

# Dynamic Proof of Weighted Work

---

DPoWW Consensus

Andrew Kessler

June 24, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Concept review: UNiCORNs</b>	<b>4</b>
2.1	Phase 1: Data Collection ( $t_{-1}$ to $t_0$ ) . . . . .	5
2.2	Phase 2: Input Formation . . . . .	5
2.3	Phase 3: Sloth Computation . . . . .	5
2.4	Phase 4: Result Publication ( $t_1$ ) . . . . .	6
2.5	Security Considerations . . . . .	6
<b>3</b>	<b>Concept review: Fireflies as a secure and scalable gossip protocol</b>	<b>6</b>
<b>4</b>	<b>DPoWW protocol</b>	<b>7</b>
4.1	UNiCORN pools . . . . .	8
4.2	Timestamps . . . . .	10
4.3	Initializing the clock . . . . .	11
4.4	Mempool election . . . . .	13
4.5	Standard operation mode . . . . .	14
4.6	Error mode handling . . . . .	16
<b>5</b>	<b>Conclusion</b>	<b>19</b>

# 1 Introduction

You can represent a contract on a computer, or the computer itself can **be** the contract. When a computer executes the logic and enforcement of a contract, it is referred to as a smart contract, as popularized by Ethereum. Similarly, a market can be represented on a computer, or a computer can **be** the market. Such systems are known as smart markets, as demonstrated by AIBlock.

Smart systems are best understood as network computers, machine systems designed to execute protocols that serve groups rather than individuals. As such, they must be architected to ensure the integrity, order, and correct execution of the contracts or markets they host.

To be functional, any computing system requires three core components:

1. An Arithmetic Logic Unit (ALU) for computation,
2. Access to an external clock, and
3. a reliable entropy pool.

In the case of blockchain-based computers, the architecture follows a layered structure:

- Resources (e.g., tokens, files, DePIN etc) are managed via transactional logic (e.g., UTXOs or accounts/contracts).
- Transactions are maintained and validated through blocks governed by a consensus mechanism.

Therefore, clocks and entropy are integrated at the consensus layer, while computation (ALU functions) occurs at the transactional layer. Put simply: *you build with time and randomness, but compute with logic*. This is why entropy and time are foundational but often external to the computation core.

In AIBlock's architecture a Nakamoto style consensus mechanism is used. To ensure a usable clock and derive suitable cryptographically secure pseudo-random number generation (CSPRNG), AIBlock has designed a fixed time Proof of Work consensus mechanism that follows not the longest chain (variable chain length is only achievable through variable block time), but through heaviest chain where multiple proof of works are recorded within each block header.

The process by which each block is formed is carried out inside of a **UN<math>\langle i \rangle</math>COntestible Random Number UNiCORN** mechanism. This allows us to randomly and provably fairly

select which proof of work submitted acts as the block linkage function as well as support the protocol as a seed for CSPRNGs.

This fixed block-time Proof-of-Work (PoW) consensus mechanism launches in a fully distributed state, during which mempool nodes are elected. Once elected, mempools coordinate via a lightweight RAFT protocol, used solely for state replication and uptime management. These nodes are economically incentivized to behave correctly.

If a mempool node fails or acts maliciously, the network leverages validator hash power to resolve the correct chain branch using the heaviest chain rule and to re-elect reliable mempool nodes. Because block validation remains fully distributed, any trust failure is detectable and automatically triggers a fallback to the fully distributed mode.

This architecture is dynamic: it can throttle between a secure-but-slower fully distributed mode and a faster-but-trust-reliant decentralized mode. By adapting in real time, the protocol balances scalability, security, and decentralization—addressing the blockchain trilemma through "just-in-time" coordination.

This paper aims to detail enough of the DPoWW consensus mechanism for the technical reader to be convinced of its validity but not to address the production ready system where a high level overview and perspective is quickly lost. This paper then acts as a primer which, with the core protocol code provide a full understanding of DPoWW.

## 2 Concept review: UNiCORNs

UNiCORNs are a cryptographic protocol designed to generate publicly verifiable and uncontestable random numbers. They are constructed using a *slow-timed hash function* called `sloth`, which enforces a minimum wall-clock computation time, while allowing fast verification. This approach ensures that no participant can bias or precompute the result, making it ideal for high-stakes applications such as lotteries, democratic sortition, cryptographic parameter generation, and trustless public beacons.

The key properties of UNiCORNs include:

- **Verifiability:** Anyone can verify the correctness of the result using a witness provided by the protocol.
- **Unbiasability:** No party, including the coordinator, can influence the outcome if at least one honest contribution is included.
- **Transparency:** The protocol allows open participation (e.g., via tweets) and commits to inputs before the time-delayed computation begins.

- **Security:** It provides strong guarantees under a conservative cryptographic slowness assumption, formalized in the random oracle model.

UNiCORNs are particularly useful in settings where public trust is paramount, and even a single dishonest actor must not be able to compromise the integrity of the random result.

For full details, see:

Arjen K. Lenstra and Benjamin Wesolowski, *A random zoo: sloth, unicorn, and trx*, IACR ePrint Archive, 2015.

