Massa

Massa Labs

Dec 07, 2022

GENERAL DOCUMENTATION

Massa is a truly decentralized blockchain controlled by thousands of people. With the breakthrough multithreaded technology, we're set for mass adoption.

CHAPTER

ONE

INTRODUCTION

Massa is a new blockchain reaching a high transaction throughput in a decentralized network. Our research is published in this technical paper. It shows that throughput of 10'000 transactions per second is reached even in a fully decentralized network with thousands of nodes.

An easy-to-read blog post introduction with videos is written here.

CHAPTER

TWO

GENERAL DOCUMENTATION

A general overview of the global architecture of a Massa Node is given *here*.

Massa introduces several new features that enables new applications and adds a layer of security to your decentralized applications:

- *Massa's decentralized web* allows you to truly secure decentralized applications by storing your dApps directly on Massa blockchain.
- *Massa's autonomous Smart Contracts* add a layer of security and reliability to your decentralized applications, by allowing your smart contracts to wake up by themselves and perform arbitrary operations.

CHAPTER

THREE

TESTNET

As decentralization is our core value, we would like to help you start running a Massa node on our testnet. You'll find a full tutorial on how to install and stake with your node on our testnet *here*.

CHAPTER

FOUR

WEB3 DEVELOPERS

If you want to build on the Massa blockchain, we recommend the following ressources:

- *Massa's smart-contracts* will get you through the various steps needed to compile and send smart-contracts to the Massa blockchain.
- *Massa's decentralized web* will show you how Massa blockchain can be used to host decentralized websites.
- *massa-web3* is a collection of useful resources for the frontend development of your decentralized application.
- *Types* is a collection of useful types for smart-contracts.
- *Local network generation* will get you through all the steps required to launch a local Massa network.
- *Useful resources* is a collection of useful resources for smart-contract development.
- *External resources* is a collection of useful resources developed by external contributors.

CHAPTER

FIVE

TECHNICAL RESOURCES

Here you'll find the documentation on the various JSON-RPC API endpoints exposed by a Massa node.

CHAPTER

SIX

COMMUNITY

If you have any questions regarding the project or wish to discuss technical aspects in more depth, feel free to reachout to us in our community channels:

- Telegram
- Twitter
- Discord

CHAPTER

SEVEN

CONTENTS

7.1 Massa's Architecture

This section gives a general overview of the Massa's architecture.

7.1.1 Introduction

We will describe in this document the global architecture of a Massa Node, from the ground up, and introduce relevant definitions and concepts.

The goal of the Massa network is to build a consensus between **nodes** to gather and order **blocks** that contains ordered lists of **operations**. An operation ultimate purpose once executed is to act as transitions for the global network state, called the **ledger**.

Operations are produced by external clients and sent to the Massa network via a node. Some operations are containing code to be run as **smart contracts**, enabling complex programmatic modifications of the ledger. Nodes will gather all the pending operations and group them to produce blocks. Each block contains a finite set of operations, limited by the fact that each block has a limited amount of space available to store operations. Traditional blockchains will then typically link blocks one after the other (including a hash of the previous block in the block header), to materialize their temporal ordering. However, unlike traditional blockchains, Massa blocks are not simply chained one after the other, but organized into a more complex spatio-temporal structure, which allows for parallelization and increased performances.

Instead of one chain, there are several threads (T=32) of chains running in parallel, with blocks equally spread on each thread over time, and stored inside **slots** that are spaced at fixed time intervals:

The time between two slots located on the same thread is called a **period** and lasts 16s (conventionally called t_0). Corresponding slots in threads are slightly shifted in time relative to one another, by one period divided by the number of threads, which is 16s/32 = 0.5s, so that a period contains exactly 32 slots equally spaced over the 32 threads. A **cycle** is defined as the succession of 128 periods and so lasts a bit more than 34min. Periods are numbered by increments of one, so can be used together with a thread number to uniquely identify a block slot. Period 0 is the genesis and contains genesis blocks with no parents.

The job of the Massa nodes network is to essentially collectively fill up slots with valid blocks. To do so, at each interval of 0.5s, a specific node in the network is elected to be allowed to create a block (more about the selection process below, and the proof of stake sybil resistance mechanism), and will be rewarded if it creates a valid block in time. It is also possible that a node misses its opportunity to create the block, in which case the slot will remain empty (this is called a **block miss**).

In traditional blockchains, blocks are simply referencing their unique parent, forming a chain. In the case of Massa, each block is referencing one parent block in each thread (so, 32 parents). Here is an example illustrated with one particular block:

Let's introduce some relevant definitions and concepts generally necessary to understand how the Massa network operates. We will then explain the node architecture and how the whole system works.

Address

Each account in Massa has a public and private key associated with it. This is how messages can be signed and identity enforced.

The address of an account is simply the hash of its public key.

Ledger

The ledger is a map that stores a global mapping between addresses and information related to these addresses. It is replicated in each node and the consensus building mechanism ensures that agreement on what operations have been finalized (and in what order) will be reached over the whole network. The ledger is the state of the Massa network, and fundamentally operations (see below) are requests to modify the ledger.

The information stored in the ledger with each address is the following:

Ledger information associated with each address	
`balance`	The amount of Massa coins owned by the address
`bytecode`	When the address references a smart contract, this is the compiled code corresponding to the smart contract (typically contains several functions that act as API entry points for the smart contract)
`datastore`	A key/value map that can store any persistent data related to a smart contract, its variables, etc

Smart Contract

Smart contracts are a piece of code that can be run inside the Massa virtual machine and which can modify the ledger, accept incoming requests through a public interface (via smart contract operations). One particularity of Massa smart contracts compared to other blockchain smart contracts is their ability to wake up by themselves independently of an exterior request on their interface. This allows more autonomy and less dependency on external centralized services.

Smart contracts are currently written in assemblyscript, a stricter derivation from typescript, which is itself a type-safe version of javascript. AssemblyScript compiles to web assembly bytecode (wasm). Massa nodes Execution Module runs such bytecode. Smart contracts have access to their own datastore, so they can modify the ledger.

Operation

Fundamentally, the point of the Massa network is to gather, order and execute operations, recorded inside blocks that are located in slots. There are three types of operations: transactions, roll operations, and smart contract code execution. The general structure of an operation is the following, and the different types of operations differ by their payload:

Operation header	
`creator_public_key`	The public key of the operation creator (32 bytes)
`expiration_period`	Period after which the operation is expired (u64 varint)
`fee`	The amount of fees the creator is willing to pay (u64 varint)
`type`	The type of operation (from 0 to 4: transaction, rollbuy, rollsell, executesc, callsc) (u64 varint)
`payload`	The content of the operation (see below)
`signature`	signature of all the above with the private key of the operation creator (64 bytes)

Transactions operations

Transactions are operations that move native Massa coins between addresses. Here is the corresponding payload:

Transaction payload	
`amount`	The amount of coins to transfer (u64 varint)
`destination_address`	The address of the recipient (32 bytes)

Buy/Sell Rolls operations

Rolls are staking tokens that participants can buy or sell with native coins (more about staking below). This is done via special operations, with a simple payload:

Roll buy/sell payload	
`nb_of_rolls`	The number of rolls to buy or to sell (u64 varint)

Smart Contract operations

Smart Contracts are pieces of code that can be run inside the Massa virtual machine. There are two ways of calling for the execution of code:

1. Direct execution of bytecode

In this case, the code is provided in the operation payload and executed directly:

Execute SC payload	
`max_gas`	The maximum gas spendable for this operation (u64 varint)
`bytecode_len`	The length of the bytecode field (u64 varint)
`bytecode`	The bytecode to run (in the context of the caller address)
`datastore_len`	The number of the datastore keys (u64 varint), each record is stored then one after the other after
list of datastore records	Concatenation of `key_len` (u8), `key`, `value_len` (u64 varint), `value`

1. Smart Contract function call

Here, the code is indirectly called via the call to an existing smart contract function, together with the required parameters:

Call SC	
`max_gas`	The maximum gas spendable for this operation (u64 varint)
`coins`	The coins transferred in the call (u64 varint)
`target_address`	The address of the targeted smart contract (32 bytes)
`function_name_length`	The length of the name of the function that is called (u8)
`function_name`	The name of the function that is called (utf8)
`param_len`	The number of parameters of the function call (u64 varint)
`params`	The parameters of the function call

Block

A block is a data structure built by nodes and its function it to aggregate several operations. As explained above, for each new slot that becomes active, a particular node in the network is elected in a deterministic way with the task of creating the block that will be stored in that slot (more about this in the description of the Selector Module below). A block from a given thread can only contain operations originating from a *creator_public_key* whose hash's five first bits designate the corresponding thread, thus implicitly avoiding collisions in operations integrated into parallel threads.

The content of a block is as follows:

Block header	
`slot`	A description of the block slot, defined by a couple (period, thread) that uniquely identify it
`creator_public_key`	The public key of the block creator (32 bytes)
`parents`	A list of the 32 parents of the block, one parent per thread (parent blocks are identified by the block hash)
`endorsements`	A list of the 16 endorsements for the block (more about endorsements below)
`operations_hash`	A hash of all the operations included in the block (=hash of the block body below)
`signature`	signature of all the above with the private key of the block creator
Block body	
`operations`	The list of all operations included in the block

Endorsements are optional inclusion in the block, but their inclusion is incentivized for block creators. They are validations of the fact that the parent block on the thread of the block is the best parent that could have been chosen, done by other nodes that have also been deterministically selected via the proof of stake probability distribution (see below). A comprehensive description of endorsements can be found here, so we will not go further into details in the context of this introduction.

7.1.2 Architecture

This is the diagram of the architecture of the software modules involved in building, endorsing and propagating blocks. The bottom part corresponds to a single process running in a node and is in charge of the execution and consensus building. The pool and factories, referred to as "factory", can be potentially running in a different process or be part of the node. Overall, each of the modules described here runs inside one or more threads attached to their respective executable process (NB: the factory/node separation is not yet implemented, but will be soon).

We will explain below the different modules present in this diagram, and simulate the production of an operation to show how it navigates through the different modules to better understand how blocks are produced and propagated.

Bootstrap Module

The bootstrap module is responsible for the initial synchronization of the node with the rest of the network. It is responsible for downloading the list of peers, the current graph of blocks, the ledger, the asyncronous pool, state of the Proof-of-Stake and latests executed operations.

The bootstrap will be done from a server that is listed on the configuration of the node. Bootstrap is the entry point of the network so you have to be careful on which node you connect to avoid downloading malicious data.

API Module

The API Module is the public window of the node to the rest of the world. It allows for interactions with external clients or factories via a JSON RPC protocol.

The API includes interfaces to do the following:

- publish a new operation from a client
- query the network about balances or ledger status
- allow for synchronization between remote pool/factory nodes and the consensus nodes, by sending/asking for blocks, best parents, draws, etc.

Protocol/Network Module

The Protocol/Network Module implements the protocol connecting consensus nodes. This protocol is supported by a binary and optimized transport layer and does not use JSON RPC.

The Protocol/Network Module will relay all operations/blocks creation and propagation, so that all other nodes in the network can synchronize their internal state, following a type of gossip synchronization protocol.

The type of messages that can be relayed via the Protocol/Network Module include:

- blocks/operations/endorsements propagation (either getting in or out of the node)
- nodes ban requests
- connectivity infos/stats

Selector Module, Proof of Stake sybil resistance

Every 0.5s, a new slot becomes active to receive a new block. A determinist selection mechanism ensures that one of the nodes in the network is elected to have the responsibility to build the block for that slot. This mechanism must have several key properties:

- it should be sybil resistant, so that it is not possible to increase one's odds of being elected by creating multiple clones of oneself (sybil) without a cost that is equal or greater than the cost of increasing one's odds for oneself only
- it should be deterministic, so that all nodes in the network will agree on the result of the selection at any given time
- it should be fair, so that each participant has a well-defined probability of being selected somehow proportional to the cost of participating, and draws converge towards this probability distribution over time

The way sybil resistance is achieved here is via the proof of stake mechanism. Nodes who want to participate in the block creation lottery will have to stake "rolls" that they buy with Massa coins. If they try to cheat by creating fake blocks or multiple blocks on the same slot, their stake will be taken away from them (slashing) and they would suffer the loss. The probabilistic "surface" of a participant is equal to its total stake, which makes the creation of sybil accounts useless because the stake would have to be split between them anyway.

The method used to draw an elected node for a given slot is simply a random draw from a distribution where addresses are weighted by the amount of stake (=rolls) they hold. The schema below illustrates how the seed and probability distribution are built, based on past cycles (two cycles are needed for the distribution update to ensure that the balance finalization has occurred and the amount of rolls is accurate):

The Selector Module is in charge of computing the formula and replying to requests regarding what node is elected for any given slot in the present or the past. The Execution Module (see below) is in charge of feeding the Selector Module with updates regarding balances, needed to compute the draws.

Graph/Consensus Module

The Consensus Module is the heart of the machinery of the Massa Network. It is in charge of integrating proposed blocks into their respective slots and verifying the integrity of the result. We have not yet talked about the various constraints regarding block creation, and in particular how parents are to be selected. In traditional blockchains, the parent of a block is simply the previous valid block in the chain. In the context of the Massa network and the parallel chains in the 32 threads, identifying the proper parent in a given thread requires a more sophisticated strategy involving the notion of block cliques.

Block cliques

At any given time, the set of all the blocks that have been produced and propagated in the network constitutes a graph (more precisely a Directed Acyclic Graph or "DAG"), where each block, except the genesis blocks, has 32 parents. All the reasoning below can be in principle done on this increasingly vast set, but in practice, we will introduce a notion of "finalized" or "staled" blocks, that can be removed from the set and that will allow us to work on a smaller subset of recent blocks that are neither finalized nor staled, so "pending" blocks. This set of pending blocks is all the network needs to know in order to incrementally build up a consensus, therefore non-pending blocks will simply be forgotten (this is a striking difference with most other blockchains that store in each node the history of all past transactions). The main benefit of this block pruning is to allow for some of the algorithms below, which are in general NP-complete, to run fast enough on a smaller subgraph, and to allow for a practical implementation.

Here is a simplified example of a graph of pending blocks over two threads, with blocks 3 and 4 competing for slot C1 (for example as a result of a multistaking attack where the block producer decided to create competing blocks for the same slot). Here the letter of a slot identifies it, while the number refers to its thread number:

In this illustration we have shown only relevant parent links in blue, to make the whole diagram more readable, but in reality, each block has 32 parents, one in each of the 32 threads.

An important notion we will use in the following is that of incompatibility between blocks. Excluding some edge cases with genesis blocks, there are two sources of incompatibilities defined for blocks:

1. **thread incompatibility**: this occurs when two blocks in a given thread have the same parent in that thread.
2. **grandpa incompatibility**: this corresponds to a case with two blocks B1 and B2 in threads t1 and t2, and where the block B1 in t1 has a parent in t2 who is an ancestor of B2's parent in t2, and symmetrically B2's parent in t1 is an ancestor of B1's parent in t1.

You will find a more formal mathematical definition of these incompatibility notions in the whitepaper.

From these definitions, you can build another graph, called the incompatibility graph, which connects any two blocks that have any form of incompatibility together:

As you can see, some blocks are isolated and therefore compatible with any other, while some are linked, because they have a form of incompatibility.

This brings us to the notion of a maximal clique which is a subset of the incompatibility graph such as none of the block members are incompatible with each other (so, no internal link withing the clique), and it is impossible to add an extra block to the set without introducing incompatibilities. In the above example, there are three maximal cliques that can be built, as illustrated below:

They represent candidates to extend the set of already finalized blocks into a coherent set of new blocks. All we need to add to be able to build a consensus rule now is to introduce a deterministic metric to rank those candidates so that nodes can independently and consistently decide on which clique is the best candidate and keep building on top of it. In particular, once the best maximal clique is identified, it becomes trivial to define the list of the parents for a new block simply by picking the oldest block from that clique in each thread.

The metric used in a traditional blockchain to rank competing chain candidates is habitually the length of the chain, or more precisely the total amount of work invested in the chain (also known as "Nakamoto consensus"). In the case of block cliques, we will introduce a notion of fitness for each block, and the fitness of the clique will simply be the sum of all its block's fitness. The block fitness $f(b)$ is simply defined as $1 + e$, e being the number of endorsements registered in the block.

Taking the maximal clique with the highest fitness (or some hash-based deterministic selection in case of equality), the Graph/Consensus module can define what is called the **blockclique** at the current time.

Finalized blocks, stale blocks

The set of pending blocks is growing each time a new block is produced and added to the current set. As we mentioned previously, there is also a pruning mechanism in charge of reducing the size of the graph by removing blocks that are considered final, and also blocks that can be considered stale and will never finalize.

If a block is only contained inside cliques that have a fitness lower than the fitness of the blockclique (the clique with the maximal fitness), minus a constant Δ_f^0, then this block is considered stale. Also, any new block that includes in its parents a stale block is stale.

A block is considered final if it is part of all maximal cliques, and included in at least one clique where the total sum of the fitness of all its descendants is greater than Δ_f^0.

Δ_f^0 is defined as a constant F multiplied by $1 + E$ (E being the total max number of endorsements in a block, currently 16), and F effectively measuring the maximum span in fully endorsed blocks of a successful blockclique, or the number of fully endorsed blocks by which an alternative clique can be shorter than the blockclique before its blocks may be discarded as stale.

Graph/Consensus Module Function

The Consensus Module (formerly known as the Graph) receives new block proposals, integrates them into the set of pending blocks, updating the blockclique with the method explained above, and verifying the legitimacy of the parenting of new blocks. It also informs other modules, like the Execution module, when blocks are finalized and the corresponding ledger modifications implied by their operations list should be made permanent.

It is also able to answer queries about the current best parents for a new block (based on the current blockclique) or the list of current maximal cliques.

Execution Module

The Execution Module is in charge of effectively executing the operations contained in blocks within the current blockclique, which is provided by the Graph/Consensus Module. Operations will typically modify the ledger, either by changing the balances of accounts or by modifying the datastore of smart contracts after the execution of some code. From an implementation point of view, ledger modifications are however stored as diff vs the current finalized ledger, until the corresponding blocks are marked as finalized by the Graph/Consensus Module.

Block creators will typically need to query the Execution Module to check current balances at a given slot and verify if some operations can be run with sufficient funds or not, before being integrated into a new block.

As a side note, it is also possible that blocks might include invalid operations, in which case the Execution Module will simply ignore them.

Being the maintainer of the ledger, the Execution Module is also queried about address information in general, via the API, for any Module that needs it.

Finally, the Execution Module will inform the Selector Module when new cycles are initiated as the finalization of blocks progresses.

Pool Module

When new pending operations reach a node, they are not immediately processed but instead are stored in a pool of pending operations, to be used by the Factory Module. Similarly, proposed endorsements coming from the Endorsement Factory are buffered inside the pool, to be integrated into new blocks by the Block Factory Module.

The origin of pending operations or endorsements inside the pool can be internal to the factory process or could come from remote nodes via the API Module. Similarly, locally produced pending endorsements are broadcasted via a gossip protocol to other pools via the API Module.

Note that operations stored in the Pool are naturally discarded after a certain time, since operations come with an expiration date in the *expiration_period* field. Still, some potential attacks can occur by trying to flood the pool with high fees operations that have no chance of being executed because the corresponding account does not have the required funds. Discussing about countermeasure for this is beyond the scope of this introduction.

Block/Endorsement Factory Module

The Block Factory Module is in charge of creating new blocks when the corresponding node address has been designated to be the block creator for a given slot. This information is provided to the Factory Module from the Selector Module via the API Module.

The Block Factory Module also needs information about the best parents (made of the latest blocks in each thread in the blockclique) from the Graph/Consensus Module. These parents will be included in the newly created block. Balance information, in order to assess the validity of pending operations, is obtained from the Execution Module, which maintains the ledger state from the point of view of the slot where the new block is supposed to be created.

The Block Factory Module picks pending operations from the Pool Module. Note that the Block Factory will regularly query the Execution Module about finalized and executed operations, and internally cleanup operations that have been handled.

Finally, the Block Factory will query the Pool Module and pick pending endorsements corresponding to the best parents that are selected for the block.

With this information, it is able to forge a new block that will then be propagated to the Graph/Consensus Module via the API Module, as well as to other nodes via gossip, to maintain a global synchronized state.

