CodeChain: an end-to-end asset tokenization system

Kwang Yul Seo
<kseo@kodebox.io>
Dec 26, 2018
Version 0.1.0

Abstract

CodeChain is a blockchain platform specialized for tokenized assets. Users can issue tokens and trade them in a regulatory compliant way. CodeChain also provides the Asset Exchange Protocol, which facilitates low friction peer-to-peer exchange of assets on the CodeChain network. The CodeChain protocol drives interoperability among different players in the asset tokenization ecosystem.

1. Introduction

Blockchain has the power to eliminate excessive fees and time delays associated with traditional long-distance transactions carried out by multiple middlemen. It opens an era of real-time cross-border trade settlements, immediate deal execution and cheaper automated services.

Tokenization introduces trustless and secure record keeping and brings full or fractional ownership of real-world assets and all accompanying rights onto the digital world. Blockchain is becoming the engine for bringing traditional finance into the digital age, and for disrupting asset markets, such as real estate, equity, debt, commodities, energy and derivatives. The potential global market is tremendous, and can possibly reach up to a quadrillion US dollars.

CodeChain is an open-source blockchain platform optimized for asset tokenization. It is equipped with a built-in multi-asset management solution, which enables users to issue, transfer and manage digital securities backed by real-world assets such as real-estate, alternative assets, commodity and equity tokens.

2. Existing Work

There have been several projects that have added tokens to the Bitcoin network[1]. The first project was Mastercoin by JR Willet, followed by Counterparty and other projects. The term “Colored Coin” describes a class of methods for representing and managing real world assets on top of the Bitcoin network.

Embedding assets in the Bitcoin network is advantageous for high security because the Bitcoin network has a tremendous amount of hash power. This makes it practically impossible to re-write, or modify the ledger without huge investments in mining power.

However, Bitcoin is not the best solution for representing real world assets because transaction confirmations take upto an hour due to its proof of work consensus algorithm. Bitcoin also lacks the compliance layer required for transferring assets in a regulatory compliant manner.

CodeChain is inspired by Colored Coin, but has overcome the limitations of the Bitcoin network by taking the requirements of security tokens into consideration when designing the protocol.

3. Asset

Assets in the CodeChain protocol represent real-world physical or digital assets in custody. A few examples are:

- Stock or shares of a company
- Fractional ownership of fine art
- Gold bars
- Physical dollars
- Energy credits (Electricity, Wood, Gas, Oil, Wind)
- Land Deeds

Or they represent virtual goods such as:

- Gifts
- Collectibles
- Airline miles
- Reward points

- Game items

4. Asset Protocol

CodeChain is a blockchain platform built on the UTXO model of Bitcoin. In the UTXO model, each transaction spends output from prior transactions and generates new outputs that can be spent by future transactions. The benefits of the UTXO model are:

1. **Scalability**: it is possible to process multiple UTXOs at the same time. Because CodeChain aims to provide a scalable blockchain solution to multi-asset applications such as security tokens, the scalability benefits of the UTXO model become very important.
2. **Privacy**: UTXO provides a higher level of privacy if a user uses new addresses for each transaction.

4.1. Asset Issuance

Assets are tokens that can be issued by users of the CodeChain protocol. Users create an asset by specifying the metadata, approver and administrator.

- *metadata* is an arbitrary string where an issuer can specify the name and description of the asset being issued.
- *approver* is an optional field, which specifies the address of the agent who grants the transfer transactions. The "Compliance" section explains more about this.
- Additionally, if the issuer specifies him or herself as an *administrator*, he/she can have additional privileges that are necessary to manage assets such as banning users and freezing accounts.

Name	Data Type	Description
metadata	String	The name and description of the asset
approver	Option<Address>	If specified, the approver must grant the asset transfer transaction
administrator	Option<Address>	If specified, the administrator can change the asset scheme.
output	AssetMintOutput	The amount and recipient of the asset issued

<Table 1. Key fields of AssetMintTransaction>

output field specifies the initial owner and the amount of the asset issued. The issuer can lock the asset however he/she wants by using *lock_script_hash* and *parameters*.

