Raiden Specification Documentation

Release 0.1

Brainbot

Feb 21, 2019

Contents:

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))
```

where `additional_hash` is the hash of the whole message being signed.

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
additional_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

This should correspond to the packed format of LockedTransfer.

Field Name	Field Type	Description
command_id	one byte	Value 7 indicating `LockedTransfer`
pad	three bytes	Contents ignored
nonce	uint64	See *Offchain Balance Proof*
chain_id	uint256	See *Offchain Balance Proof*
message_identifier	uint64	An ID for `Delivered` and `Processed` acknowledgments
payment_identifier	uint64	An identifier for the payment that the initiator specifies
expiration	uint256	See *HashTimeLock*
token_network_address	address	See `token_network_id` in *Offchain Balance Proof*
token	address	Address of the token contract
channel_identifier	uint256	See *Offchain Balance Proof*
recipient	address	Destination for this hop of the transfer
target	address	Final destination of the payment
initiator	address	Initiator of the transfer and party who knows the secret
locksroot	bytes32	See *Offchain Balance Proof*
secrethash	bytes32	See *HashTimeLock*
transferred_amount	uint256	See *Offchain Balance Proof*
locked_amount	uint256	See *Offchain Balance Proof*
amount	uint256	Transferred amount including fees. See *HashTimeLock*
fee	uint256	Total available fee for remaining mediators
signature	65 bytes	Computed as in *Offchain Balance Proof*

The sender of the message should be computable from `signature` so is not included in the message.

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 have initiated a payment to the *target* with the same `payment_identifier`, `lock_secrethash`, `payment_amount` and `expiration`.
- The *target* must have received a *Locked Transfer* for the payment.
- The `signature` must be from the *target*.

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 Reveal Secret

