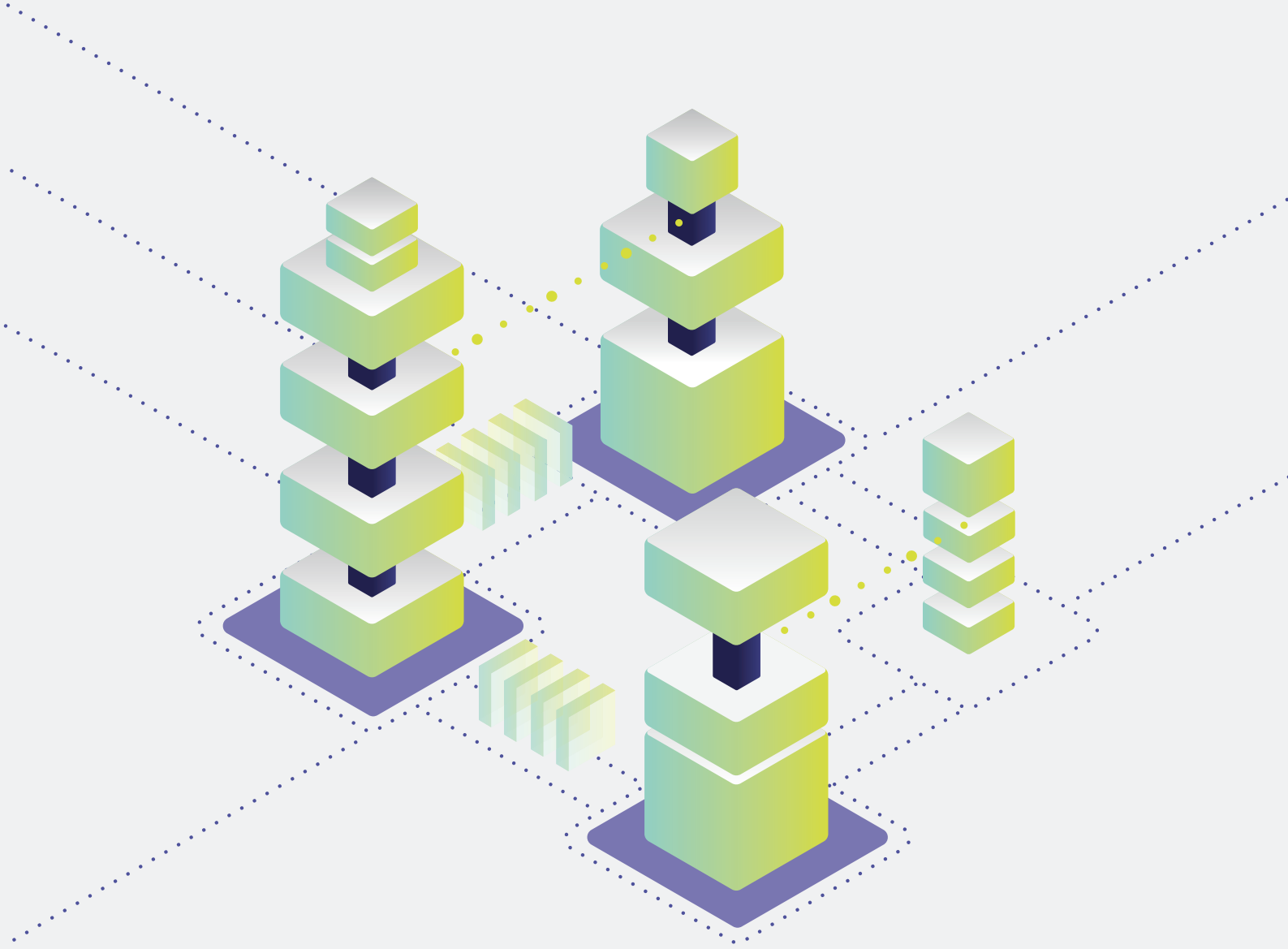
Fincor



Fincor Blockchain Research Paper

BLOCKCHAIN BUILT FOR THE FINANCE INDUSTRY.
THE WAY IT'S MEANT TO BE.

Introduction

The first major blockchains, including Bitcoin and Ethereum, elegantly proved that almost any business or industry can benefit from the blockchain technology by decentralizing and automating various business and even government-level processes. By recognizing blockchain's value beyond the initial application, such as a means of money transfer (cryptocurrencies), store of value or asset ownership and funding (ICOs/STOs), blockchain developers face growing demand and pressure to find more efficient solutions addressing the high costs, limited scalability and severe security concerns. Not to mention the regulatory requirements for the financial industry.

These issues and a thorough analysis of existing blockchain protocols led us to the decision to develop and launch our own native blockchain with Fincor Consensus Protocol (FCP).

The way existing blockchains are currently developed – the decision making, update planning and implementation processes are often politicized and lack transparency - also influenced our decision.

Our native consensus protocol – FCP will be based on the Federated Byzantine Agreement (FBA) and supplemented with sharding technology. A more detailed description and analysis can be found further in the text below.

The key advantages of the FCP+FBA+Sharding solution are:

- it liberates the blockchain from speed caps thus allowing us to significantly increase transaction speeds – up to approx. 10,000 transactions per second after implementation of the first protocol. After enabling the dynamic sharding technology, the speed will increase even further and will exceed speeds of the mainstream global payment systems;
- it allows us to implement near costless transactions within our ecosystem and provide much more favorable conditions to third-party developers of decentralized applications (dApps) and the Internet of Things (IoT) applications;
- it will provide optimal security for the blockchain network by protecting it from fraudulent participants thus minimizing the risk of cyber-attacks which are common for Proof-of-Work (PoW) blockchain platforms;
- finally, in addition to all aforementioned advantages our blockchain users will no longer need to be developers in order to enjoy the full benefits of the blockchain. Anyone will be able to create and execute smart contracts via native user interfaces and apps, provided by Fincor or third-party developers. The simplicity of our smart contracts will truly open the blockchain to all types of businesses – from mom-and-pop stores to global corporations and thus accelerate blockchain adoption worldwide.

A superstructure is built on a solid foundation, so we intend to build the entire Fincor ecosystem on top of Fincor blockchain. This will create additional speed and security for our smart financial products.

Market review

Despite the fact that the market is dominated by just a few blockchain platforms, each of these platforms suffers its own vulnerabilities, particularly in terms of speed and security. The blockchain speed is the bandwidth of the system, which is still limited by up to 5000 transactions per second. Security is paramount in running a crypto business and therefore it must be set at the highest level possible. However, a series of huge hacks of crypto exchanges and scammers' revelations have made users doubt the reliability of crypto technologies, which in turn led to panic in the market. While for the vast majority such a crisis looks like the collapse of the industry, for Fincor team it offers a huge opportunity and motivation to build superior technology and increase trust in the blockchain industry.

The current financial infrastructure is too insular, bound by archaic payment and IT systems, sewn together like a patchwork quilt. This affects the speed of movement of funds and makes clients' transactions costly. Fincors' solution to these issues is to build, from the ground up, a new financial ecosystem, based on cutting edge technologies. Our platform will be open to all and everyone, thus expanding our customer service network around the world and helping the unbanked to get financial services. Fincor blockchain will become a decentralized platform where users can independently conduct and confirm the validity of transactions under a mutual agreement based on the FBA agreement. This, in turn, will ensure the integrity of the financial ecosystem, and the decentralized management of the platform will stimulate its growth.

General information

This document presents a model of a Federated Byzantine Agreement (FBA) with sharding technology. The essence of FBA: a member of the network knows all other members. Each member considers their opinion as important. All transactions are regulated and validated based on the members' agreement or disagreement on each transaction. By agreeing on what updates to apply, nodes avoid contradictory, as well as, irreconcilable states. Network sharding is able to increase the blockchain's bandwidth – up to 10,000 transactions per second.