Name	Data Type	Description
lock_script_hash	H160	The lock script hash used to lock the asset
parameters	Vec<Bytes>	The parameters to the lock script
amount	Option<u64>	The amount of asset

<Table 2. Key fields of AssetMintOutput>

4.2. Asset Transfer

Once assets have been successfully issued, they can now be sent to other users. For instance, let's say that the initial owner of a newly issued asset is Alice. If Alice wants to send some tokens to Bob, then a transaction must be created. This transaction of sending tokens from one user to another is called the *AssetTransferTransaction*.

Name	Data Type	Description
inputs	Vec<AssertTransferInput>	The list of transaction inputs
outputs	Vec<AssetTransferOutput>	The list of transaction outputs
orders	Vec<OrderOnTransfer>	See "Asset Exchange Protocol" section
approvals	Vec<Signature>	The list of signatures gathered from approvers

<Table 3. Key fields of AssetTrasferTransaction>

The transaction format is similar to Bitcoin except that CodeChain supports multiple asset types while Bitcoin supports only one asset type, BTC. The *asset_type* field in *AssetTransferOutput* is the cryptographic hash of *AssetMintTransaction,* which uniquely identifies the asset type issued on the CodeChain network.

Assets that were created without the appointment of an approver are freely transferable to any CodeChain address. Assets with an approver need to gather the approvers' signatures

before being transferred. The *approvals* field contains the signatures of approvers. Refer to the "Compliance" section to see how approvals work.

Name	Data Type	Description
prev_out	AssetOutPoint	The pointer to the previous asset transaction output
lock_script	Bytes	The lock script used to lock the asset
unlock_script	Bytes	The unlock script that will be used to unlock the asset

<Table 4. Key fields of AssetTransferInput>

Name	Data Type	Description
lock_script_hash	H160	The hash of the lock script used to lock the asset
parameters	Vec<Bytes>	The parameters to the lock script
asset_type	H256	The hash of the *AssetMintTransaction* that uniquely identifies the asset type
amount	u64	The amount of asset

<Table 5. Key fields of AssetTransferOutput>

Name	Data Type	Description
transaction_hash	H160	A 20-byte transaction id
index	usize	An output index number
asset_type	H256	The hash of the *AssetMintTransaction* that uniquely identifies the asset type
amount	u64	The amount of asset

<Table6. Key fIelds of AssetOutPoint>

4.3. Code Execution

Codechain saves the unlock condition for each asset, and anyone who can pass this condition is authorized to use that asset. The unlock condition is represented in a byte array, and decoded as a list of instructions before execution. The script language is intentionally designed to not be Turing-complete, and all scripts are ensured to be finished in a finite time.

Assets do not hold a full script describing its unlock condition, but only a hash of it. We refer to this script as the 'lock script', and the hash as the 'script hash'. CodeChain stores only the script hash in the output to protect privacy, as it helps hiding the owner of the asset until that asset is unlocked.

Before a lock script is executed, some predefined values are inserted to stack. These values are called 'parameters', and saved along with the script hash. To consume an asset, a user must provide both the lock script and the unlock script.

Script is encoded as a byte string, and each byte of this string is referred to as a script byte. Instructions are execution units of VM and can be composed of multiple script bytes. The frontmost byte of the instruction is the identifier of instruction and is called 'opcode'.

The overall execution process is similar to P2SH in Bitcoin. The detailed execution process is as follows:

1. Check if an asset's script hash is equal to the hash of the provided lock script.
2. Decode the lock script and unlock script into a list of instructions.
3. Check if the unlock script is valid. Currently, it's considered invalid if any opcode other than PUSH-related codes is included.
4. Insert the provided parameters into the stack. The order of insertion must be last to first, so that the first parameter appears at the top of the stack.
5. Execute the unlock script, and then the lock script.

If an exception occurs during the procedure described above, the transaction will be marked as failed. The result is SUCCESS when all of the following conditions are met:

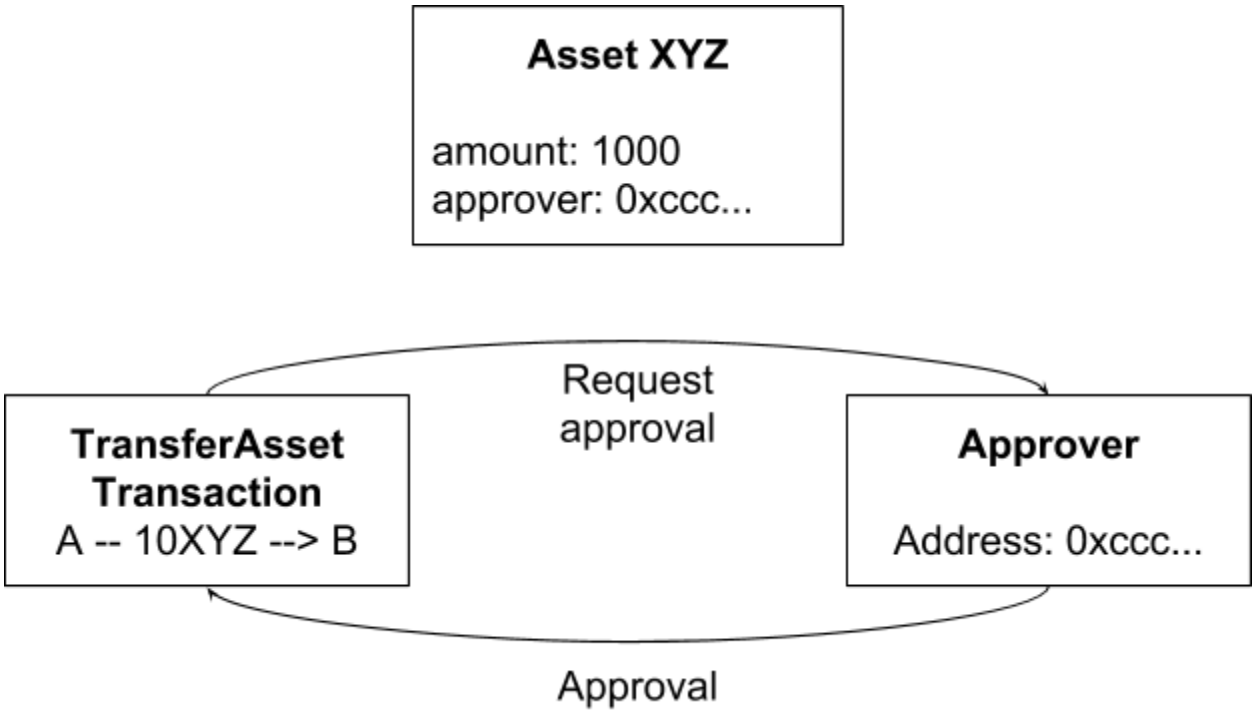
- There are no more instructions to execute
- Stack has only one item
- Stack's topmost value is not zero when converted into an integer

The result is BURNT when a self-burning instruction was executed. Otherwise, the result is FAIL and the asset is not unlocked.

5. Compliance

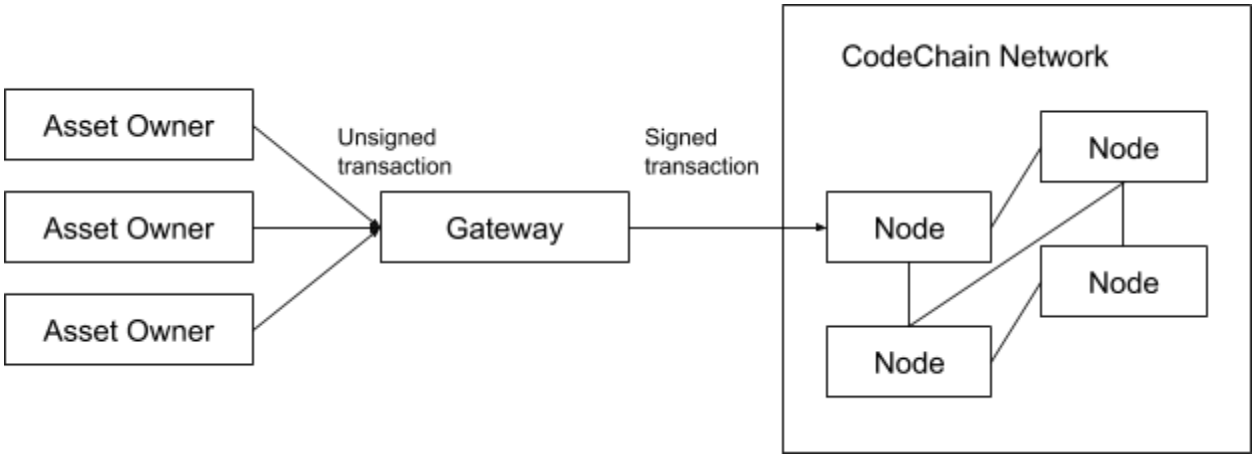
Security tokens must be compliant to the laws of their respective jurisdiction. In contrast to public platforms such as Bitcoin, which emphasize censorship resistance, security tokens must only be tradeable by investors who passed the KYC/AML requirements.

