one **stem output** in it. For the state S_T represented by the branch transaction T we will denote the unique stem output $stem(T)$. Always true that $stem(T) \in S_T$ and $stem(T) \in out(baseline(T))$.

As per the details provided in the section 7.2, different versions of the ledger state form a tree of branch transactions. Each ledger state contains exactly one stem output within it. For the state S_T represented by the branch transaction T, we denote the unique stem output as $stem(T)$. True that $stem(T) \in S_T$ and $stem(T) \in out(baseline(T))$.

Definition 7.1 (Branch output of the sequencer transaction)**.** *Branch output of the transaction, denoted as $branch(T)$, is the sequencer output of the baseline transaction $baseline(T)$*

The ledger constraints ensure that in a valid ledger L each sequencer milestone T will always have uniquely defined:

- baseline state: G_{L_T}
- baseline branch: $baseline(T)$
- stem output: $stem(T) = stem(baseline(T)) = stem(G_{L_T})$
- branch output $branch(T)$

In the figure 7 you can see various ways a sequencer can build its chain.

There are two sequencer chains: chain A and chain B. In slot N-1, branch A extends into two conflicting branches in slot N: Branch A and Branch B. This is achieved by both chains consuming the same stem output, resulting in conflicting transactions (transactions 1 and 2) in slot N.

Transaction 3 on chain A extends its own chain from branch 4 within the same slot. Since it continues from branch 4, it inherits the baseline state, so it does not need to endorse any other transaction to reference the baseline.

Transaction 5 continues from its own milestone 6 on the previous slot, so it does not have a baseline defined in slot N. Therefore, it endorses milestone 3 on another chain. This action defines the baseline of Transaction 5 as being derived from Branch 4.

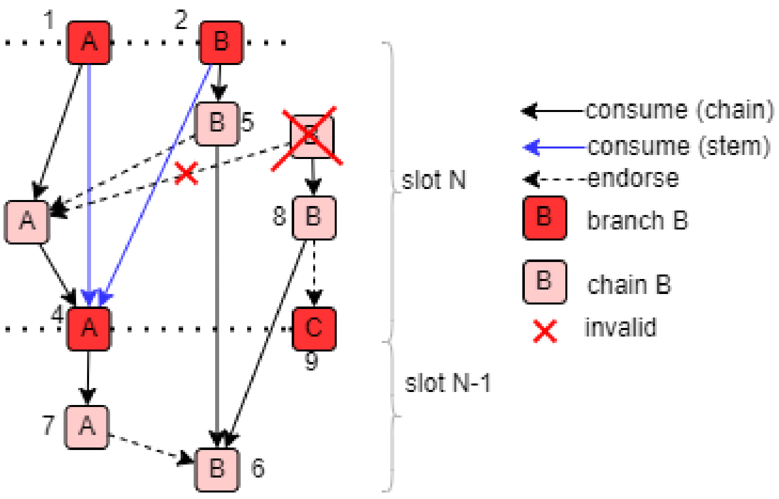


Figure 7: Branches

The sequencer chain B continued by creating another milestone 8, following the same predecessor 6 from the previous slot. However, milestone 8 endorsed a different branch 9, which resulted in the two milestones of sequencer chain B in slot N becoming conflicting.

7.4 Cross-endorsement

The important aspect of sequencer and branch constraints described above is that any sequencer chain can create its next milestone, endorsing any other sequencer transaction on another chain. If the chosen endorsement target conflicts with the latest milestone of the sequencer chain, the sequencer must revert back from its latest state on the chain to a previous state. This requires the sequencer to essentially undo some of its chain progress to reach a state where it can appropriately endorse the chosen target.

The ability to endorse any other sequencer chain with greater coverage (see also 7.5.Ledger coverage and 8.Convergence) means that a sequencer can continuously increase its ledger coverage by consolidating its state with the states of other sequencer chains. This process of consolidation allows the sequencer to continuously expand its reach and influence over the ledger, providing greater overall contribution to the consensus within the network.

Let's consider example in the figure 7 above: sequencer B just produced milestone 8 but now wants to continue by endorsing milestone 3. It cannot continue with the milestone 8, because it conflicts with to 3. So, the sequencer B finds own chain output (milestone) with *chain ID B* in the baseline state of transaction 3 (branch 4). That chain output always exists and it is unique in any state. It appears to be milestone 6 and it is consistent with baseline of 3 because it is in the because it is in the past cone of the branch 4. So sequencer is able to continue by creating milestone 5 from 6 and endorsing 3. This would be equivalent to the reverting the state from 8 to 6 and continuing from there. 5 and 8 will be different and conflicting state forks of the sequencer B.

In the scenario described in figure 7, sequencer B has just produced milestone 8 but now wants to continue by endorsing milestone 3. However, because milestone 8 conflicts with milestone 3, sequencer B cannot continue from milestone 8.

Instead, sequencer B must revert to an earlier state that is consistent with milestone 3. It does this by finding its own chain output (milestone) with *chain ID B* in the baseline state of transaction 3 (branch 4). Since there is a unique and consistent milestone with *chain ID B* in any state, sequencer B identifies milestone 6 as the milestone in its past chain that is consistent with the baseline of milestone 3.

Milestone 6 is consistent with milestone 3 because it is within the past cone of branch 4. By reverting to milestone 6, sequencer B can create milestone 5 from milestone 6 and endorse milestone 3. This process is effectively equivalent to reverting the state from milestone 8 to milestone 6 and then continuing from milestone 6.

As a result, milestones 5 and 8 will represent different and conflicting state forks within sequencer B's chain. This situation illustrates how sequencers can revert to earlier states to reconcile conflicts and make consistent progress within the ledger.

It is important to note that sequencer B does not require the full transaction 6 to produce transaction 5; it only needs the sequencer output, which is still part of the state. Meanwhile, transaction 6 itself can already be pruned from the UTXO tangle.

This example illustrates how sequencers can revert to a previous state to create a new sequencer transaction that endorses another sequencer transaction on a different chain. By reverting to a prior state, sequencers can resolve conflicts and continue their chains in a consistent manner, ensuring that their milestones remain connected and aligned with the overall ledger state.

7.5 Ledger coverage

The **ledger coverage** of the sequencer milestone T is a function of the transaction and its past cone, and it is calculated uniformly by each participant (is deterministic).

In order to define full ledger coverage value, we first need to define the **ledger coverage delta** (or **coverage delta**) of the sequencer transaction T in the ledger $L_T = past(T)$ with baseline state G_{L_T}. The *coverage delta* is the total amount of outputs that are directly and indirectly consumed by transaction T in its baseline state G_{L_T}. We will denote it as $coverage\Delta(T, G_{L_T})$ (see also figure 3).