The UNiCORN protocol proceeds through three sequential phases over a defined timeline:  $t_{-1}$  (beginning of data collection),  $t_0$  (end of collection and start of computation), and  $t_1$  (result publication).

## 2.1 Phase 1: Data Collection ( $t_{-1}$ to $t_0$ )

During this phase, participants may contribute data—such as social media posts tagged with a specified hashtag. All valid contributions are concatenated in the order received to produce the public string  $s_0$ .

## 2.2 Phase 2: Input Formation

An independent string  $s_1$ , potentially derived from an unpredictable physical process (e.g., a photograph), is generated by the operator running the protocol. The full input to the Sloth function is then formed as  $s = s_0 \| s_1$ , where  $\|$  denotes concatenation.

## 2.3 Phase 3: Sloth Computation

The Sloth function, a deliberately slow cryptographic hash, is applied to the input  $s$ . It is designed to consume a fixed minimum wall-clock time  $\omega$  (e.g., ten minutes), regardless of available parallel computation. Sloth produces the following outputs:

- $c$ : a commitment to the input  $s$ ,
- $g$ : the final hash value, which serves as the UNiCORN output,
- $w$ : a witness that enables efficient verification of correctness.

## 2.4 Phase 4: Result Publication ( $t_1$ )

At time  $t_1$ , the values  $g$  and  $w$ , along with  $s_1$ , are made publicly available.

## 2.5 Security Considerations

The integrity of the UNiCORN output is guaranteed under the following conditions:

- At least one participant provides a valid, honest contribution to  $s_0$ ,
- The commitment  $c = h(h(s))$  is published prior to disclosing  $g$ ,
- The enforced latency of the Sloth function ensures that no adversary can search for alternative values of  $s_1$  within the interval  $[t_0, t_1]$ .

# 3 Concept review: Fireflies as a secure and scalable gossip protocol

**Gossip protocols** are decentralized message dissemination techniques inspired by epidemic processes, where nodes randomly select peers to exchange information. They are valued for their robustness, fault tolerance, and scalability in large, dynamic networks.

**Fireflies** extends traditional gossip protocols by introducing a *secure and scalable membership service* that resists Byzantine failures. It organizes nodes into  $k$  pseudorandomly ordered rings, assigning each node multiple monitors and gossip partners based on secure hash functions. This structure ensures that:

- Each node has a full view of membership, enabling direct communication.
- Accusations and rebuttals maintain consistency and enable fault detection.
- The system tolerates Sybil attacks probabilistically, assuming a bounded fraction of corrupt nodes.

**Utility:** Fireflies is particularly useful for overlay networks requiring secure, scalable, and fully distributed membership management. It ensures high-probability connectivity among correct nodes, adapts to network churn, and defends against malicious behavior without requiring centralized control.

**Reference:** H. D. Johansen, R. Van Renesse, Y. Vigfusson, and D. Johansen. *Fireflies: A Secure and Scalable Membership and Gossip Service*. ACM Transactions on Computer Systems (TOCS), 33(2), Article 5, May 2015.  
[https://www.researchgate.net/publication/277647538\\_Fireflies\\_A\\_Secure\\_and\\_Scalable\\_Membership\\_and\\_Gossip\\_Service](https://www.researchgate.net/publication/277647538_Fireflies_A_Secure_and_Scalable_Membership_and_Gossip_Service)

## 4 DPoWW protocol

The DPoWW protocol uses Single Responsibility Principle (SRP) thinking by breaking up the network protocol into node types whose responsibility is limited. This allows for looser coupling. SRP reduces internal complexity and encourages decoupling. Less interdependency between responsibilities means components can be composed without dragging along unrelated behaviors. Each node type can be independently upgraded, maintained or fine tuned without needing to shut down the entire network. Also, rate limiting processes in one node type does not affect the rate limit of another node type and different vertical scaling or load balancing techniques can be applied to different sub-processes.

The three *public nodes* are:

- **Mempool nodes** which are responsible for packaging new blocks and proposing these blocks to miners for consensus defended validation.
- **Mining nodes** that validate blocks through stack validation and hash rate.
- **ledger storage node** that keeps the ledger and handles audit work.

The three *private nodes* are:

- **Client node** has AIBlock coins or ABC\$ in there wallet and wants to make purchases or trades.
- **Merchant node** has digital assets in the form of AIBlock technologies data resources and wants to sell or trade it.
- **Relay nodes** any well designed server that allows Alice nodes and Bob nodes a way to communicate asynchronously when negotiating or browsing market goods and services.

In figure 1, we see how these six node types relate to each other. As far as achieving consensus goes all private nodes are just where "onspend transactions" come from. These

nodes are discussed in more technical depth when introducing two way atomic trades. It is really the three public nodes that sum together in order to offer a functionally equivalent process when compared to Bitcoin style mining.

All nodes in the network that wish to participate at the protocol level are required to communicate over the Fireflies Gossip protocol at least to initialize. Each node type uses message encapsulation to make sure that only the messages they need to send or receive are pulled off of Fireflies into their local databases (DBs). In a similar approach to Bitcoin, nodes are not all fully connected but connect to random peer nodes and the Proof coordination schema ensures consistency over time.

