
White Paper V 1.2

# GiantContracts

Date of issue: **11 September 2018**

# GiantContracts White Paper

# Preface

Smart contracts are computer programs that automatically fulfill the terms of the contract which are fixed at the time of its creation. The concept of autonomous contracts is not new and it was first formulated in 1994 by a cryptographer Nick Szabo. However, due to the lack of a decentralized and at the same time secure environment, his idea has not been widely implemented.

The first working example of this environment appeared in 2008 and it is known as Bitcoin. To verify and confirm Bitcoin transactions the Bitcoin blockchain uses a very simple virtual machine, the so-called Bitcoin Script. Bitcoin Script allows you to implement scenarios which are more complex than just transfer of funds, for instance, a receival of payment by multi-signature, deferred payments and so on. However, Bitcoin Script cannot be described as a complete environment for smart contracts since it is not a separate entity in the Bitcoin blockchain. Due to this fact, the more complex logic with an autonomous smart contract balance or with transfers from one smart contract to another cannot be implemented.

A fundamentally new blockchain of Ethereum was launched in 2014 and it provided a full-fledged environment for the execution of smart contracts. In comparison with the Bitcoin Script, the Ethereum virtual machine allows you to realize almost any idea in the form of a smart contract. Nevertheless, the implementation of Ethereum has several disadvantages such as:

- The code of the smart contract, which is already in the blockchain, cannot be changed and this can create inconveniences for fixing bugs of smart contracts. Thus, the creation of a new code leads to a new smart contract deployment which leads to a new smart contract address. These steps cause both a change to the whole logic tied to the address of the previous version of the smart contract and a loss of confidential information stored in the contract repository. The majority of Ethereum smart contracts remain uncorrected which makes the use of such smart contracts risky for ordinary users of the Ethereum network.
- A barrier for the entrance of new developers into the development of smart contracts because of the Ethereum's own programming language Solidity.

The implementation of smart contracts by Giant offers the following features:

1. JavaScript of ECMAScript 6 standard - a programming language for writing Giant smart contracts

2. The possibility to edit the code of the deployed smart contract
3. The signature of a smart contract with a private key of the Giant masternode for its deployment into the main Giant network

## Giant Virtual Machine

The Giant virtual machine executes JavaScript code of the ECMAScript 6 standard. Giant uses a specific JS engine V8 (Chrome) version 6.6. Thus, the programming language of the Giant smart contracts supports ECMAScript 6 features as much as the V8 does. It is important to note that in V8, the majority of the ES6 features are supported only when `use strict` is on.

A number of conditionalities is applied in the GiantContracts:

- Smart contract is a class declaration
- The class of a smart contract should be exported from the source code of a module by default, that is, by `export default` command
- The class of a smart contract should be inherited from the Contract class provided by the Giant
- All declared methods in the smart contract class are public and available for call in the blockchain
- All the declared methods and properties of the smart contract class are stored as part of the smart contract object in the smart contract storage
- The source code of a smart contract can contain JS code outside the smart contract class, but the properties installed outside the class of the smart contract are not saved between the executions of the smart contract

Basic contract implies a number of certain conventions fixed in the form of a class. Each smart contract has an address which means it can accept payments and has its own balance. The basic implementation of the Contract class will provide the basic capabilities of a smart contract both to accept transactions where the recipient is the current smart contract and to send transactions where the sender is the current smart contract.

Thus, the simplest Giant smart contract is as follows:

```
'use strict'
import Contract from 'GiantContract'

export default class SimpleContract extends Contract {

  constructor(text = 'Hello world!') {
    this.text = text
  }

  setText(text) {
    this.text = text
  }

  getText() {
    return this.text
  }
}
```

The implementation of the Giant smart contract execution environment will have some limitations in order to resolve non-deterministic behavior issues of JS code on different nodes. If the smart contract is not deterministic, the results of different nodes may be inconsistent. As a result, a consensus between nodes cannot be reached and the network becomes stagnant. These restrictions will be described in more detail in a GiantContracts technical documentation.

## Smart Contract Code Updating

The smart contract code updating mechanism is a very demanded feature that is lacking in the ecosystem of Ethereum, NEO and in the other smart contracts. Smart contract code update will allow:

- not to lose sensitive data of the smart contract during its update
- to promptly fix bugs in a working smart contract
- to iteratively develop a smart contract, gradually improving its functionality

JavaScript is a dynamic programming language that allows you to define data types and to perform syntax analysis and “just in time” compilation (JIT), at the stage of program execution. Each smart contract has its own isolated storage, which stores the status of the smart contract execution. It stores the status since the moment of its deployment into the blockchain and until the last call of any smart contract method. Because of the dynamic nature of the JavaScript, the method declared in a smart contract class is also an object that is kept in the smart contract storage. This allows you to replace the smart contract method, through a special method provided by the developer of the smart contract.

