Technical Paper

TRUSTO

ICO 3.0

How Trusto Blockchain Works

Trusto blockchain will support:
• 1500 transactions per second
• cross bolckchains smart contracts
• 100% decentralized crypto currency exchange builtin the blockchain • biometric digital identity

High speed consensus protocol

At the core of consensus protocol in Trusto blockchain is a gossip peer to peer protocol that enables broadcasting of transactions and the negotiating of the winning block before broadcasting the block.

The trusto blockchain implements the principle of multiple signed blocks. It does that by splitting the blockchain into subchains. Each subchain will start off with a root block as explained bellow. The nodes will generate transaction blocks and build up a subchain starting from this root node. The subchain will end when the next root block will be mined.

Multiple signatures works like this: when a consensus on a transaction block is reached, the network will start immediately the consensus for the next block. The next block will include the hash of the current block, so, the next block will sign the current block as well. The next root node will include the hashes of all the previous transaction blocks rooted on the previous root node, thus signing all the blocks in the blockchain. The fees of transactions included in the subchain will be shared by all the nodes that have blocks included in the subchain.

A reward is also given to the miner who creates a root node, making the mining process profitable even if there are very few transactions in the network.

To enable high speed transations, the consensus in the blockchain will be given by a special hybrid proof of stake (PoS) first consensus protocol secured by a proof of work (PoW) protocol. Here is how it works. There will be 2 kind of blocks, root blocks and transaction blocks.

The root blocks are mined through a classic PoW algorithm, there will be a small number of them and they will not contain transactions.

The transaction blocks will be generated by the nodes owning Trusto currency through a trust based PoS consensus protocol. There will be no mining involved in generating transaction blocks. The transaction blocks will contain transactions. The number of transactions blocks will be much larger than the number of root blocs; there will 100 transactions blocks for one root block.
The transaction blocks will form a subchain starting always with a root block.
The root block will sign and secure a subchain of transaction blocks.

These are the steps in adding blocks to the Trusto hybrid blockchain:

1. a root block is mined through PoW consensus protocol
2. transactions are broadcast to the nodes of the Trusto network using a gossip protocol
3. nodes on the network will compete on genrating a block and include transactions in it; the block generation is done with no mining and will be linked to the most recent root block
4. each generated block has a score that takes in consideration the trust the network has in the node, the number of blocks since the network las accepted a block from this node, the number of transactions included in the block and the number of blocks since the network last accepted a block from this node
5. each node, through a gossip protocol broadcasts the generated block to the neighbouring nodes, choosing together the wining block based on the block's score, the subchain length and the root node of the subchain; the subchain with the most recent root node is choosen, then the longest subchain, then the subchain with the biggest score
6. once a block is accepted by the node as the result of the gossip negotiation described above, the node starts to build the next block linked to the accepted block, thus creating the subchain that starts with the most recent root node

Addresses and accounts

The transactions in the Trusto network have multiple inputs and multiple outputs. An input refers to the output of a previous transaction. The output is a function that, when evaluated returns the condtions needed for that output to be spendable. Those conditions, as explained in the section about smart contracts bellow, are using a runtime that gives access not only to the state of the Trusto blockchain but also to the state of other blockchains, making the smart contracts work cross blockchains. For example you may condition the spending of a transaction in the Trusto network by the availability of a certain amount on Bitcoin address. When a transaction wants to spend an output of a previous transaction, it applies the output function to its input function and to the current state of the blockchains.

The result may be:

- **valid** making the previous output spended by the current transaction (so the output of the refferred transaction is transferred to the outputs of the current transaction)
- **invalid** and the current transaction is marked as invalid and is not broadcasted to the network; broadcasting to the network an invalid transaction will be considerred an act of attack and it will lower the trust of the node and will take the trust deposit made by the node and give it to the network
- **pending** meaning that the conditions for spending this output are not yet met; this will make that all outputs of the current transaction to be pending, and transactions trying to spend one or more of these outputs will also be pending
- **relaized** when conditions for a pending condition were met
- **not-realized** if a previously pending transaction was deemed as not realizable by the current state of the blockchain

