NeoContract White Paper

1. Preface

Smart contracts refer to any computer program which can automatically execute the terms of its preprogrammed contract. The idea of smart contract was first proposed by the cryptographer Nick Szabo in 1994, making it as old as the Internet itself. Due to the lack of a reliable execution environment, smart contracts have not been widely used.

In 2008, a man under the name of Satoshi Nakamoto released Bitcoin, and outlined the foundational concepts of a blockchain. Within the Bitcoin blockchain, Nakamoto uses a set of scripting languages to help users gain more flexibility in controlling their personal accounts and the transfer process, which eventually became the embryonic form of a chain-based, smart contract system.

In 2014, a teenager called Vitalik Buterin released Ethereum, which provides a chain-based, Turing-complete, smart contract system that can be used to create a variety of decentralized blockchain applications.

NEO blockchain is a digital asset and application platform, which provides a new smart contract system, NeoContract. At the core of the Neo platform, the network provides multiples functions such as digital asset capabilities, NeoAsset, and digital identity, NeoID, allowing users to easily engage in digital businesses, and are no longer limited to just the issuance of native tokens on the blockchain.

This article will introduce features of NeoContract and explore non-technical details. Please refer to the technical documentation for technical details: docs.neo.org.

2. Features

2.1 Certainty

If a program is running on different computers, or at different times on the same computer, the behavior of the program is deterministic if the same input is guaranteed to produce the same output, and vice versa.

Blockchain is a multi-party storage, and computational method, where the data within this distributed system is the result of reliable calculations, that cannot be tampered with. Smart contracts operate within the multi-node, distributed blockchain network. If a smart contract is non-deterministic, the results of different nodes may be inconsistent. As a result, consensus between the nodes cannot be reached, and the network becomes stagnant. Therefore, in the design of a smart contract system, there is a need to rule out any factors which may lead to non-deterministic behavior.

2.1.1 Time

Obtaining system-time is a very common system function, that may be heavily applied in certain time-sensitive contract procedures. However, obtaining system-time is a non-deterministic system function, and it is difficult to obtain a unified, precise time in a distributed system, as the results of different nodes will be inconsistent. NeoContract provides a block-based system-call that treats the entire blockchain, as a timestamp server, and obtains the timestamp whenever a new block is generated. On average, NEO network generates a new block every 15 seconds, so the contract runs at approximately the same time as the latest block-time, plus-minus 15 seconds.

2.1.2 Randomness

Many smart contract programs, such as gambling contracts and small games, use random number functions. However, random number functions are a typical non-deterministic function, and each system-call will obtain different results. In a distributed system, there are many ways to solve this problem: Firstly, the same random seed can be used for all nodes, so that the return sequence of the entire random function is deterministic, but this method exposes the entire random result in advance, greatly reducing the practical value of the random number. Another possible solution, is to let all nodes communicate in a collaborative way to generate random numbers. This can be achieved by using cryptographic techniques to produce a fair random number, but the disadvantage lies in the very poor performance, and need for additional communication overhead. A centralized random number provider can be used to generate random numbers which guarantees consistency and performance, but the drawback of this approach is obvious; Users will have to unconditionally trust the centralized number provider.

There are two ways to generate a random number in NEO:

1. When each block is generated, the consensus node will reach a consensus on a random number, and fill it into the Nonce field of the new block. The contract program can easily obtain the random number of any block, by referencing the Nonce field

2. The contract program can use the hash value of the block as a random number generator, because the block hash value has certain inherent randomness. This method can be used to obtain a weak random number

2.1.3 Data Source

If a program obtains data at run-time, it might become a non-deterministic program if the data source provides non-deterministic data. For example, using different search engines to obtain the top 10 search results for a particular keyword, may yield different results, in various sort orders, if different IP addresses are used.

For smart contracts, NEO provides two types of deterministic data sources:

1. **Blockchain Ledger**

   The contract procedure can access all data on the entire chain through interoperable services, including complete blocks and transactions, and each of their fields. The data on the blocks are deterministic and consistent, so they can be securely accessed by smart contracts.

