Project Firechain: An Asynchronous, Event-Driven Decentralized Application Platform

Firechain Developers
dev@firechain.io

Abstract

Firechain is a generalized, asynchronous, event-driven distributed computing platform designed for decentralized applications. Its ledger model is a hybrid DAG-blockchain structure, wherein transaction ledgers are segregated at the account level. To strengthen the security and consistency guarantees of the DAG-based account ledger model, Firechain introduces a blockchain-based global validation and local consistency enforcement layer. Firechain uses a layered consensus model based on Bullshark, through which transactions are committed to the network and processed asynchronously. Firechain's VM, the Async VM, is largely compatible with Ethereum's EVM. However, unlike Ethereum, Firechain adopts an asynchronous, event-driven architecture. In this model, information is transmitted through messages between network participants, which greatly improves system throughput and scalability. Firechain supports digital assets natively and enables trust-minimized cross-chain value and data transfers through the Vortex. Rather than charging network participants fees for transacting over the network, Firechain manages resource allocation through the use of a novel fee-less rate limiting mechanism called Heat. Firechain natively supports advanced functionality, including reactive execution, scheduling, a DNS-like name service, state compaction, and highly targeted scaling solutions.

1 Introduction

1.1 Definitions

Firechain is a decentralized application platform that can support complex smart contracts, each of which is technically a state machine with independent state and different operational logic which can communicate asynchronously using a shared global event system.

At the most basic level, Firechain operates as a transactional state machine. The state of the system, also referred to as the *world* or *global* state ($\boldsymbol{s} \in \boldsymbol{S}$), is a composite of the state of each independent account. An event which leads to changes in any account's state is called a *transaction*.

Definition 1.1 (Transactional State Machine (TSM)) *Generally, a transactional state machine is an N-tuple* $(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{g}, \boldsymbol{\delta})$*:*

$\boldsymbol{T}$*, a set of transactions;*

$\boldsymbol{S}$*, a set of account states;*

$\boldsymbol{g} \in \boldsymbol{S}$*, the initial state (i.e. that proposed by the* `genesis block`*); and*

$\boldsymbol{\delta} : \boldsymbol{S} \times \boldsymbol{T} \to \boldsymbol{S}$*, a state transition function.*

The semantics of the TSM's discrete transition system is critical to understanding how the "current" world state is defined.

Definition 1.2 (TSM Semantics) *Semantically, TSMs are discrete transition systems defined as* $(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{s_0}, \boldsymbol{\delta})$*:*

$(\boldsymbol{S}, \boldsymbol{s_0}, \to)$

$\to \in \boldsymbol{S} \times \boldsymbol{S}$ *is a transition relationship.*

Building on these concepts, Firechain is designed as a distributed system with *causal consistency*. A fault-tolerant consensus algorithm enables nodes to come to agreement as to the exact contents of the world state at any given time, even when several malicious nodes are participating in consensus with the intention of disrupting normal operation. In real-world applications, the complete state of any application is generally too large to be frequently transmitted between nodes. Therefore, for scalability and performance reasons, nodes must only transfer a set of transactions, which, when applied in order, are guaranteed to result in the same final state. Firechain organizes related groups of transactions into a specific data structure referred to as a *Ledger*.

Definition 1.3 (Ledger) *A ledger is an append-only log containing a set of transactions with an abstract data type recursively constructed. The mathematical definition is as follows:*

$$\begin{cases} l = \Gamma(T_t) \\ l = l_1 + l_2 \end{cases}$$

$T_t \in 2^T$*: a set of transactions;*

$\Gamma \in 2^T \to L$*: a function for building the current state from that set of transactions;*

L*: a set of ledgers; and*

$+ : L \times L \to L$*: the operation of merging ledgers to determine the composite state.*

In most distributed systems, ledgers represent a group of transactions, rather than an abstractly defined composite state. For example, both Bitcoin and Ethereum use a single shared ledger wherein transactions are globally

ordered using a directed acyclic graph model commonly known as a *blockchain*. As more entries are added to the ledger, modifying any particular transaction in the ledger requires the reconstruction of all later state entries, thereby increasing the cost of tampering with the transaction as time progresses. This immutability guarantee is the basic principle upon which finality is defined.

The same group of transactions may technically result in different (but equally valid) states; however, these variations are the result of incorrectly ordered state transitions. Such variations can result in nodes building inconsistent ledgers. Most distributed ledger networks refer to this as a *fork*.

Definition 1.4 (Fork) *Given two proposed ledgers* $(T_t, T_t{}' \in 2^T$ *and* $T_t \subseteq T_t{}')$*, any case where* $l = \Gamma_1(T_t)$ *and* $l' = \Gamma_2(T_t{}')$*, but where* $l \not\simeq l'$*, it stands to reason that* l *and* l' *are forks.*

The transactional state machine's semantics allow us to prove that, given an initial state and a certain state transition, and where there are no proposed forks of either, every node will eventually reach the same state. So, if a forked ledger is proposed, will it certainly lead to nodes entering a different state? Clearly this depends on the inherent logic of the transaction in the ledger and the methodology by which partial ordering between transactions is enforced. But in today's most popular distributed networks, the unfortunate answer is yes. While some transactions may be commutative, they can still lead to forks due to flaws in the design of the ledger's organizational model.

In cases where the system starts from a given state and two ledgers which result in the same final state but where transaction ordering varies, the outcome is referred to as a *partial fork*.

Definition 1.5 (Partial Fork) *Given the state* $s_0 \in S$*, where ledger* $l_1, l_2 \in L$*,* $s_0 \xrightarrow{l_1} s_1, s_0 \xrightarrow{l_2} s_2$*, any case where* $l_1 \neq l_2$ *but* $s_1 = s_2$ *is considered a partial fork.*

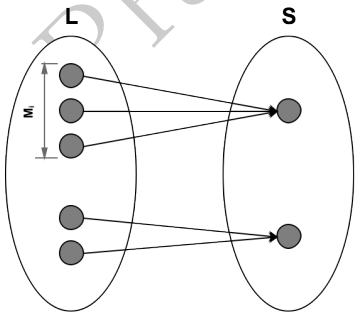

Figure 1: Partial Fork

While the ultimate state is identical in partial fork situations, a proper ledger design should aim to minimize the probability of such cases. This is because when a fork occurs, each node needs to choose one from multiple forked ledgers. In order to ensure global state consistency, nodes must use the same algorithm to enforce the exact selection and ordering criteria. Generally, this is referred to as a *consensus algorithm*.

Definition 1.6 (Consensus Algorithm) *An idempotent state resolution system that, when supplied an arbitrarily ordered set of proposed ledgers, returns the canonical state:*

$$\Phi : 2^L \to L$$

Clearly, the consensus algorithm plays a critical role in a distributed ledger system. A well-designed consensus algorithm should feature a convergence speed high enough to limit variation in cross-node consensus, as well as resilience against a range of attacks.

1.2 Background

While Bitcoin and several of its descendants proposed various designs that accommodate simple uses cases, Ethereum is widely regarded as the best example of a general purpose decentralized computing platform. Ethereum's definition of the world state is $S = \Sigma^A$, a mapping from an account $a \in A$ and the state of an account $\sigma_a \in \Sigma$. Therefore, every state transition is by definition a global one, which means that any observer can compute the current state of any account at any time.

The TSM transition function δ of Ethereum is defined by a set of instructions. These typically take the form of smart contracts, wherein groups of instructions define the logic by which transactions are governed. Ethereum's core innovation is a Turing-complete virtual machine known as the Ethereum Virtual Machine (or EVM). The instructions supported by the EVM are referred to as EVM opcodes. Developers write smart contracts through a programming language called Solidity which is similar to JavaScript in many ways. These Solidity programs are compiled into EVM-compatible instructions and published to the network using a specially crafted deployment transaction. Once the smart contract is successfully deployed, it is accessible at a given address a and defines its own state transition function δ_a.

The EVM represents a powerful innovation and has been widely used in the development of similar platforms. However, it is just another step in the evolution of distributed ledger systems which introduced new issues. For example, there is a conspicuous lack of standard libraries, as well as a host of scalability and security concerns that have plagued its adoption and severely limited scalability.

