Alephium: a scalable cryptocurrency system based on blockflow

by Cheng Wang [Version 0.1]

www.alephium.org

Abstract

Since Bitcoin's success as the first widely used digital currency, cryptocurrency has attracted tons of attention. Although there are thousands of different digital currencies existing nowadays, many open problems and challenges are still remaining. Particularly, the scalability issue is of great importance in the field of cryptocurrency. We propose a new cryptocurrency system aiming to solve this critical issue.

Firstly, we propose a novel sharding algorithm supporting native cross-shard transactions that would improve the system throughput dramatically. We call our new technology blockflow, as an improvement of the well-known blockchain technology. With blockflow, it is possible to handle a thousand times of the transaction throughputs compared to traditional blockchain algorithms.

Secondly, we decompose smart contract into token protocol and data protocol to enable developers to build scalable blockchain applications easily. With our new protocol, application data and computation could be either totally decentralized or partly centralized. Developers have the flexibility to choose the tradeoff that matches the best their specific scenario.

In this paper, we also introduce our new decentralized cryptocurrency system Alephium based on blockflow and our protocols.

Keywords: cryptocurrency, blockchain, blockflow, scalability

1 Introduction

Currently, all the blockchain protocols based on Proof-of-Work (PoW) system can not process more transactions than a single node could. In practice, Bitcoin [2] and Ethereum [1] can handle <20 transactions per second. However, VISA could handle on average thousands of transactions per second (TPS). If we want to use cryptocurrency for all kinds of transactions world-widely, we would like to scale our decentralized protocol to handle at least thousands of TPS as well. This is usually related to the well-known scalability problem of cryptocurrency. If one wants to build a decentralized exchange application, the scalability problem must be solved to a certain level. In the paper, we will introduce a new cryptocurrency system that is scalable by using a novel sharding algorithm.

Our algorithm stands somewhere between blockchain algorithms and DAG algorithms. DAG algorithms for cryptocurrency usually have security issues as it's especially hard to reach dependency consistency among participants while ensuring reasonable scalability, e.g. IOTA [3] only works with probability or relies on a centralized coordinator at the time of writing. Our algorithm uses a scalable UTXO model to achieve scalability and consensus at the same time. In our algorithm, there are a number of parallel chains. Each chain is a group of transactions with certain requirements. All these parallel chains are dependent on each other. We design an algorithm to resolve the dependencies among all chains and to add new blocks onto these chains. Nodes in our system only need to keep part of the transactions of these chains, which contributes to scalability.

Another challenge of sharding algorithms is the design of smart contracts. First of all, we agree that smart contracts are very useful and result in many useful applications. However, like evolving from single threading programming to multithreading programming, scaling smart contract to work with multiple shards would rely on locks or methods similar to locks which would make applications complicated and inefficient. We propose a practical solution to this issue by decomposing contracts into a token part and a data part. We designed several typical types of tokens and a scripting language for programming at the level of tokens. Another problem we found is that currently the models and the scripting languages designed for smart contracts are tightly coupled with the cryptocurrency system. This makes the contract language very hard to design, upgrade and run safely on chain. Experience in programming language design reminds us that it's better to have different domain-specific languages for different scenarios. In our system, we separate our underlying protocol from the concrete computation model. We achieve this by only caring about data at the protocol level. Then third parties can provide their custom interpreter to parse the data. In this sense, it is very similar to the TCP/IP protocol.

The rest of the paper is organized as follows. In Section 2, we introduce and discuss our new blockflow algorithm. In Section 3, we introduce and discuss our protocols and applications. In Section 4, we provide a high-level description of Alephium system. We discuss future researches in Section 5.

