VINE Technical Paper

Proprietary DAG as a decentralized infrastructure of JSUE

v.04

Contents

1. Introduction

Since the introduction of distributed ledger technology (DLT), blockchain has improved security, provided transparency and visibility, and decreased transaction costs. Blockchain today has many use cases, from basic token transactions to private applications for enterprises and governments. Due to various technical limitations, public blockchain vendors are working to solve issues with speed and scalability, considered the core barriers to mass adoption.

Ethereum, the second biggest blockchain, is still working on a solution to improve TPS, but currently, it can only handle around 13 TPS. Due to this limitation, projects such as Polygon, a Layer 2 solution on Ethereum, became popular. Polygon can process up to **7,000 TPS** at a fraction of the cost. Other Layer 1 solutions, such as Algorand, process **3,000 TPS**; the self-proclaimed fastest is Solana reaching up to **65,000 TPS**. In contrast, JSUE can reach **700,000 TPS**, which is not the limit, as its technology allows it to scale even further.
Blockchain is not the only type of DLT on the market. Various projects use DAG (Directed Acyclic Graph) as a foundation for a decentralized ledger. DAG differs from the usual blockchain in its record structure and asynchrony. It is not a blockchain or a consensus-building model because it has no blocks. DAG functions as a network of interconnected branches that expands in multiple directions. Transactions can be confirmed much faster while remaining decentralized since each node only confirms the previous one. Each transaction refers to one or more previous parent transactions. They, in turn, refer to their parent transactions, and so on. Thus, the system knows the exact order between transactions in this chain of transactions. The result is forming a "tree" of transactions, where each is confirmed and immutable.

JSUE aims to solve the shortcomings of DLT and existing DAG solutions. JSUE created several new technical solutions to ensure high transaction processing speeds, historically reliable and secure information storage, implementation of smart contracts, and much more. This solution is called VINE, which is JSUE's decentralized infrastructure.

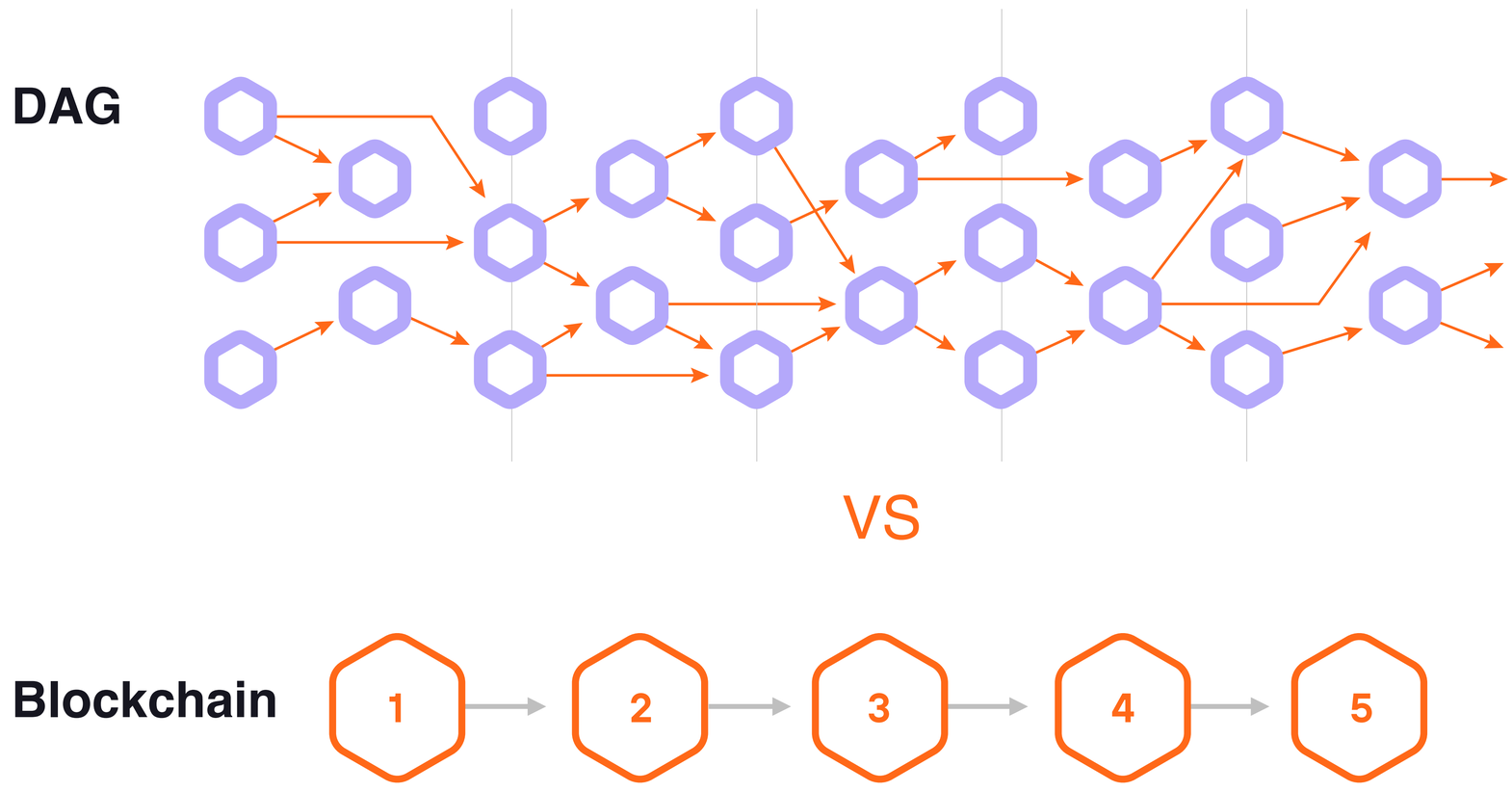


Figure 1.1

DAG and blockchain ledger structures

Core Benefits of VINE

Scalability

User growth does not create bottlenecks, but rather more nodes create greater scalability resulting in more TPS.

Asynchronous

Transactions are not queued or formed into blocks, a crucial factor for real-time focused applications used for banking, gaming, etc.

Flexibility

Microtransactions are handled much more effectively due to the lack of technical requirements affecting fees.

1.1. JSUE ’s Foundation

The JSUE project is built on the principles of a Directed Acyclic Graph (DAG) using modern solutions:

- Bulding a DLT in the form of a DAG as a chain of entities (sites) processed and included in the registry in parallel and independently by different network participants. This potentially allows for high throughput, i.e. solving the problem of DLT scaling;
- Own advanced algorithms for selecting vertices based on a random movement of particles taking into consideration the mathematical apparatus of Markov Chain Monte Carlo (MCMC). This reduces computational complexity (compared to the MCMC algorithm) and provides protection against building parallel chains of sites (solves the problem of distributed ledger security);
- Verification mechanisms designed to formally verify DAG entities (solves the security problem of a distributed ledger);
- Account models for storing the state of wallets and aggregate balances in a decentralized ledger, which speeds up fund transfer operations and simplifies the implementation of smart contracts;
- Commit transactions - special entities that are periodically (every 5 seconds) added to the parallel blockchain and are designed to synchronize local copies of the DAG on individual network nodes. Commit transactions are used to quickly load the current state of the ledger by newly connected nodes, as well as to coordinate account balances between individual network nodes.

1.2. Comparison of the DLT JSUE Project with Other Known Solutions

There are many different DLT technologies, such as blockchain, DAG, Hashgraph, Holochain, etc. A comparison of these technologies can be made based on the following criteria:

- **Scalability** is one of the key criteria when choosing a DLT technology. Blockchain can provide robust scalability, but it comes with performance and transaction speed issues. DAG provides higher scalability and transaction speed but may require more network members.
- **Transaction confirmation speed** is a criterion that determines how quickly transactions can be processed and confirmed. DAG can provide faster transaction processing than blockchain, which makes it more attractive for use in some areas.