The ledger structure of Ethereum consists of blocks, each of which contains a list of transactions, and where each new block refers to the hash of the previous block to form a chain structure.

$$\Gamma(\{t_1, t_2, ... | t_1, t_2, ... \in T\}) = (..., (t_1, t_2, ...)) \qquad (1)$$

The most notable advantage of this structure is its very strong resilience against tampering with historical entries. On the other hand, because a single global ledger maintains the complete ordering of all transactions in history, changing the ordering of any two transactions will necessarily result in a different ledger. This has the effect of dramatically increasing the probability of forks. In fact, the definition of the state space of Ethereum's TSM is technically a tree: the initial state is the root node, different ledger candidates represent various vertices, and the leaf node is the canonical final state. In real-world use cases, the vast majority of candidate leaf nodes contain the exact same state, which leads to a large number of partial forks. Partial forks, in turn, produce blocks known as *ommer* or *uncle* blocks. Such blocks represent wasted resources and may potentially introduce ambiguity with regard to the final state.

Initially, Ethereum relied on a Proof-of-Work (PoW) consensus algorithm Φ modeled after that which was first introduced by Bitcoin. Ethereum's PoW algorithm relied on a mathematical problem that is easily verifiable but required substantial resources to solve. It can be generally defined as a mathematical challenge where, given a hash function $h : N \to N$, the correct solution x is that which satisfies the requirement $h(T + x) \geq d$. In this formula, d represents the previous block's difficulty and T represents the set of transactions and their ordering within the proposed block. All proposed blocks required a valid PoW solution satisfying this constraint.

The sum of the difficulty of all blocks represents the total difficulty:

$$D(l) = D(\sum_i l_i) = \sum_i D(l_i) \tag{2}$$

Therefore, identifying the correct state given two forked ledgers is as simple as choosing the one with the highest difficulty:

$$\Phi(l_1, l_2, ..., l_n) = l_m \text{ where } m = \arg\max_{i \in 1..n}(D(l_i)) \tag{3}$$

Proof-of-Work consensus allowed Ethereum to remain secure against a variety of attacks, the most obvious of which are spam and denial-of-service. However, there are at least two major problems with this approach.

The first issue is that solving this type of equation requires a substantial amount of computing resources, which, in turn, leads to a huge waste of energy and, therefore, an increasingly large expense to participate in consensus. The downstream effect of increasing operational costs is a similarly increasing cost of transacting on the network.

The second issue is the algorithm's slow convergence, which directly impacts the system's overall throughput. To illustrate the impact of this problem, one need only point out that the maximum transactions-per-second (TPS) of Ethereum's PoW algorithm was approximately 15, which is at least an order of magnitude less than would be necessary to meet the needs of most user-facing applications.

1.3 Improvements

Following the introduction of Ethereum, the broader community of decentralized ledger developers began to consider and implement improvements to the system from different directions. Starting with the abstract model of the system as proposed by Ethereum, the following areas of focus emerged:

- System state design $\boldsymbol{S}$
- State transition functions $\boldsymbol{\delta}$
- Ledger structure $\boldsymbol{\Gamma}$
- Consensus algorithms $\boldsymbol{\Phi}$

1.3.1 System state design

The basic idea of improving the state of the system is to localize the global state of the world, such that each node is no longer concerned with all transactions and state transfers but rather only maintains a subset of this state. In this way, the potential variance of the sets S and T are greatly reduced, which has the effect of greatly improving the scalability of the system. The most notable network that has taken this approach is Cosmos.

Almost every attempt at improving this design can be generalized as a side-chain based scheme which sacrifices the wholeness of the system state in exchange for scalability. The result is that the decentralization of the system is inherently weakened: the transaction history of a smart contract is no longer saved by every node in the network, but, rather, by some subset of nodes. In addition, cross-contract interactions become a severe bottleneck that limits the scalability of such systems. For example, in Cosmos, interactions in different so-called Zones require a common Hub to complete.

1.3.2 State transition functions

Another direction taken by some projects is to provide more flexibility in system design and the selection of smart contract programming languages. For example, RChain's Rholang is based on π calculus, and smart contracts on NEO can be written in popular languages such as Java and C#. EOS contracts are written in C/C++. While this may seem like a reasonable approach that may offer an easier path for conventional software engineers entering the space, it appears to be having the opposite effect in practice.

1.3.3 Ledger structure

The ledger structure is likely the most promising area of improvement. A single, globally shared linear ledger is technically inferior to a nonlinear ledger that only records partial order relations. The basic concept of a nonlinear ledger model can be described as a Directed Acyclic Graph (DAG). Some earlier projects have attempted to leverage DAGs, including Byteball, IOTA and Nano, and others use

DAGs for a subset of network activity, for example to enable non-public account states. Very few projects based on DAGs have managed to successfully implement smart contracts, in part due to factors such as the lack of a global clock or global consistency guarantees.

1.3.4 Consensus algorithms

With consensus algorithms being the most obvious bottleneck in blockchain systems, it has been an area of intense research and development. Most of the approaches published to date aim to improve the throughput of the system, and one of the key focuses has been to suppress partial forks.

Suppose that C users have the right to produce ledgers, $M = |L|$, $N = |S|$, $M_i = |L_i|$, where $L_i = \{l | f(l) = s_i, s_i \in S\}$.

The probability of partial fork can be determined using the following formula:

$$P_{ff} = \sum_{i=1}^{N} \left(\frac{M_i}{M} \right)^C - \frac{1}{M^{C-1}} \tag{4}$$

Therefore, we can identify two possible ways to reduce the probability of partial forks:

- Establish clear definitions for equivalence of relationships in L of the ledger set, segregate equivalence classes, and construct fewer forked ledgers; or
- Restrict the set of users who have the right to produce ledgers, thereby reducing C and limiting the domain of variance by increasing centralization.

Firechain focuses heavily on the former approach, and the latter has been the primary focus of many other networks. In a purely PoW system, any participant has the right to produce a block. Generally, Proof of Stake (PoS) algorithms limit the power of the production block to those with special privileges, and Delegated Proof of Stake (DPoS) algorithms further limit the set of producers to a typically small group of delegates.

For example, Cardano uses a PoS algorithm called Ouroboros, which has been formally validated and provides strict proofs of its characteristics. EOS's BFT-DPOS algorithm is a variant of the DPoS model which improves throughput by simply producing blocks more frequently. Cosmos uses an algorithm called Tendermint, which, incidentally, is widely used in other projects as well.

2 Ledgers

2.1 Overview

The basic purpose of a ledger is to determine the ordering of transactions, which, in turn, affects the following two aspects:

- **Consistency:** Since the state of the system cannot be defined as a Conflict-free replicated data type (CRDT), not all transactions will be valid when processed out of order, thus the ordering of transactions may lead to the system entering a different state.
- **Finality:** Transactions are packaged into blocks, each of which contains a hash that references another prior block. The order of transactions is a key component in that hash linking scheme. The more times a block is directly or indirectly referenced, the greater the cost of tampering with that block. This is because any change to any transaction therein would necessarily result in a different hash, which would lead to invalid references in all later blocks.

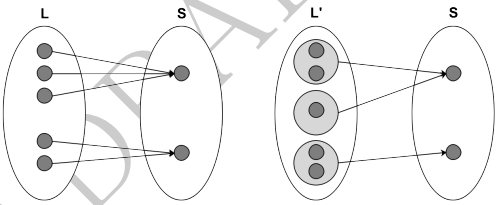

Figure 2: Merging with the canonical state

In general, the two primary objectives in designing a robust ledger model are as follows:

- **Low Collision Rate:** as discussed in the previous section, a significantly lower partial fork rate can be achieved by establishing an equivalency class and regarding groups of changes which lead the system into the same end state as a single changeset. In other words, the system should allow a loose partial ordering relationship between transactions such that they are more easily sequentially exchangeable.
- **Tamper Resistance:** when a transaction t is modified in the ledger l, in the two sub-ledgers of the book $l = l_1 + l_2$, the sub-ledger l_1 is not affected, and the hash references in the sub-ledger l_2 need to be rebuilt to form a new valid ledger $l' = l_1 + l_2{}'$. The affected sub-ledger, then, would be $l_2 = \Gamma(T_2), T_2 = \{x | x \in T, x > t\}$. To put it more plainly, the cost of tampering with transactions is proportional to the strength of the partial ordering relationship. Thus, it is necessary to maintain a reasonably strong partial order relationship between transactions as much as possible in order to increase the overhead and scope of $|T_2|$.

Clearly, the above two objectives are contradictory, and, therefore, certain trade-offs must be made when designing the ledger model. The minimum viable approach is a typical set-based structure which is common in centralized systems.

The following figure represents the spectrum of ledger models and their relative trade-offs.