2. **Contract Storage Space**

   Each contract deployed on the NEO network, has a private storage area that can only be accessed by the contract itself. NEO consensus mechanism ensures consistency of the storage status, of each node in the network.

For situations where access to non-blockchain data is required, NEO does not provide a direct way to interact with these data. Non-blockchain data will have to be transferred to the NEO blockchain using transactions, and subsequently translated into the either of the aforementioned data sources, in order to become accessible by the smart contracts.

2.1.4 Contract Call

Smart contracts in NeoContract can call each other, but not be recursively called. Recursion can be achieved within the contract, but it cannot cross the boundaries of the current contract. In addition, the call relationship between contracts must be static: The target cannot be specified at runtime. This allows the behavior of the program to be fully determined before execution, and its call relationship to be fully defined before it can run. Based on this, multiple contracts can be dynamically partitioned to achieve parallel execution.

2.2 High performance

The execution environment of a smart contract plays an integral role in its performance. When we analyze the performance of any execution environment, there are main two indicators which are critical:

1. Execution speed of the instruction
2. Startup speed of the execution environment itself

For smart contracts, the execution environment is often more important than the speed of execution of the instruction. Smart contracts are more involved in IO operation of the logic, to determine the instructions, where the implementation of these instructions can easily be optimized. Every time the smart contract is called, it must start up a new virtual machine / container. Therefore, the execution speed of the environment itself (starting a virtual machine / container) has greater impact on the performance of the smart contract system.

NEO uses a lightweight NeoVM (NEO Virtual Machine) as its smart contract execution environment, which has a very fast start up and takes up very little resources, perfect for short programs like smart contracts. Using the compilation and caching of hotspot smart contracts with JIT (real-time compiler) can significantly improve the efficiency of virtual machines.

2.3 Scalability

2.3.1 High concurrency and dynamic partitioning

When discussing the scalability of a system, it involves two main areas: Vertical scaling and horizontal scaling. Vertical scaling refers to the optimization of the processing workflow, allowing the system to take full advantage of existing equipment capacity. With this approach, limits of the system are easily reached, as series-based processing capacity is based on the hardware limit of a single device. When we need to scale the system, is there a way to transform the series system into a parallel system? Theoretically, we will only need to increase the number of devices, and we will be able to achieve almost unlimited scalability. Could we possibly achieve unlimited scaling in distributed blockchain networks? In other words, can the blockchain execute programs in parallel?

The blockchain is a distributed ledger, that records a variety of state data and the rules governing the changes in state of these data. Smart contracts are used as carriers, to record these rules. Blockchains can process programs in parallel, only if, multiple smart contracts can be executed concurrently and in a non-sequential manner. Basically, if contracts do not interact with each other, or if the contract does not modify the same state data, at the same time, their execution is non-sequential and can be executed concurrently. Otherwise, it can only execute in series, following a sequential order, and the network is unable to scale horizontally.

Based on the analysis above, we can easily design "unlimited scaling" in smart contract systems. All we must do is to set up simple rules:

- **A smart contract can only modify the state record of the contract that it belongs to**
- **In the same transaction batch (block), a contract can only be running once**

As a result, all the smart contracts can be processed in parallel as sequential order is irrelevant to the result. However, if a "smart contract can only modify the state record of the contract that it belongs to", it implies that the contract cannot call each other. Each contract, is an isolated island; if "In the same transaction batch (block), a contract can only be running once", this implies that a digital asset issued with a smart contract, can only handle one transaction per block. This is a world of difference with the original design goals of "smart" contracts, which cease to be "smart". After all, our design goals include both mutual call between contracts, and multiple execution of the same call, in the same block.

Fortunately, smart contracts in NEO have a static call relationship, and the call target cannot be specified at run time. This allows the behavior of the program to be fully determined before execution, and its call relationship to be fully defined before it can run. We require that each contract explicitly indicate the contracts which are likely to be invoked, so that the operating environment can calculate the complete call tree before running the contract procedure, and partition execution of the contracts, based on the call tree. Contracts that may modify the same state record, are executed in a sequential manner within the same partition, whereby different partitions can then be executed in parallel.