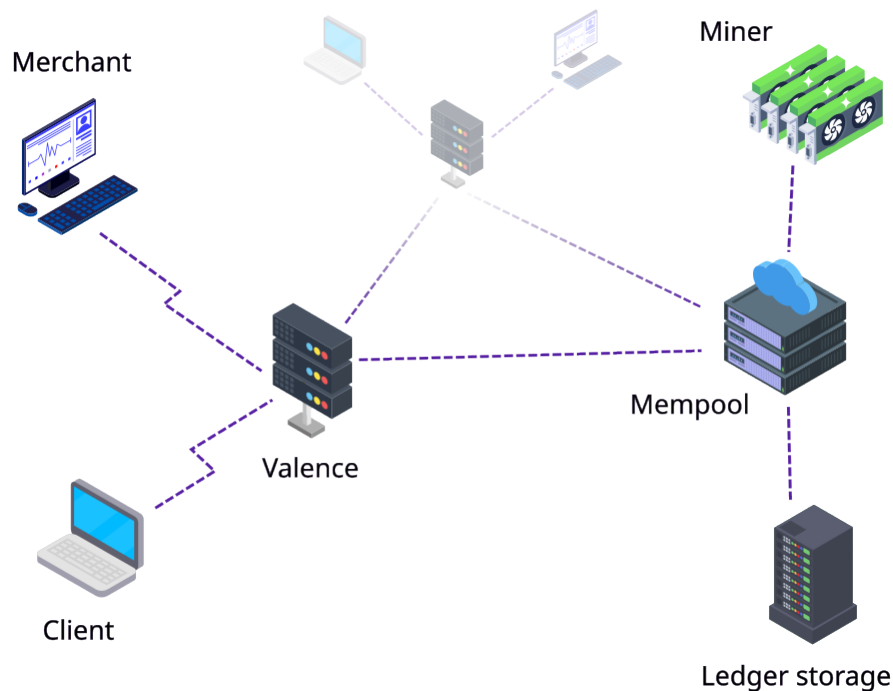


Figure 1: Node types

## 4.1 UNiCORN pools

Referring to Figure 2, we describe how the network initializes into a configuration optimized for higher throughput. A **UNiCORN pool** is a data collection mechanism in which any participant may submit data for UNiCORN ingestion. While only well-formed messages are accepted into the final production pool, strict validation is not critical at this early stage. Its primary purpose is to limit storage overhead. During initialization, no spendable transactions are being mined, eliminating the risk of double-spending or malicious activity.



