IOTA Smart Contracts
Architecture description

(ver 2, Pollen release)

Change log

When	Who	Change
Oct 2, 2020	E.Drąsutis	Initial version
Oct 31, 2020	E.Drąsutis	Ver 2. Expanded with contract chains
Nov 30, 2020	E.Drąsutis	Misc updates and wording. Builtin contracts. On-chain accounts
Dec 15, 2020	M.Bennett	Multiple edits
Feb 12, 2021	E.Drąsutis	Edit on-chain accounts
Feb 22, 2021	E.Drąsutis	Merkle proofs

Table of contents

General

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

In the version 2 of the architecture document we added new important concepts to the initial version of the protocol, such as smart contract chain. The protocol itself is implemented in the alpha release of the Wasp node software (Feb 2021). It still relies on the Pollen model of the Value Tangle on the GoShimmer node. For any subsequent implementations of other releases of the Value Tangle protocol and underlying token ledger the specification must be updated accordingly.

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 and the Value Tangle. We are also investigating the possibility of building an ISCP for Chrysalis 2 protocol for the current mainnet.

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

Motivation and goals

ISCP is a further development of the Tangle protocol: a scalable, parallelizable, feeless (miner-less) DLT. What we want to achieve is the following: we aim to develop a distributed multi-chain smart contract environment on top of the Tangle. The environment will be able to run many distributed smart contract chains (aka contract chains, chains) in parallel. Each chain will be able to host many smart contracts, sharing the same 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 software agents are able to manipulate assets on the IOTA ledger according to its algorithms. The smart contracts also are able to exchange native assets residing on the IOTA ledger, i.e. 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 such as 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 updates (unlike blockchain)**.** ISCP is a natural development of the Tangle protocol to enable parallelizable smart contracts.

Introduction

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

ISCP is built on top of the *Value Tangle*. *Value Tangle* is a protocol responsible for the *IOTA token ledger*. We say that ISCP is *level 2* (*L2)* or *off-tangle* (*off-ledger*) protocol.

The *Value Tangle* 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.

This kind of program can implement any algorithms, hence it is *Turing complete*. Smart contracts can manipulate digital assets and move them between Tangle addresses and on-chain accounts according to their algorithms. Their data, the state, is stored on the distributed ledger of the IOTA Tangle and is tamper-proof.

All smart contracts are run on **smart contract chains** (contract chain, chain). The contract chain can contain many smart contracts. The contract chain is functionally equivalent to the blockchain such as Ethereum blockchain or *parachain* in Polkadot. The smart contracts on one chain share their global state and as such, they can interact (call each other) in the context of the same state.
The smart contracts on different contract chains can interact and transact between each other in a trustless manner by sending **requests** with attached data and funds on the underlying Value Tangle.

In this version of the document we rely on the version of the *Value Tangle* which is implemented as an experimental Pollen network of *Goshimmer* nodes.

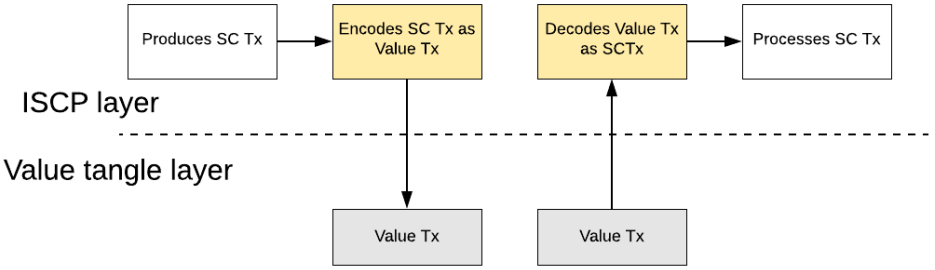
The ISCP is implemented as a network of *Wasp* nodes. Smart contracts are run on the network of *Wasp* nodes. Each *Wasp* node connects to the Value Tangle via Goshimmer nodes.

The Value Tangle is dealing with *value transactions*.

ISCP is dealing with *smart contract transactions (SC transactions).*

Each **SC transaction is a value transaction**. In other words, all SC transactions can be interpreted as value transactions, while specific value transactions can be interpreted (parsed) as SC transactions.

The ISCP sends value transactions to and receives value transactions from the *Value Tangle*. Some value transactions are recognized as SC transactions. Only these are processed by the ISCP.



Transactions

Value transactions

The structure of a value transaction is defined by the Value Tangle (Pollen release). There are the following main elements of the value transaction:

- UTXO transfer
- Signatures
- Data payload

UTXO transfer consists of

- 1 or more *inputs*. Each input refers to an *output* of some other value transaction. The output is referenced by its UTXO transfer *transaction ID* and the *address* of the output.
- 1 or more *outputs*. Each output consists of an *address* and a *colored balance*. Each output address must be unique within the transaction.

There are as many signatures in the value transaction as there are different input addresses. Each signature signs the *essence of the transaction*.

The *Data payload* from the Value Tangle perspective is just a data chunk attached to the UTXO transfer.

The *Transaction essence* is a serialized binary of UTXO transfer and data payload.

The *transaction ID* (the hash) is a hash of *transaction essence* and signatures.

The ISCP assumes that the serialized binary form of the transaction is *deterministic.* This means it is the same for any representations of the logical value transaction, i.e. it does not depend on the particular order in which elements are placed within the transaction.

Please find a short introduction to the concepts of the Pollen version of the Value Tangle here.

Smart contract chain

The ISCP can run many **parallel smart contract chains** on the Tangle.

The *smart contract chain* (or *contract chain*) contains many smart contracts. All smart contracts on one chain will work synchronously on one global state. This enables SCs on the same chain to call each other synchronously.

The *contract chain* represents one state which is updated via state transitions, defined by *blocks*.

Each smart contract on a contract chain is an immutable program which can access only its own partition of the global state in the contract chain. Its access to funds (tokens) is also limited to its own accounts, which are maintained for each smart contract in the chain.

More on state and fund access see below.

Chain ID; Contract ID

Each contract chain on the Tangle is identified by its **chain ID**.

Currently the chain ID is an IOTA address which contains the **chain token** which keeps the state of the chain locked. (Note: in the future, the alias of the address chain ID will be used instead of the chain address itself)

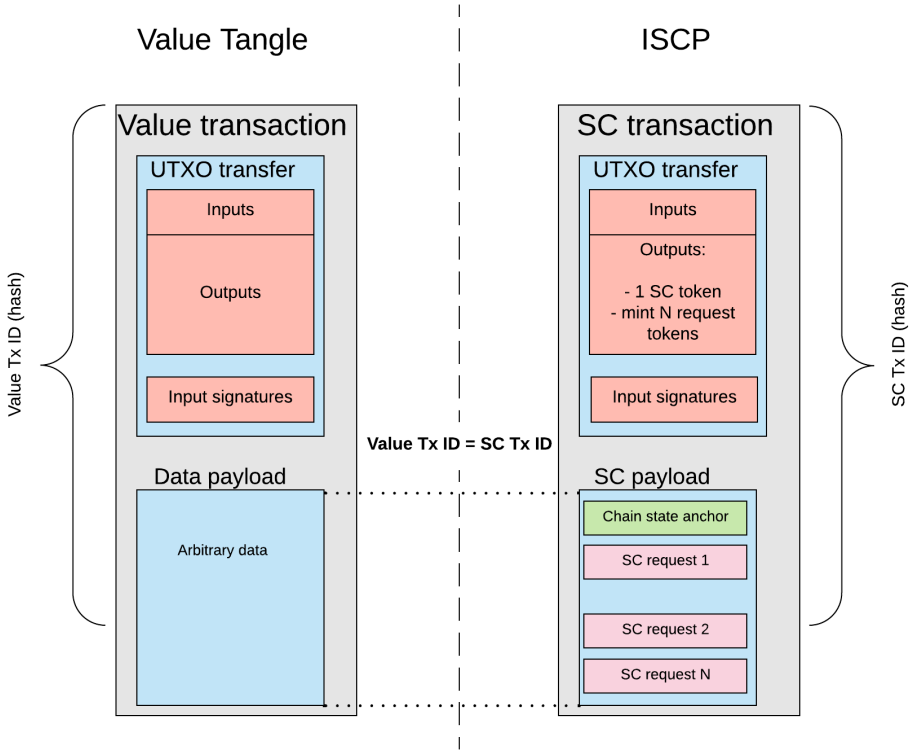
Each smart contract within the contract chain (it native chain) is identified through its hashed name or *hname*. The *hname* is a *uint32* value interpreted as little-endian from the first 4 bytes of the *blake2b hash* of the name of the smart contract (a string).

Globally on the Tangle, in ISCP each smart contract is uniquely identified by the pair *(chain ID, contract hname)*. We use term *contract ID* as binary concatenation *chainID* || *contract hname.*

Smart contract transactions

Whenever a value transaction enters the ISCP layer, the ISCP layer parses and interprets the *data payload* of the value transaction. From the perspective of the Value Tangle this Data payload is just a data chunk. It is interpreted on the ISCP layer.

