Raiden Specification Documentation

Release 0.1

Brainbot

Dec 13, 2018

Contents:

1 Raiden Messages Specification **3**
- 1.1 Overview . . . 3
- 1.2 Data Structures . . . 3
- 1.3 Messages . . . 5
- 1.4 Specification . . . 7

2 Raiden Transport **11**
- 2.1 Requirements . . . 11
- 2.2 Proposed Solution: Federation of Matrix Homeservers . . . 11
- 2.3 Use in Raiden . . . 12

3 Raiden Network Smart Contracts Specification **15**
- 3.1 Overview . . . 15
- 3.2 General Requirements . . . 15
- 3.3 Project Requirements . . . 16
- 3.4 Data structures . . . 16
- 3.5 Smart Contract Functional Decomposition . . . 20
- 3.6 TokenNetwork Channel Protocol Overview . . . 29
- 3.7 Protocol Values and Settlement Algorithm Analysis . . . 35

4 Raiden Network Monitoring Service **41**
- 4.1 Basic requirements for the MS . . . 41
- 4.2 Usual scenario . . . 41
- 4.3 Economic incentives . . . 42
- 4.4 Appendix A: Interfaces . . . 43
- 4.5 Appendix B: Message format . . . 44

5 Raiden Pathfinding Service Specification **45**
- 5.1 Overview . . . 45
- 5.2 Assumptions . . . 45
- 5.3 High-Level-Description . . . 45
- 5.4 Public Interface . . . 46
- 5.5 Implementation notes . . . 51
- 5.6 Future Work . . . 51

6 Raiden Mobile Wallet Specification **53**
- 6.1 Overview . . . 53

6.2 Goal . . . 53
6.3 Terminology . . . 53
6.4 Requirements . . . 53
6.5 Depends on . . . 54
6.6 High Level Features . . . 54
6.7 Platforms & Languages . . . 54
6.8 Components . . . 55
6.9 MobileApp . . . 56
6.10 Protocols . . . 59
6.11 Roadmap . . . 60
6.12 Issues to clarify on . . . 61
6.13 Other wallets: . . . 61

7 Raiden Terminology 63

This is a tentative specification for the Raiden Network. It's under development and will be further refined in subsequent iterations.

CHAPTER 1

Raiden Messages Specification

1.1 Overview

This is the specification document for the messages used in the Raiden protocol.

1.2 Data Structures

1.2.1 Offchain Balance Proof

Data required by the smart contracts to update the payment channel end of the participant that signed the balance proof. Messages into smart contracts contain a shorter form called *Onchain Balance Proof*.

The signature must be valid and is defined as:

```
ecdsa_recoverable(privkey, keccak256(balance_hash || nonce || additional_hash ||␣
↪channel_identifier || token_network_address || chain_id))
```

Also see *additional_hash*.

Fields

Field Name	Field Type	Description
nonce	uint256	Strictly monotonic value used to order transfers. The nonce starts at 1
transferred_amount	uint256	Total transferred amount in the history of the channel (monotonic value)
locked_amount	uint256	Current locked amount
locksroot	bytes32	Root of the merkle tree of lock hashes (see below)
token_network_identifier	address	Address of the TokenNetwork contract
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
message_hash	bytes32	Hash of the message
signature	bytes	Elliptic Curve 256k1 signature on the above data
chain_id	uint256	Chain identifier as defined in EIP155

1.2.2 Merkle Tree

A binary tree composed of the hash of the locks. For each payment channel, each participant keeps track of two Merkle trees: one for the hashlocks that have been sent out to the partner, and the other for the hashlocks that have been received from the partner. Conceptually, each direction of a payment channel has one Merkle tree, and each participant has a copy of it. Only the sender of the hashlocks can change the Merkle tree of this direction.

The root of the tree is the value used in the *balance proof*. The tree is changed by the `LockedTransfer`, `RemoveExpiredLock` and `Unlock` message types. The sender of these messages applies the change to its copy of the tree and computes the root hash of the new tree. The receiver applies the same change to its copy of the tree and checks the root hash of the new tree against the root hash in the messages.

1.2.3 HashTimeLock

Invariants

- Expiration must be larger than the current block number and smaller than the channel's settlement period.

Hash

- `keccak256(expiration || amount || secrethash)`

Fields

Field Name	Field Type	Description
expiration	uint256	Block number until which transfer can be settled
locked_amount	uint256	amount of tokens held by the lock
secrethash	bytes32	keccak256 hash of the secret

1.3 Messages

1.3.1 Locked Transfer

Cancellable and expirable *transfer*. Sent by a node when a transfer is being initiated, this message adds a new lock to the corresponding merkle tree of the sending participant node.

Invariants

Only valid if all the following hold:

- There is a channel which matches the given *chain id*, *token network* address, and *channel identifier*.
- The corresponding channel is in the open state.
- The *nonce* is increased by `1` in respect to the previous *balance proof*
- The *locksroot* must change, the new value must be equal to the root of a new tree, which has all the previous locks plus the lock provided in the message.
- The *locked amount* must increase, the new value is equal to the old value plus the lock's amount.
- The lock's amount must be smaller then the participant's *capacity*.
- The lock expiration must be greater than the current block number.
- The *transferred amount* must not change.

Fields

Field Name	Field Type	Description
lock	HashTimeLock	The lock for this locked transfer
balance_proof	OffchainBalanceProof	Balance proof for this transfer
initiator	address	Initiator of the transfer and person who knows the secret
target	address	Final target for this transfer

1.3.2 Secret Request

Message used to request the *secret* that unlocks a lock. Sent by the payment *target* to the *initiator* once a *locked transfer* is received.

Invariants

- The *initiator* must check that the payment *target* received a valid payment.

Fields

This should match the encoding implementation.

Field Name	Field Type	Description
cmdid	one byte	Value 3 (indicating `Secret Request`)
pad	three bytes	Ignored
message identifier	uint64	An ID used in `Delivered` and `Processed` acknowledgments
payment_identifier	uint64	An identifier for the payment chosen by the initiator
lock_secrethash	bytes32	Specifies which lock is being unlocked
payment_amount	uint256	The amount received by the node once secret is revealed
expiration	uint256	See *HashTimeLock*
signature	bytes	Elliptic Curve 256k1 signature

1.3.3 Secret Reveal

Message used by the nodes to inform others that the *secret* is known. Used to request an updated *balance proof* with the *transferred amount* increased and the lock removed.

Fields

Field Name	Field Type	Description
lock_secret	bytes32	The secret that unlocks the lock
signature	bytes	Elliptic Curve 256k1 signature

1.3.4 Unlock

Note: At the current (15/02/2018) Raiden implementation as of commit `cccfa572298aac8b14897ee9677e88b2b55c9a29` this message is known in the codebase as `Secret`.

Non cancellable, Non expirable.

Invariants

- The *balance proof* merkle tree must have the corresponding lock removed (and only this lock).
- This message is only sent after the corresponding partner has sent a *Secret Reveal message*.
- The *nonce* is increased by `1` with respect to the previous *balance proof*
- The *locked amount* must decrease and the *transferred amount* must increase by the amount held in the unlocked lock.

Fields

Field Name	Field Type	Description
balance_proof	OffchainBalanceProof	Balance proof to update
lock_secret	bytes32	The secret that unlocked the lock
signature	bytes	Elliptic Curve 256k1 signature

1.4 Specification

The encoding used by the transport layer is independent of this specification, as long as the signatures using the data are encoded in the EVM big endian format.

1.4.1 Transfers

The protocol supports mediated transfers. A *Mediated transfer* may be cancelled and can expire unless the initiator reveals the secret.

A mediated transfer is done in two stages, possibly on a series of channels:

- Reserve token *capacity* for a given payment, using a *locked transfer message*.
- Use the reserved token amount to complete payments, using the *unlock message*

1.4.2 Message Flow

Nodes may use mediated transfers to send payments.

Mediated Transfer

A *Mediated Transfer* is a hash-time-locked transfer. Currently raiden supports only one type of lock. The lock has an amount that is being transferred, a *secrethash* used to verify the secret that unlocks it, and a *lock expiration* to determine its validity.

Mediated transfers have an *initiator* and a *target* and a number of mediators in between. The number of mediators can also be zero as these transfers can also be sent to a direct partner. Assuming `N` number of mediators, a mediated transfer will require `10N + 16` messages to complete. These are:

- `N + 1` mediated or refund messages
- `1` secret request
- `N + 2` secret reveal
- `N + 1` unlock
- `2N + 3` processed (one for everything above)
- `5N + 8` delivered

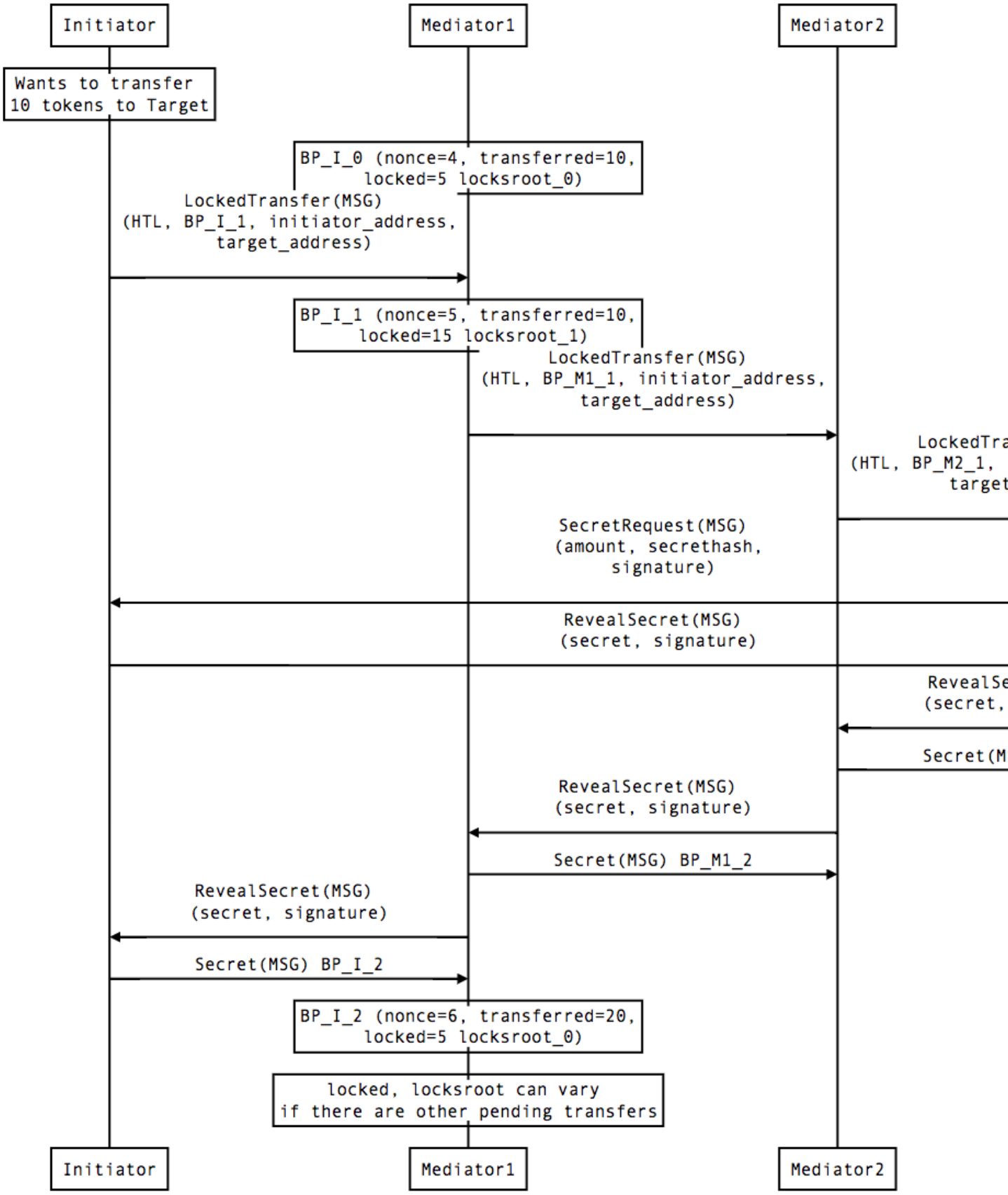
For the simplest Alice - Bob example:

- Alice wants to transfer `n` tokens to Bob.
- **Alice creates a new transfer with:**
  - transferred_amount = `current_value`
  - lock = `Lock(n, hash(secret), expiration)`
  - locked_amount = `updated value containing the lock amount`
  - locksroot = `updated value containing the lock`
  - nonce = `current_value + 1`
- Alice signs the transfer and sends it to Bob.
- Bob requests the secret that can be used for withdrawing the transfer by sending a `SecretRequest` message.

- Alice sends the `RevealSecret` to Bob and at this point she must assume the transfer is complete.
- Bob receives the secret and at this point has effectively secured the transfer of `n` tokens to his side.
- Bob sends a `RevealSecret` message back to Alice to inform her that the secret is known and acts as a request for off-chain synchronization.
- Finally Alice sends an `Unlock` message to Bob. This acts also as a synchronization message informing Bob that the lock will be removed from the merkle tree and that the transferred_amount and locksroot values are updated.