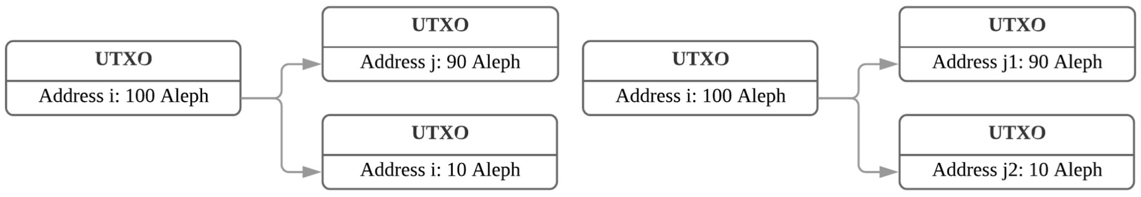
2 Blockflow

Ideally, for a currency system, we only need to care about the flow of currency among all the participants. If we use the unspent transaction output (UTXO) model like bitcoin where each transaction is a pair of inputs and outputs, then we only need to maintain the dependencies of transactions such that all the inputs and outputs match. A transaction t_1 depends on transaction t_2 if one of the inputs of t_1 depends on one of the outputs of t_2. All the transactions with such dependencies form a directed acyclic graph (DAG) with transactions as nodes and dependencies as edges. We call such graph transaction flow, inspired by network flow.

However, transaction flow is not a proper model for decentralized currency systems. The main problem is that usually there are a lot of UTXOs (unspent transaction output) in the system. Each UTXO does not have a successor in the flow graph, which would cause the UTXO to be altered by attackers easily. In blockchain technology, especially single chain approach, such transaction flow is flattened into one single chain of transactions so that every transaction would have output dependencies. While such as system is simple to implement and easy to prove safe, it suffers from the drawback of introducing too many unnecessary dependencies on transactions. It is forcing a total ordering on all transactions, while in fact they only form a partial ordering. The main drawback of totally ordered transactions is that they discourage parallel-transaction confirmation, which decreases the throughput and increases the latency of the whole system.

We propose a new consensus algorithm called *BlockFlow* to attack the scalability problem of blockchain. In our approach, all the users are divided into G separated groups. For each i, $j \in G$, there is a transaction shard which consists of all the transactions from group i to group j. Therefore, there are $G \times G$ transaction shards in total. In our cryptocurrency system, we require that each transaction shard should be constructed as one blockchain so that all the transactions in the same transaction shard would be totally ordered. Transactions in different transaction shards could be committed in parallel as long as the order does not violate all the basic dependencies in the transaction-flow graph. We use PoW to reach finality for all shards in this paper, but any other finality algorithm could be adjusted to work with BlockFlow.

Scalable UTXO model. There is an issue to apply the conventional UTXO transaction model with our transaction sharding mechanism. Let us check out the following simple example. Suppose Alice (of Address i) has a UTXO with 100 coins. She wants to send 90 coins to Bob (of Address j). This would result in two new UTXOs: one for Bob and another one for Alice containing the rest coins. If Alice is from group i and Bob is from group j, then this one transaction contains both transactions from group i to group j and from group i to group i. Therefore, we cannot put this transaction in any single transaction shard.



This problem could easily be solved with the following approaches.

- Alice could use a new address for every transaction. In our example, Alice would create a new address in group j and transfer the rest 10 coin to her new address. This is recommended in practice due to privacy and security reasons.
- If Alice makes transaction frequently, she might consider creating one address for each address group.
- Alice could split the transaction into two transactions: one changes her 100 coins into 90 coins and 10 coins, one sends 90 coins to Bob directly.

With these approaches, we could divide all transactions into $G \times G$ shards. Therefore, in our consensus algorithm, there are $G \times G$ chains in total. Each chain takes care of one transaction shard. For simplicity, let us use $\mathrm{Chain}_{i,j}$ to denote the chain consisting of transactions from G_i to G_j. $\mathrm{Chain}_{i,j}$ depends on the transactions in $\mathrm{Chain}_{k,i}$ for all $k \in G$ as a result of the UTXO model. For user in group i, it only needs to download all the data for $\mathrm{Chain}_{j,i}$ for all $j \in G$ and $\mathrm{Chain}_{i,k}$ for all $k \in G$ to know all the related transactions. That is $2G - 1$ chains for each user instead of all the $G \times G$ chains. This contributes to the scalability of our algorithm because each user only has to process part of the overall network data. Blocks in different chains could be committed in parallel to each other. This would also make mining more efficient. There are two crucial points to make our UTXO model correct.

Input/Output Dependency. Every transaction input should come from a UTXO. In our case, every transaction in $\mathrm{Chain}_{i,j}$ should depend on several UTXOs from $\mathrm{Chain}_{k,i}$ $(k \in G)$.

No Double Spending. Every UTXO should be used in at most one transaction. Especially, in our case, it's not allowed that one UTXO from group i is used in both $\mathrm{Chain}_{i,j}$ and $\mathrm{Chain}_{i,k}$.