When issuing an asset on CodeChain, one can optionally specify the approver who grants the transfer of assets. The approver decides whether transactions meet the requirements of compliance or not. Every asset transfer transaction needs approval from the specified approver. Otherwise, the transaction is rejected by validators.



6. Transaction Fee

Ethereum[2] is notorious for bad user experience because end users must pay the transaction fee whenever they interact with the network. CodeChain solves the usability problem by making it possible for service providers to pay the transaction fee in lieu of their users.



A service provider runs a gateway, which is responsible for gathering transactions and paying the transaction fee by cryptographically signing them. The fees are deducted from the service provider's account. The service provider's account must have sufficient CCC(CodeChain Coin).

7. Asset Exchange Protocol

CodeChain supports an on-chain order to support DEX (Decentralized Exchange). CodeChain Asset Exchange Protocol is inspired by 0x protocol[3].

7.1. Order

An order can be either a point-to-point order between a maker and a taker, or a broadcast order between a maker, a relayer and takers. Point-to-point orders allow two parties (makers and takers) to directly exchange assets between each other. Broadcast orders are similar to point-to-point orders, but also allow relayers to take fee assets from makers and takers. Broadcast orders are usually used in decentralized exchange platforms.

Orders on CodeChain are implemented on UTXO forms. Basically, an order is like a tag on products in grocery stores. Orders can be put on inputs and outputs of transfer transactions. If an input/output has an order, it means that that specific input/output should be exchanged as its order says. Think about this situation: Let's say we have an order to exchange 10 gold to 100 silver, and we've put that order on a 10-gold input. Then there should be a 100-silver output with the same order of 10 gold to 100 silver. If there isn't enough gold, there would perhaps be a 5-gold output and a 50-silver output, both with the same order, or, whatever outputs with the same order while maintaining the order equilibrium.

Assets with orders must be able to be spent by takers or relayers without any permission. But in the CodeChain UTXO form, those parties should provide a lock script and an unlock script to prove the ownership of the assets. In order to solve the problem, if there are orders on inputs, CodeChain runs VM on orders, not transactions, if there are orders on inputs. If there are no orders on inputs, CodeChain runs VM on transactions as usual. Partial signing is not allowed on transactions with orders. With this convention, takers and relayers can take the ownership of the assets with orders with unlock scripts, which are provided by makers.

7.2. Order format

The format of *Order* is as shown below.

Name	Data Type	Description
assetTypeFrom	H256	The type of the asset offered by maker
assetTypeTo	H256	The type of the asset requested by maker
assetTypeFee	H256	The type of the asset offered by maker to give as fees
assetAmountFrom	U64	Total amount of assets with the type assetTypeFrom
assetAmountTo	U64	Total amount of assets with the type assetTypeTo
assetAmountFee	U64	Total amount of assets with the type assetTypeFee
originOutputs	AssetOutPoint[]	The previous outputs composed of

		assetTypeFrom / assetTypeFee assets, which the order starts from
expiration	U64	Time at which the order expires
lockScriptHashFrom	H160	Lock script hash provided by maker, which should be written in every output with the order except fee
parametersFrom	Bytes[]	Parameters provided by maker, which should be written in every output with the order except fee
lockScriptHashFee	H160	Lock script hash provided by relayer, which should be written in every fee output with the order
parametersFee	Bytes[]	Parameters provided by relayer, which should be written in every fee output with the order

To make a point-to-point order, put a zero on the *assetAmountFee* field. To write an order on a transfer transaction, the order should be wrapped once more, as *OrderOnTransfer*.

7.3. OrderOnTransfer format

If there are inputs and outputs with the same order, it is wasteful to put the order in every input/output. Therefore, orders are wrapped into *OrderOnTransfer*.