If the ISCP layer is able to parse the data payload field of the value transaction successfully, the value transaction is recognized and interpreted as a *smart contract transaction* (*SC transaction*). In this case we say the parsed data field of the value transaction represents an *SC payload*.



Due to the fact that each SC transaction is also a value transaction we can assume:

- All smart contract transactions received by ISCP from the Value Tangle are confirmed transfers.
- The smart contract transaction can only be confirmed and become immutable if it is confirmed on the Value tangle.

The data payload of the smart contract transaction can’t be empty.

Note that as for any value transaction, any SC transaction always moves at least one token on the ledger: value transactions without value transfer are not allowed by the Value Tangle

Smart contract transaction payload

The SC transaction payload is parsed and interpreted by the ISCP. Valid value transactions with a data payload which can't be parsed and validated by the ISCP are not SC transactions and are ignored by the ISCP.

The SC payload contains one or several **sections** in it:

- Optional, one contract *chain state section* (also called *state section*, *chain state anchor, chain section)*. The transaction which contains the chain state section is called the **chain state anchor transaction,** *chain state transaction*.
- Optionally, any number of **SC request** sections

Either the chain state section or at least one SC request section must be present. This means that the SC payload of a valid SC transaction can't be empty, and by inference, any value transaction with an empty payload won't be parsed as an SC transaction.

The *chain section* represents a state update step in a chain of state updates in the chain. This is functionally equivalent to the block on a blockchain. It contains a **block index**, the hash of the off-tangle part of the chain state, a timestamp and other data. See the Chain state section for details.

The *request section* represents a request to some smart contract. A request to the smart contract is an *on-tangle* message sent to the smart contract. See the Request section for details.

Each SC transaction may contain many requests to many different smart contracts on the same or different chains. Having the chain state update and requests in the same SC transaction means that we are updating the state of the chain **and** sending requests to any number of other smart contracts, cross-chain, in a single transaction, i.e. atomically. If a transaction cannot be confirmed for some reason, the state of the chain will not be updated, nor will requests be sent.

Chain state section

The *chain state section*, when present in the SC transaction, represents a *chain state update*. We will discuss the SC state in detail below. The chain state block contains the following fields:

Name	Data format	Function

chainColor	32 byte array	Color code of the contract chain (see Address and color of the contract chain). It allows one to locate the chain token, the NFT, in the current address of the chain
blockIndex	*uint32, 4 bytes little-endian*	Sequential index of the state update in the chain. The Origin state block has index 0 and is subsequently incremented with each block
timestamp	*int64, 8 bytes little-endian*	Unix timestamp of the chain state in nanoseconds
stateHash	32 byte array	Hash of the contract chain state's *off-tangle* data. (*Note: in the future state hash will be the Merkle root of the chain state*)

A chain state transaction is normally issued by the committee of the chain together with the new block, the state update. The only exception is the origin transaction which has *blockIndex* = 0 and is issued by the external entity which deploys the contract chain.

If a chain state section is present in the SC transaction, the UTXO transfer part of the containing value transaction must contain exactly 1 output to the *SC Address* with exactly 1 token with color equal to the *chainColor* field in the state section (see Address and color of the contract chain). Otherwise the SC transaction is deemed semantically invalid.

Request section

A SC transaction can contain 1 or more request sections, independently of whether it contains a state section or not. Often the SC transaction is produced by external users (not the SC itself), the wallets, which are requesting some function of the SC. In these cases the SC transaction will contain only request section(s).

Each request section is interpreted as an *on-tangle* asynchronous request message sent to some target smart contract, on the same or on another chain. This also means that either all the requests in the SC transaction are sent, or none of them in the case where an SC transaction is rejected by the tangle for some reason. See also Messaging model below.

If the SC transaction also contains a state section, this means that the state is updated **and** requests are sent in the same transaction, **atomically**.

Each request section can be referenced by its *requestId*, a globally unique identifier of the request section in the Tangle ledger. This contains the following fields:

Name	Data format	Function
transactionID	32 byte hash	ID of the containing value transaction
requestIndex	uint16, little-endian 2 bytes	Index of the request section within the SC transaction

In serialized form, the *requestId* is the concatenation *transactionId* || *requestIndex*.
In the human readable form this is represented by the request index followed by a base58-encoded transaction ID, for example:
`[0]8v2G3wi1JzDb3vieAR2wZFYnxYqnaeLs9skduPpok5jp.`

The request section contains the following fields:
(here *hname* is the first 4 bytes of the hash if the name/string is interpreted as little-endian uint32. The values `0x00000000` and `0xFFFFFFFF` of *hname* are reserved)

Name	Data format	Function
senderContractHname	*hname*	If this is not 0, it is a *hname* of the sending contract on its chain. In this case the sender's chain ID is taken from the state section. If the transaction does not contain a state section, this means it is sent by a wallet, not a SC. In this case the field is uninterpreted.
targetContractID	35 byte array, concatenation *chainID* || *contractHname*	Target of the request
entryPoint	*hname*	Smart contract function ID code. *Hname* of the name of the entry point
transfer	A collection of *color:value* pairs	The information defines exactly which part of the transfer of the value transaction belongs to this request section. Note: this information is somehow redundant and this part of the request section

		may change in the future, together with the change of the structure of the value transaction. The semantic validation of the SC transactions ensures that all transfers in the request sections are consistent with the token transfer on the value transaction.
timelock	uint32, little-endian 4 bytes	If 0, the field is uninterpreted. Otherwise it is interpreted as the **Unix timestamp in seconds** after which the request becomes unlocked and can be processed by the smart contract. *See also consideration of Year 2038 problem*
args	A key/value map: []byte -> []byte	key/value map of arguments, the input data for the request processing entry point. Key cannot be 0 length and must be unique in the map. Otherwise keys and values are arbitrary byte arrays.

Address and color of the contract chain

Each contract chain on the Tangle has two fundamental values which are used to identify and locate the contract chain on the Tangle, i.e. globally:

- **chain address**
- **chain color**

The *chain address* is an address on the Tangle. This address represents an account where all tokens (UTXOs) controlled by the chain are being kept.

The tokens controlled by the chain can easily be located by querying the balance of the chain address. The sub accounts of specific smart contracts are managed by a built-in core contract on each chain (see below).

The *chain address* is currently used as the main identifier of the chain. Subsequent releases may use *chain address alias* for that.

The chain address always contains exactly one non-fungible token (NFT), the **chain token**, with color equal to the *chain color*. This makes *chain color* also a unique identifier of the contract chain globally.

The *chain token* ensures global consensus on the state of the chain (see below Consensus on state).

The state

Logically, the state of the chain consists of:

- State of the chain address, i.e. all confirmed UTXOs contained in the chain address on the IOTA ledger.
- A collection of arbitrary key/value pairs which represent use case-specific data stored in the chain by its smart contracts.

Our goal is to ensure tamper proof storage of the chain state in the *chain ledger*.

The chain state (ledger) is divided into partitions, each belonging to a particular smart contract. Each smart contract can only access its own partition of the state, so it is completely isolated from other smart contracts on the same chain.

The chain ledger consists of an *on-tangle* part and *off-tangle* parts.

We describe below the chain ledger as a whole. The partitioning of the state by a smart contract is always assumed.

On-tangle state

The **on-tangle part** of the chain state consists of:

- All UTXOs contained in the chain address
- All the value transactions, which have at least one UTXO in the chain address. These transactions can always be traced from those UTXOs.

The on-tangle part mutates each time new value transactions arrive which contain at least one UTXO to the chain address. In this way the on-tangle part of the chain state mutates asynchronously and different nodes may have a different view of this at different points in time.

It is guaranteed that the on-tangle part of the chain state always contains exactly one UTXO with the *chain token* on it. This UTXO contains the ID of the *state transaction*. The *state transaction* defined by the chain token and the *state section* in it represents the tip of the chain anchors.

The retrieval and validation of the on-tangle state for the chain with a given *chain address* and a given *chain color* consists of:

- retrieving all UTXOs contained in the chain address, i.e. the state of the *chain account*.
- finding the UTXO with the *chain token* in it
- retrieving the value transaction *vtx* whose hash is contained in the UTXO with the *chain token*
- Parsing *vtx* as a SC transaction (parsing the data payload of it into its sections).
- Checking if it has a valid state section
- Checking if the *chainColor* in the state section is the same as the color of the chain token

Note that it is also possible to determine if a given address is the address of some contract chain with given chain color by checking all UTXOs with one colored token in it, parsing the transaction referenced by the UTXO and checking its state section for the expected color. For a valid chain address, exactly one UTXO will pass the test.

Off-tangle state

Virtual state

Logically, the off-tangle part of the chain state is a collection of key/value pairs. The *key* is any byte array and the *value* is also a byte array. *Key* is always unique in the state and is a non-empty value.