- **Security** is a criterion that determines how secure users' data and funds are. Blockchain can provide high security, but centralization problems can arise. The DAG provides a more decentralized structure, making it more secure from attacks.
- **Energy efficiency** is a criterion that is becoming increasingly relevant because of growing environmental concerns. Proof-of-Work (PoW) blockchain consumes a significant amount of energy, while Proof-of-Stake (PoS) and DAG-based blockchains consume much less energy.
- **Transaction cost**: a DLT cost-effectiveness measure that determines how expensive it is for users to send their transactions. Fees for sending and processing transactions on Proof-of-Stake (PoS) blockchains and DAG-based DLTs can be very low.
- **Flexibility** is a criterion that determines how easy it is to make changes in technology and adapt to different needs and tasks.

A brief comparison of the various DLT technologies is shown in Table 1.1.

Table 1.1

Brief comparison of different DLT technologies

	DLT on blockchains		DLT on graph structures	
	PoW consensus	PoS consensus	DAG	Hashgraph / Holochain
Scalability	Very low	Low	Very high	Very high
Transaction confirmation speed	Very low	Low	Very high	Very high
Safety	Very high	High	Very high	Potentially high
Energy efficiency	Very low	High	Very high	Very high
Transaction cost	High	Low	Very low	Very low
Flexibility	Low	High	Very high	Very high

Thus, DLTs based on graph structures have clear advantages over classical chains based on the blockchain. For example, DAG technology has the following advantages:

- **High scalability**: DAG has no limit on the block size and the number of transactions that can be processed per unit of time.
- **Fast transactions**: In DAG, transactions can occur in parallel, which allows the processing of a large number of transactions per second.
- **High security and reliability**: Due to mutual confirmation of transactions and parallel processing, DAG is more secure and is not subject to long delays in transaction processing, which increases the reliability and stability of the system.

- **High energy efficiency**: In DAG, there is no need to solve complex mathematical problems, which require a lot of energy. Instead, DAGs use consensus algorithms that minimize the use of network resources and the amount of information that needs to be passed between participants.
- **Lower fees**: DAG transactions are cheaper than PoW blockchain transactions because they do not require mining. This allows users to send transactions with lower fees or no fees at all.
- **High flexibility**: DAG technology can be adapted to various use cases, capable of providing high scalability and high speed, which is useful for creating micro-transactions or solving other problems.

Table 1.2 provides a brief comparison of various DLT projects using DAG technology.

Table 1.2

Brief comparison of DLT projects using DAG technology

Technology	Release Date	Consensus	Transaction Speed (TPS)	Scalability	Security	Governance	Transaction Approval Time	Fee
IOTA	June 2016	Coordinator-based consensus	1,000 (with Coordinator)	Limited by Coordinator, it's slower without it	Cryptographically secured	Decentralized foundation	1-3 minutes	Zero
DAGCoin	July 2018	DAG-based consensus	8.000	Limited by hardware resources	Cryptographically secured	Centralized	30 seconds	0,0005 DAGCoin
ByteBall	December 2016	DAG-based consensus	100	Limited by hardware resources	Cryptographically secured	Centralized	Few minutes	1MB storage fee $0,033
Nano	November 2017	Open representative voting consensus	Up to 7,000	Limited by hardware resources	Cryptographically secured	Decentralized	Limited only by transaction transfer delays	Zero
XDag	December 2017	DAG-based consensus	200-300	Limited by hardware resources	Cryptographically secured	Decentralized	30 seconds	Min of 0,01 XDAG
Fantom	February 2018	Lachesis-based consensus	300,000 (Up to 10,000 in real tests)	Horizontally scalable	Cryptographically secured	Decentralized	Few seconds	Very low
JSUE	2023	VINE proprietary synchronization and confirmation algorithms complex	Higher than 700,000	Increased with every connected node (linear effect)	Post-quantum cryptographically secured	Decentralized	Sub-second limited by front-end	Very low or zero

* The data in this table is taken from the official pages of DLT projects

The conducted research shows that the main problem of modern DLT projects built on graphs is **ensuring consistency**.

Consistency is one of the key concepts in computer science that refers to the property of data and systems that ensures that the data is always in the correct state and is in a consistent state between the various nodes or components of the system. In the context of distributed systems and databases, consistency means that all copies of data in different nodes of the system must be in the same state at any given time. This means that any data changes made in one node of the system must be reflected in all other nodes in the system to ensure that the data does not contradict each other and does not contain errors.

In a DAG-based DLT, different participants can see different graph branches. This means that local versions of the DAG may differ between nodes, which can lead to consistency issues in transaction processing, errors in calculating wallet balances, and registry security issues. This explains the relatively low-speed characteristics of modern DAG registries. For example, with 300,000 TPS declared, real testing of DLT Fantom on TestNet of 7 nodes showed only 4000-10000 TPS. Different projects solve the problem of consistency in different and not always successful ways. For example, the IOTA project uses a centralized solution in the form of a special coordinator node, which violates the basic principle of registry decentralization.

An integrated approach has been implemented to solve the problem of consistency in DLT JSUE
, consisting of the use of new innovative solutions:

- Ledger storage model that takes into account the structure of the DAG and the height of each transaction for fast verification and calculation of balances.
- Ledger synchronization and consensus algorithms with 100% confirmed transactions included in the irrevocable part of the ledger (we call this a DAG slice).
- Algorithm for the formation of the so-called fixing transactions for fast synchronization of network nodes and acceleration of registry verification;
- Vertex selection algorithms in the DAG, which speed up the procedure for confirming previously generated transactions;
- Algorithms for processing smart contracts, taking into account the structure of the registry and processing fixing transactions.

These new high-tech solutions made it possible to solve the problem of fast synchronization of local DAG copies at different nodes, which, together with parallel processing of transactions in different branches of the graph, provide the unique characteristics of DLT JSUE
:

- Scalability is very high, limited only by the hardware resources of the connected nodes and the number of nodes.
- The speed of generating and processing transactions is more than 700,000 TPS.
- The speed of transaction confirmation is very high, limited only by the time delay when transferring transactions in a peer-to-peer network (usually, fractions of a second).
- Transaction cost is zero or very low (depending on the number of connected nodes and motivation model).

Separately, there's a possibility of including transactions in the irrevocable part of the registry. This is a new feature implemented in DLT JSUE
, which guarantees 100% confirmation of the transaction, i.e., the impossibility of canceling it under any scenarios in the behavior of
network nodes. As a rule, in modern DLT projects, the acceptance of transactions is probabilistic in nature, i.e., their confirmation can be challenged and canceled with some (usually very small) probability. At the same time, the longer the waiting time, the less likely it is for the confirmation of the transaction to be canceled. In DLT JSUE
, each transaction
enters the pool of irrevocable transactions after about 5 seconds, and the probability of confirmation cancellation is zero, i.e., confirmation of transactions is certain.

It should also be noted that in DLT JSUE
, cryptographic information protection is
implemented using fast and secure algorithms that are resistant to quantum cryptographic analysis. This guarantees historically reliable and secure storage of transactions even in the conditions of using new calculations based on physical principles and phenomena of quantum physics by intruders.

2. Distributed Ledger Structure

A distributed ledger is a set of replicated, shared, and synchronized arrays of digital data (entities) distributed among different users of the system.

DLT VINE is implemented using DAG technology by connecting individual entities (we call them sites) with edges. The direction of the edge means a link to the previous site.

2.1 Sites

Entities in DLT VINE are formed by network nodes in the form of sites.

Sites include transactions and are stored in a distributed ledger in the form of an acyclic-directed graph. Each new site links to the two previous ones, thus confirming all the sites to which it directly or indirectly links. Validation involves performing a check and establishing the correctness of the recorded information.

2.2 Commit Transaction

Architecturally, commit transactions are not included in the DAG structure, but are made in the form of a parallel linear structure of sequential entities, following the example of a classic blockchain. This is shown schematically in Figure 2.1.