Name	Data Type	Description
order	Order	The order to write on the transfer transaction
spentAmount	U64	The spent amount of assetTypeFrom of the order in the transfer transaction
inputIndices	Index[]	The indices of the transfer inputs which are protected by the order
outputIndices	Index[]	The indices of the transfer outputs which are protected by the order

7.4. How to support partial fills?

As described above in the 10-gold-to-100-silver order, it is possible to make a transfer transaction which has a 10-gold input that results in a 5-gold and 50-silver output, tagged with the same order. (Other inputs/outputs should be provided by a taker or a relayer). After this transaction, an asset contains the hash of the consumed order, which is 5-gold-to-50-silver order, not 10-gold-to-100-silver order. In order to use the 5-gold output, provide the 5-gold-to-50-silver order with the same information except for the *assetAmountFrom* and the *assetAmountTo* field. Neither a lock script nor an unlock script is needed for the 5-gold input. CodeChain will compare the order of an input and the order of the corresponding previous output, and will not run VM on the order if those orders are identical.

8. Token Model

CodeChain protocol has two tokens: fee token and stake token. The total supply of each token is as follows:

	Stake token	Fee token
Total supply	10,000,000	100,000,000,000

Stake token holders are the owners of the network and have the following rights:

1. The right to participate in the network as a validator
2. The right to earn transaction fee paid to the network

Fee tokens are required in order to use the CodeChain network and each transaction type has varying costs. A transaction which consumes more CPU and storage generally costs more. Refer to Appendix: Storage Cost for the detailed analysis of storage costs of each transaction.

Fee tokens earned by the network are redistributed to the token holders on a pro-rata basis. For example, assuming Kodebox holds 80% of the total stake tokens and 1000 fee tokens are paid, Kodebox earns 800 fee tokens.

9. Conclusion

CodeChain is a blockchain platform that is specialized for tokenized assets. It provides a base layer, where the security token ecosystem can grow. Security token issuance platforms, security token exchanges and liquidity providers can build their services and solutions on top of the CodeChain network and give benefits to one another. The protocol is intended to serve as an open standard and common building block, promoting interoperability among various players in the ecosystem.

References

[1] Bitcoin: A Peer-to-Peer Electronic Cash System
https://bitcoin.org/bitcoin.pdf

[2] Ethereum White Paper
https://github.com/ethereum/wiki/wiki/White-Paper

[3] 0x protocol
https://0x.org/whitepaper/0x_white_paper.pdf

10. Appendix

10.1. Terminology

CCC(CodeChain Coin): Fee tokens used for paying transaction fees.

P2SH: Pay to script hash is an advanced type of transaction used in Bitcoin. Unlike P2PKH, it allows sender to commit funds to a hash of an arbitrary valid script.

UTXO: An Unspent Transaction Output (UTXO) that can be spent as an input in a new transaction.

10.2. Storage Cost

Prefix	Data	Storage
C	Account	80
R	RegularAccount	64
M	Metadata	2
T	Text	X
H	Shard	72 (+X)
A	Asset	40
S	AssetScheme	48 (+X)

<Unit Cost>

	C	R	M	T	H	A	S	Storage Increment
AssetMint	-1 +1				-1 +1	+1	+1	80 + 40 + 48 + 72 + X = 240 + X
AssetTransfer	-1 +1				-1 +1	-N +M		80 + 40 * M + 72 = 152 + 40 * M
AssetSchemeChange	-1 +1				-1 +1		-1 +1	80 + 48 + X + 72 + X = 200 + X
AssetCompose	-1 +1				-1 +1	-N +1	+1	80 + 40 + 48 + X + 72 + X = 240 + X
AssetDecompose	-1 +1				-1 +1	-1 +N	-1	80 + 40 * N + 72 = 152 + 40 * N
UnwrapCCC	-1 +1				-1 +1	-1		80 + 40 = 120
WrapCCC	-1 +1				-1 +1	+1		80 + 40 = 120
Payment	-1 +2							2* 80 = 160
SetRegularKey	-1 +1	-1 +1						80 + 64 = 144
CreateShard	-1 +1		-1 +1		+1			80
SetShardOwners	-1 +1				-1 +1			80 + 72 + X (at least 160) = 152 + X
SetShardUsers	-1 +1				-1 +1			80 + 72 + X (at least 160) =

								152 + X
Store	-1 +1			+1				80 + X (at most 256)
Remove	-1 +1			-1				80

<Transaction Cost>

Relative storage costs of transactions are as follows:

1. SetShardOwners = SetShardUsers > Store > Payment > SetRegularKey > CreateShard = Remove
2. Mint = Compose > SchemeChange > UnwrapCCC
3. Transfer > Decompose > UnwrapCCC