Sharding

The essence of sharding is to divide the transaction into separate particles (shards) and send them to different nodes for validation. Such flow is able to reduce the transaction's processing time within the network using parallel execution.

In addition, implementation of shardign technology could solve the problem of network scalability by such way: as each node-validator has copies of other nodes (in order to process shards), thus increasing the speed of network, as well as, increasing the level of network resilience. Sharding is suitable for almost any network, not only blockchain, except those solutions that are using the PoW protocol. Due to the vulnerability to 51% attack of the aforementioned protocol and other malicious manipulations by attackers who can determine which part of the network will process the shard.

Distribution of transactions between shards

Any transaction, e.g., from $A \xrightarrow{n} B$, will be processed by one shard. Assuming there are shards numbered from 0 to l-1, the transaction will be assigned to the shard identified by $[\log_2 l]+1$ extreme right bits of the sender's address (in this case, account A). Since the account address is a 160-bit numerical value (integer), then l will be limited as: $[\log_2 l]+1 \leq 160$. Based on this, we derive normalized A_{normal} for each address:

$$A_{normal} = LSB_{160}(SHA3\text{-}256(\textbf{PubKey}(sk)))$$

After the designated shard has been identified, the transaction will be broadcast to the network. As soon as the transaction reaches the validators of the selected shards, they will include it into the common network (blockchain).

FBA

In the Federated Byzantine Agreement model all members consider each other trustworthy to follow the same set of rules and reach consensus on the transaction sequence in the ledger (or *block*).

By coordinating which updates to apply, the nodes will be able to avoid conflicting, irreconcilable conditions. The system will identify each update by a unique interval from which it will be possible to derive dependencies between updates. For example, slots can be consecutively numbered positions in a sequential ledger. The FBA system uses a consensus protocol, ensuring nodes will negotiate contents of the slots. A node *v* can safely apply update *x* in a slot *i* when it safely applied updates in all slots on which *i* depends on. In addition, it assumes that all correctly working nodes will eventually agree on the valid value of *x* for slot *i*. At the moment, we state that *v* externalized *x* for slot *i*.

FBA solves the risk of attackers joining the system multiple times and, therefore, exceeding the number of "correct" nodes. FBA defines quorums in a decentralized manner with each node choosing quorums.

Quorum Slice

In a consensus protocol, nodes exchange messages asserting statements about slots. We assume such assertions cannot be forged, which can be guaranteed if nodes are named by public key and they digitally sign messages. When a node hears a sufficient set of nodes assert a statement, it assumes no functioning node will ever contradict that statement. We call such a sufficient set a *quorum slice*, or, more concisely, just a *slice*. To permit progress in the face of node failures, a node may have multiple slices, any one of which is sufficient to convince it of a statement. At a high level, then, an FBA system consists of a loose confederation of nodes each of which has chosen one or more slices.

More formally: Definition (FBAS). A federated Byzantine agreement system, or FBAS, is a pair 2 (V, Q) comprising a set of nodes V and a quorum function $Q: V \to 2^{2^V} \setminus \{\varnothing\}$ specifying one or more quorum slices for each node, where a node belongs to all of its own quorum slices—i.e., 6*u* c V, 6*q* c Q(*u*), *u* c *q*. (Note 2^X denotes the powerset of *X*.)

DEFINITION (QUORUM). A set of nodes U ≤ V in FBAS V, Q is a *quorum* if *U* Ç Ø and *U* contains a slice for each member—i.e., 6*u* c *U* , E*q* c Q(*u*) such that q ≤ U .

A quorum is a set of nodes sufficient to reach agreement. A quorum slice is the subset of a quorum convincing one particular node of agreement. A quorum slice may be smaller than a quorum. Consider the four-node system where each node has a single slice and arrows point to the other members of that slice. Node *u*1's slice {*u*1, *u*2, *u*3} is sufficient to convince *u*1 of a statement. But *u*2's and *u*3's slices include *u*4, meaning neither *u*2 nor *u*3 can assert a statement without *u*4's agreement. Hence, no agreement is possible without *u*4's participation, and the only quorum including *u*1 is the set of all nodes {*u*1, *u*2, *u*3, *u*4}.

Traditional, non-federated Byzantine agreement requires all nodes to accept the same slices, meaning 6*u*1, *u*2, Q(*u*1) = Q(*u*2). Because every member accepts every slice, traditional systems do not distinguish between slices and quorums.

Let's consider a system with four nodes, where each node has one slice and interacts with other elements of this slice. The slice of the $\{v_1, v_2, v_3\}$ node is enough to convince v_1 of validation. But slices v_2 and v_3 include v_4, which means that neither v_2 nor v_3 can confirm validation without consent v_4. Therefore, no agreement is possible without the participation of v_4, and the only quorum including v_1 is the set of all nodes $\{v_1, v_2, v_3, v_4\}$:

$$Q(v_1) = \{\{v_1, v_2, v_3\}\}$$
$$Q(v_2) = Q(v_3) = Q(v_4) = \{\{v_2, v_3, v_4\}\}$$

Optimal stability

Nodes ensure network security and viability based on several factors:

- which quorum slices they chose to interact with;
- which nodes do not behave safely;
- consensus protocol and network behavior.

Usually, in the case of asynchronous systems, we assume that the network ultimately delivers messages between nodes behaving securely, but otherwise, it can arbitrarily delay or reorder messages.

This is an option where there is no quorum intersection in FBAS:

$$Q(v_1) = Q(v_2) = Q(v_3) = \{\{v_1, v_2, v_3\}\} \neq$$
$$Q(v_4) = Q(v_5) = Q(v_6) = \{\{v_4, v_5, v_6\}\}$$

Incorrect behavior of node v_7 can undermine the quorum intersection:
$Q(v_1) = Q(v_2) = Q(v_3) = \left\{\{v_1, v_2, v_3, v_7\}\right\} \notin$
$Q(v_4) = Q(v_5) = Q(v_6) = \left\{\{v_4, v_5, v_6, v_7\}\right\}$,
if$Q(v_1) = \left\{\{v_7\}\right\}$.

Below, we will take a look at the best conditions for achieving security and viability which can be guaranteed by any protocol of a Federal Byzantine Agreement, regardless of the network, by taking into account the specific $\{V; Q\}$ and an insecurely behaving specific subset of *V*.

First of all, we discuss the intersection of a quorum - a feature without which security cannot be guaranteed. Afterwards, we explain the concept of optional sets — sets of failed nodes, without which both security and viability of the network can be guaranteed.

Quorum intersection

A protocol can guarantee an agreement only if the slice quorum is represented by a function Q, satisfying the confidence property – something we call a quorum intersection.

A quorum intersection – FBAS a enjoys *quorum intersection* iff any two of its quorums share a node—i.e., for all quorums *U*1 and *U*2, *U*1 n *U*2 Ç Ø.

In fact, no protocol can guarantee safety in the absence of quorum intersection, since such a configuration can operate as two different FBAS systems that do not exchange any messages. However, even with quorum intersection, safety may be impossible to guarantee in the presence of ill-behaved nodes.

In general, FBAS $\{V; Q\}$ can survive a Byzantine fault with a set of nodes $B \subset V$ if and only if $\{V; Q\}$ crosses quorum after removing B nodes from V and from all slices in Q.

DEFINITION: if $\{V; Q\}$ is a FBAS, and $B \subset V$ is a set of nodes, then removing B from $\{V; Q\}$, recorded as $\{V; Q\}^B$, means calculating the modified FBAS $\{V \setminus B, Q^B\}$ where
$Q^B(v) = \{q \setminus B \setminus q \in Q(v)\}$.

Each node V is responsible for ensuring that $Q(v)$ does not violate the intersection of the quorum. One way to do this is to select conservative slices that result in large quorums.