An example of an implementation of such a mechanism with a simple verification where changes to the smart contract code can be made by the same user who carried out the smart contract deployment, is given below:

```
‘use strict’
import Contract from ‘GiantContract’
import {getCallerAddress} from ‘GiantBlockchain’

export default class ChangableContract extends Contract {

  constructor() {
    this.owner = getCallerAddress()
    this.a = 1
    this.b = 2
  }

  get() {
    return this.a
  }

  changeCode(newCode) {
    if (this.owner === getCallerAddress()) {
      eval(newCode)
    }
  }
}
```

Example of execution in the js console:

```
giantjs> const c = new ChangableContract()
giantjs> c.get()
1
giantjs> c.changeCode('this.get = () => this.b')
giantjs> c.get()
2
```

Even though this mechanism is highly applicable for the developers of smart contracts, it could cause a potential vulnerability to smart contracts in cases when developer treats them dismissively. JS has built-in mechanisms for protection of the properties of the object from rewriting. In such case, the code of the smart contract will look as follows:

```
…
export default class NotChangableContract extends Contract {
  …
}

Object.defineProperty(NotChangableContract.prototype, 'get', {
  value: () => this.a,
  enumerable: false,
  configurable: true,
  writable: false
});
```

In this case, the progress in the JS shell will be as follows:

```
giantjs> const c = new NotChangableContract()
giantjs> c.get()
1
giantjs> c.changeCode('this.get = () => this.b')
giantjs> c.get()
1
```

In the future, we are going to implement a support for the ES7 standard, which will have shorter solutions - decorators. With the help of the decorators, you can get the same solution in a more elegant way:

```
…
function notModifyed(target, key, descriptor) {
    descriptor.writable = false;
    return descriptor;
}

export default class NotChangableContract extends Contract {
  …

  @notModifyed
  get() {
    return this.a
  }
}
```

## Smart Contracts Interaction

In the Giant blockchain, smart contracts can interact with each other, in other words, some contracts can call the methods of other contracts or even generate new smart contracts. However, this kind of interaction has certain limitations, for instance, Giant will not support recursive calls of smart contracts. Recursion can be achieved within a contract, but it cannot cross the boundaries of the current contract.

The links between contracts can be both static and dynamic which gives greater flexibility to the interaction between smart contracts.

The static connection of the external smart contract will look like a simple import to the smart contract address, and then in the current smart contract it will be available as an object with methods available for calling, for example:

```
‘use strict’
import Contract from ‘GiantContract’
import invokableContract from ‘0x123…’

export default class InvokedContract extends Contract {

  constructor() {
    this.foo = invokableContract.bar()
  }
}
```

Dynamic connection of an external smart contract is a more flexible solution and it is implemented through the call method imported from the GiantUtils package. It allows smart contracts to interact “just in time”. An example below

demonstrates a simple interaction between two smart contracts where one smart contract can subscribe for a call of its methods with another smart contract.

The first smart contract has an interface for adding calls of other smart contracts:

```
'use strict'
import Contract from 'GiantContract'
import {sha3, call, getTime} from 'GiantUtils'

export default class InvocationContract extends Contract {

  constructor() {
    this.invocations = new Map()
  }

  addInvocation(invokedAt, invokedContractId, fnName) {
    const invocationKey = sha3(invokedAt + invokedContractId + fnName)

    this.invocations.set(invocationKey, {
      time: invokedAt,
      contractId: invokedContractId,
      fn: fnName
    }
  }

  getInvocations() {
    return this.invocations
  }

  invoke(invocationKey) {
    const invocation = this.invocations.get(invocationKey)

    if (invocation && invocation.invokedAt >= getTime()) {
      call(invocation.contractId, invocation.fnName)
      this.invocations.delete(invocationKey)
    }
  }
}
```

The second smart contract signs to execute its method in the future in the constructor.

```
'use strict'
import Contract from 'GiantContract'
import {getAddress, getBlockTime} from 'GiantBlockchain'
import invocationContract from '0x123…'

export default class InvokedInFutureContract extends Contract {

  constructor() {
   invocationContract.addInvocation(getBlockTime() + 3600, getAddress(),
'foo')
  }

  foo() {
    // doing something in future
  }
}
```

This example has a number of disadvantages, for instance `InvocationContract` does not provide for the transfer of parameters into the called methods of other smart contracts. However, it is only a demonstration of the potential of the interaction of smart contracts in the Giant blockchain.