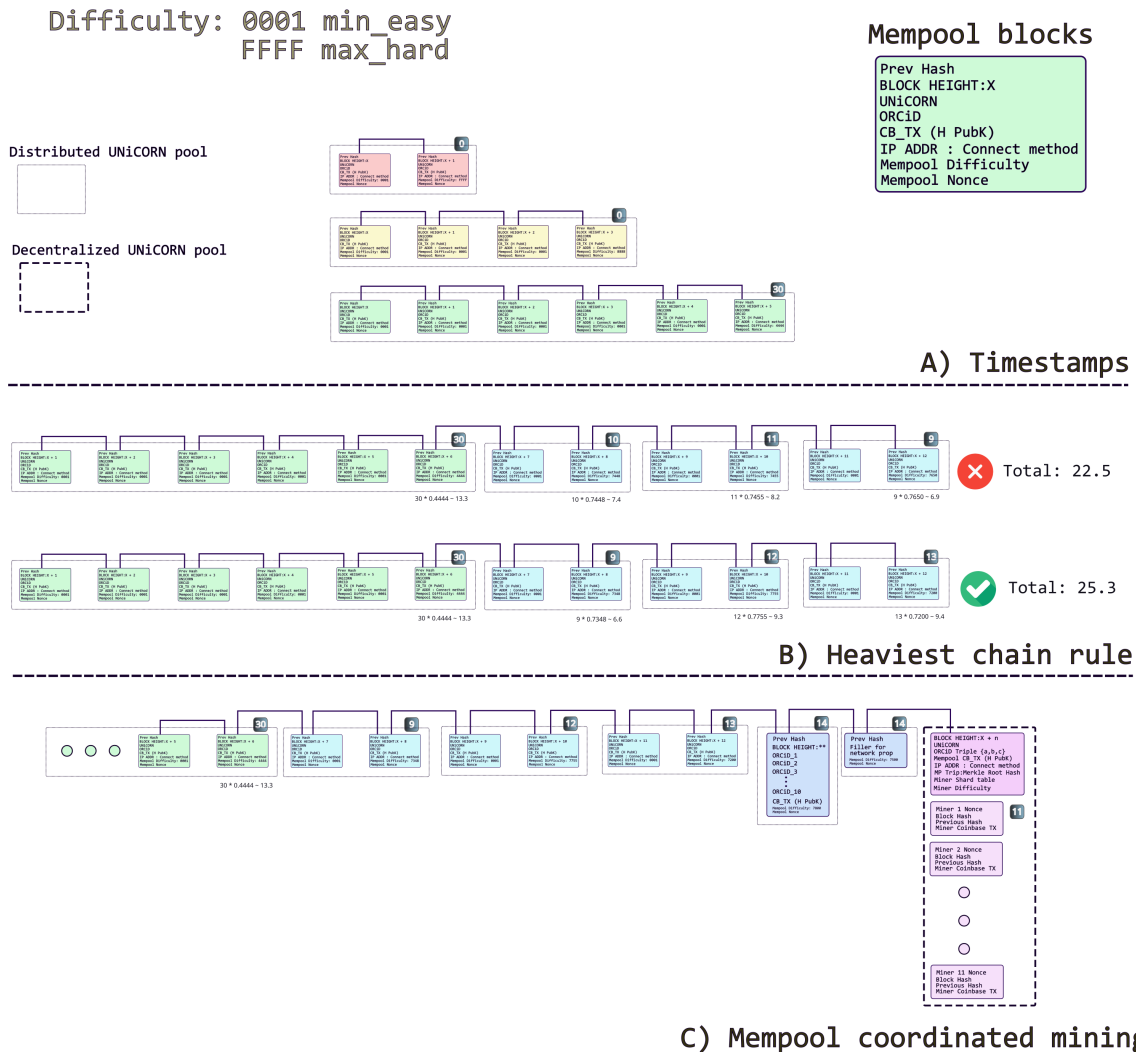


Figure 2: Initializing DPoWW

At this stage, each network participant operates their own instance of a UNiCORN pool. Participants exchange data using a gossip protocol for a predefined collection period. Once this period ends, all relevant gossiped messages are aggregated, ordered, and concatenated to prepare for UNiCORN processing. Since message order cannot be guaranteed in a distributed system, nodes agree on a deterministic sorting method (such as ordering by the numeric value of message hashes in descending order) to ensure consistent results.

Each node independently executes the UNiCORN protocol, acting as an autonomous operator. The expectation is that all nodes, using the same set of ordered messages, will produce the identical UNiCORN output. This collective behavior defines what we call a **distributed UNiCORN pool**.

In accordance with the UNiCORN protocol, each operator must generate a secondary input string  $s_1$ . The complete input to the Sloth function is constructed as  $s = s_0 \| s_1$ , where  $s_0$  is the ordered and concatenated message payload. While  $s_1$  does not need to remain secret, it must be appended correctly in the temporal sequence.

For the initial network launch, we propose using Ethereum's block randomness value as the  $s_1$  input. Specifically, the randomness will be sourced from the first Ethereum block published after the data collection period begins. An example value is:

0x486157e69879083fc42dbb42199c43cfe768650f8c3ed09556fdd67c5afd00bb

This value is publicly accessible on Etherscan.io. It does not matter whether the block is finalized or eventually orphaned, only that the data cannot be precomputed before  $s_0$  and that it is uniformly known and unpredictable. In future implementations, more diverse entropy sources—such as real-time weather data XORed with stock prices—can be used to generate secure and verifiable entropy for use as the  $s_1$  input.

Once the network is initialized, mempool nodes (elected randomly) serve as access points. Participants send their transaction data to these nodes, while miners retrieve packaged blocks from them. Although all data continues to be aggregated into UNiCORN pools, only the elected mempool nodes connected via a RAFT consensus will actively operate UNiCORN pools. This configuration is referred to as a **decentralized UNiCORN pool**.

Figure 2 illustrates which parts of the initialization process are managed under distributed versus decentralized UNiCORN pool configurations.

## 4.2 Timestamps

We chose August 28, 2025 in ISO 8601 format as *2025-08-28T00:00:00Z* to represent the start of our universal timestamp. For every 30 seconds that pass, the protocol will add one block header to the block height. However, when the network is initializing, stalls or goes through an error state, the addition of a block cannot be guaranteed. Thus, when initialization is finished, stalls recover or error states resolve, the nodes involved in consensus all inject the correct number of filler block headers to ensure that the current block height of the functional network is always canonically synced with the timestamp schema. Thus 01:00 am on the morning of August 28th 2025 at meridian is the block height set in the *phoenix block* (genesis block ideally but as the network is in production the phoenix block is a special update block to be added but acts as a sort of second genesis block) plus 120 (for the number of blocks that have elapsed in the first hour).

In general let:

- $B_c$  be the current block height,
- $B_0$  be the Phoenix block height (i.e., the block at time zero),
- $t_0 = 2025-08-28T00:00:00Z$  be the genesis timestamp,
- $\Delta t = 30$  seconds (block time interval).

Then the total elapsed time since the genesis timestamp is:

$$T_{\text{elapsed}} = (B_c - B_0) \cdot \Delta t$$

The current UTC timestamp is given by:

$$T_{\text{current}} = t_0 + T_{\text{elapsed}}$$

To convert to local time  $T_{\text{local}}$ , apply a timezone offset  $\Delta_{\text{tz}}$ :

$$T_{\text{local}} = T_{\text{current}} + \Delta_{\text{tz}}$$

In this way, block height is strictly equivalent to a UTC timestamp. This allows OP CODES, scripts, and transaction logic to trigger in absolute UTC time.