DSets

We capture the fault tolerance of nodes' slice selections through the notion of a dispensable set or DSet. Informally, the safety and liveness of nodes outside a DSet can be guaranteed regardless of the behavior of nodes inside the DSet. Put another way, in an optimally resilient FBAS, if a single DSet encompasses every ill-behaved node, it also contains every failed node, and conversely all nodes outside the DSet are correct. As an example, in a centralized PBFT system with 3ƒ + 1 nodes and quorum size 2ƒ + 1, any ƒ or fewer nodes constitute a DSet. Since PBFT in fact survives up to ƒ Byzantine failures, its robustness is optimal.

DSet definition

Let FBAS be $\{V; Q\}$, and $B \subset$ be the set of nodes. We state that B is a DSet if:

1. (the intersection of the quorum, despite B) $\{V; Q\}^B$ has the intersection of the quorum,
2. and (the presence of a quorum, despite V \ B) is a quorum in $\{V; Q\}$ or B = V.

With all things being equal, larger slices lead to an increase in quorums with a large overlap, which means that fewer unsuccessful sets of nodes *B* will undermine the intersection of the quorum during removal. On the other hand, large slices are more likely to contain failed nodes, which jeopardizes the availability of a quorum.

DSets in FBAS are a priori defined by the function of the quorum Q. Which of the nodes behave safely or insecurely depends on their "behavior at run time", e.g., when hacking machines of the nodes. The DSet sets covered in this section cover all nodes of unsafe behavior since they help us to distinguish between nodes with guaranteed safe behavior and those without such a guarantee. To do this, we introduce the following terms:

DEFINITION (INTACT). A node υ in an FBAS is intact if there is a DSet *B* that contains all non-secure behaved nodes and such that $\upsilon \in B$.

DEFINITION (BEFOULED). A node υ in FBAS is befouled if it is not damaged.
A befouled node υ is surrounded by a sufficient number of unsafe nodes to block its progress or poison its condition, even if υ behaves well.

THEOREM 1. Let U be the quorum in FBAS $\{V; Q\}$, let $B \subseteq V$ be the set of nodes, and let $U` = U \backslash B$. If $U` \neq \varnothing$, then $U`$ is the quorum in $\{V; Q\}^B$.

PROOF. Since *U* is a quorum, at each node $\upsilon \neq U`$there is a $q \in Q(\upsilon)$ such that $q \subseteq U$. Since $U` \subseteq U$, hence every $\upsilon \in U`$ has $q \in Q(\upsilon)$ such that $q \backslash B \subseteq U`$. Overwriting the deletion record gives $\forall_v \in U`, \exists q \in Q^B(\upsilon)$ such that $q \subseteq U`$, which, since $U` \subseteq V \backslash B$, means that $U`$ is a quorum in $\{V; Q\}^B$.

$U` \in FBAS \{V; Q\} \cap U` \in \{V; Q\}^B;$

If $B \subseteq V,\ U` = U \backslash B,\ U` \neq \varnothing$;

$\upsilon \neq U = q \in Q(\upsilon) = q \subseteq U;$

$\upsilon \in U` \cap q \in Q(\upsilon) = q \backslash B \subseteq U`;$

$\forall_v \in U`, \exists q \in Q^B(\upsilon) \cap U` \in \{V; Q\}^B;$

Federated voting

In this section, we will describe a federated voting method that FBAS nodes can use to align statements. At a high level, the process of matching some operator *a* includes nodes exchanging two sets of messages. First, the nodes vote. Then, if the voting is successful, the nodes confirm it by actually conducting a second vote on the fact that the first vote was

successful. From the point of view of each node, two rounds of messages divide the expression of *a* into three phases: unknown, accepted, and confirmed.

a - is the state of the network at the time of the action, which may be unknown, accepted or confirmed, depending on the node's behavior: a node and a node with bad behavior (an ill-behaved node).

Open membership voting

A secure node in the Federated Byzantine Agreement System (FBAS) acts on the statement *a* only when it knows that other safe nodes will never agree with the statements contradicting *a*. Most protocols use voting for this purpose. Nodes with safe behavior vote for the statement only if it is valid. Nodes with safe behavior also never change their votes. Consequently, in a centralized Byzantine Agreement, secure activity is accepted if a quorum with a majority of safely behaving nodes voted for it. We say that the statement is ratified when it receives the necessary votes.

DEFINITION (VOTE). A node votes for an (abstract) statement if:

1. υ states *a* that it is valid and consistent with all statements adopted by υ, and
2. υ claims that it never voted against, i.e., voted for a statement contrary to *a* -, and υ promises never to vote against *a* in the future.

DEFINITION (RATIFY). Quorum U_a ratifies declaration *a* if each member U_a votes in favor. A node υ ratifies *a*, if υ is a member of the quorum U_a, ratifying *a*.

THEOREM 2. Two contradictory statements *a* and $\underline{a}$ cannot be simultaneously ratified by FBAS, which has a quorum intersection and does not contain nodes with unsafe behavior.

PROOF. Contradiction. Let's suppose a quorum of U_1 ratifies *a*, and a quorum of U_2 ratifies $\underline{a}$ by the intersection of quorum $\exists \upsilon \in U_1 \cap U_2$. Such υ must have illegally voted for both *a* and $\underline{a}$, violating the assumption that there are no nodes with unsafe behavior.

$$U_1 \in a$$
$$U_2 \in \underline{a}$$
$$\exists \upsilon \in U_1 \cap U_2,\ U_1 \in a \neq U_2 \in \underline{a}$$

Blocking sets

In centralized consensus, liveness is an all-or-nothing property of the system. Either a unanimously well-behaved quorum exists, or else ill-behaved nodes can prevent the rest of the system from accepting new statements. In FBA, by contrast, liveness may differ across nodes.

An FBA protocol can guarantee liveness to a node *u* only if Q(*u*) contains at least one quorum slice comprising only correct nodes. A set *B* of failed nodes can violate this property if *B*

contains at least one member of each of *u*'s slices. We term such a set *B u*-blocking, because it has the power to block progress by *u*.

DEFINITION (U-BLOCKING). Let *u* c **V** be a node in FBAS **V**, **Q** . A set B ≤ **V** is *u-blocking* iff it overlaps every one of *us* slices—i.e., 6*q* c **Q**(*u*), *q* n *B* Ç Ø.

THEOREM 3. Let $B \subseteq V$ be the set of nodes in FBAS $\{V;Q\}$. $\{V;Q\}$ has the availability of quorum, despite the fact that *B*, if *B* is not υ-blocking for any $\upsilon \in V \setminus B$.

PROOF. B is not υ-blocking equivalently to $\forall \upsilon \in V \setminus B,\ \exists q \in Q(\upsilon)$ so that $q \subseteq V \setminus B$. By definition of a quorum, the latter takes place if *V* \ *B* is a quorum or B = V, a precise definition of the presence of a quorum, despite B.

As a result, the compromised nodes included in the DSet are not υ-blocking for any integer υ.

$B \subseteq V \in FBAS\ \{V;Q\}$

$\forall \upsilon \in V \setminus B \neq B$

$V \setminus B \in FBAS\ \{V;Q\}$, $\{B = V\} \in \{V;Q\}$

Accepting statements

When an intact node *u* learns that it has ratified a statement, the theorems tells *u* that other intact nodes will not ratify contradictory statements. This condition is sufficient for *u* to accept *a*, but we cannot make it necessary. Ratifying a statement requires voting for it, and some nodes may have voted for contradictory statements.

DEFINITION (ACCEPT). The FBAS node υaccepts an operator if it never accepted an operator that contradicts *a*, and determines that either:

1. There is a quorum of U such that $\upsilon \in U$, and each member of U either voted in favor or declared acceptance of, or
2. Each member of the υ-blocking set declares acceptance of *a*.

