Table of Contents

1) General presentation

Nodix is a new blockchain engine, made for the purpose of developing scalable distributed web applications based on blockchain engine.

It use an event based script engine to process p2p network events, like new blocks or transactions, or HTTP requests from web browser.

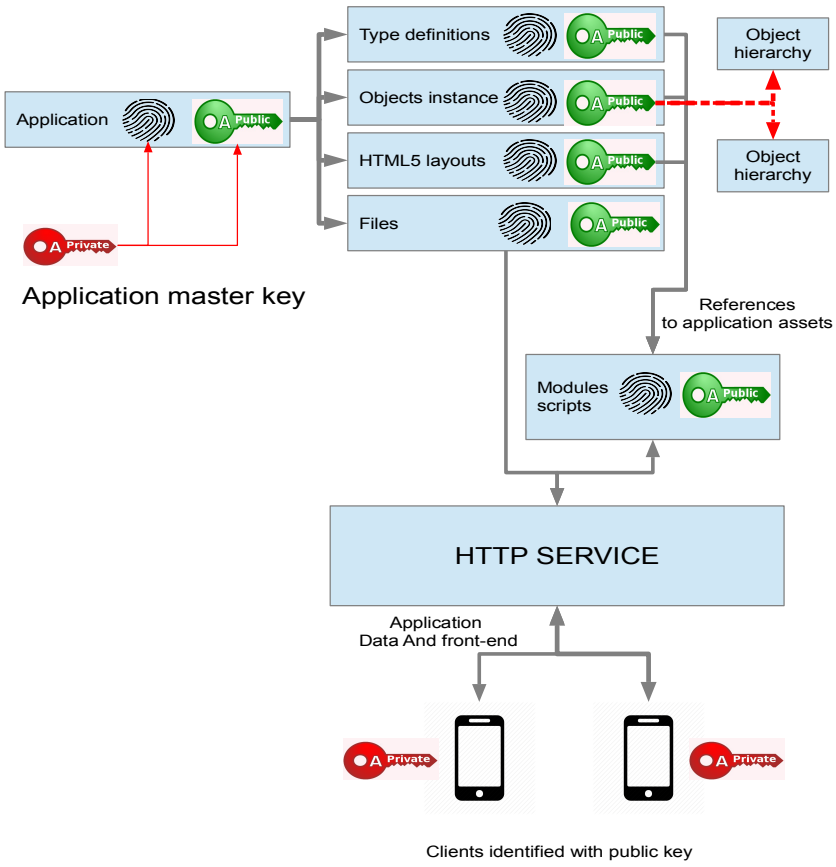
Web applications are composed of type definitions, objects, files, layout and modules/script. All together they provide a full stack environment to develop secure and fast distributed applications for the web.

Modules are created from native operating system binaries (dll for windows or elf for linux), and provide standard ABI across all operating system running on intel processors, which allow to verify the integrity and origin of all applications components and data from blockchain information.

Web clients interact with script functions to generate data and web pages with dynamic content from the blockchain, maximizing performance with lockless read access to blockchain for multi core scalability of asynchronous requests, and real time update using server push event model.

A secure HTML5 wallet is used to manipulate private key, doing transactions, staking, or interacting with applications, from any computer or device that has an HTML5 browser.

Applications are also created using the wallet, to associate a master with the application, used to add type definitions, layouts and scripts, and set basic permission for adding new objects or files.



Files are stored on the blockchain with a proof of existence that contain the file name, mime type, size , public key , and signature, and the file is shared using a specific part of the p2p protocol, so they don't have to be downloaded for validating the chain.

Objects can contain references to other objects or files, and list of references to other objects which allow to represent application data as a graph that can be modified/extended using transactions, with a permission model based on digital signature instead of cookie/sessions, which allow it to be fully decentralized and verified by all nodes.

Additionally, full binary tree can also be stored on the blockchain to represent abstract syntax tree used to represent any program in a simplified form that can be verified and executed externally.