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

This should match the encoding implementation.

Field Name	Field Type	Description
cmdid	one byte	Value 11 (indicating `Reveal Secret`)
pad	three bytes	Ignored
message identifier	uint64	An ID use in `Delivered` and `Processed` acknowledgments
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 *Reveal Secret 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

This should match the Secret message in encoding/messages file.

Field Name	Field Type	Description
command id	one byte	Value 4 indicating Unlock
padding	three bytes	Ignored
chain identifier	uint256	See *Offchain Balance Proof*
message identifier	uint64	An ID used in `Delivered` and `Processed` acknowledgments
payment identifier	uint64	An identifier for the *Payment* chosen by the *Initiator*
token network identifier	address	See *Offchain Balance Proof*
lock_secret	bytes32	The secret that unlocked the lock
nonce	uint64	See *Offchain Balance Proof*
channel identifier	uint256	See *Offchain Balance Proof*
transferred amount	uint256	See *Offchain Balance Proof*
locked amount	uint256	See *Offchain Balance Proof*
lockedsroot	bytes32	See *Offchain Balance Proof*
signature	bytes	See *Offchain Balance Proof*. Note `additional_hash` is the hash of the whole message

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:

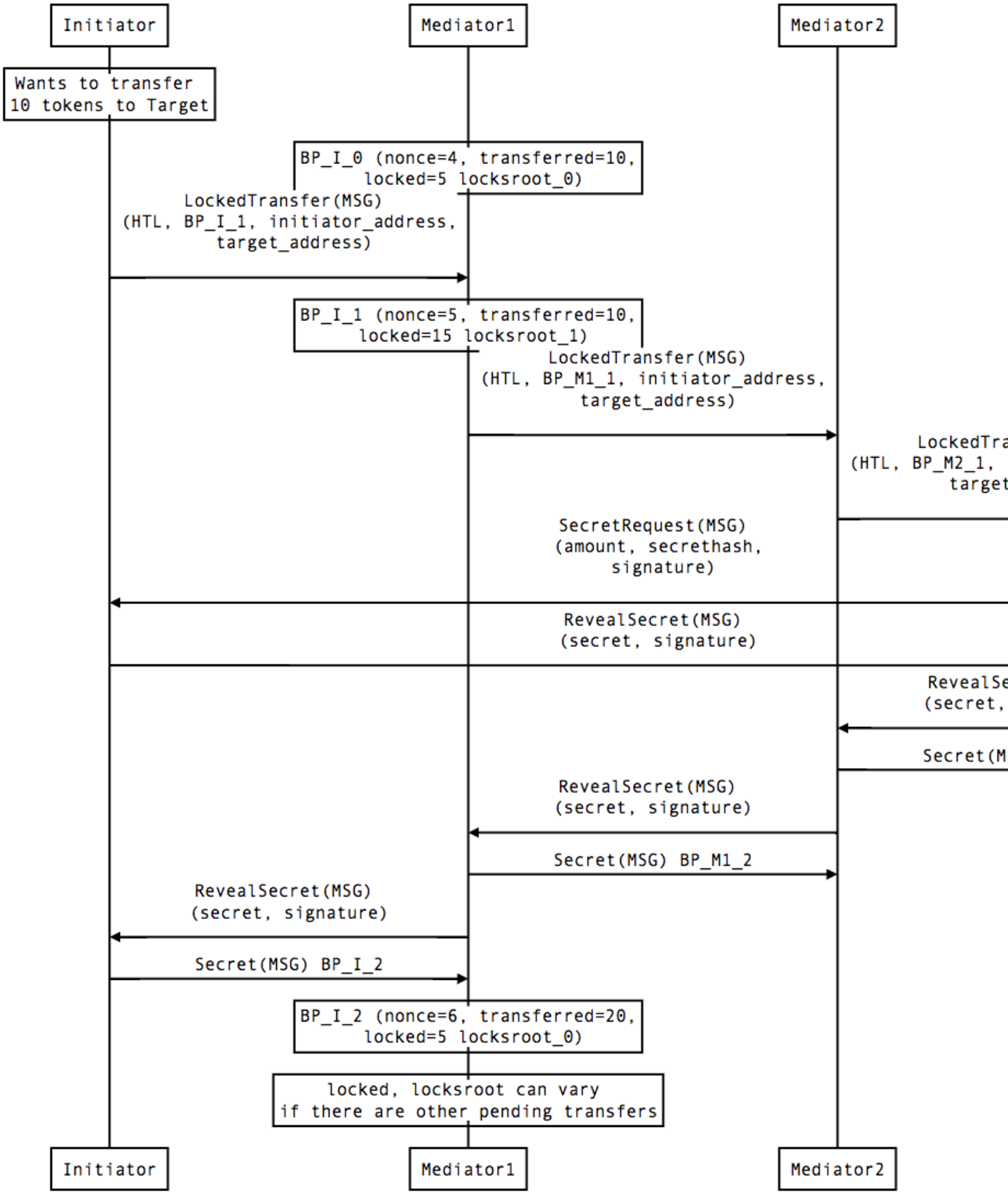
- `N + 1` *locked transfer* or *refund transfer* messages
- `1` *secret request*
- `N + 2` *reveal secret*
- `N + 1` *unlock*
- `2N + 3` processed (one for everything above)
- `5N + 8` delivered

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.

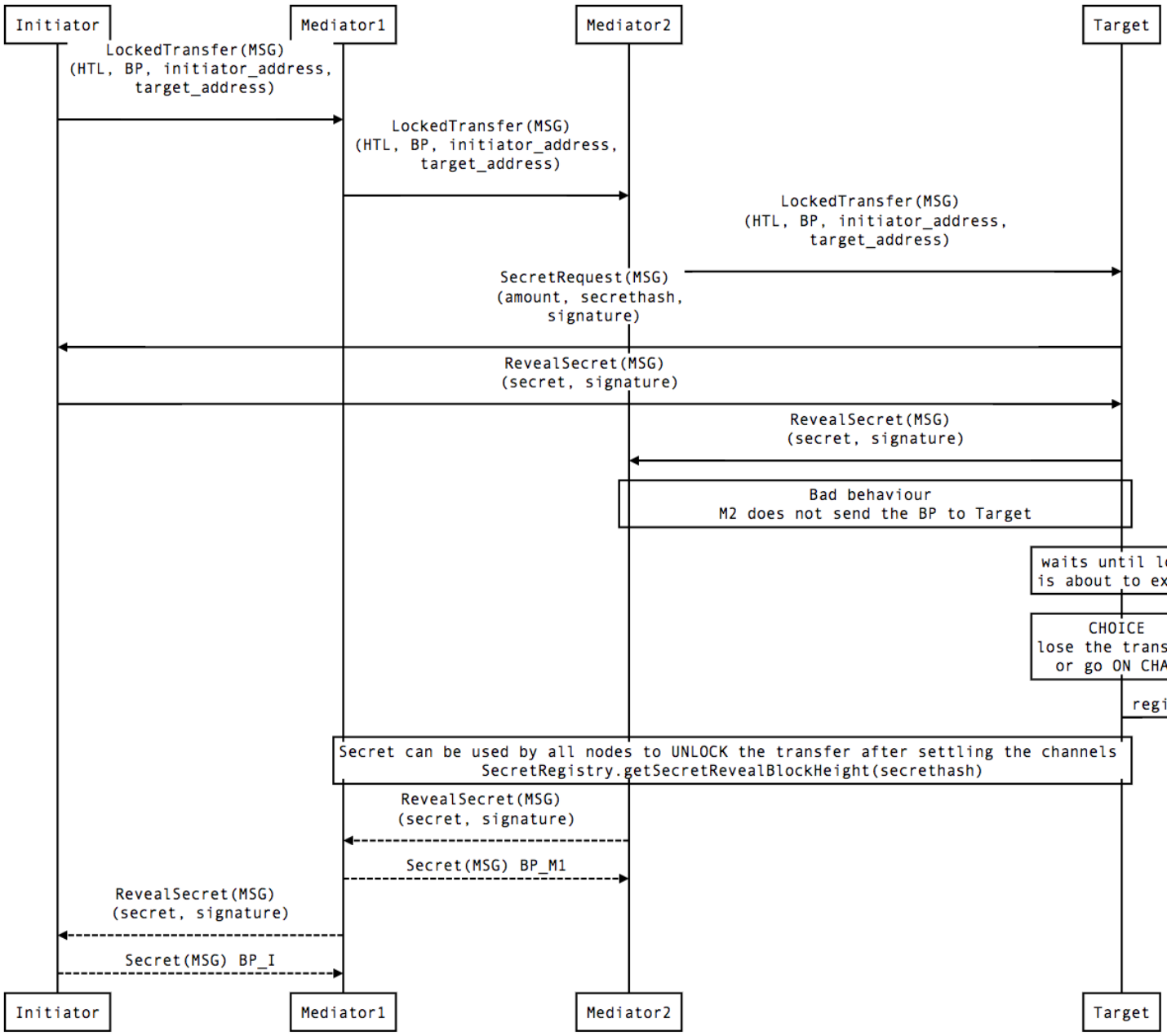
Mediated Transfer - Best Case Scenario

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 operator registered with the Raiden *ServiceRegistry* 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`: String of the 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))
```