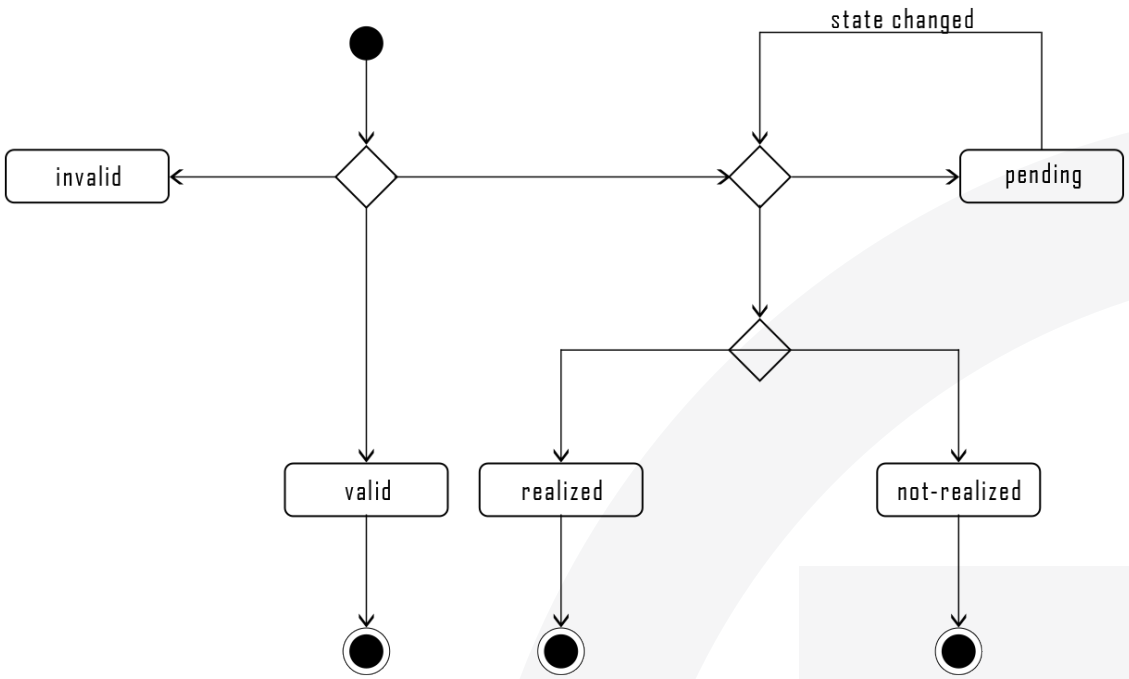


Figure 1: transaction state diagram

The mathematical model of valdating transaction is based on functions composition.

Using lambda calculus notation, the output is a function of the input (f) and the given state of blockchain (s):

λf.λs.OUT
where OUT is the lambda term describing the conditions that have to be met in order the output to be consumed by an input.

The input is also a function of the current blockchain state:
λs.I N
where IN is a lamda term reffering s which is the current state.

But, besides the status resulted of applying the refferred output's function to the current imput's function, we also must take into consideration the status of the referred output's transaction. If the referred output's transation is pending state, then then then the status of current transcation is also pending.

In lambda calculus, we define the possible values of the output's function like this:

INV ALID = λf.λg.λh.λi.λj.f
V ALID = λf.λg.λh.λi.λj.g
PENDING = λf.λg.λh.λi.λj.h
REALIZED = λf.λg.λh.λi.λj.i
NOTREALIZED = λf.λg.λh.λi.λj.g

and we can define a combination operator between these results, like this:

λx.λy.yyx(xxyyyx)(xxyxxx)(xxyyyy)

Lets use the notation t1 ∘ t2 for applying the above operator to t_1 and t_2, where t_1 and t_2 denotes states of transactions as resulted from applying the output's function to the input's function. If the transaction tn depends on transaction t_{n-1} and transaction t_0 is a generation transaction, then we can write the equation of the state of current transction like this:

$$t_n \circ t_{n-1} \circ t_{n-2} \circ \cdots \circ t_0$$

Thus, you can build a chain of pending transactions that, eventually will become *realized* or *not-realized*, depending on the conditions that were met or not. This will result in *trigger* like behaviour that is part of the *Smart Alert* functionality of the Trusto blockchain, implemented in a pure functional way, with immutable state and no imperative constructs.

The functional structure of smart contracts assures static analysis of the correct- ness of the smart contracts. Thus, a *pending* transaction will never be able to return *valid* or *invalid* states, it will only return *pending*, *realized* or *not-realized* outputs. These will be checked by the compiler of the smart contract code using static code analysis.