If we could achieve these two properties for our $G \times G$ chains, then we could create an efficient and scalable ledger.

Definition. (blockflow) *A blockflow is $G \times G$ forks with one fork for each* $\mathrm{Chain}_{i,j}$ *$(i, j \in G)$ such that these forks have correct input/output dependencies and no double spending.*

We design a novel consensus algorithm for all participants in the network to reach agreement on a same blockflow eventually. In our algorithm, each block has multiple dependencies. First, like blockchain, each new block in $\mathrm{Chain}_{i,j}$ contains a block hash representing a tip of each $\mathrm{Chain}_{i,k}$ $(k \in G)$. Second, each new block in $\mathrm{Chain}_{i,j}$ needs to contain a block hash representing one tip from one of $\{\mathrm{Chain}_{k,l}\}_{l \in G}$ for all $k \in G$ and $k \neq i$. These block hashes could not be arbitrary. They must be chosen in a way to produce valid blockflow. Let us introduce some formal notations and then describe the rules. Let $H_{i,j}$ be the hash of a new block in $\mathrm{Chain}_{i,j}$. We assume $i \neq j$ in the following discussion without loss of generality (we leave the case $i = j$ to the reader). The dependencies of $H_{i,j}$ is $d(H_{i,j}) = \{H'_{i,j}\} \cup \{H_{i,r}\}_{r \neq j} \cup \{H_{k,l_k}\}_{k \neq i}$ where $H'_{i,j}$ is $H_{i,j}$'s previous block in $\mathrm{Chain}_{i,j}$. The total number of dependencies is $2G - 1$. For simplicity, we use $H_{i,j} \rightarrow H'_{i,j}$, $H_{i,j} \rightarrow H_{i,r}$ and $H_{i,j} \rightarrow H_{k,l_k}$ to denote the block dependencies. We use $H_1 < H_2$ to denote that H_2 is a hash of a descent block of the block corresponding to H_1 in the same chain, use $H_1 \leqslant H_2$ to denote that either $H_1 < H_2$ or $H_1 = H_2$. Assume that $H_{k,l_k} \rightarrow H_{k,l}$ $(k \neq i, l \neq l_k)$ by definition. Let $D(H_{i,j}) = d(H_{i,j}) \cup \{H_{k,l}\}_{k \neq i, l \neq l_k}$. $D(H_{i,j})$ contains one hash from each chain. $D(H_{i,j})$ is supposed to decide a blockflow under certain conditions. We say $D(H_1) < D(H_2)$ if for all $H_{a,b} \in D(H_1)$ and $H^*_{a,b} \in D(H_2)$ we have $H_{a,b} < H^*_{a,b}$. Let $\mathrm{Output}(H)$ be all the transaction outputs from the chain ended with block hash H. Let $\mathrm{Input}(H)$ be all the UTXOs used as transaction inputs in the chain ended with block hash H. The rules for checking $d(H_{i,j})$ are as follows.

Admissibility.

1. If $H^*_{i,j} < H_{i,j}$ and $H^*_{i,j} \rightarrow H^*_{i,r}$, then $H^*_{i,r} \leqslant H_{i,r}$.
2. If $H^*_{i,j} < H_{i,j}$ and $H^*_{i,j} \rightarrow H^*_{k,m_k}$ $(k \neq i)$, then $H^*_{k,m_k} \leqslant H_{k,m_k}$ where $H_{k,m_k} \in D(H_{i,j})$.
3. If $H_{i,r} \rightarrow H^*_{i,l}$ in $d(H_{i,r})$ for $r \neq j$, then $H^*_{i,l} \leqslant H_{i,l}$.

4. If $H_{i,r} \rightarrow H^*_{k,m_k}$ in $d(H_{i,r})$ for $r \neq j, k \neq i$, then $H^*_{k,m_k} \leqslant H_{k,m_k}$ where $H_{k,m_k} \in D(H_{i,j})$.
5. If $H_{k,l_k} \rightarrow H^*_{s,m_s}$ in $d(H_{k,l_k})$ for $k \neq i$, $s \neq k$, then $H^*_{s,m_s} \leqslant H_{s,m_s}$ where $H_{s,m_s} \in D(H_{i,j})$.

