IOTA Smart Contracts
Architecture description

(ver 3, Nectar release)

Change log

When	Who	Change
Oct 2, 2020	E.Drąsutis	Initial version
Oct 31, 2020 - Feb 22, 2021	E.Drąsutis	Ver 2
Apr 14-21, 2021	E.Drąsutis	Ver. 3.0
Apr 28, 2021	J.Silva	Numerous edits

Table of contents

General

This document describes the architecture of IOTA Smart Contracts, the IOTA Smart Contract Protocol (ISCP).

Version 3 of the architecture document is a result of learning from implementing the Pollen alpha release of ISCP. It was significantly adjusted to reflect fundamental changes in the *UTXO ledger* model used by the Goshimmer node, the prototype implementation of IOTA 2.0 DLT. We believe in ver 3, the protocol acquires its final form and is solid enough to be the basis for highly scalable multi-chain DLT environments.

IOTA Smart Contracts are implemented in the Wasp node software. This document includes implementation details of some features. Implementations will likely change in the future, due to development of Wasp itself and because of the development of Goshimmer. We are also investigating the possibility of building an ISCP for Chrysalis protocol for the current mainnet.

This document is intended for knowledge transfer within the IOTA Foundation and committed parties.

Fundamentals and goals

ISCP is a further development of the IOTA Tangle protocol: a scalable, parallelizable, feeless (miner-less) DLT. What we want to achieve is the following: to enable trustless and distributed computation on the IOTA ledger, the programmability of it. By programmability of the distributed ledger we mean ability to extend distributed ledger state transition function with external programs. This in short is what smart contracts are.

The smart contract essentially is a state machine which takes state from the ledger and produces the state update to it. The fundamental requirement for smart contracts is an *objective state*, which is equally perceived by all smart contracts. It can only be achieved by serializing state updates to the ledger, the *total order* of events. It immediately makes the *blockchain*, a totally ordered chain of ledger updates, a natural basis for smart contracts. Blockchain DLT, as a distributed yet objective state, and smart contracts are inherently interrelated.
It also introduces the fundamental bottleneck of blockchain-based DLTs into smart contracts: the very limited scalability. In simple terms, you can’t run parallel smart contracts and parallel state updates on blockchain.

IOTA ledger overcomes scalability problems by running the ledger on a DAG (not on a chain), the structure which is not totally ordered. IOTA ledger mimics properties of the real world with fundamentally limited speed of interaction between parts of it. This way IOTA achieves scalability through the ability to run parallel state updates to the ledger which eventually converges into the global consensus. The Tangle DAG approach to global permissionless consensus is fundamentally different from classical Nakamoto consensus in blockchains while the very function of permissionless distributed ledger remains the same, just like in blockchain-based DLTs like Bitcoin and Ethereum. Same time, absence of total order and objective momental state makes the IOTA DLT very different when it comes to the pluggable Turing-complete programmability of the ledger.

With ISCP, we aim to develop a distributed multi-(block)chain smart contract environment on top of the IOTA ledger, the Tangle. The environment will be able to run many distributed smart contract chains (aka contract chains or just chains) **in parallel**.

Each ISCP chain is run under the consensus of its *validators*, so each chain is a *distributed system*, the system without a single point of failure/trust.

Each chain is able to host many smart contracts, sharing the same objective state provided by the chain. Each smart contract thus represents a decentralized and fault tolerant state machine which keeps its immutable state in the chain. Those smart contracts are able to manipulate assets on the IOTA ledger according to its algorithms. The smart contracts also are able to exchange native IOTA assets, i.e. to transact cross-chain, in a trustless manner.

On the functional side, we take classical Ethereum smart contracts as a benchmark, i.e. we aim to create at least a functionally equivalent smart contract platform at the same time improving its properties, especially scalability and cross-chain communications.

It is important to note that the ISCP design “on the layer 2” (see below) has a very different motivation from layer 2 designs of the Ethereum 2.0 (rollups). **The ISCP does not aim to solve scalability problems inherent to blockchains. The underlying layer of ISCP, the Tangle, is already scalable**, because the Tangle ledger allows parallel state updates (unlike blockchain)**.** ISCP is a natural development of the Tangle protocol to enable parallelizable smart contracts.

In this version of the document we rely on the base protocol version implemented in the experimental Pollen network since release 0.N and ongoing Nectar release.

Some concepts and names

IOTA Smart Contracts (ISC) is a protocol to run *distributed programs* - in short: *smart contract(s)* (SC, SCs). Hence IOTA Smart Contract Protocol (ISCP).

ISCP is built on top of the *IOTA UTXO ledger* (previously called *Value Tangle*, now this name is deprecated). *UTXO ledger* is a protocol responsible for the global and distributed *IOTA token ledger*. By saying the ISCP is *level 2* (*L2)* protocol we mean each smart contract chain implements its own distributed ledger on top of the *UTXO ledger*, i.e. on the level 2. We will sometimes call the *UTXO ledger* the *L1 ledger* or just L1.

We distinguish the *UTXO Ledger* (a ledger of transactions), from the Tangle the underlying DAG structure, voting and consensus mechanism), similar to the way that Bitcoin blockchain and ledger of transactions are different things. We define ISCP concepts on top of the *UTXO ledger* concepts and thus we keep ISCP largely independent from the intricacies of the underlying Tangle protocol.

The *UTXO Ledger* contains *value transactions.* It implements a distributed ledger of *tagged tokens* (a.k.a. *tokens*, *colored tokens*, *colored coins*, *digital assets*, *tokenized assets*). IOTA tokens are tokens with default color: *iota-color*.

Instances of ISCP programs, i.e. programs with data (the state) are called *smart contracts*. Smart contracts are event-driven distributed software agents. They are *distributed* because the program is run not by one, but by a set of processors, in a distributed manner under consensus.

Smart contracts can implement any algorithm, hence they are *Turing complete*. They can manipulate digital assets and move them between Tangle addresses on the *UTXO ledger* and on-chain (L2) accounts according to their algorithms. The contract state is anchored on the distributed ledger of the IOTA Tangle and is tamper-proof.

All smart contracts are run on **smart contract chains** (contract chain, chain). A contract chain can contain many smart contracts. The contract chain is functionally equivalent to a blockchain such as an Ethereum blockchain or a *parachain* in Polkadot. All smart contracts deployed on one chain share its global state and as such, they can interact (call each other) in the context of the same state.

The smart contract chain defines an immutable (append only) state anchored on the base layer.

Each chain is run by a network of **validator nodes**, nodes which run a consensus on the chain state updates. The *Wasp node* is an implementation of the *validator node*. The *validators* are the owners of the *validator nodes* but in most contexts the difference is not important. The *validators* of the chain form a *committee*, a bound together closed set of nodes. The *committee* of the chain may change. This way new *validators* and *validator nodes* may be added or replaced. This also makes the **chain itself agnostic to its validators** (the *committee)*.

The smart contracts on different contract chains can interact and transact between each other in a trustless manner by sending an **on-ledger request**, a transaction, with attached data and funds on the underlying *UTXO ledger*. The *on-ledger* requests are global entities, they are sent to the IOTA ledger directly and they will find and reach specific chains independently of its validator nodes.

We also introduce **off-ledger requests** which are sent directly to the committee or access nodes instead of being sent to the UTXO ledger as confirmed transactions. The *off-ledger requests* allow high-TPS of SC transactions processed by the chain.

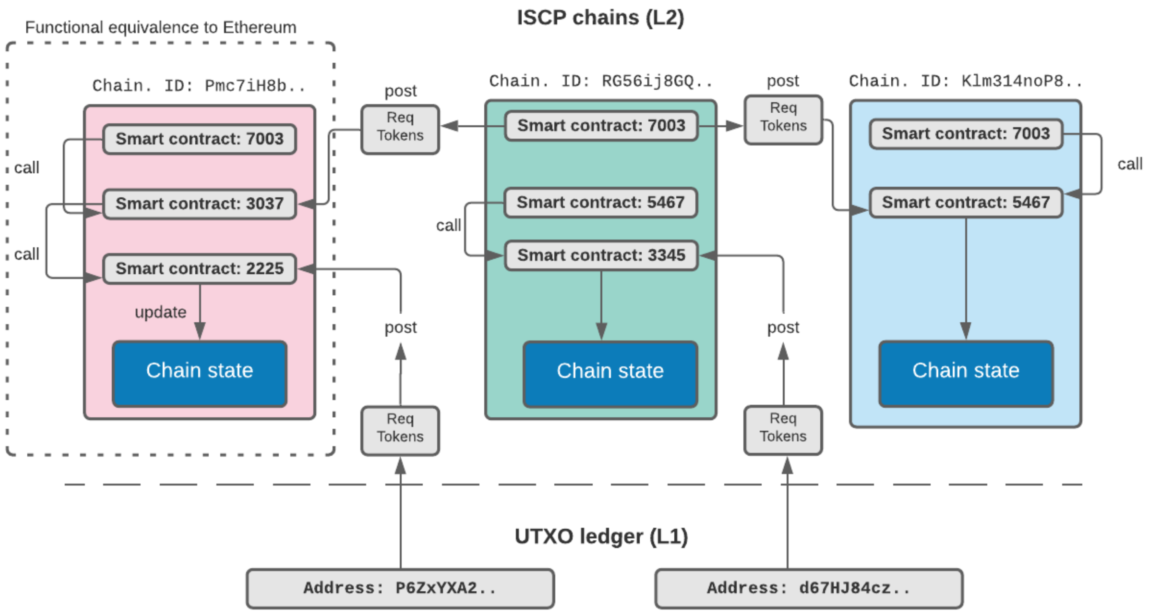
The ISCP is implemented as a network of *Wasp* nodes. Smart contracts are run on the network of *Wasp* nodes. Each *Wasp* node connects to the *UTXO ledger* via Goshimmer nodes.

Logical structure of the ISCP chain

Chains

The ISCP is a multi-chain environment.
We can run many parallel blockchains on the IOTA ledger:

- Each having own state, which can be updated in parallel
- Each state anchored on the UTXO ledger, the L1
- Each validated by a set of validators, the committee
- Each can contain multiple smart contracts
- Each smart contract is enabled to transact (exchange assets) with other smart contracts on other chains in a trustless (distributed) manner



State of the chain

The state of the chain consists of:

- Balances of the native IOTA digital assets, colored tokens. The chain acts as a custodian for those funds
- A collection of arbitrary key/value pairs, the *data state*, which represents use case-specific data stored in the chain by its smart contracts outside of the *UTXO ledger*.

The state of the chain is append-only (immutable) data structure maintained by the distributed consensus of its validators.