The Endorsement Factory Module works in a similar manner, requesting the Selector Module to find out when it has been designated to be an endorsement producer, then feeding new endorsements to the Pool Module and the API Module for global synchronization.

7.1.3 Operation lifecycle

We have now all the elements and vocabulary in place to explore the lifecycle of an operation within the network, from creation to permanent execution in a finalized block.

Operations originate externally from a client that is forging the operation, for example: a transaction or a smart contract code execution. The client will have to know the IP address of a Massa Node (this can be either because it is a node itself and will simply use localhost, or via some maintained list of known nodes and/or some browser plugin), and will then send the operation to the API Module.

When an operation is made available in a given node, it will be broadcasted to all other nodes via the Protocol/Network Module and to factories via the API Module, so that it will eventually end up in all the Pool Modules of the network.

Let's assume we just got a code execution operation from an external client. Let's suppose the client knows a particular node, which is running its block factory on the same machine, and sends the operation to this node. These are the different steps of the operation processing that will occur, as illustrated in the schema below:

1. The operation enters the node via the API Module (the operation path is marked in blue)
2. The API Module forwards the operation to the Pool Module and broadcasts it to other nodes via the Protocol/Network Module. Other nodes hearing about it will also broadcast it (gossip protocol), and feed it to their Pool Module, unless they are pure consensus nodes without factories
3. At that stage, the operation sits in the Pool Modules of most nodes
4. The Selector Module elects a particular node to handle the block production of the next current slot
5. The elected node Block Factory finds out about its election by querying a Selector Module (via the API Module)
6. It starts building a block by picking up pending operations in the Pool Module. The original operation is eventually picked and integrated into the block. We will now follow the block around (the block path is marked in green)
7. The newly produced block is sent via the API to remote or local nodes, to reach the Graph/Consensus Module
8. The new block is processed by the Graph/Consensus Module to be included into the pending blocks DAG and potentially integrated into a new blockclique
9. The Graph/Consensus Module sends the new block to other nodes via the Protocol/Network Module, to ensure synchronization of the information in the network. The new block reaching other nodes is similarly going to be integrated into their Graph/Consensus Module
10. In general, the blockclique will be extended with the new block and so will reach the Execution Module from the Graph/Consensus Module via the notification of a new blockclique. Eventually, it will also be notified as a final block if it gets finalized
11. The Execution Module will run the blocks that are part of the updated blockclique, so the original block will eventually be executed. Within the block is the original operation that was originally sent and that will then be applied to the ledger for potential modifications. At this stage, the modifications are not permanent and simply stored in a diff compared to the finalized ledger
12. Eventually, the block will be marked as final and the ledger modification, including the operation changes, will become final in the finalized ledger.

7.1.4 Conclusion

There are many more details and specific mechanisms that are not described in this short introduction, but it gives a good overview of the architecture and should help to get inside the code of the Massa Node.

Topics that were not handled here include:

- operations fees (each operation provides a fee and block factories will tend to favor operations with the highest fees for inclusion in blocks first), and how they are shared between block producers and endorsers
- ledger size limitation and the cost of storage
- slashing and node banning
- execution stack within smart contracts and what permissions smart contracts have in terms of ledger read/write access, based on their address
- details about the opcodes of the Massa WASM virtual machine
- analysis of potential attacks, like multistaking (when a block producer produces several blocks in the same slot)

For further references and technical details, you can find more information in the whitepaper.

This sections gives more details about several concepts used in the Massa blockchain.

7.1.5 Endorsements

Intro

Massa uses the Proof-of-Stake selection mechanism with Nakamoto consensus. In that context, when there are multiple cliques in close competition, we want all nodes to converge towards a single clique as fast as possible to minimize finality time and maximize the quality of the consensus. To achieve this, we draw inspiration from Tezos and introduce the concept of Endorsement.

Basic principle

Each block header has *E* ordered endorsement slots: each one can include an endorsement or not. Each endorsement contains:

- The slot *S* in which it is meant to be included. The endorsement can only be included in blocks of slot *S*.
- The hash of the endorsed block. This is the hash of the latest blockclique block of thread *S.thread* according to the endorsement creator at the moment the endorsement was created.
- The index of the endorsement slot within the header from *0* (included) to *E-1* (included). The endorsement can only be included at that endorsement slot index within the block eader.
- The public key of the creator of the endorsement
- The signature of all the previous fields with the private key of the creator of the endorsement

At every slot *S*, we use the existing Proof-of-Stake selection mechanism to not only draw the block creator for that slot, but also *E* other stakers indexed from *0* (included) to *E-1* (included) that can create Endorsements meant to be included in block headers of slot *S* at their respective endorsement slot index.

Conceptually, each endorsement meant to be included at a slot *S* can be seen as a single vote endorsing the parent in thread *S.thread* that the endorsement creator would have chosen if they had to create a block at slot *S*.

The likelihood of the attacker getting lucky and being selected for *N* consecutive PoS draws to attack/censor the system decays exponentially with *N*. With endorsements, we don't have to wait for *N* blocks to account for *N* proof-of-stake draws to happen as *E+1* draws happen at every slot (1 for the block creator and *E* for endorsement creators). In the

consensus algorithm, we choose the clique of highest fitness as the blockclique. A block including *e* endorsements out of the maximum *E* contributes a fitness *e + 1* to the cliques it belongs to. The fitness of a block is therefore reflected by the number of PoS draws that were involved in creating it.

The net effect of this mechanism is to increase safety and convergence speed by allowing block producers to quickly choose the best clique to extend according to the "votes" provided by the endorsements.

Structure of an endorsement

```
pub struct Endorsement {
  /// slot in which the endorsement can be included
  pub slot: Slot,
  /// endorsement index inside the including block
  pub index: u32,
  /// Hash of endorsed block.
  /// This is the parent in thread `self.slot.thread` of the block in which the␣
↪endorsement is included
  pub endorsed_block: BlockId,
}
```

Note that the *WrappedEndorsement* structure includes the underlying *Endorsement* as well as the signature, and the public key of the endorsement producer.

Within a block, endorsements are fully included inside the header.

A header is invalidated if:

- it contains strictly more than *E* endorsements
- at least one of the endorsements fails deserialization or signature verification
- at least one of the endorsements endorses a block different than the parent of the including block within its own thread
- any of the endorsements should not have been produced at that *(endorsement.slot, endorsement.index)* according to the selector
- there is strictly more than one endorsement with a given *endorsement.index*

Lifecycle of an endorsement

To produce endorsements for slot *S*, the Endorsement Factory wakes up at *timestamp(S) - t0/2* so that the previous block of thread *S.thread* (the endorsed block) had the time to propagate, and so that the endorsement itself has the time to propagate to be included in blocks of slot *S*. It then checks the endorsement producer draws for slot *S*. At every slot, there are *E* endorsement producer draws, one for each endorsement index from *0* (included) to *E-1* (included). The factory will attempt to create all the endorsements that need to be produced by keypairs its wallet manages. To choose the block to endorse, the factory asks Consensus for the ID of latest blockclique (or final) block *B* in thread *S.thread* that has a strictly lower period than *S.period*. Every created endorsement is then sent to the Endorsement Pool for future inclusion in blocks, and to Protocol for propagation to other nodes.

In Protocol, endorsements can be received from other modules, in which case they are propagated. They can also be received from other nodes, in which case they added to the Endorsement Pool and propagated. Endorsements are propagated only to nodes that don't already know about them (including inside block headers).

The Endorsement Pool stores a finite number of endorsements that can potentially be included in future blocks created by the node. Consensus notifies the Endorsement pool of newly finalized blocks, which allows the pool to eliminate endorsements that can only be included in already-finalized slots and are therefore not useful anymore.

When the Block Factory produces a block and needs to fill its header with endorsements, it asks the Endorsement Pool for the endorsements that can be included in the block's slot and that endorse the block's parent in its own thread.

Incentives and penalties

There needs to be an incentive in:

- creating blocks that can be endorsed, and also avoid publishing them too late so that endorsers have the time to endorse them
- creating and propagating endorsements, also doing so not too early in order to endorse the most recent block, and not too late for subsequent blocks to be able to include the endorsement
- including endorsements in blocks being created, and also not publishing them too early to include as many endorsements as possible

To achieve this, we note R the total amount of coin revenue generated by the block: the sum of the per-block monetary creation, and all operation fees. We then split R into $1+E$ equal parts called $r = R/(1+E)$.

- r is given to the block creator to motivate block creation even if there are no endorsements available
- for each successfully included endorsement:
  - $r/3$ is given to the block creator to motivate endorsement inclusion
  - $r/3$ is given to the endorsement creator to motivate endorsement creation
  - $r/3$ is given to the creator of the endorsed block to motivate the timely emission of endorsable blocks

Note that this split also massively increases the frequency at which stakers receive coins, which reduces the incentive to create staking pools.

Choosing the value of *E*

The value of E has implication both in the safety and in the performance of the protocol. In terms of performance, the greater the value of E is, the more ressources (bandwith, memory, computing power) is needed to generate, broadcast and include endorsements, which could induce latency. One the other hand, the value of E has implications in the safety of the protocol. The threat we are looking at here is the finality fork attack.

Finality fork attacks

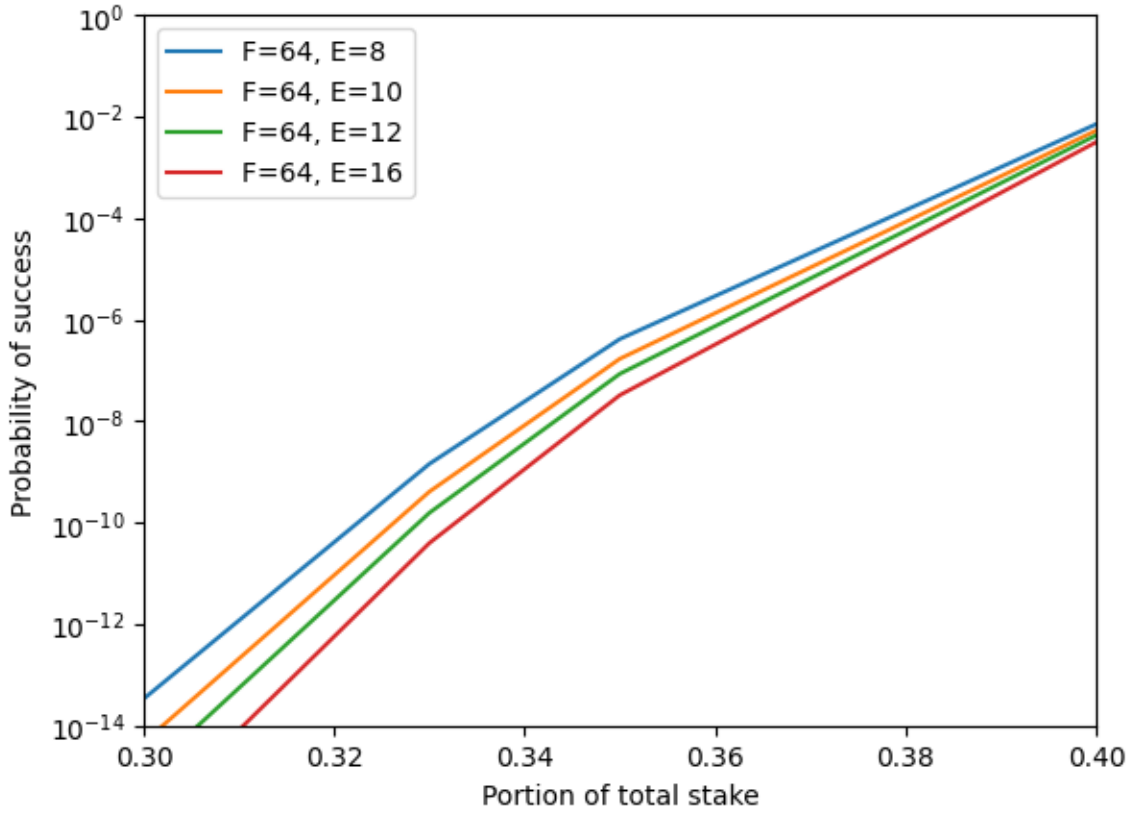
A finality fork attack is when an attacker that controls a portion *beta* of the total stake, decides not to broadcast the blocks he has been selected to produce, in order to construct an alternative and undisclosed clique. The attacker's goal is to produce an attack clique that has a fitness greater than the honnest clique. If he manages to do so, the attacker can wait until the finalization time of a block b belonging to the honest clique, to broadcast its attack clique. Because the fitness of the attack clique is greater than the honest one, a portion of the network will consider the attacker's clique to be the good version of the chain, while some other portion will have finalized block b. This results in a network partion, where two or more portion of the network do not agree on the state of the ledger.

Security level

The factors that influence the success probability of a finality fork attack are the number of endorsments per block *E*, the portion of the total stake controlled by the attacker *beta* and a finality parameter *F*. The finality parameter *F* respresents a number of descendant a block must have, before being finalized. The portion of the stake *beta* is positively correlated with the success probability of an attack, while *F* and *E* are inversly correlated. We do not have control over *beta*, thus we define our security level as a portion *beta* and a maximum success probability of an attack. We chose *beta = 1/3* because it is the maximum proportion of Byzantine participants BFT based consensus protocols assume. We chose *p_success <= 1e-11* because it corresponds to a period of time of around 10 000 years (given one block every 0.5s). We must chose the minimal parameters *F* and *E* that match this security level.

Simulation results

Our simulations results show that with *F = 64*, *E = 16* and given 32 thread and a portion *beta = 1/3* of the stake controlled by the attacker, the probability of success of an attack is in the order of *1e-11*. You can find more details in the [technical paper](https://arxiv.org/abs/1803.09029).



Future features

There is still optimizations that we can perform on the endorsements and their workflow :

- Add more verifications when receiving an endorsement from an other node of the network to avoid storing/propagating invalid endorsements. Issue
- Use stored endorsements to choose the best parents. Issue
- Split blocks and endorsements productions to an other binary so that they can be run on different machines and be more customized. Discussion
- To punish network and CPU overload attacks, a denunciations system will be implemented to point out the stakers that produces two different endorsements for the same *(slot, index)*. Issue

7.2 Massa's Decentralized web

7.2.1 Rationale

The "*code is law*" rule is a cornerstone of DeFi.

It states among other things that once successfully audited, a program can remain trusted. This implies that the program of a successfully audited smart contract may never be unexpectedly changed by an outsider (note that contracts that modify their own code during runtime cannot be trusted this way in the first place, but this is visible in the code audit). Popular ETH smart contracts essentially follow that rule.

However, most DeFi web3 apps such as app.uniswap.org are typically used through an otherwise normal website that talks to a browser plugin (typically Metamask) allowing the webpage to interact with the user's wallet and the blockchain. The website that serves as an entry point to the dApp is neither decentralized nor immutable-once-audited. This breaks the very foundation of blockchain security. And that's how you get into situations of scandalous thefts in the DeFi world, like this one.

The goal here is to allow addresses on Massa to store not only a balance, bytecode and a datastore, but also files, zip archive and websites without using any centralised party in between your client and the blockchain. Any address, through bytecode execution, can initialize, read and write the "file" part into the datastore. Files or websites, for instance, are complex objects built on top of native objects (eg: balance, bytecode…) to enable users to do more while assuring full decentralisation. The reason why we don't simply reuse the datastore for this, outside of the risk of key collisions, is for easier auditing: if the code never writes into its own bytecode nor its datastore after deployment, it is safe to assume that the stored website can't change anymore. Which means that Massa can host the files corresponding to the front-end of the decentralized applications. Since the front-end is hosted on the blockchain, allowing anyone to access it using a Massa node. For that Massa has developed a client that acts as a gateway to the blockchain preventing you from using any centralised servers but also maximising your security with immutable and censorship resistant websites.

This client is called (thyra) which in ancient Greek means door, entrance.

Start a decentralised web3 journey now, and install Thyra. It will enable you to store your own website or to simply navigate Massa web3 content.

That way, Massa allows deploying fully decentralized code-is-law apps, as it was meant to be!

7.3 Massa's Autonomous Smart Contracts

7.3.1 Introduction

Issues with current smart contracts

Automating IT processes are at the heart of pretty much every industry we can think of, and if we narrow it down to modern finance, nowadays most actions are initiated by automated mechanisms. When we take a closer look at decentralized finance, only certain actions of lending and arbitration are done automatically, however they are executed by bots operating off-chain. This is because without external calls, smart contracts as they exist in all public blockchains cannot perform automated operations.

Many decentralized protocols rely on recurrent triggers of certain functions in order to work as planned. In case of decentralized lending protocols, borrowers lock crypto assets (cryptocurrencies or fungible tokens) in order to take out loans. When the price of the collateralized asset decreases below a threshold, the borrower's position becomes under-collateralized if he doesn't promptly react to the decrease in value of that collateral. To ensure that the protocol behaves correctly, such positions must be liquidated. Liquidations are currently performed by organizations or individuals running bots, usually on some centralized cloud services.

The need for a reliable automation mechanism

There are countless applications that rely on such recurrent triggers. As a result, a lot of time and energy has been spent trying to develop more reliable networks of bots to guarantee that transactions are executed when needed. However, as those solutions are inherently off-chain, there's no guarantee that the execution will be effectively triggered. And when those bots fail to execute such transactions, those protocols are at risk (and the applications built on top of them).

7.3.2 Autonomous Smart Contracts

Massa's Autonomous Smart Contracts solve the issues of lacking reliability, sophistication and centralization around dApps that want to offer automated smart contract executions on behalf of their users. Autonomous Smart Contracts introduce self wake-up capabilities to smart contracts. In the future, smart contracts could be programmed to perform arbitrary operations, for example triggering a call when a specific exchange rate target of an LP pool is met.

Such automated capabilities open the door to various applications, from automated liquidation of under-collateralized positions on lending protocols, to on-chain trading bots, or ever evolving NFTs.

7.4 Installing a node

Note: Right now 4 cores and 8 GB of RAM should be enough to run a node, but it might increase in the future. More info in the *FAQ*.

7.4.1 From binaries

If you just wish to run a Massa node without compiling it yourself, you can simply download the latest binary below and go the the next step: *Running a node*.

- Windows executable
- Linux binary - only works with libc2.28 at least (for example Ubuntu 20.04 and higher)
- MacOS binary

7.4.2 From source code

Otherwise, if you wish to run a Massa node from source code, here are the steps to follow:

On Ubuntu / MacOS

- on Ubuntu, these libs must be installed: `sudo apt install pkg-config curl git build-essential libssl-dev libclang-dev`
- on MacOS: `brew install llvm`
- install rustup: `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
- configure path: `source $HOME/.cargo/env`
- check rust version: `rustc --version`
- install nigthly: `rustup toolchain install nightly-2022-11-14`
- set it as default: `rustup default nightly-2022-11-14`
- check rust version: `rustc --version`
- clone this repo: `git clone --branch testnet https://github.com/massalabs/massa.git`

On Windows

Set up your Rust environment

- On Windows, you should first follow the indications from Microsoft to be able to run on a Rust environment here.
  - Install Visual Studio (recommended) or the Microsoft C++ Build Tools.
  - Once Visual Studio is installed, click on C++ Build Tool. Select on the right column called "installation details" the following packages:
    - MSCV v142 – VS 2019
    - Windows 10 SDK
    - C++ CMake tools for Windows
    - Testing Tools Core Feature
  - Click install on the bottom right to download and install those packages
- Install Chocolatey and run: `choco install llvm`
- Install Rust, to be downloaded here
- Install Git for windows, to be downloaded here

Clone the Massa Git Repository

- Open Windows Power Shell
  - Clone the latest distributed version: `git clone --branch testnet https://github.com/massalabs/massa.git`
  - Change default Rust to nightly: `rustup default nightly-2022-11-14`

7.5 Running a node

7.5.1 From binaries

Simply run the binaries you downloaded in the previous step: Open the *massa-node* folder and run the *massa-node* executable Open the *massa-client* folder and run the *massa-client* executable

On Ubuntu / MacOS

Configure the node

Defaut configuration is available here.

You can override the defaut configuration via *massa-node/config/config.toml* file.

Start the node

On a first window:

```
cd massa/massa-node/
```

Launch the node, on Ubuntu:

```
./massa-node -p <PASSWORD> |& tee logs.txt
```

Replace <PASSWORD> with a password that you will need to keep to restart your node You should leave the window opened.