We call this view of the chain state a *virtual state* or (sometimes called a *data state*).

To partition the virtual state of the chain into sub-states for each SC, we use prefixing of corresponding keys with the 4 bytes of the contract *hname*.

To modify the *virtual state* one can apply a sequence of *mutations*. Each *mutation* is one of the following:

- *add* a new key/value pair
- *update* the value for some key
- *delete* a key/value pair

Note that the sequence of *mutations* applied to some *virtual state* produces a new *virtual state* and this process is deterministic.

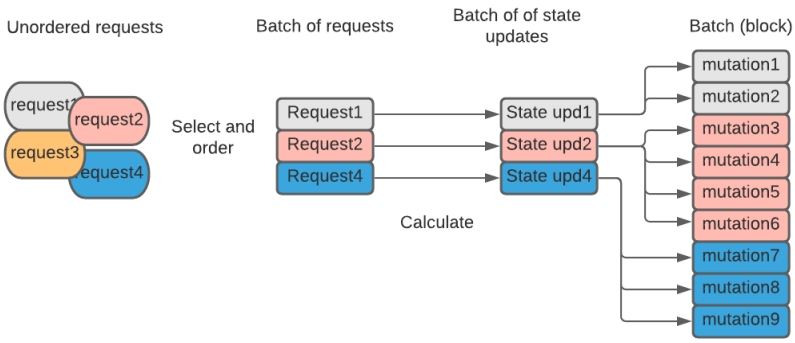
In the ISCP the following objects are used:

- A *State update,* containing:
    - A sequence of mutations
    - *Timestamp*: int64 of Unix nanoseconds. The timestamp of the first state update in the block is set by the consensus process (see below). The timestamp of each subsequent state update in the block is exactly 1 nanosecond larger than the previous. The consistency of timestamps is enforced: the timestamp of the state update must be strictly larger than the timestamp of the *virtual state* it is applied to.
- A *Block* is a sequence of *state updates*. It has the following properties:
    - sequence of state updates
    - *Timestamp.* This is equal to the timestamp of the last state update in the sequence
    - *Anchor transaction ID*: a 32-byte hash referencing the state transaction on the tangle
    - *Block index*: uint32. the sequence number in the sequence of blocks which results in the *virtual state*

Blocks and state updates in the end represent sequences of mutations of the *virtual state*.

At any given moment, the chain has an off-tangle part of its state, which can be seen as a *virtual state*. Processing of each request results in a *state update* (sequence of mutations):

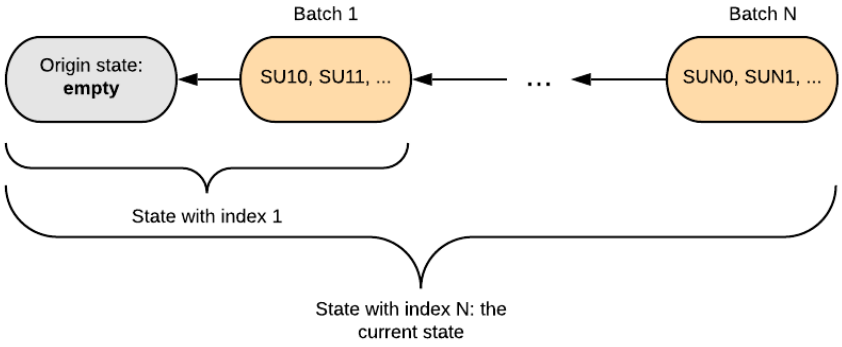
- Each SC on the chain has a program associated with it, the *SC program*
- Inputs to the program are:
    - its partition of the current *virtual state* and a request with attached data
    - Its state of accounts with colored tokens
    - Its *funds*, the colored tokens at the disposal of the SC.
- The output of the program is a *state update*, a sequence of mutations
- The *state update* is applied to the current *virtual state* which results in the next *virtual state*



At each point in time the current *virtual state* may be represented as a sequence of state updates, applied to the *origin state*. The origin state is an empty state, a *virtual state* without any key/value pairs.

In ISCP, state updates are grouped into *blocks*. A *block* is logically also a sequence of mutations.

Requests are processed by the SC in *batches* of requests, not one by one. Each batch of requests is an ordered collection of requests. State updates from the batch make up a bigger state update, the *block* [of state updates].



The *virtual state* of the chain has the following properties:

- *Block index*: the index of the last applied batch of state updates.
- *Timestamp*: int64, Unix timestamp in nanoseconds. This is equal to the timestamp of the last applied block
- *State hash*: see below

The *State hash* is recursively computed from the hash of the previous state and the hashed value of the block as follows:

$$StateHash(virtualState_N) = hash(StateHash(virtualState_{N-1}), hash(block_N))$$

The only computationally feasible (but expensive) way to compute the hash of the virtual state is to run the calculations from the origin. For this reason the state hash of the current virtual state is always stored together with it.

(Note: in the future hashing of the virtual state will based on Merkle trees)

Solid state

A s*olid state* is a representation of the current state of the chain in the database.

The *solid state* is represented in the database by:

- The *virtual state*, the collection of key/value pairs. The *virtual state* can be queried using key/value database operations: ‘get value by key’, ‘check existence of the key’, ‘iterate over keys in lexicographical order’. This is partitioned by smart contract and each smart contract can access only its own partition. The partitioning is handled by the call context behind the scenes and is transparent to the smart contract.
- Complete history of the blocks that resulted in the current virtual state.

To validate the current virtual state of the contract chain one has to run all the blocks and apply them starting from the empty state. The hash of the valid result of this activity should be equal to the hash of the current virtual state.

(Note: in the future validation of the virtual state will be done with Merkle trees)

Anchoring the state of the chain

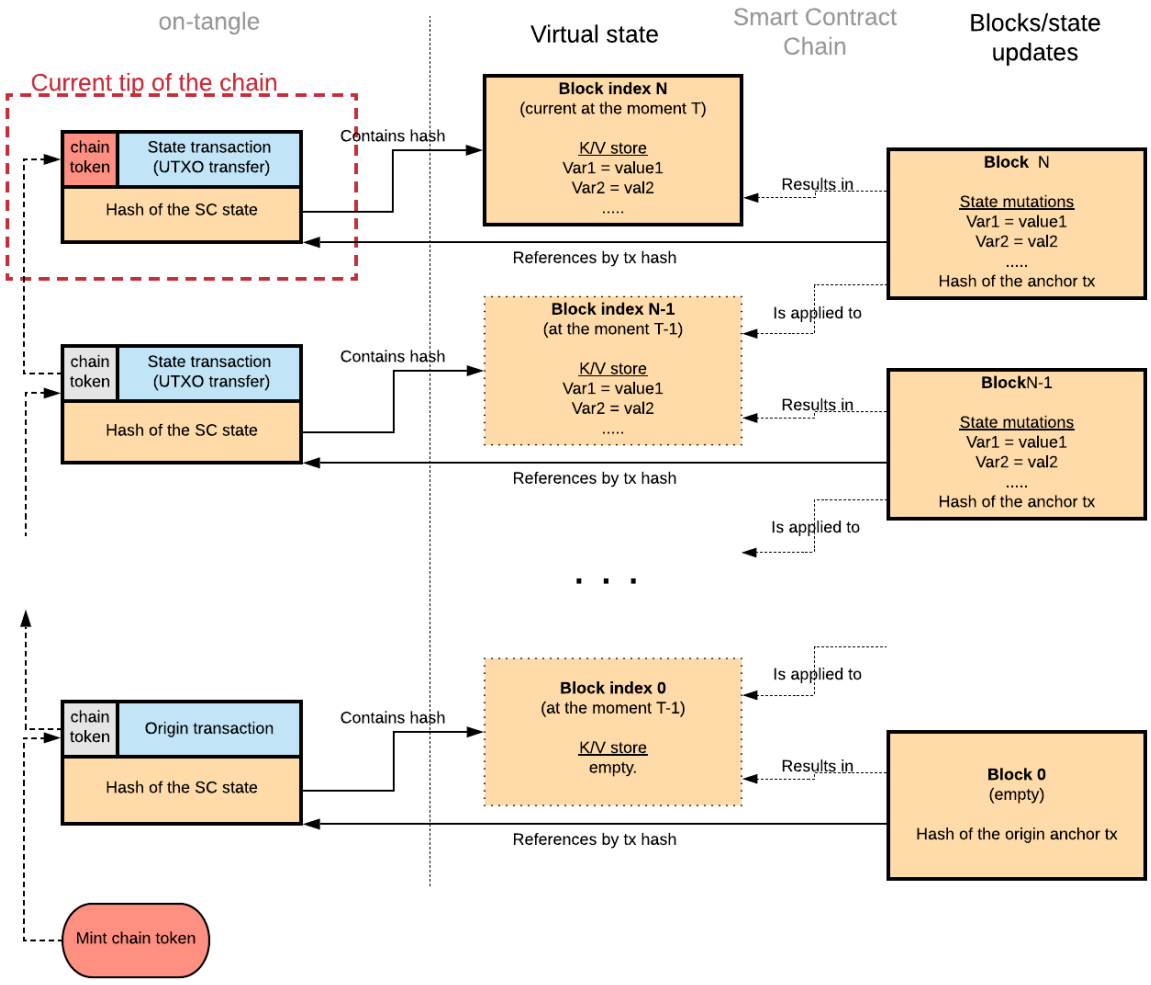
By ‘*anchoring the state’* we mean two things:

1. storing the hash of the current *virtual state* in the state section of the state transaction. Once confirmed in the Tangle, this locks the *virtual state* and makes it immutable.
2. storing the ID of the anchor transaction as part of the last block which resulted in the current state.