The chain also contains an on-ledger backlog of yet unprocessed requests.

Anchoring the state

The *data state* is stored outside of the ledger, on the distributed database maintained by validators nodes.

By *anchoring the state* we mean placing the hash of the *data state* into one special transaction and one special UTXO (an output) and adding it (confirming) on the *UTXO ledger*.

The *UTXO ledger* guarantees every moment there's **exactly one** such output for each chain on the *UTXO ledger*. We call the output the **state output** (or *state anchor*) and the containing transaction **state transaction** (or *anchor transaction*) of the chain.

The *state output* is controlled (i.e. can be unlocked/consumed) by the entity running the chain (for details on UTXO machinery *and* the *committee* see below).

With the anchoring mechanism the *UTXO ledger* supports the ISCP chain the following way:

- guarantees global consensus on the state of the chain
- makes the state immutable and tamper-proof
- makes the state consistent (see below)

The *state output* contains:

- Identity of the chain (*alias address*)
- Hash of the *data state*
- *State index*, which is incremented with each next *state output*, the state transition (see below)

Digital assets on the chain

The native L1 accounts of IOTA *UTXO ledger* are represented by addresses, each controlled by the entity holding the corresponding private/public key pair. The L1 account is a collection of UTXOs belonging to the address.

Similarly, the chain holds all tokens entrusted to it in one special UTXO, the *state output* (see above) which is always located in the address controlled by the chain.
It is similar to how a bank holds all deposits in its vault. This way, the chain (entity controlling the *state output*) becomes a custodian for the assets owned by its clients, in the same sense the bank's client owns the money deposited in the bank.

We call the consolidated assets held in the chain "*total assets on-chain", which* are contained in one and the only *state output* of the chain.

The data state

The *data state* of the chain consists of the collection of key/value pairs. Each key and each value are arbitrary byte arrays.

In its persistent form, the *data state* is stored in the key/value database outside of the UTXO ledger and maintained by the validator nodes of the chain.
The *data state* stored in the database is called a *solid state*.
By *virtual state* we mean in-memory collections of key/value pairs which can become solid upon being committed to the database. The essential property of the *virtual state* is the possibility to have several virtual states in parallel as candidates, with a possibility for one of them to be solidified.

The key/value pairs of the *data state* are used to implement all kinds of on-chain concepts: from on-chain accounts (see below) to use-case specific data of smart contract state.

The *data state* in any form has *state hash, timestamp* and *state index*.

The *data state* of the chain has its hash value or *state hash*, usually the Merkle root but it can be any hashing function of all data in the *data state*.

The *state hash* of the chain is always stored in the *state output*, the same UTXO which contains the total digital assets deposited to the chain. This makes *data state* and the *on-chain assets* both contained in one atomic unit, the *state output*, on the ledger and therefore consistent. It is only possible to change the state hash by the same entity which is controlling the funds (the *committee*). So, the state mutation (state transition) of the chain is an atomic event between funds and the data state.

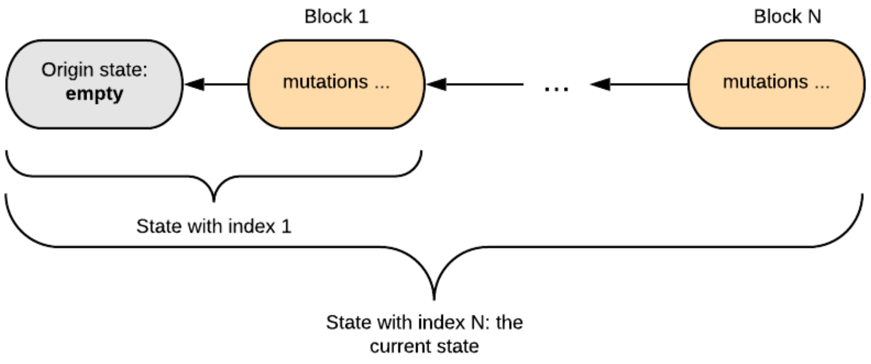
The blocks and state transitions

Data state is updated by mutations of its key value pairs. Each mutation is either setting a value of the key, or deleting the key with the value from the *data state*. Any update to the *data state* can be reduced to the partially ordered sequence of mutations. The state is *updated* (mutated, modified) by *applying* the mutations to the current *data state*.

The set of mutations to the *data state* which is applied in one piece we call a **block**:
$next\ data\ state\ = apply(current\ data\ state,\ block)$

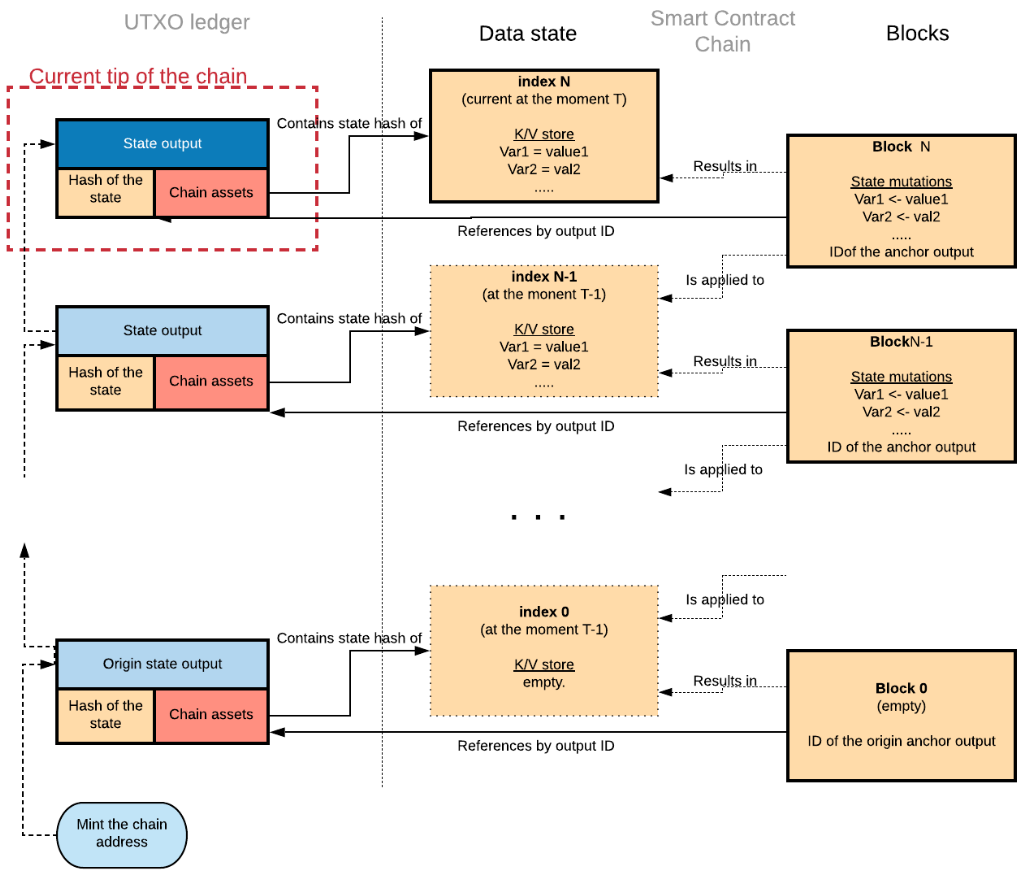
The **state transition** in the chain occurs atomically together with the movement of the chain's assets and the update of the *state hash* to the hash of the new *data state* in the transaction which consumes previous *state output* and produces next *state output*.

At any moment of time, the *data state* of the chain is a result of applying the historical sequence of blocks, starting from the empty data state. Hence, *blockchain*.



On the *UTXO ledger*, the L1, the history of the state is represented as a sequence (chain) of UTXOs each holding chain's assets in a particular state and the anchoring hash of the *data state* (see details below). Note that not all of the state transitions history may be available: due to practical reasons the older transaction may be pruned in the *snapshot* process. The only thing that is guaranteed: the tip of the chain of UTXOs is always available (which includes the latest data state).

The blocks and state outputs which anchor the state are computed by the Virtual Machine (VM), a deterministic processor, a "black box". The VM is responsible for the consistency of state transition and the state itself.



Synchronizing ('synching') of state between nodes

Any party, for example a validator node, that has access to the chain of blocks and the *UTXO ledger* can reconstruct the valid *data state* of the chain. Each party keeps its own copy of the *data state* as the *solid state*, in the database.

The process of keeping the copy of the state valid and up-to-date (i.e. latest) is called *syncing*. The latest or current state is determined by the one and the only *state output* of the chain on the *UTXO ledger*.

The party can query the *UTXO ledger* at any time and retrieve the current *state output* of the chain. If the *state index* and *state hash* contained in the *state output* is equal to the *state index* and the *state hash* of the current *solid state*, it means the party has up-to-date state in its database, i.e. it is *synced*.

If the *state index* on the *solid state* is less than the *state index* of the *state output*, it means the party is behind i.e. *not synced*.

Let's say some party is left behind and the *solid state* it has in its database has index L while the current *state index* is N > L.

The synching party does the following in a loop while L < N:

- The party asks for the block with the index L+1 from nodes. Some of these have that block and send it to the requesting party.
- The syncing party takes the ID of the anchor output from the block received upon request and asks the UTXO ledger for that confirmed transaction containing that output.
- The UTXO ledger sends the output to the node. It contains the hash of the data state L+1. The party does the following:
    a) applies the block (a sequence of mutations) to the current *solid state* L and thus creates a resulting *tentative state.*
    b) takes hash of the *tentative state*
    c) compares state hash of the *tentative state* with the *state hash* stored in the output
    d) If the two hashes are equal, the node stores a *tentative state* and the block it receives into the database as the *solid state* L+1.
    If these hashes are not equal, the database is inconsistent or the received block is wrong, i.e. sent from a malicious node. Synchronisation is not possible and is restarted with another node.
- Now the party has valid solid state L+1. If it is not the current state, it repeats the process until it is.

In this way any party (usually another node) that is entitled to receive requested blocks from other nodes can keep its state valid and in sync with the chain even without being able to participate in the calculation of new state updates.

Nodes which only synchronize their states without calculating them are called *access nodes*. They can have many functions, like providing the public network access to the chain state, while at the same time shielding non-public *committee nodes* from possible DDoS and other attacks.

Smart contract state

The *data state* of the chain is split into partitions, the sub-states, each controlled and updated by a particular smart contract. Key/value pairs of the smart contract's partition have keys prefixed with 4 bytes of ID of the smart contract on the chain, the so-called *hname* of the smart contract.