Definition 7.2 (Rooted output)**.** *The past cone of sequencer transaction T "ends" at the baseline state G_{L_T}, and we say it is "rooted" there. An output O is considered a* ***rooted output*** *when it is either the sequencer output on the baseline branch denoted as $branch(T)$ (which is uniquely defined) or when it is consumed by any transaction from the ledger L_T.*

Definition 7.3 (Rooted outputs of the sequencer transaction)**.** *All rooted outputs of the transaction T make a set:*

$$rooted(T) = \{O \in G_{L_T} \| \exists T \in L_T : O \in in(T)\} \text{ or } O = branch(T)\}$$

The output denoted as $branch(T)$ may or may not be consumed in the past cone of T. Regardless, the branch output is included in the set of rooted outputs, $rooted(T)$, as it is part of the baseline state G_{L_T} of T.

The rooted outputs are those which were removed from the baseline ledger state G_{L_T}; therefore they they are absent in the final ledger state $S_{L_T} \cup \{branch(T)\}$. $rooted(T)$ is the subset of the baseline ledger state that is **covered** by the past cone of T.

Definition 7.4 (Ledger coverage delta of a sequencer transaction)**.** *Ledger coverage delta (or simply coverage delta) of the transaction T as sum of amounts of rooted outputs:*

$$coverage\Delta(T) = \sum_{O \in rooted(T)} amount(O)$$

Some observations:

- the minimal value of $coverage\Delta(T)$ is the amount on the branch output: $amount(branch(T)) > 0$
- the maximal value is total supply on the baseline state: $coverage\Delta(T) \leq totalSupply$
- if $T \leftarrow T'$ then $coverage\Delta(T) \leq coverage\Delta(T')$: monotonicity of ledger coverage along consumption references
- if $T \Leftarrow T'$ then $coverage\Delta(T) \leq coverage\Delta(T')$: monotonicity of ledger coverage along endorsement references. Note, that endorsement references are only possible within one slot
- let's say token holder controls baseline output O on the ledger. It can consume this single output in the new transaction T, so it's coverage will be $coverage\Delta(T) = amount(O)$. If the token holder will choose in addition to endorse any other transaction T', the $coverage\Delta(T) = amount(O) + coverage\Delta(T') > amount(O)$. So, by consolidating own output with some independent transaction in one state, the token holder will increases ledger coverage of the new transaction.

- $coverage\Delta(T) \leq totalSupply$, so it can't grow forever on a given baseline state.

We define full *ledger coverage* as a weighted sum of the coverage deltas of the sequence of past branches, with weights that decline exponentially as they go further back in the sequence.

Let's say $B_0 = T$, $B_1 = baseline(T)$ and the sequence $B_1, B_2, \ldots B_N$, where $B_{i+1} = baseline(B_i), i = 1 \ldots N-1$, is a sequence of branches back to the genesis $G = B_N$ (the branch transactions in the sequence may or may not belong to strictly subsequent slots).

Definition 7.5 (Ledger coverage of a sequencer transaction)**.** *The ledger coverage (or simply coverage) of the transaction$T$ we define the following way:*

$$coverage(T) = \sum_{i=0}^{N} \lfloor \frac{coverage\Delta(B_i)}{2^i} \rfloor$$

or in the recurrent form:

$$coverage(B_i) = coverage\Delta(B_i) + \lfloor \frac{coverage(baseline(B_i))}{2} \rfloor$$

For practical reasons, we assume the values of *coverage* and *coverage*Δ fit 64-bit integers, therefore we can restrict summing along maximum 64 past branches back. If the sequence skips some slots (no branch in it) we assume $coverage\Delta = 0$ in that slot. Further to the past the previous branch is, exponentially less its contribution to the final ledger coverage.

Sometimes, for simplicity sake, we can assume coverage delta in each slot is constant C. Then:

$$coverage(T) = C \cdot (1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{63}}) = C \cdot (2 - \frac{1}{2^{63}}) < 2C$$

In general, always true the following: $coverage(T) < 2 \cdot totalSupply$

Note, that in the same manner as above, we can define $coverage\Delta(\cdot)$ and $coverage(\cdot)$ for the set of transactions with the consistent past and the same baseline, by taking union of the $rooted(\cdot)$. In that case we will use notations $coverage\Delta(\{T_1, \ldots T_K\})$ and $coverage(\{T_1, \ldots T_K\})$ respectively.

7.6 Sequencing strategy

We introduced the general idea in the section 2.3. Here, we provide a more detailed discussion.

The sequencing strategy is a complex, implementation-intensive, and heuristic-based problem. While we strive to present the approach in a generic manner based on prototype implementation, we acknowledge the lack of a deeper mathematical model and generic framework for sequencer strategy at this time. The prototype implementation demonstrates the feasibility of practical sequencer strategies, and we anticipate ongoing research and continuous improvement of existing strategies.

It is important to note, that sequencer is free to choose any strategy it likes, as long as it conforms to ledger constraints. Our hypothesis is, that, more efficient strategies will have only marginal advantage against the generic one in terms of the profitability, or, otherwise they immediately will be copied by other sequencers.

Technically, the **sequencer** is an automated process that typically has access to the private key controlling a token holder's tokens. It also interacts with the UTXO tangle and other services provided by the node. In practice, a sequencer functions as a program running as an optional part of the node.

The **sequencing strategy** refers to the algorithm or behavioral pattern employed by a program (or, potentially, an AI agent) to guide the actions and decisions of a sequencer within the network.

A sequencer must build a sequencer chain in the ledger to maximize the interests of the token holder, aiming for profit and return on capital, based on income from inflation, tag-along fees and delegation margin (see also Non-sequencing users). Sequencers have the flexibility to adopt any approach that considers other sequencers, as well as ledger and time constraints, while pursuing this goal.

There may not be a single optimal strategy for sequencing; instead, there could be multiple strategies with their own pros, cons, and specific niches. In the provided algorithm section (Sequencing strategy), a generic form of one possible strategy is described, called GenericSequencer$(\cdot)$.