Start the client

On a second window:

```
cd massa/massa-client/
```

Then:

```
./massa-client -p <PASSWORD>
```

Replace <PASSWORD> with a password that you will need to keep to restart your client

7.5.2 From source code

On Ubuntu / MacOS

Start the node

On a first window:

```
cd massa/massa-node/
```

Launch the node, on Ubuntu:

```
RUST_BACKTRACE=full cargo run --release -- -p <PASSWORD> |& tee logs.txt
```

Replace <PASSWORD> with a password that you will need to keep to restart your node

Or, on macOS:

```
RUST_BACKTRACE=full cargo run --release -- -p <PASSWORD> > logs.txt 2>&1
```

Replace <PASSWORD> with a password that you will need to keep to restart your node You should leave the window opened.

Start the client

On a second window:

```
cd massa/massa-client/
```

Then:

```
cargo run --release -- -p <PASSWORD>
```

Replace <PASSWORD> with a password that you will need to keep to restart your client Please wait until the directories are built before moving to the next step.

On Windows

Start the Node

- **Open Windows Power Shell or Command Prompt on a first window**
  - Type: `cd massa`
  - Type: `cd massa-node`
  - Type: `cargo run --release -- -p <PASSWORD>`

Replace <PASSWORD> with a password that you will need to keep to restart your node You should leave the window opened.

Start the Client

- **Open Windows Power Shell or Command Prompt on a second window**
  - Type: `cd massa`
  - Type: `cd massa-client`
  - Type: `cargo run --release -- -p <PASSWORD>`

Replace <PASSWORD> with a password that you will need to keep to restart your client Please wait until the directories are built before moving to the next step.

Warning: In case of crash of the rust compiler or at runtime, please do not report bugs to the rustlang/rust repository, but open an issue on massa instead. We will triage the issues and open them on the rust side if they are valid. This avoids polluting the main rust repository with many reports of the same error.

7.6 Update a node

If you use the binaries, simply download the latest binaries, and make sure you use the latest nightly version of rust.

Download the nightly version we use:

```
rustup install nightly-2022-11-14
```

Use the right version:

```
rustup default nightly-2022-11-14
```

Otherwise:

Make sure you you have the right git repository (especially since the change from GitLab to GitHub):

```
cd massa/
git stash
git remote set-url origin https://github.com/massalabs/massa.git
```

Update Massa:

```
git checkout testnet
git pull
```

After updating, enter the command `node_get_staking_addresses` in your client and make sure that it returns an address that has rolls according to `wallet_info`.

- If `wallet_info` does not return any address, it means that you haven't backed up your wallet.dat correctly. Close the client, overwrite wallet.dat with your backup, launch the client again and try again. You can also create a new address by calling `wallet_generate_secret_key`.
- If you can't find an address in `wallet_info` that has non-zero candidate, non-zero final and non-zero active rolls, please refer to the staking tutorial on getting rolls.
- If `node_get_staking_addresses` does not return any address or does not return an address that has active_rolls according to *wallet_info*, it means you haven't backed up staking_keys.json properly. Try stopping the node, overwriting staking_keys.json with your backup, and starting the node again to try again. You can also manually add a staking key by calling `add_staking_keys` with the KEY matching the address that has active rolls in `wallet_info` (warning: do not type the address or public key, only the SECRET KEY).

7.7 Creating a Massa wallet

A Massa wallet is a file that contains a list of your keypairs.

Like other blockchains, Massa uses elliptic curve cryptography for the security of your coins (with *secp256k1*).

It means your secret key is your password allowing you to spend coins that were sent to your address (your address is the hash of your public key).

Here is how to create a Massa wallet.

7.7.1 From the command line interface

If your client is not running

Go to the client folder:

```
cd massa/massa-client/
```

Start the interactive client and load a wallet file:

```
cargo run
```

It will ask your wallet password in order to load *wallet.dat*. If the file does not exist, you will be asked to set a password and it will be created.

If your client is running

Now you can either generate a new keypair (and associated address):

```
wallet_generate_secret_key
```

Or, if you already have one from a previous wallet, you can add manually an existing keypair:

```
wallet_add_secret_keys <SecretKey>
```

The list of addresses and keys of your wallet can be accessed with:

```
wallet_info
```

7.7.2 From the graphical interface

If you don't plan to stake or use the command-line client, you can also create a wallet on the web interface: head to the explorer, in the wallet tab.

Click *Generate secret key* then *Add*.

This generates a new random keypair from your computer randomness which stays on your side, it is never transferred on the network.

Now you can add more addresses or see the list of your addresses with their associated thread and balance.

Also, if you want to save this wallet and be able to restore it later, click *Save wallet*, and next time directly do *Load wallet*.

7.8 Staking

In Massa, the minimal number of coins required to be able to stake is 100 MAS (which is called one "roll"). The total initial supply is 500m MAS, so in theory, there could be 5 million people staking.

Addresses are randomly selected to stake (create blocks) in all threads, based on the number of rolls they possess. The list of stakers and their rolls can be seen there.

Rolls can be bought with Massa coins or sold to get the coins back. If you already have more than 100 Massa, you can continue this tutorial, otherwise, send your address to the faucet bot in the "testnet-faucet" channel of our Discord.

7.8.1 Buying rolls

Get the address that has coins in your wallet. In the Massa client:

```
wallet_info
```

Buy rolls with it: put your address, the number of rolls you want to buy, and the operation fee (you can put 0):

```
buy_rolls <address> <roll count> <fee>
```

As an example, the command for buying 1 roll with 0 fee for the address *VkUQ5MA4niNBhAEP7uVf89tvPfUHcbgy6BrdLM9SAuFSyy9DE* is: `buy_rolls VkUQ5MA4niNBhAEP7uVf89tvPfUHcbgy6BrdLM9SAuFSyy9DE 1 0`

It should take less than one minute for your roll to become final, check with:

```
wallet_info
```

7.8.2 Telling your node to start staking with your rolls

Get the secret key that has rolls in your wallet:

```
wallet_info
```

Register your secret key so that your node start to stake with it:

```
node_add_staking_secret_keys <your_secret_key>
```

Now you should wait some time so that your rolls become active: 3 cycles of 128 periods (one period is 32 blocks - 16 sec), so about 1h40 minutes.

You can check if your rolls are active with the same command:

```
wallet_info
```

When your rolls become active, that's it! You're staking! Please note, having one roll is enough. On the testnet, we don't value how many rolls you have, but how reliable is your node.

You should be selected to create blocks in the different threads.

To check when your address is selected to stake, run this command:

```
get_addresses <your_address>
```

and look at the "next draws" section.

Also check that your balance increases, for each block or endorsement that you create you should get a small reward.

7.8.3 Selling rolls

If you want to get back some or all of your coins, sell rolls the same way you bought them:

```
sell_rolls <address> <roll count> <fee>
```

It should take some time again for your coins to be credited, and they will be frozen for 1 cycle before you can spend them, again check with:

```
wallet_info
```

7.9 Routability

7.9.1 Principle

Nodes in the Massa network need to establish connections between them to communicate, propagate blocks and operations, and maintain consensus and synchrony all together.

For node A to establish a connection towards node B, node B must be routable. This means that node B has a public IP address that can be reached from node A and that ports TCP 31244 and TCP 31245 are open on node B and that inbound connection on those ports are allowed by firewalls on node B. Once such a connection is established, communication through this connection is bidirectional, and it does not matter anymore which one of the two nodes initiated the connection establishment.

If only a small number of nodes are routable, all other nodes will be able to connect only to those routable nodes, which can overload them and generally hurt the decentralization and security of the network, as those few routable nodes become de-facto central communication hubs, choke points, and single points of failure. It is therefore important to have as many routable nodes as possible.

In Massa, nodes are non-routable by default and require a manual operation to be made routable.

7.9.2 How to make your node routable

- make sure the computer on which the node is running has a static public IP address (IPv4 or IPv6). You can retrieve the public IP address of your computer by opening ipify
- if the computer running the node is behind a router/NAT, you will need to configure your router:
  - if the router uses DHCP, the MAC address of the computer running the node must be set to have a permanent DHCP lease (a local IP address that never changes, usually of form 192.168.X.XX)
  - incoming connections on TCP ports 31244 and 31245 must be directed towards the local IP address of the computer running the node
- setup the firewall on your computer to allow incoming TCP connections on ports 31244 and 31245 (example: `ufw allow 31244 && ufw allow 31245` on Ubuntu, or set up the Windows Firewall on Windows)
- edit file `massa-node/config/config.toml` (create it if absent) with the following contents:

```
[network]
routable_ip = "AAA.BBB.CCC.DDD"
```

where AAA.BBB.CCC.DDD should be replaced with your public IP address (not the local one !). IPV6 is also supported.

- run the massa node
- you can then test if your ports are open by typing your public IP address and port 31244 in yougetsignal (then again with port 31245)
- Once your node is routable, you need to send the public IP address of your node to the Discord bot. You first need to register to the staking reward program (see the last step below).

7.9.3 Last step

- To validate your participation in the testnet staking reward program, you have to register with your Discord account. Write something in the *testnet-rewards-registration* channel of our Discord and our bot will DM you instructions. More info here: Testnet rewards program

7.10 Testnet Staking Rewards Program

To help achieve our goal of a fully decentralized and scaled blockchain, we designed a staking rewards program during the testnet phase.

People that consistently run a node and produce blocks will be rewarded mainnet Massa tokens when mainnet launches.

Staking is what improves the security of the network. By buying rolls (freezing your coins) and producing your share of the blocks, you help honest nodes collectively protect against potential attackers, who must not reach 51% of the block production. On mainnet, staking is incentivized through block rewards: for each block produced, you get some Massa. On testnet however, you get testnet Massa which has no value, this is why we will reward you with mainnet Massa for learning to set up your node and stake right now, which also helps us improve the staking user experience.

On July 16th we launched the first public version of Massa, the first testnet. More than 350 nodes were connected at the same time after one week, which overloaded our bootstrap nodes which were the only nodes accepting connections. By setting your node up to be routable (with a public IP), you become a real peer in the peer-to-peer network: you not only connect to existing routable nodes, but you offer other people the possibility to access the network through your connection. We will therefore also reward how often your node is publicly accessible.

7.10.1 Episodes

We have release cycles of 1 month each, called “episodes”, the first one started in August 2021. At the beginning of an episode, participants have a few days to set up their nodes with the newest version before scoring starts, but it’s also possible to join anytime during the episode.

Throughout the episode, you can ask for coins in the Discord faucet (on channel *testnet-faucet*). No need to abuse the faucet, we don’t reward you based on the number of rolls.

At the end of an episode, all nodes stop by themselves and become useless/unusable. Participants have to download and launch the new version for the next episode. Make sure you keep the same node private key and wallet!

Scoring Formula

The score of a node for a given episode is the following:

> Score = (produced_blocks/selected_slots) * (active_cycles/nb_cycles_episode) * (50 + 40 * routable_samples/routability_trials + 30 * total_maxim_factor/routability_trials)

- 50 points of the score are based on staking:
  - (*produced_blocks* / *selected_slots*) * (*active_cycles* / *nb_cycles_episode*)
    - *active_cycles* is the number of cycles in the episode during which the address had active rolls.
    - *nb_cycles* is the total number of cycles in the episode.
    - *produced_blocks* is the number of final blocks produced by the staker during the episode.
    - *selected_slots* is the number of final slots for which the staker was selected to create blocks during the episode. If *selected_slots* = 0, the staking score is set to 0.
    - The maximum score is supposed to be reached if, during the whole episode, the node has rolls and produces all blocks when it is selected to.
- 40 points of the score are based on the routability of the node: how often the node can be reached by other nodes.
  - *routable_samples* / *routability_trials*
    - *routability_trials* is the number of connection trials that resulted in a successful connection.
    - Maximum score is achieved if the node can always be reached by other nodes.
- 30 points of the score incentivize node diversity: the network is more decentralized if nodes are spread across countries and providers than if they are all hosted at the same location/provider.
  - *total_maxim_factor* / *routability_trials*
    - *total_maxim_factor* is the total amount of *maxim_factor* accumulated at each cycle. The *maxim_factor* is a value between 0 and 1 representing the distance between this node's IP address and the IP addresses of other nodes in a given cycle.
    - Maximum score is reached when running the node at home or with a provider that is not used to run other Massa nodes.

We encourage every person to run only one node. Running multiple nodes with the same staking keys will result in roll slashing in the future. Running multiple nodes with the same node_privkey.key also reduces network health and will be a point of attention for rewards.

Registration

To validate your participation in the testnet staking reward program, you have to register with your Discord account. Write something in the *testnet-rewards-registration* channel of our Discord and our bot will DM you instructions.

From scores to rewards

The launch of mainnet is planned for 2022.

By this time, people will be able to claim their rewards through a KYC process (most likely including video/liveness) to ensure that the same people don't do multiple claims, and comply with KYC/AML laws.

The testnet score of a person will be the sum of all their episode scores.

The mainnet reward will depend on the testnet score. More info on mainnet rewards will come later.

7.11 Frequently Asked Questions

7.11.1 General questions

What are the hardware requirements?

The philosophy of Massa is to be as decentralized as possible. To fulfill this goal, we aim to have low hardware requirements so that many people can run nodes. Right now 4 cores and 8 GB of RAM should be enough to run a node. As the transaction rate increases, it might not be sufficient anymore. Ultimately, we plan that the mainnet fits on a desktop computer with 8 cores, 16 GB RAM, and 1TB disk.

Can it run on a VPS?

You can use a VPS to run a node. The pros of VPS are that they have high availability and are easy to configure. Cons are that nodes running on a VPS can lead to centralization if a lot of nodes running on the same provider (e.g. AWS).

How to keep the node running when I close the terminal?

You can run the following command in the terminal:

```
nohup cargo run --release &
```

the output will go to the *nohup.out* file. You will be able to close the terminal safely then. To kill the app you'll have to use `pkill -f massa-node`. You can also use screen or tmux for example.

Will Massa support smart contracts?

We will try to support both the EVM for retro compatibility, and a specific smart contract engine that fully leverages the Massa protocol and allows to develop in more usual languages as well as introduces several innovations.

Our smart contract system is released and run on the testnet. You can find the full documentation *here*.

We are planning some exciting features, such as self-wakeup, a bit like what is introduced here

What ports does Massa use?

By default, Massa uses TCP port 31244 for protocol communication with other nodes, and 31245 to bootstrap other nodes. Massa also uses TCP port 33034 for the new private API, and 33035 for the new public API (API v2).

How to restart the Node?

- **Ubuntu**
  [ctrl + c for killing the process and] `cargo run --release |& tee logs.txt`
- Windows : ctrl + c for killing the process and `cargo run --release`
- **Mac Os**
  [ctrl + c for killing the process and] `cargo run --release > logs.txt 2>&1`

How secure are the keypairs ?

Please note that the Testnet coins have NO VALUE. That being said, we are working on adding encryption on several levels before the Mainnet.

The staking key file in the node folder and the wallet file in the client folder are currently not encrypted but it will come soon. Also, private API communication between the client and the node is not encrypted for now but it will be implemented before the Mainnet as well.

Note that nodes don't know or trust each other, and they never exchange sensitive information, therefore cryptography is not required at that level. A handshake is performed at the connection with another peer. We sign random bytes that the peer sent us with our keypair, and same on the other side. And data that is sent after that is signed by its creator, not the node that is sending it to us. During the bootstrap, the handshake is asymmetric. We know the public key of the bootstrap node and we expect signed messages from it, but we do not communicate our public key, nor we sign the only message we send (just random bytes).

7.11.2 Balance and wallet

How to migrate from one server to another without losing staked amounts and tokens?

You need to back up the file wallet.dat and migrate it to the massa-client folder on your new server. You also need to backup and migrate the node_privkey.key file in massa-node/config to keep your connectivity stats.

If you have rolls, you also need to register the key used to buy rolls to start staking again *here*.

Why are the balances in the client and the explorer different ?

It may mean that your node is desynchronized. Check that your node is running, that the computer meets hardware requirements, and try restarting your node.

Does the command *cargo run – –wallet wallet.dat* override my existing wallet?

No, it loads the wallet if it exists, otherwise, it creates it.

Where is the wallet.dat located?

By default, in the massa-client directory.

7.11.3 Rolls and staking

My rolls disappeared/were sold automatically.

The most likely reason is that you did not produce some blocks when selected to do so. Most frequent reasons:

- Node not running 100% of the time during which you had active_rolls > 0
- Node not being properly connected to the network 100% of the time during which you had active_rolls > 0
- Node being desynchronized (which can be caused by temporary overload if the specs are insufficient or if other programs are using resources on the computer or because of internet connection problems) at some point while you had active_rolls > 0
- The node does not having the right registered staking keys (type staking_addresses in the client to verify that they match the addresses in your wallet_info that have active rolls) 100% of the time during which you had active_rolls > 0
- Some hosting providers have Half-duplex connection setting. Contact hosting support and ask to switch you to full-duplex.

Diagnostic process:

- make sure the node is running on a computer that matches hardware requirements and that no other software is hogging ressources
- type `wallet_info` and make sure that at least one address has active rolls > 0
  - if there are no addresses listed, create a new one by calling *wallet_generate_private_key* and try the diagnostic process again
  - if none of the listed addresses has non-zero active rolls, perform a new roll buy (see tutorials) and try the diagnostic process again
- type `node_get_staking_addresses` in the client:
  - if the list is empty or if none of the addresses listed matches addresses that have active rolls in `wallet_info`:
    - call `node_add_staking_secret_keys` with the secret key matching an address that has non-zero active rolls in `wallet_info`
- check your address with the online explorer: if there is a mismatch between the number of active rolls displayed in the online interface and what is returned by *wallet_info*, it might be that your node is desynchronized. Try restarting it.

Why are rolls automatically sold? Is it some kind of penalty/slashing?

It is not slashing because the funds are reimbursed fully. It's more like an implicit roll sell.

The point is the following: for the network to be healthy, everyone with active rolls needs to produce blocks whenever they are selected to do so. If an address misses more than 70% of its block creation opportunities during cycle C, all its rolls are implicitly sold at the beginning of cycle C+3.

Do I need to register the keys after subsequent purchases of ROLLs, or do they get staked automatically?

For now, they don't stake automatically. In the future, we will add a feature allowing auto compounding. That being said, some people appear to have done that very early in the project. Feel free to ask on the Discord server :).

I can buy, send, sell ROLLs and coins without fees. When should I increase the fee >0?

For the moment, there are only a few transactions at the same time and so most created blocks are empty. This means that your operation will be added to a block even if the fee is zero. We will communicate if you need to increase the fee.

I am staking ROLLs but my wallet info doesn't change. When do I get my first staking rewards?

You need to wait for your rolls to become active (around 1h45), then depending on the number of rolls you have, you might want to wait for more to be selected for block/endorsement production.

7.11.4 Testnet and rewards

How can I migrate my node from one computer/provider to another and keep my score in the Testnet Staking Reward Program?

If you migrate your node from one computer/provider to another you should save the keypair associated with the staking address that is registered. This keypair is located in the *wallet.dat* file located in *massa-client* folder. You can also save your node keypair *node_privkey.key* located in the *massa-node/config* folder, if you don't then don't forget to register your new node keypair to the Discord bot.

If your new node has a new IP address then you should not forget to register the new IP address to the Discord bot.

If you lost *wallet.dat* and/or *node_privkey.key*, don't panic, just redo the whole node setup and rewards registration process and the newly generated keys will be associated with your discord account. Past scores won't be lost.

I want to stake more! Can I abuse the faucet bot to get more coins?

You can claim testnet tokens every 24h. The tokens are worthless, you won't have any advantage over the others by doing that.

Will the amount of staked Rolls affect Testnet rewards?

No, as long as you have at least 1 roll, further roll purchases won't change your score.

I can't register with the Discord bot because the node ID is already used

If you changed your staking key, you need to register again with the bot using the `node_testnet_rewards_program_ownership_proof` command. If you are using the same install, the bot will return the following error message: "This node ID is already used or has already been used, please use another one!". To solve this, you need to generate a new node ID. Stop your node and delete the *node_privkey.key* file in *massa-node/config*. You can then start your node again and you will have a new node ID.