(continues on next page)

(continued from previous page)

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

This contract creates and remembers a TokenNetwork contract for an ERC20 Token. Raiden clients listen to `TokenNetworkCreated` events so they can notice when this contract deploys a new `TokenNetwork`.

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`.
- `settle_block_number`: the number of blocks in the *challenge period* if `state` is `Opened`; the block number after which settleChannel() can succeed if `state` is `Closed`; 0 otherwise.

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`: The amount of tokens that the `participant` has deposited through *setTotalDeposit()*. 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.

Before calling `setTotalDeposit()`, the caller needs to send the `approve` transaction on the ERC20 token contract so that the `TokenNetwork` contract can make the token transfer for the channel deposit.

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 the balance proof counting towards the non-closing participant

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,
```

(continues on next page)

(continued from previous page)

```
    address non_closing_participant,
    bytes32 balance_hash,
    uint256 nonce,
    bytes32 additional_hash,
    bytes closing_signature,
    bytes non_closing_signature
)
    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,
    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`).
- `participant1_amount`: the amount of tokens sent to `participant1` at the end of the settlement.
- `participant2_amount`: the amount of tokens sent to `participant2` at the end of the settlement.

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,
    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. Currently, `registerSecret()` fails if the `secret` is zero.

- `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.

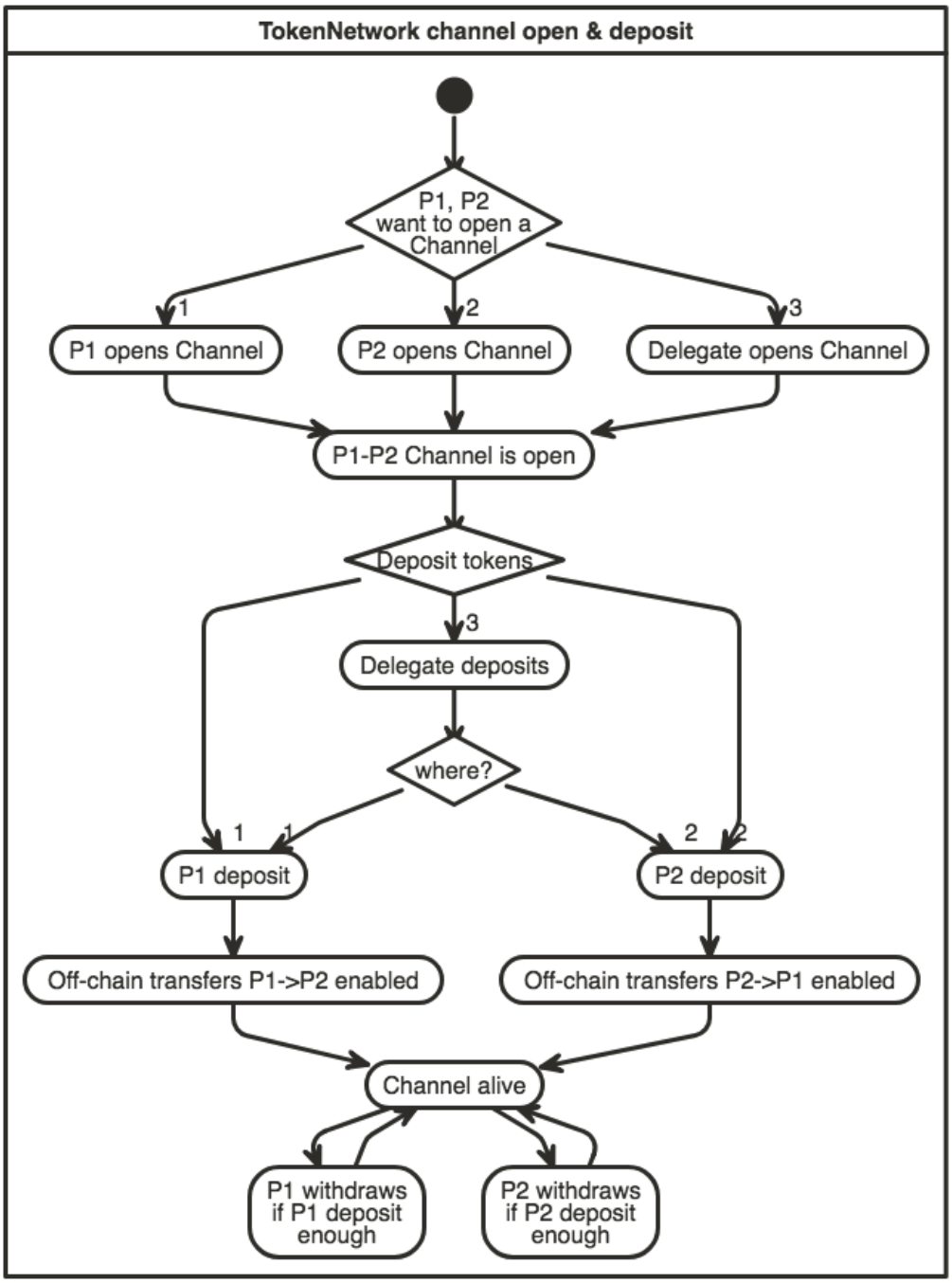
```
function findEndpointByAddress(address eth_address)
    public
    view
    returns (string endpoint)
```

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

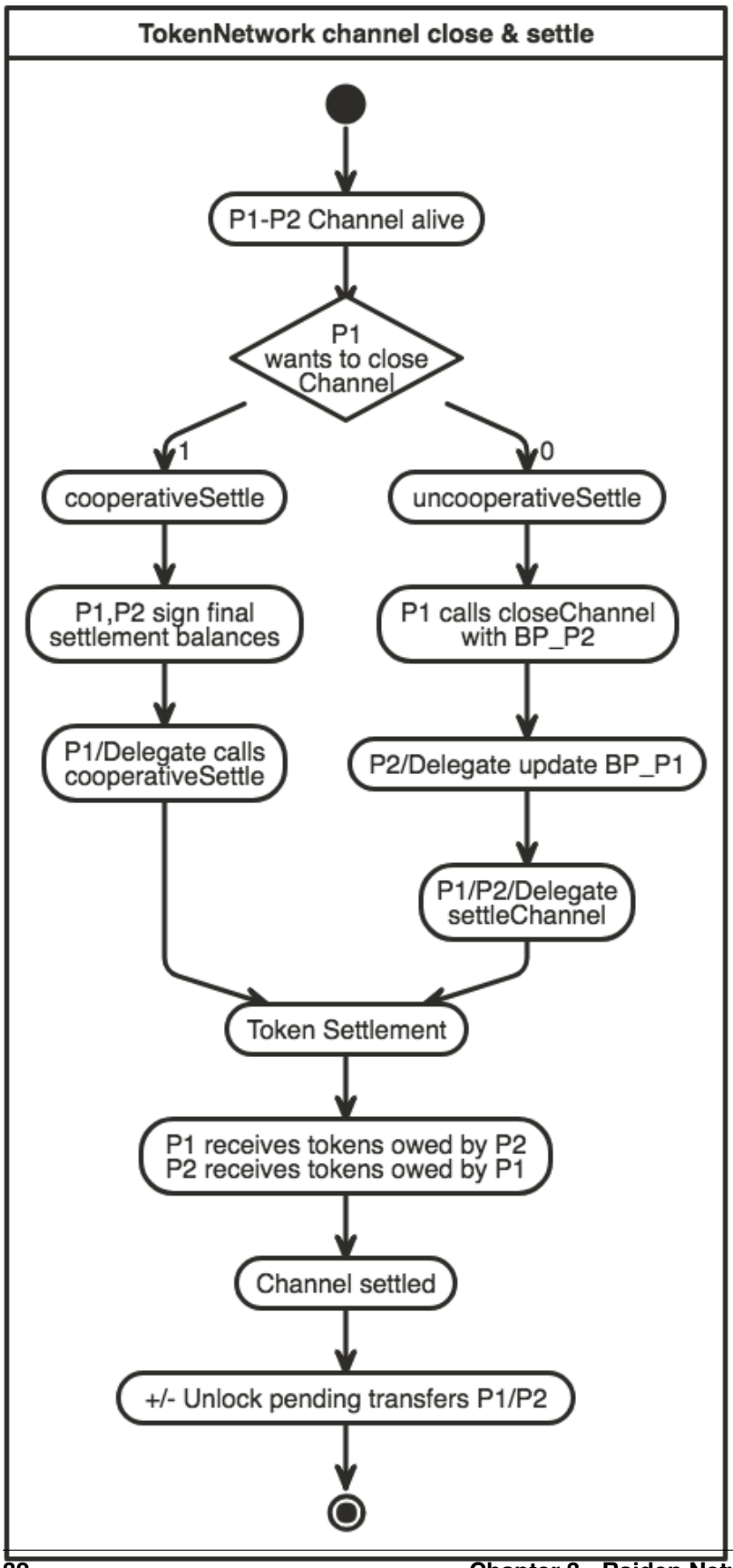
3.6 TokenNetwork Channel Protocol Overview