All together, these features allow to overcome the limitation of monolithic node/wallet that require additional software layer with an HTTP server to make complex application available to the end user, switching user identification from private key to a classic cookie based session to access the private key only known to the additional software layer, or require a patchwork of several system like IPFS, classic web server additionally to the blockchain node.

Nodix can store all components of an application on the blockchain, all nodes can host applications with verified components using digital signature, and a client browser can verify and execute the code with verified data source, making the development and deployment of distributed application easy and secure.

2) Programing model

1. *Event based model*

Blockchain allow to solve the major problem of distributed system which is to guarantee the consistency of application data across all nodes in a decentralized configuration, as well as decentralized ownership proof using digital signature.

A blockchain store data as a directed acyclic graph (DAG), each transaction reference a list of input containing a digital signature to verify its property, and a list of output that specify the new owner of this property.

This DAG is used to represent application data as a tree, using type definition for each node, and transaction to add new data into the graph following type definition, and can use queries to fetch the information.

This graph structure replace a traditional database engine using queries on hierarchized object tree, instead of flat table/records.

Another feature that distributed system must have is asynchronous programming, as request to a remote node can take time to complete, or never complete at all, and blockchain protocol is message based, so all the code for blockchain processing and distributed applications is programmed used an event based model and applications requests are processed in parallel to maximize the performance on multi core architectures.

The core node code is based on message list similar to javascript or AS3, with handlers that are triggered using expression evaluation on messages. The protocol module define the blockchain p2p messages and their network layouts, the node module pump message from the network using the protocol module and then add them into a message list, which is processed asynchronously.

The “nodix.node” file in the export folder contain the node definition and handlers for the blockchain protocol.

```
let NODE_BITCORE_NODE SelfNode = `
{
        […]
        (NODE_BITCORE_MSG_LIST) "emitted_queue" : [],
        […]
}`
```

node is the object representing the node that emitted the message

payload is the object that represent the message as defined in the protocol module.

```
handler on_verack(node,payload) = ` code `

sethandler SelfNode.emitted_queue{ "cmd=verack" } = on_verack;
```

All the dynamic memory allocation is handled by the runtime and is garbage collected, so references to objects can be passed around copied and automatically freed when no reference to it exists anymore, without pointer ownership.