7.11.5 Common issues

Ping too high issue

Check the quality of your internet connection. Try increasing the "max_ping" setting in your config file:

- edit file *massa-node/config/config.toml* (create if it is absent) with the following content:

```
[bootstrap]
max_ping = 10000 # try 10000 for example
```

API can't start

- If your API can't start, e.g. with *could not start API controller: ServerError(hyper::Error(Listen, Os { code: 98, kind: AddrInUse, message: "Address already in use" }))*, it's probably because the default API ports 33034/33035 are already in use on your computer. You should change the port in the config files, both in the API and Client:
- create/edit file *massa-node/config/config.toml* to change the port used by the API:

```
[api]
bind_private = "127.0.0.1:33034" # change port here from 33034 to something else
bind_public = "0.0.0.0:33035" # change port here from 33035 to something else
```

- create/edit file *massa-client/config/config.toml* and put the same port:

```
[default_node]
ip = "127.0.0.1"
private_port = 33034 # change port here from 33034 to the port chosen in node's bind_
↪private
public_port = 33035 # change port here from 33035 to the port chosen in node's bind_
↪public
```

Raspberry Pi problem "Thread 'main' panicked"

If you encountered an error message such as:

"Thread 'main' panicked at 'called Option::unwrap() on a None value', models/src/hasher.rs:35:46", this is a known problem on older Raspberry Pi, especially with Raspbian. Try installing Debian.

Please note, running a Massa node on a Raspberry Pi is ambitious and will probably not work that well. We don't expect raspberry to be enough powerful to run on the mainnet.

Disable IPV6 support

If your OS, virtual machine or provider does not support IPV6, try disabling IPV6 support on your Massa node.

To do this, edit (or create if absent) the file *massa-node/config/config.toml* with the following contents:

```
[network]
    bind = "0.0.0.0:31244"

[bootstrap]
    bind = "0.0.0.0:31245"
```

then restart your node.

7.12 Tutorials and resources from the community

The Massa community is growing and we want to thank everyone who is participating in the project. Some members created useful tutorials and resources and deserve to be credited for it. In this document, we are gathering links of the latest creations.

Please note, these are unofficial and not frequently checked by the core members. Some might be outdated or miss important information. For security reasons, we encourage you to go through the official installation process and tutorials or trusted parties. As always, use your best judgment when visiting third party links.

7.12.1 About Massa

- English - Presentation of Massa
- English - AMA video with RØ CRYPTØ
- English - Article about scalability

- Russian - Presentation of Massa team
- Russian - Massa raised €5 million

- Spanish - Presentation of Massa technology
- Spanish - Summary of AMA during Ep6
- Spanish - Summary of AMA with RØ CRYPTØ
- Spanish - Testnet incentive program
- Spanish - Summary of AMA during Ep10

7.12.2 Setting up a node

- English - Massa-node docker image with multiple features
- English - Tutorial to create a Massa node
- English - Tutorial and resources to run Massa node & client on Docker
- English - Tutorial to launch a node using a VPS (outdated)
- English - How to migrate Massa node to Docker container
- English - Script which sets up a node

- Russian - Article and tutorial to launch a node using a VPS

- Spanish - Tutorial to launch a node
- Spanish - Presentation and tutorial to launch a node

- French - Article and tutorial to launch a node using a VPS
- French - Tutorial to launch a node using a VPS
- French - Security best practices article

- Turkish - Article and tutorial to launch a node using a VPS
- Turkish - Tutorial video for starting a node

7.12.3 Tracking the activity of a node

- Monitoring scripts and commands
- Telegram Bot to track node activity
- Telegram Bot to track node activity and network statistics
- Telegram Bot to track node activity
- Website to monitor your node
- Another website to monitor your node
- Script to monitor faulty blocks of your node

7.12.4 Miscellaneous

- Telegram Bot answering FAQs
- A website with news, articles and tutorials about Massa

—

- Spanish - Security recommendations when running a node

7.13 Massa's Smart Contracts

7.13.1 Introduction

A smart contract is a transaction protocol used to be sure that an operation, involving different stakeholders, is executed as expected.

Having trust between stakeholders is not necessary, as long as each participant is trusting the transaction protocol.

By adding a programmable mechanism to the blockchain, we get such a protocol as long as the program has the following characteristics:

- Immutability
- Determinism
- Audit

Note: Smart contract is not about replacing paper contract with electronic one, but replacing the Law with the transaction protocol. By doing so you're also replacing trust in justice by trust in the blockchain.

In the following, and in the crypto ecosystem, we talk about smart contracts only to refer to a *program* that can be executed on a blockchain.

Nevertheless, it's important to keep in mind that the transaction protocol exists and is valued only thanks to the entanglement of the program in the blockchain and its guarantees.

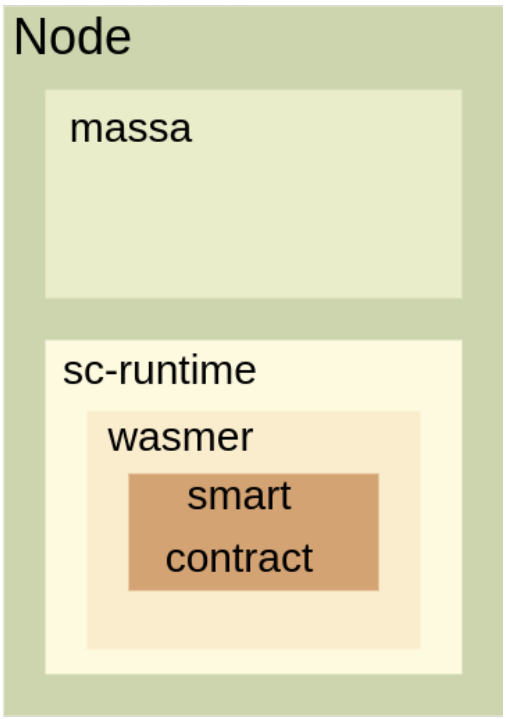
Technical choices

For all the reasons listed above and for performance, Massa chose a program in WebAssembly .

For the sake of accessibility, Massa decided to use AssemblyScript as programming language to compile the expected logic into bytecode.

Finally, at node level, the byte of a program is executed by Wasmer .

The following schema recap graphically all this:



7.13.2 Getting started

In this section you will learn how to compile your first Massa smart contract.

Setting up a new project

Make sure you have a recent version of Node.js and npm. Update or install them if needed.

massa-sc-toolkit is a tool that creates a boilerplate smart-contract project. To create a smart-contract project, invoke the toolkit by running:

```
npx @massalabs/sc-toolkit init my-sc && cd my-sc
```

You have now npm project created with AssemblyScript installed among other dependencies. It will be used to generate bytecode from AssemblyScript code.

Note: Massa smart-contract module (*@massalabs/massa-as-sdk*) contains the API you need to use to interact with the external world of the smart contract (the node, the ledger...).

Congratulations! Now you have a fully set up project and you are ready to add some code.

Note: A few words on project folders:

- *assembly* is where the code goes;
- *build* will be created during compilation and will contain compiled smart contracts.

Create your first smart contract

Since the beginning of mankind, humans explain how to use a program, a new language, a service by implementing a *Hello world!*. Your first smart contract will be no exception!

Open the *main.ts* file in the *assembly* directory at the root of your project. Replace the code in the file by the following code:

```
import { generateEvent } from "@massalabs/massa-as-sdk";

export function main(_args: StaticArray<u8>): void {
   generateEvent("Hello world!");
}
```

Don't forget to save the file. Before starting compilation, just a few words to describe what is used here:

- line 1: *generateEvent* function is imported from Massa SDK (@massalabs/massa-as-sdk). This function will generate an event with the string given as argument. Events can be later recovered using a Massa client.
- line 3: *main* function is exported. This means that the main function will be callable from the outside of the WebAssembly module (more about that later).
- line 4: *generateEvent* function is called with "Hello world!". Brian, we are thinking of you!

Now that everything is in place, we can start the compilation step by running the following command:

```
npm run build
```

Congratulations! You have generated your first smart contract: the *main.wasm* file in *build* directory. Note that a *deployer.wasm* file has also been generated. It will be used to deploy your contract on Massa network.

Note: If due to bad luck you have an error at compilation time:

- check that you properly followed all the steps,
- do a couple a internet research,
- look for any similar issue (open or closed) in massa-as-sdk.

If you find nothing, feel free to contact us on Discord or directly open an issue.

Deploy your smart contract

Your smart contract is now ready to be pushed and executed on the Massa network. In order to deploy it, you need to own a Massa wallet and some MAS coins on it.

Note:

- If you don't have any wallet configured yet, *create a new one*.
- If you're using a brand new wallet, add some coins by sending your address to testnet-faucet discord channel.

In any case, keep the *Address* and *Secret key* of your wallet, you will use it later.

There are two ways you can deploy your smart contract. The easiest and the recommended way is to deploy the smart contract with the smart-contract toolkit (*Option 1* below).

The second option is to deploy the smart contract, through Massa client, by running your own node (*Option 2*).

Option 1: Deploy your smart contract from the toolkit

To send the transaction on the network, you need to provide your wallet's secret key. This is done using environment variable in *.env* file.

```
cp .env.example .env
```

This command will create a *.env* file. Now fill it with your wallet secret key.

Then run the following command:

```
npm run deploy
```

Wait for a few seconds… It should return you the deployed smart contract address.

Option 2: Execute your smart contract on your own node

To execute the smart contract you will need:

- A client configured with an address having coins.
- A smart contract compiled in WebAssembly (see previous step).

Let's go!

Configure the client

Make sure that you have the latest version of the Massa node. If not, *install it* and *run it*.

Note: You can also execute your smart-contract on a local sandbox node. To learn more about sandbox node, follow this tutorial: *Local network generation*.

Execute the smart contract on the node

Everything is in place, we can now execute the *hello world* smart contract on your local node with the following command inside the **client cli**:

```
send_smart_contract <address> <path to wasm file> 100000 0 0
```

Note: We are executing the send_smart_contract command with 6 parameters:

- <address>: the address of your wallet kept during previous step;
- <path to wasm file>: the full path (from the root directory to the file extension .wasm) of the hello smart contract, generated in the previous chapter.
- 100000: the maximum amount of gas that the execution of your smart contract is allowed to use.
- Two 0 parameters that can be safely ignored by now. If you want more info on them, use the command *help send_smart_contract*.

Note: To go inside the **client cli**, open a terminal in *massa/massa-client* directory and run *cargo run*.

If everything went fine, the following message should be prompted:

```
Sent operation IDs:
<id with numbers and letters>
```

In that case, you should be able to retrieve the event with the *Hello world* emitted. Use the following command inside the **client cli**:

```
get_filtered_sc_output_event operation_id=<id with numbers and letters>
```

If everything went well you should see a message similar to this one:

```
Context: Slot: (period: 627, thread: 22) at index: 0
On chain execution
Block id: VaY6zeec2am5i1eKKPzuyvhbzxVU8mts7ykSDj5usHyobJee8
Origin operation id: wHGoVbp8QSwWxEMzM5nK9CpKL3SpNmxzUF3E4pHgn8fVkJmR5
Call stack: A12Lkz8mEZ4uXPrzW9WDo5HKWRoYgeYjiQZMrwbjE6cPeRxuSfAG

Data: Hello world!
```

Congratulations! You have just executed your first smart contract!

7.13.3 Massa's smart-contract examples

Note: This tutorial doesn't assume any existing knowledge of the Massa protocol.

In this tutorial, we will go through all the steps required to create a smart contract on Massa.

You can find the complete project on this Github repository.

Prerequisites

Smart contracts are written in Assembly Script, and so we'll assume that you have some familiarity with it, but you should be able to follow along even if you're coming from a different programming language. We'll also assume that you're familiar with programming concepts like functions, objects, arrays, and to a lesser extent, classes.

Writing your smart contract

Smart contracts on Massa blockchain are written in Assembly Script and then compiled to WebAssembly (WASM). We chose WebAssembly as it is efficient and can be compiled from several languages, including Assembly Script.

Setup

Let's start by cloning the sum example repository. You need *node*, *npm* and *git* to initialize the project.

```
git clone https://github.com/massalabs/massa-sc-examples && cd massa-sc-examples/sum/
↪contracts && npm install
```

Writing the smart contract

Smart contracts are in the *assembly* directory. The *main.ts* will be our smart-contract file.

For this tutorial, we will create a very simple smart contract which calculates the sum of two integers.

You can find it here *assembly/main.ts*.

```
import { generateEvent, Args } from "@massalabs/massa-as-sdk";

function add(a: i32, b: i32): i32 {
    return a + b;
}

export function sum(serializedArgs: StaticArray<u8>): StaticArray<u8> {
    const args = new Args(serializedArgs);
    const a = args.nextI32();
    const b = args.nextI32();
    const result = add(a, b);
    generateEvent(
        `Sum (${a.toString()}, ${b.toString()}) = ${result.toString()}`
    );
    return result.toString();
}
```

Calling function of a smart contract that is stored in the blockchain with some arguments will start an assemblyscript runtime (wasmer). This is why each function that you want to be able to call in your smart contract must be exported with the *export* keyword and must take one *StaticArray<u8>* argument and return a value of type *StaticArray<u8>*.

Here, we are exporting the sum function. In this function, we deserialize the argument into two integers, with the help of *fromByteString* and *toInt32*.

Compiling your smart contract

Your smart contract can be compiled using the command:

```
npm run build
```

Note that a *build/deployer.wasm* file has also been generated. It will be used to deploy your contract on Massa network.

Deploy your smart contract on the blockchain

We'll now turn to the process of putting the smart contract on the Massa blockchain.

For the deployment, you will need a wallet with some coins.

To send transaction on the network, you need to provide your wallet secret key. This is done using environment variable in *.env* file.

```
cp .env.example .env
```

This command will create a *.env* file. Now fill it with your wallet's secret key (also called a "secret key" by Massa client).

Contract deployment is done by calling a deployer smart contract which can be found here: *deployer/deployer.as.ts*. It will store our sum smart contract onto the ledger. The deployer contract already includes your compiled *main.ts* contract and has been itself compiled at the *npm run build* step.

We will send the deployer smart contract to the Massa blockchain with:

```
npm run deploy
```

This command will execute the compiled deployer *deployer.wasm*, and this smart contract will store the *main.ts* smart contract onto the ledger.

You will see an output like this:

```
> sc-example-sum@0.0.1 deploy
> ts-node deployer/deploy-contract.ts

Deploying smartcontract: build/deployer.wasm

Operation submitted successfully to the network. Operation id: <operation id string>

Waiting for the state of operation to be Final... this may take few seconds

Deployment success with event: Contract deployed at address:␣
↪A1PjpgXyXSBeiG1rbXCP4ybhVccYzpysDKYmkymXWd81idutaD9
```

Interaction with the smart contract

We will now interact with our sum smart contract.

To interact with a smart contract, we can write another smart contract that will be executed, or use the *CallSC* function. In our example, we will use the file *caller.ts* in the *assembly* directory.

```
import { Address, Args, call } from "@massalabs/massa-as-sdk";

export function main(): i32 {
    const address = new Address(
        "A1PjpgXyXSBeiG1rbXCP4ybhVccYzpysDKYmkymXWd81idutaD9"
    );
    call(
        address,
        "sum",
        new Args()
```

(continues on next page)

(continued from previous page)

```
            .add(21 as i32)
            .add(20 as i32),
        0
    );
    return 0;
}
```

Note that we use the address where the contract has been deployed: A1PjpgXyXSBeiG1rbXCP4ybhVccYzpysDKYmkymXWd81idutaD9

First we need to compile the *caller.ts* smart contract. For the covenience of this example we have added an npm script *npm run build:caller* which will compile *caller.ts* and write the generated wasm in *build/caller.wasm*

```
npm run build:caller
```

Then deploy the caller smart contract:

```
npm run deploy build/caller.wasm
```

Remember that our sum smart contract computes the sum and emits an event with the result.

You will see this output:

```
> sc-example-sum@0.0.1 deploy

> ts-node deployer/deploy-contract.ts build/caller.wasm

Deploying smartcontract: build/caller.wasm

Operation submitted successfully to the network. Operation id: <operation id string>

Waiting for the state of operation to be Final... this may take few seconds

Deployment success with event: Sum (10, 13) = 23
```

You can call the JSON RPC API function *get_filtered_sc_output_event* to get the event with:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
  "jsonrpc": "2.0",
  "method": "get_filtered_sc_output_event",
  "params": [
    {
      "start": null,
      "end": null,
      "emitter_address": null,
      "original_caller_address": null,
      "original_operation_id": "24zP8RFvj5wPEvu242WKZmCMRtxdK6gVMGkg1a2WM3YannqrMY"
    }
  ],
  "id": 0
}'
```

Do not forget to set the right operation id function params.

Here is an example of what you can find:


```
{
  "jsonrpc": "2.0",
  "result": [
    {
      "context": {
        "block": "qrMVKELonoVrPGE741NVLfELcbSXP3Lk7XHcimeyTi1GGVP5v",
        "call_stack": [
          "A12h7cTMMimawZ4o2yoc7hSJP5EuvrfZKePuPUjL94fNE3phvgo2",
          "A1PjpgXyXSBeiG1rbXCP4ybhVccYzpysDKYmkymXWd81idutaD9"
        ],
        "index_in_slot": 6,
        "is_final": true,
        "origin_operation_id": "24zP8RFvj5wPEvu242WKZmCMRtxdK6gVMGkg1a2WM3YannqrMY",
        "read_only": false,
        "slot": {
          "period": 96370,
          "thread": 27
        }
      },
      "data": "Sum (10, 13) = 23"
    }
  ],
  "id": 0
}
```


7.13.4 Smart-contract Playground

Smart-contract Playground is a simple interface for writing, unit-testing, compiling, and sharing the smart-contract code written in Assembly Script. It runs in your browser's local storage, and allows you to work on one smart-contract file.

This is the first release of the Playground, and as such it comes with a few limitations. The main ones are that you can't deploy or interact with the smart-contract in it yet.

At the moment, Playground is also limited in terms of imported smart-contract methods.

The available methods are:

- getOf & setOf to to interact with data (read and write)
- generateEvent to log blockchain content easily
- unsafeRandom to generate a random Number between 0 and the max Safe integer

The available classes are:

- Address

Despite these limitations, you can explore writing, testing and compiling following types of contracts:

- Fungible tokens
- Non-fungible tokens
- Lottery
- Video games

- Price oracles

Here's an example of a Fungible token contract and its unit-test you can explore directly in the Playground's workspace.

If you need more functionality and methods, or if you wish to request additional features, please submit an issue in the Smart-contract Playground Github repository. We will do our best to provide you with everything you need to explore the power and potential of Massa's smart contracts.

7.13.5 Stack system in Massa

- **ExecuteSC**
  When an account A send an ExecuteSC operation, the stack at the beginning of that execution is: *bottom [A] top*.
- **CallSC**
  When an account A sends a CallSC operation to call a function in a smart contract B, the stack at the beginning of the execution of that function is: *bottom [A, B] top*. Note: A and B can be the same
- **Call from one smart contract to another**
  When a function F from smart contract C is being executed with the stack [A, B, C] and calls a function on a smart contract D, the stack at the beginning of the execution of D's function becomes: *bottom [A, B, C, D] top*. When D's function finishes, the stack becomes *bottom [A, B, C] top* and the execution of F resumes
- **Autonomous SC:**
  A message sent at a moment when the stack was [A, B, C] and calling a target function F of a smart contrtact D will yield the following stack at the beginning of the execution of the target function: *bottom [C, D] top*. Note: C and D can be the same
- **Local execution (not yet implemented, see https://github.com/massalabs/massa-sc-runtime/issues/170)**
  Local executions don't change the stack: they allow executing foreign code in the current context

ABIs

In the massa assemblyscript sdk there is a Object called *Context*. You can import it in your smart contract code this way:

```
import { Context } from "@massalabs/massa-as-sdk";
```

It exposes some useful functions like:

- *addressStack()* returns the full call stack as a list, bottom to top
- *caller()* returns the stack element just below the top
- *callee()* returns the stack element at the top of the stack
- *transactionCreator()* returns the stack element at the bottom of the stack

7.14 Massa's decentralized web

Massa's decentralized web allows you to store websites directly on the blockchain. This feature enables a lot of applications and provides another layer of security to your dApps.

This section introduces all steps to host your website on the blockchain and register it on Massa's DNS service, in order to access it using any traditional browser.

To do so you will need to follow 3 simple steps:

1. Install Thyra on your machine
2. Create a wallet
3. Define your DNS & upload your website on the blockchain

For each of the steps above, you will have the choice to do it manually or automatically via Thyra.

7.14.1 *Why this name: Thyra*

As explained Thyra plays the role of a gateway to the Massa blockain. Massa being a coin minted in the city of Massalia, the ancient Marseille founded by the Greek, quite naturaly the name (entrance, front door in ancient greek) imposed itself. In order to simplify and standardize its writing, we transformed it into Thyra, but its pronunciation tý.ra remained.

7.14.2 Install Thyra

Automatically

- MacOS installation script or simply use this cmd on your terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/massalabs/thyra/main/
↪scripts/macos_install.sh)"
```

- Windows installation script

Note: If you install Thyra with this script using an Ethernet connection, there is high level of chances that you will need to re-install it if you move to wifi. Sorry about that, we're working on it!*

```
curl -fsSL https://raw.githubusercontent.com/massalabs/thyra/main/scripts/thyra-
↪installer.bat >> thyra-installer.bat && thyra-installer.bat
```

- Linux Ubuntu installation script

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/massalabs/thyra/main/
↪scripts/linux_install.sh)"
```

Manually

To install Thyra manually, follow the step-by-step process found here.

7.14.3 Create your DNS contract & upload website

Automatically

Thyra web-interface allows you to add a new domain to our DNS smart-contract, import a website file, and deploy it to blockchain in just a few clicks.

Go to this page and follow instructions: http://my.massa/thyra/websiteCreator/index.html

7.14.4 Navigate web on-chain

To browse and explore websites and other applications stored on Massa blockchain, you now simply have to run Thyra using the cmd below and access any .massa you know about.