The smart contract

Accounts or addresses are a pair of private and public keys, identifying nodes in the Trusto network. They have associated a trust level that is computed as described bellow. The account public and private key are used to identify the account in the processes of creating *transaction blocks*, broadcasting transactions, mining blocks, etc.

The Trusto network supports expressing the rules of smart contracts in a powerfull yet simple statically typed concatentative functional language with dependent types based on cat programming language.

Cat Programming Language

Cat is a statically typed stack-based pure functional language inspired by Joy.
Cat has no variables, only instructions which manipulate a stack (e.g. `dup`, `pop`, `swap`), and a special expression form called a quotation (e.g. `[1 add]`) which pushes an expression onto the stack which can be executed at a later time
(e.g. using `apply` or `dip`).

For example: `6 7 dup mul sub` results in a stack with the value 43 on top.

This is a very efficient reimplementation of cat programming language in Haskell. We added the type dependent feature to the cat type system, and a blockchain specific runtime library system based on trust signature and block chain document storage.

The Primitive Instructions of Cat

The following is a core set of general purpose primitive (built-in) instructions of Cat and their types.

```
apply     : ((‘S -> ‘R) ‘S -> ‘R)
quote     : (‘a ‘S -> (‘R -> ‘a ‘R) ‘S)
compose   : ((‘B -> ‘C) (‘A -> ‘B) ‘S -> (‘A -> ‘C) ‘S) dup :(‘a’S->’a’a’S)
pop       : (‘a’S->’S)
swap      : (‘a’b’S->’b’a’S)
cond      : (Bool’a’a’S->’a’S)
while     : ((‘S -> Bool ‘R) (‘R -> ‘S) ‘S -> ‘S)```
```

The Syntax of Types

The Cat types express a function transformation between stacks. The terms to the left and right of the arrow represent the type configuration of the stack before and after the application of the function respectively. The last type term before the arrow, and the last type term after the arrow each represent the “rest” of the stack. This is an example of “row” polymorphism.

Each identifer represents an individual type on the stack, except for the last identifier before th

Identifiers preceded by an apostrophe are type variables which can map any type, including polymorphic functions.

Standard Library

The following are some of the standard library of Cat:

```
dip         = { swap quote compose apply}
rcompose    = { swap compose}
papply      = { quote rcompose}
dipd        = { swap [dip] dip}
popd        = { [pop] dip}
popop       = { pop pop}
dupd        = { [dup] dip}
swapd       = { [swap] dip}
rollup      = { swap swapd}
rolldown    = { swapd swap}
```

The types inferred are as follows:

```
dip      : ((‘t0 -> ‘t1) ‘t2 ‘t0 -> ‘t2 ‘t1)
rcompose : ((‘t0 -> ‘t1) (‘t1 -> ‘t2) ‘t3 -> (‘t0 -> ‘t2) ‘t3)
papply   : (‘t0 (‘t0 ‘t1 -> ‘t2) ‘t3 -> (‘t1 -> ‘t2) ‘t3)
dipd     : ((‘t0 -> ‘t1) ‘t2 ‘t3 ‘t0 -> ‘t2 ‘t3 ‘t1)
popd     : (‘t0 ‘t1 ‘t2 -> ‘t0 ‘t2)
popop    : (‘t0 ‘t1 ‘t2 -> ‘t2)
dupd     : (‘t0 ‘t1 ‘t2 -> ‘t0 ‘t1 ‘t1 ‘t2)
swapd    : (‘t0 ‘t1 ‘t2 ‘t3 -> ‘t0 ‘t2 ‘t1 ‘t3)
rollup   : (‘t0 ‘t1 ‘t2 ‘t3 -> ‘t1 ‘t2 ‘t0 ‘t3)
rolldown : (‘t0 ‘t1 ‘t2 ‘t3 -> ‘t2 ‘t0 ‘t1 ‘t3)
```

Expressions are Functions

All instructions in Cat are functions which take a stack as input, and return a new stack as output. In fact all expressions in Cat are functions, including quotations. Two instructions side by side effectively are implicitly the composition of the two stack transformation operations.

Quotations are an example of a higher-order function: it pushes an expression (a stack transformation function) onto the stack.

The Type System of Cat

Cat supports higher-rank parametric polymorphism without recursive types and is fully inferred without requiring any user annotation.