Failure to accept network state assertions

In a federated consensus protocol, the nodes accepting the rationale for accepted network state assertions will be provided with suboptimal guarantees of security and survivability.

Security

Consider the FBAS $\{V;Q\}$, in which the only quorum is a unanimous agreement, i.e., $\forall_a,\ Q(\upsilon) = \{V\}$. It should be a conservative choice for the sake of security - do nothing until everyone agrees. However, since each node is υ-blocking for each υ, any node can on its own convince any other node to accept arbitrary statements.

Liveness

Another limitation of accepted statements is that other intact nodes may not be able to accept them. This feature makes bets on accepted statements problematic for vitality. If a node continues to interact with the statement because it has accepted the statement, other nodes may not be able to act in the same way.

Definition (agree). The FBAS $\{V;Q\}$ agrees with the statement, regardless of what happens afterwards, if after delivering and processing a sufficient number of messages, each intact node will accept *a*.

Statement confirmation

Both limitations of accepted statements stem from complications when a set of intact nodes *S* votes against a statement *a* that is nonetheless ratified. Particularly in light of FBA's non-uniform quorums, *S* may prevent some intact node from ever ratifying *u*. To provide *u* a means of accepting *a* despite votes against it, the definition of *accept* has a second criterion based on *u*-blocking sets. But the second criterion is weaker than ratification, offering no guarantees to befouled nodes that enjoy quorum intersection.

Now suppose a statement *a* has the property that no intact node ever votes against it. Then we have no need to accept *a* and can instead insist that nodes directly ratify *a* before acting on it. We call such statements irrefutable.

DEFINITION (IRREFUTABLE). Statements of *a* are incontrovertible in FBAS if none of the intact nodes can ever vote against it.

DEFINITION (CONFIRM). The quorum U_a in the FBAS confirms the statement if $\forall_a \in U_a$,υ declares acceptance. A node confirms *a* if it is in quorum U_a.

THEOREM 4. Let $\{V;Q\}$ be the FBAS with a quorum intersection, despite B, and assume that B contains all ill-behaved nodes. Let υ_1 and υ_2 be two nodes not belonging to B. Let *a* and $\underline{a}$ be contradictory statements. If υ_1 confirms *a*, then υ_2 cannot confirm $\underline{a}$.

PROOF. First off, please note that the acceptance of *a* contradicts the acceptance of ($\underline{a}$) - no node with safe behavior can vote for both. Please note that υ_1 must accept ($\underline{a}$) to confirm *a*.

$$\upsilon_1, \upsilon_2 \notin B,\ a \neq \underline{a}$$

Liveness and neutralization

The main issue of a distributed consensus, centralized or not, is that the statement may be stuck in a constantly uncertain state before the system reaches an agreement on it. Hence, a protocol must not attempt to ratify externalized values directly. If the statement "The value of slot *i* is equal to *x*" is stuck, the system will never be able to agree on slot *i*, losing viability. The

solution is to carefully consider statements in the votes. It should be possible to get rid of the stuck assertion in a question that we really care about, namely in the contents of the slot. We call the process of obsolescence of the stuck operator a neutralization.

At the time an FBAS transitions from bivalent to *a*-valent, there is a possible outcome in which all intact nodes accept *a*. However, this might not remain the case. Consider a PBFT-like four-node system {*u*1, … , *u*4} in which any three nodes constitute a quorum. If *u*1 and *u*2 vote for *a*, the system becomes *a*-valent; no three nodes can ratify a contradictory statement. However, if *u*3 and *u*4 subsequently vote for a^- contradicting *a*, it also becomes impossible to ratify *a*. In this case, *a*'s state is permanently indeterminate, or *stuck*.

Fincor Consensus Protocol (FCP)

In this section we present the FCP consensus protocol. At a high level, FCP consists of two sub-protocols: the nomination protocol and the protocol of voting. The nomination protocol generates candidate values for the slot. In case it runs for quite long period of time, finally it creates the same set of candidate-values on each intact node. That means that the nodes can combine candidate values in a chosen way to create a single composite value for the slot. However, there are two substantial warnings. First, nodes cannot know when the destination protocol has reached the convergence point. Secondly, even after convergence, nodes with unsafe behavior can reset the nomination process a finite number of times.

Nomination protocol

The nomination protocol works by converging the set of candidate values for a slot. Then the nodes combine these candidates into one composite value for the slot. It depends on the application how exactly the values are combined . As an example, the network uses FCP to select a transaction set and a ledger timestamp for each slot. To combine candidate values, Fincor takes the union of their transaction sets and the maximum of their timestamps (values with invalid time marks will not receive enough nominations to become candidates). Other possible scenarios include combining sets by intersecting or simply selecting the candidate value with the highest hash. The nodes give the value of candidate *x* by means of a federal vote in the approval of nominee *x*.

DEFINITION (CANDIDATE). A node *u* considers a value *x* to be a *candidate* when *u* has confirmed the statement *nominate x*—i.e., *u* has ratified *accept* (*nominate x*).

The nomination process will be more efficient if there are fewer value combinations in the game. Therefore, we assign the temporary priority to the nodes, and each node, when possible, assigns the same values as the node with the higher priority. More concretely, let *H* be a cryptographic hash function whose range can be interpreted as a set of integers $\{0, \dots, hmax - 1\}$. (*H* might be SHA-256 [National Institute of Standards and Technology 2012], in which case $hmax = 2^{256}$.) Let $Gi(m) = H(i, xi-1, m)$ be a slot-specific hash function for slot *i*, where $xi-1$ is the value chosen for the slot preceding *i* (or the sorted set of values of all immediate dependencies of slot *i* when

slots are governed by a partial order). Given a slot *i* and a *round number n*, each node *u* computes a set of *neighbors* and a *priority* for each neighbor as follows:

$$weight\,(v, v`) = \frac{|\{q \,|\, q \in Q(v) \wedge v` \in q\}|}{|Q(v)|}$$

$$neighbors\,(v;\ n) = \{v` \,|\, G_i(N, n, v`) < h_{max} \cdot weight\,(v, v`)\}$$

$$priority\,(v, v`)) = G_i(P, n, v`)$$

N and P are constants for creating two different hash functions.
The weight $(v, v`)$ function returns the slice fraction in $Q(v)$ containing $v`$. Using weight as probability as $n, v`$ appears in $neighbors\,(v;\ n)$, we also reduce the probability that nodes without much confidence will dominate the round.

Ballot protocol

Once nodes have a composite value, they participate in the ballot protocol, although the nomination may simultaneously continue updating the composite value. Bulletin *b* is a pair of the form $b = \{n, x\}$, where $x \neq \perp$ is the value and *b* is a referendum on externalization *x* for the slot in question. The value $n \geq 1$ is a counter, ensuring that higher numbers of ballots are always available. We use *C*-like notation *b.n* and *b.x* to designate the counter fields and values of the bulletin *b*, so $b = \{b\,.\,n,\ b\,.\,x\}$. Ballot papers are fully ordered, with *b.n* more significant than *b.x*. For convenience, a special invalid null bulletin $0 = \{0, \perp\}$; less than all other bulletins, and the value of a special counter is greater than all other counters.

DEFINITION (COMPATIBLE). Two bulletins *b1* and *b2* are compatible, recorded as $b_1 \sim b_2$, if $b_1\,.\,x = b_2\,.\,x$ and incompatible, recorded as $b_1 \nsim b_2$, if $b_1\,.\,x \neq b_2\,.\,x$. We also write $b_1 \lesssim b_2$or $b_2 \gtrsim b_1$ if and only if $b_1 \leq b_2$ (or equivalently $b_2 \geq b_1$) and $b_1 \sim b_2$. Similarly, $b_1 \not\lesssim b_2$ or $b_2 \not\gtrsim b_1$ means $b_1 \leq b_2$ (or equivalently $b_2 \geq b_1$) and $b_1 \nsim b_2$.