### 4.3 Initializing the clock

In the initialized network, a small number of decentralized mempool nodes are responsible for maintaining the timestamp and proposing new blocks. Miners signal or vote for which block they believe are far and trustworthy with hash rate. However we need a mechanism to freely elect new mempools at random using a distributed peer based protocol so that should mempools act maliciously or when we initialize, all network participant and miners can choose to put their hash rate behind fairly elected and honest mempools. Thus initially mempools have to mine out filler block headers to maintain the clock and we also use this process to allow the network a way to select at random (using mining) the first list of mempool nodes.

We have two challenges we need to address. Firstly we need to prevent Sybil attacks. We don't want malicious actors spamming the network during election and secondly we want to keep block height synced with UTC time. Thus we execute the following.

The UNiCORN protocol proceeds through three sequential phases over a defined timeline:  $t_{-1}$  (beginning of data collection),  $t_0$  (end of collection and start of computation), and  $t_1$  (result publication).

30 seconds is a very short span of network time to run a fully distributed UNiCORN pool and mine out block headers. Thus we select any multiple of 30 seconds and insert a multiple of block headers appropriately to initialize the clock and elect mempool nodes.

For narrative purposes we assume 60 seconds is enough per attempt to start the clock, assuming 30 seconds to mine out short blockheader chains to submit into the UNiCORN pool and another 30 seconds to sloth hash the concatenated inputs to produce a UNiCORN. In production the correct multiple of 30 seconds is chosen heuristically.

Note that while initially Bitcoin's mining difficulty was set as the number of leading zeros prefixing each block hash ie 0000 prefix meant easy when compared with prefix 0000000000000000 which would be comparatively a lot harder, here for simplicity we show a two byte value where 0000 is easy and FFFF is hard, the bigger the number the harder the block difficulty ( we could read this as 0000 means no prefix required and FFFF means the maximum bit prefix of zero's is required ).

Each attempt to start the clock follows this general schema:

- calculate the block height gap between  $g$  between the current block height and the expected block height.
- create a chain of block headers of length  $g - 1$  using the minimum block difficulty. We don't set this to zero in practice but something minimal. This sets the floor price of a submission because this cost at least must be invested into candidacy.
- suffix the  $g - 1$  chain with a  $g^{\text{th}}$  block of difficulty  $\frac{1}{2^d}$  where  $d$  is half  $|g|$ .
- If no solution is found by any network participant in the allotted 60 seconds, restart which will naturally mean that the new  $g$  needs to be two block headers longer.
- if a solution is found, then there is enough hash rate by at least one participant to advance the block height naturally. New blocks are added at the minimum length, in this example two at a time.

As a worked example referring to figure 2 every Mempool candidate attempts to submit a 2 block chain through the UNiCORN pool in 60 seconds having mined out that chain at maximum difficulty. All mempool candidates failed. Next they increased the length of the chain to 4 blocks and tried to mine it out at  $\frac{1}{2^d}$  or half the maximum difficulty but again they all failed. Finally they increased the length of the chain to 6 blocks and tried to mine it out at  $\frac{1}{2^d}$  or a quarter of the maximum difficulty where 30 mempool candidates found a solution. From now on the clock is initialized. The UNiCORN pool fed CSPRNG selected mempool 7 as the random winner from the UNiCORN submissions as the coinbase winner. That mined chain of six blocks is the official chain we all agree to append too. Now the difficulty is low enough that every submission round is now viable and the difficulty function is subtly adjusted from the coarse halving approach to find on

average 11 candidates per round. This means that there was no Sybil attack. The network was not flooded with candidates and the blockheight it now synced and progressing with UTC time.

## 4.4 Mempool election

In order to uniquely identify Mempool candidates, each block header contains an ORCID like ID and a coinbase transaction. Mempools want to be tracked and accessible because the winners of mempool election will serve the chain permanently or until they produce a bad block and through the longest chain rule miners will go to another mempool list for their blocks.

An **ORCID iD** (Open Researcher and Contributor ID) is a unique, persistent digital identifier assigned to individual researchers. It helps distinguish between contributors with similar names and ensures proper attribution of scholarly work. A typical ORCID iD looks like this: 0000-0002-1825-0097.

ORCID iDs are compliant with the ISO 27729 standard (International Standard Name Identifier - ISNI) and function as both unique identifiers and resolvable URIs. Each ORCID iD corresponds to a URI of the form <https://orcid.org/0000-0002-1825-0097>, which redirects to the researcher's public ORCID profile. This makes ORCID interoperable with web services and machine-readable, facilitating automated data exchange across scholarly communication systems.

To be clear, Mempools don't use an actual ORCID from orcid.org but rather ORCID like structures, using the format and system level thinking of ORCID.

The network can track mempool candidates by their ORCID but also challenge that node to sign a target with their corresponding private key used in their public coinbase transaction to prevent false mempool nodes from spoofing.

The chain growing with the active UNiCORN clock will now run for another 120 blocks, at which point every node in the consensus mechanism will review the last 120 blocks and build a frequency table to see how often each ORCID appeared in the last 120 block. This includes ORCIDs in the table recording who found a proof of work, even if that proof of work was not selected by the UNiCORN powered CSPRNG.