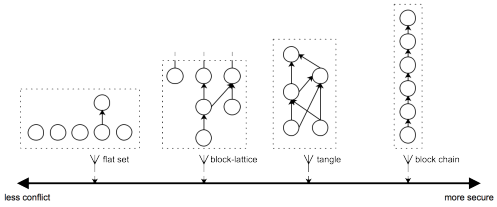

Figure 3: Comparison of Ledger Models

Set-based ledgers effectively have a near-zero partial fork rate, but the trade-off is that tamper resistance is almost non-existent. The opposite end of the spectrum would be a typical blockchain structure, which offers strong tamper resistance at the cost of a high partial fork rate. Everything in between can be classified as DAG ledgers, common examples of which include the block-lattice design popularized by Nano and the Tangle by IOTA. A block-lattice maintains relatively weak partial order relations which makes for an easily justified foundation for high-performance ledger systems. However, it also has relatively weak tamper resistance, which increases the surface area for both malicious attacks and unintended state changes.

Firechain adopts a hybrid approach. In pursuit of high performance, it uses a modified block-lattice DAG ledger structure for account interactions. To address the relatively weak security guarantees of the DAG alone, we introduce a canonical global state chain which maintains references to account-level transactions. Each of these design choices will be discussed in detail next.

2.2 Constraints

First, let's take a look at the precondition of using this ledger structure for the state machine model. This structure is essentially a combination of the entire state machine as a set of independent state machines, where each account corresponds to an independent state machine, and each transaction only affects the state of an account. Transactions are grouped and organized into account-specific chains. Therefore, we have the following restrictions on the state S and transaction T in Firechain:

Definition 2.1 (Degrees of Freedom) *At any given point in time, the system state $s \in S$ is defined as the multidimensional vector $s = (s_1, s_2, ..., s_n)$, which is a composite of the state s_i of each account. For $\forall t_i \in T$, after a given transaction t_i has been executed, the system state transitions as follows: $(s_1', ..., s_i', ..., s_n') = \sigma(t_i, (s_1, ..., s_i, ..., s_n))$. In order to achieve this transition, we must satisfy $s_j' = s_j, j \neq i$. This is referred to as the Single Degree of Freedom Constraint.*

Intuitively, a single degree of freedom transaction will only change the state of one account without affecting the state of other accounts in the system. In the multidimensional vector space of the world state, a transaction is isolated such that the state transition moves only along axes parallel to its origin.

It is important to note that this definition is more stringent than that of transactions under the models of Bitcoin, Ethereum and other popular networks. A transaction in Bitcoin necessarily changes the state of at least one account and potentially more; similarly Ethereum transactions may mutate the state of any number of accounts through message calls and other interactions. Therefore, determining the current state of any given account requires a comprehensive reconstruction of the account's entire history, which, in turn, will almost certainly require the same for other accounts, and so on. Given the open nature of common decentralized applications, the scope of this task can quickly expand and become untenable even in cases where the accounts of interest have made just one such transaction.

When enforcing the Single Degree of Freedom constraint, the possible relationships between transactions can be simplified to just three types. Any two transactions are either *orthogonal*, meaning their ordering has no bearing on the final state, *parallel*, meaning they affect the same state space and, therefore, the order of operations may be significant, or *causal*, meaning a specific ordering is necessary for both to be valid. This constrained relationship model provides a clear set of conditions for grouping transactions by accounts.

Here is an example to illustrate the difference between these approaches:

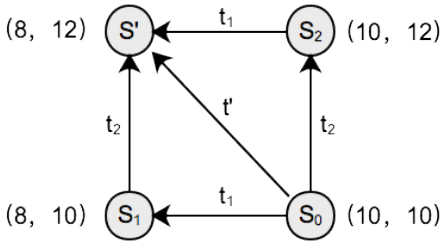

Figure 4: Single Degree of Freedom

Suppose Alice and Bob each have $10 USD respectively. The initial state of the system is $s_0 = (10, 10)$. Should Alice transfer $2 to Bob using Bitcoin or Ethereum, a single transaction t' will cause the system to transition directly into the final state $s_0 \xrightarrow{t'} s'$.

On Firechain, a transaction t' that changes the state of two accounts is not possible, because it does not conform to the single degree of freedom constraint. Therefore, the proposed interaction must be split into two transactions:

1) A transaction t_1 that initiates a transfer of \$2 USD sent by Alice; and

2) A transaction t_2 that acknowledges receipt of \$2 USD sent by Bob.

As such, there are two different paths $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s_0 \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$. These two paths are respectively passed through the intermediate state s_1 and s_2, and these intermediate states are the mapping of the final state s' in their respective account dimensions.

In simpler terms, this means that if you only care about the state of a certain account, you need not be concerned with the transactions of any others.

Definition 2.2 (Circuits) *A pair of related transactions which together result in the intended final state, called the request and its corresponding response transactions, are referred to as a circuit. The account responsible for initiating the request transaction t is recorded as $A(t)$; the counterparty who sends the corresponding response transaction $\widetilde{t}$ is recorded as $A(\widetilde{t})$.*

Given the above definition, it becomes clear that the set of possible relationships between any two transactions in history is quite small. In fact, there are only three possible relationships, which can be defined as follows:

Definition 2.3 (Transaction Relationships) *The possible relationship between two transactions t_1 and t_2 are:*

***Orthogonal**: If $A(t_1) \neq A(t_2)$, the two transactions are orthogonal. This is recorded as $t_1 \perp t_2$;*

***Parallel**: If $A(t_1) = A(t_2)$, the transactions are parallel. This is recorded as $t_1 \parallel t_2$;*

***Causal**: If $t_2 = \widetilde{t_1}$, then the two transactions are causal. This can be recorded as $t_1 \rhd t_2$, or $t_2 \lhd t_1$.*

These definitions are important to understand prior to the introduction of Firechain's unique ledger model.

2.3 Defining Firechain's Ledger

First, let's clarify the characteristics of the partial ordering relationship between any given set of transactions in Firechain:

Definition 2.4 (Partial order of transactions) *One can represent the partial order relationship between any two transactions in the network's history as a dualistic relationship $<$:*

A response transaction must follow a corresponding request transaction :$t_1 < t_2 \Leftrightarrow t_1 \rhd t_2$;

All transactions in an account must be strictly and globally ordered: $\forall t_1 \parallel t_2$, it must hold true that: $t_1 < t_2$, or $t_2 < t_1$.

Due to the strict partial ordering relationship established here, on the transaction set, any set of T is guaranteed to have the following characteristics:

- Irreflexive: $\forall t \in T$, by definition there is no such relationship as $t < t$;
- Transitive: $\forall t_1, t_2, t_3 \in T$, if $t_1 < t_2, t_2 < t_3$, then $t_1 < t_3$;
- Asymmetric: $\forall t_1, t_2 \in T$, if $t_1 < t_2$, then $t_2 < t_1$ is impossible.

Definition 2.5 (The HotDAG Ledger Model) *Firechain's ledger model can be described as a strict partial order set which is composed of the set T of transactions and a partial poset $<$ of the ledger state. This concept is well represented as a DAG structure, such as shown in the following figure.*

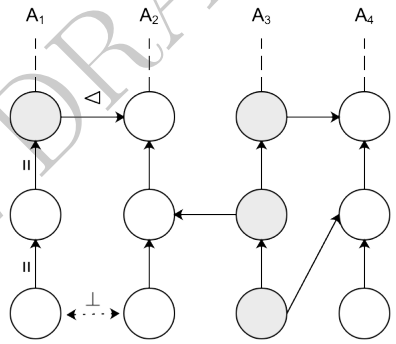

Figure 5: Transaction relationships in the HotDAG

In the graph diagram above, each node represents a transaction and vertices indicate dependencies between transactions. For example, $a \rightarrow b$ indicates that transaction a depends on transaction b.

The ledger model described here is structurally and conceptually very similar to a block-lattice. Account interactions are divided into request and response transactions, each of which corresponds to a separate block, each account Ai corresponds to a single chain, a transaction pair, and a response transaction referencing the hash of its corresponding request transaction.

3 Canonical Global State Chain

3.1 Transaction Confirmation

When an account's ledger is forked, the result of consensus may oscillate between two forked ledgers. For example, based on a blockchain structure, if a node receives a longer forked chain, the new fork will be selected as the consensus result, and the original fork will be abandoned. The result is that the transaction on the original fork will be rolled back. Of course, transaction rollbacks are very problematic, and

they can lead to various inconsistencies potentially including double-spends.

To illustrate the risk of such an event, imagine that a business receives a payment, provides goods or services, and then that payment is subsequently retracted. Clearly, the merchant will bear a loss in this situation. Therefore, when a user receives a payment transaction, it is prudent to wait for the system to confirm the transaction such that the probability of retraction is sufficiently low to consider the transaction final.