2.3.2 Low coupling

Coupling is a measure of the dependency between two or more entities. NeoContract system uses a low-coupling design, which is executed in the NeoVM, and communicates with the non-blockchain data through the interoperable service layer. As a result, most upgrades to smart contract functions can be achieved by increasing the API of interoperable services.

3. Contract Use

3.1 Contract Verification

Unlike the public-key account system used in Bitcoin, NEO's account system uses the contract account system. Each account in the NEO corresponds to a verification contract, and the hash value of the verification contract, is the account address; The program logic of the verification contract controls the ownership of the account. When transferring from an account, you firstly need to execute the verification contract for that account. A validation contract can accept a set of parameters (usually a digital signature or other criteria), and return a boolean value after verification, indicating the success of the verification to the system.

The user can deploy the verification contract to the blockchain beforehand, or publish the contract content directly in the transaction during the transfer process.

3.2 Contract Application

The application contract is triggered by a special transaction, which can access and modify the global state of the system, and the private state of the contract (storage area) at run time. For example, you can create a global digital asset in a contract, vote, save data, and even dynamically create a new contract, when the contract is running.

The execution of the application contract requires charging by instruction. When the transaction fee is consumed, the contract will fail and stop execution, and all state changes will be rolled back. The success of the contract does not affect the validity of the transaction.

3.3 Contract Function

The function contract is used to provide some public or commonly used functions, which can be called by other contracts. The smart contract code can be reused, so that developers are able to write increasingly complex business logic. Each function contract, when deployed, can choose to have a private storage area that is either read or written to data in a future contract, achieving state persistence.

The function contract must be pre-deployed to the chain to be invoked, and removed from the chain by a "self-destructing" system function, which will no longer be used and its private storage will be destroyed. The old contract data can be automatically migrated to another subcontract before it is destroyed, using contract migration tools.

4. Virtual Machine

4.1 Virtual Hardware

NeoVM provides a virtual hardware layer, to support the execution of smart contracts, including:

- **CPU**

CPU is responsible for reading and sequentially order the execution of instructions in the contract, according to the function of the instruction flow control, arithmetic operations, logic operations. The future of the CPU function can be extended, with the introduction of JIT (real-time compiler) function, thereby enhancing the efficiency instruction execution.

- **Call Stack**

  The call stack is used to hold the context information of the program execution at each function call, so that it can continue to execute in the current context after the function has finished executing and returning.

- **Calculate Stack**

  All NeoVM run-time data are stored in the calculation stack, when after the implementation of different instructions, the stack will be calculated on the corresponding data elements of the operation. For example, when additional instructions are executed, the two operations participating in the addition are ejected from the calculation stack, and the result of the addition is pushed to the top of the stack. Function call parameters must also be calculated from right to left, according to the order of the stack. After the function is successfully executed, the top of the stack fetch-function returns the value.

- **Spare Stack**

When you need to schedule or rearrange elements in the stack, you can temporarily store the elements in the spare stack and retrieve them in the future.

4.2 Instruction set

NeoVM provides a set of simple, and practical instructions for building smart contract programs. According to functions, the main categories are as follows:

- Constant instruction
- Process control instruction
- Stack operation instruction
- String instruction
- Logic instruction
- Arithmetic operation instruction
- Cryptographic instruction
- Data operation instruction

It is worth noting that the NeoVM instruction set provides a series of cryptographic instructions, such as ECDSA, SHA and other algorithms to optimize the implementation efficiency of cryptographic algorithms in smart contracts. In addition, data manipulation instructions directly support arrays and complex data structures.

4.3 Interoperable service layer

The virtual machine where smart contract executes is a sandbox environment, that requires an interoperable service layer, in times when it needs to access data outside of the sandbox or to keep the run-time data persistent. Within the interoperable service layer, NEO contract can open a series of system function and services with the smart contract program, and these contracts can be called and accessed, like ordinary functions. All system functions are being conducted concurrently, so there is no need to worry about scalability.