The smart contract program can only update the key value pairs of its own partition. The virtual machine (VM) ensures (via the *sandbox* interface) the partition can only be accessed by the smart contract controlling it.

On-chain accounts

To distinguish between different owners of tokens deposited to the chain, the chain keeps a ledger of *on-chain accounts* in the *data state* of the chain. It is analogous to a bank accounting of the holdings deposited by its clients.

The ledger of on-chain accounts is a collection of $accountID : token\ balances$ pairs.

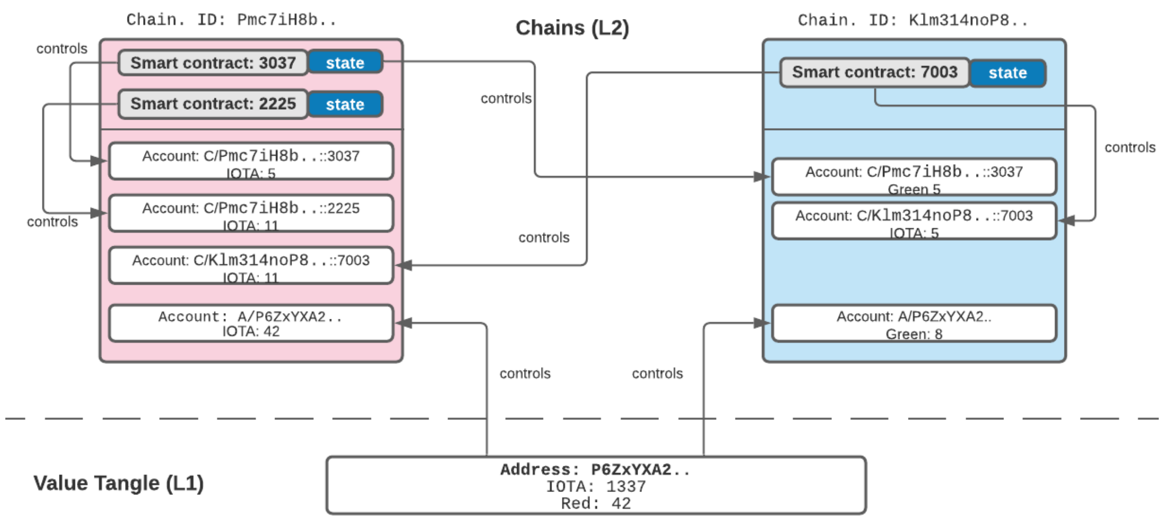
Any movement of tokens between accounts on the chain is reflected by changes of their respective account balances. Any deposit/withdrawal of funds to/from the chain always means credit/debit to/from the respective accounts and is accounted for in the balances of the accounts of that chain's ledger.

Each smart contract deployed on the chain automatically acquires an on-chain account with the *accountID* derived from the ID of the smart contract. This account contains tokens which can only be moved by the smart contract. This way, each smart contract controls its data sub-state plus all tokens in its on-chain account.

In addition, any entity which is securely identified on the UTXO ledger, i.e. the one represented by an address, can securely keep its tokens on any of ISCP chains. For example, the usual L1 wallet with an address can securely deposit and withdraw tokens from its on-chain account (with *accountID* derived from the address) by sending special requests to the core contracts of the chain.

The VM logic guarantees the main invariant of the on-chain account ledger: with any state transition the ***total assets*** **on the chain are always equal to the balances of colored tokens locked in *state output* of the chain**.

The system of on-chain accounts means that the L2 chain uses the **account-based ledger model** as opposed to the UTXO ledger model used by the L1 ledger


Chains (L2)
Chain. ID: Pmc7iH8b..
Smart contract: 3037
state
Smart contract: 2225
state
Account: C/Pmc7iH8b..::3037 IOTA: 5
Account: C/Pmc7iH8b..::2225 IOTA: 11
Account: C/Klm314noP8..::7003 IOTA: 11
Account: A/P6ZxYXA2.. IOTA: 42
Chain. ID: Klm314noP8..
Smart contract: 7003
state
Account: C/Pmc7iH8b..::3037 Green 5
Account: C/Klm314noP8..::7003 IOTA: 5
Account: A/P6ZxYXA2.. Green: 8
controls
Value Tangle (L1)
Address: P6ZxYXA2.. IOTA: 1337 Red: 42

Identification of on-chain accounts

The *accountID* is always derived from the controlling entity of the account.
The special type *AgentID* is used for account identifiers.

AgentID has a form of *(address, hname)*, where *address* is some address (one of 3 types currently supported: ED25519 address, BLS address and alias address) and *hname* is any *uint32* value, usually first 4 bytes of *blake2b* hash of some name, interpreted as little-endian *uint32*. So, *AgentID* is some *address* and some *hname*.

If the controlling entity is an usual cryptography-based *address* like ED25519 or BLS, the *AgentID* of the account is *(address, 0)*.

If the controlling entity of the account is a smart contract on chain with *alias address (a special type of address, the chain ID)* A and *hname* H, the *AgentID of the account is (A, H). Common (*or *default)* account on each chain has *AgentID* (A, 0).

The combination of (A, H) is not valid if H != 0 and A is not of *alias address* type.

This way the *AgentID* contains information about the type of the controlling entity: whether it is an L1 address (usually a wallet address) or a smart contract on a particular chain.

UTXO ledger

ISCP uses the *UTXO ledger* to implement the logical structure of chains described above.

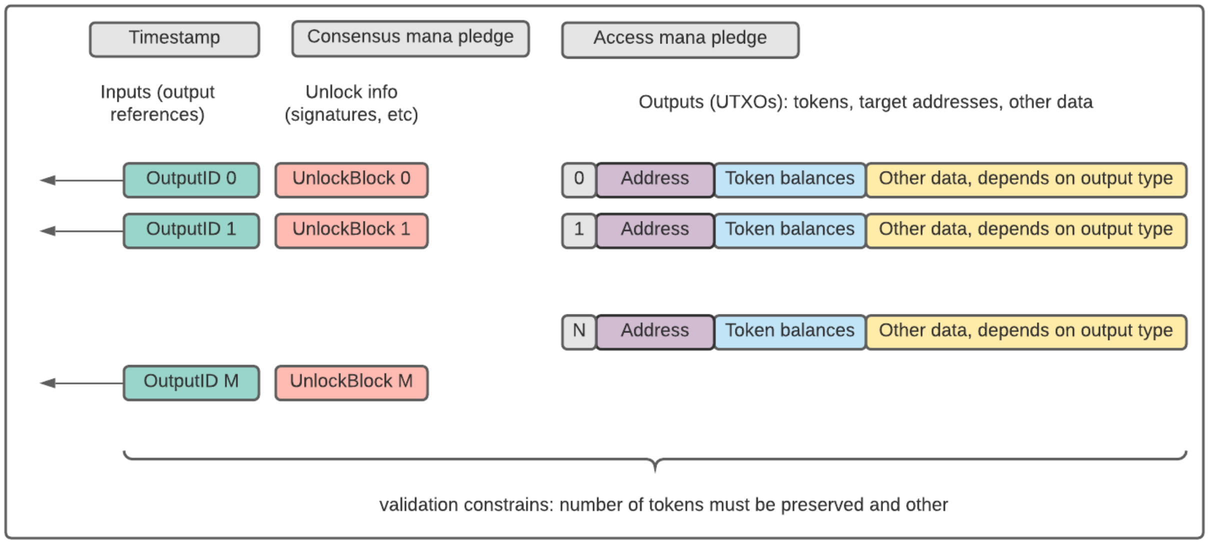
Here we describe the *UTXO ledger* as it is implemented in the Goshimmer starting from breaking release 0.4.0. The general design of *UTXO ledger* follows specifications of Chrysalis mainnet and the generally accepted principles of UTXO. The level of detail described here is enough to understand ISCP architecture (for more in-depth information about the UTXO ledger, please see "<link to the goshimmer docs?>").

Transaction

A *transaction* consists of:

- Ordered list of **inputs**. Each input is a reference to the consumed output
- Ordered list of **outputs**
- Ordered list of **unlocks block**, one unlock block for each input
- Timestamp
- Mana pledge targets

UTXO transaction: generic structure


Timestamp
Consensus mana pledge
Access mana pledge
Inputs (output references)
Unlock info (signatures, etc)
Outputs (UTXOs): tokens, target addresses, other data
OutputID 0
OutputID 1
OutputID M
UnlockBlock 0
UnlockBlock 1
UnlockBlock M
0
1
N
Address
Token balances
Other data, depends on output type
validation constrains: number of tokens must be preserved and other

The *UTXO ledger* contains transactions itself and outputs contained in those transactions.

A transaction can be present in the *UTXO ledger* if and only if all the following conditions are satisfied:

- Each of its input references an existing output on the ledger. We say the transaction *consumes* or *spends* the output
- No more than 1 transaction can consume an output. (We do not consider parallel ledger states (aka parallel realities) in the context of ISCP).
  From the point of view of ISCP each output is either *spent* (exactly by 1 transaction) or otherwise is *unspent* (an UTXO).
- The transaction itself is valid according to the validation rules (constraints) brought by its outputs, inputs and other elements (see below). The transaction describes a transformation of the ledger state by consuming inputs and producing new unspent outputs. The transformation is a subject of ledger constraints. The validity constraints of the transition are designed so that adding a transaction to the ledger preserves certain invariants of the *UTXO ledger*.

One of the validation constraints of the transaction enforced by the UTXO ledger is that the *timestamp* of each transaction must be strictly larger than timestamps of transactions of any of its inputs.

Ledger state. UTXO DAG

The UTXO ledger consists of all transactions added to it. Due to the nature of transactions consuming unspent outputs, the *UTXO ledger* is a Directed Acyclic Graph, the UTXO DAG.

The *ledger state* consists of unspent outputs or UTXOs (**U**nspent **T**ransa**X**ion **O**utput).

By adding a (valid) transaction to the ledger some outputs are consumed as inputs and new UTXOs (unspent outputs) appear. This way adding a transaction to the ledger is a **transition of the ledger state**.

Validity of transaction guarantees consistency of the ledger state any moment of time.

Note that the *ledger state* always exists and is consistent. Meanwhile, older transactions in the UTXO DAG with all outputs consumed, can be pruned in the *snapshotting* process, due to practical limits of the database. Speaking from the practical perspective, only transactions containing UTXOs (unspent outputs) are guaranteed to be present in the ledger database at any moment of time. Once all outputs are spent, there’s no guarantee the transaction is present in the ledger database.