This table is then sorted from the most frequent ORCID to the least. Next, anyone in the network, miner or mempool, can mine out a block of the top 10 Mempools by ORCID frequency and publish this as the initialization block for a substantive block reward.

It may well be that the network has two or more different chains growing, either through deliberate means or through network latency. In either case, we use the heaviest chain rule to resolve which chain is canonical. Figure 2 illustrates a "heaviest chain rule" calculation

where the canonical chain has a total of 25.3 k units of proof of work behind it when compared to its nearest competitor with a weight of 22.5 k.

In production we must honor the fact that latency can fragment the network. Therefore we mine out a filler block after the publication of the elected 10 mempools just to allow the majority of nodes to get the mempool list propagated to them because the network has now initialized and its mode of operation has changed. It is now capable of processing onspend transactions in a more scalable way.

## 4.5 Standard operation mode

Resetting daily at midnight, miners consult their personally held copies of the 10 mempool nodes that are currently known to be in good state and randomly select 3 of them to form a RAFT. The selection is made as follows:

- Under normal operations, one day is represented by the network processing 2880 blocks.
- Miners extract UNiCORN from the daily block height 2760 that occurs at 23h00 UTC+00:00 and feed it into a suitable CSPRNG.
- The miners record a three-byte output of the CSPRNG MOD 10 to coordinate them switching to a new mempool triple.

The first operation the new mempool triples do is to form a raft.

The **raft** protocol is a practical choice for maintaining consistent, reliable, and fault-tolerant server clusters.

Raft is a leader-based consensus algorithm designed to be easy to understand and implement—intended as a more approachable alternative to Paxos. It ensures that a collection of servers, or cluster, agree on a sequence of commands (a replicated log)—essential for fault-tolerant systems.

However, Raft is not suited for decentralized trustless environments where adversarial behavior must be tolerated—making it fundamentally different from blockchain consensus protocols.

The raft protocol here does not in any way contribute to securing the consensus mechanism. In principle the protocol would work "just as well" without the raft. The use of it here is merely practical to ensure the the mempool triple can handle momentary loss of communication, fall over recovery and day to day management of issues that usually befall networks.

Users (merchants and clients) exchange data over a relay node as the messaging layer until they can form a two-way transaction. The payment transaction (customer to merchant) against the receipt (or the DRM needed to redeem a file or digital content). The relay node (or the merchant or client) send the two way transaction to the mempool nodes for processing.

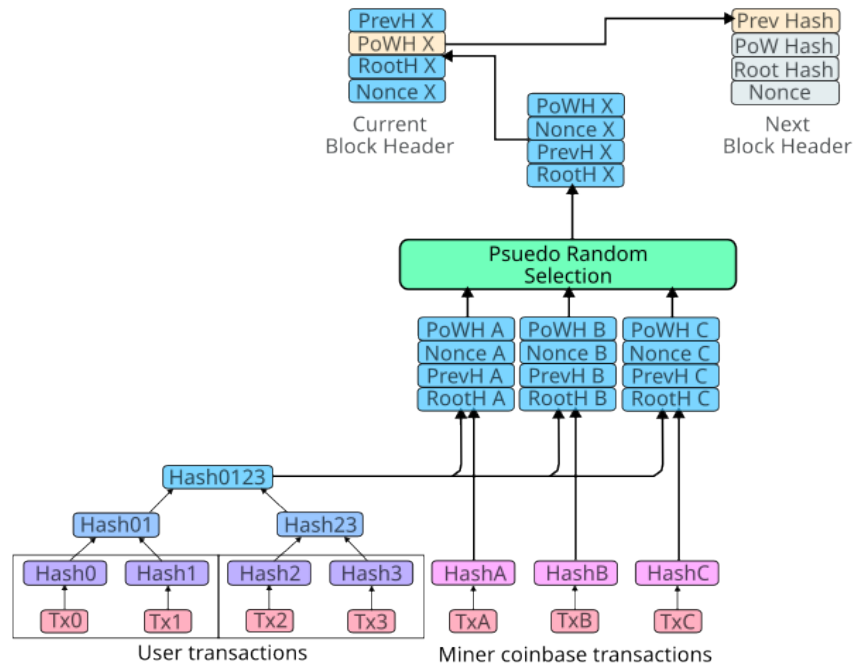


Figure 3: Nested Merkle trees

The Mempool nodes under raft agree that the two way payments are well formed and that there should not be any double spent transactions. They aggregate many such transactions into a block, and compute a mempool merkle tree as seen in figure 3. It is important to note that the Merkle tree structure is nested. The mempool merkle root is denoted Hash0123 in figure 3. This hash value is universal to the entire network for a given block. As such, it is unsuitable for mining, as mining this value cannot "select a miner at random". Each miner takes Hash0123 and the hash of their coinbase transaction (Hash A for miner A, Hash B for miner B and so on) and use these two values to compute the miner's Merkle root, called the Root Hash in figure 3.

Thus, all miner B needs to validate miner A's winning proof of work is Hash0123 (which is the same hash miner A was using to attempt mining out their coinbase) and Tx A (miner A's coinbase transaction).