Definition 3.1 (Confirmation and Finality) *When a transaction is referenced in a global block, it is considered confirmed. After a certain number of confirmation where the probability of the transaction being reversed is below a certain threshold ϵ, the transaction is considered final.* $P_r(t) < \epsilon \Leftrightarrow t$ *is* ***final****.*

Understanding transaction finality is rather complex, because whether a transaction is truly final depends on the implicit confidence level of $1 - \epsilon$. Not every transaction has the same risk of loss in the case of a reversal; for example the seller of a high-value item and a donation recipient simply would not be equally impacted should the transaction be invalidated. In this way, the ϵ necessary to consider the transaction final may vary dramatically depending on use case.

This is a well known concern in distributed ledger networks, and it's common for merchants to require a certain number of confirmations in order for a transaction to be considered complete. In Ethereum and other blockchains, the number of confirmations indicates the depth of a transaction in the blockchain. That is, the number of new blocks that have been added subsequent to the block containing the transaction in question. The greater the number of confirmations, the lower the probability of the transaction being reversed. As a result, merchants can implicitly set the confidence level required for transactions by simply not accepting deposits until a desired threshold of confirmations has been crossed. This is acceptable in a blockchain ledger because the probability of reversal decreases with time due to the hash-based relationship of blocks. As new transactions are constantly being added to a single global ledger, the probability of any block being tampered with will naturally decrease over time.

Conversely, because a block-lattice type ledger groups transactions by account, the depth of a given block will only increase when the same account has additional activity. In other words, transactions sent by other accounts will have no effect on the confidence of a given account's transactions. Therefore, it is necessary to implement certain consensus rules that reasonably address the inherent risks in this ledger design.

Nano, as previously noted, introduced the first block-lattice implementation which is backed by a voting-based consensus algorithm wherein transactions are validated by a set of delegate nodes. Each node has a certain amount of delegated voting power, and once a transaction has received enough votes, it is considered to be confirmed. However, there are a few problems:

- The time for a given transaction to reach the threshold can't be reasonably estimated as it relies on factors that are completely uncontrolled by almost every network participant, such as the number and relative voting power of nodes online at any given time.
- It is possible that a transaction may never receive enough votes to be fully confirmed but have received too many votes to be ignored outright, resulting in undesirable outcomes up to and including funds being stuck in limbo forever.
- If a higher degree of confidence is desired for any subset of the network, the threshold of voting must be raised globally, which exacerbates the above concerns.
- The probability that any given transaction can be reversed does not actually decrease with time, because the cost of overriding a historical vote does not implicitly change over time.
- Because historical voting data is not persisted in the ledger, there is no reliable way to estimate the probability of reversal for any given transaction.

Firechain addresses these concerns by referencing newly confirmed account transactions in a canonical global state chain. As a result, the HotDAG ledger attains a similar level of tamper resistance as pure blockchains while exhibiting the speed and scalability of a pure DAG model. Furthermore, because global blocks are known to all nodes in the network, conditions for finality can be unambiguously defined. In this way, the global state chain is one of the most powerful components of the HotDAG ledger model.

3.2 Definition of the Global State Chain

The global state chain is the most important storage structure in Firechain. Its main function is to maintain the consensus of Firechain ledgers.

Definition 3.2 (Global Blocks and Global State Chain) *A global block is a structure that captures the latest state of any account that has had any interactions on the network since the prior global block. The account state includes the balance of the account, the Merkle root of the storage state if the account is a contract, and the hash of the last block in the account's chain. The global state chain, then, is a chain structure composed of global blocks, where each block refers to the hash of the previous block.*

The state of an account includes the current balance of its token holdings and the hash of the account's last block. In the case of contract accounts, it also contains the Merkle root of its storage state. Account states are represented in global state as follows:

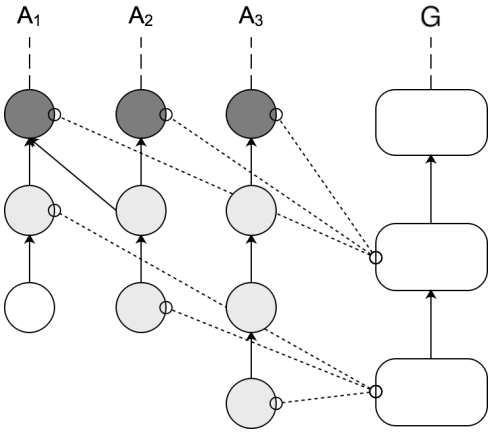

Figure 6: The Global State Chain

```
struct AccountState {
  // balances of tokens held by the account
  map<uint32, uint256> balances;

  // Merkle root of the contract's state
  optional uint256 storageRoot;

  // hash of the last transaction
  // of the account chain
  uint256 lastTransaction;
}
```

The structure of global blocks is defined as follows:

```
struct GlobalBlock {
  // hash of the previous block
  uint256 prevHash;

  // account states
  map<address, AccountState> state;

  // signature
  uint256 signature;
}
```

Note that the structure used to record balances in Firechain's account state is not a single balance value; rather, it is a mapping of token IDs to balances. In this way, global account state entries can support balances for multiple native tokens.

3.3 Global chain and transaction confirmation

The inherent security flaws of the block-lattice model are well compensated by the introduction of the global state chain. For an attacker to successfully execute a double-spend attack, in addition to rebuilding the hash references in the account's own ledger, the entire global chain would also need to be rebuilt for all the blocks after the first global block containing a reference to the transaction in question, as well as to produce a longer global chain such that other nodes will accept it as the canonical chain state. In this way, the cost of any such attack is dramatically increased and the probability of success approaches impossibility.

In Firechain, the confirmation mechanism of transactions is similar to Ethereum:

Definition 3.3 (Transaction Confirmation in Firechain) *Once a transaction is referenced in any global block, the transaction is confirmed. Therefore, the number of confirmations a transaction has is equal to the depth of the first global block in which it is referenced.*

Under this definition, a transaction's confirmation count will necessarily increase by 1 every time the global chain grows, and the probability of reversal decreases to near zero after approximately 5 such confirmations. In this way, it is possible to customize the threshold for finality by simply waiting for the requisite number of confirmations. It is possible to define this requirement at the contract level, and the network will automatically withhold the transaction from processing until the desired thresholds has been reached.

The global chain itself relies on consensus. In the event of a global chain fork, the fork with the most weight, i.e. the longest chain, is considered the canonical chain. As with any blockchain, if the global chain is successfully forked, some previous state will necessarily be rolled back. In this way, the global chain is the cornerstone of the network's security, and as such these events should occur only under extremely rare circumstances.

3.4 Compressed storage

Clearly, it is not sustainable to capture all account states in every global block because the storage requirements would rapidly balloon to an unmanageable level.

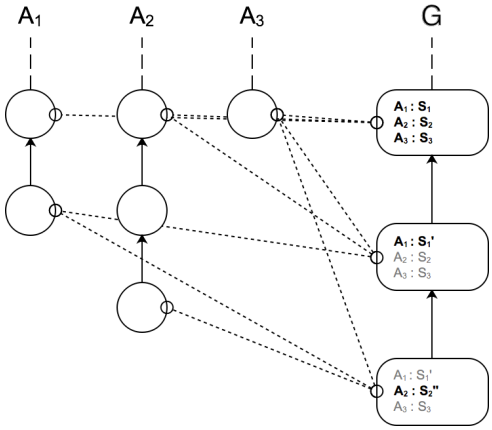

Figure 7: Naive Global State

Firechain uses a simple incremental storage approach to reducing global chain storage space. That is to say, a global block records only a delta changeset for account states. In other words, if there have been no transactions for an

account between the two global blocks, the latter need not contain any reference to that account.

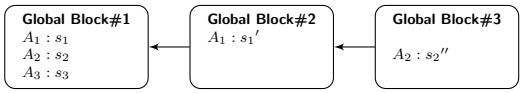

Figure 8: Incremental Global State Updates

Only the final state of each account is persisted in the global chain. In other words, at most one entry per account will be captured in any given global block whether an account sends 1 or 1,000 transactions since the last global block. Therefore, a global block takes up to $S * A$ bytes in maximum. Among them, $S = \text{sizeof}(s_i)$, is the number of bytes occupied for each account state, and A is the total number of system accounts. If the average ratio of active accounts to total accounts is a, the compression rate is $1 - a$.

4 Consensus

4.1 Design Goals

When designing a consensus protocol, the following factors are necessary to consider:

- **Performance**. The highest priority goal for Firechain is to support real-world production use cases, and meeting that goal demands a highly performant network. In order to ensure a consistently high level of throughput and to minimize latency within the system, the consensus protocol must feature a high convergence speed.
- **Scalability**. Firechain is intended to be an open platform that can power applications of any scale, therefore, flexibility of scaling solutions is a key requirement. While the ability to scale the network as a whole is obviously important, one of Firechain's primary goals is to enable much more granular and narrowly targeted scaling solutions, such as at the application or consensus group level.
- **Security**. Given the open nature of decentralized ledger networks and the permanence of transaction outcomes, it is necessary to provide strong security guarantees. At a minimum, Firechain must be resilient to typical threats seen in the wild, such as Sybil attacks, double-spending, so-called 51% attacks, and DoS and other network disruption attacks.