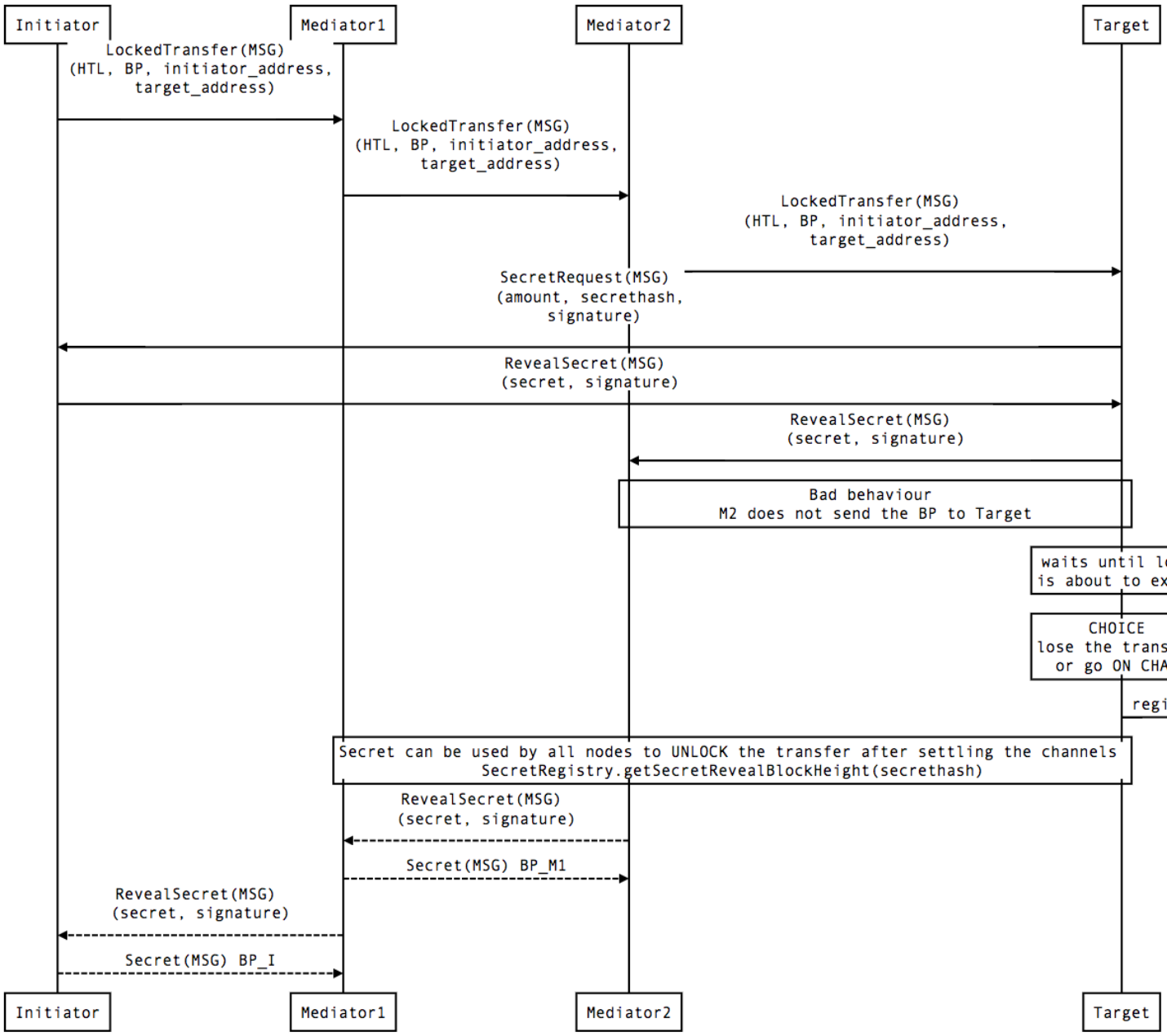
Mediated Transfer - Best Case Scenario

In the best case scenario, all Raiden nodes are online and send the final balance proofs off-chain.


Initiator
Mediator1
Mediator2
Wants to transfer
10 tokens to Target
BP_I_0 (nonce=4, transferred=10,
locked=5 locksroot_0)
LockedTransfer(MSG)
(HTL, BP_I_1, initiator_address,
target_address)
BP_I_1 (nonce=5, transferred=10,
locked=15 locksroot_1)
LockedTransfer(MSG)
(HTL, BP_M1_1, initiator_address,
target_address)
LockedTra
(HTL, BP_M2_1,
target
SecretRequest(MSG)
(amount, secrethash,
signature)
RevealSecret(MSG)
(secret, signature)
RevealSe
(secret,
Secret(M
RevealSecret(MSG)
(secret, signature)
Secret(MSG) BP_M1_2
RevealSecret(MSG)
(secret, signature)
Secret(MSG) BP_I_2
BP_I_2 (nonce=6, transferred=20,
locked=5 locksroot_0)
locked, locksroot can vary
if there are other pending transfers
Initiator
Mediator1
Mediator2

Mediated Transfer - Worst Case Scenario

In case a Raiden node goes offline or does not send the final balance proof to its payee, then the payee can register the `secret` on-chain, in the `SecretRegistry` smart contract before the `secret` expires. This can be used to `unlock` the lock on-chain after the channel is settled.



Limit to number of simultaneously pending transfers

The number of simultaneously pending transfers per channel is limited. The client will not initiate, mediate or accept a further pending transfer if the limit is reached. This is to avoid the risk of not being able to unlock the transfers, as the gas cost for this operation grows with the size of the Merkle tree and thus the number of pending transfers.

The limit is currently set to 160. It is a rounded value that ensures the gas cost of unlocking will be less than 40% of Ethereum's traditional pi-million (3141592) block gas limit.

CHAPTER 2

Raiden Transport

2.1 Requirements

- Unicast Messages
- Broadcast Messages
- E2E encryption for unicast messages
- Authentication (i.e. messages should be linkable to an Ethereum account)
- Low latency (~100ms)
- Scalability (??? messages/s)
- Spam protection / Sybil Attack resistance
- Decentralized (no single point of failure / censorship resistance)
- Off the shelf solution, well maintained
- JS + Python SDK
- Open Source / Open Protocol

2.2 Proposed Solution: Federation of Matrix Homeservers

https://matrix.org/docs/guides/faq.html

Matrix is a federated open source store+forward messaging system, which supports group communication (multicast) via chat rooms. Direct messages are modeled as 2 participants in one chat room with appropriate permissions. Homeservers can be extended with custom logic (application services) e.g. to enforce certain rules (or message formats) in a room. It provides JS and python bindings and communication is done via HTTP long polling. Although additional server logic may be implemented to enforce some of the rules below, this enforcement must not be a requirement for a server to join the servers federation. Therefore, any standard Matrix server should work on the network.

2.3 Use in Raiden

2.3.1 Identity

The identity verification MUST not be tied to Matrix identities. Even though Matrix provides an identity server, it is a possible central point of failure. All state-changing messages passed between participants MUST be signed using the private key of the ethereum account, using Matrix only as a transport layer.

The messages MUST be validated using ecrecover by receiving parties.

The conventions below provide the means for the discovery process, and affect only the transport layer (thus not tying the whole stack to Matrix). It's enforced by the clients, and is not a requirement enforced by the server.

Matrix's `userId` (defined at registration time, in the form `@<userId>:<homeserver_uri>`) is required to be an `@`, followed by the lowercased ethereum address of the node, possibly followed by a 4-bytes hex-encoded random suffix, separated from the address by a dot, and continuing with the domain/server uri, separated from the username by a colon.

This random suffix to the username serves to avoid a client being unable to register/join the network due to someone already having taken the canonical address on an open-registration server. It may be pseudo-randomly/deterministically generated from a secret known only by the account (e.g. a python's `Random()` generator initialized with a secret derived from the user's privatekey). The same can be applied to the password generation, possibly including the server's URI on the generation process to avoid password-reuse. These conventions about how to determine the suffix and password can't be enforced by other clients, but may be useful to allow retrieval of credentials upon state-loss.

As anyone can register any `userId` on any server, to avoid the need to process every invalid message, it's required that `displayName` (an attribute for every matrix-user) is the signature of the full `userId` with the same ethereum key. This is just an additional layer of protection, as the state-changing messages have their signatures validated further up in the stack.

Example:

```
seed = int.from_bytes(web3.eth.sign(b'seed')[-32:], 'big')
rand = Random()
rand.seed(seed)
suffix = rand.randint(0, 0xffffffff)
# 0xdeadbeef
username = web3.eth.defaultAccount + "." + hex(suffix)  # 0-left-padded
# 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.deadbeef
password = web3.eth.sign(server_uri)
matrix.register_with_password(username, password)
userid = "@" + username + ":" + server_uri
matrix.get_user(userid).set_display_name(web3.eth.sign(userid))
```

2.3.2 Discovery

In the above system, clients can search the matrix server for any "seen" user whose `displayName` or `userId` contains the ethereum address (through server-side user directory search). The server is made to know about users in the network by sharing a "global" presence room. Candidates for being the actual user should only be considered after validating their `displayName` as being the signed `userId`. Most of the time though trust should not be required and all possible candidates may be contacted/invited (for a channel room, for example), as the actual interactions (messages) will always be validated up the stack. Privacy can be provided by using private p2p rooms (invite-only, encrypted rooms with 2 participants). The global presence rooms shouldn't be listened for messages nor written to in order to avoid spam. They should have the name in the format `raiden_<network_name_or_id>_discovery` (e.g. `raiden_ropsten_discovery`), and be in a hardcoded homeserver. Such a server isn't a single point of

failure because it's federated between all the servers in the network but it must have an owner server to be found by other clients/servers.

2.3.3 Presence

Matrix allows to check for the presence of a user. Clients should listen for changes in presence status of users of interest (e.g. peers), and update user status if required (e.g. gone offline), which will allow the Raiden node to avoid trying to use this user e.g. for mediating transfers.

2.3.4 Sending transfer messages to other nodes

Direct Message, which is modeled as a room with 2 participants. Channel rooms have the name in the format `raiden_<network_name_or_id>_<peerA>_<peerB>`, where `peerA` and `peerB` are sorted in lexical-order. As the users may roam to other homeservers, participants should keep listening for user-join events in the presence room, if it's a user of interest (with which it shares a room), with valid signed `displayName` and not yet in the room we share with it, invite it to the room.

2.3.5 Updating Monitoring Services

Either a) direct (DM to MS) or b) group communication (message in a group with all MS), possibly settings could be such, that only the MS are delivered the messages.

2.3.6 Updating Pathfinding Services

Similar to above

2.3.7 Chat Rooms

Peer discovery room

One per network. Participants can discover peers willing to open more channels. It may be implemented in the future as one presence/peer discovery room per token network, but it'd complicate the room-ownership/creation/server problem (rooms need to belong to a server. Whose server? Who created it? Who has admin rights for it?).

Monitoring Service Updater Room

Raiden nodes that plan to go offline for an extended period of time can submit a *balance proof* to the Monitoring Service room. The Monitoring Service will challenge Channel on their behalf in case there's an attempt to cheat (i.e. close the channel using earlier BP)

Pathfinding Service Updater Room

Raiden nodes can query shortest path to a node in a Pathfinding room.

Direct Communication Rooms

In Matrix, users can send direct e2e encrypted messages to each other through private/invite-only rooms.

Blockchain Event Rooms

Each RSB operator could provide a room, where relevant events from Raiden Token Networks are published. E.g. signed, so that false info could be challenged.

CHAPTER 3

Raiden Network Smart Contracts Specification

3.1 Overview

This is the specification document for the Solidity smart contracts required for building the Raiden Network. All functions, their signatures, and their semantics.

3.2 General Requirements

3.2.1 Secure

- A `participant` `MUST NOT` be able to steal funds. Therefore, a participant `MUST NOT` receive more tokens than he is entitled to, after calculating his final balance, unless this is due to his `partner`'s attempt to cheat.
- A participant `MUST` be able to eventually retrieve his tokens from the channel, regardless of his partner's availability in the network.
- The `sum of the final balances` of the two channel participants, after the channel lifecycle has ended, `MUST NOT` be greater than the entire channel deposit available at settlement time.
- The signed messages `MUST` be non malleable.
- A participant `MUST NOT` be able to change the state of a channel by using a signed message from an old and settled channel with the same `partner` or from another channel.

3.2.2 Privacy

- A participant's payment pattern in time `MUST NOT` be public on-chain (smart contracts only know about the final balance proofs, not all the intermediary ones).
- Participant addresses can be public.
- The final transferred amounts of the two participants can be public.

- The channel deposit can be public.

3.2.3 Fast

- It must provide means to do faster transfers (off-chain transaction)

3.2.4 Cheap

- Gas usage optimization is a target

3.3 Project Requirements

- The system must work with the most popular token standards (e.g. ERC20).
- There must not be a way for a single party to hold other user's tokens hostage, therefore the system must hold in escrow any tokens that are deposited in a channel.
- Losing funds as a penalty is not considered stealing, but must be clearly documented.
- The system must support smart locks.
- The system must expose the network graph. Clients have to collect events in order to derive the network graph.

3.4 Data structures

Note: The signed message format used in the data structures below is of this format: `ecdsa_recoverable(privkey, keccak256("\x19Ethereum Signed Message:\n" || message_length || message))`

Where:

- `message_length`: Length of the actual message to be signed in decimal representation (not null-terminated).
- `message = token_network_address || chain_id || message_type_id || message_specific_data`
- `message_type_id` has a different value depending on the type of message signed

This is compatible with https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_sign and https://github.com/ethereum/EIPs/blob/master/EIPS/eip-191.md.

Message content is tightly packed as described here: https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#abi-packed-mode.

3.4.1 Balance Proof

Onchain Balance Proof is generated by clients from *Offchain Balance Proofs*.

```
ecdsa_recoverable(privkey, sha3_keccak("\x19Ethereum Signed Message:\n212" || token_
↪network_address || chain_id || message_type_id || channel_identifier || balance_
↪hash || nonce || additional_hash))
```

Fields

Field Name	Field Type	Description
signature_prefix	string	`\x19Ethereum Signed Message:\n`
message_length	string	`212` = length of message = `20 + 32 + 32 + 32 + 32 + 32 + 32`
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
message_type_id	uint256	`1` = message type identifier
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
balance_hash	bytes32	Balance data hash
nonce	uint256	Strictly monotonic value used to order transfers. The nonce starts at 1
additional_hash	bytes32	Hash of the offchain message that contains the balance proof (possibly application-specific metadata can be also hashed in here)
signature	bytes	Elliptic Curve 256k1 signature on the above data

Balance Data Hash

`balance_hash = keccak256(transferred_amount || locked_amount || locksroot)`

Field Name	Field Type	Description
transferred_amount	uint256	Monotonically increasing amount of tokens transferred by a channel participant
locked_amount	uint256	Total amount of tokens locked in pending transfers
locksroot	bytes32	Root of merkle tree of all pending lock lockhashes

3.4.2 Balance Proof Update

```
ecdsa_recoverable(privkey, sha3_keccak("\x19Ethereum Signed Message:\n277" || token_
↪network_address || chain_id || message_type_id || channel_identifier || balance_
↪hash || nonce || additional_hash || closing_signature))
```

- `closing_signature` is the closing participant's signature on the *balance proof*

Fields

Field Name	Field Type	Description
signature_prefix	string	`\x19Ethereum Signed Message:\n`
message_length	string	`277` = length of message = `20 + 32 + 32 + 32 + 32 + 32 + 32 + 65`
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
message_type_id	uint256	`2` = message type identifier
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
balance_hash	bytes32	Balance data hash
nonce	uint256	Strictly monotonic value used to order transfers. The nonce starts at 1
additional_hash	bytes32	Hash of the offchain message that contains the balance proof (possibly application-specific metadata can be also hashed in here)
closing_signature	bytes	Elliptic Curve 256k1 balance proof signature from the closing participant
signature	bytes	Elliptic Curve 256k1 signature on the above data from the non-closing participant

3.4.3 Withdraw Proof

Data required by the smart contracts to allow a user to withdraw funds from a channel without closing it. It contains the withdraw proof which is signed by both participants.