Figure. 2.1

A simplified structure of the formation of a DAG and a parallel structure from a linear sequence of commit transactions.

Each commit transaction is formed as a result of:

- counting the share of confirmations of each site by the current DAG vertices. If the site has a 100% confirmation rate, it is marked as 100% confirmed (this is indicated by the corresponding color in the figure);
- synchronization of the generated list of 100% verified sites between the validating nodes.

Each commit transaction stores:

- up-to-date balance of accounts corresponding to the result of completing 100% of the share of verified sites;
- information about smart contracts, the result of their verification, and execution.

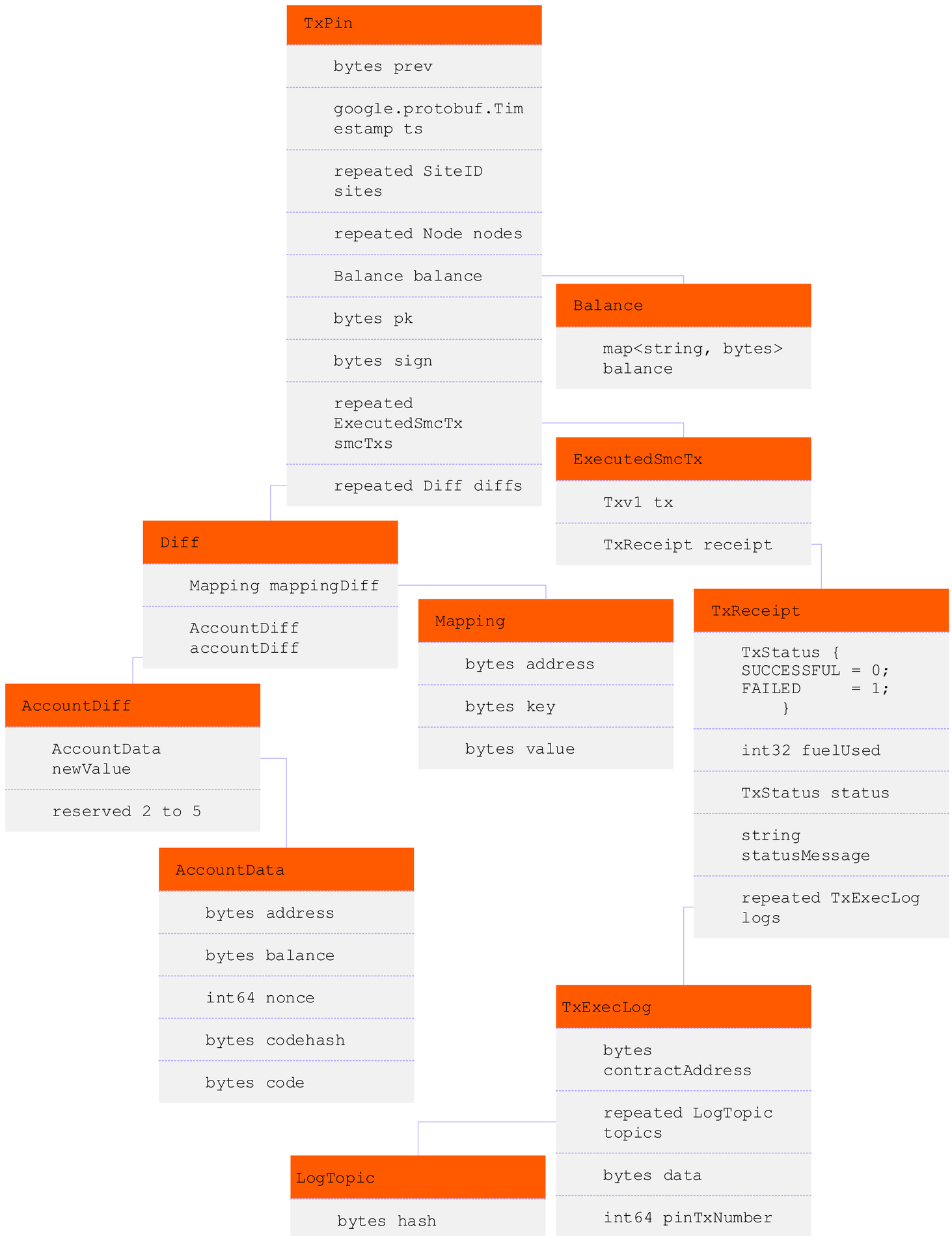
Thus, smart contracts in JSUE
only appear in commit transactions.

The format of commit transactions (`TxPin`) complies with the specification given in annex A.

Figure 2.2 shows the structure of a commit transaction in the notation of a UML diagram.

Figure. 2.2

UML diagram of the commit transaction structure



2.3 Smart Contracts

VINE is made using DAG technology. This means that individual entities are stored not in the form of a directed linear structure of a classical blockchain, but in the form of a tree-like, branching graph with cross-references and a complex non-linear structure.

To simplify the work with smart contracts, VINE uses only the linear part of the distributed ledger. In other words, the only “visible” part of the ledger for smart contracts is the information published in commit transactions (see Figure 2.1). This means that smart contracts can only operate on information (account balances, accounts, validators, etc.) that is directly specified in commit transactions. However, it also has its own characteristics:

- The balances of accounts in VINE between commit transactions are changed by entities “invisible” to smart contracts – transactions from DAG sites. This means that at the time of the next commit transaction, the states of the accounts will be provided (for a smart contract) "a priori", i.e. given "as is" with already changed states that are "invisible" entities for the smart contract;

- The “invisible” part of the registry for smart contracts is executed before the formation of a commit transaction. This means that the following is the final order of execution of smart contracts: the smart contract is executed in relation to the state of the system that exists at the time the initiating (publishing or calling) smart contract transaction is written to the commit transaction.

3. Sites and Transactions

The main entities of VINE are sites (`Node`) that include transactions as well as links to verified sites.

Site Format (`Node`) complies with the specification given in Appendix B.

Figure 3.1 shows the structure of the site in the notation of the UML diagram.

The stages of creating a site are shown in Figure 3.2.

.