4.4 Debugging Function

Often, the development of smart contracts is very difficult, due to the lack of good debugging and testing methods. NeoVM provides program debugging support at the virtual machine level, where you can set the breakpoint on the contract code, or single-step, single-process execution. Thanks to the low coupling design between the virtual machine and the blockchain, it is easy to integrate NeoVM directly with various IDEs, to provide a test environment that is consistent with the final production environment.

5. High-level language

5.1 C#, VB.Net, F#

Developers can use NeoContract for almost any high-level language they are good at. The first batch of supported languages are C #, VB.Net, F #, etc. We provide compilers and plug-ins for these languages, allowing compilation of these high-level language into the instruction set, supported by NeoVM. As the compiler focus on MSIL (Microsoft intermediate language) during compilation, so theoretically, any. Net language can be translated into MSIL language, and become directly supported.

A huge number of developers are proficient in these languages, and the above languages have a very strong integrated development environment. Developers can develop, generate, test and debug, all within Visual Studio, where they are able to take full advantage of the smart contract development templates we provide, to gain a head start.

5.2 Java, Kotlin

Java and Kotlin forms the second batch of supported languages, where we provide compilers and IDE plugins for these languages, to help developers use the JVM-based language to develop NEO's Smart Contract applications.

Java is widely used, and Kotlin has recently become the official Google recommended, Android-development language. We believe that supporting these languages will help drastically increase the number of NEO smart contract developers.

5.3 Other Languages

Afterwards, NeoContract will add support for other high-level languages, based on the degree of difficulty, in the complier development process. Some of the languages that may be supported, include:

- C, C++, GO
- Python, JavaScript

In the future, we will continue to add more high-level language support. Our goal is to see more than 90% of NEO developers developing with NeoContract, without needing to learn a new language, and even possibly transfer existing business system code directly onto the blockchain.

6. Service

6.1 Blockchain Ledger

NEO Smart Contracts can obtain complete block data for the NEO blockchain, including complete blocks and transactions, and each of their fields, at runtime, through the system functions provided by the interoperable service. Specifically, you can query these data:

- Height of the blockchain
- Block head, current block
- Transactions
- Type of transaction, attributes, input, output, etc.

Through these data, you can develop some interesting applications, such as automatic payouts, smart contracts based upon proof of workload.

6.2 Digital Assets

Through the interoperable services provided by the digital asset interface, smart contracts not only can query the NEO blockchain on properties and statistics of various digital assets, but also, create new digital assets during its run-time. Digital assets created by smart contracts can be issued, transferred, traded outside of the contract. They are the same as original assets on NEO, and can be managed with any NEO-compatible, wallet software. This specific interface includes:

- Asset attribute inquiry
- Asset statistics query
- Asset life cycle management: create, modify, destroy, etc.
- Asset management: multi-language name, total change, precision change, changes in the administrator

6.3 Persistence

Each smart contract program deployed on the NEO blockchain, will have a private storage area that can only be read and written by the contract itself. Smart contracts have full operational permissions on the data in its own store: can be read, written, modified, deleted. The data is stored in the form of key-value pairs and provides these interfaces:

- Traverse all the records stored
- Return to a specific record according to the specified key
- Modify or write new records according to the specified key
- Delete the record according to the specified key

In general, a contract can only read and write data from its own store, with one exception: when a contract is invoked, the invoked contract can access the caller's store through a cross-domain request, provided that the caller provides authorization. In addition, for a sub-contract that is dynamically created at the time of contract execution, the parent contract gets instant access to its store.

Cross-domain requests enable NeoContract to implement rich library capabilities, that provide highly scalable data management capabilities for the callers.

7. Fees

7.1 Deployment Fee

NEO's distributed architecture provides high redundancy of storage capacity, and the use of this capacity is not free. Deploying a smart contract on the NEO network requires a fee, currently fixed at 500GAS, which is collected by the system and recorded as a system gain. Future fees will be adjusted according to the actual operating cost in the system. The smart contract deployed on the blockchain can be used multiple times, until the contract is destroyed by the deployer.

7.2 Implementation Fee

NEO provides a credible execution environment for smart contracts, and the execution of contracts requires the consumption of computing resources for each node, therefore users are required to pay for the execution of smart contracts. The fee is determined by the computational resources consumed with each execution, and the unit price is also in GAS. If the implementation of the smart contract fails due to lack of GAS, the cost of consumption will not be returned, and this prevents malicious attacks on the network power consumption.