Types of outputs. Validity of the transaction

The UTXO ledger can support many types of outputs. Each type of output brings its own type of validation constraints (ledger constraints) into the transaction.

The output types and validation constraints are hardcoded into the ledger logic. Some of this constraints may be very simple, others very complex. Many unlocking options and validation logic may even be described by some script which is part of the output.

All output types share some **common properties and validation logic** which is independent of the type of the output:

- *Output ID*. It is a concatenation (34 bytes) of the containing transaction ID (32 bytes) and index of the output in the transaction (2 bytes): *transaction ID || output index*
- *Token balances*. The digital assets attached to the output. See Tokens of the output.
- *Address*: an identifier of the entity which can *spend* (*consume*) the output. Often it is an hash of the public key of the account (with private/public key pair behind), however in the context of ISCP we can use *alias addresses* (see below)

Tokens of the output

Output must contain at least one non-zero colored balance, i.e. all UTXO transactions are value transfers.

The *colored balance* is a pair $color : balance, where\ balance > 0$. The $color$ is a 32-byte code assigned by the *color minting process* in the origin transaction. In short: the $color$ is $blake2b$ hash of the $outputID$ of the minting output (we skip details here).

Each output contains a collection of colored balances: one (non-zero) balance for one color code.

The following *token ledger constraint* is always applied to the transaction, independently on types of outputs:

- Total number of tokens in outputs, referenced by inputs of the transaction must be equal to the number of tokens in outputs produced by the transaction
- Number of tokens of specific color not equal to the *iota-color* in outputs cannot be larger than the number of tokens of the same color in outputs referenced (consumed) by inputs.

These validation constraints guarantees ledger invariants independently of types of outputs (other constraints, supported by various types of outputs):

- Constant total supply of tokens on the ledger
- Does not allow to inflate colored supplies, once minted, number of the tokens with specific color stay the same or otherwise is uncolored back to $iota - color$.

Unlocking the inputs

Each input in a transaction is a reference to the unspent output of another transaction that is being spent.
For a transaction to be valid, each input must be unlocked by the *unlock block*. There are as many unlock blocks in the transaction as the number of inputs.

The main type of unlock block is a *signature unlock block*. It can be used to unlock inputs which reference outputs with addresses representing a public/private key pair. The signature unlock block contains a digital signature (ED25519 or BLS) of the transaction essence: inputs, outputs, timestamp and mana pledge.

Many inputs referencing UTXOs owned by the same address may be unlocked by providing only one *signature unlock block* in the transaction and *referencing unlock blocks* which reference the unlocking signature. This makes it possible to unlock many inputs with a single signature.

A special type of unlocking block must be used to unlock UTXOs owned by *alias addresses* (see below).

Types of outputs

The Goshimmer implements 4 types of outputs:

- *SigLockedSingleOutput*. An output with iota-only tokens. Not used in ISCP
- *SigLockedColoredOutput*. An output with any colored balances. Not used in ISCP
- *AliasOutput*. ISCP uses it to build smart contract chains and ensure consistency of state updates. See details below
- *ExtendedLockedOutput.* ISCP uses it to represent on-ledger requests between chains

Implementation of chains on UTXO ledger

ISCP allows many parallel chains on the UTXO ledger, each with its own identity and state. The UTXO ledger on L1 supports the logical structure and anchoring of ISCP chains with its output types and ledger constraints brought by them.

Representation of the chain on the UTXO ledger. Chain ID

The chain on the UTXO ledger is identified by a special type of address called *alias address*. It is the address of the chain, also known as *chain address* or *chain ID*. The chain address is a permanent identifier of the chain for its lifetime. It cannot be changed.

The *chain ID*, being an address, also fixes the “location” of the chain on the ledger: it can always be located on the ledger by querying balances of the *alias address*.

Each chain is represented on the UTXO ledger by the globally unique output of the *AliasOutput* type with the *alias address* equal to the *chain ID*. This type of output guarantees the single output with a particular address in the ledger state.

AliasOutput type

This type of output was designed to support requirements to the UTXO ledger from ISCP. However, constraints and concepts brought by *AliasAddress* are universal and can be used in other use cases. Here we describe only a part of the options of *AliasOutput.*
The requirements are:

- Implement *chain constraint* type on the ledger, ability to support non-forkable chains of transactions
- Implement *address aliasing:* to decouple *chain ID* from the address controlled by the private/public key pair. The requirement comes from the fact that the controlling entity of the chain, usually represented by the address (the hash of the public key) should be able to be changed without changing the identity of the chain. It won't be possible if *chain ID* would be based on the controlling address. (This enables the chain committee to change members, while keeping the same *chain ID*).

AliasOutputs and *alias address* together with *ExtendedLockedOutput* brings flexibility to the ledger even outside ISCP scope, for example makes it possible to have wallet accounts which can change controlling keys without changing the address.

Let's describe universal properties of the *AliasOutput* and how these properties are used in ISCP. In the description below we present a simplified version of the *AliasOutput*, skipping some optional features (governance, delegation) of the real *AliasOutput* which are implemented in the current versions of Goshimmer but not used in ISCP.

Like any other output, *AliasOutput* contains $outputID$, *token balances* and *the address*. It means the *AliasOutput* is just a normal output residing in the transaction and in the address and is subjected to the token constraints of the ledger.

In addition, each *AliasOutput* contains these data elements:
- *Alias address* - a special type of address
- *State address* - any type of address
- *Immutable data* - a byte array
- *State data* - a byte array

Alias address. Immutable data

Alias address is the address of the *AliasOutput* type of output. It is a special type of value which has the same“rights” as any other address in the UTXO ledger. However, the *alias address* is not backed by the private/public key pair like “conventional” addresses, for example based on ED25519 elliptic curve cryptography. Instead, the *alias address* is a *minted* value, i.e. it appears in the ledger as a result of the “minting” action. The value of the *alias address* cannot be neither chosen or predicted nor minted twice.

The *alias address* is minted in the *origin* output, the output of *AliasOutput* type which starts the chain. “Minting” means taking the $blake2b$ hash of the $outputID$ of the *origin* alias output. Note that the $outputID$ contains the hash of the transaction itself. This makes the value of the *alias address* impossible to predict, it also is impossible to mint the same in another transaction (assuming $blake2b$ hash function is practically collision-free).

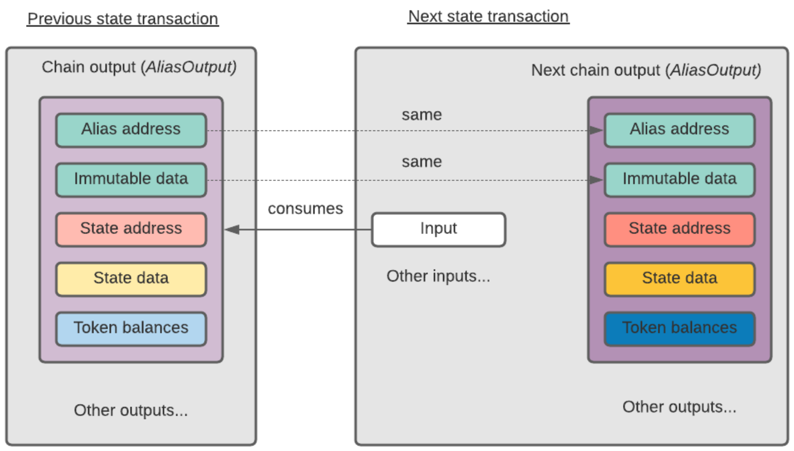
Alias address is the identity of the chain of outputs. The minting of the alias address is the same process as minting a new color for the tokens. In ISCP the *alias address* has a special function: it is used as a unique identifier of the smart contract chain, the $chainID$.

Consuming the alias output. Building a chain

Alias output can only be consumed (with few exceptions, see below) in the transaction which contains **exactly one** *alias output* with the **same *alias address* and same *immutable data***. It means that alias outputs form non-forkable chains of UTXOs on the UTXO ledger.

The tokens balances, state data and state address can be different in the next alias output in the chain:

- Token balances may change when tokens are consumed from several inputs and distributed among several outputs
- State data chain for example when it reflects change in the anchored *data state* (the hash of it)
- Change of the *state address* has a special meaning. See Unlocking the AliasOutput. State address.



So, usually the transaction which represents the state transition contains one alias output as inputs and produces another alias output with the same immutable part as output. Hence, the chain.

The exceptions are:

- If the output of the type *AliasOutput* doesn't have corresponding input, it is considered an *origin*. It mints the *alias address* and sets the *immutable data*. The value of the *alias address* is set equal to the $blake2b$ hash of the $outputID$ of the *origin* output.
  So, the *alias address* is created by the *origin* alias output.
- If the alias output is used as an input and it doesn't have corresponding output, it is being destroyed. Special conditions must be met to be able to destroy the alias output.

The output of the transaction can contain different *state data* and a different *state address*, (but can't change *alias address* and immutable data), that would be violation of the validity constraint.

Unlocking the AliasOutput. State address

AliasOutput consumed as input of the transaction is unlocked by the usual *SignatureUnlockBlock* which corresponds to the *state address.*
It means, the entity which controls the private key behind the *state address* can create the next alias output. It is the *controlling entity* which controls the tokens, anchored state and its transition.
The controlling entity is able to change the controlling address itself.
We also call the *state address a controlling address*.

The meaning of changing the *state address* in the next alias output means a change of the controlling entity. In the context of ISCP it means **changing the committee of validators** to the new one. Note, that change of committee of validators does not change the chain's identity, state nor on-ledger backlog. It makes possible seamless transition to a new committee, the *dynamic committees* feature.

Properties of the AliasOutput

The important properties of the *AliasOutput*:

- The *AliasOutput* and the *alias address* represent non-forkable chain on the UTXO ledger
- The *alias address* is created together with the chain and remains its immutable ID for the lifetime.
- There's exactly 1 unspent alias output for a specific alias address in the ledger state at any moment in time
- The unspent output and the tokens reside in the account represented by the *alias address*. I.e. if you query the ledger for the balances of the specific *alias address* you will always find exactly one alias output with this address.
- The tokens (funds) locked in the alias output are controlled by the entity which controls keys behind the *state address.*
- The controlling entity can change the *state (controlling) address*. In this case the next output in the chain will only be possible to unlock with another private key.

Requests. Backlog of on-ledger requests

In this chapter we will describe only *on-ledger* requests. Another type, the *off-ledger requests,* will be described further in the document.

Any state transition on the chain is triggered by issuing a *request*. A *request* to a smart contract may be seen as an asynchronous (deferred) call to the smart contract entry point.

Each request triggers a specific smart contract entry point on a specific chain. The smart contract (a program) processes the request and produces a state update of the chain.