This may seem like a loop but it is not. The state of the smart contract is updated in an append-only manner, block by block. Each update to the current state goes through the following steps:

1. The smart contract program(s) while processing the request
    a. calculates (a) the block of updates to the current *virtual state* and (b) the next state transaction with token transfers etc.
    b. puts the hash of the next state transaction into the block
    c. posts the transaction to the Tangle and waits for confirmation.
    d. applies the block to the current *virtual state*. This results in a new *virtual state*.

e. stores the new virtual state and the block as the *solid state* in the database, the SC ledger



Merkle proofs

The state anchoring scheme described above is cryptographically secure, however to prove a particular key/value pair is part of the virtual state is quite a costly task.

In the future we will implement a much more optimal solution: Merkle trees will be used to represent proof of inclusion of state elements into the root (tip) state, i.e. into the root hash of the current state. This approach will enable inclusion proofs at log(N) cost.

In further developments we intend to follow guidelines of how Merkle proofs are used in Ethereum.

Consensus on the state

The *virtual state* is a result of applying a sequence of blocks (state updates) to the empty state. This is a deterministic process; the contract chain is a blockchain.

The chain of blocks has following properties:

- any block can be validated by checking its immutable anchor on the tangle
- anyone can validate the *virtual state* by the sequence of blocks
- anyone can reconstruct the *virtual state* from the sequence of blocks.

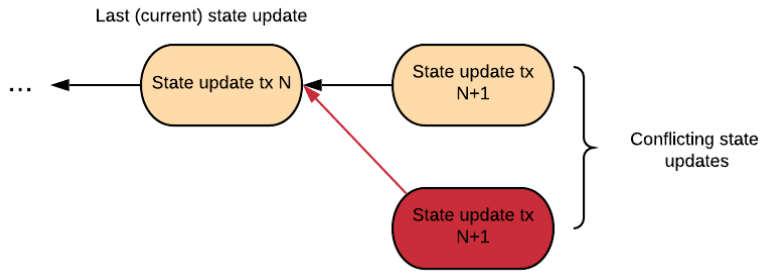
The off-tangle part of the state, the chain of blocks, is stored on the distributed database run by the nodes which run the contact chain, the *committee nodes* (see below).

This gives rise to the following question: What guarantees are there that the distributed nodes always have a final consensus on the current chain? In other words, how can we guarantee that the chain never forks and that all participants of the chain see it as an *objective state*?

To address this problem, in ISCP we associate each smart contract with a non-fungible token called a *chain token* (see also Address and color of the contract chain). The *chain token* is always moved by the anchor transaction to the **same chain address** (but new UTXO) whenever a new block is added to the chain. This means that it is always contained by the last anchor transaction. See detail of state transaction in Committee of the chain.

The guarantee that the chain doesn't fork comes from the fact that only one state transaction can contain an UTXO with the *chain token*, namely the non-fungible token with the color of the chain on the ledger, forever. Furthermore, at any point in time the network has a consensus as to which address contains the *chain token*: the *chain address*. If, for some reason, we want to change the *chain address*, moving the *chain token* to another address doesn't change the ultimate identity of the chain.

Any attempt to fork the chain of transactions and thereby the anchored *virtual state* will result in a double spend, i.e. a conflict. This is not allowed by the underlying token ledger, the Value Tangle.



This property ensures global consensus on the *virtual state* of the contract chain and of each smart contract run on it.

Synchronizing (‘synching’) between nodes

Any party that has access to the chain of blocks and the tangle can reconstruct the valid *virtual state* of the chain. Each committee node keeps its own copy of the *solid state*. This means that the *solid state* of the smart contract is maintained in the distributed database. Nodes which are left behind for any reason can synch their database to the current state using the procedure below.

Let’s say some node 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 node does the following in a loop while L < N:

- The node asks for the block with the index L+1 from other nodes. Some of these have that block and send it to the requesting node
- The node takes the ID of the anchor transaction from the block received upon request and asks the Value Tangle for that confirmed transaction.
- The Value Tangle sends the transaction to the node. This contains the hash of the virtual state L+1. The node does the following:
    a) applies the block (the sequence of mutations) to the current *virtual state* L
    b) takes hash of the resulting tentative *virtual state*
    c) compares state hash of the *tentative virtual state* with the state hash stored in the transaction
    d) If the two hashes are equal, the node stores a tentative *virtual state* and the block it received, into the database as the *solid state* L+1. If these hashes are not equal, the database is inconsistent.
- Now the node has valid solid state L+1. If it is not the current state, it repeats the process until it is.

In this way any node that is entitled to receive requested blocks from other nodes can keep its state in sync with the contract chain even without it being able to participate in the calculation of new state updates.

Nodes which only synchronize their states without calculating them are called *access nodes*.

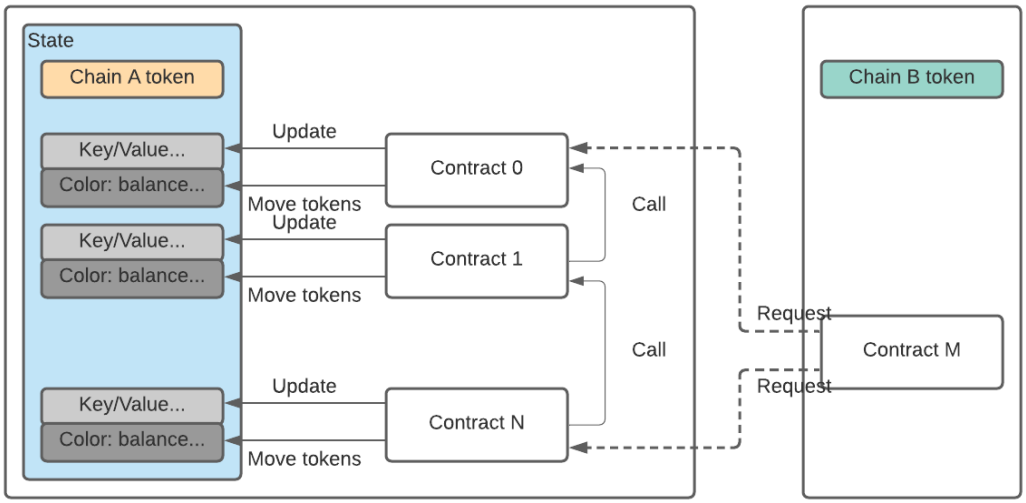
Access nodes may be exposed to the public network to provide access to the SC state to external users, while at the same time shielding non-public *committee nodes* from possible DDoS and other attacks.

Smart contract

Structure of the smart contract chain

Each contract chain has the following properties:

- Chain token and chain color
- Chain address
- Owner ID of the chain: the privileged user of a core smart contract entitled to perform certain administrative actions, such as initiating rotation of the committee. The chain can be self-owned, i.e. the administrative actions can be undertaken only by that specific smart contract on the chain
- One or many smart contracts, deployed on the chain
- The state of the chain



The contract chain is deployed on the Tangle as the *origin transaction* signed by the *chain owner address*. The hash of the origin transaction becomes the *SC color*. Please see details on chain deployment and bootstrap below.

Structure of the smart contract

Smart contracts are software agents 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*.

State partitions

Each smart contract has a partition of the chain state associated with it.

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

Each smart contract (instance) has the following properties:

- *ContractID.* This consists of the *chain ID* and the *contract hname*
- *SC program*, the binary loaded in the corresponding interpreter called a *virtual machine* or *VM*. The binary is always stored in the on-chain registry, maintained by the built-in *blob* smart contract. See also Virtual machine.
- *SC state*, a partition of the chain's virtual state, associated with the smart contract
- *SC account*, an on-chain account with colored funds under control of the smart contract

A smart contract is deployed on the chain by a special built-in smart contract, the *root contract*. A smart contract may deploy another smart contract by calling the *root contract* (see below).

Entry points; Views.