Input/output dependency. $\mathrm{Input}(H_{i,j}) \subset (\cup_{k \in G} \mathrm{Output}(H_{k,i}))$.

No double spending. $\mathrm{Input}(H_{i,j}) \cap \mathrm{Input}(H_{i,k}) = \varnothing$.

In our algorithm, every node only accepts new blocks that are validated by these rules. These rules guarantee that every valid new block implies one valid blockflow. Let us prove this important fact.

Lemma 1. *$D(H'_{i,j}) < D(H_{i,j})$.*

Proof. We only need to show that $D(H'_{i,j}) \leqslant D(H_{i,j})$ since $H'_{i,j} < H_{i,j}$. Let us prove it for three different cases.

1. If $H'_{i,j} \rightarrow H^*_{i,r}$ in $d(H'_{i,j})$, then $H^*_{i,r} \leqslant H_{i,r}$ by rule 1 of admissibility checking.
2. If $H'_{i,j} \rightarrow H^*_{k,m_k}$ in $d(H'_{i,j})$ for $k \neq i$, then $H^*_{k,m_k} \leqslant H_{k,m_k}$ by rule 2 of admissibility checking.
3. If $H^*_{k,m_k} \rightarrow H^*_{k,m}$ in $d(H^*_{k,m_k})$ for $m \neq m_k$, then by rule 1 of admissibility checking and case 2 in this proof, we have $H^*_{k,m} \leqslant H^{\times}_{k,m} \in d(H_{k,m_k})$. Again, by rule 3 of admissibility checking, we have $H^*_{k,m} \leqslant H^{\times}_{k,m} \leqslant H_{k,m}$. □

Lemma 2. *$D(H_{i,r}) < D(H_{i,j})$ for $r \neq j$.*

Proof. We only need to show that $D(H_{i,r}) \leqslant D(H_{i,j})$ since $H'_{i,r} < H_{i,r}$. Let us prove it by induction for three different cases.

1. If $H_{i,r} \rightarrow H^*_{i,l}$ in $d(H_{i,r})$, then $H^*_{i,l} \leqslant H_{i,l}$ by rule 3 of admissibility checking.
2. If $H_{i,r} \rightarrow H^*_{k,m_k}$ in $d(H_{i,r})$ for $k \neq i$, then $H^*_{k,m_k} \leqslant H_{k,m_k}$ by rule 4 of admissibility checking.
3. If $H^*_{k,m_k} \rightarrow H^*_{k,m}$ in $d(H^*_{k,m_k})$ for $m \neq m_k$, then by rule 1 of admissibility checking and case 2 in this proof, we have $H^*_{k,m} \leqslant H^{\times}_{k,m} \in d(H_{k,m_k})$. Again, by rule 3 of admissibility checking, we have $H^*_{k,m} \leqslant H^{\times}_{k,m} \leqslant H_{k,m}$. □

Lemma 3. *$D(H_{k,l}) < D(H_{i,j})$ for $k \neq i$.*

Proof. We only need to show that $D(H_{k,l_k}) < D(H_{i,j})$ since we have $D(H_{k,l}) < D(H_{k,l_k})$ for $l \neq l_k$ by Lemma 2 and induction. Again, we only need to show that $D(H_{k,l_k}) \leqslant D(H_{i,j})$ since $H'_{k,l_k} < H_{k,l_k}$. Let us prove it by induction for three different cases.

1. If $H_{k,l_k} \rightarrow H_{k,l}$ in $d(H_{k,l_k})$ for $l \neq l_k$, then $H_{k,l} \in D(H_{i,j})$ followed from the definition of $D(H_{i,j})$.
2. If $H_{k,l_k} \rightarrow H^*_{s,m_s}$ in $d(H_{k,l_k})$ for $s \neq k$, then $H^*_{s,m_s} \leqslant H_{s,m_s}$ by rule 5 of admissibility checking.
3. If $H^*_{s,m_s} \rightarrow H^*_{s,m}$ in $d(H^*_{s,m_s})$ for $m \neq m_s$, then by rule 1 of admissibility checking, we have $H^*_{s,m} \leqslant H^{\times}_{s,m} \in d(H_{s,m_s})$. Again, by rule 3 of admissibility checking, we have $H^*_{s,m} \leqslant H^{\times}_{s,m} \leqslant H_{s,m}$. □

Lemma 4. *If $H_{i,j}$ is the hash of a new valid block satisfying all the checking rules, then $D(H_{i,j})$ is a blockflow.*

Proof. We just need to show that $D(H_{i,j})$ has correct input/output dependencies and no double spending. We prove this inductively.