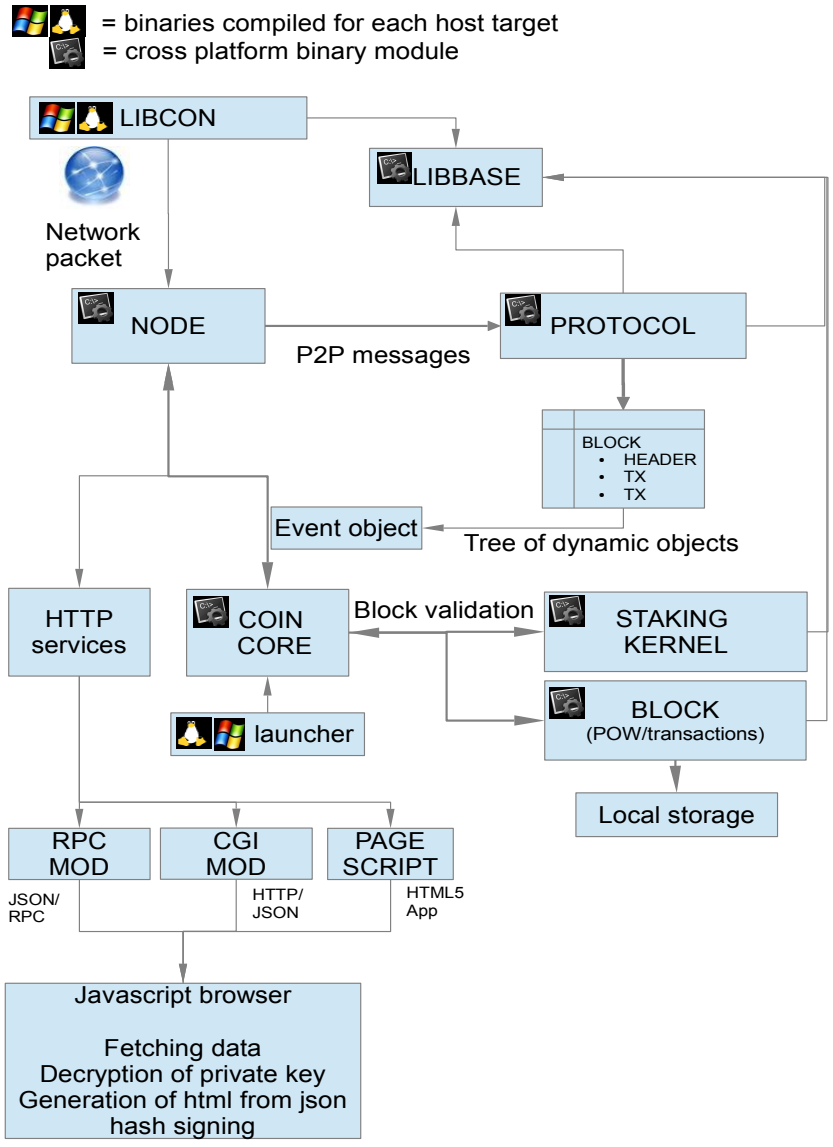
2. Local storage

The local storage exploit the fact that blockchain data is immutable and doesn't need update or delete operation that need to use non atomic operations to modify local database. Blockchain data can only be appended on top of existing data, or last block truncated off the end of the file, those operations are atomic on most operating system which allow lockless parallel read access, and prevent data corruption in case of crash or abnormal termination.

3. Modules

A specific binary module format is used to encapsulate different part of the code with a portable ABI. Each module contain a list of function that can be called from other modules or script.

Low coupling and garbage collection allow to easily modify module code independently of each others, and customize the blockchain protocol and validation code, or cryptographic primitives.



I. Core modules

Core modules are defined in the nodix.node file and loaded when the node start, and implement core node functions.

```
let NODE_MODULE_DEF  vec = `{"order":0, "file" : "modz/vec3.tpo"}`
let NODE_MODULE_DEF  protocol_adx = `{"order":1, "file" : "modz/protocol_adx.tpo"}`
let NODE_MODULE_DEF  block_adx = `{"order":2, "file" : "modz/block_adx.tpo"}`
let NODE_MODULE_DEF  wallet= `{"order":3, "file" : "modz/wallet.tpo"}`
let NODE_MODULE_DEF  node_adx = `{"order":4, "file" : "modz/node_adx.tpo"}`
let NODE_MODULE_DEF  nodix = `{"order":5, "file" : "modz/nodix.tpo"}`
```

vec implement SIMD vector operations.
protocol implement p2p protocol messages definitions.
block implement block and transaction validation.
wallet implement access to local cache of keys.
node implement blockchain state, network and peer management.
nodix is top level application event handlers.

II. Remote modules

Remote modules are modules that can be invoked remotely, from a client browser or application client.

They can be invoked via HTTP protocol using json/rpc format to represent the request with parameters and result or traditional CGI interface for direct HTTP request.

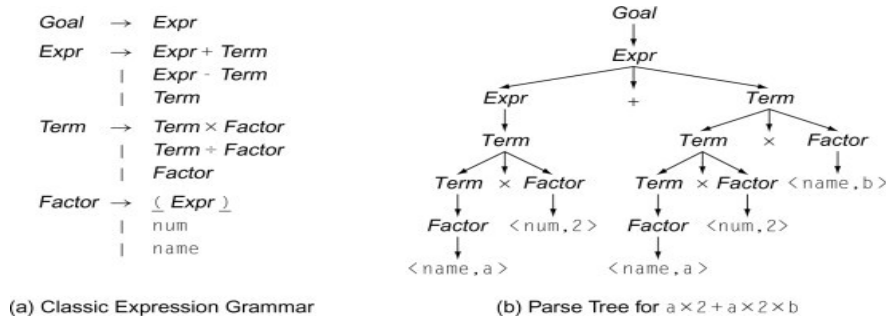
Remote modules can be added manually in the HTTP service definition of the nodix.node file, or loaded from trusted applications scripts hosted on the node.