Several miners will have found a viable candidate proof of work per block and the difficulty function works to adjust the difficulty such that on average 11 miners find a solution in the fixed block time. Which candidate proof of work is the coinbase transaction that is

valid for the network is found by CSPRNG based selection using the latest UNiCORN seed and the valid proof of works submitted.

## 4.6 Error mode handling

There are two types of attacks that can be launched against a Proof of Work coordinated network namely soft attacks and hard attacks. Both are handled by miners using their hash-rate to signal their acceptance of the "more correct" chain. Eventually, the chain with the most weight is accepted by the network, assuming that a simple majority of mining nodes are honest.

Nakamoto style consensus works because a miner who invests time and cost into mining a chain in a bad state will find that other miners won't work off the chain they contributed too because it won't reach majority hash rate and none of their mined coins on the incorrect chain will be spendable on the predominant network. Thus a miner would want the network to know which chain they are mining on to get more hash rate behind their asset base.

In Bitcoin this is done more passively as the longest chain is a very simple piece of data to evaluate and no miner is reliant on another node for its Mempool but the rational actor security and data integrity is the same.

A hard attack would be something like a double spend that is directly measurable. A soft attack would be something like a miner's submission into a UNiCORN not being counted.

A hard attack threatens the validity of the data structure and a soft attack, while still a threat to ideal operations and fairness of the protocol, is not detectable at the final ledger record level.

Hard attacks are resolved when a miner gossips about the attack over fireflies and then proceeds to de-list the offending node from their local membership table and retrieve the block from Mempools that act correctly. If none can be found then that miner seeks to re-initialize a network from new candidate nodes.

The incentive for a mempool to act correctly is significant because they receive block reward just for showing up. Their returns are guaranteed which maximizes their rationality.

Thus the expected error rate for a mempool is low. For both hard and soft errors the solution is that once discovered, an error is gossiped through the network by the miner. Other host miners accept the error, amend their mempool membership lists accordingly and put their hash rate behind the right chain. This will have three possible correction mechanisms that need to be employed namely:

1. Filler block headers may need to be added to synchronism the timestamp.



2. A new mempool candidate (or several) may need to be elected.
3. In the event of a catastrophic attack the entire protocol might need to be restarted.  
(If there are not at least 4 good mempools on the membership list.

Censorship resistance for processing transactions is a good example of a soft attack whereby mempools may not display net neutrality as they ought to and deliberately slow down processing certain transactions over others or even drop transactions. Relay nodes collect transaction fees only from transactions processed by the network and thus they are incentivized to act as a transaction processing watchdog. Ultimately a user could opt to be their own relay node in an extreme situation. Relay nodes start by directly sending a transaction to a mempool node for processing. In the event of non-processing, the transaction is Gossiped for the entire network to witness. The mempool triple have the obligation to pull the gossiped transaction into the next available block. Failure to accept a gossiped transaction will result in several relay nodes and miners to drop the offending mempool triple from their membership table and gossip their new table plus the error to the larger network who must then decide to join them or continue mining out the current block. This will lead to consistency eventually. It may take several offenses to get the entire network across to a different membership list but that will depend on the nature of the claimed offense, the magnitude of it and the will of the network but it will correct over time as per Nakamoto consensus mechanisms.

In the scenario above, the error was gossiped to the mempool as a warning to pick up the transaction immediately of face delisting. In the case of a hard attack such as a double spend, there is not recourse for an offending mempool. The error is gossiped and miners will migrate within one block height onto a new membership list and all coins mined on the incorrect chain from this point onward are guaranteed to be invalid assuming there is no 51 attack.

As detailed in figure 4 three different mechanisms are used to recover the network depending on the attack against the network.

Firstly, in figure 4 clock balancing might be required. For example, two of three of a mempool triple might discover themselves that there is an error in the proposed block due to say the mempool elected leader acting maliciously or unfairly. To ensure that they are not kicked from the network they publish a different block than the malicious mempool node as follows. The malicious mempool (Mempool 1) in figure 4 publishes a bad block list as Mempool triple 1,6,7. Nodes 6 and 7 reject this by forcing a mempool triple re roll through the CSPRNG and publish a filler block to keep the block height synced and the new mempools agreeing with the error signaled create a new minable block at block height  $x + 2$ . We see that one miner found a proof of work and submitted it for the bad chain, while 9 miners found a solution for the good chain. In production it will take several blocks to guarantee that the network is in agreement as to the correct chain. At which point, the heaviest chain is the only chain accessed by honest users and the minority chain with the incorrect data is no longer accessed.