The security of Proof of Work (PoW) consensus is well established. Under this model, consensus, and therefore security, can be maintained so long as at least 50% of all contributing nodes are honest and malicious nodes control no more than 50% of the network. However, PoW is not a viable approach for two main reasons: the convergence speed is too slow and, therefore, performance-related requirements must be relaxed to compensate, and the energy waste needed to secure a PoW network is simply untenable in today's climate.

Proof of Stake algorithms remove the requirement to provide proof of computational effort, which improves convergence speed, increases the cost of executing one-off attacks, and dramatically reduces the energy consumption of the network's validators. However, the scaling potential of most PoS solutions is limited, and the so-called *Nothing at Stake* problem is challenging to address.

Some Byzantine Fault-Tolerant (BFT) algorithms perform better than PoW and offer stronger security guarantees than PoS, but the hurdle of scalability remains. Thefore, BFT-based networks are generally only suitable for private consortia and other closed networks where scaling isn't critical.

Delegated Proof of Stake algorithms offer balanced performance and scalability while lowering the probability of partial forks by constraining the set of validators capable of producing blocks. As a result, DPoS's security level is somewhat lower than PoS and BFT consensus models, but they generally outperform other models in most practical ways. The most notable weakness in this model is that no less than 2/3 of the network's validators must be honest in order to maintain consensus.

Firechain's consensus model takes a layered approach to security and is based on DPoS. This model, called Layered Delegated Proof of Stake (LDPoS), complements Firechain's asynchronous design well and offers a unique blend of the three primary design goals which should meet the needs of most real-world use cases.

4.2 Layered Consensus

The goal of LDPoS is to functionally decompose the consensus function Φ:

$$\begin{aligned}\Phi(l_1, l_2, \ldots, l_n) = \Psi(&\Lambda_1(l_1, l_2, \ldots, l_n),\\ &\Lambda_2(l_1, l_2, \ldots, l_n), \ldots\\ &\Lambda_m(l_1, l_2, \ldots, l_n))\end{aligned} \tag{5}$$

$\Lambda_i : 2^L \to L$ is referred to as the *local consensus function*, and, therefore, the result is called *local consensus*;

$\Psi : 2^L \to L$ is referred to as the *global consensus function*, and it is responsible for further validating local consensus results. Its output is called *final consensus*.

With this separation in place, the system's consensus is segregated into two independent processes:

- **Local consensus**: the process of validating account blocks corresponding to *request* and *response* transactions at the user account or contract account level, which includes proposing updated account ledgers to global consensus.
- **Global consensus**: the process of validating and including by reference the local consensus results,

which includes the creation of global blocks. In the event of a local consensus conflict, the global consensus function is responsible for determining which is the correct state.

4.3 Block Production and Consensus Groups

The ledger structure of Firechain is organized into account-specific chains, backed by a single global chain. This design offers a convenient way to define the most appropriate validator set for both local and global consensus: the right to produce account blocks should generally be granted or otherwise controllable by the account's owner, and the right to produce global blocks should be subject to a democratic election process. While it is technically possible for user accounts to be locally validated exclusively by the account holder, forcing such a task on most users would be unreasonable. Therefore, it is necessary for account holders to have the ability to delegate that responsibility to others who are willing and able to perform the role on the user's behalf. Clearly, a unified process of generating and validating blocks is desirable, and this is the main reason for the introduction of *consensus groups*.

Definition 4.1 (Consensus Group) *A consensus group is a 4-tuple* (L, U, Φ, P) *which describes the consensus mechanism of some subset of the network's accounts or the global state chain.*

L *represents either the global chain* A *or a set of account chains* A_s *in the ledger* $(L \in A|\{A_s\})$*;*

U *is a party with the right to produce blocks for the chain(s) defined by* L*;*

Φ *specifies the consensus algorithm itself; and*

P *specifies the parameters of the consensus algorithm.*

As shown in the above definition, consensus groups can offer a great deal of flexibility by applying a particular set of consensus parameters designed to meet a specific goal. In fact, it is possible for consensus groups to implement entirely different consensus rules from the broader network, so long it remains compatible with the global consensus algorithm.

4.3.1 The Global Consensus Group

The consensus group of the global chain is called, unsurprisingly, the global consensus group. It is by definition the most important consensus group in Firechain. The global consensus group's Φ is bound to the LDPoS algorithm, and it represents the final layer Ψ in the LDPoS design. The number of validators in the group and the block creation interval are specified by the parameters P.

Initially, Firechain's global consensus group contains 25 validators and is expected to produce blocks at a 1-second interval. Under these parameters, a typical transaction will be validated within approximately one second and be considered final after approximately 5 seconds.

4.3.2 Private Consensus Groups

Private consensus groups produce transaction blocks on behalf of one account or a set of accounts controlled by the same party. By default, user accounts act as their own single-member private consensus group, which is to say that the account itself is responsible for maintaining its own ledger and propagating changes for validation at the next layer of consensus. In order to fulfill this role, the user must run a local Firechain node such that they may interact with the broader network.

The clear advantage of private consensus is that the probability of unintentional partial forks is very close to zero. This is because only the user (or their trusted agent) has the right to produce blocks against its ledger, which means the only way for a fork to occur is when the user attempts to propose a competing ledger, whether maliciously or as a result of some type of software error. On the other hand, the biggest disadvantage of this approach is that the user's node must be online in order for their transactions to be validated and propagated.

It is important to note that private consensus is not applicable to contract accounts. This is because, should the contract's private consensus provider fail to operate as expected for any reason, no other node in the network would be capable of producing response transactions for that contract. The effect would be equal to the application being completely removed from the network for all users unless and until it returns to normal operation. Clearly this goes against the basic principles of decentralized networks, and as such, private consensus is not allowed for contract accounts.

4.3.3 Delegated Consensus Groups

Delegated consensus groups act as sets of designated proxy nodes that are authorized to validate and propagate signed transactions through network on behalf of accounts. Both user accounts and contract accounts are able to be added to delegated consensus groups, and the network maintains one public delegated consensus group which freely performs this role on behalf of any account for which it is appointed. In this way, the public delegated consensus group is operated as a permissionless public service.

Other delegated consensus groups may be created by private node operators and consortia thereof, and it's possible for this type of group to define a customized set of consensus parameters P that serve particular use cases. For example, private node operators may choose to create groups that guarantee faster local consensus than the public group. Because operating such groups isn't free of overhead costs, it is also possible for these groups to apply transaction fees.

Contract accounts are, by default, assigned to the public group, and the owners thereof may choose to assign another delegated consensus group if desired. If desired, any user account may delegate its local consensus authority to any delegated consensus group.

4.3.4 Selective Consensus Groups

It is also possible to create a special type of delegated consensus group which can restrict its validation to accounts of its choosing, rather than performing the role blindly for any account for which it is appointed. These permissioned delegated consensus groups are suitable for a wide range of specialized use cases, including application-specific scaling solutions, private relays and targeted enforcement of white- or blacklists. Because this type of group can also become a point of centralization or censorship, they are explicitly ineligible to receive any type of native network rewards at this time. However, they represent an important role in the network, and it's possible for the applications they serve to compensate the group directly using non-native rewards if desired.

4.4 Consensus Priority

The priority of global consensus is higher than that of local consensus. In the case of a local consensus conflict, the global consensus group's selected fork will prevail. In other words, once the global consensus group seals a particular local consensus result, that result is sealed and final, and it is not possible to trigger a subsequent reversal of that sealed state by presenting a longer local chain.

This is vitally important when considering cross-chain transfers or other out-of-band events. To illustrate this point, should a remote chain's history be rolled back, the corresponding account chain of the relay contract mapped to that remote chain must adjust accordingly. However, if the local consensus results of the relay account have already been captured by the global consensus group, it is not possible to successfully complete the rollback. This may lead to data inconsistencies between the relay and the remote chain.

One way to avoid this scenario is to set a *delay* in the consensus group's P, which instructs the global consensus group to record the local result after a given number of blocks have passed. This approach greatly reduces the probability of such inconsistencies, but it is important to note that this risk can't be mitigated with absolute confidence. Therefore, any Firechain application which deals with out-of-band effects must logically handle any event that may lead to inconsistency.