The smart contract program exposes *entry points*, these being the functions that can be called. Each *entry point* is identified within a smart contract with the *hname* of its name. 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 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:

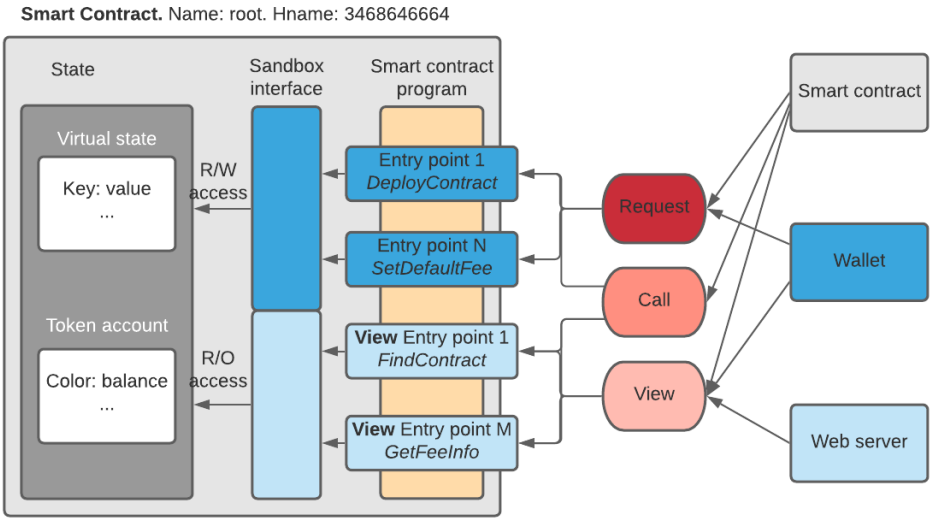
- Has read/write access to its own partition of the state on the chain
- May call another contract on the same chain synchronously (on the same state of the chain). The target of such calls may either be full entry points or views
- May pass and receive funds as *transfers* in calls to full entry points 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 several 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
- ***Governance*** smart contract (not implemented in the *alpha* version): this is responsible for certain functions relating to the chain itself. These include managing authorisations to

functions (such as the authorisation to deploy a new contract), setting minimum fees, moving a chain to another committee, maintaining an optional governance body with on-chain voting rights for decisions and so on.

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 called 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 *requests*. Each request has the following logical structure:

arguments, funds → contract ID, (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
- *contract ID* represents the target smart contract on a particular chain.

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 value 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*.
- Smart contracts themselves also can send requests to other smart contracts. In this case the sender will be identified using the *contract ID* 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-tangle**. Requests are represented by request sections in a payload of value transactions. 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 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 Tangle). It cannot be lost and is fully auditable
- One smart contract transaction can contain 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-atomic* property (see below) there's a guarantee that once sent, a package of requests to many SCs will all be processed eventually.
- sending a 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 smart contract transactions **non-atomic**.

There is a requirement in ISCP for SC transactions to be **quasi-atomic**.

By *quasi-atomic transactions* we refer to a requirement that **any confirmed request must be processed exactly once**. Under no circumstances can a confirmed request be lost, or processed more than once. The material below describes how the backlog of requests is processed in ISCP to satisfy this requirement.

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

State transition

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 transaction to the Tangle. 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* 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 Tangle, while the state updates contained in the block are applied to the current virtual state, as follows:

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

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.
(Note that making the SC Program completely deterministic is not always easy because some runtimes may produce different results at the bit level, even though they may be logically equivalent)

Once all the inputs $State_N,\ Request$ and the $SCProgram$itself is immutable on the tangle, 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 'aBFT' setting, to determine the consensus arrived at by the committee of different processors.

Let's say we have a committee of Nprocessors. 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: 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 often is the case $T = floor(2N/3) + 1$ is 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 ("bizantine").

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.

Committee

Each contract chain is run by a *committee* of Wasp nodes. These are the processors mentioned above: a minimum of 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 Tangle, i.e. committee nodes are usually connected to different IOTA nodes. The reason for this is to not make access to the Tangle 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 aspect 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 *request section* is one of the inputs to the program. Request transactions may reach Wasp nodes in any arbitrary order and with arbitrary delays (even if these are usually close to the network latency).

We assume the committee nodes have consensus on the main thing that matters: the last transaction in the chain of anchor transactions. This ensures that all nodes always have consensus on the current *virtual state* of the contract chain. This assumption is realistic if

network propagation latency is not very large due to the *chain token* mechanism described above.

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

- Balance of the chain address, the UTXOs contained in the chain address
- Timestamp to be used for the next state
- Ordered batch of requests to be processed
- Address where node fees for processing the request must be sent (if enabled)

In order to achieve a bigger throughput, the committee picks requests from the 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, where it matters, the order of the requests matters. Where the order of requests matters is in cases where a different order of requests would give rise to different resulting blocks and different final *virtual states* with no consensus on the result.

After at least a quorum of committee nodes have a consensus on the above, honest committee members will always produce identical results of calculations. See also The consensus algorithm.

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, among other things, 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 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 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 (Pollen release) 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 aBFT algorithms for finite sets of participants.

The *consensus operator* object is responsible for the consensus on the state transition in the contract chain on each Wasp node.

During execution of the consensus protocol the committee nodes (*consensus operators*) are communicating between each other via asynchronous messaging over UDP or direct TCP or connections which were established when the committee was activated on the network.

The context of the *consensus* object consists of:

- Current *state* of the chain: the *anchor transaction* and the *virtual state*. The current state is a consensus among all nodes because it is coming from the Tangle.
- Current *backlog of requests*: an unordered set of request sections. Each request section is represented by the request transaction and the index of the section in the payload. With this data, the node can access all data attached to the request and all tokens sent by the request transaction to the smart contract. Note that each node may see different backlogs at any particular moment.
- Current state of the *chain address*. This includes all UTXOs contained in the chain address as perceived by the node. Note that nodes can see the state of the account (list of UTXOs) slightly differently due to network delays. It is guaranteed however that all the UTXOs contained in transactions of requests in the backlog are in the state of the account. See details in Backlog.
- Pseudo-random permutation (ordering) of indices of nodes in the committee, i.e. numbers from 0 to N-1. This is calculated by each node deterministically using as a seed the hash of the current state anchor transaction. For this reason, all nodes have a deterministic consensus on the permutation of indices of its peers.

Each state transition changes the context of the *consensus operator*.

Below are the stages of the consensus process.

1. Immediately after state transition, all nodes compute the permutation of indices. Each of them takes the first index in the permutation as an index of the leader. Subsequent indices in the permutation will be used for leader rotation. As a result, the quorum consensus on the first leader is reached within the period of time in which a quorum of nodes passes the state transition.
2. Upon selection of the leader, all committee nodes (except the leader itself) send to the leader a subset from their backlogs, of requests which they can process looking from their perspective (for example these should be not time-locked according to the local clock, or similar considerations). The leader collects the information about which node is ready to process which requests.
3. The leader selects requests from its own backlog based on the current information from other nodes. The selected list includes all requests which satisfy the following criteria:
    a. At least a quorum of nodes has seen the request
    b. The request is not time locked according to the local clock of the leader
    c. The request can be processed according to other criteria
    d. Among the requests which pass criteria a, b, c above, it selects a maximum set of them, which has been seen by the same quorum of nodes. This guarantees any request in the list can be processed by the quorum of peers
4. If the list of selected requests in (3) is not empty, the leader orders them and composes a batch of requests for processing.
5. The leader creates the *computation context* for the VM calculation. The *computation context* contains:

    a. Ordered batch of requests
    b. Timestamp taken from the leader's local clock
    c. Own reward address (if rewards are enabled)
    d. All UTXOs from the chain account balance in its own context
6. The leader initiates the calculations in the committee in exactly the same *computation context*:
    a. Sends *computation context* to all nodes in the committee for calculations
    b. Starts the computation with the VM itself
7. VM computations in each node result in a new anchor transaction and a block candidate. For non-faulty nodes, all results produced will be equal, within a tolerance of one bit.
8. Each node produces a partial signature of the anchor transaction with its own partial private key.
9. Non-leader nodes send to the leader:
    a. The hash of the concatenation of the computation context with the result
    b. The partial signature
10. The leader ignores invalid results, i.e. those in which the hash is different from its own. Optionally, if the context hash is valid and the signature is valid but the result hash is wrong, it is a misbehaving node and this is proof of that fact (*This can be used for the sentinel function*).
11. The leader checks the validity of partial signatures against its own computed transaction. Results with invalid signatures are ignored (alternatively the leader might fill a complaint to some staking/auditing authority because it is a proof of malicious behavior)
12. When the leader collects a quorum of valid partial signatures, it reconstructs the final signature using the threshold signature algorithm and puts it into the transaction.
13. The leader posts the finalized transaction to the Tangle.
14. Each node adds the block it calculated to the list of *pending blocks*. Pending blocks are waiting for the anchor transaction to be confirmed on the Tangle. When this happens, the node commits the block to the database with updated *virtual state*. This results in an updated solid state of the chain.

Each node sets timeouts for the leader operations. The leader is rotated upon expiration of timeouts according to some deterministic permutation known equally to all nodes.