Signatures must be valid and are defined as:

```
ecdsa_recoverable(privkey, sha3_keccak("\x19Ethereum Signed Message:\n168" || token_
↪network_address || chain_id || message_type_id || channel_identifier || participant_
↪address || total_withdraw))
```

Invariants

- `total_withdraw` is strictly monotonically increasing. This is required for protection against replay attacks with old withdraw proofs.

Fields

Field Name	Field Type	Description
signature_prefix	string	`\x19Ethereum Signed Message:\n`
message_length	string	`168` = length of message = `20 + 32 + 32 + 32 + 20 + 32`
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
message_type_id	uint256	`3` = message type identifier
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
participant_address	address	Channel participant, who withdraws the tokens
total_withdraw	uint256	Total amount of tokens that participant_address has withdrawn from the channel
participant_signature	bytes	Elliptic Curve 256k1 signature of the participant on the withdraw data
partner_signature	bytes	Elliptic Curve 256k1 signature of the partner on the withdraw data

3.4.4 Cooperative Settle Proof

Data required by the smart contracts to allow the two channel participants to close and settle the channel instantly, in one transaction. It contains the cooperative settle proof which is signed by both participants. Signatures must be valid and are defined as:

```
ecdsa_recoverable(privkey, sha3_keccak("\x19Ethereum Signed Message:\n220" || token_
↪network_address || chain_id || message_type_id || channel_identifier ||␣
↪participant1_address || participant1_balance || participant2_address ||␣
↪participant2_balance))
```

Fields

Field Name	Field Type	Description
signature_prefix	string	`\x19Ethereum Signed Message:\n`
message_length	string	`220` = length of message = `20 + 32 + 32 + 32 + 20 + 32 + 20 + 32`
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
message_type_id	uint256	`4` = message type identifier
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
participant1_address	address	One of the channel participants
participant1_balance	uint256	Amount of tokens that participant1_address will receive after settling
participant2_address	address	The other channel participant
participant2_balance	uint256	Amount of tokens that participant2_address will receive after settling
participant1_signature	bytes	Elliptic Curve 256k1 signature of participant1 on the message data
participant2_signature	bytes	Elliptic Curve 256k1 signature of participant2 on the message data

3.5 Smart Contract Functional Decomposition

3.5.1 TokenNetworkRegistry Contract

Attributes:

- `address public secret_registry_address`
- `uint256 public chain_id`
- `uint256 public settlement_timeout_min`
- `uint256 public settlement_timeout_max`

Register a token

Deploy a new `TokenNetwork` contract and add its address in the registry.

```
function createERC20TokenNetwork(address token_address) public
```

```
event TokenNetworkCreated(address token_address, address token_network_address)
```

- `token_address`: address of the Token contract.
- `token_network_address`: address of the newly deployed `TokenNetwork` contract.
- `settlement_timeout_min`: Minimum settlement timeout to be used in every `TokenNetwork`
- `settlement_timeout_max`: Maximum settlement timeout to be used in every `TokenNetwork`

Note: It also provides the `SecretRegistry` contract address to the `TokenNetwork` constructor.

3.5.2 TokenNetwork Contract

Provides the interface to interact with payment channels. The channels can only transfer the type of token that this contract defines through `token_address`.

Channel Identifier is currently defined as `uint256`, a global monotonically increasing counter of all the channels inside a `TokenNetwork`.

Note: A `channel_identifier` value of `0` is not a valid value for an active channel. The counter starts at `1`.

Attributes

- `Token public token`
- `SecretRegistry public secret_registry;`
- `uint256 public chain_id`

Getters

We currently limit the number of channels between two participants to one. Therefore, a pair of addresses can have at most one `channel_identifier`. The `channel_identifier` will be `0` if the channel does not exist.

```
function getChannelIdentifier(address participant, address partner)
    view
    public
    returns (uint256 channel_identifier)
```

```
function getChannelInfo(
    uint256 channel_identifier,
    address participant1,
    address participant2
)
    view
    external
    returns (uint256 settle_block_number, ChannelState state)
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `participant1`: Ethereum address of a channel participant.
- `participant2`: Ethereum address of the other channel participant.
- `state`: Channel state. It can be `NonExistent` - `0`, `Opened` - `1`, `Closed` - `2`, `Settled` - `3`, `Removed` - `4`.

Note: Channel state `Settled` means the channel was settled and channel data removed. However, there is still data remaining in the contract for calling `unlock` - for at least one participant.

Channel state `Removed` means that no channel data and no `unlock` data remain in the contract.

```
function getChannelParticipantInfo(
        uint256 channel_identifier,
        address participant,
        address partner
)
    view
    external
    returns (
        uint256 deposit,
        uint256 withdrawn_amount,
        bool is_the_closer,
        bytes32 balance_hash,
        uint256 nonce,
        bytes32 locksroot,
        uint256 locked_amount
    )
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `participant`: Ethereum address of a channel participant.
- `partner`: Ethereum address of the other channel participant.
- `deposit`: Can be `>=0` after the channel has been opened. Must be `0` when the channel is in `Settled` or `Removed` state.
- `withdrawn_amount`: Can be `>=0` after the channel has been opened. Must be `0` when the channel is in `Settled` or `Removed` state.
- `is_the_closer`: Can be `true` if the channel is in `Closed` state and if `participant` closed the channel. Must be `false` otherwise.

- `balance_hash`: Can be set when the channel is in `Closed` state. Must be `0` otherwise.
- `nonce`: Can be set when the channel is in a `Closed` state. Must be `0` otherwise.
- `locksroot`: Can be set when the channel is in a `Settled` state. Must be `0` otherwise.
- `locked_amount`: Can be set when the channel is in a `Settled` state. Must be `0` otherwise.

Open a channel

Opens a channel between `participant1` and `participant2` and sets the challenge period of the channel.

```
function openChannel(address participant1, address participant2, uint256 settle_
↪timeout) public returns (uint256 channel_identifier)
```

```
event ChannelOpened(
    uint256 indexed channel_identifier,
    address indexed participant1,
    address indexed participant2,
    uint256 settle_timeout
);
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `participant1`: Ethereum address of a channel participant.
- `participant2`: Ethereum address of the other channel participant.
- `settle_timeout`: Number of blocks that need to be mined between a call to `closeChannel` and `settleChannel`.

Note: Anyone can open a channel between `participant1` and `participant2`.

A participant or delegate `MUST` be able to `open` a channel with another participant if one does not exist.

A participant `MUST` be able to `reopen` a channel with another participant if there were previous channels opened between them and then settled.

Fund a channel

Deposit more tokens into a channel. This will only increase the deposit of one of the channel participants: the `participant`.

```
function setTotalDeposit(
    uint256 channel_identifier,
    address participant,
    uint256 total_deposit,
    address partner
)
    public
```

```
event ChannelNewDeposit(
    uint256 indexed channel_identifier,
    address indexed participant,
    uint256 total_deposit
);
```

- `participant`: Ethereum address of a channel participant whose deposit will be increased.
- `total_deposit`: Total amount of tokens that the `participant` will have as `deposit` in the channel.

- `partner`: Ethereum address of the other channel participant, used for computing `channel_identifier`.
- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `deposit`: The total amount of tokens deposited in a channel by a participant.

Note: Allowed to be called multiple times. Can be called by anyone.

This function is idempotent. The UI and internal smart contract logic has to make sure that the amount of tokens actually transferred is the difference between `total_deposit` and the `deposit` at transaction time.

A participant or a delegate `MUST` be able to deposit more tokens into a channel, regardless of his partner's availability.

Withdraw tokens from a channel

> **Warning:** `setTotalWithdraw` function is currently commented out and is not available.

Allows a channel participant to withdraw tokens from a channel without closing it. Can be called by anyone. Can only be called once per each signed withdraw proof.

```
function setTotalWithdraw(
    uint256 channel_identifier,
    address participant,
    uint256 total_withdraw,
    bytes participant_signature,
    bytes partner_signature
)
    external
```

```
event ChannelWithdraw(
    uint256 indexed channel_identifier,
    address indexed participant,
    uint256 total_withdraw
);
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `participant`: Ethereum address of a channel participant who will receive the tokens withdrawn from the channel.
- `total_withdraw`: Total amount of tokens that are marked as withdrawn from the channel during the channel lifecycle.
- `participant_signature`: Elliptic Curve 256k1 signature of the channel `participant` on the *withdraw proof* data.
- `partner_signature`: Elliptic Curve 256k1 signature of the channel `partner` on the *withdraw proof* data.

Note: A `participant MUST NOT` be able to withdraw tokens from the channel without his `partner`'s signature. A `participant MUST NOT` be able to withdraw more tokens than his available balance `AB`, as defined in the *settlement algorithm*. A `participant MUST NOT` be able to withdraw more tokens than the available channel deposit `TAD`, as defined in the *settlement algorithm*.

Close a channel

Allows a channel participant to close the channel. The channel cannot be settled before the challenge period has ended.

```
function closeChannel(
    uint256 channel_identifier,
    address partner,
    bytes32 balance_hash,
    uint256 nonce,
    bytes32 additional_hash,
    bytes signature
)
    public
```

```
event ChannelClosed(uint256 indexed channel_identifier, address indexed closing_
↪participant);
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `partner`: Channel partner of the participant who calls the function.
- `balance_hash`: Hash of the balance data `keccak256(transferred_amount, locked_amount, locksroot)`
  - `transferred_amount`: The monotonically increasing counter of the partner's amount of tokens sent.
  - `locked_amount`: The sum of the all the tokens that correspond to the locks (pending transfers) contained in the merkle tree.
  - `locksroot`: Root of the merkle tree of all pending lock lockhashes for the partner.
- `nonce`: Strictly monotonic value used to order transfers.
- `additional_hash`: Computed from the message. Used for message authentication.
- `signature`: Elliptic Curve 256k1 signature of the channel partner on the *balance proof* data.
- `closing_participant`: Ethereum address of the channel participant who calls this contract function.

Note: Only a participant may close the channel.

A participant `MUST` be able to set his partner's balance proof on-chain, in order to be used in the settlement algorithm.

Only a valid signed *balance proof* from the channel `partner` `MUST` be accepted. This *balance proof* sets the amount of tokens owed to the `participant` by the channel `partner`.

A `participant` `MUST` be able to close a channel regardless of his `partner`'s availability (online/offline status).

Update non-closing participant balance proof

Called after a channel has been closed. Can be called by any Ethereum address and allows the non-closing participant to provide the latest *balance proof* from the closing participant. This modifies the stored state for the closing participant.

```
function updateNonClosingBalanceProof(
    uint256 channel_identifier,
    address closing_participant,
    address non_closing_participant,
    bytes32 balance_hash,
    uint256 nonce,
    bytes32 additional_hash,
    bytes closing_signature,
    bytes non_closing_signature
```

(continues on next page)

(continued from previous page)

```
)
    external
```

```
event NonClosingBalanceProofUpdated(
    uint256 indexed channel_identifier,
    address indexed closing_participant,
    uint256 nonce
);
```

- `channel_identifier`: Channel identifier assigned by the current contract.
- `closing_participant`: Ethereum address of the channel participant who closed the channel.
- `non_closing_participant`: Ethereum address of the channel participant who is updating the balance proof data.
- `balance_hash`: Hash of the balance data
- `nonce`: Strictly monotonic value used to order transfers.
- `additional_hash`: Computed from the offchain message. Used for message authentication. Potentially useful for hashing in other application-specific metadata.
- `closing_signature`: Elliptic Curve 256k1 signature of the closing participant on the *balance proof* data.
- `non_closing_signature`: Elliptic Curve 256k1 signature of the non-closing participant on the *balance proof* data.
- `closing_participant`: Ethereum address of the participant who closed the channel.

Note: Can be called by any Ethereum address due to the requirement of providing signatures from both channel participants.

The `participant` who did not close the channel `MUST` be able to send to the *Token Network* contract his `partner`'s *balance proof*, in order to retrieve his tokens.

Only a valid signed *balance proof* from the channel's `closing participant` (the other channel participant) `MUST` be accepted. This *balance proof* sets the amount of tokens owed to the `non-closing participant` by the `closing participant`.

Only a valid signed *balance proof update* `MUST` be accepted. This update is a confirmation from the `non-closing participant` that the contained *balance proof* can be set on his behalf.

Settle channel

Settles the channel by transferring the amount of tokens each participant is owed. We need to provide the entire balance state because we only store the balance data hash when closing the channel and updating the non-closing participant balance.

Note: For an explanation of how the settlement values are computed, please check *Protocol Values and Settlement Algorithm Analysis*

```
function settleChannel(
    uint256 channel_identifier,
    address participant1,
    uint256 participant1_transferred_amount,
```

(continues on next page)

(continued from previous page)

```
    uint256 participant1_locked_amount,
    bytes32 participant1_locksroot,
    address participant2,
    uint256 participant2_transferred_amount,
    uint256 participant2_locked_amount,
    bytes32 participant2_locksroot
)
    public
```

```
event ChannelSettled(
    uint256 indexed channel_identifier,
    uint256 participant1_amount,
    uint256 participant2_amount
);
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `participant1`: Ethereum address of one of the channel participants.
- `participant1_transferred_amount`: The monotonically increasing counter of the amount of tokens sent by `participant1` to `participant2`.
- `participant1_locked_amount`: The sum of the all the tokens that correspond to the locks (pending transfers sent by `participant1` to `participant2`) contained in the merkle tree.
- `participant1_locksroot`: Root of the merkle tree of all pending lock lockhashes (pending transfers sent by `participant1` to `participant2`).
- `participant2`: Ethereum address of the other channel participant.
- `participant2_transferred_amount`: The monotonically increasing counter of the amount of tokens sent by `participant2` to `participant1`.
- `participant2_locked_amount`: The sum of the all the tokens that correspond to the locks (pending transfers sent by `participant2` to `participant1`) contained in the merkle tree.
- `participant2_locksroot`: Root of the merkle tree of all pending lock lockhashes (pending transfers sent by `participant2` to `participant1`).