DEFINITION (PREPARED). Bulletin *b* is prepared if each statement in the following set is true: $\{abort\,b_{old} | b_{old} \not\lesssim b\}$.

Correctness

The FCP node cannot vote to confirm *commit b* until it has voted to confirm *abort* for all lower-numbered incompatible ballots. Since a node with safe behavior cannot take conflicting statements (and, therefore, vote to confirm), this means and guarantees for a given $\{V; Q\}$, that many nodes with safe behavior cannot externalize conflicting values while *S* has an intersection quorum despite *V\S*. This safety is maintained if *V* and *Q* only change between slots, but what if they change the middle slot (e.g., in response to a node failure)?

In terms of security during reconfiguration, we combine all the old and new sets of quorum slices, reflecting the fact that nodes can make decisions based on a combination of messages

from different configuration eras. To be conservative, we may need to intersect the aggregation quorum of the current configuration with each past configuration. However, we can weaken this a bit by separating the nodes that sent the invalid messages from those that failed.

Implementation of sharding in FBAS and FCP protocol

To implement sharding into the network, you need to change the transaction structure at the following levels:

- client's side (immediate split of transaction {tx} into shards before sending);
- server's side (the validator nodes υ- *valid*, the collector nodes $\upsilon \in tx_{\{shard\}}$ after validation and the nodes υ- *history,* that store history).

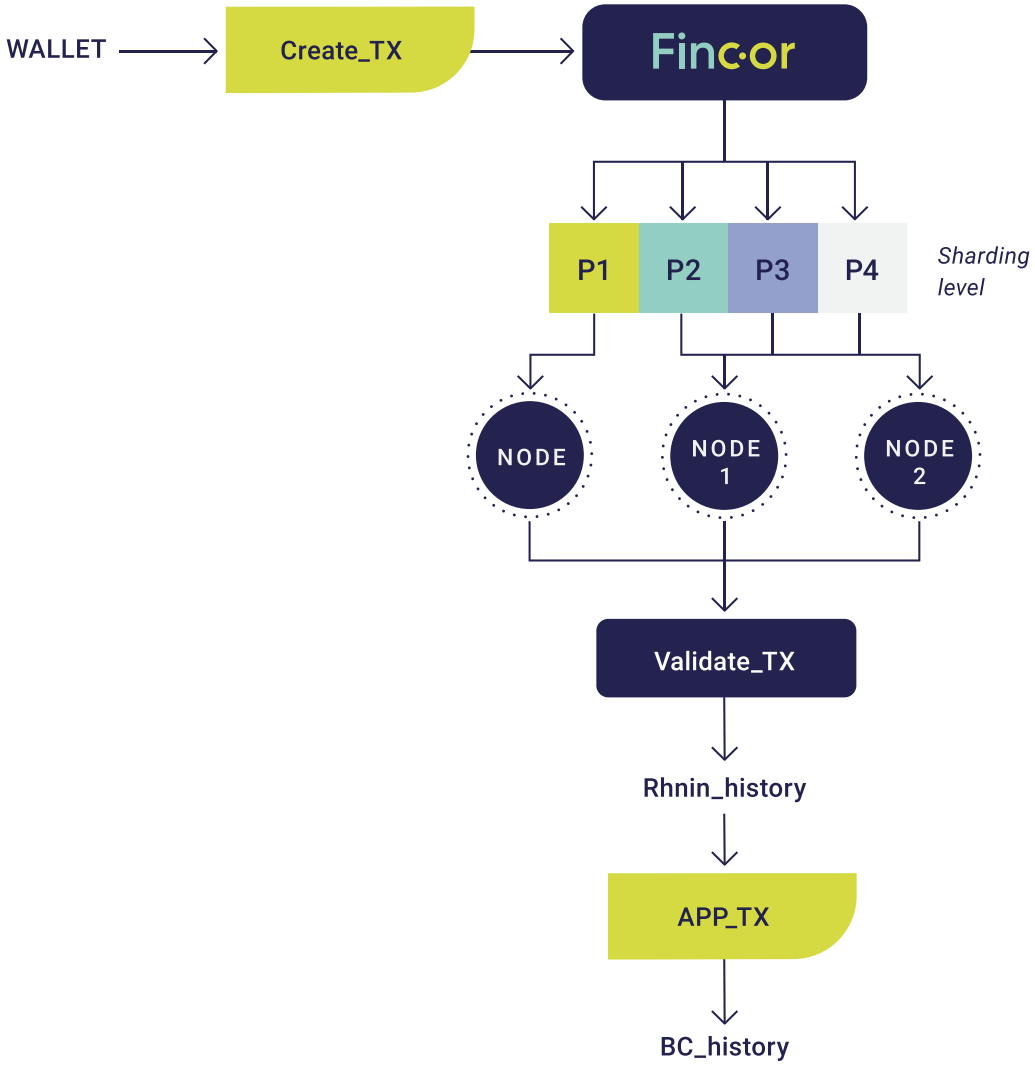
The validator nodes will keep the block history, which all nodes have in common as the nodes also hold the network). After validation, nodes that validate shards collect *{tx}* into normalized transaction *{tx}* $\in normal$.

The concept of implementing sharding in Fincor blockchain will allow the network to validate the transaction a lot faster. The validator nodes υ- *valid* will work as validators (not to be confused with classical nodes). Implementing the sharding concept will allow us to speed up the processing and validation of a transaction.

Algorithm for implementing sharding technology within Fincor blockchain

- The user conducts a transaction
- The user conducts a transaction
- The wallet on the user's device splits the transaction into shards
- A node, which collects shards after validations, creates a unique identifier for parts of the transaction
- The validators validate parts of the transaction
- The validators collect the transaction into a block,

$$\{tx\} \Rightarrow a(wallet) \Rightarrow tx_{\{shard\}} \cap i\{tx_{\{shard\}}\} \Rightarrow \upsilon \in tx_{\{shard\}} \Rightarrow \{tx\}.$$



The implementation of sharding within Fincor blockchain will depend on the file entering the network in order to ensure a fast transaction (basically, it is also the configuration file 'fincor.toml' which is also a configuration file node $\upsilon \in tx_{\{shard\}}$). It can be functionally divided into parts (hashes, addresses, etc.)and subsequently collected into a single block transaction after validation.

Below is an example of a 'fincor.toml' file. As you can see, it can work with ledgers and, more specifically, with contracts within the network. Finally, we can get the implementation of sending funds in order to get a token: $GET_{token} = (B_{address} + T_{address} + amount)/3$ (for shards when sending funds to the address of the token).

However, as a more optimal solution for implementation, we leave 'fincor.toml' as a configuration file for the server part of the $\upsilon \in tx_{\{shard\}}$ nodes, since for splitting a transaction, creating shards at the $tx_{\{shard\}}$ transaction data level (addresses, number of sent / received coins and memo).