Figure. 3.1

UML site structure diagram

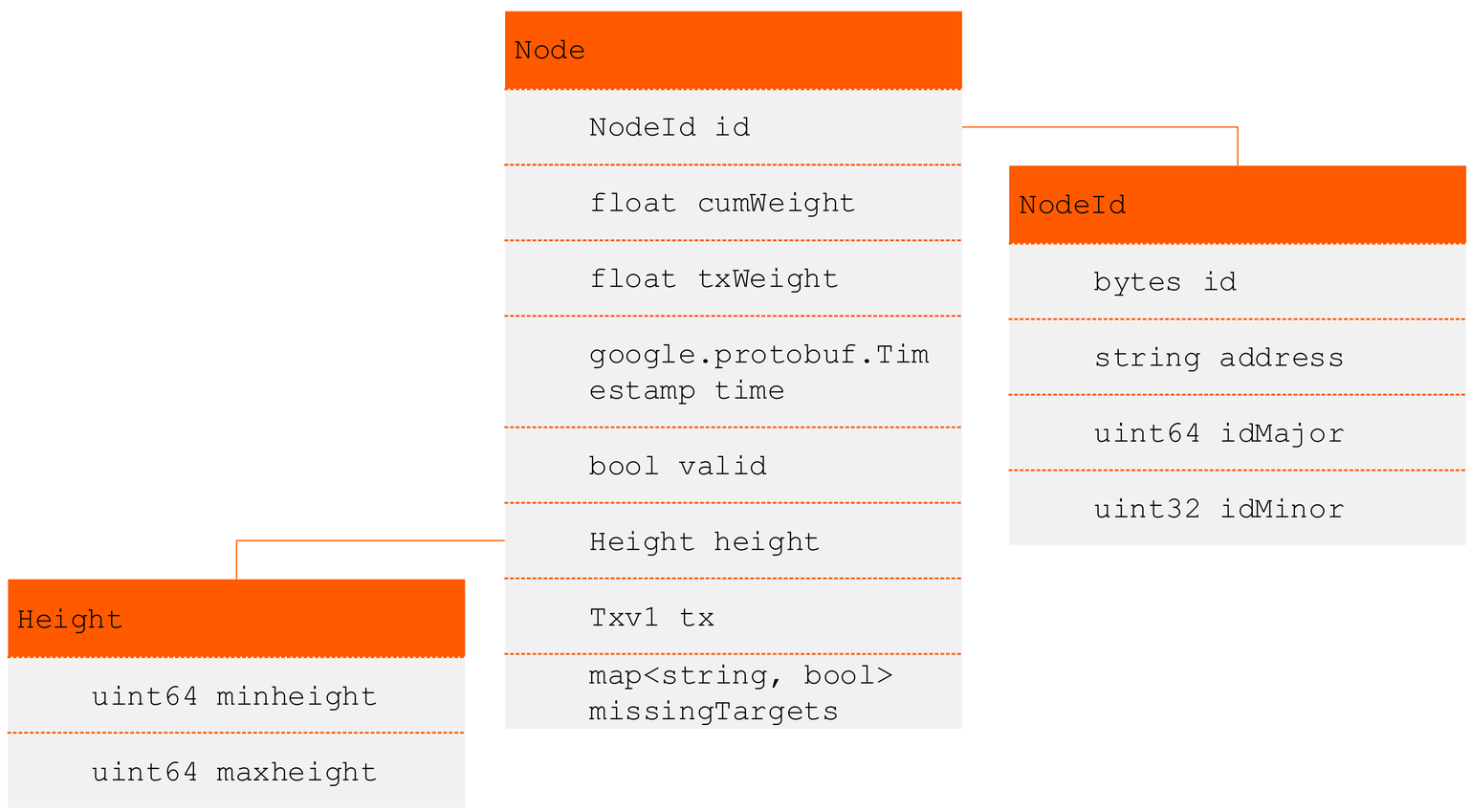
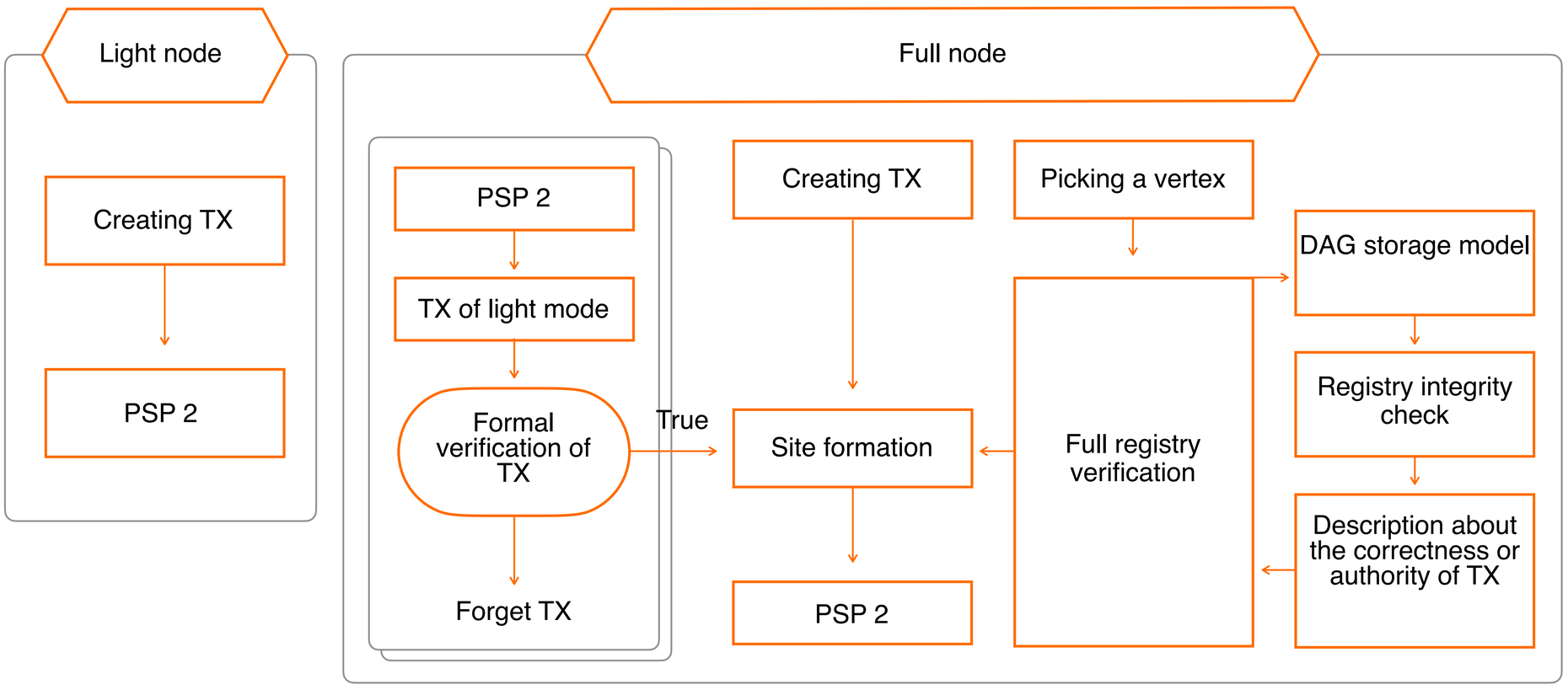


Figure 3.1

Stages of creating a site



3.1 Transaction Lifecycle

The transaction lifecycle includes:

- Creation - the client application creates a transaction, serializes, signs, and sends it to a peer-to-peer network node;
- Formal verification - the node that received the transaction performs its formal verification;
- Encapsulation - the node that received and verified the transaction encapsulates the verified transaction in the site. If the node generated the transaction, the first two steps are skipped;
- Site formation. The node, using the vertex selection algorithm, selects two previously unconfirmed sites. The identifiers of the selected vertex sites are specified in the generated site. The site is serialized and signed by the host;
- Distribution - a node sends a site to peers connected to it;
- Synchronization - all nodes that received the site perform their formal verification (checking the structure, version, signatures, etc.) and synchronize their own copies of VINE;
- Formation of a commit transaction.

3.2 Algorithms for Selecting Vertices for Sites

Each new site links (verifies) two other (previous) unconfirmed sites (vertices).

The choice of site (transactions) for confirmation is based on the vertex selection algorithm. VINE implements three main algorithms:

- by default - random selection algorithm (Algorithm 1);
- algorithm based on random walk MCMC+ with additional normalization of cumulative site weights (Algorithm 2).
- algorithm based on random walk MCMC++ with processing only own site weights (Algorithm 3).

The random selection algorithm is the fastest and most economical to use. Algorithms based on MCMC provide resistance to the imposition of third-party DAG branches and protection from "lazy" (non-vertex-confirming) sites.

3.2.1 Algorithm 1 (random selection of vertices)

Input: weight set of vertices (unverified sites);

Output: number of a selected vertex.

1) generate x - implementation of a uniformly distributed random variable on the interval [0,1];

2) normalize the weights. For this, we calculate for everyone:

Check the correctness of normalization:

3) Accept and select the range that belongs to x :

Return a number .

$\{w_1, w_2, ..., w_n\}$

$i \in \{1, 2, ..., n\}$

$i \in \{1, 2, ..., n\}$ $\overline{w_i} = \frac{w_i}{\sum_{j=1}^{n} w_j}$

$\sum_{i=1}^{n} w_i = 1$

$w_0 = 0$

$\sum_{j=0}^{i-1} w_j \le x < \sum_{j=0}^{i} w_j$

i

3.2.2 Description of MCMC, MCMC+, MCMC++ Algorithms

The MCMC (Markov Chain Monte Carlo) algorithm is described in detail in the article "The Tangle", section 4.1 "A parasite chain attack and a new tip selection algorithm."