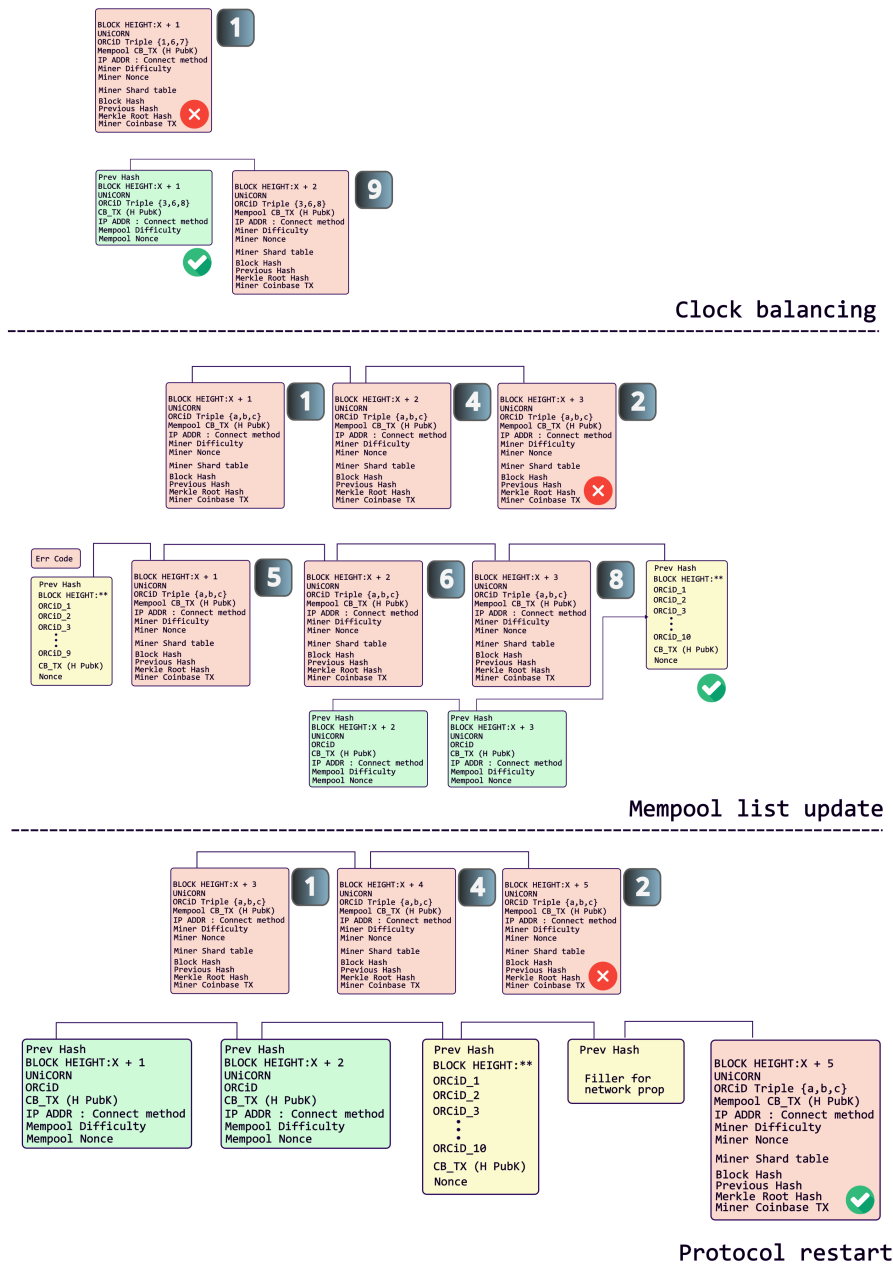


Figure 4: Consensus recovery

Clock balancing might be initiated for more mundane reasons such as clock drift through latency and the need to rebalanced. For such reasons no other action is required by the network. If however it has been detected by miners of fellow mempools that the mempools are acting maliciously, not only does the clock need to be balanced but the network require an updated mempool membership table. This can be achieved in one of two ways depending on the severity of the network failure.

As seen in figure 4, if at least 4 or more mempool nodes are still acting honestly, those fewer nodes will still be able to publish and process blocks correctly while the gossip network elect new mempool nodes to make up the total mempool candidates to 10. These processes run in parallel and as soon as election is finished, a new membership list is merged with the chain and the network continues under normal operations mode.

If three or less mempools remain honest, while bad blocks are being mined out, the network protocol is restarted as if from the phoenix block and once elected it will slowly over time catch up on heaviest chain until it can clearly demonstrate it is the heaviest chain and the other chain is orphaned.

## 5 Conclusion

The Dynamic Proof of Weighted Work (DPoWW) consensus mechanism integrates deterministic timing, verifiable randomness, and adaptive coordination to achieve secure, scalable blockchain operation. By using fixed block intervals and multiple proofs-of-work per block, DPoWW provides a reliable on-chain clock and high-quality entropy source. The UNiCORN protocol ensures unbiased, publicly verifiable randomness for selecting block linkages and CSPRNG seeds. Mempool nodes are dynamically elected and managed, allowing the network to maintain high throughput while adapting to changing conditions. Collectively, these mechanisms enable DPoWW to scale transaction throughput without sacrificing security or decentralization.

Overall, DPoWW addresses key challenges of blockchain design: providing an externalizable clock and entropy source, preventing censorship, and maintaining a robust membership. Future work could include formal security analysis of DPoWW, performance evaluation in real network environments, exploration of alternative entropy sources, and fine-tuning protocol parameters (such as block time or election intervals) for different use cases.