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

3.6.1 Opened Channel Lifecycle

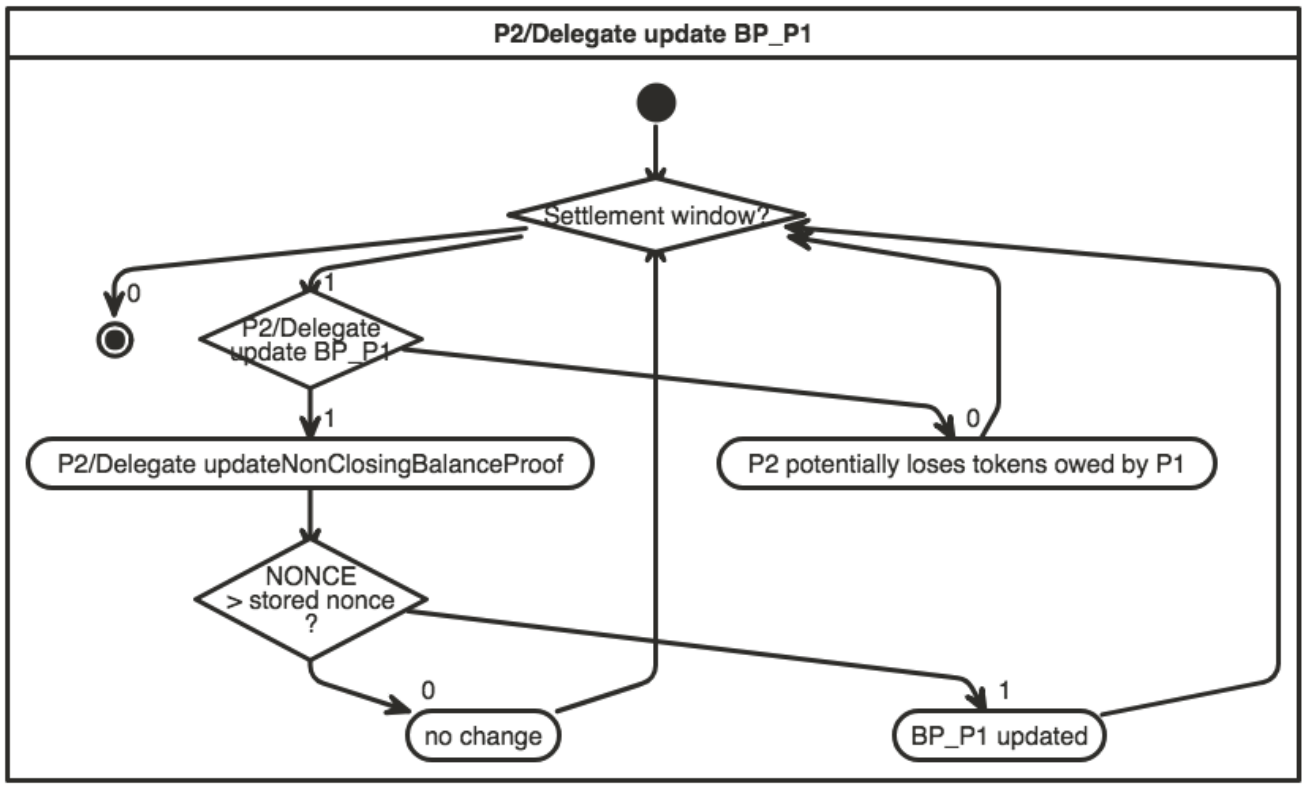


3.6.2 Channel Settlement

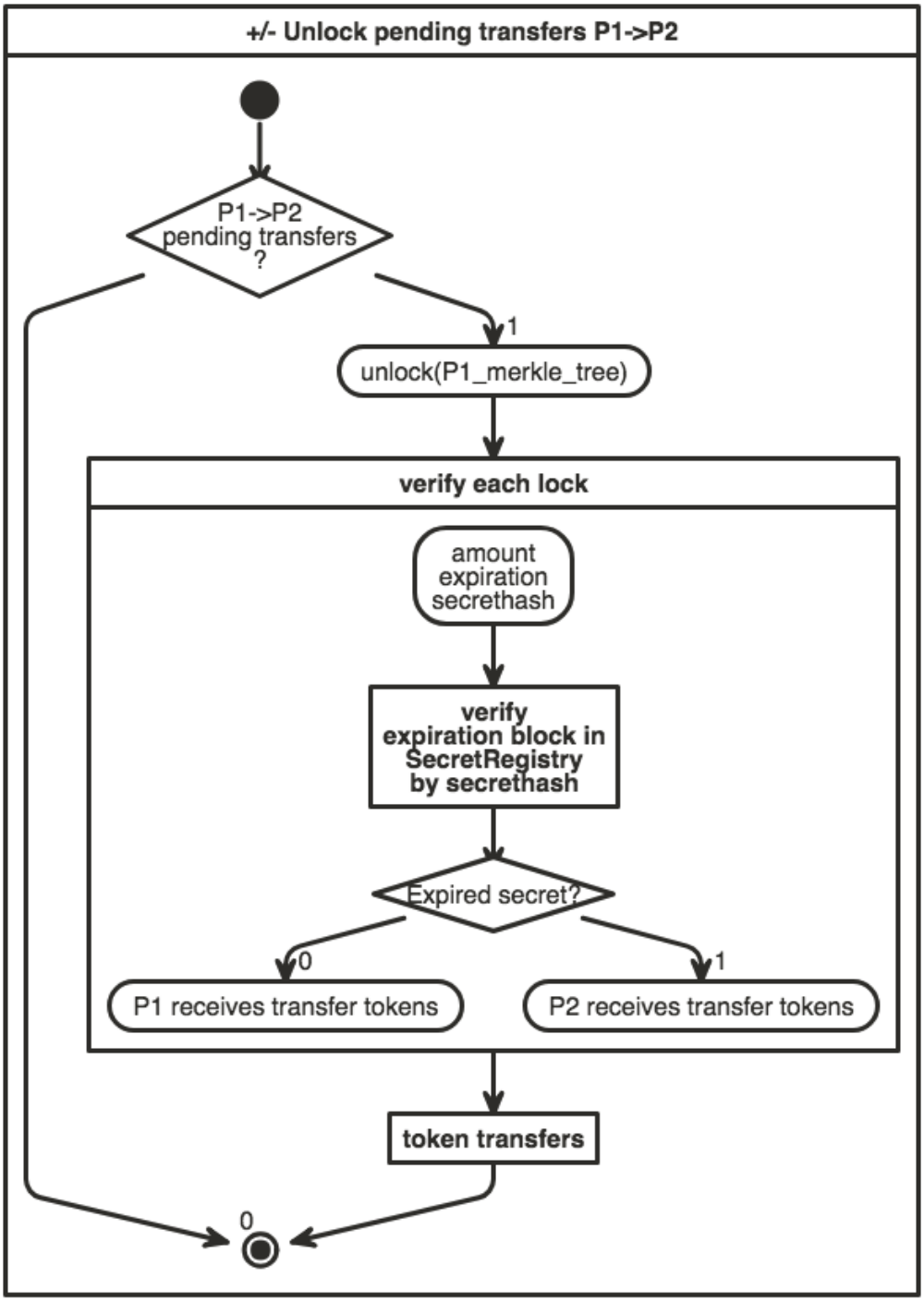


3.6.3 Channel Challenge Period

The non-closing participant can update the closing participant's balance proof during the challenge period, 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

Smart Contracts for Raiden Services

4.1 Overview

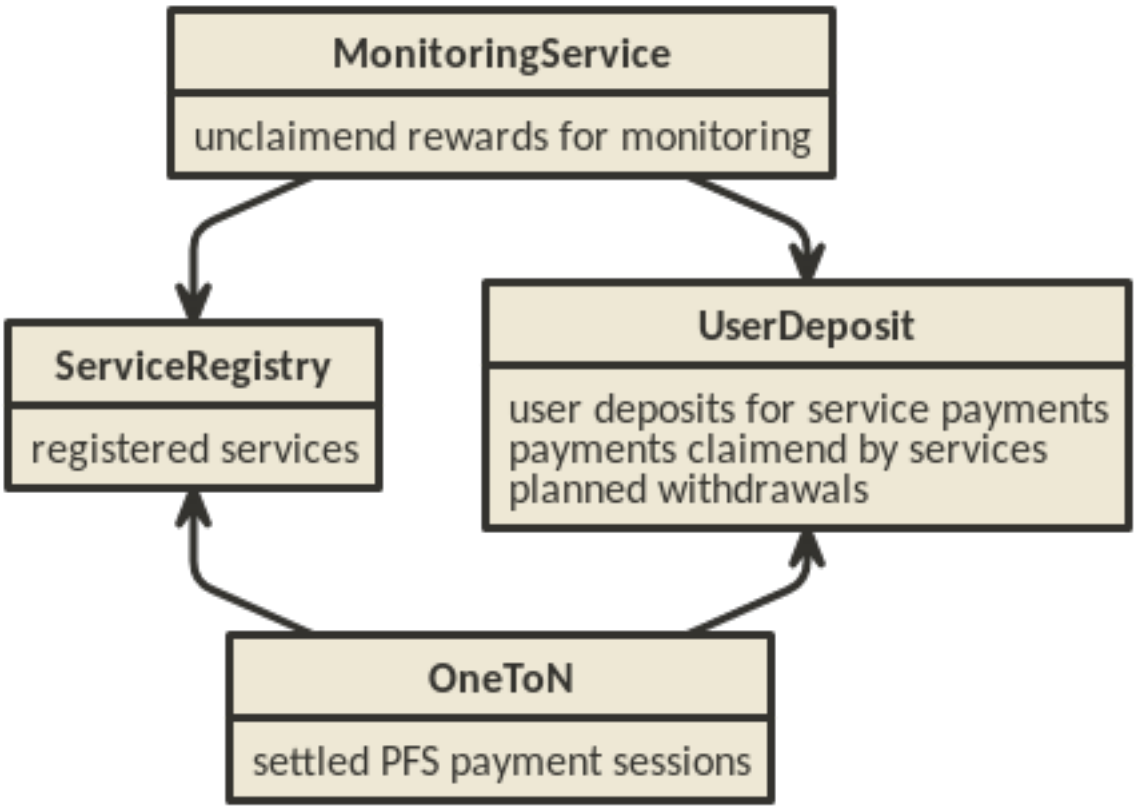
The Raiden services (*Raiden Monitoring Service* and *Raiden Pathfinding Service*) require a set of smart contracts to function. There are two general smart contracts:

- UserDeposit
- ServiceRegistry

and one additional contract for each of the services

- MonitoringService, used as integral part of how the MS functions
- OneToN, a minimal one-to-N payment solution used to pay fees to the PFS

which depend on the former two contracts.



There might also be an additional contract to facilitate the onboarding of new Raiden users, which has been called "Hub Contract" in some discussions. There are no detailed plans for that contract, yet.

4.2 ServiceRegistry

The ServiceRegistry provides a registry in which services have to register before becoming a full part of the Raiden services system. Services have to deposit RDN tokens in this contract for a successful registration. This avoids attacks using a large number of services and increases the incentive for service provider to not harm the Raiden ecosystem.

4.3 UserDeposit