Algorithm 1 Generic sequencer

```
procedure GenericSequencer($sequencerID, privateKey, utxoTangle, localClock$)
    $timestampTarget \leftarrow 0$
    loop
        $bestCandidate \leftarrow nil$                    ▷ assume $coverage(nil) = 0$
        $timestampTarget \leftarrow$ decideTimestampTarget($localClock.now, timestampTarget$))
        while $localClock.now < timestampTarget.realTimeValue$ do
            $chainTip, endorse, consume \leftarrow$ SelectInputs($sequencerID, timestampTarget, utxoTangle$)
            $candidate \leftarrow$ MakeTransaction($privateKey, timestampTarget, chainTip, endorse, consume$)
            if $candidate \neq nil$ then
                $candidate \leftarrow$ MineBranchInflationBonus($candidate$)
                if $coverage(candidate) > coverage(bestCandidate)$ then
                    $bestCandidate \leftarrow candidate$
                end if
            end if
        end while
        if $bestCandidate \neq nil$ then
            submitTransaction(bestCandidate)
        end if
    end loop
end procedure
```

The sequencer is constantly monitoring the local UTXO tangle, which is dynamically updated with new transactions as they arrive, and continuously producing sequencer transactions in real time to build on the sequencer chain.

There are two primary tasks that define the sequencer's strategy: decideTimestampTarget(·) and selectInputs(·) (see descriptions below).

The task mineBranchInflation(·) involves mining the largest feasible value of the branch inflation bonus, which can be resource-intensive if the sequencer aims to compete for the branch inflation bonus aggressively.

Similarly, the task selectInputs(·) can be computationally expensive in high-load situations (high *transactions-per-second*).

These tasks impose significant real-time constraints on the sequencer, affecting its overall performance.

7.6.1 Pace of the sequencer chain

Function decideTimestampTarget(·) is responsible for calculating the timestamp of the next sequencer transaction, effectively setting the pace of the chain. The optimal pace, or frequency of sequencer transactions, is a matter of strategy and optimization, and the sequencer must consider the following factors:

- Transactions are subject to a minimum interval between timestamps, defined by the **transaction pace validity constraint**. This constraint sets a lower bound on the timestamp for the next sequencer transaction based on the given inputs.

- The sequencer must produce the next sequencer transaction in real time without issuing transactions too frequently or with excessive gaps between them. Producing transactions too frequently may hinder consolidation of ledger coverage and disrupt synchronization with other sequencers. Conversely, long gaps between transactions may limit the sequencer's ability to process tag-along and delegation outputs and disrupt other sequencers' ability to catch up.

- The target timestamp cannot be too far ahead or behind real time (see Nodes. Ledger time) to avoid sequencer transactions being placed on hold (if ahead of time) or being disregarded (if behind other transactions with close timestamps).

- The sequencer aims to generate a branch transaction at the slot edge and reserve sufficient time to mine a larger branch inflation bonus.

Considering the above factors, choosing an optimal next timestamp target typically relies on heuristics. Fortunately, experiments have shown that reasonable heuristics exist, providing a foundation for optimization.

7.6.2 Select inputs for the transaction

When the $slot(timestampTarget) = 0$, the sequencer is simply creating a branch transaction. There is little to optimize in this scenario, as the only inputs needed for the branch transaction are the stem output and the chain predecessor, which will be consumed without endorsements.

In other cases, the function $\textsc{selectInputs}(\cdot)$ analyzes the UTXO tangle and returns a collection of consistent inputs needed to construct the transaction optimally. There can be a large number of such collections, and the function is stateful, keeping track of all the collections it has previously returned. Subsequent calls to $\textsc{selectInputs}(\cdot)$ enumerate all or a reasonable majority of all possible collections and return them in a heuristic order, prioritizing those that provide the most significant ledger coverage first.

The function $\textsc{selectInputs}(\cdot)$ must choose the following inputs from the UTXO tangle for each candidate transaction:

- This output will be used as a predecessor in the transaction, forming the basis for the chain.
- The function should identify and select a set of endorsement targets, which are transactions that maximize the ledger coverage of the future transaction. This selection should prioritize maximizing coverage to the greatest extent possible.
- Tag-along outputs provide income for the sequencer and increase ledger coverage. The function should strive to maximize the consumption of tag-along and delegation outputs to enhance the sequencer's income and coverage.

The function $\textsc{selectInputs}(\cdot)$ must ensure that all selected inputs are consistent, i.e. does not contain double-spends in the consolidated past cone.

Suppose at some point in time UTXO tangle contains N sequencer outputs, M tag-along outputs, K delegation outputs that can be potentially consumed by the sequencer plus, in addition, E potential endorsement targets, out of which P we want endorse in the transaction (P usually is 1 or several). There's a vast number of possible combinations for the collection, candidates for selection: $O(N \cdot 2^M \cdot 2^K \cdot \binom{E}{P})$.

For each of these combinations, the function will have to check if the combination does not contain double spends in the past cone. This way the task of selecting optimal combination may become too computationally costly for the real time optimization. This points to the importance of the heuristics.

To spare the reader from complexities, we will skip details of $\textsc{selectInputs}(\cdot)$ as it is implemented in the prototype.

In general: the $\textsc{selectInputs}(\cdot)$ first finds the heaviest other sequencer transaction in the slot and iterates over own past milestones to find the best one to be combined with the heaviest milestone. At least one own milestone consistent with the endorsement target always exists, but it may be not the heaviest, so the sequencer needs to try a set of possibilities to select best consistent extent/endorse pairs.

Then, for each potential extent/endorse combination, sequencer collects as much pending tag-along and delegation outputs as it fits into the transaction. Each of those combinations is checked for being consistent. The consistent collection is returned.

Note, that repetitive call to $\textsc{selectInputs}(\cdot)$ will return different results because situation on the UTXO tangle is constantly evolving.

This simple strategy assumes only one endorsement target. It results in a minimal rate of state consolidation and coverage growth with each transaction. Experiments show, that even with this limitation, the convergence of the state toward consensus is relatively fast. Strategies that consolidate two or more sequencer chains, as well as strategies that are more adaptive to the current situation on the network, should significantly increase the rate of convergence (see also Convergence).

7.7 Non-sequencing users

The sequencers will be the primary proactive participants in the network. The network's security depends on the cooperation and agreement of major players on the shared ledger state. However, what about other users who cannot or do not want to be sequencers for various reasons?

Two types of lock constraints—*tag-along lock* and *delegation lock*—allow non-sequencer users to contribute to the consensus on the ledger state and earn inflation rewards.

Each output on the ledger must include a mandatory *lock constraint* that enforces certain requirements for the consuming transaction, such as unlocking conditions. A common example is the *address25519* lock, where the output can be unlocked with a transaction signature corresponding to a specific address, or a *chain lock*, where the output can be unlocked with the signature on a transaction that unlocks a chain output with a specified *chain ID* in the same transaction, i.e., the chain controller's signature.