By Lemma 1, Lemma 2 and the induction fact that $D(H_{i,k})$ is a blockflow for $k \neq j$, we know $\mathrm{Input}(H_{i,k}) \subset (\cup_{k \in G} \mathrm{Output}(H_{k,i}))$. For $k \neq i$, by Lemma 3 and the fact that $D(H_{k,l})$ is a blockflow, we know that $\mathrm{Input}(H_{k,l}) \subset (\cup_{m \in G} \mathrm{Output}(H_{m,k}))$. Therefore, input/output dependencies are always kept.

No double spending is easy to prove. For $k \neq i$, since $D(H_{k,l_k})$ is a blockflow, we have $\text{Input}(H_{k,l_1}) \cap \text{Input}(H_{k,l_2}) = \varnothing$ for all l_1, l_2. Together with the no-double-spending check for all the inputs of group i, no double spending is kept for all address groups. □

This important lemma shows that a valid new block actually extends one previous blockflow. There are two problems left: the existence of valid new blocks following all checking rules; and how to choose the best of such blocks. The existence is easy since an empty block with trivial dependencies satisfies all the rules. The challenge is how to make the blockflow include as many blocks and valid transactions as possible. In blockchain, the system considers the best chain or best fork and always tries to extend that best chain. Similarly, in blockflow, we propose an algorithm for users to find the best blockflow and then to extend it.

Mining. Suppose a miner M wants to mine $\text{Chain}_{i,j}$. M should have an address in group j to receive coinbase rewards. By the above lemma, each latest block header in each $\text{Chain}_{i,j}$ defines a valid blockflow. In order to propose a new block, a miner needs to first find valid dependencies $d(H_{i,j})$ for $\text{Chain}_{i,j}$ and then to calculate a valid mining nonce. Since we exploit PoW for consensus, a miner wants to make its block proposed accumulates as much work as possible. In general, there are two ways to find a proper $d(H_{i,j})$: heuristic approach and optimization approach.

Let us introduce a new definition to help describe our mining algorithm. Let H_{k_1,l_1} and H_{k_2,l_2} be two block hashes. We say that blockflow BF_1 is compatible with blockflow BF_2 if $\{H^1_{m,n}\}_{n\in G} < \{H^2_{m,n}\}_{n\in G}$ or $\{H^2_{m,n}\}_{n\in G} < \{H^1_{m,n}\}_{n\in G}$ for all $m \in G$ where $H^1_{m,n} \in BF_1$ and $H^2_{m,n} \in BF_2$. If blockflow BF_1 is compatible with BF_2, we can define the union of two blockflows $BF_1 \cup BF_2$ be the union of each $\text{Chain}_{i,j}$. This concept of compatibility is useful because of the following lemma.

Lemma 5. *If blockflow BF_1 is compatible with blockflow BF_2, then $BF_1 \cup BF_2$ is a blockflow.*

Proof. It's easy to check the input/output dependencies and no double spending for $BF_1 \cup BF_2$, since $BF_1 \cup BF_2$ is the union of each chain with the proper precondition. □

One simple heuristic approach for resolving dependencies works iteratively as follows. Miner M first selects the best blockflow $D(H_{k_1,l_1})$ so far. Based on this initial blockflow, M then try to find compatible blockflows $D(H_{k',l'})$ for other user group $k' \neq k_1$. M also needs to find a compatible dependencies $H_{i,r}$ for each r. The miner expands the initial blockflow step by step. The final blockflow BF is the union of all these compatible blockflows. If the new block to be mined is of hash $H_{i,j}$, then BF uniquely defines $d(H_{i,j})$. It's very easy to show that $d(H_{i,j})$ satisfies all the checking rules using Lemma 5. After M gets BF, it could start to collect valid transactions for BF. M puts all valid transactions together to create a new block template for mining. This heuristic approach could be improved further, e.g. $\{H_{i,r}\}_{r\in G}$ could be resolved more aggressively using checking rules. We do not want to go depth here so as to make the content more compact. The optimization approach for resolving dependencies is by modeling the dependencies as an optimization problem. Then one could use integer programming for example to calculate the best dependencies with best-accumulated PoW work. In general, we think that the heuristic approach is faster and usually optimal when there are not so many chain forks.