Note that during the whole consensus process:

- Each node calculates its own result, and this result (block candidate and anchor transaction) is not communicated to other committee nodes. This prevents "classroom" attacks (a.k.a. freeloading) when nodes could just copy the results calculated by others without doing calculations itself. To produce the valid final transaction the leader must produce a valid result itself.
- Each node knows the partial public keys of other committee nodes (from the DKG), so that each message can be authenticated, thus preventing man-in-the-middle type of attacks.

- The leader provides an address to which send rewards as a part of the context for each state transition. Normally this is its own address. If a node fails to fulfill its duties as a leader, it will not be rewarded. Apart from this case, rewards will be distributed uniformly on average due to the random nature of the permutation.

Backlog

Above we described the messaging model of the ISCP and formulated the requirement of **quasi-atomic** SC transactions. Software agents other than smart contracts on the same chain invoke functions of a smart contract by sending it a *request on-tangle*.

The requirement of a *quasi-atomic SC transaction* is that the protocol has to guarantee each request sent this way will be processed exactly once by the corresponding SC (assuming of course that the quorum of the SC committee is up).

The backlog of the smart contract is a set of all as yet unprocessed requests to that smart contract. It is contained on-tangle in the SC transactions which contain the corresponding request blocks.

An on-tangle backlog implies that all participants will have eventual consensus on the relevant matter within some relaxation period. It also means the backlog is distributed, tamperproof and immutable (and therefore auditable).

Committee nodes have to be able to do the following under consensus:

- Load the backlog from the Tangle: that is, the complete list of as yet unprocessed requests, without there being the possibility that any are lost
- Prevent the inclusion the backlog of any already processed requests
- Mark the requests as processed thus removing them from the backlog

To be able to track the backlog, we organize SC transactions as described below.

Let's say we want to send a transaction to a SC with target contract ID A. We, the sender, include the corresponding request section in the SC transaction in the following way:

- We add the request section as described above into the SC payload of the transaction being constructed
- We mint exactly one colored token with the output to the chain address A of the target smart contract in the transfer part of the transaction being constructed

The newly minted token is called a *request token*. A request token is a token which has color equal to the hash of the SC transaction containing the request block. For example, let's say the ID of the request is `[3]8v2G3wi1JzDb3vieAR2wZFYnxYqnaeLs9skduPpok5jp`

The hash `8v2G3wi1JzDb3vieAR2wZFYnxYqnaeLs9skduPpok5jp` will represent both the transaction (the ID of that transaction) and the color of any request token, minted in that transaction to the target chain address.

Any SC transaction can contain many request sections for requests to many different target contract chain addresses. Therefore the number of new tokens minted by that transaction will be at least equal to the number of request sections. This is one of the semantic validity criteria of the SC transaction.

If the request transaction for some reason wants to mint more colored tokens, it can do so on top of this. However it can only mint these to addresses that are different to any of the target chain addresses of contained requests. The total number of minted tokens minus the number of blocks is called *free minted tokens*. In most cases this is 0.

Fetching the backlog

The chain address will always have in its balance all UTXOs with request tokens corresponding to requests sent to it on-tangle. So, to fetch the backlog for the contract chain, the node first of all fetches all UTXOs belonging to its own address.
The backlog can be located from retrieved UTXOs in the following way. Let's say a node has a balance of chain address A in the following form:

C1: B1
C2: B2
…

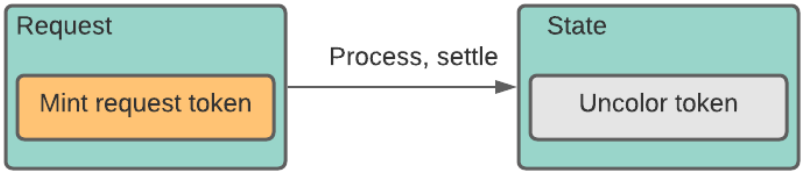
Here Ci are colors (which are not *iota-color*) and Bi is each color's corresponding balance
For each color Ci:

1. The node takes the balance Bi of some color Ci in the chain address.
2. Ci is a transaction hash. The node loads that value transaction and parses it as a SC transaction. If this does not parse correctly, it will skip it.
3. If it is a SC transaction, the node takes all its request sections with target chain address A: {R1, R2, …}
4. If Bi is not equal to the number of request sections with target chain address A, the transaction is skipped (with the exception when Bi is equal to the color of the smart contract; in this case the number of requests must be equal to the number of tokens minus 1, because the SC token is always present in the SC address.)
5. Otherwise all request sections {R1, R2, …} are added to the backlog

Note that for a valid SC transaction each request section must contain exactly one corresponding colored token with the color of the request transaction hash in the balance of the chain address. The origin transaction is an exception.

Removing the request from the backlog

When a request is processed, it must be removed from the backlog.



The request is removed from the backlog by uncoloring its corresponding request token back to the default IOTA color. The UTXO which uncolors the request token is part of the resulting anchor transaction. The token is erased from the backlog automatically even before the VM starts computing the result.

The request token is uncolored by **sending it back to the address of the request sender chain**. This means that you need at least 1 iota token to send the request, even though in the end this is returned and the transaction remains feeless (assuming node fees are disabled).

After the resulting anchor transaction is confirmed on the Tangle, the colored token disappears from the chain balance and therefore, from the backlog. If for some reason the produced anchor transaction is rejected by the Tangle, the request stays in the backlog.

In this way we ensure atomicity of state transitions and the maintenance of the on-tangle backlog.

The whole backlog processing guarantees that each request is processed exactly once, hence *quasi-atomic SC transactions*.

Deployment of the contract chain

Deployment of the contract chain consist on the following steps:

- Selecting of the committee
- DKG
- Creating the *origin transaction* 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 *the owner* or *initiator*. *The owner* is usually represented by its address; specifically by its address on the Tangle and the

private key(s) which control that address. The owner sends the *init* request and the *root* contract stores this as the *chain owner ID* in the state of the chain. The chain owner 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*. The chain address, generated during DKG, is an input for the process.

The *origin transaction* is an SC transaction with the following characteristics:

- it mints 1 new colored token into the *chain address*. That token is a *chain token* and will represent the *contact chain* during its lifetime
- It contains a state section with:
    - Block index 0
    - chain color *nil* (all 0)
    - State hash equal to the hash of the empty *virtual state* (deterministically known).

That is, *the origin transaction* mints 1 new token with the color of origin transaction hash into the address of new SC. The hash of the origin transaction will be the color code of the non-fungible *chain token*.

Init request transaction

Immediately after confirming the origin transaction, an *init* request transaction is created by the same initiator address and confirmed on the tangle. This will contain a request section with an *init* request code to the built in *root smart contract*.

The *init* request will contain initialization data for the new contract chain namely: *chain color* (hash of the origin transaction) and *description*. 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 Tangle, the chain needs to be activated on the committee (Wasp) nodes. Activation involves the following steps:

- Placing the *bootup record* of the chain into the registry of each committee node. The *bootup record* contains all the information needed for that node to boot the chain, namely: chain address, chain color and list of committee nodes.
- Creating a *committee* object which runs consensus and other committee functions. The parameter for this object is the *bootup record*. The node starts its activity in the committee from subscribing to the backlog from the IOTA node and synching the state of each node with other committee nodes.

- The *committee* object receives an *init* request, the only one in the backlog at that moment. The *root* contract runs the request and produces *bock #0* which contains the initial state with all built-in smart contracts deployed.

All chains (committees) are activated automatically upon node start from bootup records stored in the registry of the node.

Virtual machine

VM abstraction

The SC program is represented by its binary code and the hash of this binary code (the *program hash*). Each chain contains many smart contracts and therefore many VMs running those programs.

The binary of the program is loaded into the interpreter, the *virtual machine (VM)*. The interpreter instance with the binary code of the program loaded into it we call the *processor*.

Each binary of a program is contained in the on-chain registry maintained by the *blob* smart contract. The key to the registry is the hash of the blob data.

The *root contract* maintains a registry of smart contracts deployed on the chain in the state of the chain. The registry links contracts with the binary code of the program contained in the *blob* contract. In this way, the smart contract program is itself a part of the immutable elements of the contract chain.

Each *processor* implements an abstract interface which exposes only the function *GetEntryPoint* for finding an *entry point* corresponding to the specific *entry point code*. For this reason, we assume that each callable function of a smart contract program has a corresponding *entry point* in the *processor.*

Computations of the next state are initiated by *calling* the *entry point*. Smart contact programs can locate and call entry points of their siblings on the same chain.

A parameter of each call to the VM processor (SC program) is the interface to the *Sandbox*.

The abstract *sandbox* interface is the only parameter passed to the call to the entry point. Behind the *sandbox* interface there is a global state of the chain as well as a local call context. The smart contract program can access this data only via the sandbox. The *sandbox* implementation ensures security and privacy of the chain data to external and potentially non-secure smart contract programs.