4. Scripts

Scripts can be used to define json data sources, event sources, or web page generation. They can load objects, layouts and files from the blockchain, and generate a web page with dynamic data and event sources based on blockchain data.

They run in a protected memory environment, are fully sandboxed and can run with a capability based permission model to run untrusted code. They use managed multi thread memory and can execute in parallel like go routine with lockless read access to blockchain data.

5. Binary tree



(a) Classic Expression Grammar

(b) Parse Tree for a×2+a×2×b

Binary trees are trees for which each node has 1 parent and 2 children. Such trees can be used to represent any combination of algebraic expressions as a parse tree.

Parse tree can be represented on the blockchain using parse tree transactions. Such transactions have one output that contain the name of the function or operation that is executed on the transaction inputs, and either 2 inputs for operations or 1 input for functions, containing either a reference to the output of another binary tree transaction, the name of a constant supplied as an input value, or an immediate value for operations.

The parse tree can be executed on very simple trustless environment, and allow to build composable functional graph that can be verified to be conform to blockchain data.

Such parse trees can be used to represent a web page generation, using predefined composable HTML generation functions and operation including reference to application objects or files from the blockchain.

Complex computation can be represented this way and distributed for parallel execution of different branches.

Combined with the event programming model, such trees can be used to implement reactive functional style of programming.

6. Automated transaction (smart contract)

When an operation involving more than one party or when an actions involving a signature need to be automatized, smart contract can be used to delegate the property of an object or token on an automated execution.

A smart contract can be registered on the blockchain with a signature proving the ownership of the assets / data, defining the assets / data that it will manage, and condition that need to be met to validate a transaction involving those assets/data.

A transaction involving a smart contract will contain an input reference to the assets managed by the smart contract with a blank signature, the input that has to be evaluated to validate the transaction signed by the user initiating the transaction, and the outputs contain the assets to be transferred to the smart contract address, as well as the assets to be transferred to the transaction initiator.

The smart contract will then evaluate the conditions based on the assets transferred to the smart contract address, and validate the transaction of the assets it manage if the conditions are met.

3) Application consensus

The consensus model for applications use transaction to add new application item on the blockchain, but unlike transaction that transfer a value between two owners, application transactions are used to add items in the application.

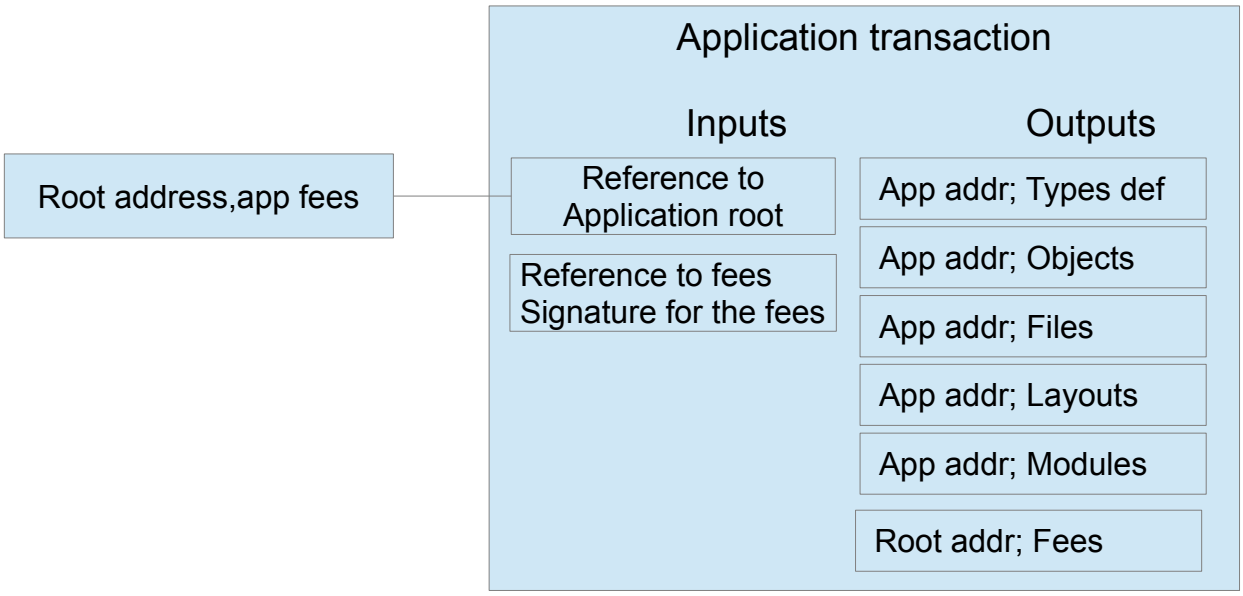
Each node can choose to host an application by adding it in the trusted applications section of the nodix.node file. Files and modules / scripts will only be hosted for applications in this list. Not every node has to run all applications, most node will only host a few or none.