This algorithm is similar to blockchain mining, each miner tries to extend the best blockflow. The crucial difference is that our algorithm could confirm $G \times G$ blocks in one new block. Miners could mine and commit its block in parallel. Since our algorithm always extends the best blockflow, it reaches consensus eventually as do blockchain. Once a set of valid dependencies are computed, it determines tips for each chain. The mining target would be adjusted based on all the latest blocks decided by the dependencies from all chains.

In practice, a node for group i could compute the same $d(H_{i,j})$ for different group j, since $\text{Chain}_{i,j}$ $(j \in G)$ shares the same dependencies structure. Further, the node for group i could maintain the state of best blockflow and dependencies $D(H_{i,j})$, and update them whenever it receives new blocks and headers. In our mining algorithm, we also encode the group information of each block into its hash, which brings optimizations for indexing in our reference implementation. We could also use digests of hashes for cross-chain dependencies to save some storage space.

Our concrete hash function for mining algorithm would work more or less like this. The algorithm first computes the hash string H_1 of the blocks with a random nonce, then it uses H_1 as a random number to get a random block hash H_2 from history. In the last step, it uses H_2 as input to compute a final hash string. With this mining function, miners have to access CPU, memory, and even disk. This is our current design for hash function, but we might change it if we find a better alternative.

Analysis. With our new BlockFlow algorithm, the TPS could be improved dramatically. Let's analysis TPS in a semi-formal way here. Let P be the value of TPS for each chain, S be the average size in KB per transaction, D be the propagation delay of one block, T be the average mining time, the orphan rate R could be roughly estimated as D/T. Let B be the average network bandwidth of nodes inside our network, we have the following constraints:

$$\begin{aligned} \frac{D}{T} &\leqslant R \\ (2G-1)\times P\times S\times\left(1+\frac{D}{T}\right) &\leqslant B \end{aligned}$$

On the other hand, we could estimate $\frac{D}{T}$ based on empirical observations. We could assume that the propagation delay per KB is d when block size is not too large. Then we have $D = T \times P \times S \times d$ and $\frac{D}{T} = P \times S \times d$. Therefore,

$$\begin{aligned} P\times S\times d &\leqslant R \\ (2G-1)\times P\times S\times(1+P\times S\times d) &\leqslant B \end{aligned}$$

If we take G to be 32, R to be 6%, B to be 10Mb, S to be 0.5KB, d to be 8ms (1.5MB broadcast in 12s), then P could be 15. The final TPS is $P \times G \times G = 15360$ which is greater than 10000.

As for security, our algorithm suffers from 51% attack instead of 1% attack. Thanks to the sophisticated block dependencies, if one wants to attack one chain, it has to attack all the other chains as well. All the proof-of-work are accumulated into new blocks, so we don't diverse the mining work.

3 Protocols and Applications

We believe that in general it's impossible to scale smart contract to work efficiently with sharding algorithms. However, it's possible to scale the most useful functionalities while keeping the other functionalities still available in a non-scaling setup. In Alephium, we decompose smart contract into token part and data part. Both tokens and data are easily scalable, but the cost is that we lose some generality. However, we plan to support smart contracts in specific shards for the cases where developers really need powerful contracts (e.g. turning complete contracts). We will give a high-level description of our protocols in plain English. It's in no way to be detailed documentation.

3.1 Token Protocol

In our protocol, we support the creation and removal of customized tokens. As we have seen in recent years, tokens are crucial for the adoption of blockchain protocols. In our design, we want to make token creation as easy as possible, but at the same time, we don't want people to abuse token creation leaving many null coins in the system. For each token creation, the creator needs to burn a specific number of coins. Also, every customized token has an expiry date which depends on how many coins were burnt during the creation, in order to further reduce storage and computation cost. We category tokens into several classes: convertible token, inconvertible token, and mixed token.

Inconvertible token. An inconvertible token is just a simple digital token got created. It would have very simple meta info.

Convertible token. A convertible token is used for the cases where the token created could be converted to Aleph (built-in coin) automatically.