The main idea of the MCMC algorithm is to put some "particles" (also known as random wanderers) at different nodes of the graph and let them move toward the vertices in a random way. The vertices "chosen" during the random walk become candidates for approval.

The MCMC random walk algorithm is described as follows.

1. Consider all nodes on the interval $[W;2W]$, Where W large enough[1].
2. Self-post N particles at nodes in this interval.
3. Let these particles perform independent random walks with discrete time "to the vertices", which means that the transition from the node x to node y is possible if and only if the node y confirms the node x.
4. The two random walks that reach the set of vertices first, determine the two vertices to be validated. However, it may be wise to modify this rule as follows: first, discard those random walks that reached the vertices too quickly because they could end up on one of the "lazy vertices."
5. The walk transition probabilities are defined as follows: if y confirms x, then the transition probability $P_{x,y}$ is proportional to:

Where:

- $\alpha > 0$ – adjustable parameter of the MCMC algorithm;
- cumulative node w_x, w_y, w_z weights x, y, z;
- denotation $z \to x$ means that the node z confirms node x.

$$P_{x,y} = \frac{e^{-\alpha(w_x - w_y)}}{\sum_{z:z\to x} e^{-\alpha(w_x - w_z)}} \tag{3.1}$$

The most genuine approach to modernize the MCMC algorithm is to normalize the weights, i.e. transform the formula (3.1) as follows:

Where:

- $\alpha > 0$ – adjustable parameter of the MCMC algorithm;
- $\overline{w_x} = 1, \overline{w_y}, \overline{w_z}$ – normalized aggregate (cumulative) node weights x, y, z:

$$P_{x,y} = \frac{e^{-\alpha(1-\overline{w_y})}}{\sum_{z:z\to x} e^{-\alpha(1-\overline{w_z})}} \tag{3.2}$$

- denotation $z \to x$ means that the node z confirms node x.

$$\overline{w_y} = \frac{w_y}{\sum_{z:z\to x} w_z} \tag{3.3}$$

[1] The indicated interval depends, first of all, on the order (rule) of numbering the nodes of the graph. In further studies, we directly set this interval by indicating the serial numbers of nodes corresponding to the order in which they are formed.

Thus, different variants of the MCMC algorithm differ only in the way the probabilities are calculated in Step 5:

- The MSMS algorithm (in the author's interpretation) uses formula (3.1), where w_x, w_y, w_z are the cumulative weights of the nodes x, y, z;
- The MCMC+ algorithm uses formula (3.2), where $\overline{w_x} = 1, \overline{w_y}, \overline{w_z}$ are the normalized aggregate (cumulative) node weights x, y, z;
- The MCMC++ algorithm uses formula (3.2), where $\overline{w_x} = 1, \overline{w_y}, \overline{w_z}$ are the normalized own weights of the nodes x, y, z.

Input data for the implementation of algorithms:

- $\alpha > 0$ – adjustable parameter of the algorithm. At $\alpha = 0$ the algorithm also works, but only takes into account the network topology. At $\alpha >> 0$ the algorithm gives preference to vertices that are located farther from the genesis and there are many paths to these vertices (higher throughput);
- W – depth of "throwing" of a random walk particle. The higher W there is a more reliable algorithm, potential protection from "parasitic" chains;
- set of weights w_x, w_y, w_z at each "transition" of the random walk.

3.2.3 Algorithm 2 (MCMC+)

Input:

- α – adjustable parameter of the algorithm;
- W – depth of "throwing" of a random walk particle. By default, we select the maximum depth, i.e. we consider the genesis block to be the throwing point;
- DAG as a set of transactions with assigned weights w_j, including a set of weights $\{w_1, w_2, ..., w_n\}$ vertices (unverified sites).

Output: number $i \in \{1, 2, ..., n\}$ selected vertex.

We implement a random walk through a chain of transitions $x \to y_j$ “from the site x to the site y_i ". Let's assume that the random walk particle is located in the site x and this site confirms k sites $\{y_1, y_2, ..., y_k\}$ with own weights $\{w_{y_1}, w_{y_2}, ..., w_{y_k}\}$.

Then **transition algorithm** $x \to y_j$ consists of steps:

1. Accept $\overline{w_x} = 1$. We normalize the weights $\{w_{y_1}, w_{y_2}, ..., w_{y_k}\}$. For this, we calculate $j \in \{1, 2, ..., k\}$ to all, according to the formula (3.3):

Checking the correctness of normalization:

$$\overline{w_{y_j}} = \frac{w_{y_j}}{\sum_{\ell=1}^{k} w_{y_\ell}}$$

$$\sum_{j=1}^{k} \overline{w_{y_j}} = 1$$

2. Calculate the transition probabilities $x \to y_j$. For this, we expect for everyone $j \in \{1, 2, ..., k\}$ the following:

$$P_{x,y_j} = \frac{e^{-\alpha(1-\overline{w_{y_j}})}}{\sum_{\ell=1}^{k} e^{-\alpha(1-\overline{w_{y_\ell}})}}$$

3. Select a vertex for the transition. To do this, we use Algorithm 1 with the input $\{P_{x,y_1}, P_{x,y_2}, ..., P_{x,y_k}\}$ (the weight normalization step can be omitted, the weights are already normalized). The output of Algorithm 1 gives the number $j \in \{1, 2, ..., k\}$ of the selected site from $\{y_1, y_2, ..., y_k\}$ to go to $x \to y_j$.

Then we repeat the transition algorithm until one of the vertices of the graph with weights is selected $\{w_1, w_2, ..., w_n\}$. We return the number $i \in \{1, 2, ..., n\}$ of selected vertex.

3.2.4 Algorithm 3 (MCMC++)

The main difference in the use of MCMC+ is the rapid growth of cumulative weights. For example, for a throwing depth of 100, the values of the cumulative weights can be on the order of 2_{100} which is very difficult to store and process. In addition, with a large discrepancy in the cumulative weights, the probabilities can also differ significantly. For example, $\{w_{y_1} = 2^{100}, w_{y_2} = 2^{80}, w_{y_k} = 2^{50}\}$ after normalization will be $\{w_{y_1} \approx 1, w_{y_2} \approx 0, w_{y_3} \approx 0\}$, i.e. we have a practically uncontested transition of a random walk.

The MCMC++ algorithm uses only its own weights (determined by node ratings, commissions, etc.).

3.3 Formal Verification

The formal verification is performed by a full node upon receipt of each new transaction. Verification is performed for all transactions. During the synchronization process, each node receives sites from a trusted node (the leader node acting as a validator) and includes them in its copy of VINE without verification, since it does not have all the necessary information to conduct a formal verification on its own. If at least one of the formal requirements is not met, the check is recognized as not passed, the site is not included in the registry. Full verification of the registry is not carried out in this case.

3.4 Verification

A simplified registry verification process is shown in Figure 3.2.