Application data follow a type model that allow to shard data efficiently, and instant confirmation of new objects based on application type system. Each application establish its own consensus model, which allow instant confirmation and scaling at machine speed in most cases.

1. Application root

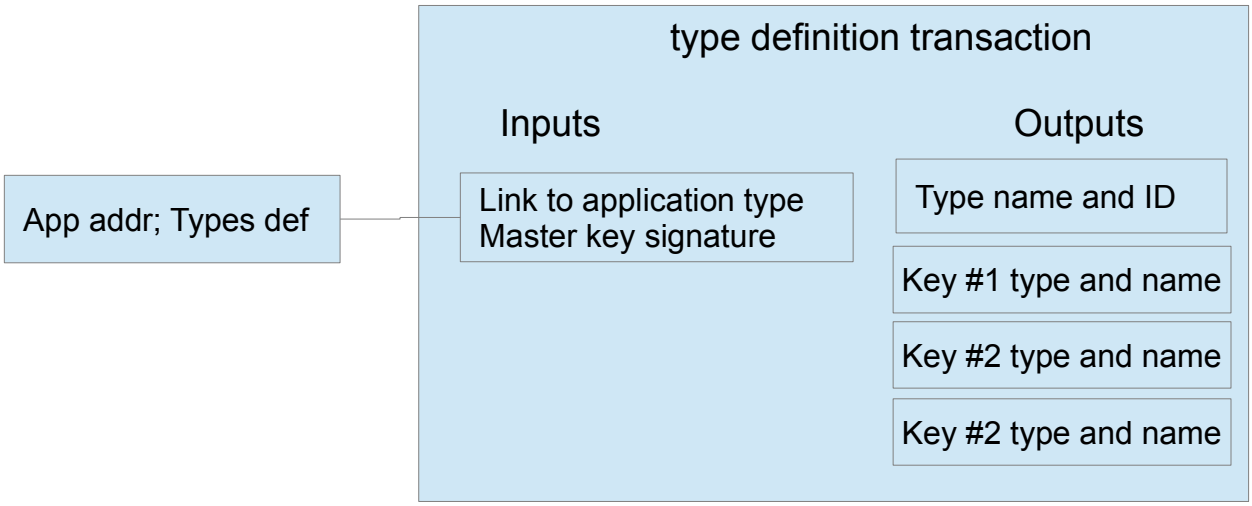
This transaction is used as the root node for all applications. It doesn't have any input, and output contain the fees that has to be paid to add a new application, and the destination address for the fees.

2. Application transaction



Application transaction create a new application on the blockchain. Its input reference the application root, and its 5 outputs are used to add type definition, objects, files, layouts and modules/scripts. The keys used to sign the new application become the master keys and is required to add type definitions, layouts and modules/scripts. The last output contain the application fees sent to the address specified in the application root.