The structure of the request

The request contains the following properties:
- identification of the sender
- Target:
    - chain
    - smart contract on the chain
    - entry point of the smart contract
- Attached tokens (digital assets) which are transferred with the request
- Parameters of the request: a collection of key/value pairs

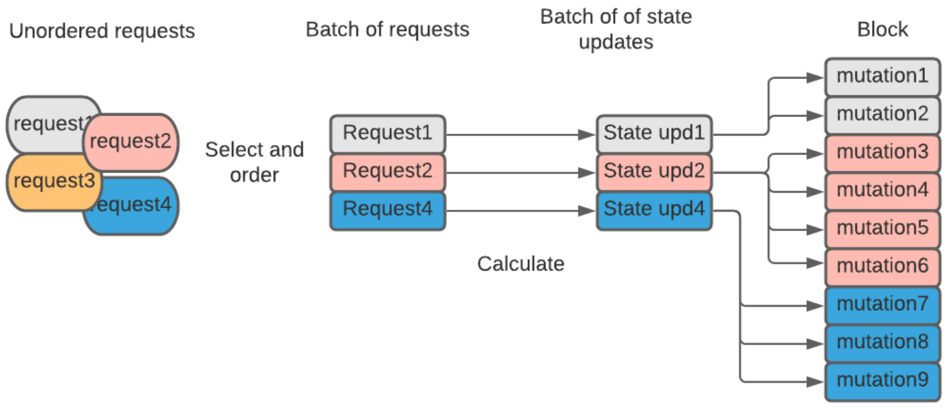
The *on-ledger* request is an inter-chain entity, it is used for cross-chain transactions (asset transfers) and for sending tokens to smart contracts by ordinary wallets which own tokens on a regular address on the UTXO ledger. The *off-ledger* requests are processed differently.

The VM takes the request, the SC program and the current data state and produces a state update:

$$state\ update \leftarrow smart\_contract\_program(current\ state,\ request)$$

The state update contains a sequence of mutations to the current *data state*.

The VM is processing requests in batches. One batch of requests produces a sequence of state updates which makes a *block*:



Requirements to request processing

Request is a message with the payload of data and assets. The following requirements seems to be natural:
Guaranteed delivery: the message must always reach the target, i.e. payload can't just disappear due to malfunction or malicious attack

Atomicity of backlog: once the target is reached and request is processed, the request must disappear from the backlog of unprocessed requests in the same (atomic) event as the state update to the chain it produced.

Fallback: if request is not processed until (optional) deadline, the funds attached to the request must be returned to the sender.

The above requirements we call **quasi-atomicity** of request processing. It guarantees that either the request is processed by the target exactly once, or the attached funds are returned back to the sender without any effect on the state of the target.

Requests as UTXOs

In order to achieve desired requirements described above, the *requests* heavily use special UTXO types on the UTXO ledger.

In ISCP, **each on-ledger request is represented by one UTXO** (unspent output) of type *ExtendedLockedOutput*.

The *ExtendedLockedOutput* output type is an extension of the *SigLockedColoredOutput*: the former functionally is a subset of the latter, i.e. *ExtendedLockedOutput* allows us to implement all the behavior of the *SigLockedColoredOutput*.

Naturally, the *ExtendedLockedOutput* has properties common to all types of outputs:

- *OutputID* is treated as a unique request ID
- *Address* is an address of the target chain. So, when output represents a request, it is an *alias address*.
- *Token balances* are digital assets transferred by the request

In addition, *ExtendedLockedOutput* has the following properties:

- *Output payload*: arbitrary binary data.
- *Fallback data* (optional): the *fallback deadline* and the *fallback address*. See Unlocking ExtendedLockedOutput

In ISCP the *output payload* is interpreted as a serialized form of the following request data:

- Target smart contract ID (4 bytes of *hname*)
- Target smart contract entry point ID (4 bytes of *hname* of the name or function signature of the entry point)
- key/value pairs of request parameters.

So, all the request data is contained in the output.
Note that one transaction may contain many outputs which represent a request. It makes it possible to place many requests to different chains and smart contracts atomically in one package.

Unlocking the *ExtendedLockedOutput*

This type of output presents several unlocking options, depending on how the output is constructed. It allows flexible implementation of requirements to requests, it also may be useful in implementing functional extensions of conventional crypto wallets.

If fallback options are not used

Option 1. If the *address* is a conventional ED25519 or BLS address, the output can be unlocked by corresponding signature, i.e. *SignatureUnlockBlock*.

This option corresponds to the simplest scenario also covered by simple *SigLockedColoredOutput*: just sending funds to an normal address.

Option 2. If the *address* is an *alias address*, the output can be unlocked by presence of unlocked *AliasOutput* with the same *alias address* in the same transaction. In this case the unlock block of the *ExtendedLockedoutput* should be a special *unlock block,* the *AliasUnlockBlock*, which is referencing the *AliasOutput*.
This option corresponds to the scenario of sending funds to the *alias address*. This is the form used for requests: the request is a transaction output with the target address the *alias address* of the target chain. The target chain can unlock and consume it in the state transition transaction.

If fallback options are set

If fallback options are set it means *fallback deadline* and *fallback address* are specified in the output. This creates 2 ways of unlocking the output (with one of options above):

- If the deadline is **before or equal** the timestamp of the consuming transaction, the input (the output of the containing transaction) can be unlocked by the main address of the output
- If the deadline is **after** the timestamp of the consuming transaction, the input (the output of the containing transaction) can be unlocked by the *fallback address* of the output

The sender of the output may set *fallback address* and *fallback deadline* to his/her liking.
In practice, the deadline may be say 10 min or 1 hour from now and the fallback address may be the sender's address.
It means that before the deadline only the target address may consume the output. If it fails to consume until deadline, the control of the output falls back to the sender.

Processing the requests. Producing next state output

AliasOutput and *ExtendedLockedOutput* provide powerful L1 support for the ISCP requirements.

Suppose we have a chain C is identified with *alias address* A. It is represented by the only *state output* in address A. The *state output* contains the state of the chain (the anchoring hash of it) and the on-chain assets.

The *on-ledger request* to the chain C is represented by the unspent output of type *ExtendedLockedOutput* with lock (target) address A.

Backlog of requests of the chain C consists of all UTXOs (unspent outputs) of type *ExtendedLockedOutput* in the address A, i.e. unlockable by the address A.

Processing the request is consuming it in the state transition transaction. I.e. in the same transaction where chain is extended with next *state output* (of *AliasOutput type*). The result of processing the request is reflected in the changed *total assets* and changed *state hash* contained in the *state output*.

This way:
Processing the request means it disappears from the backlog (because it is consumed) and it happens in the same transaction as the state transition, i.e. atomically.

The fallback option allows you to take control of funds back in case the *request UTXO* is not processed for some reason, for example all nodes processing the chain are down.
The part of the Wasp node responsible for atomic state transition of the chain is VM. It takes a backlog of requests (the batch of outputs), current *data state* and computes the *block* and the *anchor transaction* of the next state.

$$VM(data\ state,\ request1,\ ...,\ requestN)\ \Rightarrow next\ data\ state,\ anchor\ transaction$$

The *anchor transaction* consumes (unlocks) all UTXOs/requests and this way produces next state in the atomic event which wraps:
- update of *data state*
- movement of assets
- sending requests to other chains
- updating the backlog of on-ledger requests

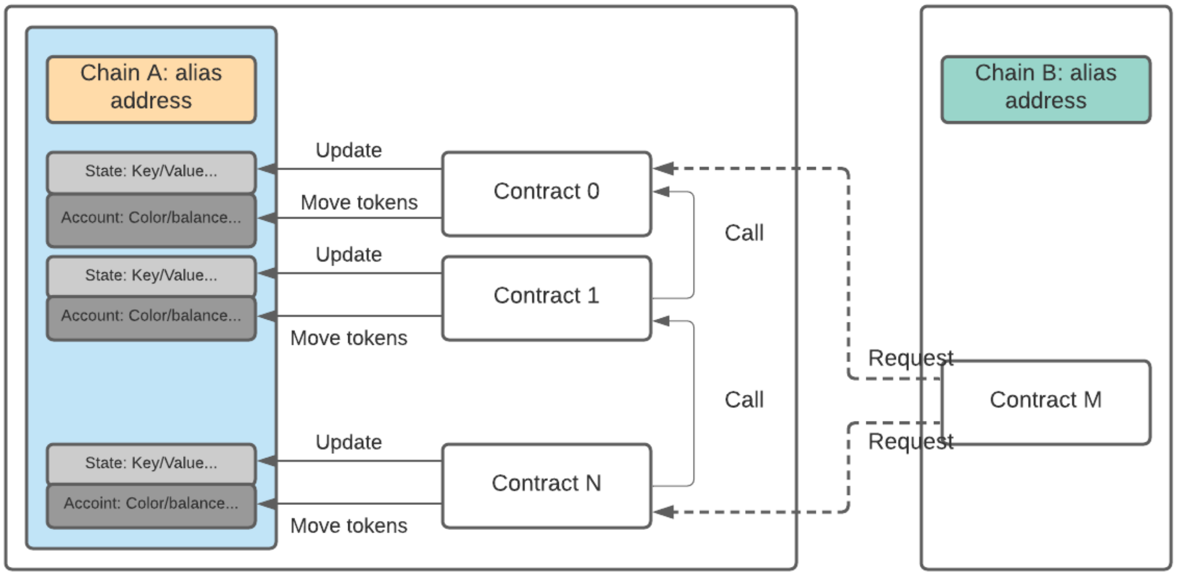
Smart contracts

Structure of the smart contract chain

Each chain can have many smart contracts deployed on it. Some smart contracts, the *core contracts*, have special functions and are deployed by default, as a part of the deployment process of the chain.

Other smart contracts are deployed on the chain by sending requests to the *root* core contract. The *root* contract keeps a registry of deployed smart contracts and performs factory function. See Virtual Machine (VM).

The smart contracts can be seen as a part of VM, the deterministic processor. The builtin (hardcoded) smart contracts are always present in the VM while dynamically loaded smart contracts (like wasm smart contracts) may be seen as plugins to the VM.



Structure of the smart contract

Smart contracts are event-driven programs which keep their state locked on the Tangle in a tamper-proof manner.

Each smart contract has some program immutably associated with it, called the *smart contract program*.

The smart contract itself is an *instance* of the program associated with it along with the data in its partition of the chain state and its token account. The terms *smart contract instance* and *smart contract* are used interchangeably, as synonyms.