Mixed token. A mixed token is a combination of several convertible tokens and inconvertible tokens. The converting ratio could be revealed lazily.

In order to create a convertible token, the creator needs to deposit twice amount of the converted token value as reserves. For example, if one wants to create a new token which would be converted to 1000 Aleph in the future, one needs to deposit 2000 Aleph in the beginning. If users buy out this new token, the creator still has 1000 Aleph deposit in the system. If the creator does not operate properly, he will lose all the reserves and all the token buyers would get refunded. With this stake-based mechanism, the creator would be more likely to operate this token correctly. For a mixed token, as there might be several convertible tokens got created, we enforce that the converting ratios would sum up to one so that no Aleph coin would be created from air. We design a token-level script language for the creation and removal of new tokens. Once a token is created, it will follow the UTXO model, and users could exchange them. Combining our data model with our token-level script language, we could build almost all practical applications.

3.2 Data Protocol

Most of the modern blockchain protocols have a built-in virtual machine for general computing. With such VMs, blockchains can implement useful and complicated decentralized applications. However, such generality also comes with a high cost. All the nodes in the network have to verify all the computation, even though the computation might not be relevant to most of the nodes. It's also very challenging to scale such virtual machines for sharding algorithms.

In Alephium, we shift from this code-centric approach to a data-centric approach. That is, users only submit data to our blockflow network and get data stored decentralized. The data submitted could be raw digital data, data digest, or code. Most of the nodes in the network do not need to know the real meaning of the data. Only the users that are related to the data could use specific applications to play with the specific data. By this design, we could achieve both decentralization and computing scalability. In some sense, our philosophy is very similar to the philosophy of TCP/IP protocol. We provide a very low-level infrastructure for decentralizing data. The application layer would take care of the usage of data. The wisdom of Lisp indicates that data is the same as program, so we don't lose any generality. On the other hand, using data instead of program allows high-level applications to design and upgrade their own domain-specific languages, which could reduce the security risks of our system. The other benefit of this data-centric approach is that it could be easily integrated with UTXO model. We believe that developers could build efficient and scalable DAPPs by combining our token protocol and data protocol in a smart way.

3.3 Smart Contract

Note that in our sharding algorithm we have G specific shards which are intra-group shards consisting of transactions from group i to group i. Each of these intra-group shards is no different from a traditional blockchain. We plan to design a VM language for these specific shards so that we could build complicated applications. Since we have G such special shards, our system could support more smart contracts compared to a single blockchain. Combining smart contract with our scalable token protocol, developers could build efficient and powerful dApps.

3.4 Features

With our new consensus algorithm and protocols, we could achieve scalability for both data and computation. We introduce several useful features in this section to make our system even more practical and sustainable.

Mining Rewards Lockup In Alephium, miners could choose optionally to lockup their coinbase rewards for a certain period. There will be some base coinbase rewards for each newly mined block. Furthermore, there will be extra rewards that are proportional to the lockup period. We want to incentive miners to keep their mining rewards for long term interests instead of cashing out immediately.

Multisignature For all the data submitted and token exchanges in our protocol, there could be multiple owners and multiple signatures. It could be used for security reason, but it could also be used to represent ownership. Although each chain involves at most two user groups to participate in, it's not really a problem to support multiple signatures in general. Users could create new accounts in specific address group easily.

Expiry Date All the data and customized token have an expiry date which could be represented as timestamp or block heights. If data got expired, then the network does not need to store it anymore and could remove it from history. If a customized token expired, it would either get discarded if it's not convertible or got converted automatically otherwise. Expiration date provides limited functionality of garbage collection.

Lock Time An UTXO could be locked by setting a lock time for it. We make lock time as a built-in feature for UTXOs because it's useful in many real scenarios.

Pricing Model The whole PoW network is a game between users, miners, and protocol. In order to keep this game in a good shape, it's crucial to design a reasonable economic model for it. In our case, it's about how to price transactions and data. It's relatively simple, as we don't need to price computation. Therefore, we could consider simply the size and the expiry date of the transactions and data submitted. For transactions, we also price each input based on how long the tokens/coins have been stored in order to reduce the number of small UTXOs.