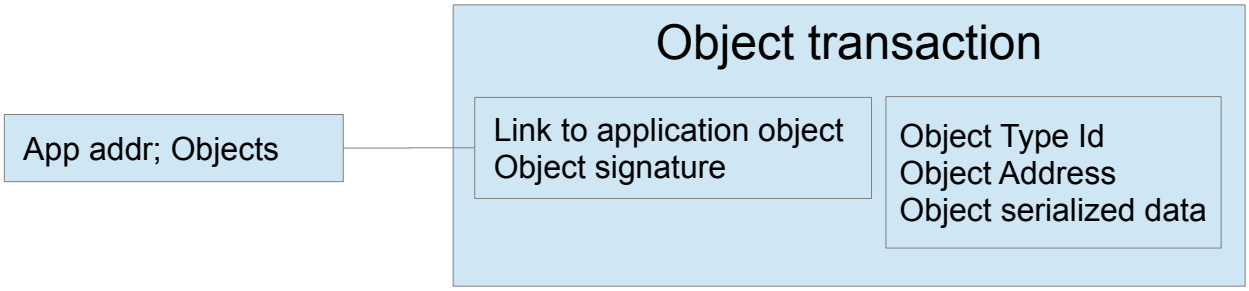
3. Type definition transaction



Type definition transactions are used to define the model that application will use to store data on the blockchain. Its input reference the application transaction, and must be signed used the application master key. Each output define a new key for the object, using a key type, key name, and indexing options similar to a classic database table.

Object keys can be either plain values, unboxed array of integer or floating points values, strings, references to other objects or files, or a list of reference to other objects. List of reference can have different access model, limited to the object owner or other conditions.

4. *Object transaction*



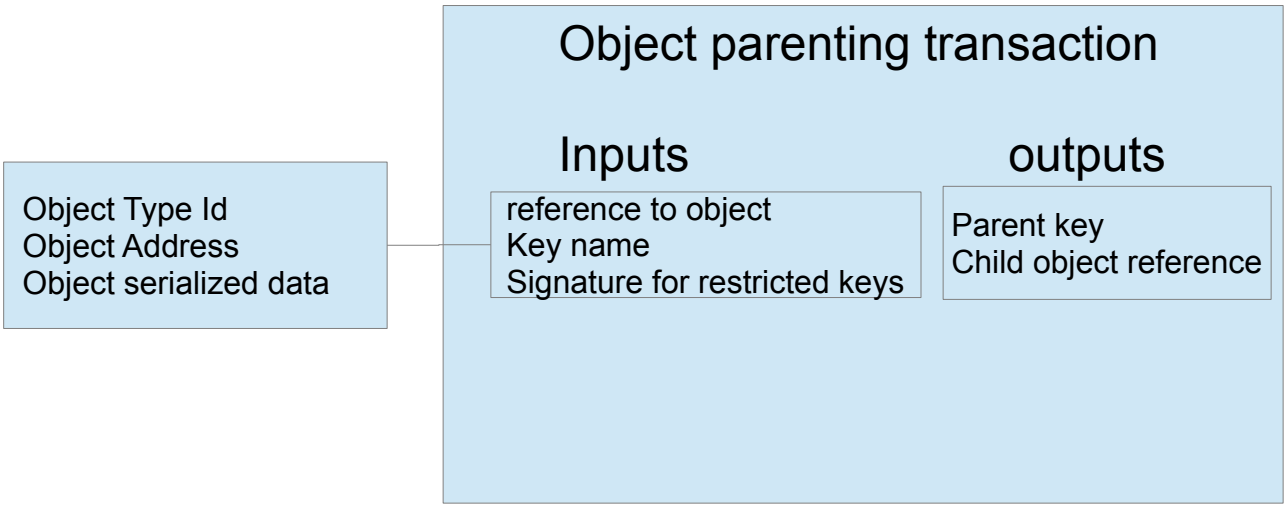
Object transactions are used to add new data to an application. Its input reference the application transaction, and the output contain the object type and object's data serialized using the type definition transaction corresponding to the type.