7.7.1 Tagging-along

If *Alice* wants to transfer an amount of tokens to *Bob* and *Alice* is not a sequencer, she can create a transaction T that consumes some of her outputs and produces an output of amount A with the target lock constraint set to *address25519* using *Bob*'s wallet address. Additionally, T would produce a remainder output that goes back to *Alice*'s wallet.

However, if Alice submits T to the network, the transaction will not be processed because there is no sequencer to include the output in the consensus ledger state. As a result, T will become orphaned, meaning it will be "disregarded" by the network.

To fix that, in the transaction T, *Alice* produces an additional output with an amount of tokens ϕ and *chain-locks* it to make it unlockable by a sequencer with the specified *sequencerID*. This way, the transaction T produces three outputs: (1) with amount A to *Bob's* address, (2) with amount ϕ to the chain $sequencerID$ and (3) the remainder.

The output (2) with amount ϕ that is chain-locked to the sequencer with *sequencerID* is known as a **tag-along output**. The amount ϕ is referred to as the **tag-along fee**, and the sequencer with *sequencerID* is known as the **target tag-along sequencer**.

When *Alice* creates a transaction T that includes a tag-along output targeted at a specific sequencer, the sequencer becomes interested in consuming that output because it provides income in the form of the tag-along fee. Consequently, the sequencer naturally includes the tag-along output in its own sequencer transaction.

By incorporating the transaction T into its own transaction, the sequencer ensures that T becomes part of the consensus ledger state. This allows *Bob* to receive his tokens as intended, and the sequencer earns the tag-along fee as income. In this way, *Alice*'s transaction is successfully processed, and both the target recipient (*Bob*) and the sequencer benefit from the transaction.

The tag-along fee amount ϕ is generally small compared to the amount being transferred to *Bob* (A), which can vary in size. The amount of the tag-along fee is determined by the sequencer and, ultimately, by market forces.

Alice is naturally motivated to choose tag-along sequencers that offer smaller fees, as this will minimize her transaction costs and make her transaction more cost-effective. This market-driven competition among sequencers to provide lower fees encourages efficiency and cost-effectiveness in the network, benefiting users like *Alice*.

It's important to recognize that *Alice* contributes to the ledger coverage of the sequencer transaction with the full amount of the transaction, not just the tag-along fee ϕ. This contribution to coverage can include the amount transferred to *Bob* (A) and any remaining funds returned to Alice as change.

Sequencers have an inherent interest in maximizing ledger coverage to improve their chain's reach and influence within the network. This dynamic creates the potential for larger transfers to be entirely fee-less. Since sequencers have an independent interest in attracting more coverage to their chains, they may choose to process larger transactions without imposing a fee to gain the associated ledger coverage benefits.

Tag-along mechanism allows non-sequencer users earn inflation just like sequencers. *Alice* produces transaction with the chain transition and inflation on it. She tags-along the transaction to some sequencer. If tag-along fee is smaller than inflated amount, the transaction will be profitable for *Alice*.

Allowing tag-along output with zero or very small fees creates also a problem of *dusting.* We will elaborate on the solution for that problem in section 10.Spamming prevention.

7.7.2 Delegation

Delegation allows passive, non-sequencer capital to be involved in cooperative consensus without the token holder actively participating in sequencing. This is achieved by allowing the token holder to delegate the rights to use their capital as ledger coverage for sequencing to a sequencer while maintaining ownership and control over their tokens.

Key points regarding delegation:

- Through delegation, the token holder grants the sequencer the right to use their tokens as part of the sequencer's ledger coverage. This boosts the sequencer's ability to participate in consensus and increases their potential earnings.
- Delegation does not involve transferring custody of the tokens from the token holder to the sequencer. The token holder retains ownership of the tokens and can revoke the delegation at any time.
- In exchange for delegating their tokens, the token holder can earn a share of the inflation rewards or other earnings generated by the sequencer's activities.
- Delegation allows more capital to be involved in the consensus process, contributing to network security and stability. This helps improve overall decentralization and consensus efficiency.
- Sequencers benefit from having access to additional capital for increasing their ledger coverage, which can improve their chances of successfully contributing to the consensus and earning rewards.

In summary, delegation is a useful mechanism for involving passive capital in cooperative consensus, providing benefits to both the token holder and the sequencer while maintaining security and ownership of the assets.

The delegation lock constraint enables a chain-constrained output to be unlocked by the target sequencer without the risk of the sequencer stealing the funds. Additionally, it allows the delegated capital to be returned to its owner if and when necessary.

Let's say output O is locked with the target $sequencerID$ and a return specification. It can be unlocked by the $sequencerID$ chain if the transaction that consumes O as a predecessor satisfies the following conditions:

- it produces the successor output with exactly the same set of constraints as the predecessor, except the amount may increase with inflation, possibly minus a delegation margin.
- The consuming transaction and the successor output satisfies the return specification. Normally, the return specification is a deadline by which the tokens must be returned to the owner. This allows the owner to unlock the output and reclaim the tokens after the deadline. Another return option could involve specifying periodic return time windows; for example, the owner can consume the output and withdraw tokens during the last 10 minutes of each hour.

8 Convergence

8.1 Growing the UTXO tangle

The UTXO tangle is developing as long as there are active sequencers aiming to earn inflation by building sequencer chains with growing ledger coverage. The set of ledger constraints must ensure it is always possible create new transaction with growing $coverage\Delta(\cdot)$ within one slot.

Let's simplify situation by assuming to tagging-along and delegating by sequencers. Also, let's assume every existing sequencer is present in any branch ledger state of consideration.

The reasoning below holds also without simplifications.

Statement 8.1. *Every sequencer can create a sequencer transaction in the slot.*

Proof. Let's say we have N branches which starts the slot: $B_1, \dots B_N$. For a sequencer with ID A, there are two (non-exclusive) cases:

- one of transactions B_i belongs to the sequencer A. In this case the sequencer just extends the transaction B_i by consuming its sequencer output in the next sequencer transaction T_{next} belonging to the same slot. The baseline of the transaction will be the branch itself: $baseline(T_{next}) = B_i$
- sequencer A has its tip in at least one of branches ledger states, say B_i. The state S_{B_i} contains exactly one output (sequencer milestone) or tip of the chain A, denoted as $tip_{B_i}(A)$.