Note: Can be called by anyone after a channel has been closed and the challenge period is over.

We expect the `cooperativeSettle` function to be used as the go-to way to end a channel's life. However, this would require both Raiden nodes to be online at the same time. For cases where a Raiden node is not online, the uncooperative settle will be used (`closeChannel` -> `updateNonClosingBalanceProof` -> `settleChannel` -> `unlock`). This is why the `settleChannel` transaction `MUST` never fail from internal errors - tokens `MUST` not remain locked inside the contract without a way of retrieving them. `settleChannel` can only receive balance proof values that correspond to the stored `balance_hash`. Therefore, any overflows or underflows (or other potential causes of failure) `MUST` be handled graciously.

We currently enforce an ordering of the participant data based on the following rule: `participant2_transferred_amount + participant2_locked_amount >= participant1_transferred_amount + participant1_locked_amount`. This is an artificial rule to help the settlement algorithm handle overflows and underflows easier, without failing the transaction. Therefore, calling `settleChannel` with wrong input arguments order must be the only case when the transaction can fail.

Cooperatively close and settle a channel

Warning: `cooperativeSettle` function is currently commented out and is not available.

Allows the participants to cooperate and provide both of their balances and signatures. This closes and settles the channel immediately, without triggering a challenge period.

```
function cooperativeSettle(
    uint256 channel_identifier,
    address participant1_address,
    uint256 participant1_balance,
    address participant2_address,
    uint256 participant2_balance,
    bytes participant1_signature,
    bytes participant2_signature
)
    public
```

- `channel_identifier`: *Channel identifier* assigned by the current contract
- `participant1_address`: Ethereum address of one of the channel participants.
- `participant1_balance`: Channel balance of `participant1_address`.
- `participant2_address`: Ethereum address of the other channel participant.
- `participant2_balance`: Channel balance of `participant2_address`.
- `participant1_signature`: Elliptic Curve 256k1 signature of `participant1` on the *cooperative settle proof* data.
- `participant2_signature`: Elliptic Curve 256k1 signature of `participant2` on the *cooperative settle proof* data.

Note: Emits the ChannelSettled event.

A `participant MUST NOT` be able to cooperatively settle a channel without his `partner`'s signature on the agreed upon balances.

Can be called by a third party because both signatures are required.

Unlock lock

Unlocks all pending transfers by providing the entire merkle tree of pending transfers data. The merkle tree is used to calculate the merkle root, which must be the same as the `locksroot` provided in the latest *balance proof*.

```
function unlock(
    uint256 channel_identifier,
    address participant,
    address partner,
    bytes merkle_tree_leaves
)
    public
```

```
event ChannelUnlocked(
    uint256 indexed channel_identifier,
    address indexed participant,
    address indexed partner,
    bytes32 locksroot,
```

(continues on next page)

(continued from previous page)

```
    uint256 unlocked_amount,
    uint256 returned_tokens
);
```

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `participant`: Ethereum address of the channel participant who will receive the unlocked tokens that correspond to the pending transfers that have a revealed secret.
- `partner`: Ethereum address of the channel participant that pays the amount of tokens that correspond to the pending transfers that have a revealed secret. This address will receive the rest of the tokens that correspond to the pending transfers that have not finalized and do not have a revelead secret.
- `merkle_tree_leaves`: The data for computing the entire merkle tree of pending transfers. It contains tightly packed data for each transfer, consisting of `expiration_block`, `locked_amount`, `secrethash`.
- `expiration_block`: The absolute block number at which the lock expires.
- `locked_amount`: The number of tokens being transferred from `partner` to `participant` in a pending transfer.
- `secrethash`: A hashed secret, `sha3_keccack(secret)`.
- `unlocked_amount`: The total amount of unlocked tokens that the `partner` owes to the channel `participant`.
- `returned_tokens`: The total amount of unlocked tokens that return to the `partner` because the secret was not revealed, therefore the mediating transfer did not occur.

Note: Anyone can unlock a transfer on behalf of a channel participant. `unlock` must be called after `settleChannel` because it needs the `locksroot` from the latest *balance proof* in order to guarantee that all locks have either been unlocked or have expired.

3.5.3 SecretRegistry Contract

This contract will store the block height at which the secret was revealed in a mediating transfer. In collaboration with a monitoring service, it acts as a security measure, to allow all nodes participating in a mediating transfer to withdraw the transferred tokens even if some of the nodes might be offline.

```
function registerSecret(bytes32 secret) public returns (bool)

function registerSecretBatch(bytes32[] secrets) public returns (bool)
```

```
event SecretRevealed(bytes32 indexed secrethash, bytes32 secret);
```

Getters

```
function getSecretRevealBlockHeight(bytes32 secrethash) public view returns (uint256)
```

- `secret`: The preimage used to derive a secrethash.
- `secrethash`: `keccak256(secret)`.

3.5.4 EndpointRegistry Contract

This contract is a registry which maps a Raiden node's Ethereum address to its endpoint `host:port`. It is only used when starting the Raiden client with the UDP transport layer (the current default is the Matrix-based transport). For the UDP transport, the Raiden node must register its Ethereum address in this registry, so its endpoint can be found by other nodes in order to send the Raiden protocol messages.

Register endpoint

Registers the Ethereum address to the given endpoint. The Ethereum address saved in the registry is the address that sends the transaction (contract uses `msg.sender`).

```
function registerEndpoint(string endpoint) public
```

- `endpoint`: String in the format `127.0.0.1:38647`.

Find endpoint

Finds the endpoint if given a registered Ethereum address.

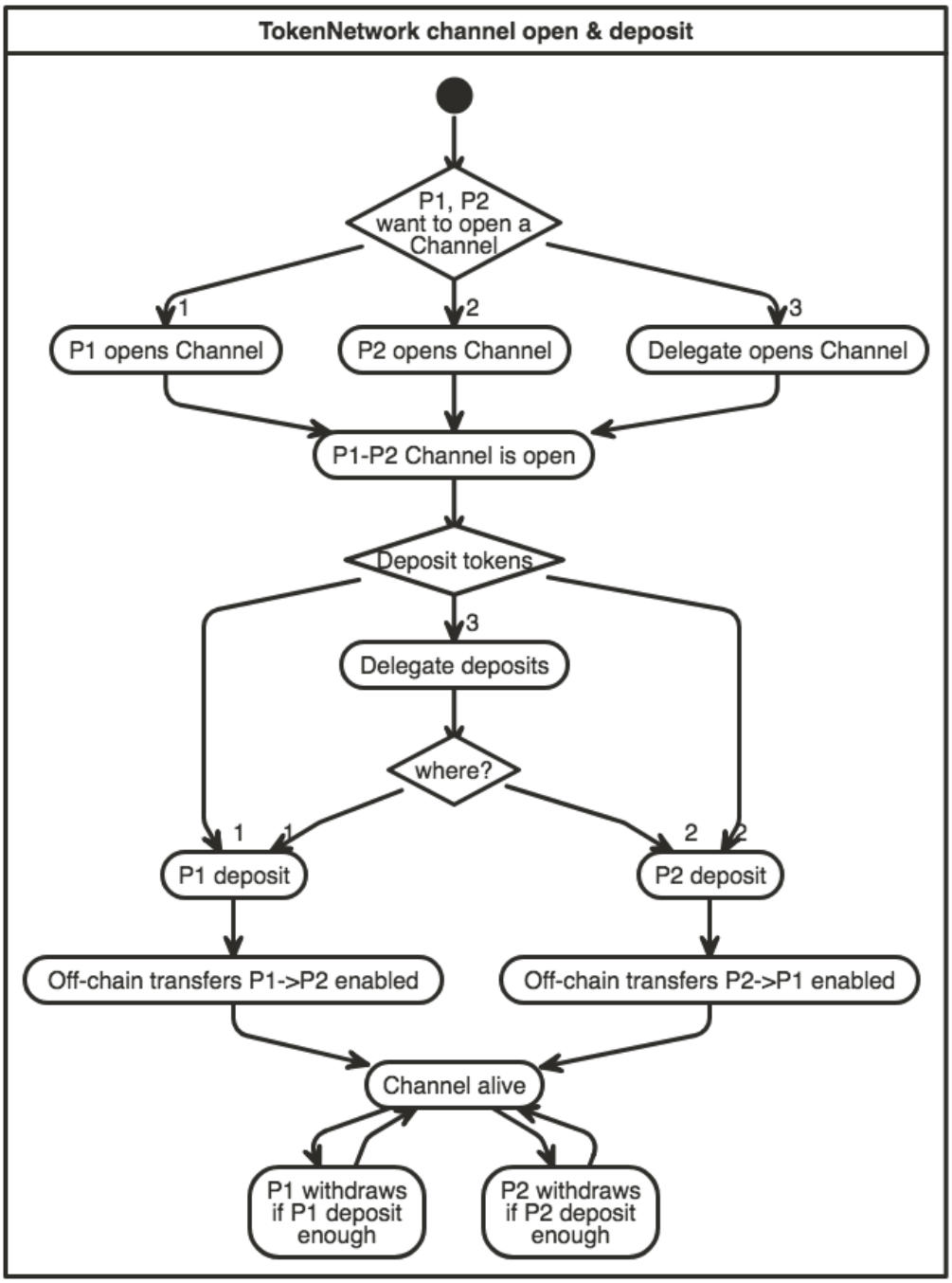
```
function findEndpointByAddress(address eth_address)
    public
    view
    returns (string endpoint)
```

- `endpoint`: String in the format `127.0.0.1:38647`.
- `eth_address`: The Raiden node's 20 byte Ethereum address.

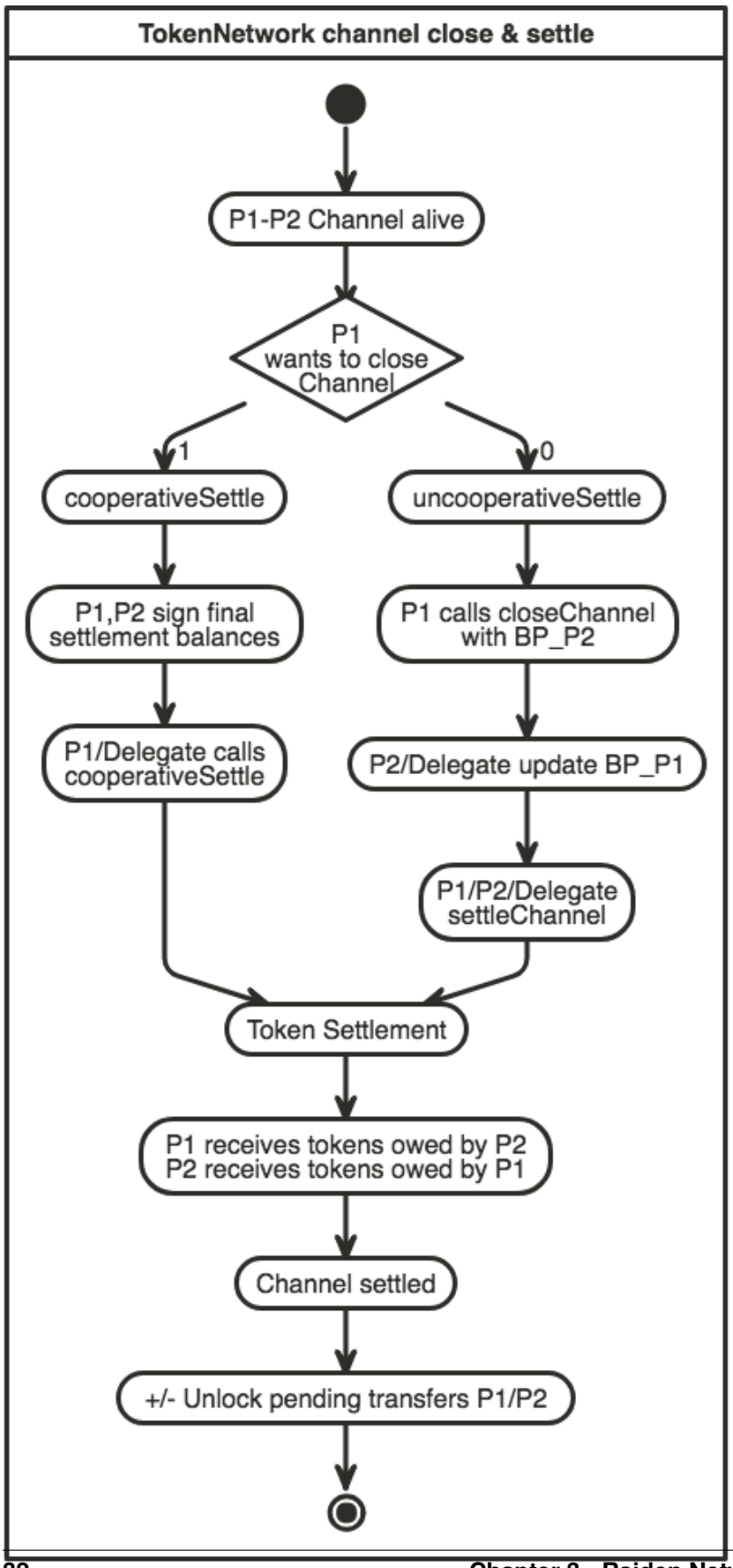
3.6 TokenNetwork Channel Protocol Overview

This section contains a few flowcharts describing the token network channel lifecycle.