The Raiden services will ask for payment in RDN. The Monitoring Service and the Pathfinding Service require deposits to be made in advance of service usage. These deposits are handled by the User Deposit Contract. Usage of the deposit for payments is not safe from double spending, but measures can be taken to reduce the likelihood to acceptable levels. This is a good trade off as long as the money lost on double spending is less than the savings in gas cost.

4.3.1 Requirements

- Users can deposit and withdraw tokens.
- Tokens can be deposited to the benefit of other users. This could facilitate onboarding of new Raiden users and allow a MS to defer the monitoring to another MS.

- Tokens can't be withdrawn immediately, but only after a certain delay. This allows services to claim their deserved payments before the withdraw takes place.
- Services can read the effective balance of a user (current balance - planned withdrawals)
- Service contracts are trusted and can claim tokens for the service providers.
- Services can listen to events which notify them of decreasing user balances. A service can then claim payments before double spending becomes too likely.

4.3.2 Use cases

Monitoring Service rewards

The MS is promised a reward for each settlement in which it took part on behalf of the non-closing participant. Before accepting a monitor request, the MS checks if enough tokens are deposited in the UDC. The MS that has submit the latest BP upon settlement will receive the promised tokens on it's UDC balance.

1-n payments

The PFS will be paid with signed IOUs, roughly a simplified uRaiden adapted to 1-n payments. The IOU contains the amount of tokens that can be claimed from the signer's UDC balance. See *OneToN* for details.