  The sequencer A always can create next sequencer transaction T_{next} by consuming $tip_{B_i}(A)$ and endorsing B_i. The latter becomes the baseline of T_{next}

□

Definition 8.1. *We say that full sequencer coverage is reached in the slot τ by a transaction T when the set $rooted(T)$ contains all sequencer milestones in the state $baseline(T)$ (in other words, when transaction T covers all sequencers in its baseline).*

If full sequencer coverage is reached in the slot by some transaction T, coverage of any new transaction (assuming no tag-along and delegation inputs) can not be bigger than the reached maximum $coverage\Delta(T)$.

Statement 8.2. *Maximal coverage among transaction of the UTXO tangle in one slot can grow as long as full sequencer coverage is not reached in that slot and time constraints permit*

Proof. Let's say T is the transaction with the biggest coverage in the slot. By assumption, some sequencer A is not covered by T, i.e. $tip_{baseline(T)}(A) \notin rooted(T)$.

The sequencer A can always create transaction T_{next} by consuming its own milestone $tip_{baseline(T)}(A)$ and endorsing T. The coverage of the new transaction will be:

$$coverage\Delta(T_{next}) = coverage\Delta(T) + amount(tip_{baseline(T)}(A)) > coverage\Delta(T)$$

□

Statement 8.3. *Sequencers can build UTXO tangle continuously*

Proof. We start at some genesis slot τ. According to the statements above, each sequencer can create transactions in the slot. After adding new transaction to the slot, one of transaction will have maximal coverage. Until full sequencer coverage is reached and ledger time constrains permit, new transactions can keep increasing ledger coverage.

After full sequencer coverage or end of the slot τ is reached, sequencers can create branches at the edge of the next slot $\tau + 1$.

This process can be repeated again and again. □

8.2 Rich chain bias. Meta-stability. Random branch inflation

Each sequencer is trying to reach as big coverage in the slot as it can. This leads to the conclusions, that very likely ledger coverages of branches on the next slot will be close to each other or even equal. It can lead to a situation of meta-stability.

Meta-stability in this context refers to a state where the system is stuck or oscillates between different branches without a clear preference for any single one due to their equal coverages. While meta-stability may be broken by other random factors, in theory it could lead to slower convergence or even in parallel chains of branches (forks) with approximately equal coverage. This may create a problems, known as **nothing-at-stake**, when sequencers, blindly following biggest ledger coverage, maintain several conflicting chains of branches (forks) in parallel.

One approach to address this problem would be, in case of equal coverage, to prefer higher transaction hash. This, however, can lead to the undesirable hash mining competition among sequencers.

Let's consider figure 8 (numbers will be used as indices of transactions). Let's say the sequencer outputs of sequencers A (T_4) and B (T_9) at the end of the previous slot have respective token amounts

$amount_A$ and $amount_B$. Additionally, let's assume that the ledger coverages of transactions (T_4) and (T_9) are equal. This means coverages of branches A and B are equal too:

$$coverage(T_3) = coverage(T_8) = C$$

The amounts on the sequencer outputs of branches A (T_3) and B (T_8) are $amount_A + I_A$ and $amount_B + I_B$, where I_A and I_B are branch inflation bonus amounts.

Now let's consider two options, labeled as **red** and **green**, for how sequencer A could continue with a sequencer transaction to the next slot:

- the red option means A will consume its own branch and continue with the red A transaction T_1 which endorses T_2. The coverage of it: $coverage(T_1) = amount_A + I_A + amount_B + \frac{C}{2}$.

- the green option means A will endorse branch of B T_8 and consume its own output in T_4 from the baseline state of B, the $baseline(T_8)$. The coverage of transaction T_6 will be $coverage(T_7) = amount_A + amount_B + I_B + \frac{C}{2}$

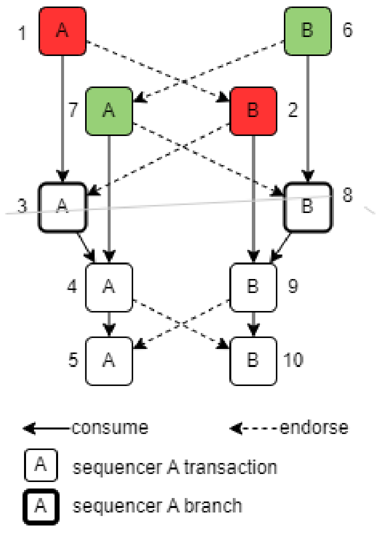


Figure 8:

Between the two possible options T_1 and T_7 the sequencer A and all subsequent sequencers will choose the one with bigger coverage, which, in this case, is the one with the bigger branch inflation bonus. Symmetrically, the same situation will happen with sequencer B.

In case of $I_A > I_B$, the red option will win while the green option will never appear or will be orphaned. In case of $I_A < I_B$, the green option will prevail.

Conclusion 8.1. *in the frequent situation when ledger coverages are equal between branches, the branch with the bigger branch inflation bonus will prevail, i.e. biggest coverage rule, followed by sequencers, will point to the one with bigger branch inflation bonus. The amount of tokens on the sequencer outputs does not influence the outcome.*

If $I_A = I_B$, the situation will be indecisive, i.e. may lead to a meta-stability. We conclude that fixed (as ledger constant) branch inflation bonus would not be a good idea because of this reason.

On the other hand, if we assume the branch inflation bonus is proportional to the on-chain balance (same how inflation is calculated inside the slot), it would result in a situation where the richer chain will always win the branch inflation bonus, i.e. what we call *rich chain bias.* While this may not pose a security problem, though it would be an unfair reward distribution which would disincentivize smaller sequencers from issuing branches.

To address both problems of meta-stability and rich chain bias, we introduce a **random branch inflation bonus**. The random branch inflation bonus does not create any advantage for the richer sequencer and it also breaks meta-stability.

The biggest coverage rule becomes two-fold:

- if coverages are different, the transaction with the bigger one prevails

- if coverages are equal, the transaction T with the bigger transaction hash $id(T)$ prevails

We also introduce a special chain constraint on the branch transaction. The constraint imposes the following condition on the branch inflation bonus I_{T_B} on the branch transaction T_B:

$$I_{T_B} \le hash(T_B) \mod (I_{max} + 1)$$

where
- I_{max} is a static ledger constant of maximum possible branch inflation bonus
- $hash(T_B)$ is a cryptographic hash taken from transaction bytes, interpreted as integer

Observations:

- validity of the constraint is easy to check.
- $I_{T_B} = 0$ always satisfies the constraint. Other values **must be mined by brute force**. Maximum value of I is I_{max}
- assuming the hash value is uniformly distributed, the probability of I_{T_B} being greater than x is:

$$P\{I_{T_B} \geq x\} = \frac{I_{max} + 1 - x}{x_{max} + 1} = 1 - \frac{x}{I_{max} + 1}, 0 \leq x \leq I_{max}$$

- bigger value of I_{T_B} requires more effort. Probability of finding a random value from upper 5% in one step is 1/20, so the cost of mining is low. Expected number of steps needed to find a random value from the interval close to I_{max} grows very steeply (exponentially).
- probability of satisfying the constraint with exact maximal value I_{max} in one step is $\frac{1}{1+I_{max}}$. So, for $I_{max} \approx 10^7$ the number of expected mining steps will be at the order of 10^8.