4.5 Asynchronous Design

The three stages of a transaction's lifecycle are initiation, propagation and confirmation. In order to improve the performance of the system, these three steps should be performed asynchronously. This is because, while the speed of propagation and confirmation are relatively static, the load of transactions pending confirmation at any given time may vary dramatically. By processing each stage asynchronously, the network is more resilient to peaks and troughs of activity, and the overall throughput and stability of the system can be greatly increased.

The transaction model of Bitcoin, Ethereum and similar networks is simple: all transactions are placed in an unconfirmed pool, and miners subsequently packages these transactions into blocks. In this way, the transaction is propagated and confirmed at the same time. As the depth of the blockchain grows, the transaction eventually reaches a suitable confidence level and is considered final.

There are two main problems in this model:

- Transactions are not persisted to ledgers in an unconfirmed state. As a result, unconfirmed transactions are unstable, and there is no easy way to identify which, if any, pending transactions are indeed valid without simulating the outcome of every one.

- There is no asynchronous mechanism for propagating and confirming transactions. Transactions are only propagated once confirmed, and the speed of propagation is therefore constrained by the convergence speed of the consensus algorithm.

Firechain introduces a fully asynchronous transaction model; transactions are first written to the ledger of the sender, propagated throughout the network, and confirmed in three distinct stages. In other words, immediately upon local validation, a transaction can be persisted to the account's chain, and the updated ledger is immediately propagated to nodes throughout the network without being blocked by the global confirmation process. In this way, transactions are irrevocable in the sense that they are known throughout the network almost instantly.

This is similar to a typical producer-consumer model. In the lifecycle of the transaction, regardless of changes in account-level transaction frequency, consensus groups can process transactions at a constant rate. In this way, Firechain ensures the maximum utilization of the platform's resources, which greatly improves the system's capacity and throughput.

5 Async Virtual Machine (AVM)

5.1 EVM compatibility

Given the scale of the Ethereum developer community and the rate of adoption of EVM-based networks more broadly, Firechain aims to offer as much EVM compatibility as practical. As such, most of the original semantics of the EVM instruction set is maintained in the AVM. Firechain's account structure and transaction definition is rather different from Ethereum, therefore the semantics of certain instructions must be redefined. The detailed semantic differences can be found on our website. The most significant difference in semantics is that of message calls.

5.2 Event Driven

In Ethereum, a transaction or message may affect the state of multiple accounts. For example, a contract call may result in state changes across multiple contract accounts at the same time through message calls. These changes occur either at the same time, or not at all. Therefore, Ethereum transactions conform to the ACID principle (Atomicity, Consistency, Isolation, Durability), which incidentally is also a key reason for Ethereum's relative lack of extensibility.

In pursuit of greater scalability and performance guarantees, Firechain adopts a final consistency scheme aligned with the BASE principles (Basic Availability, Soft-state, Eventual consistency). Specifically, Firechain is designed as an Event-Driven Architecture (EDA), wherein each smart contract is considered to be an independent service, and messages may be passed asynchronously between contracts, but no internal state is shared.

As a result, the AVM does not support synchronous function calls across contracts. The instructions affected by this decision are mainly the **CALL** and **STATICCALL** instructions. In other words, calls to remote contracts can't be executed immediately, nor can they be relied upon for the result of any remote execution. Instead, a remote contract call generates a request transaction which is asynchronously handled.

5.3 Smart Contract Language

Most smart contracts on Ethereum are written in a Turing-complete, synchronous programming language called Solidity. To support asynchronous semantics, Firechain introduces an async-first smart contract programming language called Pyro.

Most Solidity semantics are similar in Pyro, with the notable exception of any type of remote operation. Pyro developers can define *observables* and *listeners* to implement asynchronous communication. In this way, Pyro message calls are handled similarly to asynchronous callbacks in other programming languages.

For example, suppose a contract A needs to call the `get()` method in contract B and use the result to update its own state. In Solidity, this functionality can be implemented using a simple function call:

```
pragma solidity ^0.4.0;

interface B {
  function get(uint a, uint b) returns (uint);
}

contract A {
  uint total = 10;
  function test(address addr, uint a, uint b) {
    total += B(addr).get(a, b);
  }
}
```

This works in Solidity due to the synchronous execution model of the EVM. In Pyro, however, this type of call would not assign the result to the variable `value` because it is impossible to know the result of that remote call in real-time. Instead, contract A and contract B communicate asynchronously by transmitting messages between each other. The above example could be expressed in Pyro as follows:

```
/**
 * This example is from the perspective of
 * A, which must know about B's observable
 * events and the signature of functions A
 * might want to invoke. In Pyro, the type
 * 'abstract entity' is analogous to that
 * of an 'interface' in Solidity.
 */
pragma pyro ^0.1.0;

abstract entity B {
  // define B's observable Done event
  observable Done(uint result);

  // define the signature of B.get()
  @public function get(a: uint, b: uint);
}

entity A {
  public $B: B = B('fire:a1b...2c3');
  protected $total: uint = 10;

  function test(a: uint, b: uint) {
    // request the result from $B
    $B.get(a, b);
  }

  // define the handler - note that the
  // function name is not significant
  @trigger($B, 'Done')
  listener onDone(result: uint) {
    // use the return data
    $total += result;
  }
}
```

In contract A, when the `test()` function is called, a message is sent to the contract B. That message will be asynchronously handled and the result will therefore not be available immediately. Therefore, it is necessary to define a handler on A by using the `@listener` keyword which can, in turn, receive the result and update A's local state.

In contract B, the function `get()` performs some internal computation, which may include further asynchronous calls as the case may be, and, upon completion, emits an event `Done()`. This event is observable, which is to say that other contracts may register *listeners* that are asynchronously executed with the same signature as the event itself.

Message calls in Pyro are translated into **CALL** instructions, and a request transaction will be added to the contract's ledger as a result. In this way, Firechain ledgers can serve as a sort of message bus for asynchronous communication between contracts. This approach ensures reliable storage of messages, prevents duplication, and provides a simple and semantically clear way of managing asynchronous operations. Messages sent to a contract by the same caller can guarantee FIFO (First In First Out), which is to say that several calls to the same remote contract will execute listeners in the same order they were made. Notably, messages sent by other network participants to the same contract *do not* offer any such guarantee of a particular order.

It should be noted that the concept of events in Solidity, defined using the `event` keyword, are conceptually different from Pyro's observables. Events are indirect messages that are consumable only by off-chain readers, such as a back-end process or web- or mobile-based user interface. Pyro supports both of these concepts.

5.4 Standard Libraries

Smart contract developers on Ethereum are frequently plagued by the lack of standard libraries in Solidity. For example, there are no canonical solutions available for common use cases such as user signature validation. As a result, many smart contract developers rely on code written by others and for which little to no support or documentation exists. Many decentralized applications have suffered catastrophic losses due to bugs in custom implementations of common patterns that by all accounts should be covered by standard libraries. Indeed, countless billions of dollars of user funds have been permanently lost or stolen due to trivial, avoidable logic errors.

In Ethereum's EVM, contracts can delegate certain internal functionality to another deployed contract using the **DELEGATECALL** instruction to implement library-like functionality, and several so-called Precompiled Contracts exist to perform certain operations that would be impossible or economically infeasible to perform on-chain such as certain types of hashing operations. However, these functions are not designed to address complex requirements.

Firechain will offer a set of standard libraries accessible from Pyro contracts to support use cases such as string processing, floating point numbers, complex mathematical operations, sorting functionality, hashing operations, encryption schemes and so on. These standard libraries are designed to be highly performant and are implemented as native extensions to the AVM. These functions will be accessible using the **DELEGATECALL** instruction.

Firechain's standard libraries can be extended as needed, however, because the system's state machine model is deterministic, it is not possible to support non-deterministic functions like secure random numbers through this mechanism.

5.5 Gas

There are two main functions for Gas in the Ethereum:

- To quantify and constrain the resource consumption by EVM code execution; and
- To ensure that EVM code is eventually halted in all cases.

Interestingly, computability theory asserts that the so-called Halting Problem is an incomputable one within a Turing-complete system. In other words, it is technically impossible to guarantee whether a smart contract can be stopped after limited execution through static analysis alone. Therefore, the concept of gas has been retained in the design of Firechain's AVM. However, there is no such concept as a Gas Price in Firechain, which means there is no fee for using gas within the system. Instead, Firechain implements a dynamic resource sharing mechanism to control usage. This mechanism is called *Heat*.

6 Economic Model

6.1 Native Token

In order to control resource usage and to encourage maximum decentralization through widespread node operation, Firechain implements a native token represented by the symbol *$FIRE*. The whole unit of the *$FIRE* token is referred to simply by the symbol *$FIRE*, while the smallest divisible unit is referred to as a *spark*, $1 FIRE = 10^{18} spark$.