```
thyra-server
```

Or you can also search for one you like here.

7.15 Massa web3

massa-web3 is a TypeScript library that allow you to interact with the Massa blockchain through a local or remote Massa node. In particular the massa-web3 library will allow you to call the *JSON-RPC API*, but also to fetch and poll events from smart contracts on the Massa blockchain, deploy smart contracts and much more. Documentation for the massa-web3 library is available on the Github repository. The library is published on npmjs under @massalabs/massa-web3

create-react-app-massa is a minimal React template made for our massa-web3 library.

You'll find examples of frontends using the *massa-web3* library in the massa-sc-examples repository:

- blog is an example of a decentralized blog platform.

7.15.1 Building Decentralized Application

For the decentralized website part, we'll assume that you have some familiarity with HTML and JavaScript. If you want to have more details, you can follow this great tutorial from React from which the dApp is inspired from.

Here you can find an example dApp for the *sum smart-contract*: Massa DApp example: Sum.

Prerequisites:

- having Thyra installed and running on your computer
- having a wallet in Thyra
- the wallet needs to own some Massa coins
- having a smart contract *deployed*

Network: as any blockchain is a network of nodes, there could be practically several networks of different nodes. The main Massa network is called the MainNet. There is also a TestNet which is ran by Massa Labs team, where you can build applications without paying real Massa coins. You can also run your own network, with only one node (see *Local network generation*).

A decentralized application is an application running on a blockchain. Even the frontend can be hosted on the blockchain with *web-on-chain*.

To build such application you will need to use some libraries:

- massa toolkit
- massa assemblyscript sdk
- massa javascript web3

Then you will be able to build a frontend application that communicates with Thyra's API to sign transactions and submit it to the blockchain.

Then, use Thyra to deploy your frontend application.

7.15.2 Debugging

This section will give you some tips on how to debug your dapp while building it.

Work in a local development environment

Run *a local node*.

Start Thyra and bind it to your local node:

```
go run ./cmd/thyra-server/main.go --node-server LOCALHOST
```

Create a new wallet in Thyra.

Modify the file *massa-node/base_config/initial_ledger.json*:

```
{
    "ADDR": {
        "balance": "80000000",
        "datastore": {},
        "bytecode": []
    }
}
```

to add some Massa tokens to your newly created address.

In your smart-contract projects, use this *.env* file:

```
WALLET_PRIVATE_KEY=""
JSON_RPC_URL_PUBLIC=http://127.0.0.1:33035
JSON_RPC_URL_PRIVATE=http://127.0.0.1:33034
```

Use the secret key that the node-client gave you when calling *wallet_generate_secret_key*. This wallet also needs to have some Massa coins, so it must appear in *massa-node/base_config/initial_ledger.json* before running the node with *cd massa-node && cargo run –features sandbox* in massa directory.

Modify the file *massa/massa-client/base_config/config.toml* to the localhost RPC:

```
ip = "127.0.0.1"
private_port = 33034
public_port = 33035
```

Start a client with the command *cargo run* in *massa/massa-client* directory.

You might want to sign transactions to call smart contracts in your frontend application:

```
const options = {
  method: "POST",
  url: "https://my.massa/cmd/executeFunction",
  headers: { "Content-Type": "application/json" },
  data: {
    nickname: "wallet",
    name: "hello",
    at: "A1nyzu9rJKnf2zz8F7mkM5d6ZoThnMuAtUdij9WmcwXWMWJFmEB",
    args: "",
    gaz: { price: 1000, limit: 700000000 },
    coins: 0,
    expiry: 3,
    fee: 0,
    keyId: "default",
  },
};

return axios
  .request(options)
  .then(function (response) {
    return response.data;
  })
  .catch(function (error) {
    console.error(error);
  });
```

This performs an HTTP POST call to Thyra, asking to create a transaction that will call the function named *hello* of the smart contract located at *A1nyzu9rJKnf2zz8F7mkM5d6ZoThnMuAtUdij9WmcwXWMWJFmEB* with the wallet name *wallet*.

This opens a password prompt and you will be able to see the response of the call in the development tools of the browser. The response is the operation id, for example *2mLLkdKB4BY4hJQFNwGZ5oQVhky13EVZAwHJXCGQnd3FRHUoHw*.

If the operation failed, the response will contain the details.

Now you can now use commands of the node client to see the details of your operation:

```
get_operations 2mLLkdKB4BY4hJQFNwGZ5oQVhky13EVZAwHJXCGQnd3FRHUoHw
```

This will output something like

```
Operation 2oCTeeYMUt6hQuGAzCWbGBajXPPhz48VfHaEMWd6MJkVywa7Sa[ (in pool)]
In blocks:
  - 2Fvqyzd7jfYjPxS9t9UCEGoHhhuvD8HHAmD45CNzEnFp4bbFT5
Signature:␣
↪ELrvQeuso9pjk5ErcG4WjESDpV8iFdweB45jic2UErSugSLLinbqx6JeKamxfXRZtBQoZi9ZcbvsGK858yx2FPsUoHrJJ
```

(continues on next page)

(continued from previous page)

```
Creator pubkey: P12fnkn6fMhjZHatWsQ3k1L4B7hb99XdjbWCmVthxCwbisRAMk2P
Creator address: A12LempfubRfRZoRh7Dr4nWyTUYKAQs7Be1cZ9Gejcx7NXYG4HXD
Id: 2oCTeeYMUt6hQuGAzCWbGBajXPPhz48VfHaEMWd6MJkVywa7Sa
Fee: 0
Expire period: 2314
Operation type: CallSC:
  - target address:A1nyzu9rJKnf2zz8F7mkM5d6ZoThnMuAtUdij9WmcwXWMWJFmEB
  - target function:sum
  - target parameter:
  - max_gas:700000000
  - coins:0
```

Note: *target parameter* is empty because parameters are encoded into bytes so it is unlikely to have printable characters.

You can also see events emitted by your contract (assuming that your contract is deployed at *A1nyzu9rJKnf2zz8F7mkM5d6ZoThnMuAtUdij9WmcwXWMWJFmEB*):

```
get_filtered_sc_output_event caller_
↪address=A1nyzu9rJKnf2zz8F7mkM5d6ZoThnMuAtUdij9WmcwXWMWJFmEB
```

```
Context: Slot: (period: 2238, thread: 22) at index: 0
On chain execution
Block id: 7ALxLxsJfLMMTYRLcGA42bXGbDKSiTPYWyoGWupKMJxXYNHmb
Origin operation id: 24kaBAdcCHUsgcfCUBxUXNBquJyz91Whon1xwbWmux1j29MEC
Call stack: A12LempfubRfRZoRh7Dr4nWyTUYKAQs7Be1cZ9Gejcx7NXYG4HXD,
↪A1nyzu9rJKnf2zz8F7mkM5d6ZoThnMuAtUdij9WmcwXWMWJFmEB

Data: Sum (2, 3) = 5
```

In this example the event message is *Sum (2, 3) = 5*.

7.16 Types

The following AssemblyScript types can be helpful in your smart contract journey without having to reinvent the wheel.

Note: You know a nugget that could be added to this list or you have a specific need for a new type? Open an issue to discuss about it!

7.16.1 Currency

A representation of a monetary unit used to express a value.

Usage

```
import {Currency} from 'mscl-type';

const euro = new Currency("Euro", 2);
const yen = new Currency("Japanese yen", 0);
const isSame = euro.sameAs(yen); // False
```

More info at module repository.

7.16.2 Amount

A representation of a value in a *Currency*.

> **Warning:** *Amount* implements *Valider* as some operations, such as subtraction leading to a negative value, can result in an invalid *Amount*.

Usage

```
import {Currency} from 'mscl-type';
import {Amount} from 'mscl-type';

const euro = new Currency("Euro", 2);

const price = new Amount(500, euro);
const accountBalance = new Amount(100, euro);

cont isEnough = price.lessThan(accountBalance); // False
const isValidAmount = accountBalance.substract(price).isValid(); // False
```

More info at module repository.

7.16.3 Valider

An interface to unify how invalid types are handled.

Note:

- Exception handling proposal is not yet implemented in Wasmer or in AssemblyScript;
- *Result* type is not implemented;

Then this is the only way to perform an action on a type and check later if the type is still valid.

Usage

```
import {Valider} from 'mscl-type';

export MyAwesomeType implements Valider {
    ...
    isValid():bool {
        // check if the type is still valid
    }
}
...
```

More info at module repository.

7.17 Local network generation

7.17.1 How to launch a local network with custom settings and initial coins & rolls repartition

On Docker

Full documentation about launching a local network on Docker is available here : https://github.com/massalabs/massa-network-simulator

On your OS

Clone massa:

```
git clone git@github.com:massalabs/massa.git
```

Compile it with the sandbox feature enabled:

```
cd massa && cargo build --release --features sandbox
```

Create a keypair in massa-client:

```
cd massa-client && cargo run
wallet_generate_secret_key
```

For the rest of the tutorial we will use theses abbreviations:

- *SECRETK* : The secret you just generated
- *PUBK* : The public key corresponding to SECRETK
- *ADDR* : The address corresponding to PUBK

Setup your node to use the secret you just generated as its public key and staking key:

- modify or create the file *massa-node/config/node_privkey.key* :

```
{"secret_key":"SECRETK","public_key":"PUBK"}
```

- modify the file *massa-node/base_config/initial_ledger.json*:

```
{
    "ADDR": {
        "balance": "80000000",
        "datastore": {},
        "bytecode": []
    }
}
```

- CLEAR and modify the file *massa-node/base_config/initial_rolls.json* :

```
{
    "ADDR": 100
}
```

You can now launch your node :

```
cd massa-node && cargo run --features sandbox
```

On your client run the following command to add your secret key as staking key:

```
cd massa-client && cargo run node_add_staking_secret_keys SECRETK
```

The network with your node all start in 10 seconds and you can now interact it with the CLI client like a testnet node. If you want to run multiple nodes on your local network you need to use *On Docker*.

7.18 Useful resources

Here is a list of useful resources for smart contract development.

- massa-web3 is a TypeScript library that allow you to interact with the Massa blockchain through a local or remote Massa node.
- create-react-app-massa is a minimal React template made for our massa-web3 library.
- massa-sc-examples is a collection of smart contracts examples.
- massa-sc-toolkit is a toolkit meant to facilitate smart contract development, testing and deployment.
- massa-sc-tester minimal testing environment made for running massa smart contracts locally.

7.19 External resources

Warning: Links to external websites and material are provided for informational purposes and do not constitute endorsement or approval by Massalabs. We are not responsible for the accuracy, legality or content of any external sources referenced on this website.

As always, use your best judgment when visiting third party links.

- Bearby extension: a Massa wallet plugin for Chrome and FireFox.

7.20 Bootstrapping in Massa

7.20.1 Introduction

Nodes that are already part of the network are able to follow the State by observing the blocks passing through the network, verifying them, and applying the state changes they cause.

However, new nodes joining the network need to get an absolute "current" version of the state, which is called "bootstrapping". In some blockchains like Bitcoin, full nodes joining the network are recommended to download all blocks from the beginning (genesis) of the blockchain in order to re-verify the whole state change history.

However, Massa has a triple decentralization/security/performance goal:

- maximal decentralization requires that node hardware requirements stay consistent with a typical consumer desktop computer to lower the entry barrier of becoming a node runner
- maximal security requires that all nodes verify all blocks and operations
- maximal performance requires using the node hardware to its fullest (CPU, network, memory, storage)

This means that the State in Massa evolves almost as fast as typical consumer desktop computers can run blocks, which implies that catching up blocks since genesis goes only slightly faster than new blocks appear in the meantime, and would take a very long time. Moreover, Massa aims at processing thousands of operations per second, which means that it produces a lot of block data every second, thus preventing nodes with the target hardware from storing the full block history and making bootstrapping from genesis impossible since old blocks are forgotten.

Massa nodes joining the network must therefore bootstrap by downloading the absolute current State.

Note that nodes can recover from short-term disconnects by asking for missing data from nodes around them once they come back in the network. However, since Massa nodes only store a short history of blocks and forget older ones, it is impossible to recover from long disconnects since the surrounding nodes have forgotten the blocks needed by the recovering node. In that case, a new State bootstrap is required.

7.20.2 Security model

The example of Bitcoin

To understand the security model of node bootstrap, Bitcoin is a good starting example.

When Bitcoin node runners decide to join the network, they first download the node software from a central source (eg. bitcoin.org). If that source is compromised, the node might end up on a different network, and/or private key theft might happen. Bitcoin therefore requires trust in the entity sourcing the node software.

Note that optionally, if the nodes don't wish to download the full block history they resort to trusting a more recent "checkpoint" state encoded in the node software. If that state is compromised, the node's knowledge of the whole ledger might be skewed.

Assuming the node software is not corrupted, new nodes joining the network need an initial list of peer nodes to which they need to connect first in order to discover the network. This list is hardcoded in the node software (see the bitcoin docs). If all peers pointed by this initial list are compromised, the node can end on a different (non-bitcoin) network, even if the node software itself is not compromised. Bitcoin therefore also requires trust in at least one of the initial peers.

Note that ending up on the wrong network can be detected by checking block hashes with an external source. But this requires trusting yet another source of data.

The case of Massa

The Massa case is very similar to Bitcoin's. Node runners also need to trust the source of the node software they download, as well as the initial list of peers.

Similarly to Bitcoin checkpoints, bootstrapping Massa nodes must obtain the current State from a trusted source, ideally the same source as the one they downloaded the node software from, in order to avoid having to trust multiple entities.

Downloading the state from an untrusted source can result in major issues such as coin theft. As such, bootstrapping from untrusted sources should be discouraged, and bootstrapping other nodes should be opt-in for node runners to avoid "bootstrap lists" circulating as the default way of bootstrapping from unaware node runners.

7.20.3 Implementation details

Procedure from the point of view of the node being bootstrapped

Massa nodes that bootstrap start by connecting to a randomly chosen node among the ones listed in *massa-node/base_config/config.toml* (section *Bootstrap/bootstrap_list*).

The bootstrap process uses a separate port and protocol than the normal Massa peer communication.

All communications with the chosen bootstrap node are authenticated using the public key (node ID) of the bootstrap node (in the *config.toml* file, section *Bootstrap/bootstrap_list*) to prevent man-in-the-middle attacks.

The node being bootstrapped then attempts to download the current State, as well as an initial list of peers from the bootstrap node.

Once successfully bootstrapped, the node can then connect to peers, discover the rest of the network, and process live incoming blocks to keep its state up to date.

In Massa, the hash of the state is used as part of the proof-of-stake seed, which is a safety mechanism against malicious bootstrap nodes sending a compromised State. It ensures that nodes with an altered State eventually end up isolated from the real network because their proof-of-stake draws differ which causes them to discard incoming honest blocks. Note however that a PoS seed mismatch can take up to 2 cycles to be detected.

In case of bootstrap failure, the bootstrapping node retries with another randomly chosen bootstrap node after a delay.

Procedure from the point of view of the bootstrap node

Massa nodes can bootstrap other nodes, with certain limitations because the procedure is heavy for the bootstrap node.

The bootstrap system listens on the address/port defined in *massa-node/base_config/config.toml* (section *bootstrap/bind*). The node's bootstrap server can be disabled by removing the *bind* entry from the config file.

The Massa State is large (~1 terabyte in the worst case), and takes time to upload to bootstrapping nodes. During that time, new changes to the state continue to appear, so new changes affecting already-uploaded parts need to be sent on-the-fly.

By default, Massa nodes only allow a whitelist of IP addresses to bootstrap from them. This list is present in the *massa-node/base_config/bootstrap_whitelist.json* file. This list is intended to prevent flooding attacks by attackers pretending to be bootstrapping, and also makes it more difficult for node runners to bootstrap from untrusted sources. If you wish to disable whitelisting and allow anyone to bootstrap from your node, simply delete the *bootstrap_whitelist.json* file and restart your node.

A complementary *bootstrap_blacklist.json* (absent by default) can also be created alongside *bootstrap_whitelist.json* (and following the same syntax) in order to explicitly prevent certain IP addresses from bootstrapping from the node.

Once a node has accepted to bootstrap an incoming node, it adds the incoming node's IP address to a local cache preventing that IP from bootstrapping again for a time defined in *massa-node/base_config/config.toml* (section *bootstrap/per_ip_min_interval*) by refusing subsequent connections from that IP during the config-defined delay. The exclusion delay is not extended if the remote IP attempts new connections during the exclusion delay. The exclusion delay is however applied if the bootstrap was accepted but failed for any reason. This aims at limiting the load on individual bootstrap nodes, and spreading the load among bootstrap nodes.

The number of nodes simultaneously bootstrapping from the local node is limited (*massa-node/base_config/config.toml* section *bootstrap/max_simultaneous_bootstraps*). Excess attempts are refused but do not trigger the exclusion delay mechanism.

7.20.4 Future optimizations

We plan to add the possibility to download bootstrap data from untrusted sources for load-balancing, but then check the hash of the obtained state from trusted sources, and only fallback to downloading everything from trusted sources if multiple bootstrap attempts from this hybrid approach fail.

7.21 Storage Costs

7.21.1 Explanations

In Massaa the ledger is shared across all nodes of the network. We need to set a size limit to be able to run a node without having 100TB of storage which will cause a barrier for adoption and running nodes at home. We chose to limit the size to 1TB. Everyone can store data until the ledger reach 1TB. But how can we ensure this limit ?

We chose to force users to lock coins when they claim storage space and so we created a correlation with storage and circulating coins.

For each byte of storage you claim (for your address and balance, a key in your datastore, bytecode, ...) you need to lock coins. The coins are released when you release your space in the storage.

The amount of coins you need to lock for one byte is 0.00025 Massa. This value has been chosen so that if half of the coins (250 000 000 Massa) are locked, we will reach the 1TB.

As balances are stored as varint in the ledger their size can vary. To avoid difficulties and incomprehension we decided to use a fixed size for each balance. This size is 8 bytes and so initial ledger entry (address + balance) cost (8 + 32) * 0.00025 = 0.01 Massa . Datastore keys also have a variable size and so we decided to use a fixed size of 10 bytes for calculating storage cost.

If you want to calculate the storage cost of your address in the ledger the formula is : address_size + balance_constant + bytecode_length + sum of (constants datastore key + value size) = 32 + 8 + bytecode.len() + sum_i(10 + datastore[i].len())) * 0.00025

The storage costs are always paid by the address that calls the ABI. For instance, if you are using the ABI *set_bytecode* or *set_bycode_for* you will be charged for the storage costs.

7.21.2 Example

To create your address on the blockchain someone need to send at least 0.00025 * (32 + 8) = 0.01 Massa when sending the operation that will create your address (a transfer for example). This is the cost of creating your address and balance.

You want to store your birth date that is 30 bytes long in your datastore so you need to send an operation that will create a key in your datastore using a SC. This operation will cost you in storage costs at least 0.00025 * (10 + 30) = 0.01 Massa.

Now you want to delete this entry on your datastore you will be refunded of the storage costs (0.02 Massa).

7.21.3 Notes

In case of refund of the storage costs, after realeasing space, the address reimbursed is the one that calls the ABI, possibly different from the one that paid for this storage.

In the case of a *CallSC* operation, the storage costs are paid by the SC.

If you are a SC developer and you want your users to pay for the storage costs of your smart contract you can use the coins that are passed by the *coins* parameter of CallSC. You can also save their address in your datastore in order to refund them later.

7.22 Massa JSON-RPC API

This crate exposes Rust methods (through the *Endpoints* trait) as JSON-RPC API methods (thanks to the ParityJSON-RPC crate).

Massa JSON-RPC API is splitted in two parts :

- **Private API**: used for node management. Default port: 33034 e.g. http://localhost:33034
- **Public API**: used for blockchain interactions. Default port: 33035 e.g. http://localhost:33035

Find the complete Massa OpenRPC specification here.

7.22.1 Integrations

- **JavaScript**: use massa-web3.js.
- **Playground**: use Massa Playground.
- **Postman**: use Postman collection.

7.22.2 Error codes

When a call to Massa API fails, it **MUST** return a valid JSON-RPC error object .

Code	Message	Meaning
-32600	Invalid request	The JSON sent is not a valid Request object
-32601	Method not found	The method does not exist / is not available
-32602	Invalid params	Invalid method parameter(s)
-32603	Internal error	Internal JSON-RPC error
-32700	Parse error	Invalid JSON, parsing issue
-32000	Bad request	Indicates that the server cannot or will not process the request due to something that is perceived to be a client error (for example, malformed request syntax, invalid request message framing, or deceptive request routing)
-32001	Internal server error	The server encountered an unexpected issue
-32003	Service Unavailable	Indicates that the server is not ready to handle the request
-32004	Not found	Indicates that the server cannot find

Error example:

```
{
"jsonrpc": "2.0",
"error": {
    "code": -32400,
    "message": "Bad request: too many arguments, maximum authorized 2 but found 3"
},
"id": 1
}
```

7.22.3 Explore Massa Blockchain

In this section we'll learn how to interact with Massa blockchain via *curl* commands which will create JSON-RPC request calls.

Warning:

- We'll use only public API methods in testnet node.

Public API

a.k.a. **user mode** methods (running on https://test.massa.net/api/v2)

get_status

Summary of the current state: time, last final blocks (hash, thread, slot, timestamp), clique count, connected nodes count.

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_status",
    "params": []
}'
```

- Result:

```
{
 "jsonrpc": "2.0",
 "result": {
     "config": {
         "block_reward": "0.30",
         "delta_f0": 1088,
         "end_timestamp": 1667142000000,
         "genesis_timestamp": 1665405000000,
         "max_block_size": 500000,
         "operation_validity_periods": 10,
```

(continues on next page)

(continued from previous page)


```
        "periods_per_cycle": 128,
        "roll_price": "100",
        "t0": 16000,
        "thread_count": 32
    },
    "connected_nodes": {
        "P126wpqvDP8GQqeS4WZq9fwRsmzAKrKfQQdXigK3zw53Ai1HW3aN": [
            "147.182.147.178",
            false
        ],
        "P1278WinKWC2RtrbskXwksrGXcHCAWwX8aBhvvMhtEaj3KjejsND": [
            "185.138.164.167",
            true
        ],
        "P12M7iQ4AmbkC2UZXRamHKHgGrq3dJmgCun8QjYQYRRaPcG8Zpvw": [
            "77.222.63.32",
            false
        ],
        "P12QQG516ahWuNaPnRbV4FU8RYuUAH6V7oFqVrotg5xJTXiq73tV": [
            "165.227.35.214",
            true
        ],
        "P12TK7PJreAzh9NrRWFXkKpm354piPupdZsd9i1B7geLJk2fYBVA": [
            "94.250.203.240",
            false
        ],
        "P12cJRDAyctwMQcJ6bxxNbjsnWyKxgHfKVzr9xsJ9A741u4kzYWa": [
            "95.216.156.29",
            true
        ],
        "P12eHJrC3WdZ2qdaLUHP7jNRuaK9WoAV5W4NDjXTgb5mNv38unc8": [
            "167.86.111.35",
            false
        ],
        "P12p6axwXgMW2RrUdFojKaRGvTnb1ajyLkXTnEUcqUjXwnfQMk9w": [
            "173.212.236.220",
            true
        ],
        "P12rJQaPcxj4XNKz1GhfQftxFLNEfJRQzzuXKngimq3VPRSBUAeF": [
            "167.235.145.174",
            true
        ],
        "P12rPDBmpnpnbECeAKDjbmeR19dYjAUwyLzsa8wmYJnkXLCNF28E": [
            "158.69.120.215",
            false
        ],
        "P12vxrYTQzS5TRzxLfFNYxn6PyEsphKWkdqx2mVfEuvJ9sPF43uq": [
            "149.202.89.125",
            true
        ],
        "P12wgY28tM7DY9xD7Auwh3oCijX3XgvkCHnrTqfD5VH6kXp6dkzF": [
            "146.19.24.215",
```


(continues on next page)

(continued from previous page)


```
            true
        ],
        "P16nCxGtVUoEbJE6gGMjPABq3V5RQ1dVB17hqxSSERViB8b1WJN": [
            "159.203.14.185",
            false
        ],
        "P1P19Xw3Kb7bVeQkxpKaJkE5zY7u64gMiJHcVEHpTPwtzvUMa6Q": [
            "5.161.84.250",
            false
        ],
        "P1W6qg7AGkukq16ikJD2Aa41pW6cQfNr123u1KK9yBf92wsi3vj": [
            "84.54.23.207",
            false
        ],
        "P1bxqhJfzre8sGYCF6MA5MW4utVvTJPEKEVnWCLLLifKCUwGsqx": [
            "194.163.189.5",
            true
        ],
        "P1g7MNCLjL9DdFRUWvnwPJg4fjaCQCVke3mSc5k7rFUA7wRbiZB": [
            "95.111.248.121",
            false
        ],
        "P1gP6pLsToXZuFawvcdfYaARv787ezdsQW1Hw27SkwZz2ZgKH9J": [
            "209.126.13.129",
            false
        ],
        "P1hX6SBEU3duEmNuab9QWbh8uLPx7gxDHzSFgNJw4XX5AND5WQs": [
            "139.162.110.127",
            false
        ],
        "P1m6jR5Si3KKQb7VDjpd4HhVstdHJYFHGAKnK9GGszheN6hVtM3": [
            "62.171.141.30",
            false
        ],
        "P1rN38cfybsGB3UgLxWB6qr57MyThVc43imJSKkg5YNgjswnMUF": [
            "144.126.146.140",
            false
        ],
        "P1siwj1nNwHh3HB2bHqU94ESjMgicvxq5kfjDyBpDEUVgwYDFvH": [
            "162.55.181.167",
            true
        ],
        "P1sr9pwXuGAPsrvdnHtiRvQaTkGap92YPptQrEjRcrq8sfqodye": [
            "213.21.221.200",
            false
        ],
        "P1ubGD5Mm3wNmh3zawVR6DUDc3CB4pkDjqmntUGDyVQk4ddAXQa": [
            "194.195.120.50",
            true
        ],
        "P1zGmtwZ7g9tdtwmmhyNvoAE8tdk6qMLw7Pf4uRsBGwyhKEhV6S": [
            "34.125.115.189",
```


(continues on next page)

(continued from previous page)


```
            false
        ],
        "P1zVQSNYWA6bXEGZeJwCgntFJjmvMu8YtgNw9fkiKJ4WmBYXLzo": [
            "65.108.53.204",
            true
        ],
        "P1zb2dnsQpDxcQL3R77fSnhzYXYpwnH5gDXZh4HMa7iAxx57s24": [
            "38.242.158.106",
            false
        ]
    },
    "consensus_stats": {
        "clique_count": 1,
        "end_timespan": 1666542101196,
        "final_block_count": 118,
        "stale_block_count": 0,
        "start_timespan": 1666542041196
    },
    "current_cycle": 555,
    "current_time": 1666542101196,
    "execution_stats": {
        "active_cursor": {
            "period": 71068,
            "thread": 22
        },
        "final_block_count": 105,
        "final_executed_operations_count": 53541,
        "time_window_end": 1666542101196,
        "time_window_start": 1666542041196
    },
    "last_slot": {
        "period": 71068,
        "thread": 26
    },
    "network_stats": {
        "active_node_count": 27,
        "banned_peer_count": 0,
        "in_connection_count": 16,
        "known_peer_count": 10033,
        "out_connection_count": 11
    },
    "next_slot": {
        "period": 71068,
        "thread": 27
    },
    "node_id": "P1VRyXjUaHeJd4Rmr3waVmpZDFzzH5ARRi3f5ye5BYgxBmxHC7X",
    "node_ip": "141.94.218.103",
    "pool_stats": [
        168394,
        1344
    ],
    "version": "TEST.17.2"
```


(continues on next page)

(continued from previous page)

```
 },
 "id": 1
 }
```

get_cliques

Get information about the block cliques of the graph.

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_cliques",
    "params": []
}'
```

- Result:

```
{
    "jsonrpc": "2.0",
    "result": [
        {
            "block_ids": [
                "4ba4uaiofBMAdgXC8zxLaBygieGBB3KyeSGcSrSMMbV9m6sK1",
                "2sMQGvSuoA1VzqPPFksLDQbE6zsKzVyzaBqDVuEH7W1DRuB2q8",
                "LmEh7ttGxVr8nFi4t9VNMzCXt3PkjgwFg7pEbbqfkqcarco7r",
                "t6NxNRaiimUGctzfiMhwqA3uYFGcCAs3KawwgzP7huwAbki88",
                "Va6njRuA9UyXKDyzj5FqWH7yRsanMBtZByBxfGhqapTyZRvYS",
                "Do79PdHf7rrssmHmgay4Ypy4kGw4rNwzPq1LZdhdizHSUpj2C",
                "Ke1LUGXeHNwo7EWpNbVMLQvtCAQdKcdvsVqNJuEAwgiHz8RNP",
                "2NMDv2JHUKDEFsGm6HAsznAeCKbkLdMmMuTkchUKFMeT1R95rx",
                "24iPfMJBE5Z89UW3QN1uY4Eu9p6vvToLpqgA3XYAhUSF7XqcUJ",
                "23dAyzzeKuREJPverLXfVLiXkErz69Rj5dPsrCAua8Rq2Bebe3",
                "5L1SZveEZBqKHMNwihHLp94hZweiC3m3aAJ68hSDRNpCN8GAx",
                "5M1W27bkw8zr4PWTQpW61F8eNS1YBYrN9n9ZWP69cSwKDfx1p",
                "TkVkE8aFTU3qW2cH4PgdAuicaRojkwN9HSfg3TVzY32XKab3R",
                "2tZckQhui23EEw96awyfYn7B4TUSukoGoZwypkNM1fHymz82dg",
                "2qsYEAg243dVmj3jCFbiy3BxokPfa5FscYaGoMybSHMW7Rgt1P",
                "2hoTAg6gK25xde5NbvuoWY5kHwqoLW8vQLDnkx2H5oefHYFswv",
                "2oacHhWZDpgLTVTZ1m3zaF59dBbKcysvn35jFCptkdVuS9D6go",
                "5eexoZkpCUEDsiBKPquix1ivwvoS74DTPkqagVAs9kHTrN2cW",
                "23V2yXFGu8PEgmeb3AyGWWtC74PqZ7krnnaTM7Zcf4rVMbFavR",
                "aHCAUV5aLLgotKB8V1WJqAaWJGpsYo9RV7T9DRaEtgXTnTKoP",
                "SVGhtws4yWB1Q1LKcrFR83CnArCqT6kptnpAuxUc4bYtLByQB",
                "e7JUqLsrAQa9mNsbUzb8rtbSjRMi6kt4fuBNYWJ4qLBFxHx5B",
                "2fcDY7aiLGL9o8PkssgANsLRcQ1gZBi75L1hP3hyGRNDpnaFdN",
                "kqCFcbpM4PB3SL411Kvi9yo5jhCwy7ZZZLfE5NV7qeDWeLhwT",
                "qpm9CSQgKmqomNZpG7yQNsoB2qz3GLyduECaybvqDfVNZMU5X",
                "SaHyMRiWXLn5GVWQtiUpXFY17qpDN3LENRgDuQmoL7BuKCNEZ",
```

(continues on next page)

(continued from previous page)


```
                "2SGMPZWdhYm9yvDt3yKuBRa2hJaCQB411U455jwYzKQqY6cXjG",
                "23pyjncfV9xZYkmQEJjLuxtW8uCC714TR2qHoNQ1XthpmowGym",
                "2AtdgJJqjV7fN9sm48swQ5cwDpzwznmsPerfJJwj9UfzV4E9zn",
                "dz9VRGrgfuxuyDPwPNsKQTTtWzsYcHHCFKbX74hKeiZiHWnpk",
                "2t8WKiLTjtVy8bG8kY4NweqSNCZXHYsiPotAz1mQPYb5uBSg7z",
                "v1QPVXtuF4qq1bep4QEZXbTHZNEq1XNzx85z87zjfDx99MX2j",
                "Ba1qfZUgsamwbzRbC37C7qktGZHm8S7vNvtjFY5ZMQdE5JtFX",
                "2sVmHfTCV1wiEL2TvHtsFm6z72T6fQrCL3hHo791x1G2XeeFo8",
                "2DQQN8ZVm7Uemp6SSGj57Haa3Z6dmThcMLSFCRfqeN3PGCo2mg",
                "2SdGdaqiMn1ygxnxSR9rze6H4xRwgDCzMQ654hGKCAHqKYhi79",
                "t6JkC332fABHr3Eyu3AjxXotuAxCLme9noyq2btoQyd98bb9n",
                "3nPhmDJFs21QqKLM8QnH1YXBY97ntam2xSEMiGkQnUtLBL1gF",
                "ZijiXRYJTvw1rDYRQVysQqKVx4Qr3icEYAXdP8a7EeE7BLh6N",
                "2QjvGQXNGAv1UnEbrbfgsivJicF6GMkGFGS9Bbtn5BrG2xZqoN",
                "dvTaZff8Myoff5HrQxZGkhtoVjQEM6Pctb54kvYHFGbXLPQM1",
                "itp7T8Y8zdkGjMRdtrRGMoV9u76RXvqZ9BCgLhxzKMvaiSGV",
                "9QRsXmeTr5AYRv6sLxNFZ6wDFwH5EX4BwBc43uMEJTWUydgws",
                "stxd3WPygF2f7oBPbcCJPKSLSftg16VAnZBYieBVt9yUi9hLQ",
                "5JFwVNbK3unFqq3L5dEno44eJp4KCxiic7T8NaFhpqHuLocRb",
                "5zcfYVtQAiVNe4o8SFNd1xXY2idF5rVG7Qx8cYPQaRWdkapTJ",
                "2w2z9ygGDRSMYvA86Mx5SE1fKgd5brcCLQ6xEb3JfWjyu932y6",
                "2nnFjXay8V8bVhTsf3PgxZwX3Hbpnjq1XHQUG7yFqbD2NAQo2C",
                "QVniT5MFNwCxoE8DmxmVbHjEAHMreoQsyhE4XFgAr9GeLHxdD",
                "CarArU3oR94FmbBcerU2agh5tb9g1Y4di9NrzUf3kt5LwrZKQ",
                "2BG36zn9QMe2hDrVov3JbHeYYKg4LxEJv3Fo7eAogMTLn9agJ9",
                "HQjsVy1LdscK3dNywgcbsZhVRn81VAPKVdhVQ7oipnzrEanR9",
                "UC1nMQoCStJ47BXPwrqjKX7Symfv4a9yj7FHCCK1HGAsvBnFH",
                "26dJaQ8tSyES5NoPLRGcdpawGH8ZX2irMhUUokZ12taZNcwTsV",
                "grz7vaHHRsKPfiAkNHjt2A5GDDVAHNnThrwQ85iBRgWRLgudX",
                "RoHBFgxpafkXR9utXJLAanbVAs3Qo8NfCJUtBfR5dnfyWikcs",
                "2M5mLpaWBkcWB7EkqdAJjcPBWUxBKzBwYMLDB6EjWXch3qTQt6",
                "2WYzs5RTQasBPpYYsqdAnoVBSuQkHcAM45tcn7a6Eh9omizNZw",
                "etKfjwLKac5TfnGwJqkyTUHSTp1bmQrqMYpsVXUJs9cpUzPYn",
                "jUkFF9VfBwsPqm5hZuUJ6JqKfRUNLcki1dWEZLQQvnvWzs55",
                "hrRHmATD2tNT1LvFUuxH92eGsPCkd7ADwYPEfZsANdCTR1NGR",
                "2dqq6SN1KTV9QpMUV6cQKkzBYKL7uaDq7vVeDPLkfUJfGWmtdh",
                "6XpZdE1jZ2HwFx4Uekk2udCdqF6C1SZjDDwAdf6nyHbK6zN22",
                "k7gHH5YPQ2sxKNbq7fFimrSTH5UuHoyHdn8KKKJT2DdYJBWow",
                "xSaWHnBY9amMunmZzuk8jHvAi5QMWCBYkhUadisWWDhKLt9zf",
                "fHiU5y72SmHqS1aBsauar1vCA3XCtn8jivsoCApfsVH4Mfdd"
            ],
            "fitness": 1112,
            "is_blockclique": true
        }
    ],
    "id": 1
}
```

get_stakers

Get information about active stakers and their roll counts for the current cycle.

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_stakers",
    "params": []
}'
```

- Result:

```
{
    "jsonrpc": "2.0",
    "result": [
        [
            "A12RHPuU7JFS2rxvxL6MnzVoBJAZr7ivFFJuiRPv4mi5wv8z8VYm",
            112
        ],
        [
            "A12axF2vj3GMV87LV5cEtJwntrzTJXQsYCsp1jtXXqCkiF1X6VwX",
            80
        ],
        ...
        [
            "A112oKyfHsRyaLHdgRDY7EkD1X2Rt8UnMr226BjPxirEsJbFjez",
            1
        ],
        [
            "A114oowRjFLH5nWuL2nhc6RmN2RYZpXu6TXbs1dTxF41Qvwd3Ku",
            1
        ],
    ],
    "id": 1
}
```

get_addresses

Get information about address (es) (balances, block creation, ...).