3.6.1 Opened Channel Lifecycle

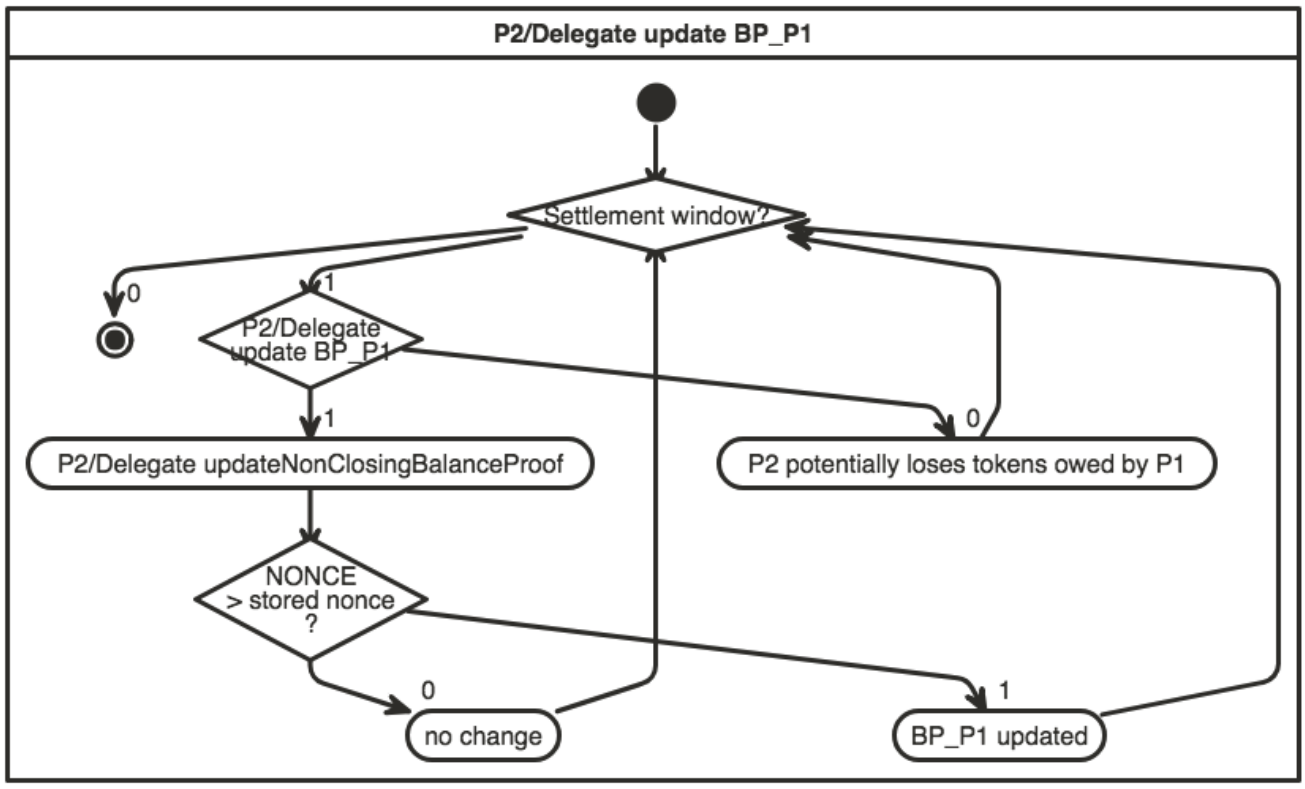


3.6.2 Channel Settlement

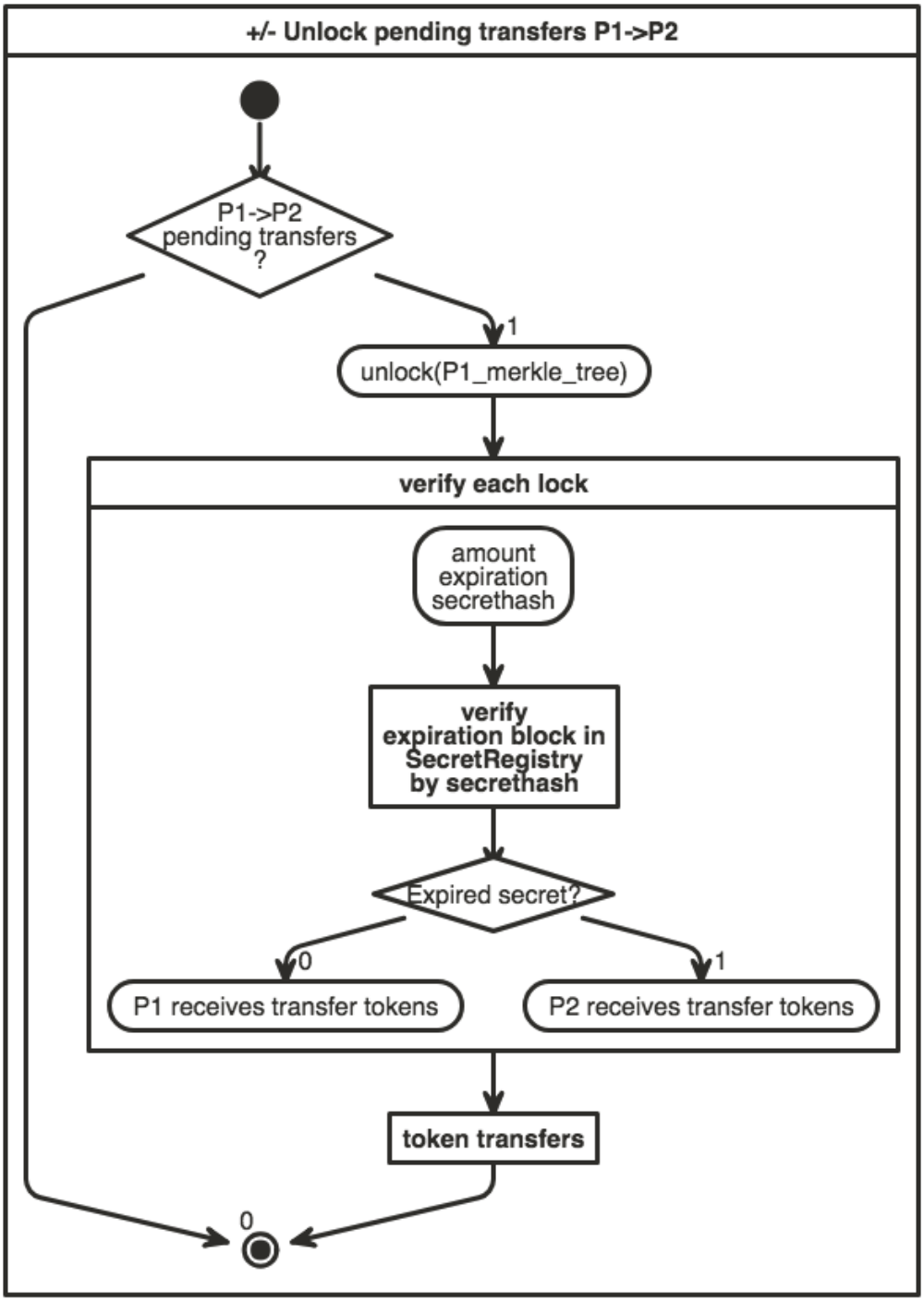


3.6.3 Channel Settlement Window

The non-closing participant can update the closing participant's balance proof during the settlement window, by calling `TokenNetwork.updateNonClosingBalanceProof`.



3.6.4 Unlocking Pending Transfers



3.7 Protocol Values and Settlement Algorithm Analysis

3.7.1 Definitions

- `valid last BP` = a balance proof that respects the official Raiden client constraints and is the last balance proof known
- `valid old BP` = a balance proof that respects the official Raiden client constraints, but there are other newer balance proofs that were created after it (additional transfers happened)
- `invalid BP` = a balance proof that does not respect the official Raiden client constraints
- `P`: A channel participant - *Participants*
- `P1`: One of the two channel participants
- `P2`: The other channel participant, or `P1`'s partner
- `D1`: Total amount of tokens deposited by `P1` in the channel using *setTotalDeposit* and shown by *getChannelParticipantInfo*
- `W1`: Total amount of tokens withdrawn from the channel by `P1` using *setTotalWithdraw* and shown by *getChannelParticipantInfo*
- `T1`: Off-chain *Transferred amount* from `P1` to `P2`, representing finalized transfers.
- `L1`: Locked tokens in pending transfers sent by `P1` to `P2`, that have not finalized yet or have expired. Corresponds to a *locksroot* provided to the smart contract in *settleChannel*. `L1 = Lc1 + Lu1`
- `Lc1`: Locked amount that will be transferred to `P2` if *unlock* is called with `P1`'s pending transfers. This only happens if the *secret* s of the pending *Hash Time Locked Transfer* s have been registered with *registerSecret*
- `Lu1`: Locked amount that will return to `P1` because the *secret* s were not registered on-chain
- `TAD`: Total available channel deposit at a moment in time: `D1 + D2 - W1 - W2, TAD >= 0`
- `B1`: Total, final amount that must be received by `P1` after channel is settled and no unlocks are left to be done.
- `AB1`: available balance for P1: *Capacity*. Determines if `P1` can make additional transfers to `P2` or not.
- `D1k` = `D1` at `time = k`; same for all of the above.

All the above definitions are also valid for `P2`. Example: `D2`, `T2` etc.

3.7.2 Protocol Values Constraints

- `TN` = enforced by the TokenNetwork contract
- `R` = enforced by the Raiden client

```
(1 TN) Dk <= Dt, if time k < time t
(2 TN) Wk <= Wt, if time k < time t
(3 R) Tk <= Tt, if time k < time t
```

Channel deposits, channel withdraws, off-chain transferred amounts are all monotonically increasing. The `TokenNetwork` contract must enforce this for deposits (code here) and withdraws (code here). The Raiden client must enforce this for the off-chain transferred amounts, contained in the balance proofs (code here and here).

```
(4 R) Tk + Lck <= Tt + Lct, if time k < time t
```

The sum of each transferred amount and the claimable amounts from the pending transfers MUST also be monotonically increasing over time. The claimable amounts Lc correspond to pending locked transfers that have a secret revealed on-chain.

- at time=t we will always have more secrets revealed on-chain than at time=k, where k < t
- even if the protocol implements off-chain unlocking of claimable pending transfers, in order to reduce the size of the merkle tree of pending transfers, the off-chain unlocked amount will be added to T and subtracted from Lc, maintaining monotonicity of T + Lc.

Note: Any two consecutive balance proofs for P1, named BP1k and BP1t were time k < time t, must respect the following constraints:

1. A succesfull *HTL Transfer* with value tokens was finalized, therefore T1t == T1k + value and L1t == L1k.
2. A *locked transfer message* with value was sent, part of a *HTL Transfer*, therefore T1t == T1k and L1t == L1k + value.
3. A *HTL Unlock* for a previous value was finalized, therefore T1t == T1k + value and L1t == L1k - value.
4. A *lock expiration* message for a previous value was done, therefore T1t == T1k and L1t == L1k - value.

```
(5 R) AB1 = D1 - W1 + T2 - T1 - L1; AB1 >= 0, AB1 <= TAD
```

The Raiden client MUST not allow a participant to transfer more tokens than he has available. Enforced here, here and here. Note that withdrawing tokens is not currently implemented in the Raiden client.

From this, we also have:

```
(5.1 R) L1 <= TAD, L1 >= 0
```

A mediated transfer starts by locking tokens through the *locked transfer message*. A user cannot send more than his available balance. Enforced in the Raiden client here.

This means that for P1:

- we need to calculate the netted transferred amounts for him: T2 - T1
- subtract any tokens that he has locked in pending transfers to P2: -L1
- do not take into consideration the pending transfers from P2: L2, because the token distribution will only be known at unlock time.

Also, the amount that a participant can receive cannot be bigger than the total channel available deposit (9). Therefore, the available balance of a participant at any point in time cannot be bigger than the total available deposit of the channel ABI1 <= TAD.

```
(6 R) W1 <= D1 + T2 - T1 - L1
```

(6 R) is deduced from (5 R). It is needed by the Raiden client in order to not allow a participant to *withdraw* more tokens from the on-chain channel deposit than he is entitled to.

Not implemented yet in the Raiden client.

```
(7 R) -(D1 - W1) <= T2 + L2 - T1 - L1 <= D2 - W2
```

`T2 + L2 - T1 - L1` is the netted total transferred amount from `P2` to `P1`. This amount cannot be bigger than `P2`'s **available** deposit. We enforce that a participant cannot transfer more tokens than what he has in the channel, during the lifecycle of a channel. This amount cannot be smaller than the negative value of `P1`'s **available** deposit `- (D1 - W1)`. This can also be deducted from the corresponding `T1 + L1 - T2 - L2 <= D1 - W1` The Raiden client `MUST` ensure this. However, it must use up-to-date values for `D2` and `W2` (e.g. Raiden node might have sent an on-chain transaction to withdraw tokens; this is not mined yet, therefore it does not reflect in the contract yet. The Raiden client will use the off-chain `W2` value.)

Not implemented yet in the Raiden client.

3.7.3 Settlement Algorithm - Protocol

The scope is to correctly calculate the final balance of the participants when the channel lifecycle has ended (after *settlement* and *unlock*). These calculations will be done off-chain for the *cooperative settle*.

The following must be true if both participants use a `last valid BP` for each other:

```
(8) B1 = D1 - W1 + T2 - T1 + Lc2 - Lc1, B1 >= 0
(9) B2 = D2 - W2 + T1 - T2 + Lc1 - Lc2, B2 >= 0
(10) B1 + B2 = TAD, where TAD = D1 + D2 - W1 - W2, TAD >= 0
```

For each participant, we must calculate the netted transferred amounts and then the token amounts from pending transfers. Note that the pending transfer distribution can only be known at the time of calling *unlock*.

The above is easy to calculate off-chain for the `cooperativeSettle` transaction, because the Raiden node has all the needed information.