The global state chain is central to the network's security and performance. In order to encourage node participation in global consensus, the Firechain protocol issues rewards for the production of global blocks. When users deploy contracts, issue tokens, or register FNS domain names, $FIRE is consumed. However, sending transactions *does not* consume *$FIRE* in the way that Ethereum transactions consume *ether*; instead, users need only stake in order to obtain heat capacity and thereby the right to send and receive transactions on the Firechain network.

6.2 Resource Allocation

Firechain is designed as an open, permissionless decentralized computing platform, and the functional requirements and capabilities of smart contracts vary widely. For instance, smart contracts have different requirements for scale, availability, and finality. For many smart contracts, these requirements may vary at different times.

On Ethereum, transactions are subject to a typical bidding model for resource usage, which can effectively control the balance between supply and demand in principle. However, it is difficult to estimate market demand at any given time and it is, therefore, very likely that users will fail to assign a reasonable value for any given transaction.

Furthermore, the nature of resource bidding is inherently competitive, and there is no generally accepted way to rationalize the allocation of resources at the account level. As a result, most users simply accept the gas price suggested by their wallet software, which enables market failures to occur quite frequently at the expense of users.

6.2.1 Heat Calculation

We have adopted a dynamic resource allocation model in Firechain, which allows users to ensure resource allocation in four ways:

- Completing a PoW challenge when the transaction is initiated;
- Staking a certain amount of *$FIRE* in the account;
- Burning a certain amount of *$FIRE* for a one-off transaction;
- Other accounts may delegate heat capacity to the user.

Any account's specific heat capacity can be calculated through the following formula:

$$H = H_m \cdot \left(\frac{2}{1 + exp\left(-\rho \times \xi^{\top}\right)} - 1 \right) \tag{6}$$

In this formula, H_m is a constant representing the maximum limit of a single account's heat capacity, which is relative to the total capacity of the system and the total number of accounts. $\xi = (\xi_d, \xi_s, \xi_f)$ is a vector representing the requirement for a given amount of resource usage: ξ_d is the PoW difficulty that the user may use to calculate a solution when submitting a transaction, ξ_s is the amount of *$FIRE* staked by or on behalf of the account, and ξ_f is the one-time value that the user is willing to burn for a transient increase of their account's heat capacity for a single transaction. It should be noted that ξ_f is not a fee, which is to say this value will be permanently destroyed and removed from circulation rather than paid another network participant.

In the formula, the vector $\rho = (\rho_d, \rho_s, \rho_f)$ represents the weight of the ways to obtain heat capacity; that is, the heat capacity obtained by the destruction of 1 *$FIRE* is equivalent to staking ρ_s/ρ_f *$FIRE* for an instant.

If the user neither stakes *$FIRE* nor burns the one-time value, a PoW must be submitted with the transaction. This is because there would otherwise be no available heat capacity to initiate the transaction. This is an effective mechanism to protect against spam and dust attacks, and more generally it protects the system's resources from being abused. This approach enables users to get relatively low heat capacity for practically no cost, thereby reducing the barrier of entry for low frequency users, whereas high-demand users must invest substantial resources in order to maintain sufficient heat capacity for their usage.

6.2.2 Resource Quantification

It is important to note that the global state chain also serves as a global clock. That is to say, the global chain can be used to quantify the resource usage of an account accurately, both instantaneously and over time. For every account block, the hash of the most recent global block is referenced, and the height of that global block is used as the timestamp of the transaction. Therefore, by simply quantifying the delta between two transaction timestamps, the network can determine whether the interval between the two transactions has been sufficient to regenerate the necessary amount of heat capacity for the latter to be valid.

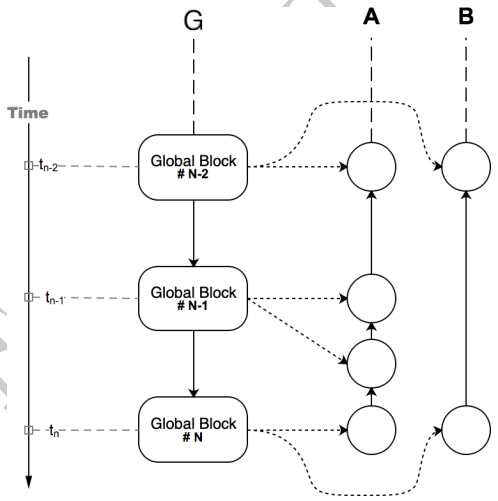

Figure 9: The Global Chain Clock

In the figure above, account A generates four transactions over two time intervals, while account B generates only two transactions over three intervals. Therefore, the average transactions-per-second of A for this period is twice that of B. For simple transfers, the average TPS of an account is sufficient. For smart contracts and more complex interactions, however, each transaction consumes a different amount of resources. Thus, it is necessary to account for the resource usage of each transaction in a more granular way in order to calculate the account's average consumption over a given period of time. The average resource consumption of the recent k transactions in an account chain with a height of n can be quantified as follows:

$$Cost_k(T_n) = \frac{k \cdot \sum_{i=n-k+1}^{n} gas_i}{timestamp_n - timestamp_{n-k+1} + 1} \tag{7}$$

In this formula, for a transaction T_n, $timestamp_n$ is the timestamp of the transaction and gas_n is a measure of the transaction's resource usage. When verifying a transaction, the node responsible for validating it will determine whether the account's heat capacity satisfies the condition: $Cost(T) \leq H$, and in the event it is not satisfied, the transaction will be rejected. In this case, the user would need

to resubmit the transaction after either increasing their heat capacity or waiting for sufficient capacity to be restored.

6.2.3 Delegated Heat Capacity

It is likely that some users will maintain larger *$FIRE* stakes than necessary to cover their own transactions. If so, it's possible for that user to delegate some or all of their heat capacity to others who may in fact need more than they can afford to obtain through staking or PoW solutions. Firechain supports this type of operation natively. In order to delegate heat capacity, a user submits a transaction to the staking contract containing the amount of *$FIRE* stake that should be delegated, the address of a beneficiary, and the period of time for which the delegation is valid unless manually revoked at an earlier point. Once the transaction is confirmed, the heat capacity corresponding to the delegated stake will be immediately available to the beneficiary. Once the delegation period ends, the stake will once again be assigned to the owner and no longer be available to the beneficiary.

While the native solution does not provide the delegating user with any personal benefit, it is possible for users to implement some type of income-generating scheme that automates this workflow, such as a heat capacity leasing service.

6.3 Asset Issuance

In addition to native token *$FIRE* token, Firechain also allows network participants to issue custom tokens. These can be issued through a special type of transaction, namely an Issue transaction. This type of transaction contains the following parameters:

```
Issue: {
    name: "TrashToken",
    totalSupply: 999999999000000000000000000,
    decimals: 18,
    owner: "fire:abc...0123",
    symbol: "TRASH"
}
```

Once the transaction is confirmed, the issuance fee will be deducted from the owner's account and the token will be immediately available for use. The system records the information of the new token and assigns an internally generated *token_id*. The owner may then mint and transfer tokens as desired.

6.4 Vortex Cross-Chain Protocol

In order to support cross-chain value transfer of digital assets and avoid the so-called *value island* problem, Firechain supports a native cross-chain value transfer protocol, called *Vortex*.

In principle, any remote chain with sufficient smart contracting capabilities can be integrated with Vortex such that any asset on that network can be locked remotely and made accessible within the Firechain ecosystem. In order to support cross-chain interoperability, a synthetic token that corresponds to its remote native counterpart is created on Firechain. For example, in order to import *$ETH* from an Ethereum account to Firechain, one would issue a synthetic *$ETH* token on Firechain whose initial supply is equal to the total amount transferred to the Gateway contract deployed on Ethereum.

Every supported remote chain is assigned a Gateway Contract on Firechain which maintains the mapping relationship between relevant Firechain transactions and those of the remote chain. A specialized consensus group is responsible for validating interactions with the Gateway, and the nodes that operate this group are known as Vortex Relays. A Vortex Relay operator must maintain both a full Firechain node and the equivalent type of node for the target chain, and it must monitor and process transactions on both sides of the relay at all times.

In order for a Vortex Relay to work, a synthetic asset on Firechain corresponding to the remote chain token should be created and ownership transferred to the gateway contract. This enables the supply of the synthetic token to be managed exclusively by the Gateway on Firechain such that the system can enforce the 1:1 exchange ratio between the synthetic asset and its remote counterpart. Similar conditions apply to remote Gateways on target chains.