Object transaction doesn't "spend" anything so transaction containing new object can be confirmed instantly, as they cannot be canceled with a double spend. They only need to contain valid reference to an application and a type definition, the public key of the owner and the corresponding signature, and follow the application type model to be considered valid.

Objects are then referenced through a pair containing application name and object's transaction hash, called the genesis transaction.

A fraction of each block can be allocated to contain only object transactions additionally to usual property transfer transaction that require confirmations, to include them in the blockchain as soon as they are received, or they can stay in the memory pool and still be used in an application, as there is no transaction that can be mutually exclusive to it and need conflict resolution.

5. *Object parenting transaction*

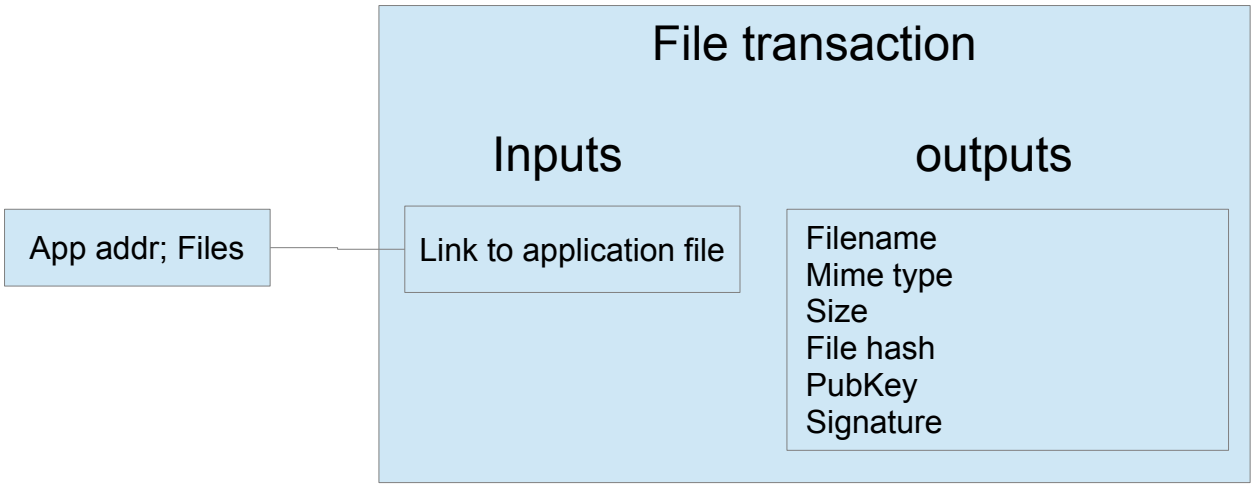


Objects definitions can contain restricted list keys into which new objects can be added dynamically using an object parenting transaction. Its input reference the parent object, and its output the name of the list's key, and the hash of the object to add to the list. The public key used to sign the transaction must match with the permissions defined in the object type of the parent object.

6. Object property transfer transaction

Object property transfer transaction are used to transfer the ownership of an object to a new owner. Its input contain a reference to the source object and the signature proving its ownership. Its output contain the address of the new owner. Objects instances are always referenced by the genesis transaction, and transfer history show the current owner.