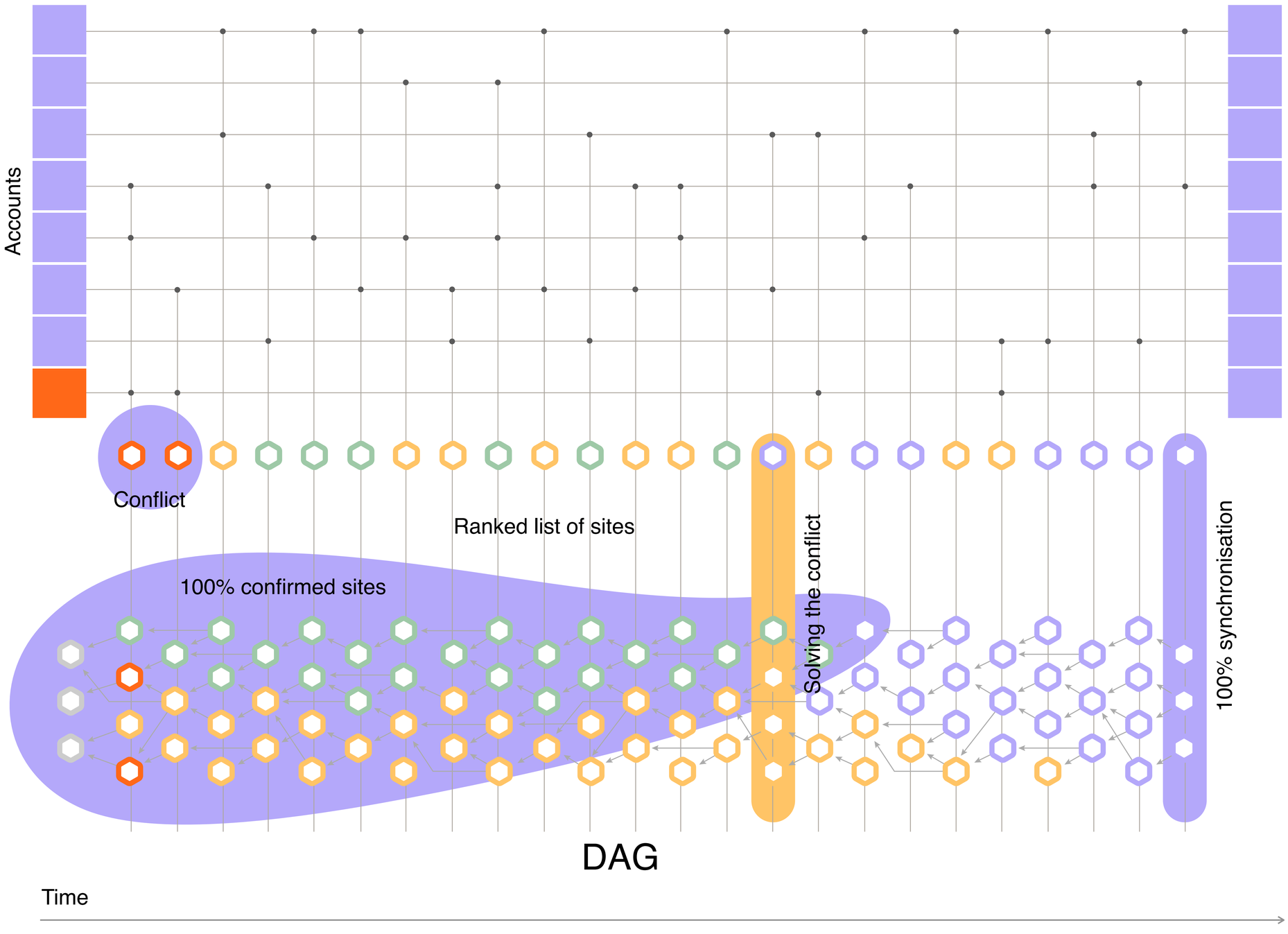


Figure 3.2
Simplified registry verification

The registry verification includes:

- representation of the registry as an ordered array of all sites. Changes in account balances are directly related to site transactions, i.e. these changes are also ranked;
- making a decision on the correctness of transactions included in the checked part of the registry. The check is performed according to the criterion: "The value of the balance of the wallet must be above zero.”

4. Processing Smart Contracts

The following stages of the life cycle are applicable to each smart contract:

- *Creation (construction) of a smart contract*. At this stage, the logic of the contract actions is programmed (in the form of program code);
- *Verification (audit) of a smart contract*. It is performed by the smart contract developer, as a rule, before the smart contract is published in the distributed ledger. This stage is designed to check the logic of the program code and the correctness of the possible results of its completion with different initial data. Verification (audit) of a smart contract is not recorded in the distributed ledger;
- *Publication (initialization, deployment) of a new smart contract*. The transaction is formed and published to the network by any user of the system. At this stage, the smart contract account is created, the internal state is initiated, etc.;

- *adding a smart contract* to a special Diff object for the commit transaction, where the new account for the published contract is stored;
- *smart contract call*. The transaction is formed and published to the network by any user of the system. The same smart contract can be called multiple times;
- *adding the results of a smart contract call* to the internal storage (MappingDiff object) of the Diff object for the committing transaction. The results of the call are reflected as a change in the smart contract storage;
- *validation* of initialization correctness / execution of the smart contract. In the current implementation, verification, and publication of the results of the execution of a smart contract is performed by one node - the leader, whose actions imitate the work of validators;
- *validation* of initialization / execution of the smart contract. Initialization/execution of a smart contract is considered *correct* if no conflicts are found in the system as a result of the check.

Thus, actions with smart contracts are recorded in the distributed ledger in two cases:

- publication of a new smart contract and the results of this publication (with a mark of correctness or incorrectness);
- calling a smart contract and the results of this call (with a mark of correctness or incorrectness).

Publishing and calling a smart contract are made in the form of corresponding transactions, and further, when referring to a smart contract transaction, we will mean any of these options unless otherwise specified.

4.1 Stages of Processing of Smart Contracts

The general process for processing smart contracts and generating a commit transaction is presented in the UML notation of the sequence diagram in Appendix B.

Smart contract processing includes:

- creation of a smart contract;
- formation of a pool of unconfirmed transactions of smart contracts;
- adding of smart contracts to the commit transaction;
- execution of smart contracts and updating balances

4.1.1 Creating a Smart Contract

The user (light client or full node), having created a smart contract, draws it up in the form of a special transaction. This transaction propagates across the network.

When a transaction with a smart contract is distributed over the network, it is formally verified. If the formal verification was successful, then the transaction with the smart contract is broadcast by the node further. If the formal verification fails, the transaction with the smart contract is forgotten.

4.1.2 Pool with Unconfirmed Smart Contract Transactions

Pool of unconfirmed smart contracts - a pool with a set of transactions with smart contracts that have successfully passed formal verification but have not yet been added to the commit transaction.

After receiving a transaction with a smart contract:

- a check is performed to see if this transaction was previously added to the pool of unconfirmed smart contracts. If the transaction has not yet been added, then
- it is formally tested. If the formal check was successful, then
- a transaction with a smart contract is added to the local pool of unconfirmed smart contracts;
- a transaction with a smart contract is broadcast further over the network.

4.1.3 Verifying and Adding a Smart Contract to a Commit Transaction

Verification of smart contracts is performed at the stage of formation of a commit transaction. The transactions themselves with smart contracts, the result of their verification and execution, are added to a special section of the fixing transaction.

4.1.4 Execute Smart Contracts and Update Balances

The result of the execution of smart contracts is a change in account balances. Balances are updated within a commit transaction. The result of the update, in the form of a new account balance, is stored in the commit transaction and used as input for further transactions in VINE.

4.2 Storing Information About Smart Contracts in a Commit Transaction

All information regarding smart contracts is an integral part of the commit transaction. There are two attributes for this:

- ExecutedSmcTx – information about executed smart contracts;
- Diff - differences in the state store that appeared during the execution of the smart contract.

A detailed specification of these attributes is given in Appendix A, and is also schematically presented in Figure 2.2.

5. Site Verification and Registry Synchronization

Each node in the local copy of the registry recalculates the proportion of vertices that directly or indirectly confirm a site that has not yet been added to the commit transaction. If this share reaches 100%, the site is added to the pool of 100% verified sites.

For all transactions included in the pool of 100% verified sites, based on the decisions made by the full nodes, we determine their correctness.

5.1 Algorithm for Calculating the Share of Vertices that Directly or Indirectly Confirm the Site

Input:

- own copy of the registry with a list of current vertices $TipsList$;
- new vertex $NewTip$, added to the DAG;
- W – particle throwing depth (adjustable parameter of the MCMC
- algorithm). In this algorithm$_W$ measured from genesis, i.e., for example:
- at $W = 0$ particles are thrown into the genesis site;
- at $W = 1000$ particles are thrown into the 1000th site after genesis.