## Smart Contracts Destruction

Smart contracts destruction is an extremely useful feature for the Giant ecosystem. It will make impossible for users to utilize the outdated or irrelevant smart contracts and to clear all the data they use in the smart contract storage. Definitely, this mechanism should be used by the developers with caution. However, it is important to understand that all calls of the methods of the smart contract remain in the blockchain, therefore, the state of the storage of the smart contract until its destruction is completely recoverable. In fact, when a smart contract is destroyed, only the off-chain data of the smart contract is cleared.

Below you can see an example of a smart contract with a destroy function that is available to any user:

```
'use strict'
import Contract from 'GiantContract'
import destroy from 'GiantDestroy'

export default class DestroyableContract extends Contract {

  constructor() {
  }

  destroyContract() {
    destroy()
  }
}
```

## Smart Contracts Signing

In the Giant mainnet, the smart contract method call and deployment are possible only if it is signed by a private key of an active Giant masternode. This creates a barrier for a smart contract creation, but the use of this mechanism is due to several reasons:

- It does not allow the creation of garbage smart contracts in the Giant blockchain. Each smart contract should be useful and applicable for users of the Giant network
- The creator of the smart contract has to become an investor of the Giant network. It should encourage the growth of the audience interested in the development of the Giant blockchain
- The masternode owner has an opportunity to influence the development of the Giant technology through voting. In this case, the creators of smart contracts will have a real opportunity to influence the further development of the Giant smart contracts ecosystem

This restriction may be lifted in the future, but only if the Giant community votes so.

In the testnet of Giant, smart contract methods placement and call will not depend on the availability of an active masternode.

## Efficiency

The efficiency of execution of smart contracts is essential for the success of the ecosystem, as it directly affects the potential number of transactions per time unit and increases (or decreases) the requirements for the hardware on which the full Giant network node is deployed. When we analyze the efficiency of any execution environment, there are two main metrics that are critical:

- The speed of code execution
- The speed of initialization of the execution environment

For smart contracts, the execution environment is often more important than the speed of instructions execution. Each time a smart contract is called, it must start a new virtual machine. Therefore, the speed of launching the virtual machine has a great impact on the efficiency of the smart contract system.

Giant uses the JS V8 engine as its smart contract execution environment, which runs very quickly, takes up few resources and has built-in mechanisms for optimizing JS code.

V8 engine differs from the other JS engines by a high performance, and this has become more noticeable after the V8 switched to the use of Ignition + TurboFan.

## Fees

Unlike in Ethereum, NEO and other networks, where an additional token, the so-called GAS, is charged for the use of their ecosystem, we use a different approach to pay for the network resources in Giant. In the Giant network, the transaction fee does not go to the miner, but simply burns helping to counter GIC inflation. The same happens when a smart contract is placed or its methods are executed. In doing so, the user operates the price of the transaction in the smart contract (GasPrice analogue) and the amount that he can spend on the execution of the smart contract code. Means of deployment and call of smart contracts will assess the number of GIC tokens required for the execution of an operation. The unused commission will be burned.

The Giant architecture provides high redundancy of the storage capacity of the smart contract (essentially it is limited only by physically available space on the host), and the use of this capacity is not free. The deployment of a smart contract in the Giant network requires a fee which will be determined further in the technical documentation. A smart contract deployed on the Giant blockchain can be used multiple times until it is destroyed by its creator.

Giant provides a secure execution environment for smart contracts. In order to execute the contract, the consumption of computing resources for each full node in the network is required, so the users of smart contracts have to pay for the execution. If the commission is not enough to perform a smart contract then its execution will be interrupted and the commission will be burned.

The majority of simple smart contracts will be executed almost free of charge. More expensive will be the operations of deploying/updating the code of the smart contract in the blockchain, of calling other smart contracts and of using the storage of smart contracts.

## Smart Contracts Implementation Examples

All code samples shown below are intended to demonstrate the ideas we want to implement and they cannot be taken as the final description of the syntax and vocabulary of the Giant smart contracts. Perhaps during the implementation, more appropriate technical solutions will be adopted.

Follow the documentation on our website and on our GitHub account. We consider the code documentation as important as the code writing.

We want to make an open platform and we hope to attract a wide audience of developers, both for the development of smart contracts on the Giant blockchain, and for the participation in the development of the Giant itself.

### Locking Smart Contract

The contract implements a function that defines a specific time stamp. No one can withdraw any assets from the smart contract before the specified time expires. As soon as the specified time is reached, the owners of the contract can withdraw their assets.