7. File transaction

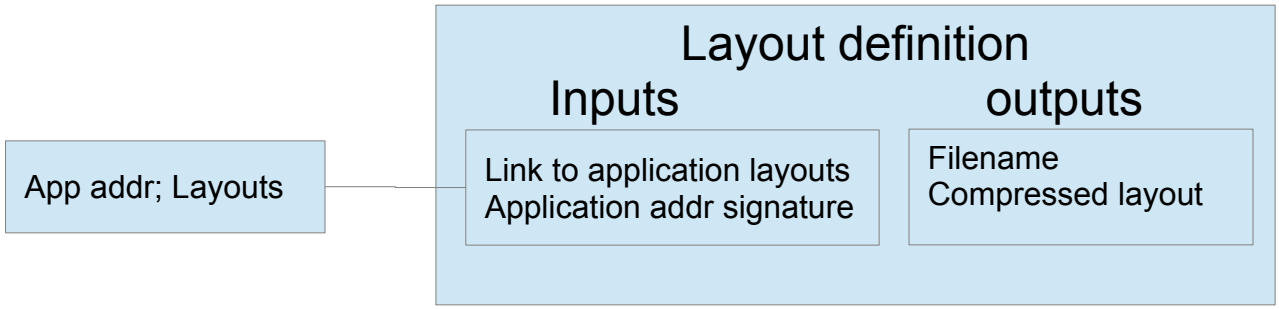


File transactions are used to store files references in the blockchain. Their input contains a reference to the application, and the output contain meta information on the file, the hash of the data, the name of the file, it's mime type and public key with the signature.

Files will be uploaded to a node hosting the application, then the hash of the file and meta are included into a file transaction. When another node receive this transaction, it optionally can make a request to download the file from the emitting node.

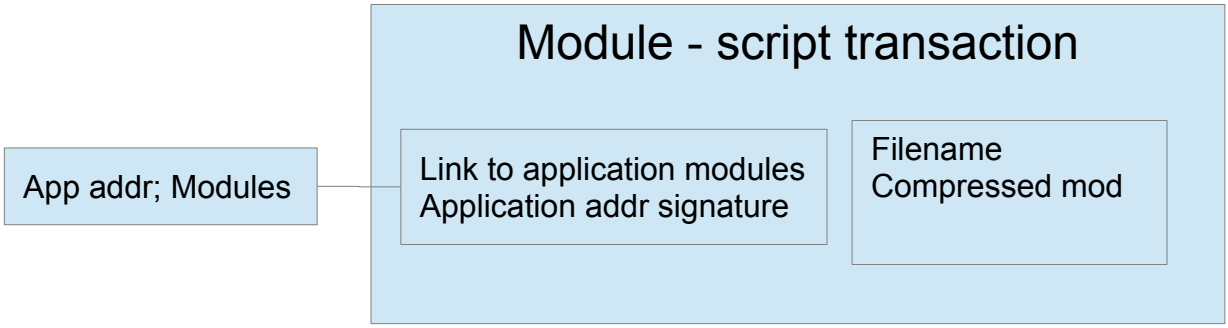
Files are then indexed using the hash of the file's data in sort that a client can check the file integrity from it's reference link.

8. *Layout transactions*



Layout transactions are used to store HTML layout on the blockchain. Its input contain a reference to the application, and the output contain the compressed HTML data. They must be signed using the application master key. Application scripts can reference the layout by its name, if a new layout is added with the same name, it will overwrite the old layout with the same name.

9. *Module – script transaction*



Module transactions are used to store new script or module on the blockchain. The input reference an application, and the output contain the compressed data of the script or module. If the application is in the node's trusted list, they will be hot loaded / updated when the transaction is received, and made available through the node's HTTP service.

Scripts are sandboxed and run with a capability permission model.

Binary module can either be run in separate process with IPC or run in sandbox for trustless computing.

4) Node services

Each node can run services additionally to the blockchain protocol, like HTTP service for serving client browser request, or STRATUM protocol for pooled mining.

The HTTP service bind modules or script either from local files from the web directory, or from trusted applications scripts hosted by the node and make the applications files available to an HTTP client, as well as pages and data sources implemented in the application script.

Binary trees can be executed via the wallet API, providing the input constants and the root node of the tree.

The stratum module allow for miners to submit share using the stratum protocol, giving a payment address as username. The reward for the found block transferred to the address depends on the amount of valid shares submitted by the miner.

The node has built-in javascript API available through the HTTP service to access the block explorer, wallet, application registry, and parse tree manipulation/execution.