Output:

- own copy of the registry indicating the share of vertices δ_i (each i-th site), directly or indirectly confirming the site;
- a pool of 100% confirmed vertices ready to be sliced.

Algorithm:

1. For each i-th site in the DAG, create a list $TipsList_i$ indexes (numbers) of vertices that directly or indirectly confirmed the site. Each index j from $TipsList_i$ specifies the vertex number from $TipsList$. The ratio of the number of elements $TipsList_i$ to the number of elements $TipsList$ sets the desired parameter – the proportion of vertices that directly or indirectly confirmed i-th site:

List initialization $TipsList$, $TipsList_i$ and relevant δ_i:

- $TipsList = \{1, 2, ..., k\}$ – list of vertices referring to genesis;
- $TipsList_0 = TipsList$ – list of indexes for the genesis site ($i = 0$);
- $TipsList_1 = TipsList_2 = ... = TipsList_k = \varnothing$ – empty set (for all current vertices);
- $\delta_0 = 100\%$ (for the genesis site);
- $\delta_1 = \delta_2 = ... = \delta_k = 0\%$ (for the first k vertex sites referring to genesis).

$$\delta_i = \frac{|TipsList_i|}{|TipsList|} \cdot 100\%$$

All new items from the list $countList_k$ are initialized to zero.

2. When adding each new vertex $NewTip$ list of vertices $TipsList$ updated:
   - some vertices $OldTip$ "closed" by links from $NewTip$, after which they are excluded from the list $TipsList$;
   - new vertex $NewTip$ added to the list $TipsList$.

3. If $W \leq i$ (particles in MCMC are thrown up to i-th site in DAG) or ($W > i$ and $\delta_i < 100\%$):

- update $TipsList$ after every update $TipsList$ update for all sites $TipsList_i$. To do this, we consider only indexes that refer to $OldTip$ These indexes are excluded from$TipsList_i$, replacing them with indexes referring to $NewTip$;
- update δ_i.

4. If $W > i$ (particles in MCMC are thrown after i-th site in DAG) and $\delta_i = 100\%$, Then $TipsList_i$ don't update anymore i-th site is included in the pool of 100% confirmed vertices ready to be sliced.

The algorithm is executed when adding each new vertex, Step 3 (the most costly) is performed only for $W \leq i$ or ($W > i$ and $\delta_i < 100\%$).

5.2 Algorithm for Accepting a Commit Transaction

A commit transaction is generated at least once every 5 seconds. With a significant decrease in the intensity of the receipt of transactions, up to their complete absence, the frequency of formation of commit transactions should not change.

The algorithm consists of five-time phases, as shown in Figure 5.1.

.

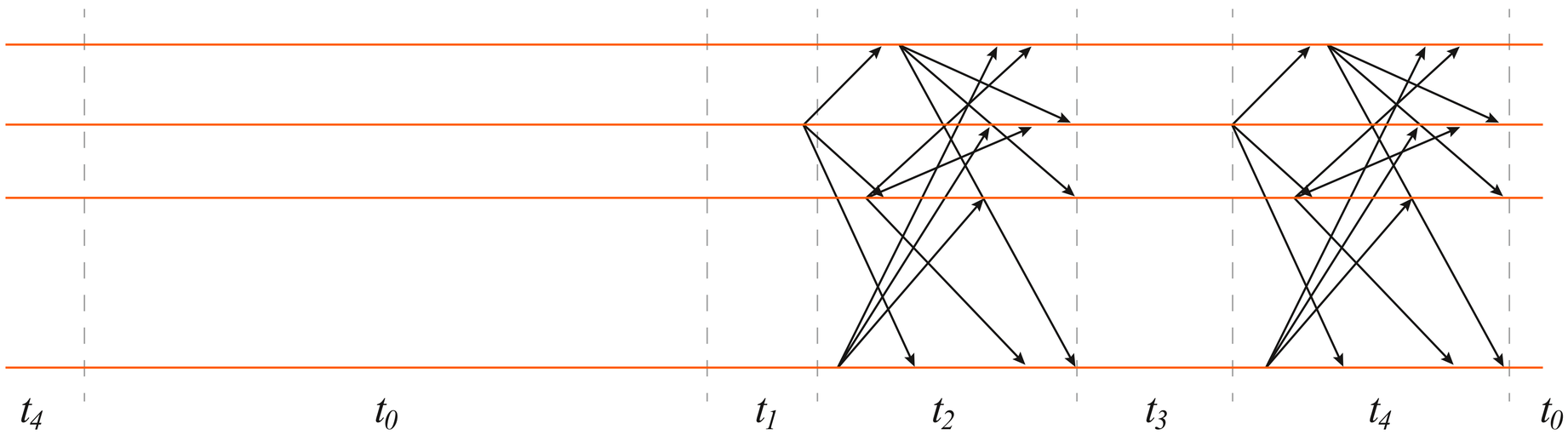


Figure 5.1

Commit phases of a commit transaction

- t_0 – formation of a "blank" of a commit transaction.
- t_1 – formation of a local pool of 100% confirmed sites.
- t_2 – exchange of information about commit sites.
- t_3 – verification of smart contracts.
- t_4 – exchange of prototypes of a commit transaction.

5.3 Synchronization

Synchronization is the process of updating information about the contents of the registry and writing the updated information to the local copy of VINE.

For different tasks (in order to reduce the time of updating and reduce the volume of stored data) synchronization of commit transactions can be performed – updating information only about a chain of commit transactions.

5.3.1 Synchronization of Commit Transactions

Synchronization starts with the last known CT (commit transaction). After adding the last CT to the local copy, the node can proceed in parallel to the synchronization and subsequent work in the normal mode, without waiting for the end of the CT synchronization process.

Commit transactions refer to each other, all queries and storage are in the same order. The update is subject to a formal review.

Synchronization occurs starting from the last received CT in the direction of earlier CTs (as needed).

The necessary synchronization ends when the required number of links to sites is reached, which is necessary to enable the node to work.

If a node connects to the system for the first time and does not have a local copy of VINE, then the last CT included in the local copy is considered to be the genesis of the CT.

When connecting to the network, each node notifies the network of its latest CT version and receives a copy of the current CT from the leader node acting as a validator. The last CT is the current starting position for the new node. All other artifacts are requested from the validator as needed.

5.3.2 Site Synchronization

The node synchronizes sites that are not yet included in the commit transaction, i.e. those that have been generated since the last commit transaction and up to the current time if needed (e.g. the new site refers to sites not yet included in any commit transaction).

5.3.3 Information Exchange (normal synchronization mode)

One of the modifications of the Gossip algorithm, the HyParView (Hybrid Partial View) protocol, is taken as the basis for distributing information between peers.

By constant exchange of information, we mean the exchange of all new objects (sites, smart contract transactions and commit transactions).

To speed up the check and fight against spam objects, we can use a local database of rejected objects. In this database, we will record the identifiers of objects that have not passed formal verification. This base has a fixed size and when it is filled with new objects, older objects are removed from the base.

Information Synchronization Algorithm:

1. According to the HyParView protocol, we get an object.

   Possible events; probabilities; actions:

- Object received; the probability is high; go to the next step of the algorithm (normal mode);
- Object not received; the probability is extremely low; reinstalling the software, communicating with the community, looking for a problem on our server;

2. Check if the given object is in the local database of rejected objects. If the object is in the database, then exit the algorithm.
3. Check if the given object exists in the local copy of VINE. If it is a network, then exit the algorithm.
4. We carry out a formal check (for the site we do not check the condition whether the verified sites exist in the registry). If the formal check fails, then we write the object identifier to the local database of rejected objects and exit the algorithm.