The current time received under the contract is the time of the last block in the blockchain (the error is the average block time).

```
'use strict'
import Contract from 'GiantContract'
import {GetBlockTime} from 'GiantBlockchain'

export default class LockContract extends Contract {

  constructor(timestamp) {
    this.timestamp = timestamp
  }

  // overriding the method inherited from Contract class
  send(to, amount) {
    if (this.timestamp >= getBlockTime()) {
      super.send(to, amount)
    }
  }
}
```

### Domain Name System Smart Contract

The contract introduced a system of domain names in which domain names point to data in the blockchain. These are not real Internet domain names. The above code implements requests, registration, transfer, and deletion of domain names.

```
'use strict'
import Contract from 'GiantContract'
import {GetBlockTime, getCallerAddress} from 'GiantBlockchain'

export default class DomainNameContract extends Contract {

  constructor() {
    this.domains = new Map()
  }

  query(domain) {
    return this.domains.get(domain)
  }

  register(domainName, ipv4) {
    if (this.domains.has(domainName)) {
      return false
    }

    this.domains.set(domainName, {
      owner: getCallerAddress(),
      address: ipv4
    })
    return true
  }

  transfer(domainName, ipv4) {
    const domain = this.domains.get(domainName)

    if (domain && domain.owner === getCallerAddress()) {
       domain.address = ipv4
    }
  }

  delete(domainName) {
    const domain = this.domains.get(domainName)

    if (domain && domain.owner === getCallerAddress()) {
       this.domains.delete(domain)
    }
  }
}
```

### Token Smart Contract

The contract implements a simple logic of the token issued on the Giant blockchain. The smart contract provides for the preliminary release of 1,000,000 smart contract tokens and for the exchange of GIC tokens for smart contract tokens at the rate of 1:1.

```
'use strict'
import Contract from 'GiantContract'
import {getAddress, getCallerAddress} from 'GiantBlockchain'

export default class TokenContract extends Contract {

  constructor() {
    this.balances = new Map()
    this.balances.put(getAddress(), 1000000)
  }

  onTransaction(tx) {
    const contractAddress = getAddress()
    const contractBalance = this.balances.get(contractAddress)
    const recipientAddress = getCallerAddress()
    const recipientBalance = this.balances.get(recipientAddress)

    this.balances.set(contractAddress, contractBalance - tx.amount)
    this.balances.set(recipientAddress,
      (recipientBalance ? recipientBalance : 0) + tx.amount)
  }

  getBalance(address) {
    return this.balances.get(address)
  }
}
```

## Testing and Debugging the Smart Contracts

Due to the low connectivity between the virtual machine and the blockchain, it is very easy to integrate Giant directly with any IDE to provide a test environment compatible with the execution environment of the Giant. Therefore, for the Giant smart contracts it will be possible to easily write both modular and integration tests, using the popular testing frameworks of JS code.

The Giant team values the efficient use of the eponymous blockchain network. In order to prevent the emergence of new smart contracts with mistakes we introduce GiantJS — a software developer's kit (SDK) for Giant smart contracts developers. It provides a sandbox environment for automated testing before using new smart contracts in the mainnet.

We would like to describe the advantages of GiantJS in the following list.

1. Automated contract testing with Mocha and Chai. Testing smart contracts using popular JS frameworks called Mocha and Chai. They allow to write modular, E2E and integration tests. In this case, 'automatic' means that GiantJS makes possible to automatically call them — for example before calling the deploy, not just manually typing the command 'giantjs test'.
2. Configurable build pipeline with support for custom build processes. A customizable build-and-deployment process. The pipeline (or the process of building) includes compilation, validation, tests launch, migration scripts execution and, finally, the deployment. Every process stage can be customized.

3. Scriptable deployment & migrations framework. The deploy and migration of smart contracts can be programmed with a script. Here, 'migration' means data operations before or after the deployment, for instance, updating the price balance with a new formula.

4. Instant rebuilding of assets during development. GiantJS will check if the code you are writing will actually work and will offer to change in necessary strings in case the code is corrupted.

5. External script runner that executes scripts within a GiantJS environment - sometimes the migration logic suffers from various obstacles e.g. asynchronous interaction. An external script will allow to interact easily in the context of the environment of the launched network and the deployed smart contract.

## Summary

Bitcoin has created an era of blockchain and private decentralized money. Ethereum has created an era of smart contracts. Ethereum made a huge contribution to the idea, the economic model and the technological implementation of the smart contract system. Currently, the Ethereum platform has already implemented many distributed applications, such as gambling, digital assets, electronic gold, gaming platforms, various types of insurance which are already widely used in many industries. Theoretically, all these Dapps can be easily transferred to the Giant blockchain platform.

Giant does not just simply replicate the technological solutions of Ethereum or other networks, but offers new key features, namely:

- Development of smart contracts using JavaScript standard ES6 (and ES7 in the near future)
- Smart contract code update
- Smart contracts destruction