Uncooperative Settlement Algorithm - Protocol

For the uncooperative settle protocol, there are also some additional contraints:

- `settleChannel` must never fail (see *settleChannel noted*)
- `settleChannel` must calculate correctly the amount of tokens transferred to the participants at settlement time and the amount of tokens remaining in the contract for a later `unlock`, even if the `TokenNetwork` smart contract has no way of knowing the pending transfers distribution at this time (`Lc1, Lu1, Lc2, Lu2`)
- the `settleChannel` transaction `MUST` be able to handle `valid old` balance proofs in a way that participants cannot be cheatead if their partner uses such a balance proof.
- `settleChannel` `MUST` be able to handle `invalid` balance proofs (not constructed by an official Raiden client). However, the smart contract has no way to ensure correctness of the final balances.

For the ideal case (both balance proofs are *valid last*), we could compute the netted transferred amount balances and distribute them within the `settleChannel` transaction, leaving all the pending transfer amounts inside the contract:

- `S1`: amount received by `P1` when calling `settleChannel`
- `SL1`: pending transfer locked amount, corresponding to `L1` that will remain locked in the TokenNetwork contract when calling `settleChannel`, to be unlocked later.

```
S1 = D1 - W1 + T2 - T1 - L1
S2 = D2 - W2 + T1 - T2 - L2

SL1 = L1
SL2 = L2
```

Because the TokenNetwork contract can receive old balance proofs from participants, the balance proof values might not respect B1 + B2 = TAD. The TokenNetwork contract might need to retain SL1 != L1 and SL2 != L2, as will be explained below.

3.7.4 Settlement Algorithm - Solidity Implementation

The problem is that, in Solidity, we need to handle overflows and underflows gracefully, making sure that no tokens are lost in the process.

For example: S1 = D1 - W1 + T2 - T1 - L1 cannot be computed in this order. D1 - W1 can result in an underflow, because D1 can be smaller than W1.

The end results of respecting all these constraints while also ensuring fair balances, are:

- a special Solidity-compatible settlement algorithm
- a set of additional constraints that MUST be enforced in the Raiden client.

Solidity Settlement Algorithm

- TLmax1: the maximum amount that P1 might transfer to P2 (if his pending transfers will all be claimed)
- RmaxP1: the maximum receivable amount by P1 at settlement time; this concept exists only for handling the overflows and underflows.

```
TLmax1 = T1 + L1
TLmax2 = T2 + L2
RmaxP1 = TLmax2 - TLmax1 + D1 - W1
RmaxP1 = min(TAD, RmaxP1)
SL2 = min(RmaxP1, L2)
S1 = RmaxP1 - SL2
RmaxP2 = TAD - RmaxP1
SL1 = min(RmaxP2, L1)
S2 = RmaxP2 - SL1
```

Additional Overflow Constraints

```
(11 R) T1 + L1 < 2^256 ; T2 + L2 < 2^256
```

This ensures that calculating RmaxP1 does not overflow on T2 + L2 and T1 + L1. Enforced by the Raiden client here.

```
(12) D1 + D2 < 2^256
```

This is enforced by the TokenNetwork contract here.

Solidity Settlement Algorithm - Explained

Note: The overflows and underflows do not happen for a valid last pair of balance proofs. They only happen when at least one balance proof is valid old or the TokenNetwork contract receives invalid balance proofs.

```
TLmax1 = T1 + L1
TLmax2 = T2 + L2
RmaxP1 = TLmax2 - TLmax1 + D1 - W1
```

- `(11 R)` solves overflows for `TLmax1` and `TLmax2`
- `TLmax2 - TLmax1` underflow is solved by setting an order on the input arrguments that *settleChannel* receives. The order in which `RmaxP1` and `RmaxP2` is computed does not affect the result of the calculation for valid balance proofs.
- `(7 R)` solves the `+ D1` overflow: `T2 + L2 - T1 - L1 <= D2 - W2` –> `T2 + L2 - T1 - L1 + D1 <= D1 + D2 - W2`. `(12)` makes sure `D1 + D2` has no overflow.
- `(6 R)` solves the `- W1` underflow

```
RmaxP1 = min(TAD, RmaxP1)
```

We bound `RmaxP1` to `TAD`, to ensure that participants do not receive more tokens than their channel has available.

```
RmaxP2 = TAD - RmaxP1
```

- underflow is solved by the above bounding of `RmaxP1` to `TAD`.

```
SL2 = min(RmaxP1, L2)
```

We bound `L2` to `RmaxP1` in case old balance proofs are used. There are cases where old balance proofs can have a bigger `L2` amount than a later balance proof, if they contain expired locks that have been later removed from the merkle tree of pending transfers or contain claimable locked amounts that have been later claimed on-chain.

```
S1 = RmaxP1 - SL2
```

- underflow is solved by the above bounding of `L2` to `RmaxP1`.

```
SL1 = min(RmaxP2, L1)
```

We bound `L2` to `RmaxP1` in case old balance proofs are used.

```
S2 = RmaxP2 - SL1
```

- underflow is solved by the above bounding of `L1` to `RmaxP2`.

Note: Demonstration that the above Solidity implementation results in fair balances for the participants at the end of the channel lifecycle can be found here: https://github.com/raiden-network/raiden-contracts/issues/188

CHAPTER 4

Raiden Network Monitoring Service

4.1 Basic requirements for the MS

- Good enough uptime (a third party service to monitor the servers can be used to provide statistics)
- Sybil Attack resistance (i.e. no one should be able to announce an unlimited number of (faulty) services)
- Some degree of redundancy (ability to register balance proof with multiple competing monitoring services)
- A stable and fast ethereum node connection (channel update transactions should be propagated ASAP as there is competition among the monitoring services)
- If MS registry is used, a deposit will be required for registration

4.2 Usual scenario

The Raiden node that belongs to Alice is going offline and Alice wants to be protected against having her channels closed by Bob with an incorrect *balance proof*.

1. Alice broadcasts the balance proof by sending a message to a public chat room.
2. Monitoring services decide if the fee is worth it and picks the balance proof up.
3. Alice now goes offline.
4. Bob sends an on-chain transaction in attempt to use an earlier balance proof that is in his favor.
5. Some of the monitoring servers detect that an incorrect *BP* is being used. They can update the channel closing state as long as the *Challenge Period* is not over.
6. After the Challenge period expires, the channel can be settled with a balance that Alice expects to be correct.

4.3 Economic incentives

4.3.1 Raiden node

A Raiden node wants to register its *BP* to as many Monitoring Services as possible. The cost of registering should be strictly less than a potential token loss in case of malicious channel close by the other participant.

4.3.2 Monitoring service

Monitoring Service is motivated to collect as many BP as possible, and the reward **should** be higher than cost of sending the *Challenge Period Update*. The reward collected over the time **should** also cover costs (i.e. electricity, VPS hosting...) of running the service.

4.3.3 General requirements

MS that wish to get assigned a MS address (term?) in the global chat room MUST provide a registration deposit via SC [TBD]

Users wishing to use a MS are RECOMMENDED to provide a reward deposit via smart contract [TBD]

Users that want channels to be monitored MUST post BPs concerning those channels to the global chat room along with the reward amount they're willing to pay for the specific channel. They also MUST provide proof of a deposit equal to or exceeding the advertised reward amount. The offered reward amount MAY be zero.

Monitoring services MUST listen in the provided global chat room

They can decide to accept any balance proofs that are submitted to the chat room.

Once it does accept a BP it MUST provide monitoring for the associated channel at least until a newer BP is provided or the channel is settled. MS SHOULD continue to accept newer balance proofs for the same channel.

Once a *ChannelClosed* or *NonClosingBalanceProofUpdated* event is seen the MS MUST verify that the channel's balance matches the latest BP it accepted. If the balances do not match the MS MUST submit that BP to the channel's *updateNonClosingBalanceProof* method.

[TBD] There needs to be a selection mechanism which MS should act at what time (see below in "notes / observations")

MS SHOULD inspect pending transactions to determine if there are already pending calls to *updateTransfer* for the channel. If there are a MS SHOULD delay sending its own update transaction. (Needs more details)

4.3.4 Fees/Rewards structure

Monitoring servers compete to be the first to provide a balance proof update. This mechanism is simple to implement: MS will decide if the risk/reward ratio is worth it and submits an on-chain transaction.

Fees have to be paid upfront. A smart contract governing the reward payout is required, and will probably add an additional logic to the channel contract code.

Proposed SC logic

1. Raiden node will transfer tokens used as a reward to the channel smart contract.
2. Whoever calls SC's updateTransfer method MUST supply payout address as a parameter. This address is stored in the SC. updateTransfer MAY be called multiple times, but it will only accept BP newer than the previous one.

3. When settling (calling contract suicide), the reward tokens will be sent to the payout address.

4.3.5 Notes/observations

How will raiden nodes specify/deposit the monitoring fee? How will it be collected?

A scheme to prevent unnecessary simultaneous updates needs to exist. Options: MS chose an order amongst themselves

4.4 Appendix A: Interfaces

4.4.1 Broadcast interface

Client's request to store Balance Proof will be in the usual scenario broadcasted using Matrix as a transport layer. A public chatroom will be available for anyone to join - clients will post balance proofs to the chatroom and Monitoring Service picks them up.

4.4.2 Web3 Interface

Monitoring service requires a synced Ethereum node with an enabled JSON-RPC interface. All blockchain operations are performed using this connection.

Event filtering

MS MUST filter events for each onchain channel that corresponds to the submitted Balance Proofs. On `ChannelClosed` and `NonClosingBalanceProofUpdated` events state the channel was closed with MUST be compared with the Balance Proof. In case of any discrepancy, channel state must be updated immediately. On `ChannelSettled` event any state data for this channel MAY be deleted from the MS.

4.4.3 REST interface

The monitoring service MAY expose some of the functionality over RESTful API. There might be API endpoints that SHOULD be protected from public access (i.e. using some form of authentication).

Endpoints

- `GET /api/1/balance_proofs` - return a JSON list of known balance proofs
- `DEL /api/1/balance_proofs/<channel_address>` - remove balance proof from the internal database
- `PUT /api/1/balance_proofs` - register a balance proof
- `GET /api/1/channel_update` - return a JSON list of already performed channel updates.
- `GET /api/1/channel_update/<channel_address>` - return a list of updates for a given channel
- `GET /api/1/stats` - various statistics of the server, including count of balance proofs stored, count of balance proofs submitted, count of unique Participants etc.

4.5 Appendix B: Message format

Monitoring service uses JSON format to exchange the data. For description of the envelope format and required fields of the message please see Transport document.

4.5.1 Balance proof

- nonce (uint64) - it is expected that nonce is incremented by 1 with each Balance Proof exchanged between Channel Participants
- transferred_amount (uint256) - amount of tokens transferred
- channel_address (address) - address of the netting channel
- locksroot (bytes32) - lock root state of the channel
- extra_hash (bytes32) - implementation dependent extra data
- signature (bytes32) - ecrecoverable signature of the data above, in order they are listed here

All of this fields are required. Monitoring Service MUST perform basic verification of these data, namely channel existence. Monitoring service SHOULD accept the message if and only the sender of the message is same as the sender address recovered from the signature.

4.5.2 Example data: Balance proof

‘‘ {

‘nonce’: 13, ‘transferred_amount’: 15000, ‘channel_address’: ‘0x87F5636c67f2Fd4F11710974766a5B1b6f33FB1d’, ‘extra_hash’: ‘0xe0fa3e376941dafc9b3836f80bee307ab2eacb569ec7ccce ‘locksroot’: ‘0xebd7dc7d6dd7956e62104182194939a1223c738ffc2a14dbbecb6191cf76f211’, ‘signature’: ‘0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470’

CHAPTER 5

Raiden Pathfinding Service Specification

5.1 Overview

A centralized path finding service that has a global view on a token network and provides suitable payment paths for Raiden nodes.

5.2 Assumptions

- The pathfinding service in its current spec is a temporary solution.
- It should be able to handle a similar amount of active nodes as currently present in Ethereum (~20,000).
- Uncooperative nodes are dropped on the Raiden-level protocol, so paths provided by the service can be expected to work most of the time.
- User experience should be simple and free for sparse users with optional premium fee schedules for heavy users.
- No guarantees are or can be made about the feasibility of the path with respect to node uptime or neutrality.
- Hubs are incentivized to accurately report their current balances and fees to "advertise" their channels. Higher fees than reported would be rejected by the transfer initiator.
- Every pathfinding service is responsible for a single token network. Pathfinding services are scaled on process level to handle multiple token networks.

5.3 High-Level-Description

A node can request a list of possible paths from start point to endpoint for a given transfer value. The `get_paths` method implements the canonical Dijkstra algorithm to return a given number of paths for a mediated transfer of a given value. The design regards the Raiden network as an unidirectional weighted graph, where the default weights (and therefore the primary constraint of the optimization) are the fees of each channel. Additionally we applied two heuristics to quantify desirable properties of the resulting graph:

1. A hard coded parameter `DIVERSITY_PEN_DEFAULT` defined in the config; this value is added to each edge that is part of a returned path as a bias. This results in an output of "pseudo-disjoint" paths, i.e. the optimization will prefer paths with a minimal edge intersection. This should enable nodes to have a suitable amount of options for their payment routing in the case some paths are slow or broken. However, if a node has only one channel (i.e. a light client) payments could be routed through, the method will still return the specified `number of paths`.
2. The second heuristic is configurable via the optional argument `bias`, which models the trade-off between speed and cost of mediated transfer; with default 0, `get_paths` will optimize with respect to overall fees only (i.e. the cheapest path). On the other hand, with `bias=1`, `get_paths` will look for paths with the minimal number of hops (i.e. the -theoretical - fastest path). Any value in `[0,1]` is accepted, an appropriate value depends on the average `channel_fee` in the network (in simulations `mean_fee` gave decent results for the trade-off between speed and cost). The reasoning behind this heuristic is that a node may have different needs, w.r.t to good to be paid for - buying a potato should be fast, buying a yacht should incorporate low fees.

5.4 Public Interface

5.4.1 Definitions

The following data types are taken from the Raiden Core spec.

Channel_Id

- uint: channel_identifier

Balance_Proof

See *offchain balance proof*.

Lock

- uint64: expiration
- uint256: locked_amount
- bytes32: secrethash

5.4.2 Public Endpoints

A path finding service must provide the following endpoints. The interface has to be versioned.