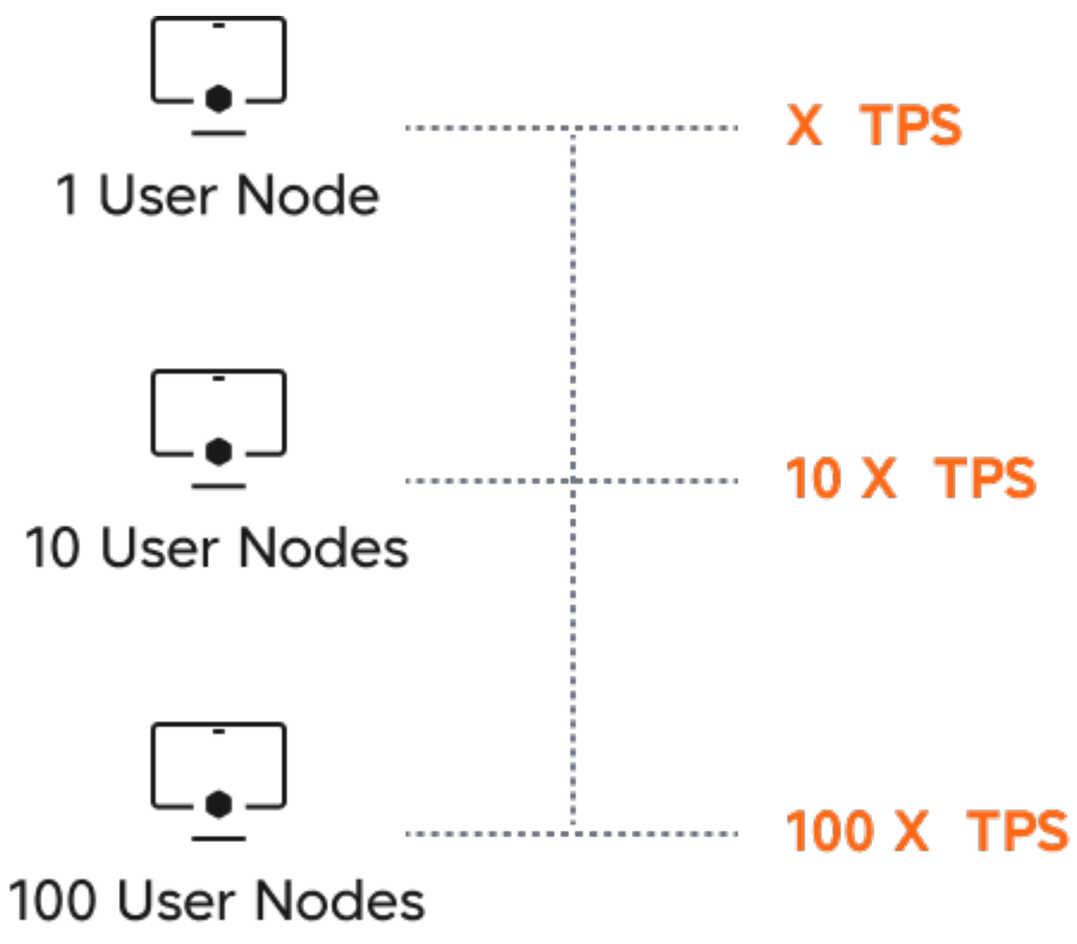
   4.1 Check whether verified sites exist in the local copy of VINE. If at least one confirmed site does not exist, then we add the site to the database of orphan sites and exit the algorithm.

   4.2 Check whether orphan sites are linking to this site. If they are, then depending on the information about the remaining confirmed sites of the referring site, we make an appropriate note about it about the reduction of "orphan" links or perform step 4.2 and then step 5.
5. We add the object to the local copy of VINE.
6. We leave the algorithm.

6. Scalability

Testing the DLT JSUE
 functionality, which, with an increase in the number of generator nodes, leads to a multiple increase in the number of generated transactions per unit of
time, each validator node connected to the network becomes a project scaling step.

Thus, when adding additional nodes to the network, we get a multiple increase in TPS (transactions per second) due to an increase in the number of transactions processed and included in VINE on the node.



In the future, in order to process all generated transactions by individual nodes, synchronize their local registries, verify and validate, it is necessary to perform additional computational operations, the volume of which increases rapidly over time (as the size of the registry increases). To stabilize the high-performance of the network with a large number of generator nodes and transactions generated by them, DLT JSUE provides for the formation of slices - an irrevocable part of the registry, consisting of 100% verified sites. All sites and slices' transactions are excluded from further verification as unconditionally valid, which allows for stabilizing the computing load and achieving high-speed characteristics as the registry grows.

Slicing and mechanisms for consensus acceptance of commit transactions by validators are not implemented in JSUE DLT, this is one of the most relevant areas for further development of the project. Other areas for the development of JSUE DLT are working on optimizing site processing, including block processing of transactions, the introduction of fast post-quantum cryptography algorithms, registry sharding, etc. These activities allow for repeated increases in the performance of the network and successful reaching of the target of 700,000 TPS which with more nodes will increase indefinitely.

Appendix A.

Commit Transaction Format

The commit transaction format is specified in vine-dev/p2p/tx/pb/txpin.proto and looks like this:

```
// TxPin - pinning transaction definition
message TxPin {
    bytes prev                  = 1; // prev pin tx id
    google.protobuf.Timestamp ts  = 2; // timestamp when the tx was created
    repeated SiteID sites       = 3; // a collection of site ids (a list of fully confirmed sites in the current slice)
    repeated Node nodes           = 4; // sites included in this pin tx
    Balance balance             = 5; // all balances for all the tx in the current pinning tx
    bytes pk                    = 6; // public key to verify this pin tx signature
    bytes sign                  = 7; // this pin tx signature
    repeated ExecutedSmcTx smcTxs = 8; // executed smart contract transactions within this pin tx
    repeated Diff       diffs  = 9; // diffs on smart contract state store appeared after executing smart contract transactions
}

message Balance {
    map<string, bytes> balance = 1; // a map of wallet - balance (big.Int as []byte)
}

message LogTopic {
    bytes hash = 1;
}
message TxExecLog {
    bytes contractAddress = 1;
    repeated LogTopic topics = 2;
    bytes data = 3;
    int64 pinTxNumber = 4;
}
```

```
message TxReceipt {
    enum TxStatus {
    SUCCESSFUL = 0;
    FAILED    = 1;
    }
    int32 fuelUsed     = 1; // used fuel for execution
    TxStatus status    = 2; // tx failed or successful
    string statusMessage = 3; // details about tx status
    repeated TxExecLog logs    = 4; // logs generated during tx execution
}

message AccountData {
    bytes address   = 1;
bytes balance   = 3;
    int64 nonce    = 4;
bytes codehash  = 5;
    bytes code     = 6;
}

message AccountDiff {
    AccountData newValue = 1;
    reserved 2 to 5;
}

message Mapping {
    bytes address = 1;
    bytes key  = 2;
    bytes value   = 3;
}

message Diff {
    oneof mappingOrAccount {
    Mapping mappingDiff = 1;
    AccountDiff accountDiff = 2;
    }
}

message ExecutedSmcTx {
    Txv1 tx = 1;
    TxReceipt receipt = 3;
}
```

Appendix B.

Site Format

The site format is specified in vine-dev/p2p/tx/pb/node.proto and looks like this:

```
message Node {
    message NodeId {
    bytes id = 1;
    string address = 2;
    uint64 idMajor = 3;
    uint32 idMinor = 4;
    }
    message Height {
    uint64 minheight = 1;
    uint64 maxheight = 2;
    }
    NodeId id = 1;
  float cumWeight = 2;
  float txWeight = 3;
  google.protobuf.Timestamp time = 4;
  bool valid = 5;
  Height height = 6;
  Txv1 tx = 7;
     map<string, bool> missingTargets = 8;
}
```

Figure B.1

UML diagram of sequences for processing smart contracts and forming a commit transaction

Processing Smart Contracts and Generating a Commit Transaction