4.4 OneToN

4.4.1 Overview

The OneToN contract handles payments for the PFS. It has been chosen with the following properties in mind:

- easy to implement
- low initial gas cost even when fees are paid to many PFSs
- a certain risk of double spends is accepted

The concept is based on the idea to use a user's single deposit in the UDC as a security deposit for off-chain payments to all PFSs. The client sends an IOU consisting of (sender, receiver, amount, expiration, signature) to the PFS with every path finding request. The PFS verifies the IOU and checks that `amount >= prev_amount + pfs_fee`. At any time, the PFS can claim the payment by submitting the IOU on-chain. Afterwards, no further IOU with the same (sender, receiver, expiration) can be claimed.

Related:

- https://github.com/raiden-network/team/issues/257
- https://github.com/raiden-network/team/issues/256
- https://gist.github.com/heikoheiko/214dbbd954e0f97e0e13b2fefdc7c753

4.4.2 Requirements

- low latency (<1s)
- reliability, high probability of success (P > 0.99)
- low cost overhead (<5% of transferred amount)

- low fraud rate (< 3%, i.e. some fraud is tolerable)
- can be implemented quickly

4.4.3 Communication between client and PFS

When requesting a route, the IOU is added as five separate arguments to the existing HTTP params.

Field Name	
sender	
receiver	
amount	
expiration_block	
signature	
none	
	The *Raiden Monitoring Service* submits an up-to-date *balance proof* on behalf of users who are offline when a channel

contract **MonitoringService** is Utils

Notice Called by a registered MS, when providing a new balance proof to a monitored channel. Can be called multiple times by different registered MSs as long as the BP provided is newer than the current newest registered BP.

Parameters

- **nonce** – Strictly monotonic value used to order BPs omitting PB specific params, since these will not be provided in the future
- **reward_amount** – Amount of tokens to be rewarded
- **token_network_address** – Address of the Token Network in which the channel being monitored exists.
- **reward_proof_signature** – The signature of the signed reward proof

function **claimReward**(*uint256 channel_identifier*, *address token_network_address*, *address clos-public ing_participant*, *address non_closing_participant*)

Notice Called after a monitored channel is settled in order for MS to claim the reward Can be called once per settled channel by everyone on behalf of MS

Parameters

- **token_network_address** – Address of the Token Network in which the channel
- **closing_participant** – Address of the participant of the channel that called close
- **non_closing_participant** – The other participant of the channel

CHAPTER 5

Raiden Monitoring Service

5.1 Summary

- Monitoring Service (MS) listens to blinded Balance Proofs (BP) and proposed service fees in a public Matrix Room
- Any pre-registered MS can decide to monitor a channel and store corresponding BPs
- Then, whenever a channel is closed by calling `closeChannel` and after a period of x blocks where the client is expected the react, the MS will call `updateNonClosingBalanceProof` with the submitted BP by its client
- Service fees are paid in RDN, there is no free tier
- In the short term we go for a simple design which will allow us to reach the Ithaca milestone earlier
- In the long-term we see the PISA approach as more economically viable and user friendly, but this design requires additional features and can be developed independently after Ithaca

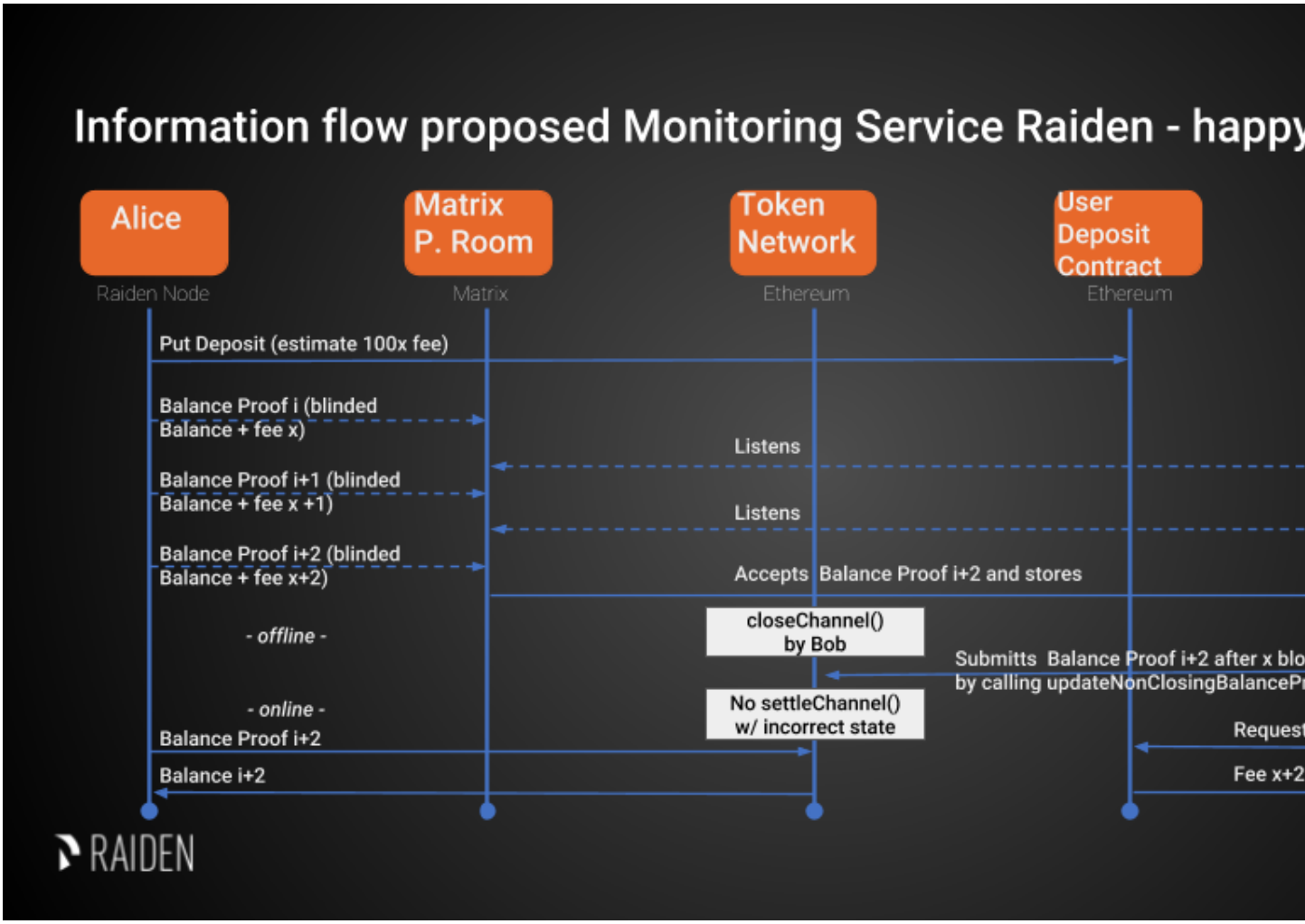
5.2 Usual Scenario

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. Whenever Alice wants to get her channel monitored, Alice publishes her *Monitor Request* - including a blinded balance proofs - into a public chat room
2. Updates can be sent by Alice whenever the state or proposed fee changes
3. MSs can pick up these messages and then listen to `ChannelClosed` events regarding this particular channel
4. If so, whenever the channel is closed and before the settlement period has ended, the MS will call the `updateNonClosingBalanceProof` function with the provided information, after a period of x blocks where the client is expected to react. This way the settleChannel function that calculates the token distribution can only be executed submitting the corresponding balance values of the hashed balances provided before