  Taking into account that each mining step requires not just hashing (like in Proof of Work), but also signing the transaction with elliptic curve operation, this may impose a significant computational cost[22].

  Even if a sequencer invests in hardware powerful enough to mine the maximum branch inflation bonus, the bigger hash of the transaction data will be used for comparing equal ledger coverages. This would lead to the mining of a larger hash value (essentially without limits), making the process impractical.
- assuming $I_{max} \approx 10^7$, number of values in the upper 1% will be $\approx 10^5$ and the random branch inflation bonus will be among those values. It is enough for randomness and does not require any special hardware.

There are big chances, however, that sequencers will run to the maximum value. It will lead to equal branch inflation bonus, not random. Then random hash rule will be applied and winning choice will be distributed randomly among branches with the maximal inflation bonus. To become really preferred, sequencer will need to mine the bigger possible hash, in each step mining maximal value. This scenario won't be realistic even for ASIC or GPU. This leads to the following strategic choices:

1. leave the maximum bonus and face full randomness of selection among those who mined maximal value. That would result in mining race and *fastest processor bias*, sub-optimal behavior for Proxima. This will impose certain costs.
2. just relax an enjoy mining a reasonably big random value with random selection among all sequencers. This approach will have minimal costs.

The possible strategies for sequencers must balance the following factors:

- Value of the constant I_{max} which balances cost of the mining and reward.
- Cost of engaging in the race. Strategy can have almost zero mining cost.
- The real time constraints. Sequencer has some 100 milliseconds to produce the transaction right before the slot edge. Waiting longer for bigger coverage to come for endorsement may be better, more profitable strategy, than spend all those 100 milliseconds for mining.

[22]how significant it may be is debatable. Mining ASICs and GPUs nowadays are extremely powerful. However, cost and strict time constraints mus also be considered. In the prototype we use constant $I_{max} = 5.000.000 = 5\imath 0^6$

The above makes us to believe, that stable sweet spot for branch generation mining strategies might exist. The topic require more detailed modeling, analysis, experimentation and optimization of constants and, likely, the ledger constraint to rule out undesirable behaviors.

Note that, as any selected branch with big enough coverage is good, so selection of branch it is not about security, but about incentivization bias. For this we need unpredictable randomness (which is not easy to obtain). The amount of branch inflation bonus must be just big enough to make biggest sequencers to bother to issue branches. The biggest part of the inflation rewards should come from the ordinary inflation om chains within slot.

8.3 Randomness of convergence. Heuristics

The growth of the UTXO tangle is a random process not only because the randomised branch inflation bonus, but also due to other unpredictable variables, such as communication delays, the random time needed to produce transaction and sequencer strategy. Each sequencer selects inputs for the next transaction in the chain from an asynchronously changing set of transactions, which arrive at random intervals and in random order from other sequencers. The produced transaction reaches other sequencers after a random time interval.

The mathematical modeling of this random process is beyond the scope of this paper. Meanwhile, we we use our common judgement to come up with some rules of thumb for the sequencers.

Figure 9 illustrates a possible situation on the UTXO tangle. For simplicity's sake, the diagram assumes that all sequencer transactions share the same baseline state (below the grey line), represented as a collection of outputs shared by all sequencers.

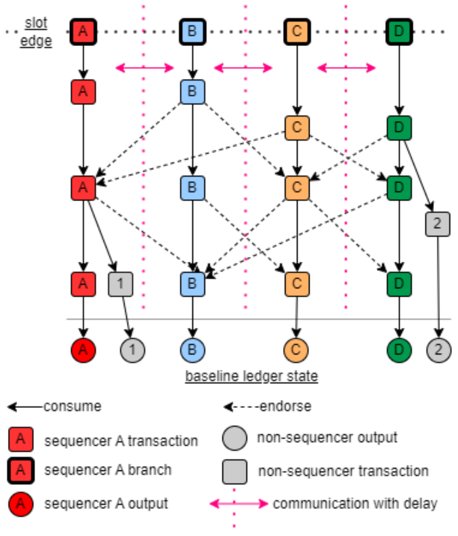


Figure 9:

Each sequencer is separated from other sequencers by network communication delays. Sequencers B and C successfully cover the entire baseline ledger state, with all outputs included except 2. Thus, their coverage will be equal (largest), causing them to compete as branches. The branch that is chosen from B and C will continue, while the other will be orphaned.

Meanwhile, sequencers A and D did not manage to cover all possible outputs in the baseline due to network delays. Their coverage will be smaller, and their branches and respective ledger states will have no chance of being chosen as preferred, so they will be eventually orphaned.

The non-sequencer output 1 was tagged along to sequencer A and then endorsed by sequencers B and C, so it will make it into the ledger state in the next slot anyway, even if sequencer A has no chance of producing the preferred branch and regardless of which of the two heaviest branches wins. Output 2 was tagged along to sequencer D but will not be included in the state in the next slot (it can be tagged along in the next slot, though).

Observations:

1. Sequencer transactions with more tokens are more likely to be chosen as endorsement targets by others, leading to faster inclusion in the (future) ledger state.

2. Employing delegated capital in the sequencer and consuming tag-along inputs helps the sequencer transaction to be included in the (future) ledger state more quickly.

3. Endorsing more other sequencer chains generally leads to faster convergence than endorsing fewer.

4. Smaller communication delays between the heaviest sequencers favor convergence.

5. Submitting a transaction later (in real terms) than its timestamp may result in it no longer being the heaviest by the time it reaches other sequencers. Therefore, a **local clock that is behind the average is unfavorable for convergence**.