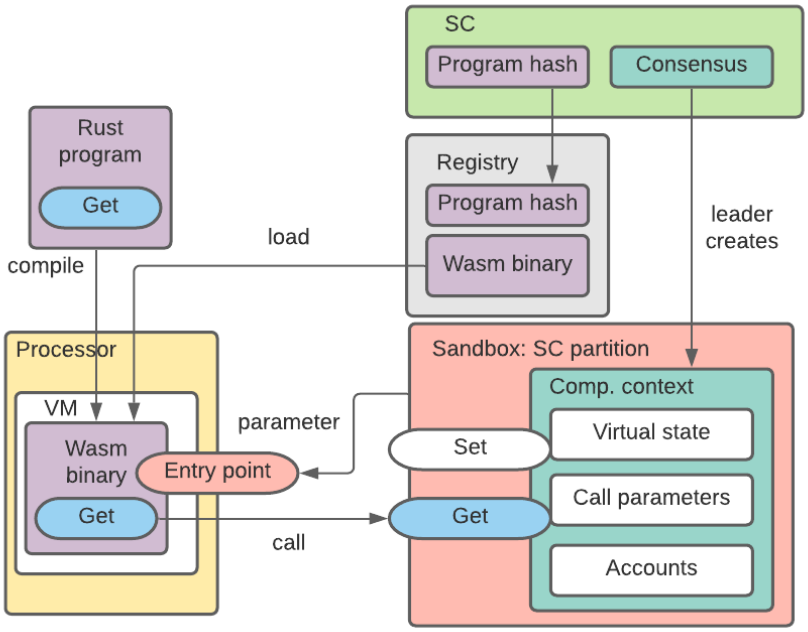
The *sandbox* interface and its implementation must be *deterministic*, i.e. the result of the call to the interface must be a deterministic function of the state of the chain and call parameters.

We refer to the above as '*VM abstraction'* because the interfaces are agnostic to the specific type of *VM/processor* and the binary. They are also agnostic about the specific implementation of the *sandbox*: for any SC program this is just an abstract interface.

The Wasp node can support several types of VMs, if needed. The *processor* is created by the *VMConstructor* which takes *VM type* and other metadata of *VM* and the program binary as parameters.

We will be using WebAssembly (*wasm*) for ISCP VM, so the binary code will be *wasm* code. The different types of *VM* in the *VM abstraction* can be used for different versions of *wasm* interpreters as well as non-wasm VMs, such as EVM.

We assume *wasm* is the binary and *wasm* interpreter is used as the *VM* for SC programs.



SC programming environment

Theoretically any high level language can be used to write SC programs. 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

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 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

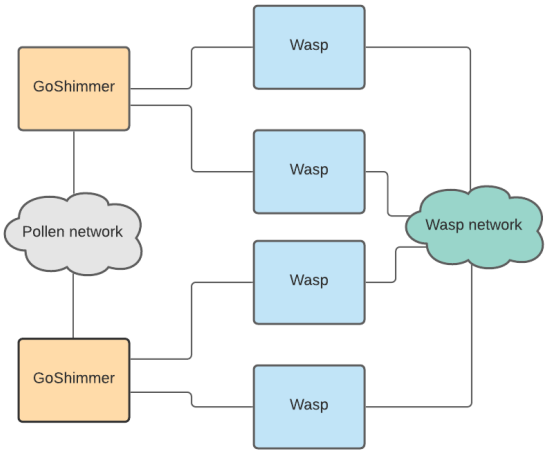
The *root contract* finds binary code in the *blob* contract and loads it into the VM processor. After that the root processor immediately calls the *init* entry point of the new VM processor with the parameters from the *DeployContract* call.

Wasp node

The ISCP is run by a network of Wasp nodes. To get access to the IOTA ledger on the Tangle, each Wasp node connects to a Goshimmer node on the Pollen network.

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 using a specialized asynchronous protocol:

- On the Goshimmer side this is implemented by the *WaspConn* plugin
- on the Wasp side it is implemented by the *NodeConn* plugin

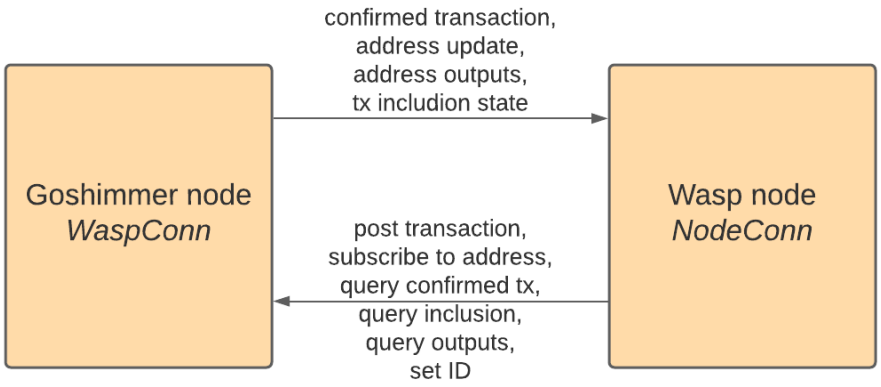
The protocol exchanges messages using TCP or UDP as the transport.

The protocol between Goshimmer and Wasp is asynchronous and session-less. This means that each party sends a message to another party and does not wait for the result or confirmation of delivery, just as in gossip. Goshimmer and Wasp do not exchange any information about the success or failure of requests.

As a general rule, if the Wasp node does not receive the requested information after some timeout, it repeats the request: the *pull*. The queried information always comes without any contextual information about the query.

The Goshimmer side of the protocol, *WaspConn,* deals only with value transactions, and is not aware of the SC transaction payload at all, i.e. from the perspective of Goshimmer and WaspConn the SC payload is just a data chunk. It is up to the Wasp side, the *NodeConn*, to parse incoming value transactions into SC transactions and serialize/deserialize SC payloads and validate them.

Wasp has *Goshimmer* as a dependency. *Goshimmer* is not dependent on Wasp.



Messages sent from Wasp to Goshimmer

Name	Payload	Function
waspToNodeTransaction	Value transaction to post to the tangle.	The SC transaction is produced by the committee of nodes. The leader of the round finalizes the transaction, produces the value transaction and sends it to Goshimmer to post on the tangle. Goshimmer posts it to the tangle.
waspToNodeSubscribe	chain address	Wasp instructs the Goshimmer node that it is interested in any new outputs confirmed for the address. Goshimmer inserts the address in the list of subscriptions. It also retrieves and sends to the node the backlog of the address (see Backlog handling below)
waspToNodeGetConfirmedTransaction	Transaction ID	Wasp asks for a value transaction. If a transaction is found and confirmed,

		Goshimmer sends it to the Wasp node.
waspToNodeGetTxInclusionLevel	Transaction ID chain address	Wasp asks for the inclusion information about the transaction, if it has outputs in the chain address. If found, it sends one of 3 conditions to Wasp: 'booked', 'confirmed', 'rejected'
waspToNodeGetOutputs	chain address	Wasp asks for confirmed UTXOs from the node. Node sends these if any.
waspToNodeSetId	Wasp's unique ID	Sends own ID for tracing and logging

Messages send from Goshimmer to Wasp

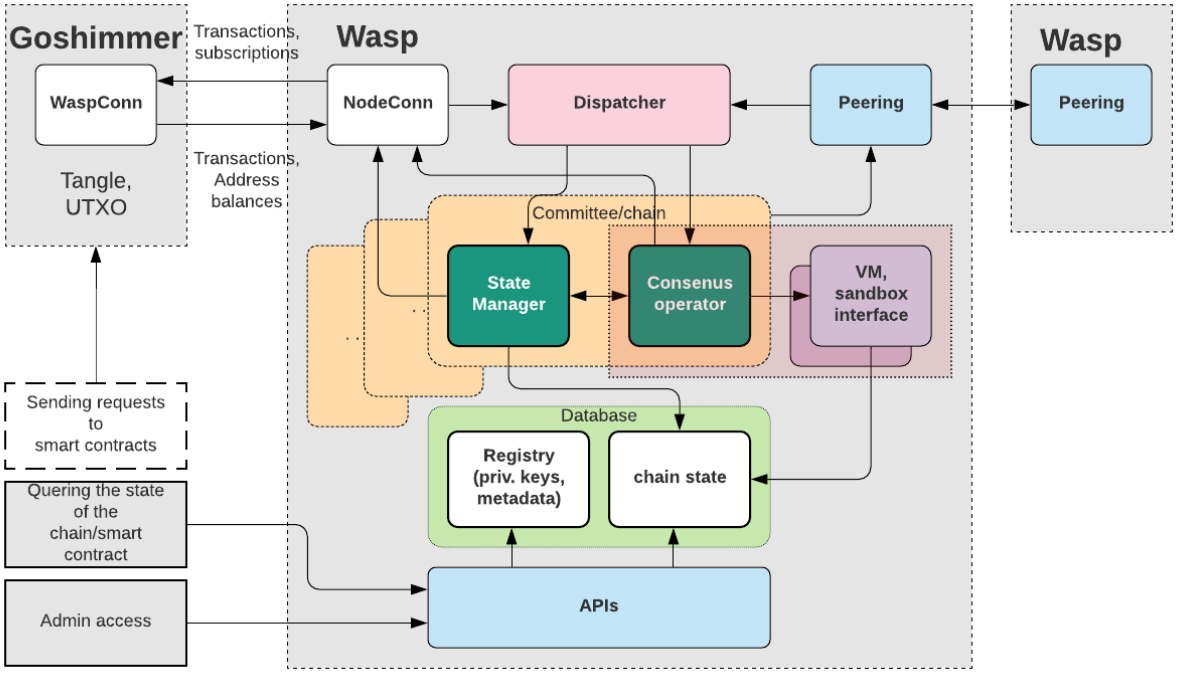
The following are messages sent by Goshimmer to Wasp as a response to incoming messages (the exception is sending subscribed information).