Example of fincor.toml

fincor.toml

```
FEDERATION_SERVER="https://api.fincor.org/federation"

# The endpoint used for the compliance protocol
AUTH_SERVER="https://api.fincor.org/auth"

# The signing key is used for the compliance protocol
SIGNING_KEY="GBBHQ7H4V6RRORKYLHTCAWP6MOHNORRFJSDPXDFYDGJB2LPZUFPXUEW3"
# convenience mapping of common names to node IDs.
# You can use these common names in sections below instead of the less friendly nodeID.
# This is provided mainly to be compatible with the fincor-core.cfg
NODE_NAMES=[
"GD5DJQDDBKGAYNEAXU562HYGOOSYAEOO6AS53PZXBOZGCP5M2OPGMZV3  lab1",
"GB6REF5GOGGSEHZ3L2YK6K4T4KX3YDMWHDCPMV7MZJDLHBDNZXEPRBGM  donovan",
"GBGR22MRCIVW2UZHFXMY5UIBJGPYABPQXQ5GGMNCSUM2KHE3N6CNH6G5  nelisky1",
"GDXWQCSKVYAJSUGR2HBYVFVR7NA7YWYSYK3XYKKFO553OQGOHAUP2PX2  jianing",
"GAOO3LWBC4XF6VWRP5ESJ6IBHAISVJMSBTALHOQM2EZG7Q477UWA6L7U  anchor"
]



#   A list of accounts that are controlled by this domain.
ACCOUNTS=[
"$sdf_watcher1",
"GAENZLGHJGJRCMX5VCHOLHQXU3EMCU5XWDNU4BGGJFNLI2EL354IVBK7"
]

#   Any validation public keys that are declared
#   to be used by this domain for validating ledgers and are
#   authorized signers for the domain.
OUR_VALIDATORS=[
"$sdf_watcher2",
"GCGB2S2KGYARPVIA37HYZXVRM2YZUEXA6S33ZU5BUDC6THSB62LZSTYH"
]
# DESIRED_BASE_FEE (integer)
# This is what you would prefer the base fee to be. It is in stroops.
DESIRED_BASE_FEE=100
# DESIRED_MAX_TX_PER_LEDGER (integer)
# This is how many maximum transactions per ledger you would like to process.
DESIRED_MAX_TX_PER_LEDGER=400
#   List of IPs of known fincor-core's.
#   These are IP:port strings.
#   Port is optional.
#   By convention, IPs are listed from most to least trusted, if that information is known.
KNOWN_PEERS=[
"192.168.0.1",
"core-testnet1.fincor.org",
"core-testnet2.fincor.org:11290",
"2001:0db8:0100:f101:0210:a4ff:fee3:9566"
]
# list of history archives maintained by this domain
HISTORY=[
"http://history.fincor.org/prd/core-live/core_live_001/",
"http://history.fincor.org/prd/core-live/core_live_002/",
"http://history.fincor.org/prd/core-live/core_live_003/"
]



#   This section allows an anchor to declare currencies it currently issues.
#   Can be used by wallets and clients to trust anchors by domain name
[[CURRENCIES]]
code="USD"
issuer="GCZJM35NKGVK47BB4SPBDV25477PZYIYPVVG453LPYFNXLS3FGHDXOCM"
display_decimals=2 # Specifies how many decimal places should be displayed by clients to end users.
[[CURRENCIES]]
code="BTC"
issuer="$anchor"
display_decimals=7 # Maximum decimal places that can be represented is 7
# asset with meta info
[[CURRENCIES]]
code="GOAT"
issuer="GD5T6IPRNCKFOHQWT264YPKOZAWUMMZOLZBJ6BNQMUGPWGRLBK3U7ZNP"
display_decimals=2
name="goat share"
desc="1 GOAT token entitles you to a share of revenue from Elkins Goat Farm."
conditions="There will only ever be 10,000 GOAT tokens in existence. We will distribute the revenue share annually on Jan. 15th"
image="https://pbs.twimg.com/profile_images/666921221410439168/iriHah4f.jpg"
#   Potential quorum set of this domain's validators.
[QUORUM_SET]
VALIDATORS=[
"$self", "$lab1", "$nelisky1","$jianing",
"$eno","$donovan"
]

```

Squared sharding

Squared sharding - a variation of sharding, allowing to validate significantly higher number of nodes for node-validators. It works due to the increased number of validators, checking shards from one user (user with specific ID).

Sharding is often presented as a solution for infinite scaling with an increase in the number of nodes. Probably, you can really create a system with this property, but systems with a central blockchain have a upper limit of the number of shards, and as a result do not have infinite scalability. It is easy to understand why: the central blockchain performs some calculations, such as assigning validators and storing the latest states of the shard, whose complexity is proportional to the number of the shard. Since the central blockchain itself is not shaded, and its bandwidth is limited by the bandwidth of each node, the number of shards that it can support is limited.

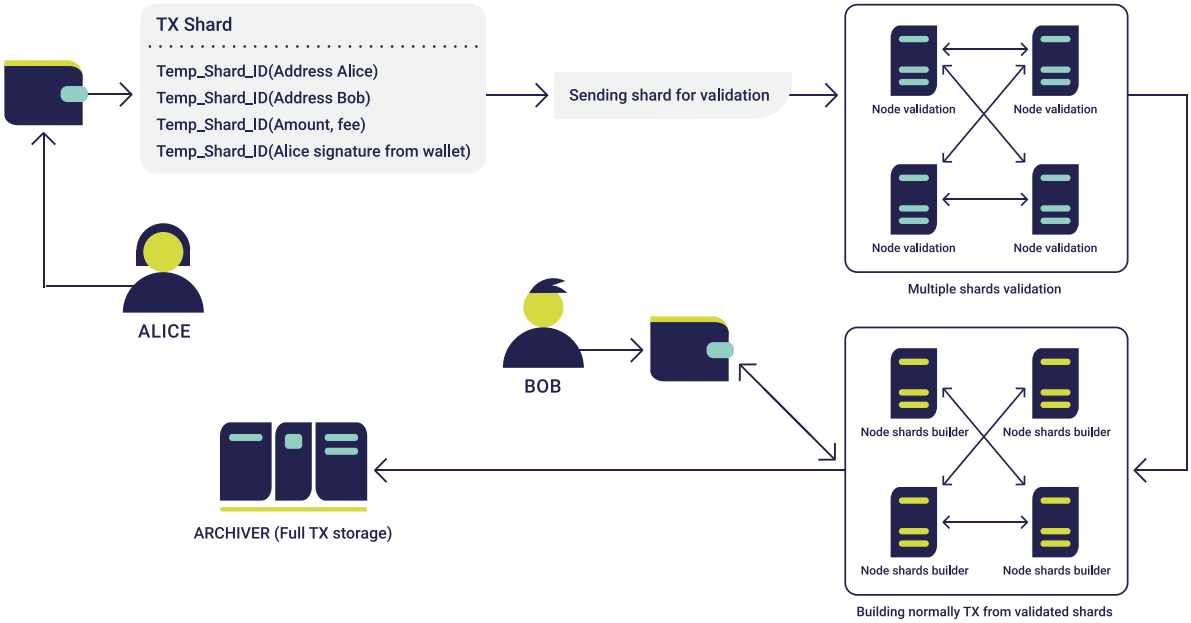
A squared sharding will make sense if the power of the nodes supporting it grows n times. Each shard will be able to process n times more transactions, and a validating shards node will be able to support a times larger shard. Thus, the bandwidth of the entire system will grow n ^ 2 times.

We will use quadratic sharding, which will scale the network in the context of increased transaction processing per second, increased transaction validation security, and a fault-tolerant network (due to validations and network capacity).

$$FIN_{netw\ speed} = (Node_{validators} + Node_{builder} + Node_{archiver})*quantity$$

This model of sharding is strongly dependent on the number of nodes, which make it possible to obtain a scalable network.

From a technical point of view, after implementing quadratic transaction shardbook, we get a scalable, secure network that is designed for quick exchange of funds.



Basic implementation of transactions in the network

Transactions are commands that change the state of the ledger. Among other things, transactions are used to send payments, enter orders in a decentralized exchange, change settings in accounts and authorize another account to store currency.

Set of transaction

In between the closure of the ledgers, all nodes on the network collect transactions. When it is time to close the next ledger (block), the nodes collect these transactions into a set of transactions. FCP is managed by the network to reach an agreement where transaction will be associated with the last ledger.