6. Submitting a transaction too early (in real terms) than its timestamp may result in the nodes putting it on hold, causing the submitting node to lose the opportunity to wait a few extra milliseconds and endorse a heavier transaction. Therefore, a **local clock that is ahead of the average is unfavorable for convergence**.
7. Sequencer must consume tag-along outputs as early as possible, making the sequencer chain heavier thus boosting convergence.
8. Many sequencers will likely come to the end of the slot with equal (or almost equal) ledger coverage, which is close to the maximum possible. This will lead to multiple competing branches with equal coverage at the slot edge.
9. To compete for the branch inflation bonus, sequencers must cooperate beforehand.
10. Lazy, isolated, or non-cooperative sequencers have a smaller chance of winning the branch bonus.

9 Security considerations

Let's consider the following scenario: *Alice* sends A tokens to *Bob* in transaction T.

In a distributed ledger with an open and unbounded set of writers to the ledger (fully permissionless), the finality of transactions is probabilistic. It will be converging over time to one chain of branches, with probability to keep several equal branches exponentially diminishing with each slot.

As we do not have an *objective* (deterministic) finality criteria, *Bob* must establish his own criteria for recognizing the finality ("confirmation") of the transaction T before delivering goods, services, or money to *Alice*.

The confirmation criteria would be similar to the Bitcoin's 6-block rule, which aims to ensure finality by waiting for the ledger to accumulate sufficient proof-of-work blocks to prevent a 51% attack.

In the context of our scenario, *Bob* may choose to wait until slot N and several subsequent slots for finality. Specifically, *Bob* could wait until a slot with every branch B with a coverage delta $coverage\Delta(B) \geq \theta \cdot totalSupply$ contains the transaction T.

$1/2 < \theta \leq 1$ is a threshold constant. By setting θ to a value that balances safety and liveness, Bob can ensure that T is included in the heaviest branches, making the transaction more likely be irreversible.

9.1 Safety. Double-spending

The safety question is: under which conditions transaction T can be cancelled, i.e. double-spent?

Here we present a rough model of double-spending, with the gives idea about practical bounds of *Bob's* assumptions.

In *Bob's* situation, all chains of branches $\{B_N, B_{N-1} \dots)$ which satisfy criteria $coverage\Delta(B) \geq \theta \cdot totalSupply$ will contain T: i.e. $T \in B_i$. Let's say C is the heaviest of those chains.

For the network to abandon chain C, the attacker *Eve* has to create an alternative chain $C' = \{B'_N, B'_{N-1} \dots\}$ (indices means same slots) without T, (where $T \notin B'_i, i \leq N$) so that $coverage(B'_N) > coverage(B_N)$.

Let's assume $V = coverage\Delta(B_N)$ and $V' = coverage\Delta(B'_N)$, $V' > V > \theta \cdot totalSupply$

There's up to $H = totalSupply - V$ tokens, which could contribute to C but not to C' (*"honest"* tokens). So, there's at least $E = V' - H$ tokens which where contributed by *Eve* to both C and C'.

$$E = V' - totalSupply + V > \theta \cdot totalSupply - totalSupply + \theta \cdot totalSupply = (2\theta - 1) \cdot totalSupply$$

Eve would need $E > (2\theta - 1) \cdot totalSupply$ of tokens to contribute to C and, in secret, to C'.

If chain C' would be public, it should become main chain earlier, but it did not: *Bob* knows that because he waited long enough after the slot N and didn't receive the chain C'.

So, *Eve*'s secret double-contribution of her tokens to the alternative chain C' without T, is a *"malicious"* behavior (**long range attack**). *Eve* (which also contributed to C) secretly, with other (innocent) token holders which didn't contribute to C, built an alternative chain C' in violation of the biggest coverage rule.

If *Bob* believes, that there's at maximum $(2\theta - 1) \cdot totalSupply$ "malicious" tokens in the network, this conditions could not be satisfied and alternative chain C' would be impossible to build.

If *Bob* naively assumes some $\theta = 1/2 + \delta$, where $\delta > 0$ is small, *Eve* could revert the chain with just a small number $E > (2\theta - 1) \cdot totalSupply = totalSupply + 2\delta \cdot totalSupply - totalSupply = 2\delta \cdot totalSupply$ of tokens.

If *Bob* is over-cautious and assumes θ close to 1, he will be very safe, because then *Eve* would need almost half of the supply in her hands (which is not practical). This safety, however, will have implications for liveness (see section 9.2).

Perhaps reasonable values might be $\theta = 2/3$ or $\theta = 3/4$, which means assumption of no more than 1/3 (1/4) of the *totalSupply* is in hands of *Eve*. It also depends on other conditions, such as capital distribution among particular sequencers and on overall health of the network. The latter could be measured as a fraction of the *totalSupply* actively contributing to the consensus.

Bob's confirmation rule can also take an equivalent form of *waiting for* $coverage(B) > 2\theta \cdot totalSupply$ (instead of using $coverage\Delta(\cdot)$).

The waiting time to make it sure all branches in the slot have been received, depends on the communication latency. It should be estimated more precisely, but one slot should be enough.

Note, that safety problem may appear when *Alice* and *Bob* are different entities. In case transaction is moving tokens in a loop without changing owner, tokens are safe, no matter if transaction is confirmed or not. In particular, for sequencers, there's no need to wait until its transaction will be confirmed (in the sense, described above): this is important only for the non-sequencer entities.

9.2 Liveness

The presented confirmation rule necessitates at least $\theta \cdot \text{totalSupply}$ of tokens actively contributing to the ledger coverage in sequencers. If less than θ of the total on-ledger capital participates, *Bob* may perceive it as a **liveness problem**, meaning his transactions would cease being confirmed.

Inflation incentives motivate token holders to avoid the opportunity cost of holding idle capital that does not generate income. Non-sequencing capital can be delegated to sequencers to contribute to consensus. These factors are expected to result in high capital participation in consensus, potentially nearing *totalSupply*.

One potential approach to alleviating liveness issues is to exclude "sleeping" capital from the calculation of ledger coverage. By introducing additional validity constraints, "sleeping" capital would need to be "awaken" and only after some period of time it could be included in ledger coverage. This could allow the reduction of the malicious token threshold to a fraction of the active capital, similar to proof-of-stake systems with validator security bonding (staking).

9.3 General security. Social consensus

Token holders exchange transactions through nodes, which is the sole category of messages on the network. Sequencers are distinct from other token holders only in the specific way they contribute to the shared ledger.

Token holders controlling the dominant majority of on-ledger capital define the global consensus on the ledger state. The rest of the network follows the ledger version with the largest coverage, which is secured by the majority.