5. Either Alice or Bob (or anyone else) can now anytime call `settleChannel` and initiate the token distribution
6. After a successful submission of the hashed balance by the MS calling `updateNonClosingBalanceProof` (might be a defeat of an attack), the monitoring service gets an on-chain payment from a smart contract where Alice deposited some tokens beforehand
7. This will result in a correct distribution of tokens, in accordance with the off-chain transfers

5.3 Information Flow



5.4 General Requirements for the Design of the Monitoring Service

- Sybil Attack resistance (i.e. no one should be able to announce an unlimited number of possibly faulty services)
- Some degree of redundancy (ability to register a balance proof with multiple competing monitoring services)

5.5 Design of the Monitoring Service (still work in progress)

5.5.1 Monitoring Service Registration

The Monitoring Service has to be registered in the *ServiceRegistry* contract. Registry slots will be auctioned. If chosen in the auction, the service provider will become part of the list of MSs and must deposit some RDN.

5.5.2 Client Onboarding

Clients that want to request this service have to deposit an amount of RDN into the UDC in order to provide rewards to the MS. This happens during the general Raiden onboarding process, so that no additional preparation is necessary when usage of a MS is desired.

5.5.3 Service Discovery

All MS listen to a public Matrix room. Monitor Requests are broadcast and no specific MSs are appointed. The MSs can also publish their expected rewards in this room, which does not provide any guarantees, but increases the chance of reliable monitoring if both parties cooperate.

5.5.4 Monitoring Service Payment

The MS is paid after successfully submitting its client's balance proof update. The payment is paid out from a deposit in the User Deposit Contract (UDC). Ideally, only one MS submits the latest BP to the SC to avoid unnecessary gas usage. This can be made more likely by choosing the rewarded MS based on a function of the MS's address and the current block number. MSs which have a low f(address, block_num) would be incentivized to wait for a block number which yields a higher f for them, since they would probably lose out to another MS if they submitted the BP during the current block. Incentivizing MSs to wait in some cases greatly reduces the number of MSs submitting BPs simultaneously.

5.5.5 Ensuring MS Reliability

The MS has an incentive to intervene in case of a dispute, since it is only paid in that case. There are no incentives for a high level of reliability and the client knows neither how many MSs are monitoring his channel nor how reliable they are. These tradeoffs are made to favor simplicity of implementation.

5.5.6 Privacy

The Recipient and the actual transferred amounts amount are hidden by providing a hashed balance proof (or state). This provides some sort of privacy even if it can potentially be recalculated.

5.6 Security Analysis (inspired by PISA)

5.6.1 State Privacy

Blinded BPs are published to the MS as part of the Monitor Request in the matrix room and then submitted to the smart contract.

5.6.2 Fair Exchange

Clients can freely choose the reward for the MS, so it is easy for him to choose the amount in a way that makes the exchange attractive for himself. The client can't know if a MS started monitoring his payment channel, so he can't use such feedback to arrive at a reward where he knows that the deal is attractive for both him and the MS. Neither can he recognize if there is no such possible reward. The MS on the other hand can freely choose to ignore requests when the reward is too low, so he will only choose requests that he deems fairly rewarded. If the MS ignores the client's request, the client keeps his deposit and it can be used by other MSs or for later BPs. In summary, the exchange is fair for both parties, but there is a high likelihood that no exchange will happen at all.

5.6.3 Non-frameability

MSs can put the clients channel deposit at risk by ignoring all client requests. But since a MS can't force other MSs to ignore client requests, this can not be considered as framing. When only a single MS is monitoring the channel, the MS's dispute intervention and the reward payment happen atomically inside the SC. In this case, no party can frame the other.

When multiple MSs try to settle the same dispute, only the first one doing so receives a reward, but all of them have to invest resources to monitor the channel and spend gas to interact with the SC. If you find a way to continuously front run other MSs, you can drain their resources and block their only income. However, while doing so you fulfilled the MS's duty to settle the payment channel correctly and protect the client's deposit. In the short run, this is an acceptable outcome for the client. In the long run, this will drive other MSs out of business and thus reduce redundancy and reliability of the overall MS ecosystem. Since all MSs try to be the first to submit a BP, it is unlikely that a single MS will continuously be the fastest, but slightly slower MSs will still not get any rewards even if they are well behaved and reliable.

If a client wants to waste the resources of MSs, he can first broadcast a BP with a high reward and keep more recent BPs to himself. When a dispute happens, he can wait for the MSs to act before submitting his latest BPs, which prevents the MSs from receiving a reward. Doing this at a large scale is expensive, since the client needs to open and close a payment channel for this at his own cost.

5.6.4 Recourse as a Financial Deterrent

There is no possibility of recourse which lets MSs operate without any incentive of high reliability. A client must expect MSs to ignore their requests and have no means to force a highly reliable monitoring.

5.6.5 Efficiency Requirements

For each channel, only the latest (as indicated by the nonce) BP has to be saved. Unless an extremely high amount of channels is being monitored, this efficiency should not be a concern for the MS. A client can use a single deposit to request an MS to monitor all his payment channels. If this causes the MS to monitor a problematically high amount of channels, he can start to ignore requests made by this client, or even drop old requests. Since there is no punishment for failing to monitor a channel, stopping to monitor is a simple way to reduce resource usage when desired, although it should not be necessary under normal circumstances.

Proposed SC Logic

1. Client (Raiden node) will transfer tokens used as a reward to the User Deposit Contract (UDC)
2. Whoever calls SC's `updateTransfer` method MUST supply payout address as a parameter. This address is stored in the UDC. `updateTransfer` MAY be called multiple times, but it will only accept a balance proof newer than the previous one

3. When calling `claimReward`, the reward tokens will be sent to the payout address

5.7 Appendix A: Interfaces

5.7.1 Broadcast Interface

Client's request to store a 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 Services picks them up.

5.7.2 Web3 Interface

Monitoring Service are required to have 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 the Monitoring Service must call `updateNonClosingBalanceProof` with the respective latest balance proof provided by its client. On `ChannelSettled` event any state data for this channel MAY be deleted from the MS.