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_addresses",
```

(continues on next page)

(continued from previous page)

```
    "params": [["A12s675r1Kn1i7BF8QRVCdqPFiNeAZ1fojs1q2jun6wEGbow1brZ"]]
}'
```

- Result:


```
{
    "jsonrpc": "2.0",
    "result": [
        {
            "address": "A12s675r1Kn1i7BF8QRVCdqPFiNeAZ1fojs1q2jun6wEGbow1brZ",
            "candidate_balance": "84.243137236",
            "candidate_datastore_keys": [],
            "candidate_roll_count": 1,
            "created_blocks": [],
            "created_endorsements": [
                "Yed7BJj9QqGG3tDCqoDTn7uMfGJrvPVh9agCYhNoCUUPwHfD3",
                "TLrtZAgEyHuUooRMCZj6mVXz11QeRvr8WoudTSFLeTku5J5nf"
            ],
            "created_operations": [],
            "cycle_infos": [
                {
                    "active_rolls": null,
                    "cycle": 590,
                    "is_final": true,
                    "nok_count": 0,
                    "ok_count": 2
                },
                {
                    "active_rolls": null,
                    "cycle": 591,
                    "is_final": true,
                    "nok_count": 0,
                    "ok_count": 0
                },
                {
                    "active_rolls": null,
                    "cycle": 592,
                    "is_final": true,
                    "nok_count": 0,
                    "ok_count": 0
                },
                {
                    "active_rolls": 1,
                    "cycle": 593,
                    "is_final": true,
                    "nok_count": 0,
                    "ok_count": 0
                },
                {
                    "active_rolls": 1,
                    "cycle": 594,
                    "is_final": false,
```


(continues on next page)

(continued from previous page)


```
                    "nok_count": 0,
                    "ok_count": 0
                }
            ],
            "deferred_credits": [],
            "final_balance": "84.243137236",
            "final_datastore_keys": [],
            "final_roll_count": 1,
            "next_block_draws": [
                {
                    "period": 76077,
                    "thread": 4
                }
            ],
            "next_endorsement_draws": [
                {
                    "index": 15,
                    "slot": {
                        "period": 76081,
                        "thread": 1
                    }
                },
                {
                    "index": 0,
                    "slot": {
                        "period": 76081,
                        "thread": 2
                    }
                }
            ],
            "thread": 30
        }
    ],
    "id": 1
}
```


get_graph_interval

Get information about block graph within the specified time interval.

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_graph_interval",
    "params": [{"start": 1666559894589, "end": 1666559896589}]
}'
```

- Result:

```
{
    "jsonrpc": "2.0",
    "result": [
        {
            "creator": "A1DGpvoAMv2GAWKS2BGF4iFQaq6bHDgpfu2vhGFogZMcaSsy7DY",
            "id": "D6kTS4Wh3B7FRDCw6ncqrCuY7NVPYBbGwdSg814Kd13fS7xQa",
            "is_final": false,
            "is_in_blockclique": true,
            "is_stale": false,
            "parents": [
                "2GcJHmGY1QEyWmr4Rh2QSWcxE5icu8szTLJMyZx6fSGZDFETBZ",
                "2A1PFNRpR1MtJYwnp9vF3Pcc8xQ54mpPDjLgjJfZb1yxzTDXjZ",
                "2nL4CBXZKiv4szqvq4cBTtnfUtk5ozRg9Kd45y4UTRNLuHBLHT",
                "2ZuyfDizeBfMgUHgsLgYd7nRMVbk253A1YZUSpjY6bq3med7LY",
                "2Sdvt2oBdYXJ2LSP4AYfJ98DE4mGsBpC2pSLWudYkL92utv9EU",
                "24VPQmcBaFCma9ypn6MMki8FxNQYwcyYnXhhCdAACX1dqFQB94",
                "trM5GjcVp6z4MWrTxyNJGzPQSR8mbNAnaPqLBLUw8vVaTEdNb",
                "AfsZ11V2sCcJnWjD68yrXq1D7gvcv36vByXUAGsuohp93yW7u",
                "eDdqMET8smfSpu93pd8iPsNnnEuhutvH9AqXyXdRRf5GVDK2w",
                "tbszcUiCBqq6ty33wHq52wZ6kdyTo591KyBDfY4FeWaDueKM7",
                "tXTMdL67gYMFiNyugNTBLP9dXMrd5hW4yYG7k4iwZtzhWsD8u",
                "YtQ1UxfuKrgCNYCzbjhDEUbGeRP52j2XizHuK785L7DhHJ1Xr",
                "ybCiSCUPGJBo9FAKE4zus4CG2sSsxFNmoc7qD2Xrn7TjG5TqN",
                "zLPd1vNoYuzHuy3kWg5hbfxKaxSAKk9JGYYd38QyMiVZ6K3BF",
                "2SAHtG3Jn4VHnbzo5bohbLqL7cx7MwQGUr8V6CRaLWsVQetBfu",
                "2N6Wa26nMkx6yuCPwSGm9Wd1EP7u2Ad67et1evuLj4osEWUTYJ",
                "2NVq8igEKnXZ1ysatU9xo66PVGZKx8MwQqRfHmRPg6vjeTy3pP",
                "W5vKLkAyVtG6BNjm4WHC3a7Dz53KDf8v8aKcDjYxRFTXmDd1q",
                "2q3KnZ4tPfEZStQw7LbGCwfehyPckSpWjQc3cuwCWpzSb7XxRr",
                "2akcCnYAFnG9RgVWxprsNgwAv5WDQ8Lm9TfTGb7nrsHn4iiDA2",
                "k3PNeH648id9knfLT2qPpv46AVpZUXaM8qSHgNDx7uw1ieCZL",
                "2YTQhFoBdBo8ofq4ZRJofCQAGgbrQR27CcvgDkdUKMU4H1Fv8t",
                "2XcCTND2GpDSmouGD7ev5JHJNYP5gisgpruMkC5G48d57rBQ5i",
                "4MEYgdhAUUgXLy9CgCMuZTfwTZPLhMbrNm5TFWhDk6EVAoScf",
                "exawL8H3A42zLr5UcCvjMY4TBBn5u84PMtBCrfyyZPWUhayZt",
                "LP8FabDMiAwNkqcDs4z163fsU4jEtRqK9j6sfeXPyNqLJye2L",
                "2gnfhUjLvbRjzu95iTcHSAwF8SenfsCwtLQP3HAEo5Y1NdTUqs",
                "44S9aCFRVd7zDeTBbbTnjHoqy7Up7EpzLVBmARnyfb6HiJENE",
                "uM6w8xGvBxjUYSFzn7DUUV3RUoj8V1iPGGka4ap2g6vCvCqoE",
                "2eejA5Y81RdvDbk2iVFayPvkFpZvm91dNPkq1r1TQMFoaxdwFA",
                "2o1oMCY867kndLRXQ2AhscxhoTE5Q9xDdZYwu2hKViZh1JV2oa",
                "gDSvWadXAy2dX9TQya5a5Rj7G7oZSJ3ztsrfKjJMYMFFWwyNA"
            ],
            "slot": {
                "period": 71152,
                "thread": 2
            }
        },
        {
            "creator": "A1f2cgeNKMWtauyAxLy1LMqiVt7ZShgffqc9DbfMSCLpv5xovkP",
            "id": "UXCyVSHg18AraZP9BG6gWRszPyVpasQ6NMc5aBJezYyQibnL1",
            "is_final": false,
            "is_in_blockclique": true,
```


(continues on next page)

(continued from previous page)


```
            "is_stale": false,
            "parents": [
                "2GcJHmGY1QEyWmr4Rh2QSWcxE5icu8szTLJMyZx6fSGZDFETBZ",
                "2A1PFNRpR1MtJYwnp9vF3Pcc8xQ54mpPDjLgjJfZb1yxzTDXjZ",
                "2nL4CBXZKiv4szqvq4cBTtnfUtk5ozRg9Kd45y4UTRNLuHBLHT",
                "2ZuyfDizeBfMgUHgsLgYd7nRMVbk253A1YZUSpjY6bq3med7LY",
                "2Sdvt2oBdYXJ2LSP4AYfJ98DE4mGsBpC2pSLWudYkL92utv9EU",
                "24VPQmcBaFCma9ypn6MMki8FxNQYwcyYnXhhCdAACX1dqFQB94",
                "trM5GjcVp6z4MWrTxyNJGzPQSR8mbNAnaPqLBLUw8vVaTEdNb",
                "AfsZ11V2sCcJnWjD68yrXq1D7gvcv36vByXUAGsuohp93yW7u",
                "eDdqMET8smfSpu93pd8iPsNnnEuhutvH9AqXyXdRRf5GVDK2w",
                "tbszcUiCBqq6ty33wHq52wZ6kdyTo591KyBDfY4FeWaDueKM7",
                "tXTMdL67gYMFiNyugNTBLP9dXMrd5hW4yYG7k4iwZtzhWsD8u",
                "YtQ1UxfuKrgCNYCzbjhDEUbGeRP52j2XizHuK785L7DhHJ1Xr",
                "ybCiSCUPGJBo9FAKE4zus4CG2sSsxFNmoc7qD2Xrn7TjG5TqN",
                "zLPd1vNoYuzHuy3kWg5hbfxKaxSAKk9JGYYd38QyMiVZ6K3BF",
                "2SAHtG3Jn4VHnbzo5bohbLqL7cx7MwQGUr8V6CRaLWsVQetBfu",
                "2N6Wa26nMkx6yuCPwSGm9Wd1EP7u2Ad67et1evuLj4osEWUTYJ",
                "2NVq8igEKnXZ1ysatU9xo66PVGZKx8MwQqRfHmRPg6vjeTy3pP",
                "W5vKLkAyVtG6BNjm4WHC3a7Dz53KDf8v8aKcDjYxRFTXmDd1q",
                "2q3KnZ4tPfEZStQw7LbGCwfehyPckSpWjQc3cuwCWpzSb7XxRr",
                "2akcCnYAFnG9RgVWxprsNgwAv5WDQ8Lm9TfTGb7nrsHn4iiDA2",
                "k3PNeH648id9knfLT2qPpv46AVpZUXaM8qSHgNDx7uw1ieCZL",
                "2YTQhFoBdBo8ofq4ZRJofCQAGgbrQR27CcvgDkdUKMU4H1Fv8t",
                "2XcCTND2GpDSmouGD7ev5JHJNYP5gisgpruMkC5G48d57rBQ5i",
                "4MEYgdhAUUgXLy9CgCMuZTfwTZPLhMbrNm5TFWhDk6EVAoScf",
                "exawL8H3A42zLr5UcCvjMY4TBBn5u84PMtBCrfyyZPWUhayZt",
                "LP8FabDMiAwNkqcDs4z163fsU4jEtRqK9j6sfeXPyNqLJye2L",
                "2gnfhUjLvbRjzu95iTcHSAwF8SenfsCwtLQP3HAEo5Y1NdTUqs",
                "44S9aCFRVd7zDeTBbbTnjHoqy7Up7EpzLVBmARnyfb6HiJENE",
                "uM6w8xGvBxjUYSFzn7DUUV3RUoj8V1iPGGka4ap2g6vCvCqoE",
                "2eejA5Y81RdvDbk2iVFayPvkFpZvm91dNPkq1r1TQMFoaxdwFA",
                "2o1oMCY867kndLRXQ2AhscxhoTE5Q9xDdZYwu2hKViZh1JV2oa",
                "gDSvWadXAy2dX9TQya5a5Rj7G7oZSJ3ztsrfKjJMYMFFWwyNA"
            ],
            "slot": {
                "period": 71152,
                "thread": 1
            }
        },
        {
            "creator": "A1zbiUJjfAjcKg5N2AfMRgHz9Fo4SxhBSNgSv5TrFaDp8t2SfCG",
            "id": "cSuzktjQWxtijFMkBDCzuxnrWv6LgMqcZKoJxE3xhyMgDig6n",
            "is_final": false,
            "is_in_blockclique": true,
            "is_stale": false,
            "parents": [
                "2GcJHmGY1QEyWmr4Rh2QSWcxE5icu8szTLJMyZx6fSGZDFETBZ",
                "2A1PFNRpR1MtJYwnp9vF3Pcc8xQ54mpPDjLgjJfZb1yxzTDXjZ",
                "2nL4CBXZKiv4szqvq4cBTtnfUtk5ozRg9Kd45y4UTRNLuHBLHT",
                "2ZuyfDizeBfMgUHgsLgYd7nRMVbk253A1YZUSpjY6bq3med7LY",
                "2Sdvt2oBdYXJ2LSP4AYfJ98DE4mGsBpC2pSLWudYkL92utv9EU",
```


(continues on next page)

(continued from previous page)


```
                "24VPQmcBaFCma9ypn6MMki8FxNQYwcyYnXhhCdAACX1dqFQB94",
                "trM5GjcVp6z4MWrTxyNJGzPQSR8mbNAnaPqLBLUw8vVaTEdNb",
                "AfsZ11V2sCcJnWjD68yrXq1D7gvcv36vByXUAGsuohp93yW7u",
                "eDdqMET8smfSpu93pd8iPsNnnEuhutvH9AqXyXdRRf5GVDK2w",
                "tbszcUiCBqq6ty33wHq52wZ6kdyTo591KyBDfY4FeWaDueKM7",
                "tXTMdL67gYMFiNyugNTBLP9dXMrd5hW4yYG7k4iwZtzhWsD8u",
                "YtQ1UxfuKrgCNYCzbjhDEUbGeRP52j2XizHuK785L7DhHJ1Xr",
                "ybCiSCUPGJBo9FAKE4zus4CG2sSsxFNmoc7qD2Xrn7TjG5TqN",
                "zLPd1vNoYuzHuy3kWg5hbfxKaxSAKk9JGYYd38QyMiVZ6K3BF",
                "2SAHtG3Jn4VHnbzo5bohbLqL7cx7MwQGUr8V6CRaLWsVQetBfu",
                "2N6Wa26nMkx6yuCPwSGm9Wd1EP7u2Ad67et1evuLj4osEWUTYJ",
                "2NVq8igEKnXZ1ysatU9xo66PVGZKx8MwQqRfHmRPg6vjeTy3pP",
                "W5vKLkAyVtG6BNjm4WHC3a7Dz53KDf8v8aKcDjYxRFTXmDd1q",
                "2q3KnZ4tPfEZStQw7LbGCwfehyPckSpWjQc3cuwCWpzSb7XxRr",
                "2akcCnYAFnG9RgVWxprsNgwAv5WDQ8Lm9TfTGb7nrsHn4iiDA2",
                "k3PNeH648id9knfLT2qPpv46AVpZUXaM8qSHgNDx7uw1ieCZL",
                "2YTQhFoBdBo8ofq4ZRJofCQAGgbrQR27CcvgDkdUKMU4H1Fv8t",
                "2XcCTND2GpDSmouGD7ev5JHJNYP5gisgpruMkC5G48d57rBQ5i",
                "4MEYgdhAUUgXLy9CgCMuZTfwTZPLhMbrNm5TFWhDk6EVAoScf",
                "exawL8H3A42zLr5UcCvjMY4TBBn5u84PMtBCrfyyZPWUhayZt",
                "LP8FabDMiAwNkqcDs4z163fsU4jEtRqK9j6sfeXPyNqLJye2L",
                "2gnfhUjLvbRjzu95iTcHSAwF8SenfsCwtLQP3HAEo5Y1NdTUqs",
                "44S9aCFRVd7zDeTBbbTnjHoqy7Up7EpzLVBmARnyfb6HiJENE",
                "uM6w8xGvBxjUYSFzn7DUUV3RUoj8V1iPGGka4ap2g6vCvCqoE",
                "2eejA5Y81RdvDbk2iVFayPvkFpZvm91dNPkq1r1TQMFoaxdwFA",
                "2o1oMCY867kndLRXQ2AhscxhoTE5Q9xDdZYwu2hKViZh1JV2oa",
                "gDSvWadXAy2dX9TQya5a5Rj7G7oZSJ3ztsrfKjJMYMFFWwyNA"
            ],
            "slot": {
                "period": 71152,
                "thread": 3
            }
        }
    ],
    "id": 1
}
```


get_block

Get information about block given its hash.

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_block",
    "params": ["D6kTS4Wh3B7FRDCw6ncqrCuY7NVPYBbGwdSg814Kd13fS7xQa"]
}'
```

- Result:


```
{
    "jsonrpc": "2.0",
    "result": {
        "content": {
            "block": {
                "header": {
                    "content": {
                        "endorsements": [
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 0,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A12N9nUN9r1eUheMZ36AA3RTDYepLtEMpHZoBvzQmxw4hNcJV7tH",
                                "creator_public_key":
↪"P12qBafeiXMypqiChy7KEjqgAaUzbWJHhJALjfxzzY5hEH5BwL2c",
                                "id": "2jtHfATDrho9Ttkxz3xp26WwjjREPVQV16fwMUCGyjnEQoyU8p
↪",
                                "signature":
↪"XLJd5dSZsaQ3UYuuSGBGCbVsEM3aGTxAGigT81bVto7CypivDDwoPb6kJWXKzhvRi14qh3ReFqa7zzf3r5hYf343nqceH
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 1,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A1RMafAnGhMHoVzvtBBP1u6PTCoMRBpQSQxxJb4e6ySDS6BpHxE",
                                "creator_public_key":
↪"P1GJCRP8UYmkt1ZUYScjuGcXXLmDzq1ijJmYtqKpkgKPBtazRGo",
                                "id": "qELXLSgYd7aRBqgASfm5u3k4QMBSYBuQK79oMmf9Yohtr71ZH
↪",
                                "signature":
↪"8kPzmEiku3FNbYgHVeY6cY14cQskDqBS2trH2z8NLyPaU8xauXa7dFMKKKpb88b1eEx1QsSLmTx7iHXrCKYgKm6vjz8EC
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
```


(continues on next page)

(continued from previous page)


```
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 2,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A123ingVJVrQkHveBCoXCUePWnkYjJ5mJE1gHiEqu1zpqvXJuBSK",
                                "creator_public_key":
↪"P1tNLmbgiqNjYfA7Xy6QNCVMEPkDtUqHuw9DuVcWf4FoYimJPwb",
                                "id": "2TbWNGBkPyXHGqVeTQMJrrt2E2858FtiCcBZoySQ5rXVToYDT
↪",
                                "signature":
↪"XYj5LByWXoi2EXsJh4MVEAzrGy9evcwHFywh4cYFc9S6xEUoyg4wnCqUcy2GA9K4SxK4H4AZyAoE5u4H6dgv1h5Gk8R3H
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 3,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A1bVpBkQo3nt8tKFcCojbB7Nt179finvmm9TJJ7JWqrX5a2cHZM",
                                "creator_public_key":
↪"P1Jk6uzT4iryr1Q8ACqnKoQjPNKraQnwk972TpPkvpj4JTs5MGS",
                                "id": "G6bLTh1BWzywrF9tEyScJyVLDxuk3n3abePWxrpcbqh9QQ1ah
↪",
                                "signature":
↪"JScDT2tpLD8RULoPHCU2HMyxGxaxFExCVYjivbL4cbsyNy5J53pDWHne911eug1UMZFJr3s5y1t6NYy5Mf8zkRCg1JUjY
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 4,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A1HRFkU7Mhww3BckCuvaXizvGCVNWG6ZiERLfUhRytCQibwWBgV",
                                "creator_public_key":
↪"P1dv7uM55mh39PUrmuYbE7uWi66EUuRGQCRyPh43DPgbgT5DSpJ",
                                "id": "Utxvrhw5X6rh5JPRE9LEExY1EMYWmoXYoPCApMb6ZTbd1rL73
```


(continues on next page)

(continued from previous page)


```
↪",
                                "signature":
↪"Mg7ZP3SJgZ97eGEoeuMHxvhv3FGREYqjANHupbjRb1qcDaEfih9xnA5zB5SZfiCRVpFZYXAsbET4GKi7Ne8uNGeS9AHsq
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 5,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A1VRf7guK6CrNkCz8PLAwtG18zrjZTd8PEnWXfjS6RmLLXvNjtu",
                                "creator_public_key":
↪"P1YudM7ga88ArqVmFipS6Qs36apViTde8MrdfUxmPcT8mEJ6vXh",
                                "id": "NrCTcEdKB6CWrJnkBarPaseYUNx7uisq73u5PvuDaU7MnyUY5
↪",
                                "signature":
↪"6CpqaEgzv59QakG8xf1QYwQgaSJxudk1GEGyRdBxFBpG5F3756hULFrZWFmdz66RcWtpT7TZ5CzPPACGLZxLCJow79L4W
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 6,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A12RHPuU7JFS2rxvxL6MnzVoBJAZr7ivFFJuiRPv4mi5wv8z8VYm",
                                "creator_public_key":
↪"P1LUAvxdts32qQHnCPMqvf3F8WsScshVY1og7d16x8SDvWtCT2Z",
                                "id": "Rph1xFyRnarENnrm6ZzS8XvDfDPwVf7XVWm2CTZBDSfew3uaY
↪",
                                "signature":
↪"YyKkN79gvpiEo6zQFbSgZv84sB34EvQ5LXgNp2MVoZAAbSKstGTopJ6t6fzzeDRFNVjbyqc7ZnDbLPq8PZ9WDo1yoepqo
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 7,
                                    "slot": {
                                        "period": 72180,
```


(continues on next page)

(continued from previous page)