Name	**Payload**	**Function**
waspFromNodeConfirmedTransaction	Confirmed value transaction	Goshimmer sends confirmed value transaction as a response to *waspToNodeGetConfirmedTransaction*
waspFromNodeAddressUpdate	Chain address Balances Value Transaction	Node sends message - upon new transactions where the output to the subscribed address is confirmed - when the backlog of the address is pushed to Wasp. The message contains the address to which information is related, the value transaction in question and all confirmed outputs of the address (without repeating the address part)

waspFromNodeAddressOutputs	Chain address Balances	Node sends this upon request from Wasp *waspToNodeGetOutputs*. It includes all confirmed outputs (without repeating the address)
waspFromNodeTransactionInclusionState	Transaction ID chain address One of: 'booked', 'confirmed', 'rejected'	Response to *waspToNodeGetTxInclusionLevel*

Modules of the Wasp node

A Wasp node can participate in running many smart contracts. It communicates with other Wasp nodes to collectively and in a distributed manner run programs of smart contracts.



Here are the main modules of the Wasp node.

Module	**Function**
peering	Responsible for maintaining persistent connections with other peers in the Wasp network and sending messages between them. Ensures one TCP connection between

	two peers no matter how many committees they are both participating in.
dispatcher	Responsible for routing the messages from other peers and the Goshimmer node to the respective chain object the node is engaged in. The dispatcher also parses value transactions to SC transactions and *vice versa*. It analyzes all sections in the SC transaction and converts them into messages to the corresponding chain object.
nodeconn	Responsible for maintaining connection and protocol with the goshimmer (the *waspconn* plugin on the goshimmer side)
database	Responsible for storing registry data and smart contract state data. The 'Badger' key/value database is used. This is partitioned into partitions, one for each chain, and each chain is partitioned into smart contract partitions.
committees	The registry of *committee* objects, one per *contract chain.*
Committee object	Object responsible for consensus on the state of a particular contract chain. One such object is created per active chain. Contains *state manager* object and *consensus operator* object.
state manager object	One object per chain. Performs validation of block candidates by anchor transactions and commits new states into the database (solidifies). Keeps state synced with other committee nodes. Informs consensus operator about state transitions.
Consensus operator	Keeps backlog of requests and runs distributed consensus between committee nodes (see The consensus algorithm). As a leader, creates a computation context for other nodes. Calls SC program via

	processor/VM/sandbox interface to calculate next state update etc. Sends calculated pending batch to the *state manager*
processor/VM	VM wrapped into the *processor* interface. Contains a wasm program loaded and dynamically linked. Is called to find an entry point and run it each time the *consensus operator* object needs to calculate the next state.
webapi	Implements external APIs

On-chain accounts

Each chain maintains a set of accounts on it, controlled by identified entities on the ISCP network. So, the on-chain entities, the smart contracts, use an account based model (metaphor) rather than underlying UTXO. The entities are securely identified as senders of requests.

There are two types of securely identified entities on the ISCP network:

- Addresses on the Value Tangle. The request is securely identified as sent from the address by the fact that the request transaction is spending funds from that address, i.e. signature
- Smart contracts. The request is identified as sent from the smart contract ID, if the transaction contains spent funds (and signature) from the address (chain ID) in the contract ID and the *senderContractHname* field in the request section.

We introduce an identifier type, which encompases both types of senders, *contract ID* and *address* into one: the ***agent ID***. Detailed definition of types can be found here.

Agent ID is first interpreted as *contract ID*. If contract ID contains *hname* 0 (reserved), it is treated as an address.

On-chain accounts are identified by the *agent ID* of their owners. So, the account may be controlled by an L1 address or by a smart contract on the same or another chain. The core *accounts* contract checks if the sender of the request is equal to the *agent ID*, the owner of the account, and if so allows it to move funds from it to any other location.

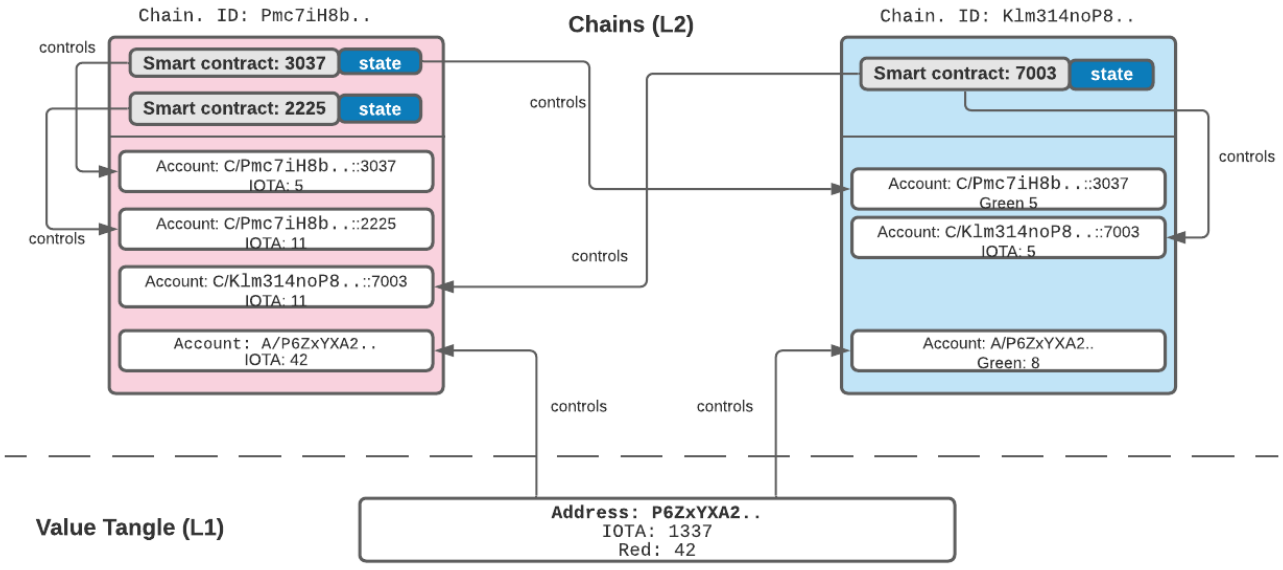
The balance of the account is colored balances: a collection of *color: amount* pairs.
The *accounts* contract has the following entry points:

- *Deposit:* deposits sent funds into the account owned by the sender of the request

- *WithdrawToAddress:* sends funds to the sender, an address, from the account controlled by it
- *WithdrawToChain*: sends funds to the sender, as smart contract, from the account controlled by it to its account on its native chain
- *GetBalance* is a view which allows one to examine balances belonging to any *agent ID* on the chain
- *GetAccounts* is a view which returns all *agent IDs* of non-empty accounts of the chain
- *GetTotalAssets* is a view which returns total balances of colored tokens contained on the chain. It will always be equal to the sum of accounts.

The consistency of the on-chain account ledger is validated by checking if three colored balances are equal:

- The *TotalAssets*
- The sum of all the rest of the accounts, not equal to *TotalAssetsAccount*
- The total balance of the only UTXO which is contained in the chain address and which contains the non-fungible chain token.



The picture above illustrates an example situation.
There are two chains deployed, with respective IDs `Pmc7iH8b..` and `Klm314noP8...`
The pink chain `Pmc7iH8b..` has two smart contracts on it (`3037` and `2225`) and the blue chain `Klm314noP8..` has one smart contract (`7003`).

The Value Tangle ledger has 1 address `P6ZxYXA2...`
The address `P6ZxYXA2..` controls 1337 iotas and 42 red tokens on the Value Tangle ledger. The same address also controls 42 iotas on the pink chain and 8 green tokens on the blue chain.

So, the owner of the private key behind the address controls 3 different accounts: 1 on the L1 ledger (the Value Tangle) and 2 accounts on 2 different chains on L2.

Same time, smart contract `7003` on the blue chain has 5 iotas on its native chain and controls 11 iotas on the pink chain.

Note that “control over account” means the entity which has the private key can move funds from it. For an ordinary address it means its private key. For a smart contract it means the private keys of the committee which runs the chain (the smart contract program can only be executed with those private keys).