The examples provided for each of the endpoints is for communication with a REST endpoint.

`api/1/<token_network_address>/<channel_id>/balance`

Update the balance for the given channel with the provided *balance proof*. The receiver can be read from the balance proof.

Arguments

Field Name	Field Type	Description
token_network_address	address	The token network address for which the balance is updated.
channel_id	int	The channel for which the balance proof should be updated.
balance_proof	OffchainBalanceProof	The new balance proof which should be used for the given channel.
locks	List[Lock]	The list of all locks used to compute the locksroot.

Returns

True when the balance was updated or one of the following errors:

- Invalid balance proof
- Invalid channel id

Example

```
// Request
curl -X PUT --data '{
    "balance_proof": {
        "nonce": 1234,
        "transferred_amount": 23,
        "locked_amount": 0,
        "locksroot": "<keccak-hash>",
        "channel_id": 123,
        "token_network_address": "0xtoken",
        "chain_id": 1,
        "additional_hash": "<keccak-hash>",
        "balance_hash": "<keccak-hash>",
        "signature": "<signature>",
        "message_type": "BalanceProof",
    },
    "locks": [
        {
            "expiration": 200
            "locked_amount": 40
            "secrethash": "<keccak-hash>"
        },
        {
            "expiration": 50
            "locked_amount": 10
            "secrethash": "<keccak-hash>"
        },
    ],
}'  /api/1/0xtoken_network/balance
// Result for success
{
    "result": "OK"
}
// Result for failure
```

(continues on next page)

(continued from previous page)

```
{
    "error": "Invalid balance proof"
}
```

api/1/<token_network_address>/<channel_id>/fee

Update the fee for the given channel, for the outgoing channel from the partner who signed the message. A nonce is required to be incorporated in the signature for replay protection.

- Reconstructs the signers `public_key` of a requested fee update with coincurve's `from_signature_and_message` method.
- Derives the two `channel_participants` with `from channel_id`. Checks if the signing `public_key` matches one of the `channel participant`'s `address` or returns an error if the signature doesn't match.

Arguments

Field Name	Field Type	Description
token_network_address	address	The token network address for which the payment info is requested.
Channel_id	int	The channel for which the fee should be updated.
Nonce	int	A nonce for replay protection.
Fee	int	The new fee to be set.
Signature	bytes	Signature of a channel partner

Returns

True when the fee was updated or one of the following errors:

- Invalid channel id
- Invalid signature

Example

```
// Request
curl -X PUT --data '{
    "fee": 3,
    "signature": "<signature>"
}'  /api/1/0xtoken_network/123/fee
// Result for success
{
    "result": "True"
}
// Result for failure
{
    "error": "Invalid signature."
}
```

api/1/<token_network_address>/paths

The method will do `num_paths` iterations of Dijkstras algorithm on the last-known state of the Raiden Network (regarded as directed weighted graph) to return `num_paths` different paths for a mediated transfer of `value`.

- Checks if an edge (i.e. a channel) has `capacity > value`, else ignores it.
- Applies on the fly changes to the graph's weights - depends on `DIVERSITY_PEN_DEFAULT` from `config`, to penalize edges which are part of a path that is returned already.
- Depends on a user preference via the `bias` argument, to decided the trade off between fee-level vs. path-length (i.e. cost vs. speed) - default `bias = 0`, i.e. full fee minimization.

Arguments

Field Name	Field Type	Description
token_network_address	address	The token network address for which the paths are requested.
from	address	The address of the payment initiator.
to	address	The address of the payment target.
value	int	The amount of token to be sent.
num_paths	int	The maximum number of paths returned.
kwargs	any	Currently only 'bias' to implement the speed/cost opt. trade-off

Returns

A list of path objects. A path object consists of the following information:

Field Name	Field Type	Description
path	List[address]	An ordered list of the addresses that make up the payment path.
estimated_fee	int	An estimate of the fees required for that path.

If no possible path is found, one of the following errors is returned:

- No suitable path found
- Rate limit exceeded
- From or to invalid

Example

```
// Request
curl -X GET --data '{
    "from": "0xalice",
    "to": "0xbob",
    "value": 45,
    "num_paths": 10
}'  /api/1/paths
// Request with specific preference
curl -X PUT --data '{
    "from": "0xalice",
    "to": "0xbob",
```

(continues on next page)

(continued from previous page)

```
    "value": 45,
    "num_paths": 10,
    "extra_data": "min-hops"
}'  /api/1/0xtoken_network/paths
// Result for success
{
    "result": [
    {
        "path": ["0xalice", "0xcharlie", "0xbob"],
        "estimated_fees": 3
    },
    {
        "path": ["0xalice", "0xeve", "0xdave", "0xbob"]
        "estimated_fees": 5
    },
    ...
    ]
}
// Result for failure
{
    "error": "No suitable path found."
}
// Result for exceeded rate limit
{
    "error": "Rate limit exceeded, payment required. Please call 'api/1/payment/info'␣
↪to establish a payment channel or wait."
}
```

api/1/<token_network_address>/payment/info

Request price and path information on how and how much to pay the service for additional path requests. The service is paid in RDN tokens, so they payer might need to open an additional channel in the RDN token network.

Arguments

Field Name	Field Type	Description
token_network_address	address	The token network address for which the fee is updated.
rdn_source_address	address	The address of payer in the RDN token network.

Returns

An object consisting of two properties:

Field Name	Field Type	Description
price_per_request	int	The address of payer in the RDN token network.
paths	list	A list of possible paths to pay the path finding service in the RDN token network. Each object in the list contains a *path* and an *estimated_fee* property.

If no possible path is found, the following error is returned:

- No suitable path found

Example

```
// Request
curl -X GET --data '{
    "rdn_source_addressfrom": "0xrdn_alice",
}'  api/1/0xtoken_network/payment/info
// Result for success
{
    "result":
    {
        "price_per_request": 1000,
        "paths":
        [
            {
                "path": ["0xrdn_alice", "0xrdn_eve", "0xrdn_service"],
                "estimated_fees": 10_000
            },
            ...
        ]
    }
// Result for failure
{
    "error": "No suitable path found."
}
```

5.5 Implementation notes

5.5.1 Network topology updates

Note: A pathfinding service might want to cover multiple token networks. However, it always needs to cover the *RDN* token network in order to be able to provide routing information for payments.

The creation of new token networks can be followed by listening for *TokenNetworkCreated* events on the *TokenNetworksRegistry* contract.

To learn about updates of the network topology of a token network the PFS must listen for the following events:

- *ChannelOpenened*: Update the network to include the new channel
- *ChannelClosed*: Remove the channel from the network

Additionally it must listen to the *ChannelNewDeposit* event in order to learn about new deposits.

Updates for channel balances and fees are received over the designated API endpoints.

5.6 Future Work

The methods will be rate-limited in a configurable way. If the rate limit is exceeded, clients can be required to pay the path-finding service with RDN tokens via the Raiden Network. The required path for this payment will be provided

by the service for free. This enables a simple user experience for light users without the need for additional on-chain transactions for channel creations or payments, while at the same time monetizing extensive use of the API.

CHAPTER 6

Raiden Mobile Wallet Specification

(This is work in progress and will be updated as soon as integration related specifications for core protocol and other services are settled on and implemented)

6.1 Overview

Specification document for a Raiden Network Mobile Wallet implementation.

6.2 Goal

Build an easy to use wallet where (in general) the user does not need to worry about how sending and receiving payments is done.

6.3 Terminology

- RW = Raiden Mobile Wallet
- TS = Transport Service
- MS = Monitoring Service
- PFS = Path Finding Service
- RLC = Raiden Light Client

6.4 Requirements

- secure: keeps private keys safe, only in the user's custody

- good connection with 3rd party services - MS, PFS, hubs
- easy enough onboarding
- easy to use for your mom

6.5 Depends on

- specs for MS
- specs for PFS
- specs for RLC
- onboarding new users (now in PFS)

6.6 High Level Features

- support both off-chain (RN) and on-chain payments (sending and receiving)
- iOS, Android; depending on tech & tooling -> +/- web & desktop
- language: English (internationalization?)
- support main net + test nets (Ropsten, Kovan, Rinkeby) + custom test net
- support official/popular token standards
- atomic token swap transactions (through Raiden API)
- request payments via SMS, Email, Whatsapp or Whisper (Shh) maybe with prepared data (like an order) -> receiver should click on the link and the wallet should already display the transaction and ask the user to sign it.
- **notifications for**
  - successful on-chain transaction (wait for confirmations)
  - incoming on-chain transactions (normal on-chain payments, channel-related activity)
  - off-chain payments (payment received, successful payment sent)
- hub reputation system
- user encrypted chat? (Signal protocol, Whisper, Matrix etc.)
- adding new/custom tokens to interact with ? - this is easy for on-chain, but for off-chain it would mean deploying a new TokenNetwork contract (will probably not be supported in the wallet)

6.7 Platforms & Languages

(TODO: pros & cons for each)

6.7.1 Native vs. Progressive Apps

Progressive Apps

- service workers handle push notifications easier

- good cache mechanism for offline / low connectivity - IndexedDB (event based, other wrapper libs exist), Cache API (Promise based)
- smaller effort for app development and maintenance

Native

- more efficient, can be compiled to wasm

6.7.2 Languages

RLC

- https://pybee.org/ - "native" apps with Python
- compile RN code Python -> JS https://www.transcrypt.org/, https://github.com/QQuick/Transcrypt
- Python -> asm.js: http://pypyjs.org/ (step2 - asm.js -> wasm might not support everything needed for the initial py implementation)
- Python -> wasm (WIP): https://github.com/almarklein/wasmfun
- C/C++ or Rust -> wasm
- TypeScript

6.7.3 Other wallets

- React Native + Redux (Trustlines)
- Cordova, Ionic (LETH)
- Android native (Walleth)
- iOS + Android native (Trust Wallet)

6.8 Components

6.8.1 Raiden Light Client Library

- reusable
- non-mediating transfers
- IoT compatible
- can connect to hubs (Raiden Full Nodes) for off-chain & channel-related on-chain transactions
- can connect to a PFS
- can connect to a MS
- has APIs for the same TS used by RN
- uses the same types of messages as the Raiden Full Node (except those for mediating transfers)
- (possible) also communicates with the Relay Server for (at least) push notifications for off-chain payments / channel-related events.

6.8.2 Raiden Full Node

- either ran by BB or chosen from the network based on predefined logic / random (handled by the PFS)

6.8.3 On-Chain Client Library

- for normal on-chain transactions logic (except channel-related)
- wallet library (keystore, account management)
- communicates with the Relay Server

6.8.4 Relay Server

- will talk with an Ethereum Node for normal on-chain transaction needs (web3, RPC)
- push notifications server

6.8.5 Ethereum Node

- provides read & write access to the blockchain

6.9 MobileApp

6.9.1 Visuals

https://www.ethereum.org/images/logos/Ethereum_Visual_Identity_1.0.0.pdf

6.9.2 User Onboarding Flow Example

- install the app
- sign Terms of Use
- **import wallet / generate new wallet**
  - if importing a new wallet, off-chain data has to be retrieved (open channels, last balance proofs; maybe automatically add the channel 2nd parties to the address book)
- fund wallet with ETH (should be easy to copy/paste address or share)
- fund wallet with RDN / have an easy way to buy RDN from the app (agreement with an exchange or Vending-Machine)
- **choose automatically or show list of trustworthy hubs that have connections with the 3rd party services (settlement, path f**
  - prompt the user to choose one -> this means he has to put some tokens into escrow and pay some ETH, so he might not want to do that right away unless the hub is a goodwill hub and provides some funds himself
- if the user does have any channels open, he cannot make any transactions yet; a notification can be shown that he has not completed this step (e.g. action todo list)

- show a list of tokens that RN has in the registry -> show relevant tokens (high liquidity) + a search input
- prompt the user to choose token networks (he can join even without having any tokens in his wallet, because he can just receive tokens - tbd)
- when joining the token networks, the tokens should also be added for the on-chain transactions (seamless, user should not know the difference between on-chain / off-chain ; Raiden Network token registry should have an api for the token abis & addresses)
- user can deposit tokens to his wallet (easy way to copy/paste/share the address)
- prompt user to add contacts (address book) or share his address with others (link with an api that adds the address to the address book - will need the user approval in the app)

6.9.3 Transaction Flow Example

- choose contact from address book or paste and address one time
- use default on-chain/off-chain setting, but show the option in the transaction page with possibility to change it.
- **if off-chain -> check if there is a path to the contact / big enough capacity / or if he is connected to a hub -> if not, ask the u**
  - note - a hub might open channels himself, depending on his terms of service
  - yes -> open a channel, do the tx
  - no -> he can choose to do it on-chain

6.9.4 UI Features Example

About

- version
- Terms of Use
- License

Settings

- adding / removing custom token for on-chain transactions (address, name, token symbol, decimals)
- choosing between off-chain (default) and on-chain; this change can also be done in the payment flow if needed (e.g. no available channels, one time payment etc.)
- choosing currency to show along ETH / token values (BTC / USD / EUR / custom

Account

- wallet = 1 Ethereum address
- no registration or sign up; private keys remain with the user
- backup & restore wallet from seed words (BIP39 Mnemonic code)
- backup & restore wallet from private key / JSON file
- **generate new wallet**

    - pick account identicon
    - show seed words / recovery phrase
    - force user to select / write seed words
- download state logs per account (list transactions)
- share checksummed address via QRcode, SMS, Email, Whatsapp, Whisper (should be easy to use the shared address from inside the app)
- address book - custom address names & identicons
- **User Authentication**
    - uPort?
    - passcode, custom passphrase
    - **iOS:**
        * Touch ID for storing data securely using Secure Enclave chip
        * PIN code
        * FACE ID

Setup

- (probably not, but just mentioning it:) support for on-chain transactions targeting custom contracts (contract address, abi, assign name & identicon ; remove contract, UI for contract interface, notifications about contract events?)
- (possible) default token for paying 3rd party services / transaction gas

Channel info

- top up the channel
- close the channel & settle
- channel history - open, top ups, payments