Each smart contract is identified on a particular chain with its *hname* (hashed name). *Hname* is a 4 bytes identifier interpreted as a little-endian integer. It can be any *uint32* value, usually it is 4 bytes of the hash of the deployment name (a string) of the smart contact.

Hname 0 is reserved on each chain for the *default smart contract*.

State partitions. Accounts.

Each smart contract has a partition of the chain's *data state* associated with it. The VM restricts access to the chain state to this partition only.

Each smart contract has an associated on-chain account of tokens it controls. The *AgentID* of the account is always composed of the chainID (the *alias address* of the chain) and *hname* of the smart contract: (*alias address, hname*).

Entry points. Views

The smart contract program exposes *entry points*, these being functions that can be called. Each *entry point* is identified within a smart contract with the *hname* of its name or signature. The entry point provides access to a program's function, the part of its algorithm.

There are two types of entry points: *Full entry points* (also known simply as *entry points*) and *view entry points* (also known as *views*).

Entry points can be called with supplied parameters, these being a collection of key/value pairs.

(Full) entry point

By calling the entry point, attached funds (if any) will be transferred automatically from the caller to the on-chain account of the target contract. The call fails if there are not sufficient funds in the caller's account.

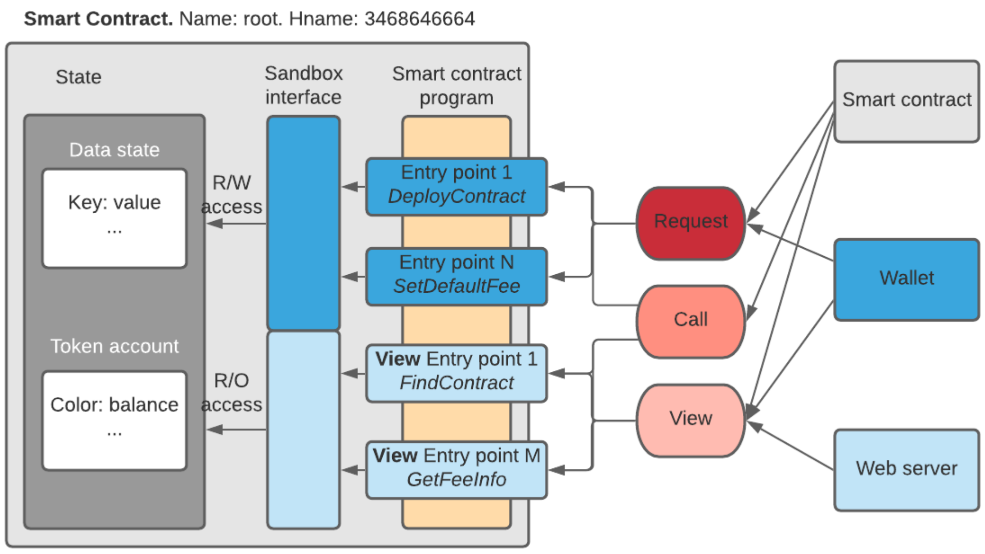
At the entry point, the smart contract program:

- Has read/write access to its own partition of the state on the chain
- May call another contract on the same chain synchronously (on the same state of the chain). The target of such calls may either be full entry points or views
- May pass and receive funds as *transfers* in calls to full entry points of contracts on the same chain
- May send requests to contracts on other chains, with attached funds

View entry point

The view entry point is used to retrieve data from the contract's state. It is a read-only access to the state. The view may be called from the same or another smart contract on the same chain, or from outside, for example from a web server. In the latter case the underlying state is always a solid state, i.e. the database.

The program processing the view entry point can also call other views. It can't call full entry points.



Builtin (default) smart contracts

Each chain contains 5 smart contracts, which are deployed automatically along with the chain itself. The following are the default smart contracts (note this is not the final list of these):

- ***Root*** smart contract: this is responsible for maintaining the fundamental parameters of the chain, such as *chain owner ID,* as well as for the deployment of new smart contracts and for maintaining the on-chain registry of these (known as a *factory* function*).* It also exposes function of fee handling
- ***Accounts*** smart contract: This is responsible for maintaining the on-chain ledger of accounts, controlled by smart contracts and L1 addresses
- ***Blob*** smart contract: this maintains a registry of big binary data objects, such as Wasm binaries. It is used by the *root* contract to deploy new contracts.
- ***Eventlog*** smart contract: this maintains on-chain log of events emitted by contracts
- ***_default*** smart contract: it is called always when target contract is not found

Builtin smart contracts may be seen as a static part of *The VM* itself.

Interaction between smart contracts

Calling another smart contract

Any smart contract can call the entry point of another smart contract on the same chain. The target entry point is specified by its contract's *hname* (local ID of the contract on the chain) and entry point code (*hname*).

Each call to another contract's full entry point can contain:

- Parameters, the collection of key/value pairs
- The *transfer*, the collections of *color:balance* pairs (only for full entry points)

The sandbox environment ensures:

- moving of funds from the caller's on-chain account to the target contract's on-chain account (only for full entry points). The call fails if there are not sufficient funds
- The proper state context for the called contract

Cross-chain messaging

Smart contracts which are on different chains, can communicate via *on-chain requests*. Each such a request has the following logical structure:

arguments, funds → target chain ID, target contract hname, (full) entry point code

Where:

- *Entry point code* is a code of the function of the SC we want to invoke
- *arguments* are inputs to that function
- *funds* are IOTA Tokens sent together with the request, and
- *Target chainID* (the *alias address*) and *target contract hname* represents the target of the request. Note that the target is globally addressed: it contains both global ID of the chain, the ID of the smart contract and the ID of the entry point

Upon invocation of the entry point at the target smart contract:

- arguments of the request are passed as parameters of the call
- funds of the request are passed as a transfer to the call

The important characteristics of the ISCP cross-chain messaging are:

- Senders of requests can be any software which is able to create a UTXO transaction, i.e. which has a valid private key(s) to sign the value transaction.
- If an external sender (not a smart contract, a wallet) is sending a request to the smart contract, it will be identified using the *sender address* for the signature of the transaction.
- Smart contracts themselves also can send requests to other smart contracts on same or other chains. In this case the sender will be identified using the *target chain ID* and *contract hname* of the sending smart contract.
- Requests are completely asynchronous. This means that the only effect the sender can see is a change of the SC state upon settlement. There's no such thing as a return value.

- The *request* is a message sent to the smart contract instance **on-ledger**. Requests are represented by UTXOs. This means that each message is a part of some transaction which must be confirmed to become part of the immutable ledger.
- To send a *on-ledger request* to an entry point of a smart contract, one only needs to have a signed transaction and access to the Tangle: there is no need to access any Wasp node.
- *On-tangle* messaging also means that the whole backlog of requests is on the immutable storage structure (the *UTXO ledger*). It cannot be lost and is fully auditable due to it's DAG structure.
- One smart contract transaction can contain many outputs and therefore many requests to many targets. By putting several requests into one transaction we have a guarantee of **atomic sending**, i.e. we are guaranteed that either all requests will appear on the backlogs of corresponding smart contracts, or none of them.
- This *atomic sending* property above does not guarantee any particular order requests will be processed by recipients, even if the recipient is the same for several requests. However, in combination with the *quasi-atomicity* there's a guarantee that once sent, a package of requests to many SCs will all be processed eventually.
- sending an *on-ledger request* involves two separate steps and two subsequent confirmed transactions: confirming the request transaction and the state update in the target smart contract. This means that in general these two steps make this kind of smart contract transaction **non-atomic**, therefore we call it **quasi-atomic** which is a different way to achieve the same thing.

The *quasi-atomic* messaging ensures trustless, guaranteed and tamper proof transacting among smart contracts on different chains in ISCP.

The distributed system

The state of the contract chain and of each smart contract is maintained on the immutable ledger, meaning that this is an append-only structure. The state is mutated by adding new blocks to the contract chain and the anchor output/transaction to the UTXO Ledger. State mutation is also called *state transition*.

The smart contract program is an immutable program on the contract chain. Each smart contract on the chain may call another smart contract on the same state of the chain. On this basis, we can regard the totality of smart contracts as one big program updating the state of the contract chain. We refer to this as the *SC program* or *VM* even though it may consist of many smart contract programs deployed on the chain.

A state transition occurs when the *SC program* calculates a block of state updates and the anchor transaction. The anchor transaction is confirmed on the UTXO Ledger, while the state updates contained in the block are applied to the current *data state*, as follows:

$$SCProgram(State_N,\ Request)\ \rightarrow Block,\ anchor \rightarrow State_{N+1}$$

If the *SC Program* is deterministic, the state transition is a deterministic process too: whenever this is repeated with the same inputs, it will produce the same output.

Once all the inputs $State_N,\ Request$ and the $SCProgram$ itself is immutable on the *UTXO ledger*, the next state is always deterministic and can be verified.

Verification would mean re-running the program ourselves whenever we needed to determine if we trust the state of the smart contract. This is not practical. Instead, to make the state transition trusted, we need to make the processor which runs the *SCProgram* trusted.

This is the reason smart contracts are run by a **distributed processor**. The term "distributed processor" refers to many processors performing the same calculations and coming to a **consensus** on the result of these computations. This consensus is reflected in the next block which updates the state of the chain.

It may be that the whole network is that distributed processor, as is the case in Ethereum.

In ISCP we are using a quorum majority voting in the 'BFT' setting, to determine the consensus arrived at by the committee of different processors.

Let's say we have a committee of N processors. We determine the *threshold* T, $N/2\ <\ T\ \leq\ N$ of the committee. We require that at least T processors produce exactly the same result. If they do so, then we consider the result a valid result. The remaining $N - T$ processors may be faulty, malicious or unavailable ("bizantine"): these have no effect on the final result.

It must always be the case that $T\ >\ N/2$ in order to avoid two different quorum majority opinions. It usually is the case $T = floor(2N/3) + 1$ as an optimal value.

To break the quorum one would need to stop more than $N - T$ processors. To fake the result of calculations one will need T nodes to be malicious.

Note that even in the case of a faked result, there is an immutable audit trail on the Tangle which can be deterministically verified. This property of having a cryptographical proof of malicious behavior may be used for automated procedures punishing and discouraging such behaviors.
The distributed consensus systems are often called *trustless*. It means we assume each node in the system may be malicious, so we do not trust it. However, the distributed system does not suffer from the single point of failure and, if we trust the majority (quorum) is always honest, we do not have to put any trust into individual nodes to trust the result.

Committee

Each contract chain is run by a *committee* of Wasp nodes. These are the processors mentioned above: the quorum of nodes in the committee must come to the same result in order for the valid state transition to occur.