For most simple contracts, they can be executed for free, so long as the execution costs remain under 10 GAS, thus greatly reducing costs for the user.

8. Application Scenarios

8.1 Superconducting Transactions

Digital assets on the blockchain inherently require some form of liquidity and usually point-to-point transactions cannot provide it sufficiently. Therefore, there is a need for exchanges to provide users with trading services. Digital asset exchanges can be broadly divided into two categories:

1. **Central exchanges:** where the user needs to deposit the digital assets with the exchange, and subsequent place pending orders for trading, on the website
2. **Decentralized exchanges:** where its trading system is built into the blockchain, and the system provides the matching services.

Centralized exchanges can provide very high performance and diversified services, but need to have a strong credit guarantee, otherwise there will be moral hazards; such as misappropriation of user funds, fraud, etc. Comparatively, decentralized exchange has no moral hazard, but the user experience is poor, and there is greater performance bottleneck. Is there a way to combine both solutions and achieve the best of both worlds?

Superconducting transactions is a mechanism that can do this; Users do not need to deposit assets, where they are able to use their own assets on the blockchain in trading. Transaction settlement complete on the blockchain, but the process of matching orders occurs off-chain, by a central exchange that provides matching services. Since the matching is conducted off-chain, its efficiency is like centralized exchanges, but the assets remain under the control of the user. Exchanges uses the user's trading intent to carry out matching services, with no moral hazards involved, such as misappropriation of user funds, fraud, etc.

At present, within the NEO community, development of smart contracts to achieve blockchain superconducting transactions has emerged, such as OTCGO.

8.2 Smart Fund

Smart funds based on the blockchain have the benefit of being decentralized, open and transparent, possessing a higher degree of security and freedom as compared to traditional funds. These smart funds are also cross-border and open to investors worldwide, where outstanding projects can be funded with capital from all across the world.

Nest is a NeoContract-based smart fund project, which is very similar to the TheDAO project based on Ethereum, where improved security measures is needed to avoid the mistakes of TheDAO (hackers).

8.3 Cross-chain Interoperability

In the foreseeable future, there will be many public chains and thousands of alliance chains or private chains in existence worldwide. These isolated blockchain systems are islands of value and information, which are not interoperable with each other. Through the cross-chain interoperability mechanism, numerous isolated blockchains can be linked, so that the values in different blockchains can be exchanged with each other, to achieve the true value of the Internet.

NeoContract provides support for the implementation of cross-chain interoperability, ensuring consistency within cross-chain asset exchange, cross-chain distributed transactions, and execution of smart contracts on different blockchains.

8.4 Oracle Machines

The concept of oracles in folktale lies in the ability of a certain supernatural entity that can answer a particular set of questions. In the blockchain, oracle machines open the door to the outside world for smart contracts, making it possible for smart contracts to use real-world information as a condition for contract execution.

NeoContract does not provide the ability to directly access external data, such as access to resources on the Internet, because this will introduce non-deterministic behavior, resulting in inconsistencies between nodes during contract execution. Implementing the oracle machine in NeoContract requires that external data be sent to the blockchain by a trusted third party, integrating these data feeds as part of the blockchain ledger, thereby eliminating non-determinism.

The credible third party mentioned above, may be a person or institution that is co-trusted by both parties in the contract, or a decentralized data provider that is guaranteed by economic incentives. In this manner, NeoContract can be used in the implementation of such Oracle machines.

8.5 Ethereum DAPP

Bitcoin created the era of blockchains and electronic cash, and Ethereum created the era of smart contracts. Ethereum, the pioneers of smart contract on the blockchain, has made great contributions to the design idea, economic model and technological realization of a smart contract system. At the same time, the Ethereum platform has seen many DAPPs (distributed applications), where functionalities including: gambling agreements, digital assets, electronic gold, gaming platform, medical insurance, marriage platform, with widespread use over many industries. In theory, all of these DAPPs can be easily transplanted onto the NeoContract platform, as a NEO application.