```
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A1u9fCMusV2rp3m7uoi2e5EuKNGpgfw9nFxcgH7MxPzrd9nB8Mj",
                                "creator_public_key":
↪"P12ZVaa7sNWPuMyTC1ijJYHr1NuF2DvZotRxrjZCoHmHJnzk3cUp",
                                "id": "XxYECs1HHJMqwavdoXx9WDEMGWxgM1ainm2VxtT2Fvan8yEJC
↪",
                                "signature":
↪"Cg6Ajsp1QrSeLfeTFb8vyHZD63hWzMRyRQRT14MYDFthAfYcocjh4aWTqv8zQCyr2SXBHapiBeaYY6NXRJtzPqukULFQi
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 8,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A1jXja9XVuepgpq94XzfHM6t1GjGJusRVuqA65ADdkupGXkrfCR",
                                "creator_public_key":
↪"P13wkLrigKC2R8LbUpafsBAA3H6GwnL9DbawDe8Q8uMGqdeeqgv",
                                "id": "2izC3L1eT9RzANsoowY4SDYbvRH14HFPGMRL3Y2sGP1EjiXR9H
↪",
                                "signature":
↪"73nbrBKBpyW1unPhioCRwyA6ebqT2MZCU6LJazxLMX4qiKzy9mciwHnZLAnjrMCN6AReqaYE8E4TFQLRSgiYZ2KpqmCUW
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 9,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A121RTpsgvPJtxj9FnSvVJUqMahintXyaATdBcfRdtva9xBhvLmR",
                                "creator_public_key":
↪"P12qaehFSeCjQu5dqxeYDW8fuz3MieQXDPuCNmL1BpaswezhnDNp",
                                "id": "hESY57Jhd2JhJfwK4yfFNhMBoDPQp1uhNppCCjB9nbFeeSair
↪",
                                "signature":
↪"H1st549STBKmehRtqwFTnHVxNX3UrPzfPJU7fhNP3Q1JvpoCjzLEewKRzb7YV6u9oKXEEaPdWXmj5bDGXWJK1mLfAWp8w
↪"
```


(continues on next page)

(continued from previous page)


```
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 10,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A12VJh2HhXBTxDHDr9cwayHQTbVbkbu3soQqsekckH3gXLWvxkZ2",
                                "creator_public_key":
↪"P1ss4j58UtMbVjtP3pKK76Q7mfC3ArEsLr35e9UuCALHLjbsv58",
                                "id": "2SPHdLgWHYaGbZfV6ZUKJuJVLrWSGXauHHjRJS2tdaS382g4uQ
↪",
                                "signature":
↪"Kyvz5rJf5x8jVkHxqeqLzsdHgeYWqSYKwmgLXotX8readhkj2Hvbrzkyiwfu96atwRjnm2wsRQRzM6R6AwrrL5nh7y2qa
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 11,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A12fCAsSsLnm6BMkmtq57YJtuPpLPb35H7Q9LoJLBgRptxWsFwnm",
                                "creator_public_key":
↪"P12DHUNJiWTYzU2hWV4CJH5KT99A4jrEQQte3gkKtnLNj9oWd78w",
                                "id": "2LXXatLWkH8M1adBeGKwfH4GM7xG7JGYXpxiJYo89Jy1SQ9Cg
↪",
                                "signature":
↪"QqCYp9oAUjuYppHmLVxyik7A1JqFvYdPpVNXpunzRqHc6QHD56Kundv2vcGaFViozQHGRmyAbjo1JLcp7npTiZmSmT6UY
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 12,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
```


(continues on next page)

(continued from previous page)

```
↪"A129Eya4XLQ2nuDJjhFqrEsgdH27g89yVFedR1H2CiDy129Bxn8",
                                "creator_public_key":
↪"P1xAszTFsXawBtUoJE4hvKrEFYG7DDpZvdkHjvtTH332EF2PwkT",
                                "id": "YoNAoDbYmFQE2X2G2TPxr2J3UTFrdTYLXEcYgVSLDtoi8iUBF
↪",
                                "signature":
↪"Ja4Bo1ymdbX7FfgAPxfLBhbGG6KVECvD7GoAxEYqXUsX5y4K4JerNvQS1jBFMdHxNQDYfgG4E8xdm354tegEWujFzuCnL
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 13,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A13CUpBmzTC53uud6XccjPuLLTWVn6A6isfuWrmG4JyUzJocdYY",
                                "creator_public_key":
↪"P14kHa3rmmFzeH3CeaUaYxvyrE9NpDJXzGrNso42j6wNBRJ8RVL",
                                "id": "U8weB8dyRFKhFwqwyU2q1BqSkHhFedUB1gJxNS63svqMCah71
↪",
                                "signature":
↪"A8LDSGq7wJffHrqDaQhWePvPKRU1PGMhBFw9TeqfaK8PGxV7u9tSjWiiWbUjAaWHWWdwF5Cp7htx8MZ8ZKeziLkNizckZ
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 14,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A15dQSTXEj9heazPXuWrqb7YzvK6DkLbafE4iHziSybwyFEw2Pc",
                                "creator_public_key":
↪"P1Hg9TwLttUaJXVMByr88G9YHsJ7yEtonpXgYBwxezzwhmKfSWk",
                                "id": "SpZHoVJUekZGAqtj2t5jEoKMHs5mi6oHVzYn1nm4GgoCd53cb
↪",
                                "signature":
↪"RfwXPp9cHvKpXGyN5M9BQPAgfLqUxE6EbaqQcve3xHWsTQ5b6GiVNYkQjEhEWdLEDB7Z8qQt4TRAK6Rp7aABoWFQ6PF5c
↪"
                            },
                            {
                                "content": {
                                    "endorsed_block":
```

(continues on next page)

(continued from previous page)


```
↪"AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV",
                                    "index": 15,
                                    "slot": {
                                        "period": 72180,
                                        "thread": 31
                                    }
                                },
                                "creator_address":
↪"A12Em4NP9afTfCQkopdF8WsPkU8PazVGf9J4kzLwBgyc187q6L5d",
                                "creator_public_key":
↪"P12683Xab7njz4K991Vvx39yBaMmpgJuB3A6d7U8uheYtYraw89X",
                                "id": "gCeNaZy2ihWV7XfLwnNNsQ8G1m5sMzhMBd8aP8q4JFHUCqV9S
↪",
                                "signature":
↪"D6A5HYT4FavULrTLWm4CibbWPNNBMtmXBi4XXWF3HDV8cMngs9DNy8PriPJasRodvTGpgEFyP1gowrXDaFU2PcRkB5mit
↪"
                            }
                        ],
                        "operation_merkle_root":
↪"27m2HEnhXU78gob1PUrXqpW8wek49enJGEf4SDo4f2RQ1j3fKD",
                        "parents": [
                            "voH6upJppWUeyZhcZMprzKhyVi5iHhXXpY3UasoUpAQaSi4xX",
                            "PcbnbucTNdMcfUqjGWWwJG5eLMJwvjdx2XTVGDAqGnh9zkYPn",
                            "FBKV2AU9iBnBLpsykQnqADVosCC417o1AdGZzbTCLPe6ENEZS",
                            "v8FBFQ7wshibyME8bTkJbq3HQwqpDvkidB43go2fY1wCrxFGg",
                            "2fc9VWfnH8f883Pf4wouDMuvtpxR7F2fzsf9cbq4A13UXNphdE",
                            "6zvFwHh373wJAKZs3oMRYdw2KwEqsjDQNoqbDjN89UUUEMe5M",
                            "Cpcjt6FcebQwwgS5RVdGKwpCbndVxZJrrFqhA82SFKCZRDEuB",
                            "AcWddtC2cqu9d2pSzyCVea84TZY5bP2bs8CKU8YYjw8vnexzM",
                            "2NehYFSDhAf3cZWb7fDXAWnYYD11uaYYZbuqUX13CBoVchAzGr",
                            "euSAdXo5QudXdFzwqpU67eCUF9b6VM8i9Qy5UKJFFC6EiLiJ4",
                            "DMh3dNWRNsVrV6P5SY9p5RCpvYpjaGdFWUu23HFL6TfKwy63n",
                            "pNKpTb8vh8eZ7YtB11psDH6TX6w9SsLdAwAV5oG6XR4P1Mdmz",
                            "2wenb1UzqRiHV5tixjMMqVkrwb8ywm7HT2UBP9dDZET7mnm3bi",
                            "pwhBtRuLNWP19hKJ5kvKdQjLjnjsJkeMqwMHmHt6ZhdNcYjuf",
                            "En16Wb744Rn2trfkeQREG3HSCju39xu9dK6EXiVhMaQ7QNTQc",
                            "Pn2yvpsFq6YYfH26RqXye9S25hbXgy8gvPQPFnrdVxxcqrZPv",
                            "2AamJsrQ68r3r9bhgsiAKkH7JGUSDLPCZUh91DxeHTBQWBRjQu",
                            "2LSHyysq3PRDhfYF68haUWHUW7oAwZXhzqewLJhAmVRwH6JjVh",
                            "jWvGUobun7mzuaJye5nYEFvSGrmyW4sjrmEZ3mvTAo7iQtmxQ",
                            "JsfRiXaoZY4tDqJWyEEpJkjchrcrGpdi2im77KW9bcGjX8FrW",
                            "FCzk7rYB8ZAqkkSchdxPeZQohjNU4Wwi2TdvNJe1Df14LnxW4",
                            "2HrJiz9fBDVUjVTF6aA3y8bZQcHVpnJygRkt8EtnkCB3HT9dxM",
                            "242ghWYwu8PKZSQoPoij2S8CE2u46zfcGdC42mNN12HFrj55aQ",
                            "2UCUqS26mbdbsFZkL1rNrrQu2SMZQnUZdTJ7tjo7QVjVVFvfxR",
                            "2Ch6hGDfEeUBamUrWojzrTPeswTuGVfwvjfDEKChNHU96A3PXo",
                            "pyK7qz9ebZ9bModjmuvvUoVmvBrLHUsUy8uqWRKemTh2zLFyd",
                            "2SJhPnDC8nV4SkaxTai4Gvxpzv14DeD164XHqKfSbV5byTjfqq",
                            "jsZEsz4U3jazbpLfXvK6BVRKyE4F3Sh6bS26AN9i7vM274XQc",
                            "2K75STq46JPJ2eUzZrepRNDPrfg6NgKC9cZFYFVXfKpJVBQAP8",
                            "3QYdeQrNsbBXpEUbqBY1v1UPWiwnvgSjL2mcG1fzkRC4Mso5J",
                            "V58LdmjJLvQLRVm4bqqiMn2ChGprgLDxwNm4gKJqz3UXFwonK",
```


(continues on next page)

(continued from previous page)


```
                            "AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV"
                        ],
                        "slot": {
                            "period": 72180,
                            "thread": 31
                        }
                    },
                    "creator_address":
↪"A12Lh4yCTdF5P9Unt3Aq4aimUFafTiuGoYYHfqU2P68RkY35tFmQ",
                    "creator_public_key":
↪"P12uRiEAgEGHPpNZ7fMFsh3XPYS6A9GB9PgDAQdwFbxCH4nxSQEF",
                    "id": "Fb46NHJCFTVgddSEZMEcmYeYpokvQv8gCYjnDBpXbAQBKpVE3",
                    "signature":
↪"KMcsEkC6CEDE55tmvAfR47vhH1PvAd52SZWUyQ9kdobvkvEGMZS2TrDYHWP6jCod1QeQpaRY1BXM2FMSWn5EuCS5HcVti
↪"
                },
                "operations": [
                    "177bzpUmukLarBiRGcTCDE63xqc5nkAKUja414HDmsNS2T3Gy",
                    "1G9oj5pX7Ruj1w6cT9DXY2KMNKM9JgHVfA6JN5nhZchnHCm6s",
                    ...
                    "2wX8fv5WaPpEXFYPbzmyeShVMRNdcqGCxnUCdfXKW2LrVvpkM1",
                    "2wdwXBXxpBeQuPtT5kkEiV3AKsTqWkzN9QLHXskDsZgD8XpdTN"
                ]
            },
            "is_candidate": true,
            "is_discarded": false,
            "is_final": false,
            "is_in_blockclique": true
        },
        "id": "Fb46NHJCFTVgddSEZMEcmYeYpokvQv8gCYjnDBpXbAQBKpVE3"
    },
    "id": 1
}
```


get_operations

Get information about operation (s) information associated to a given operation(s) ID(s).

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_operations",
    "params": [["177bzpUmukLarBiRGcTCDE63xqc5nkAKUja414HDmsNS2T3Gy"]]
}'
```

- Result:

```
{
    "jsonrpc": "2.0",
    "result": [
        {
            "id": "177bzpUmukLarBiRGcTCDE63xqc5nkAKUja414HDmsNS2T3Gy",
            "in_blocks": [
                "Fb46NHJCFTVgddSEZMEcmYeYpokvQv8gCYjnDBpXbAQBKpVE3"
            ],
            "in_pool": true,
            "is_final": true,
            "operation": {
                "content": {
                    "expire_period": 72188,
                    "fee": "0",
                    "op": {
                        "Transaction": {
                            "amount": "0.00040048",
                            "recipient_address":
↪"A1Czd9sRp3mt2KU9QBEEZPsYxRq9TisMs1KnV4JYCe7Z4AAVinq"
                        }
                    }
                },
                "creator_address": "A12teNrVETiAfCHHNrDwcxLFZ2WUhtKk1suym6nLPBFcaxjP188w
↪",
                "creator_public_key":
↪"P1cjQAvB8b2RxpqxVCn54KDjYDmC1wer6tJofohBCToKHWsgoVB",
                "id": "177bzpUmukLarBiRGcTCDE63xqc5nkAKUja414HDmsNS2T3Gy",
                "signature":
↪"MnDMrajkMDzRJxiRyWgZCoyTP4k4yWM3raY4vo4SJ8o3CnBFrBnfc15C35xiemJ1zQqtYzYssWN5hWytGDVCsjuT2dt3p
↪"
            }
        }
    ],
    "id": 1
}
```


get_endorsements

Get information about endorsement (s) (content, finality ...)

- Query:

```
curl --location --request POST 'https://test.massa.net/api/v2' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "get_endorsements",
    "params": [["2jtHfATDrho9Ttkxz3xp26WwjjREPVQV16fwMUCGyjnEQoyU8p"]]
}'
```

- Result:

```
{
    "jsonrpc": "2.0",
    "result": [
        {
            "endorsement": {
                "content": {
                    "endorsed_block": "AvvHCpxnX8U6uTQKmpze55vzhkhdbbst9rrhPwbykymjJyzoV
↪",
                    "index": 0,
                    "slot": {
                        "period": 72180,
                        "thread": 31
                    }
                },
                "creator_address": "A12N9nUN9r1eUheMZ36AA3RTDYepLtEMpHZoBvzQmxw4hNcJV7tH
↪",
                "creator_public_key":
↪"P12qBafeiXMypqiChy7KEjqgAaUzbWJHhJALjfxzzY5hEH5BwL2c",
                "id": "2jtHfATDrho9Ttkxz3xp26WwjjREPVQV16fwMUCGyjnEQoyU8p",
                "signature":
↪"XLJd5dSZsaQ3UYuuSGBGCbVsEM3aGTxAGigT81bVto7CypivDDwoPb6kJWXKzhvRi14qh3ReFqa7zzf3r5hYf343nqceH
↪"
            },
            "id": "2jtHfATDrho9Ttkxz3xp26WwjjREPVQV16fwMUCGyjnEQoyU8p",
            "in_blocks": [
                "Fb46NHJCFTVgddSEZMEcmYeYpokvQv8gCYjnDBpXbAQBKpVE3"
            ],
            "in_pool": true,
            "is_final": false
        }
    ],
    "id": 1
}
```


7.23 Glossary

7.23.1 Where is the money?

- **wallet**: A set of keypairs (with their associated address).
- **address**: The hash of a public key preceded by an indentifier and a version bit. Tokens, rolls, bytecode and datastore are associated with an address. An address is associated with a thread for sharding purposes.
- **smart-contract address**: A specific kind of address that was generated by bytecode execution, that is not associated with a keypair. Hence, it is immutable. Note: a deletion mechanism can be implemented in the associated bytecode.
- **final balance**: Balance at latest final blocks.
- **candidate balance**: Balance at latest blocks.
- **deferred credits**: Coins to be reimbursed upon a given slot. Produced by a roll sell or a roll slash.

- **ledger vs balance**: The ledger is the long-term memory of the blockchain. For every address it may contain a balance, bytecode and a datastore (key / value container).
- **staking address**: An address that has at least one roll.

7.23.2 Graph stuff

- **compatibility graph**: A graph of all compatible blocks (see https://arxiv.org/pdf/1803.09029.pdf at III. ARCHITECTURE, D. Consensus Rule, 1. Compatibility Graph). Theoretical construction. Never computed by the Massa node.
- **head incompatibility graph**: A graph of incompatible active blocks (see https://arxiv.org/pdf/1803.09029.pdf). Computed by the Massa node. Simpler to compute than the compatibility graph, practical application of it.
- **fitness**: For a block: 1 + number of endorsements included in the block. For a clique: sum of its blocks' fitness.
- **final**: A block becomes final if its fitness summed to the fitnesses of all the following blocks is above a defined threshold.
- **stale**: A block is stale when it is only contained in cliques that have been abandoned.
- **blockclique**: The clique of higher fitness.

7.23.3 Consensus and block production

- **proof of stake**: A *type of consensus mechanism <https://en.wikipedia.org/wiki/Proof_of_stake>* in which power in the network is directly related to the amount of tokens one possesses.
- **thread**: Addresses are sharded across threads.
- **period**: On a thread, a block is produced every period.
- **cycle**: Config-defined number of periods. Unit of time and associated information used by the proof of stake algorithm to make the upcomming selections for block and endorsement creation. Lasts for about 30 minutes.
- **slot**: Point in time defined by a period and a thread, at which a block is expected to be created. Can be empty if the selected creator missed the block opportunity.
- **slashing**: Process in which the rolls of an address are remove and set for future reimbursement at the end of a cycle. Happens when an address block production is too low compared to its block opportunities.
- **staker**: Owner of at least one roll. They are expected to create blocks and endorsements when selected.
- **roll**: Akin to a lottery ticket to be selected for block and endorsement creation. A roll costs 100 MAS. The more rolls you have, the more chances you have to be selected to produce a block.
- **endorsement**: Part of a block header, used to improve security. Any staker can be selected to create an endorsement. They are produced automatically by any node with at least one staking address. An endorsement carries the hash of the last block in the given thread. A block creator must include enough of them in order for the block to have a good fitness and be included in the blockclique.
- **block**: A block is produced by a staker. They are produced automatically by any node with at least a staking address. It includes operation and is checked by other stakers.

7.23.4 How to interact with the blockchain

- **operation**: The only way to inject information in the blockchain. It is produced by an emitter that will provide a fee. The operation will be valid (ie includable in a block) only for a limited amount of time. Can be a simple transaction, a roll buy, a roll sell or a smart contract operation. The latter will execute bytecode on the blockchain.
- **transaction**: Coin transfer between a sender (that created and signed this operation) and a receiver.

7.23.5 Miscellaneous

- **peer VS node**: 1 peer <=> 1 ip address, whereas 1 node => 1 public key. A peer can run multiple nodes.
- **node address**: Address used to identify a node. Not related to a staking address.

7.24 Contributing

Thank you for your interest in contributing to Massa! There are many ways to contribute and we appreciate all of them.

7.24.1 Asking Questions

The best place to ask general questions about Massa is on the Massa discord.

It might also be helpful to review existing and previous Github issues to see if your question has been answered already.

For specific technical questions, submitting an issue in the relevant repo may be the best place to ask.

7.24.2 Giving Feedback

We greatly appreciate your feedback. We love to hear about things you like, as well as about things that are not working, or could be improved! The Massa discord is a great place for submitting feedback on Massa.

7.24.3 Solve issues

We welcome contributions from community members. If you'd like to work on some issue, don't hesitate to ask questions and we'll try to help you as soon as possible.

Issues tagged as *good first issue* are a great place to start. If you are unsure with how and where to start, feel free to reach out to us on the Massa discord.

7.24.4 Contributing to documentation localization

One way to contribute to Massa is by contributing to the localization effort. We aim to have Massa accessible to as many people as possible accross the globe. We believe that localization of the documentation helps making the blockchain technology understandable and available to all. If you'd like help us in this endeavor and work on localization, everything is detailed in this repo.

7.24.5 Funding Opportunties

Massa is offering grants to people and teams that want to build on Massa. You can find more information about the grants program here.

7.24.6 Full-time Contributors

Massa is hiring a wide variety of software engineers, so if you would like to work on Massa full-time, please consider applying for an open position.

- genindex
- modindex
- search

Keyword Index, Search Page

INDEX