For this reason, the smart contract is a distributed and redundant structure (a distributed processor); that is, it does not depend on a single point of failure (SPF).

Consensus on computations

Even if all committee nodes are honest (i.e. they have no malicious intent), there are factors which may make each node see things differently. This can lead to different inputs to the same program on different nodes and, consequently, to different results.

There are several possible reasons for such an apparently non-deterministic outcome.

Each committee node has its own access to the UTXO ledger, i.e. committee nodes are usually connected to different IOTA nodes. The reason for this is to not make access to the UTXO ledger a single point of failure, i.e. we also want access to the Tangle to be distributed. This may often lead to a slightly different perception of some aspects of the ledger, for example of the token balance in a particular address. Also, each node has its own local clock and those clocks may be slightly different, so there is no one objective time for nodes.

The *requests* (UTXOs) may reach Wasp nodes in any arbitrary order and with arbitrary delays (even if these are usually close to the network latency).

Before starting calculations, nodes have to have consensus on the following things:

- The current state of the chain i.e. on the *state output*
- Timestamp to be used for the next state transaction
- Ordered batch of requests to be processed
- Address where node fees for processing the request must be sent (if enabled)
- Mana pledge targets

In order to achieve a bigger throughput, the committee picks requests from the on-ledger backlog and **processes requests in batches**, not one by one. This means the committee has to have a consensus on the batch of the requests and the order of the requests in the batch. After at least a quorum of committee nodes have a consensus on the above, honest committee members will always produce identical results of calculations.

Proof of consensus; BLS; Distributed keys

Suppose a quorum of committee nodes has reached consensus on inputs and produced identical results, these being the block of state updates and the anchor transaction.

The anchor transaction contains chain state transition, the *AliasOutput* and token transfers, so it must be signed.

The requirement is that it should **only be possible to produce valid signatures of inputs of the anchor transaction by the quorum** of nodes. In this case, a confirmed anchor transaction becomes a cryptographical **proof of consensus** in the committee.

BLS threshold signatures

In ISCP we use **BLS cryptography and threshold signatures in combination with polynomial (Shamir) secret sharing** to achieve the requirement above.

In short, we use BLS addresses as *state addresses* (*controlling addresses*) of the chain account where the state of the chain is locked. The secret sharing and threshold signatures allow for control of the address by any T out of N secret keys (partial private keys), where N is the size of the committee and T is a *quorum factor*.

The "control of the address" means the ability to produce a valid signature of the transaction to the corresponding address. In threshold signatures the valid (master) signature can be reconstructed from any T out of N *partial signatures*. There's no need for all N of them and there's no need to know the master private key in order to reconstruct a valid signature. Each *partial signature* is a signature by one out of N of secret keys, while each of those secret keys is known only to the corresponding committee node.

There is more information on BLS and threshold cryptography here, here, here and here. We use Dedis Kyber library in the implementation of Wasp. Goshimmer has BLS addresses implemented at its core. This means that BLS addresses are conventional addresses and BLS signatures in transactions can be validated by the IOTA node just like any other type of signature.

Distributed Key Generation (DKG)

For the committee nodes to be able to collectively produce a valid threshold signature, they must secretly hold the corresponding private keys.

Those private keys are generated during formation of the committee during a *DKG procedure*. There are well known algorithms of *Verifiable Secret Sharing (VSS).* We are using the Rabin-Gennaro algorithm implemented in the *Dedis Kyber* library.

In the decentralized DKG no single entity, nor even a colluding set of less than T nodes, has the ability to produce a master signature unless it corrupts a quorum and steals its partial keys, which in most practical situations is not feasible.

BLS also makes any master signature produced by the committee, a source of deterministic randomness, unpredictable as long as the quorum of partial private keys is safe. This feature makes this unfakeable randomness available to any SC program.

The consensus algorithm

We will describe here the currently implemented consensus algorithm in general terms. It will likely change in the future; therefore, the main features are more important than the implementation details. This follows the general pattern of classical BFT algorithms for finite sets of participants.

TBD

Deployment of the chain

Deployment of the contract chain consist of the following steps:

- Selecting of the committee.
- DKG. (this may be a completely separate step).
- Creating the *origin transaction* with *origin output* on the Tangle
- Creating the *init* request transaction
- Activating the committee
- The *init* request transaction is processed by the *root* contract (which is always present). The *root* contract deploys all other built-in contracts on the chain

The whole deployment process is initiated by some entity called *initiator*. The *initiator* sends the *init* request and the *root* contract stores this as the *chain owner ID* in the state of the chain. The *initiator* becomes the *governor* of the chain. It can pass ownership to another entity, address or smart contract.

Committee selection

The committee of the chain is the main element of security of a smart contract on the chain and therefore the main contributor to trust in it. The significant characteristics are the size of the committee, the quorum factor, who is behind each node and what is at stake for each participant.

It is important to note that there are a lot of use cases where committee selection is done not by some automated protocol but by outside entities. For example consortia may run chains under their consortium agreements with nodes operated by individual consortium members. Another example would be a corporation which may run the committee of nodes distributed among departments of that corporation, enabling it to maintain certain governance rights over these within the company's own headquarters. Similarly, a private person may run a small committee of their own nodes aiming for fault-tolerance. And of course, committees may be formed by decentralized consensus in the open market, taking into account the reputation and stakes that each participant is willing to pledge.

It is our intention to leave the question of the committee selection outside of ISCP, at least in the current version, because we want variety and open choices for ecosystem development.

Therefore, when talking about deployment of the chain we assume the committee is already selected and all committee members agree to participate in that committee (btw, this means actual consensus between participants of the committee itself).

DKG

The first thing the committee should do is to run DKG.

Running DKG results in a distributed set of private keys each known only by the corresponding node. It also generates the *chain address* as well as partial public keys and the master public key.

All of this is stored in the registry of each committee node. Note that partial private keys are known only to the corresponding nodes. The master private key is not known to anyone, not even *the initiator of the DKG*.

Strictly speaking, the DKG is a separate process from the actual deployment of the chain: having distributed keys and chain address is a prerequisite to the deployment of the chain.

Origin transaction

The initiator should create the SC transaction called *origin transaction* with *origin output* in it.

The *origin output* does the following:

- It starts the chain
- It mints the *alias address* which will be used as *chainID*
- It contains the address generated during DKG as *state address* (the *controlling address*). It means, *state output* will be unlocked only by signing it with the private key(s) behind the *state address*.
- It contains a *state hash* of the *zero state*, the initial state of the chain. The *zero state* is a result of applying an empty block to the empty *data state*.

Init request transaction

Immediately after confirming the *origin transaction*, an *init* request (the output)and the transaction is created by the same *initiator* address and confirmed on the *UTXO ledger*. This will contain request data for initialization of the *root smart contract*.

The *init* request will contain initialization data for the new contract chain. The *root smart contract* will initialize its state on the chain with *chain owner ID* equal to the sender of the *init* request and will mark the chain as fully functional (this happens after activation of SC).

Activation of SC

After keys are generated and the *origin transaction* and *init transaction* are confirmed on the *UTXO ledger*, the chain needs to be activated on the committee (Wasp) nodes.
All chains are activated automatically upon node start from bootup records stored in the registry of the node.
After activation, the chain start pulls the backlog from UTXO ledger and processes the *init* request.

Virtual machine

In ISCP we distinguish two things:
- The *VM* (*Virtual Machine*) itself
- *VM plugins* (*processors, smart contracts*), a pluggable part of VM

The VM itself is a deterministic executable, a "black box", which is used by the distributed part of the protocol to calculate output (result) from inputs before coming to consensus among multiple VMs and finally submitting the result it to the ledger as the state update of the chain, the block.
Naturally, results of calculations, the output, is fully defined by inputs, i.e. VM calculations are deterministic.
The VM can be extended dynamically by adding *VM plugins*, the *VM processors* which run deterministic *smart contract programs*.

The VM (including all VM plugins) is a deterministic function which computes next state of the chain from requests and the previous state:

$$VM(chain\ state_N,\ request1,\ ...,\ requestM)\ \Rightarrow chain\ state_{N+1}$$

The *VM plugins*, actually smart contracts, itself are immutable parts of the *chain state*. They are *deployed* on the chain by adding their program binaries and metadata to the *data state*, just like any other smart contract data.

That makes the whole system fully deterministic and therefore auditable.

VM abstraction. VM processors

By *VM abstraction* in ISCP we understand a collection of generic interfaces which makes the whole architecture of ISCP and Wasp node agnostic about what exactly kind of deterministic computation machinery is used to run smart contract programs.

The VM contains multiple dynamically attached *VM plugins* or *VM processors*. The VM invokes *VM plugins* to perform user-defined algorithms, the smart contracts. The *processor* is attached to the VM through the generic *VMProcessor* interface.

Each VM processor (and therefore each smart contract) exposes call-able *entry points*. Each *entry point* can be called with call parameters and attached token transfer.

Full entry points (or just *entry points*) can modify the state of the smart contract, i.e. the *data state* and the account: for example transfer tokens from the account or modify state variables. The *full entry point* is called from requests and from other *full entry points*, i.e. always in the environment where the caller is securely identified.

View entry points or just *views* provide read-only access to the smart contract state. The can be called from an outside environment which needs access to the chain's state, for example from a web server.

Each VM processor has a *default entry point*. The *default entry point* is automatically called by the VM logic whenever a specified target entry point is not found.

The VM Type abstraction

The *VMType* defines a specific implementation of the *VMProcessor* interface. The VM Type provides the constructor which creates a *VMProcessor* from the program binary. For the smart contract *VMType* defines an interpreter of the smart contract program in its binary form.

Each *VM type* is statically built into the Wasp node. To introduce a specific *VM type* into the Wasp node it must be implemented and statically registered into the Wasp node. The *VM abstraction* provides interfaces for it, so the rest of the code is agnostic about specific *VM types*.

Once *VM Type* becomes known for the core logic, it can use generic tools of the smart contract deployment machinery from binary code of smart contact programs.

The following VM Types are available by default:

- *builtinvm* is the native smart contract processor, hard coded into the Wasp core. The core contracts are of this type.
- *wasmtimevm* is the *VM Type* which loads smart contract programs from WebAssembly binaries, compiled from Rust into Wasm using *wasmlib* and interpreted by *Wasmtime* interpreter.
- *examplevm* the hard coded smart contract processors for all kind of examples

Usually, one *VM processor* represents one smart contract. However, one *processor* may represent an entirely new plugged-in VM, such as EVM, with its own VM sandbox.
To implement EVM on the ISCP one will need to implement a new *VM Type* for EVM, the *VM processor* capable to load binary EVM code and interpret it.