Life cycle (flow)

CREATION: the user creates a transaction, fills in all the fields, assigns the correct sequence number for the operation/account, adds any operations he/she wants, etc.

SPLIT: $FNRtx = A_{addr} + amount + B_{addr} + sign_{tx}$ (description of the basic transaction with the signature from the sender).

SIGNING: after the transaction is completed, all the necessary signatures must be collected and added to the transaction envelope. Usually, this is just the signature of the account performing the transaction, but more complex settings may require collecting signatures from several parties, e.g., if Multisig is implemented.

SENDING: after signing, the transaction is valid and can now be sent to the Fincor network. Transactions are usually sent using the API.

DISTRIBUTION: as soon as the Fincor kernel receives a transaction (or rather, the core sends transaction shards for validation to the validation node/validation nodes υ- *valid*), provided to it by the user or another Fincor core, it performs preliminary checks to check the validity of the transaction. Among other checks, it guarantees that the transaction will be properly created and the balance of the original account will be sufficient to cover the amount of the transaction. Fincor-core does not check the status of the ledger other than searching for the original account. If preliminary checks are passed, Fincor-core spreads the transaction to all other connected servers. Thus, a valid transaction is fed into the entire Fincor network.

INCLUSION IN THE TRANSACTION SET: when it is time to close the ledger, Fincor-core accepts all the transactions it received after closing the last ledger and collects them into a set of transactions. If it discovers any incoming transactions, it sets them aside until the next ledger is closed. Fincor-core assigns a set of transactions it has collected.

APPENDIX: as soon as FCP agrees with a specific set of transactions, this set applies to the ledger. Operations are performed in the order they occur in the transaction. If any operation fails, the entire transaction is not executed, and the results of previous operations in this transaction

are rolled back. After applying all the transactions in the set, a new ledger is created (block), and the process begins anew.

Another step for implementing sharding is splitting a transaction into specific parts based on the data it contains (separating addresses, how many coins were sent/received, fee charge etc.). These parts will be assigned with a specific identifier in order to assemble the transaction into a single part after validation.

Integration of sharding on standard hardware can overload the network, since the highest load on the network are processed by nodes-validators υ- valid. If you create a simple validation, you will also get a less secure network.

However, the following steps will allow us to develop a faster network:

- change HTTP to HTTP/2 (HyperText Transfer Protocol, HTTP) is an application-level Protocol of data transmission (initially — in the form of hypertext documents in the format of "HTML", currently used to pass arbitrary data). HTTP is the basis of technology “client-server";
- change additional databases in LevelDB (LevelDB is a high-performance NoSQL system for storing data in the format key/value developed by Google. The LevelDB store is written in C++ and connects to applications as shared libraries (like SQLite and BerkeleyDB), providing the ability to store ordered data sets i where the string keys mapped to string values;
- LevelDB stores keys and values in arbitrary byte arrays, while data is sorted by key. It supports batch entry, forward and reverse iteration, and the data compression using Google's Snappy compression library;
- LevelDB is used as the backend database for Google Chrome's IndexedDB. In addition, the kernel stores the Bitcoin blockchain metadata using LevelDB database;
- Data is stored in the SS table (Sorted String Table) as pairs of key/value. The multitude of SS tables form a LSM-tree. LSM-trees (Log-structured merge-tree) consist of several storage tiers. The first (zero, or MemTable) is in memory, and the rest are on the disk. Each level is a tree or list of trees, and each level has a limit size growth (usually 10 times per level).

By replacing the Horizon bridge for application with the database (MySQL/Postgres), changing the database to LevelDB(NoSQL) and the protocol on which the network will work (HTTP/2, TCP), we achieve a faster network (x2 increment thanks to this solution):

$$TX_{per\ second} = (Current_{tx\ speed} * LDB_{tx\ speed}) * 0.00001 = (1000 * 5210000) * 0.000001 = 5210$$

This is solely achieved by implementing the replacement of the communication protocol and the sharding provides double increase allowing us to reach $\approx 10\ 000\ tx$

$$ShardsTX_{per\ second} = (TX_{per\ second} * Val_{nodes}) / Link_{nodes}$$

$$FinalTX_{per\ second} \left\{ s^{N_{bandwidth}}_{Net\ speed} node_{sum},\ node_{load} \right\}$$

In order to solve the issue of peak loads on validator nodes υ- *valid*, we apply the solution below. There will be 3 main units of the implementation of the federation server:

- shard validator;
- a standard node and its variations, which contain a copy of the ledger;
- the node for the assembly of the $\upsilon \in tx_{\{shard\}}$ transaction after validation (it will be divided into separate parts of the transaction that have been validated, and these parts will be collected into a single transaction, the so-called new $\{tx\}$ collector, after which the collected $\{tx\}$ are saved in block chains).

Each node has its own configuration file, in which there is a choice of the role of the node.

Description of the used nodes:

	Archiver	**Basic validator**	**Full validator**
Description	Storing info from full validator	Validation TX shard	basic validator + building TX + publish to archive
Transaction validation	yes	yes	yes
API support	yes	yes	yes
Consensus participation	no	yes	yes
Helps other nodes synchronize and join the network	yes	no	yes
Work with shards	no	Yes	no
Increases network resiliency	Medium	Medium	High

The processing speed and the formation of a transaction directly depend on several factors.
A typical case:

- a large number of validator nodes - υ- *valid;*
- a large number of nodes that collect a transaction from valid shards $\upsilon \in tx_{\{shard\}}$
- $FincorTX_{speed} = (basefee^2 + node_{sum} + node_{valid\ sum})$
- $FincorTX_{shard\ valid} = FincorTX_{speed} + val_{steps}$

The algorithm for splitting into shard transactions $tx_{\{shard\}}$

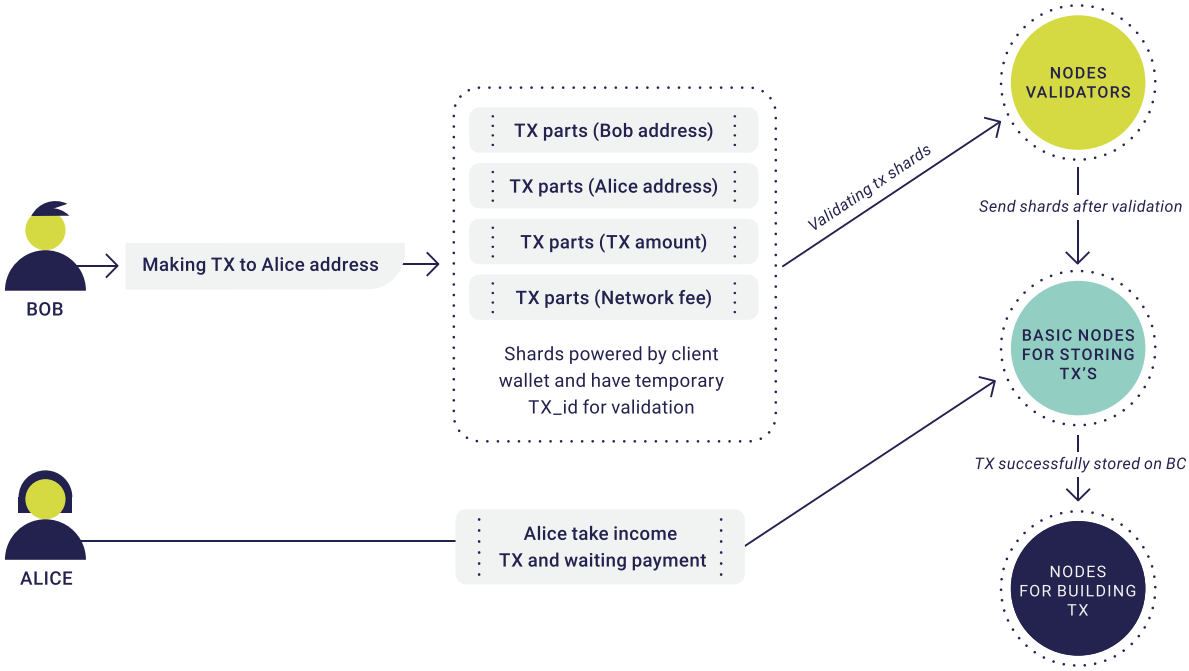
Since we know the type of transaction and have data about the created transaction and its parameter set, we use the required number of parameters corresponding to this type of transaction and divide the transaction into shards according to the rules written in 'fincor.toml'.