3.5 Applications

All the classical dApps in Bitcoin and Ethereum platforms could be migrated to Alephium platform without major changes given our native cross-shard transaction support. Besides that, we could enable a new type of high-throughput applications. The workflow of this type of application using our platform would be like this:

1. The developers develop a decentralized application (usually should be open sourced) and publish it on the internet. The project owner creates a token for the project and publishes the digital digest of the application to Alephium platform (We might implement a decentralized application market for Alephium).
2. Users download the application from internet and verify the digest of the application. Then users run this application on the top of Alephium platform.
3. All the data and transactions generated by this application would be stored in the P2P network. **Nodes that do not use this application would not need to verify the content of the data.**

In this approach, the application is still reversion resistant with a single unchanging application history, but the applications will not use the network to do all the computation.

In some scenarios, the application could be semi-decentralized if decentralization could be traded off for performance. For example, the project could have a centralized service to interact with all its users. However, all the transactions the application made would be published to Alephium network, so that all the users could check by downloading all the relevant application data.

4 System

In this section, we give a high-level description of some of the key parts of our system. We will not go deep into details which should be covered in the reference implementation.

4.1 Network

The Alephium network consists of nodes for different address groups. Nodes from different groups need to be connected so that blocks for $\mathrm{Chain}_{i,j}$ would be propagated to both group i and group j. We design a variant of Kademlia protocol for network discovery. Each node keeps a local table of the currently active nodes in the network along with necessary meta information (e.g. group id). Nodes in the network could update their table regularly by scanning the network. The communication protocol we use is very similar to the Kademlia protocol. The main difference is that we use hamming distance instead of xor metric to computing the distance of two nodes. The reason we made this change is that with xor metric nodes with addresses starting with 0 are less likely to be connected to nodes with address starting with 1.

4.2 Nodes

With blockflow algorithm, each node stores only part of the transaction/data history. For the sake of reliability and security, a "full node" in our network would run a cluster of G nodes with one node for each address group. All the other services of this cluster could access these G nodes through RPC calls. When initializing connection with other nodes or clusters, this cluster will send its network topology so that the other nodes could propagate blocks more efficiently.

In practice, a light client that handles transaction/data only related to a very small number of addresses is very useful. In our network, we could implement light client similar to Bitcoin by using Bloom filters. Since a light client actually downloads history for very few addresses, it could deal with transactions/data from different groups in a single node, unlike full nodes.

Note that we could start with a small G when there are not enough participants and full nodes. The system would gradually increase G along with time. In practice, we could set up G to be high in the beginning, but the number of groups that are actually used internally would upgrade from small to larger.

4.3 Application Layer

The core protocol of our network provides a very low-level infrastructure, not suited for high-level applications design. To bridge the low-level part and high-level application, our system provides official APIs for accessing transactions and data. We also plan to provide an application market for developers to publish their applications. As we mentioned early that users only need to use high-level applications that are related to them. Application market would provide a convenient way for users to fetch useful applications. This application market could store useful meta information about applications in a decentralized way using Kademlia protocol. For the application itself, it could be distributed either in a decentralized or centralized fashion depending on the requirements.

5 Further research

In the following, we describe several research directions that would help to improve our system.

Privacy For our data protocol, it would be relatively easy to enforce privacy by applications. However, in our current design, all the transactions are always transparent which might cause potential problems for some business.

Stateless client Every node in our system stores the entire UTXO set for verifying the incoming transactions like Bitcoin. There are potential ways to alleviate the need for such full state. The stateless client concept uses an additional data structure (e.g. Merkle tree, dynamic accumulators) for transaction membership proofs. The network updates and maintains the proofs for new blocks. Nodes in the network could validate a transaction without the entire UTXO set. Existing stateless technologies are not practical enough for our system by the time of writing.

PoW Improvements/Alternatives Though the robustness of PoW has been proved by time, it's heavily criticized due to its energy consumption. However, by the time of writing, we did not really find mature PoW alternatives for our platform. Further researches might alleviate the energy consumption issue of PoW. As mentioned early, our algorithm actually could work with any finality algorithm for blockchain. If there is any PoW alternative suitable for our algorithm in the future, it's not difficult to replace PoW in our protocol.

6 Acknowledgment

The author thanks Aloïs Cochard for reviewing the draft version of this paper.

Bibliography

[1] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *White paper*, 2014.
[2] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system. 2008.
[3] S Popov. The tangle, iota whitepaper. 2018.