A set of token holders controlling the dominant majority of the total supply can decide on any option allowed by the ledger validity constraints: creating forks, running parallel forks, abandoning one fork, and continuing with another, and so on. As in all distributed ledgers, the protocol cannot prevent this, usually undesirable, possibility. In the cooperative consensus, though, forking and stitching forks together is a routine, low-cost process by design.

The *security aspect* of a distributed ledger is closely related to *decentralization*, which in turn is closely associated with being *permissionless writing to the ledger* (*open participation*), although these concepts are not equivalent.

The observation is that, over time, open participation can lead to a high concentration of influence on the network, such, as hashrate in PoW networks, which, paradoxically, may result in less decentralization rather than more.

In PoS networks, joining and leaving the committee of validators is not as straightforward, which, again, paradoxically, can lead to more decentralized influence compared to completely permissionless PoW networks (although this also depends on other factors, such as how exactly decentralization of the committee is ensured).

The observation above can be seen as a **demonstration of the social consensus** among the dominant Bitcoin miners, who understand that the vast majority of Bitcoin's real value stems from trust in the network and its expected future. Undermining this trust publicly by compromising decentralization through a protocol fork supported by the majority of hashrate would have unknown consequences, likely resulting in irreversible damage to the Bitcoin narrative and substantial losses for everyone invested in it. Therefore, **the fundamental source of Bitcoin's myth (and, ultimately, its market value) is the strong social consensus among dominant miners**. The ledger consensus protocol serves as a tool to coordinate that real-world (social) consensus.

This leads us to another universal observation: the security of any distributed ledger is proportional to the amount of skin-in-the-game committed by a limited set of dominant stakeholders.

In other words, the **security of the distributed ledger ultimately depends on the social consensus among entities with the dominant skin-in-the-game**.

The *cooperative consensus* very likely will follow the universal patterns and will converge to a pattern of high capital concentration, similar to other fully open-participation networks (actually, this concentration may persist starting from the network's bootstrap phase).

As stated earlier, the Proxima design assumes that capital concentration is inevitable and that the network will be run by a limited number (several dozen, probably up to 100) of sequencers, supported by a substantial amount of delegated capital. The fate of the distributed ledger will depend on the social consensus among these dominant players, such as major exchanges, established L2 chains, and other capital custodians that attract other capital (likely a majority) through delegation by non-sequencing token holders.

The bootstrap process of such a network begins with the distribution of the genesis supply to a select group of initial token holders, typically around half a dozen to a dozen. These initial token holders will launch sequencers following the initial coin offering (ICO). Further decentralization of the network will then be driven by market dynamics.

10 Spamming prevention

By *spamming*, we mean all kinds of DDoS-like attacks that aim to overwhelm the network and disrupt its operation. We identify several different targets of such attacks and respective countermeasures.

In general, we rely on two main principles for the spamming-prevention: **Sybil protection** and **limit on transaction-per-second rate per user**.

The *Sybil protection* means, that in related attack vectors, the attacker will need to splits its holdings among many different fake token holders (addresses), which will incur cost of storage deposit (see below).

10.1 Spamming the ledger state

Ledger state spamming refers to the act of issuing many small-amount outputs, commonly known as dust. There are well-known strategies to prevent the dusting of the ledger state. The most efficient method is to enforce a validity constraint that specifies a minimal amount of tokens on each output, based on the byte size of the output. This is known as a **storage deposit**. The storage deposit is refunded when multiple outputs are compacted into one[23].

Proxima adopts the same strategy for dust prevention by enforcing a storage deposit in mandatory output constraints. The token cost per byte is typically a constant in the ledger. However, it may also depend on other deterministic variables, such as the total number of outputs in the ledger state. This allows the cost per byte to be raised when the number of outputs approaches certain limits.

The minimal storage deposit may lead to sub-optimal consequences in the context of tag-along fees. By using an ordinary *chain-locked* output as a tag-along output, a portion of the output is essentially paid as a fee to the sequencer. This can potentially create problems, such as:

[23]see storage deposit in the Stardust ledger on IOTA

- The fee cannot be less than the storage deposit, which may result in an excessively high fee.
- If, for some reason, the target chain never consumes the tag-along output, but the transaction is recorded in the ledger for other reasons, the storage deposit may be lost for the sender and the ledger.

To address the problem, a special type of lock constraint script called the *tag-along lock script* is introduced. It possesses the following properties:

- There is no minimum amount for the output; it can also be a zero-amount output.
- A tag-along output can be consumed by the target chain for the first, say, 12 slots (approximately 2 minutes).
- If it is not consumed within 12 slots, the sender can unlock it for the next 100 slots. This allows the sender to reclaim any unused fee.
- If the output is still not consumed, it becomes consumable by any transaction, enabling a "vacuum cleaning" process to remove any remaining dust.

10.2 Spamming the ledger

By sending tokens in a consume/produce loop, one can issue an overwhelming number of chained transactions that would all be included in the ledger on the UTXO tangle without spamming the ledger state. That could negatively impact node's performance.

To prevent this attack vector, we introduce **transaction pace** constraint for transaction timestamps. The *transaction pace* is a ledger constant that defines the minimal number of ticks between consumed and consuming transactions, as well as between endorsing and endorsed transactions. This limits the number of chained transactions that can fit into one slot. It is reasonable to have different paces for sequencer milestones and user transactions: smaller for sequencers and more significant for users[24].

10.3 Spamming the UTXO tangle

Spamming with conflicts cannot be prevented by ledger constraints, as an attacker could produce potentially unlimited amounts of double-spends from one output and post them to the UTXO tangle.

Our approach to prevent spamming the UTXO tangle is to implement rate limiting per user (per token holder). This is possible, because each transaction (which also is the network message) is signed and identified by the sender's public key.

Nodes are responsible for enforcing rate limits. If a node receives several transactions from the same sender within a real-time period equal to the transaction pace according to its local clock, subsequent transactions from the user are delayed as a form of punishment. The strategy must be applied subtly; for example, transactions should not be punished with delay if they were pulled by the node itself.

Repeated transactions with the same small interval will cause them to be dropped immediately.

This punishing strategy would limit transaction rates per sender to, for example, one transaction per second or even less, which is reasonable for non-sequencer transactions. The natural Sybil-protection of token holders and minimum storage deposit means that every attempt to bypass the rate limiter by creating many accounts will have the cost for the attacking entity.

Determining the optimal constants for the minimum storage deposit and rate limits is subject to simulations and modeling.

Vilnius, 2024

[24] in the experimental implementation sequencer milestone pace is 1 tick, non-sequencer transaction pace is 10 ticks (10 transactions per slot)