FincorTX = $(address\,A + address\,B + amount + fee)/n + tempTXID\,(for\,validation)$
(initialization of splitting the transaction into separate shards for the implementation of the sharding mechanism)

$TX_{for\,validation} = check\,addresses\,(A\,and\,B) + checking\,amount + checking\,network\,fee$
(transaction formula after the split into shards, which is ready for validation)

$TX_{after\,validation} = TXID + TX_{for\,validation} + block\,\#$
(the status of the transaction after validation and the process of reassembling the shards of the transaction using the collector nodes in order to get the TX in a usual for the network form)

Figure 2. Implementing the online transaction flow



The diagram above describes the implementation of the transaction in the network.

The diagram clearly shows that we have a transaction split into shards on the client side, the client signs the transaction, and the transaction is divided into shards, after which we get a three-level passage within the network:

Shards first pass through the validation node, which checks the signature with the client's public key, the ID of the shards, the recipient's address and the network fee, after which the v- *valid* validating node signs the shards again and sends them to the tx {shard} node $v \in tx_{\{shard\}}$, which collects a single transaction.

$Valid_{shards} = FincorTX + Validation_{sign\ node\ key}$ ($Validation_{sign\ node\ key}$

is generated based on a one-time private key / token for this transaction)

$Validation_{sign\ node\ key} = Private_{node\ key} + temp_{txID} + Node_{ID}$

The node which collects the transaction collects the shards into a single transaction and signs the transaction completely and afterwards we get the usual * cryptocurrency-like * transaction, which is sent to the common Fincor network, where the user receives the status of the confirmed transaction after committing to the transaction nodes. Consequently, the recipient receives his/her coins.

$$Processed_{tx} = Valid_{shards} - temp_{txID} + Network_{txID} + Network_{txHASH}$$

($temp_{txID}$ is removed from the verified transaction to get the main transaction ID based on the temporary transaction ID; based on this, the generation of the main *txID* works due to the temporary ID + signature with a unique key for signing this transaction).

$temp_{txID} = (A_{sign} + node_{val\ sign}) + Shard_{send\ sign}$($Shard_{send\ sign}$ is responsible for the shard's signature when sending).

A node storing transactions provides a transaction in the usual way.

$TX = Processed_{tx} + TX_{info}$ (TX_{info} means all information about the transaction, specifically the recipient's address and the sender's address, the number of sent coins, the network fee, the nonce and the sender's signature, the validator node and the node that collects the transaction based on the checked shards and signature of the node, which validates the transaction)

At the same time, we do not change the FBAS consensus $\{V; Q\}$ leaving it in an original state, which is important.

Implementing NFT (Non-fungible Tokens)

NFT is a way to define sets of elements when each element in these sets is individually unique.

$NFT_{uniqueness} = NFT_{key} + NFT_{token}$

Currently, the most well-known example of NFT is CryptoKitties of the Ethereum project. This token operates on the Ethereum network. However, NFT can be used in a broader range of applications— provision of ownership rights to both digital and physical assets or, for instance, tokenization of land, artwork, game resources, and software licenses.

Fincor-NFT-Issuer provides an API and code execution environment for releasing your own NFT on the platform.

Fincor can be used to digitize any assets (stocks, gold, oil, other assets), as well as custodial services by issuing NFT tokens.

Implementation details

Fincor-NFT-Issuer will use Fincor accounts to release its NFT tokens.

Metadata that uniquely identifies the set of token parameters is digitally signed by the issuer before being stored as an IPFS object.

$$NFT_{deploy} = T_{owner} + Sign_{owner}$$

The IPFS multihash, in turn, is used as a secret key for the account corresponding to the token.

The account uses its home domain to specify its token class and search method for the public key, used to verify its authenticity, and (optionally) as a method to search for the IPFS object's multihash in which its metadata is stored.

Ownership of the token is represented by the right of signature.

$$FNR_{token-storage} = T_{owner} + Addr_{owner} + T_{metadata} + T_{sign_{key}}$$

T - *Token*

$T_{metadata}$ - *Token metadata and token config*

T_{owner} - *Token owner*

$Addr_{owner}$ - *Token owner address*

Since the tokens in the first version are implemented as accounts, they cannot be sold on Fincor DEX.

Since fincor-nft tokens are implemented using Fincor accounts, they are subject to the same minimum balance requirements as on regular accounts. Therefore, the interaction is calculated in basic reserve fees.

$$FNR_{releaseT} = NFT_{deploy} + (accountt_{balance} - (T_{fee} + T_{deploy_{price}}))$$

The base reserve fee and the base fee are calculated on the basis of the transaction fee model.

The transaction fee is the number of operations within a transaction multiplied by the base fee, equal to 100 units: $(n_{operations} * basefee)$

In this case, the reserve fee also affects the minimum balance on the account. A distinction must be made between a base fee, which is equal to 100 minimum units, and a base reserve commission.

The minimum balance is calculated using the base reserve, which is 0.5 Fincor Coin:
$(2 + n_{entries})$ * *base reserve.*

The entries are:

- trustlines;
- suggestions;
- signers;
- data input.

Thus, for transactions with NFT, the cost of the commission will be considered as:

- 2 base reserve fees - account itself: *2 * base reserve*
- 1 base reserve fee - additional signatory representing the owner: *(2 + 1_{signer}) * base reserve*
- 1 base reserve fee - as a buffer for one additional signer when transferring ownership (a new subscriber must be added before deleting the old signer):
  $(2 + 1_{signer\ additional})$ * base reserve*

Additional base reserve fee will be added if/or when data input is used instead of reverse federation: $(2 + 1_{data\ entry})$ * base reserve*

Thus, the maximum possible fee can be:
$(2 + (2 + 1_{signer}) + (2 + 1_{signer\ additional}) + (2 + 1_{data\ entry}))\ *base\ reserve.$

Thus, after the implementation of the above functionality, we achieve faster transaction processing and a more secure network. Based on the replacement of Horizon and Mongo/ MySQL/Postgres on LevelDB + driver for working with Fincor, we receive faster transaction processing and with the integration of sharding we accomplish a more secure exchange of funds (since sharding allows for processing transaction parts, not the entire transaction).

Summary

In this paper, we provided a detailed explanation, arguments and calculations laying out and justifying our vision of the blockchain with an emphasis on efficiency, speed and security – which current popular platforms often lack.

This vision includes Fincor's own blockchain enhanced with network sharding – the most promising transaction processing technology out there. Unlike other blockchains, Fincor's blockchain will split transactions into pshards and process these separately, rather than as entire transactions. Sharding will enable us to bring transaction processing speed to a completely new level – a capacity of 10,000 transactions per second after the first deployment.

The reliability and security of Fincor's blockchain will be stronger and more efficient than those offered by existing platforms. Fincor's blockchain, including transactions processed on it, will offer better security thanks to a synergy between the Federated Byzantine Agreement and Fincor's own Fincor Consensus Protocol (FCP). It will allow us to build a blockchain with an augmented network resiliency and a decentralized network control while providing low entry barriers.

Fincor's blockchain, together with the entire ecosystem of smart financial products, will become a gateway for businesses, service and product developers, as well as end-users into a new era of financial services.