Core contracts

There are 5 *default processors* which implement 5 *core contracts* on each chain. These processors are natively implemented in the Wasp code with the *builtinvm* VM type, so the *core contracts* are statically hardcoded into *The VM*.

The *core contracts* are:

- *root* contract: deployment of smart contracts, factory and registry function, fee management and other admin functions
- *accounts* contract: on-chain account management
- *blob* contract: registry and storage of big binary objects, for example smart contract program binaries
- *eventlog* contract: keeps log of events on the chain
- *_default* contract: empty contract with default entry point. The *_default* contract is invoked when a target contract with specified *hname* is not found on the chain. The *_default* contract is also a holder of the *common account* controlled by the chain *owner*. All tokens transferred to the *core contracts* (probably by mistake) end up in the *common account.*

Calling the VM. Input, output and sandbox interface

Main inputs for the VM are:

- *State output* of the current state of the chain
- *Solid data state* of the chain
- A batch of requests to process. Each request is either an on-ledger request (the output to consume) or an off-ledger request.
- Timestamp
- Some other (fee destination, mana pledge targets and other)

The VM returns the result of calculations, a deterministic function of inputs:

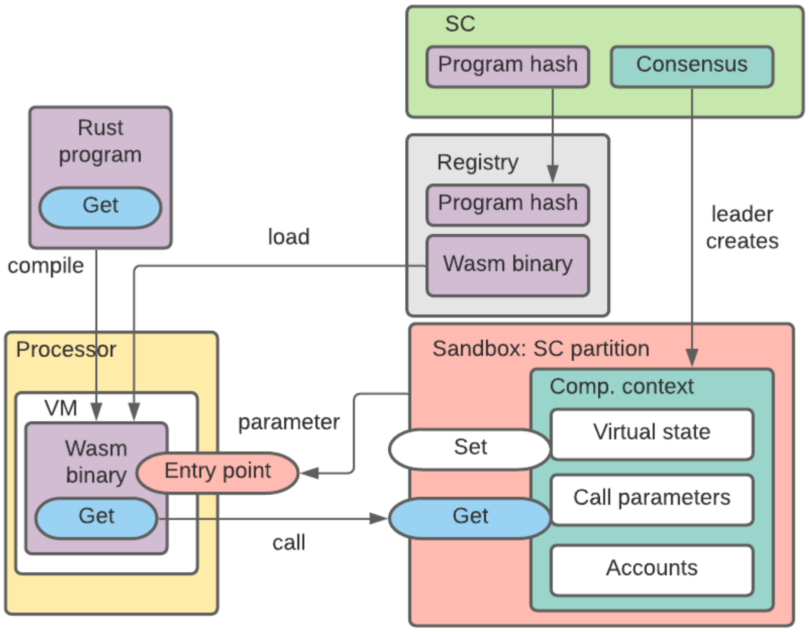
- The *block*. It consists of a batch of *state updates*, one state update per request
- The essence (not signed yet) of state transaction. The state transaction contains the next *state output*. The state translation has timestamp equal to timestamp in inputs plus T nanoseconds, where T is number of requests

For each request *the VM* locates the target VM processor (the smart contract program) and calls its target *entry point* with the parameters and token information wrapped in the request. If the target smart contract does not exist it selects the _*default* processor.
If the target *entry point* does not exist, the default *entry point is called*.

The called *entry point* of the smart contract program is provided with the access to its partition of the *chain's data state* and its on-chain account via the *Sandbox* interface. The *Sandbox interface* provides limited and deterministic access to the state through abstraction of the key/value storage and similar metaphors.

The VM handles fees and exceptions outside the plugins (smart contracts). Each call to the smart contract program also means crediting of token balances provided in the call parameters into the on-chain account of the called smart contract.

The call to the VM plugin during processing the request is fully virtualized. It means all updates to the state are collected in the virtual (tentative) state and not committed into the database until confirmation of the anchoring output on the UTXO ledger.



Rust/Wasm smart contract programming environment

We provide Rust/Wasm smart contract programming environment as a part of Wasm in the first versions. In next versions we plan to implement other VM Types such as EVM
Theoretically any high level language can be used to write SC programs to compile them into the *wasm*. We need only be able to generate a *wasm* binary from the source and load it into the *VM processor*.

The main requirement for the language is that it should be possible to implement *sandbox* bindings such that, after the program is compiled into *wasm*, these bindings would be converted into the binary which can be dynamically linked with the *processor* and *sandbox* interfaces when the binary is loaded into the *VM*.

Another requirement is to support a fully deterministic *wasm* runtime: the *wasm* interpreter must produce exactly the same result for the same inputs (the inputs being the computation context, accessed through the *sandbox* interface).

We use *Rust* as a high level language to write SC programs for several reasons :

- Rust programs can easily be compiled into to *wasm* binaries
- The flexibility and memory management model of Rust allows minimum runtime overhead in the *wasm* binary
- The *Rust*/*wasm* environment can be made completely deterministic
- *Wasm* binaries generated from Rust can be very small, just a few kilobytes
- *Rust* is becoming a popular language of choice for SC development with good support and a wide developer community.

Deploying the smart contract on the chain

The deployment of the smart contract on the chain is essentially plugging the SC program into *The VM*. The VM provides generic logic of smart contract deployment, independent from the specific *VMType*.

To deploy a smart contract, first its binary must be present in the built-in *blob* smart contract. To achieve this, send a *StoreBlob* request to the *blob* smart contract. The blob itself is a named collection of binary arrays, so the program binary can be packaged, for example with its source or any other data that needs to be stored immutably with the program binary. The blob data is accessed on the chain by its hash, a deterministic value computed from the blob data.

The *VMType* is specified as a part of the blob data, so the blob is self-describing.

The Smart contract is deployed by calling the *DeployContract* entry point of the *root* or by sending a respective request.

The parameters of the deployment are:

- Hash of the blob with binary code
- Name of the contract (used to calculate its *hname*)
- Description

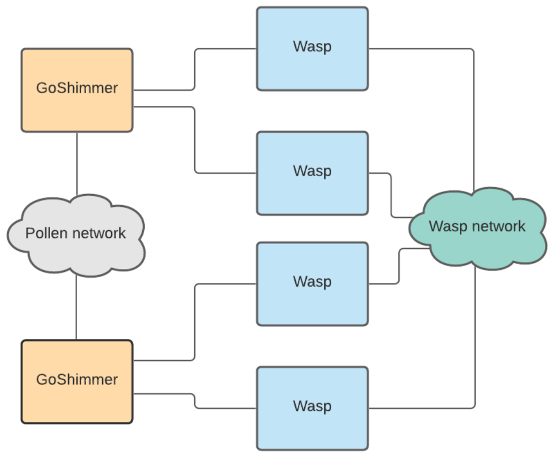
The *root contract* finds binary code in the *blob* registry and loads it into the VM processor using known VMType. After that the *root* processor immediately calls the *init* entry point of the new VM processor with the parameters from the *DeployContract* call.

Wasp node

The ISCP is run by a network of Wasp nodes. To get access to the *UTXO ledger* on the Tangle, each Wasp node connects to a Goshimmer node on the network. Note that each node may be connected to a different Goshimmer node. This way we avoid a single point of failure/attack w.r.t. the L1 ledger.

One Goshimmer node can connect to many Wasp nodes. Each Wasp node is connected to exactly one Goshimmer node.

To reach consensus and to sync state, Wasp nodes send messages among themselves directly, without involvement of the Pollen network of Goshimmer nodes. That is, Wasp nodes constitute their own network.



Communications between Wasp node and Goshimmer node

Wasp and Goshimmer nodes exchange data fast through asynchronous duplex connections. The Wasp node which runs a particular chain subscribes to the Goshimmer node to receive all confirmed transactions which contain outputs to the *alias address* of the chain.

This way Goshimmer node *pushes* state and backlog updates to subscribing Wasp nodes.

The protocol also implements *pull* possibility. The Wasp node can query specific transactions and individual outputs form the Goshimmer node.

Off-ledger requests

The *off-ledger requests* complement *on-ledger requests*, described above.

The *on-ledger requests* are transactions on the ledger. They are processed in a *quasi-atomic* manner which makes the cross-chain communications trustless and tamper-proof. Each individual request can be sent to the UTXO ledger and it will find its way to the target chain.

The *on-ledger request* thus:
- require first to confirm the request transaction and then it must be settled on the chain's state. Therefore the confirmation latency is twice the usual confirmation time. For some use-cases it may be too long
- the throughput of the whole multi-chain system is limited by the throughput of the Tangle ledger.

The *off-ledger* requests are intended to be used for fast micro-transactions and high-throughput for an individual chain.

The off-ledger request is similar to the on-ledger request, however:
- It is not a confirmed transaction. There's no UTXO behind it
- It is sent directly to one of the access or committee nodes (a Wasp node) of the chain using provided API (as opposed to just posting it to the Tangle)
- It contains the same data just like *on-ledger* request, i.e. tokens, parameters, target specification
- It contains mandatory increasing counter to prevent replay
- It contains the ED25519 digital signature of the request's data essence by the sender

To be accepted by the chain, the sender of the *off-ledger request* must have an on-chain account controlled by it with non-zero balance. It means, the sender first must post a request to the *accounts* core contract to deposit some tokens (it may be 1 iota) and this way securely identify itself by its sender address (backed by a private/public key pair).

Each *off-ledger request* must be signed with the private key corresponding to the on-chain account. If signature is invalid or chain does not have the account, the *off-ledger request* is rejected

The chain will maintain the increasing counter from the last *off-ledger request* in the corresponding account. It will reject any subsequent *off-ledger request* with the counter less or equal to the stored on the chain. This way any attempt to replay the *off-ledger request* will be rejected.

The colored token balances sent in the *off-ledger request* will be treated as debit of tokens from the sender's on-chain account. It means, if the request wants to send some tokens to the target smart contract, those tokens must be prepaid (deposited) into the sender's on-chain account beforehands by using *on-ledger requests.* Ths same logic is applied to the fees: fees must be prepaid.

The Wasp node which receives the *off-ledger request* from the API will gossip the request to other committee nodes. The receiving nodes will validate the request the same way as if it received it directly. Later the *off-ledger request* will participate in the processing of the batch equally with *on-ledger requests*.

This way each chain is able to achieve high throughputs, likely 1000+ **per each chain** without requiring significant throughput from the UTXO ledger.
Note, that *off-ledger requests* do not require consumed inputs in the anchor transaction. That makes it possible to place a big amount of requests in each block.