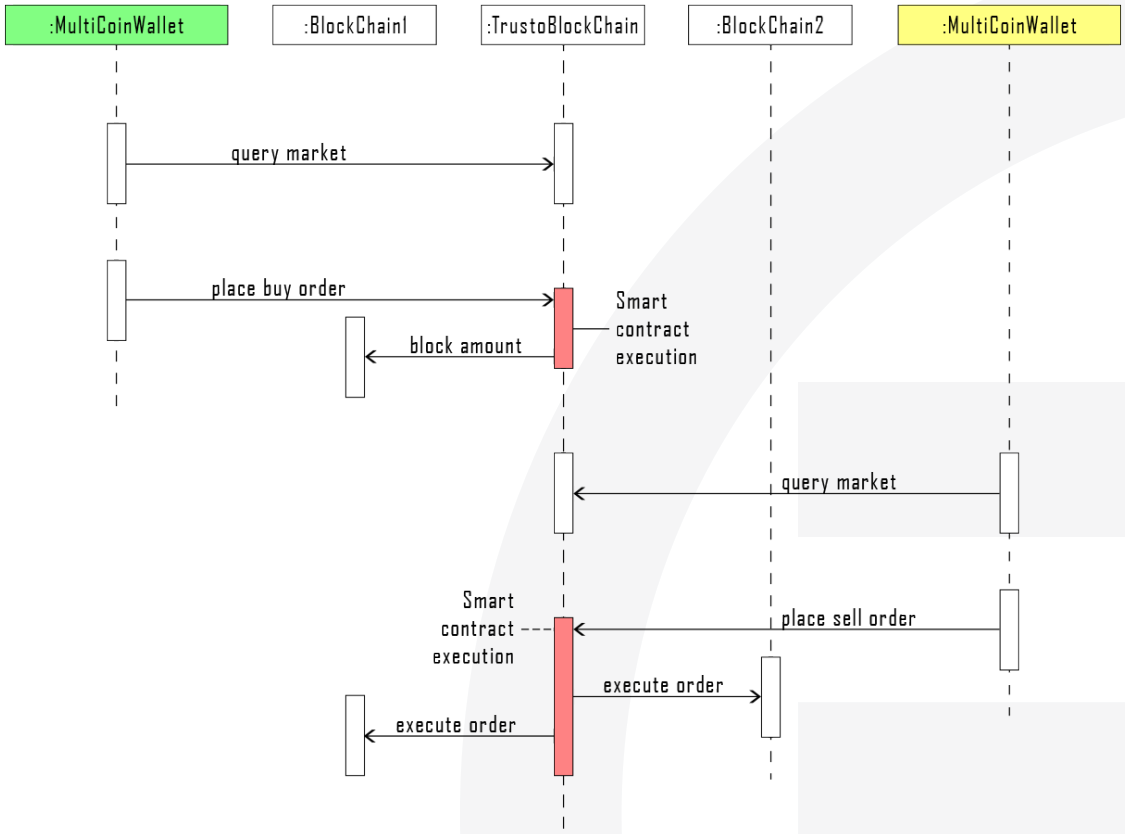
Types describe the effect of a function on a stack. Every function requires a well-typed stack and generates a well-typed stack of a particular configuration. The type configuration of the output stack can always be statically determined based on the types of the input stack.

The type inference module is a reimplementation of cat’s type inference module in Haskell and adds to the original implementation the type dependent feature. Thus the contracts written with this language are provable correct, and the language extensions supports assertions that are automatically proven correct by the included theorem prover system.

The Trusto runtime system

The runtime system of Trusto is an extensible runt time system that provides services to the programs written in Cat to interface with the world. It supports services to access data in the Trusto blockchain, in other blockchains like Bitcoin or Ethereum and to express side effects in a functional and safe way that assures a side effect is executed exactly once, no matter how many times the expression is evaluated, as long it is evaluated at least once.

The runtime system is a library of modules of functions. The modules are identified by their hash and are stored in the document block chain and signed by the network users. A new addition to the library will only be valid if a majority of the network votes and sign that extension.



100% descentralized crypto currency exchange

The Trusto smart contracts, through its extensible runtime system is supporting autonomous distributed applications running in the blockchain. As an example is the 100% descentralized crypto currency exchange application that runs completely in the blockchain and is able to support all the existent crypto currencies, as well as future cryptocurrencies.
Another application would be supporting the creation of other tockens on the Trusto blockchain, similar to ERC20 Tokens on Ethereum network.

Trust level

The trust level is a value between 0 and 1. Trust value of 0 means 0% trust, completely untrusted account, where a value of 1 means 100% trusted account. The trust level is associated with an address, also known as account, in the Trusto network.