On-chain transaction UI

- input: receiver address, ETH / tokens value, data (bytes), gas limit, gas price
- show: Max Transaction Fee, Max Total, Fiat equivalent in chosen currency

Off-chain transaction UI

- input: receiver, token type, amount of tokens, payment metadata for the receiver (ex. shopping cart items, order number etc)
- show: tbd

Hub reputation system (tbd)

- 3rd party services chosen automatically by reputation vs. manually by the user (or both)
- have a rating system for good hubs - count only the good feedback
- **feedback can be from:**
  - initial reputation deposit in the Raiden Network
  - other hubs with which the hub can gossip
  - users
- **feedback can be acquired:**
  - automatic metrics: response time after sending a request (have a time threshold over which the hub is awarded points), threshold for path length for PFS (shorter, the better)
  - manual rating system - users / other hubs can rate the hub

6.10 Protocols

6.10.1 Easy onboarding

- https://github.com/ethereum/EIPs/issues/865#issuecomment-362920866 pay with tokens for gas

6.10.2 Payment Requests

- https://github.com/ethereum/EIPs/pull/681 - Payment request URL specification for QR codes, hyperlinks and Android Intents. (the way to go)
- https://github.com/ethereum/EIPs/pull/831 - Extracting the container format from EIP681
- https://github.com/ethereum/EIPs/issues/67 - Standard URI scheme with metadata, value and byte code (IBAN) (outdated)
- https://github.com/ethereum/wiki/wiki/ICAP:-Inter-exchange-Client-Address-Protocol

6.10.3 Push Notifications

- webrtc, websockets
- https://medium.com/uport/adventures-in-decentralized-push-notifications-3c64e700ec18 , https://github.com/uport-project
- https://github.com/walleth/walleth-push - Service that watches one ethereum-node via RPC and triggers FCM pushes when registered addresses have new transactions; uses https://firebase.google.com/docs/cloud-messaging (iOS, Android, JavaScript)
- https://github.com/status-im/status-go/wiki/Whisper-Push-Notifications
- polling (LETH)

6.10.4 Other

- https://github.com/ethereum/go-ethereum/wiki/Mobile:-Account-management
- https://github.com/ethereum/go-ethereum/wiki/Mobile%3A-Introduction
- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md - address checksums

6.10.5 Existing Tools/Services

Wallet

- https://github.com/ConsenSys/eth-lightwallet - Lightweight JS Wallet for Node and the browser
- https://github.com/petejkim/wallet.ts - Utilities for cryptocurrency wallets, written in TypeScript
- https://github.com/TrustWallet/trust-keystore

Wallet SC

- https://github.com/gnosis/MultiSigWallet (old one)
- https://github.com/gnosis/gnosis-safe-contracts (new)

Account identity

- https://www.uport.me/
- https://github.com/ethereum/blockies

Event Watching

- https://infura.io/
- https://etherscan.io/apis#logs
- Eth.Events

6.11 Roadmap

(purely estimative)

- Finalize feature specs (5 PD)
- Finalize protocols and standards research (+ competition research) (5 PD)
- Align with Raiden Network after core, MS, PFS specs are somewhat finalized (4 PD)
- Plan milestones (4 PD)
- Prototype (to test chosen frameworks - native vs. progressive apps etc.) (7 PD)
- Prototype 2 - standard wallet implementation (10 PD)
- Prototype 3 - add off-chain logic (15PD)
- MVP - off-chain + on-chain (15 PD)

6.12 Issues to clarify on

- 3rd party APIs
- onboarding
- seamlessly switch from off-chain to on-chain and when (no hub available etc.)
- see overlap with uRaiden and make a first usable version for it if possible (not sacrificing the architecture - which should be made with RN in mind)
- build a micropayments-only wallet first? (advantages: lowers complexity for IoT support)

6.13 Other wallets:

- https://www.cipherbrowser.com/ (iOS, Android), https://github.com/petejkim/cipher-ethereum - ETH, ERC20 tokens; dapp browser, FACE ID, support for main net and test nets
- **https://github.com/inzhoop-co/LETH (cross-platform)**
  - ETH, ERC20 tokens
  - Set host node address private/test/public
  - List your transactions
  - Share Address via SMS, Email or Whisper v5 (Shh)
  - Share your geolocation
  - Request payments via SMS, Email or Whisper (Shh)
  - Send messages / images to friends and community using Whisper protocol in unpersisted chat
  - Send private unpersisted crypted messages to friends
  - Backup / Restore wallet using Mnemonic passphrase
  - Protect access with TouchID / PIN code
  - Currency convertion value via Kraken API
  - Add Custom Token and Share it with friends
  - Run DAppLeth (Decentralized external dapps embedded at runtime)
- https://github.com/walleth (Android)
- https://www.toshi.org/ (iOS, Android)
- https://github.com/status-im (iOS, Android)
- https://github.com/TrustWallet (iOS, Android)
- **https://github.com/manuelsc/Lunary-Ethereum-Wallet (Android)**
  - uses Etherscan API for notifications - https://github.com/manuelsc/Lunary-Ethereum-Wallet/blob/3553765fb1a1cd7a9d6cae3badbdd66ab00b7061/app/src/main/java/rehanced/com/simpleetherwallet/services/TransactionService.java
  - ETH & tokens
  - Multi wallet support
  - Support for Watch only wallets

  - Notification on incoming transactions
  - Combined transaction history
  - Addressbook and address naming
  - Importing / Exporting wallets
  - Display amounts and token in ETH, USD or BTC
  - No registration or sign up required
  - Price history charts
  - Fingerprint / Password protection
  - ERC-67 and ICAP Support
  - Adjustable gas price with minimum at 0.1 up to 32 gwei
  - Supporting 8 Currencies: USD, EUR, GBP, CHF, AUD, CAD, JPY, RUB
  - Available in English, German, Spanish, Portuguese and Hungarian
- https://token.im/ (iOS, Android) ; https://github.com/consenlabs
- https://jaxx.io (iOS, Android, OSX, Linux, Windows, Web) - multiple currencies
- https://freewallet.org/currency/eth (iOS, Android)
- **https://www.blockwallet.eu/ ; https://github.com/cybertim/blockwallet**
  - Signs transactions on the device itself
  - Sends signed transactions through SSL to a secured RPC Geth server
  - SSL Server Certificate Fingerprint check implemented to warn about MITM Proxys (compromised networks)
  - AES Encryption on Private Key with custom Passcode, only decoded when needed
  - All Data stored in AES128 Encrypted container Stanford Javascript Crypto Library
  - Uses BIP39 Mnemonic code for Recovery of Private Keys
  - Implemented EIP55 capitals-based checksum on send addresses
  - Using QR Codes and Scanner with checksum to prevent typo errors
- https://eidoo.io/ (iOS, Android) - BTC, ETH, ERC20, atomic swap transactions, ICO manager
- https://wallet.mycelium.com/ (iOS, Android) - BTC wallet
- https://vynos.tech/ (in-browser, OFF-CHAIN)
- https://github.com/ethereum/mist (OSX, Linux, Windows)
- https://www.myetherwallet.com/ (web)
- https://www.exodus.io/ (OSX, Linux, Windows)
- https://electrum.org/#home (lightning) (Android, OSX, Linux, Windows)
- https://github.com/LN-Zap (lightning) (OSX, Linux, Windows)

CHAPTER 7

Raiden Terminology

Additional Hash

additional_hash Hash of additional data (in addition to a balance proof itself) used on the Raiden protocol (and potentially in the future also the application layer). Currently this is the hash of the offchain message that contains the balance proof. In the future, for example, some form of payment metadata can be hashed in.

balance proof

Participant Balance Proof

BP Signed data required by the *Payment Channel* to prove the balance of one of the parties. Different formats exist for offchain communication and onchain communication. See the *onchain balance proof definition* and *offchain balance proof definition*.

Bidirectional Payment Channel Payment Channel where the roles of *Initiator* and *Target* are interchangeable between the channel participants.

Capacity Current amount of tokens available for a given participant to make transfers.

chain id Chain identifier as defined in EIP155.

Challenge Period The state of a channel initiated by one of the channel participants. This phase is limited for a period of `n` block updates.

Challenge Period Update Update of the channel state during the *Challenge period*. The state can be updated either by the channel participants, or by a delegate (*MS*).

channel identifier Identifier assigned by *Token Network* to a *Payment Channel*. Must be unique inside the *Token Network* contract. See the *implementation definition*.

cooperative settle proof Signed data required by the *Payment Channel* to allow *Participants* to close and settle a *Payment Channel* without undergoing through the *Settlement Window*. See the *message definition*.

Deposit Amount of token locked in the contract.

Fee Model Total fees for a Mediated Transfer announced by the Raiden Node doing the Transfer.

Hash Time Lock

HTL An expirable lock locked by a secret.

Hash Time Locked Transfer

Mediated Transfer A token Transfer composed of multiple HTL transfers.

HTL Commit The action of asking a node to commit to reserving a given amount of token for a *Hash Time Lock*. This is the message used to find a path through the network for a transfer.

HTL Transfer An expirable potentially cancellable Transfer secured by a Hash Time Lock.

HTL Unlock The action of unlocking a given *Hash Time Lock*. This is the message used to finalize a transfer once the path is found and the reserve is acknowledged.

Inbound Transfer A locked transfer received by a node. The node may be a *Mediator* in the path or the *Target*.

Initiator The node that sends a *Payment*.

lock expiration The lock expiration is the highest block_number until which the transfer can be settled.

locked amount Total amount of tokens locked in all currently pending *HTL* transfers sent by a channel participant. This amount corresponds to the *locksroot* of the HTL locks.

lockhash The hash of a lock. `sha3_keccack(lock)`

Mediator A node that mediates a transfer.

merkletree root

locksroot The root of the merkle tree which holds the hashes of all the locks in the channel.

Message Any message sent from one Raiden Node to the other.

Monitoring Service

MS The service that monitors channel state on behalf of the user and takes an action if the channel is being closed with a balance proof that would violate the agreed on balances. Responsibilities - Watch channels - Delegate closing

Net Balance Net of balance in a contract. May be negative or positive. Negative for `A(B)` if `A(B)` received more tokens than it spent. For example `net_balance(A) = transferred_amount(A) - transferred_amount(B)`

nonce Strictly monotonic value used to order off-chain transfers. It starts at `1`. It is a *balance proof* component. The `nonce` differentiates between older and newer balance proofs that can be sent by a delegate to the *Token Network* contract and updated through *updateNonClosingBalanceProof*.

Off-Chain Payment Channel The portion of a Payment Channel that is used by applications to perform payments without interacting with a blockchain.

Outbound Transfer A locked transfer sent by a node. The node may be a *Mediator* in the path or the *Initiator*.

Participants The two nodes participating in a *Payment Channel* are called the channel's participants.

Partner The other node in a channel. The node with which we have an open *Payment Channel*.

Pathfinding Service A centralized path finding service that has a global view on a token network and provides suitable payment paths for Raiden nodes.

payment The process of sending tokens from one account to another. May be composed of multiple transfers (Direct or HTL). A payment goes from *Initiator* to *Target*.

payment channel An object living on a blockchain that has all the capabilities required to enable secure off-chain payment channels.

Payment Receipt TBD

Raiden Channel The Payment Channel implementation used in Raiden.

Raiden Light Client A client that does not mediate payments.

Raiden Network A collection of Token networks.

Receiver The node that is receiving a Message.

Reveal Timeout The number of blocks in a channel allowed for learning about a secret being revealed through the blockchain and acting on it.

Secret A value used as a preimage in a *Hash Time Locked Transfer*. Its size should be 32 bytes.

secrethash The hash of a *secret*. `sha3_keccack(secret)`

Sender The node that is sending a Message. The address of the sender can be inferred from the signature.

Settle Expiration The exact block at which the channel can be settled.

Settlement Window

Settle Timeout The number of blocks from the time of closing of a channel until it can be settled.

Sleeping Payment A payment received by a *Raiden Light Client* that is not online.

Target The node that receives a *Payment*.

Token Network A network of payment channels for a given Token.

Token Swaps Exchange of one token for another.

Transfer A movement of tokens from a *Sender* to a *Receiver*.

Transferred amount Monotonically increasing amount of tokens transferred from one Raiden node to another. It represents all the finalized transfers. For the pending transfers, check *locked amount*.

Unidirectional Payment Channel Payment Channel where the roles of *Initiator* and *Target* are determined in the channel creation and cannot be changed.

withdraw proof

Participant Withdraw Proof Signed data required by the *Payment Channel* to allow a participant to withdraw tokens. See the *message definition*.

Index

A

Additional Hash, **63**
additional_hash, **63**

B

balance proof, **63**
Bidirectional Payment Channel, **63**
BP, **63**

C

Capacity, **63**
chain id, **63**
Challenge Period, **63**
Challenge Period Update, **63**
channel identifier, **63**
cooperative settle proof, **63**

D

Deposit, **63**

F

Fee Model, **63**

H

Hash Time Lock, **63**
Hash Time Locked Transfer, **64**
HTL, **63**
HTL Commit, **64**
HTL Transfer, **64**
HTL Unlock, **64**

I

Inbound Transfer, **64**
Initiator, **64**

L

lock expiration, **64**
locked amount, **64**
lockhash, **64**
locksroot, **64**

M

Mediated Transfer, **64**
Mediator, **64**
merkletree root, **64**
Message, **64**
Monitoring Service, **64**
MS, **64**

N

Net Balance, **64**
nonce, **64**

O

Off-Chain Payment Channel, **64**
Outbound Transfer, **64**

P

Participant Balance Proof, **63**
Participant Withdraw Proof, **65**
Participants, **64**
Partner, **64**
Pathfinding Service, **64**
payment, **64**
payment channel, **64**
Payment Receipt, **64**

R

Raiden Channel, **64**
Raiden Light Client, **65**
Raiden Network, **65**
Receiver, **65**
Reveal Timeout, **65**

S

Secret, **65**
secrethash, **65**
Sender, **65**

Settle Expiration, 65
Settle Timeout, 65
Settlement Window, 65
Sleeping Payment, 65

T

Target, 65
Token Network, 65
Token Swaps, 65
Transfer, 65
Transferred amount, 65

U

Unidirectional Payment Channel, 65

W

withdraw proof, 65