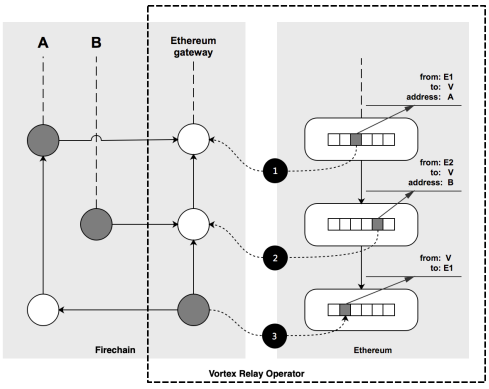

Figure 10: Vortex Cross-Chain Protocol

The figure above describes of the cross-chain value transmission between Firechain and Ethereum. When the Ethereum user $E1$ wants to transfer the token from the Ethereum to the Firechain, it can send a transaction to the Firechain Gateway contract address V, supplying the user's address A on Firechain as a parameter. The balance of the transfer will be locked in the Gateway contract and become part of the exchange reserve. Having observed this transaction, a Vortex Relay node generates a corresponding transaction on Firechain issuing the same amount of the synthetic asset to the user's account A. In the diagram,

① and ② respectively indicate that $E1$ and $E2$ transfer to Firechain account A and B. It is important to note that if the user does not specify the Firechain address when transferring, the cross-chain transfer is invalid and therefore will reject the transaction.

The inverse workflow is represented by ③. The user A initiates a transfer from Firechain to their Ethereum account by sending the desired quantity of synthetic tokens to the Gateway and specifies the remote address $E1$ on Ethereum to which the original asset will be transferred. The Vortex Relay node will initiate a transaction on Ethereum to complete the cross-chain transfer as well as a corresponding transaction on Firechain burning the synthetic asset. On the Ethereum side, the Gateway contract will verify whether this transaction is initiated by a trusted Vortex relay and transfer the original asset from the Gateway contract to the target account $E1$ as requested.

All cross-chain relay nodes are expected to monitor the target network, and in doing so they can readily verify that every cross-chain transaction is correctly executed and reach consensus within the consensus group. However, it is important to note that the global consensus group will not monitor the transaction of the target chain, nor will it verify whether the mapping between the two chains is correct at any point. If the target network experiences a reorg or hard fork, the mapped transactions in the Firechain system cannot be reverted; similarly, if the cross chain transactions in the Firechain are rolled back, the corresponding transaction of the target network can not be rolled back at the same time. Therefore, when doing cross-chain transactions, it is necessary to deal with such events within the contract's logic. At minimum, it must demand a sufficient number of confirmations for incoming requests so as to decrease the likelihood of undesirable outcomes.

7 Other Features

7.1 Scheduling

On Ethereum and other networks, smart contracts are driven by transactions, and the execution of contracts can therefore only be triggered by users initiating a transaction. Of course, many applications require some form of scheduled or recurring execution. Because Ethereum offers no native solution for scheduling execution, some form of external helper is needed to trigger the execution of a contract through a clock. Many such services exist, for instance Gelato and OpenZeppelin Defender offer this functionality. However, myriad problems exist with this approach, not the least of which are that performance and security are not guaranteed.

Firechain natively supports scheduling functionality, and any network participant can schedule transactions using a built-in contract. In this way, both user accounts and contracts can register logic to be executed at a particular time in the future. The public delegated consensus group will use the global chain as the network's clock and send the request transaction to the target contract according to the desired scheduling logic. Pyro supports this functionality through a special `job` definition syntax.

7.2 Naming Service

On Ethereum and other networks, accounts are generally referenced by their public key or some derivative thereof. There are two problems in identifying network participants using these raw address formats:

- An Ethereum address, for example, is a 20-byte identifier with no practical relevance to users;
- Sending and receiving transactions using this type of address is inconvenient to users and introduces several types of attacks related to misidentification of remote accounts, and, indeed, attacks that rely on this fact are quite common;

In order to address these concerns, the developer community has created a non-native service known as ENS, or the*Ethereum Name Service*, which can be used for name-to-address resolution, similar to how DNS translates domain names to IP addresses. Unfortunately, Solidity smart contracts are unable to resolve these names to their respective addresses natively; instead developers must implement address resolution where needed.

Firechain natively supports a naming service through which users can register easy-to-remember names and resolve them to the actual address seamlessly. Names are formatted and organized similarly to web domain names, such as *somefunguy.fire*. Once a FNS name has been registered, the owner can allocate arbitrary subdomains as desired.

7.3 Contract Updates

Ethereum smart contracts are immutable by design. Once deployed, the code of a contract can generally not be modified, even in cases where a catastrophic bug has been identified in the contract. This is very unfriendly to developers and makes continuous integration and iterative updates cumbersome or impossible.

Firechain supports a non-destructive upgrade scheme for smart contracts natively through FNS resolution which remains consistent with the network's immutability guarantees. The process of contract updating includes:

- An updated version of the contract is deployed and inherits the state of the existing version;
- The FNS name of the original contract is updated to point to the new address;
- The original contract is archived and permanently frozen through the **FREEZE** instruction.

These three steps are to be completed at the same time, and the Firechain protocol ensures the atomicity of the operation. Developers need only ensure that each version of their contracts are backward-compatible such that no state is lost.

It should be noted that the new contract will not inherit the address of the old contract. If a transaction is sent directly to the original address, the transaction will still be sent to the old contract. However, because the contract is frozen and archived, the transaction will be rejected. This is because different versions of contracts are technically completely different entities; while only the latest version is capable of processing requests, all historical versions will be retained indefinitely.

8 Governance

For a decentralized application platform, an efficient governance system is crucial to a healthy ecosystem. Firechain's governance model is composed of two separate but equally important parts: on-chain and off-chain. On-chain governance is handled through a voting mechanism, and off-chain governance is the human-driven process of iteration for the network itself.

The on-chain voting mechanism is further divided into two types of votes: global and local. Global voting is based on the balance of *$FIRE* held by the user, which corresponds to a calculated voting weight. Global governance is mainly used for the election of global consensus group nodes. Local governance is generally used by smart contracts. For a contract to leverage the native governance mechanism, it must register a token to be used as the basis for the calculation of voting weight. It can be used to, for example, elect nodes for the delegated or selective consensus group to which the contract is assigned.

Aside from global block production, the global consensus group has the responsibility of choosing whether and when to upgrade the Firechain network. Delegated consensus group votes can be used to signal support for community-level decisions on behalf of users, for example to improve the efficiency of decision making and avoid the all too common failure scenario in which users who stand to gain or lose something as a result of a certain outcome fail to participate in voting for any number of reasons. In cases where delegates do not use their decision-making power according to the community's expectations, users can register their disapproval by voting to revoke their right to perform such a role.

Off-chain governance is a somewhat looser concept, and it is generally realized by the community. Any member of the Firechain community who meets certain requirements as set by the community can propose an improvement plan for the Firechain protocol itself or its related systems. These proposals are called FIP (Firechain Improvement Proposal). FIPs are intended to be widely discussed in the community, and whether to implement the proposed measures is determined by community vote. Of course, community members may differ in their understanding and approval of changes, so it is recommended that large proposals be split into several smaller proposals which can be approved over several rounds of voting in order to gather support.

Although some Firechain participants may not have enough *$FIRE* tokens to have a meaningful impact on voting outcomes, they can still submit FIPs and fully express their views through voting. Those users who have the privilege to sway the outcome of proposals must take full account of the health of the greater community, rather than simply voting for their own benefit. In this way, Firechain's governance process may remain truly democratic and reflect the needs of the entire community.

9 Future Work

Transaction confirmation through a single global chain remains a performance bottleneck of the system. Because Firechain adopts an asynchronous design and DAG account structure, transaction validation can be executed in parallel. However, due to the relationship between certain transactions, the degree of parallelism realized can never reach its true potential. Techniques for improving the parallelism of transaction validation or adopt a distributed verification strategy will be an important direction for future optimization.

There are some shortcomings in the current LDPoS consensus algorithm. The community must continue to investigate optimizations that improve the consensus algorithm, or to be identify alternative consensus algorithms that can be implemented for delegated or selective consensus groups.

In addition, the virtual machine is a very important area of focus for reducing system delay and improving system throughput. Because the design of the AVM is relatively simple, it may be necessary to implement a more powerful virtual machine in the future with more flexibility and resilience to certain types of threats.

Finally, besides the Firechain core protocol, ecosystem tooling and other measures aimed at improving the development experience is another key area of future work. While the Firechain core team has developed many such solutions, there is much work to do to support the future growth of the Firechain ecosystem.