5.8 Appendix B: Message Format

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

5.8.1 Monitor Request

Monitor Requests are messages that the Raiden client broadcasts to Monitoring Services in order to get monitoring for a channel.

A Monitor Request consists of a the following fields:

Field Name	Field Type	Description
balance_proof	object	Latest Blinded Balance Proof to be used by the monitor service
non_closing_signature	string	Signature of the Onchain Balance Proof by the client
reward_amount	uint256	Proposed fee in RDN
reward_proof_signature	string	Signature of the reward proof data.

- The balance proof and its signature are described in the Balance Proof specification.
- The creation of the `non_closing_signature` is specified in the Balance Proof Update specification.
- The `reward_proof_signature` is specified below.

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

5.8.2 Example Monitor Request

```
{
  "balance_proof": {
      "token_network_address": "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2",
      "chain_id": 1,
      "channel_identifier": 76,
      "balance_hash":
↪"0x1c3a34a22ab087808ba772f40779b04e719080e86289c7a4ad1bd2098a3c751d",
      "nonce": 5,
      "additional_hash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
      "signature":
↪"0xd38c435654373983d5bdee589980853b5e7da2714d7bdcba5282ccb88ffd29210c3b1d07313aab05f7d2a514561b679
↪"
  },
  "non_closing_signature":
↪"0x77857e08793165163380d50ea780cf3798d2132a61b1d43395fc6e4a766f3c1918f8365d3bef173e0f8bb32c1f373be
↪",
  "reward_amount": 1234,
  "reward_proof_signature":
↪"0x12345e08793165163380d50ea780cf3798d2132a61b1d43395fc6e4a766f3c1918f8365d3bef173e0f8bb32c1f373be
↪"
}
```

5.8.3 Reward Proof

```
ecdsa_recoverable(privkey, sha3_keccak("\x19Ethereum Signed Message:\n148" || channel_
↪identifier || reward_amount || token_network_address || chain_id || nonce ))
```

Fields

Field Name	Field Type	Description
signature_prefix	string	`\x19Ethereum Signed Message:\n`
message_length	string	`148` = length of message = `32 + 32 + 20 + 32 + 32`
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
reward_amount	uint256	Rewards received for updating the channel.
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
nonce	uint256	Strictly monotonic value used to order transfers. The nonce starts at 1
signature	bytes	Elliptic Curve 256k1 signature on the above data from participant paying the reward

CHAPTER 6

Raiden Pathfinding Service

6.1 Overview

A path finding service having a global view on a token network can provide suitable payment paths for Raiden nodes. Raiden nodes can request paths via public endpoints and pay per request. The service will keep its view on the token network updated by listening to blockchain events and a public matrix room where balance proofs and fees are being published. Nodes will publish their current balance proofs and fees in order to advertise their channels to become mediators.

6.2 Implementation Process and Assumptions

- The path finding service will be implemented iteratively adding more complexity on every step.
- There are three steps planned - (1) Pathfinding Minimal Viable Product, (2) Adding service fees, (3) Handling mediation fees.
- It should be able to handle a similar amount of active nodes as currently present in Ethereum (~20,000).
- Nodes are incentivized to publicly report their current balances and fees to "advertise" their channels.
- 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.

6.3 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) * at step (1 and 2) are 1 (no fees being implemented) and * at step (3) are the fees of each channel.

Additionally, we will apply 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. (From step (3) on) The second heuristic is configurable via the optional argument `bias`, which models the trade-off between speed and cost of a 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.

6.4 Public Interfaces

The path finding service needs three public interfaces

- a public endpoint for path requests by Raiden nodes
- an endpoint to get updates from blockchain events
- an endpoint to get updates about current balances and fees

6.4.1 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.

POST api/v1/<token_network_address>/paths

The method will do `max_paths` iterations of Dijkstras algorithm on the last-known state of the Raiden Network (regarded as directed weighted graph) to return `max_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.

Routing 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.
max_paths	int	The maximum number of paths returned.

Payment Arguments

See *Communication between client and PFS*.

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
- The 'token_network_address' is invalid
- 'bias' is invalid
- 'max_paths' is invalid
- 'value' is invalid

Errors

Note: In addition to the error messages, error codes will be added to easily identify the different error cases and handle them automatically.

- Wrong `receiver`
- Outdated payment session. Please choose new `expiration_block`.
- Too low payment `amount`. The last IOU for the current session is included in the `last_iou` field of the returned object.
- Invalid payment signature
- Deposit in UserDeposit contract is too low.

- Bad client. The client behaved badly in the past and the PFS does not want to provide service to it, anymore. One reason for this could be by using a new `expiration_block` for each request, so that it is not profitable for the PFS to claim the service payments.

Example

```
// Request
curl -X POST --data '{
    "from": "0xalice",
    "to": "0xbob",
    "value": 45,
    "max_paths": 10
}'
// Result for success
{
    "result": [
    {
        "path": ["0xalice", "0xcharlie", "0xbob"],
    },
    {
        "path": ["0xalice", "0xeve", "0xdave", "0xbob"]
    },
    ...
    ]
}
// Result for failure
{
    "errors": "No suitable path found."
}
// Result for exceeded rate limit
{
    "errors": "Rate limit exceeded, payment required. Please call 'api/v1/payment/info
↪' to establish a payment channel or wait."
}
```

GET api/v1/<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

A JSON object with the following properties:

Field Name	Field Type	Description
price_per_request	int	The address of payer in the RDN token network.
pfs_address	address	The PFS address 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/v1/0xtoken_network/payment/info
// Result for success
{
    "result":
    {
        "price_per_request": 1000,
        "paths":
        [
            {
                "path": ["0xrdn_alice", "0xrdn_eve", "0xrdn_service"],
            },
            ...
        ]
    }
// Result for failure
{
    "errors": "No suitable path found."
}
```

6.4.2 Network Topology Updates

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.

6.4.3 Balance and Fee Updates (Graph Weights)

Updates for channel balances and fees are published over a public matrix room. Path finding services can pick these balance proofs from there and update the topology represented internally. The Raiden nodes that want to earn fees mediating payments would be incentivized to publish their balance proofs in order to provide a path.

Balance Update

Balance Updates are messages that the Raiden client broadcasts to Pathfinding Services in order to let them know about updated channel balances.

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
chain_id	uint256	Chain identifier as defined in EIP155
reveal_timeout	uint256	Reveal timeout of this channel
signature	bytes	Elliptic Curve 256k1 signature on the above data

Signature

The signature of the message is calculated by:

```
ecdsa_recoverable(privkey, sha3_keccak(nonce || chain_id || token_network_address ||␣
↪channel_identifier || transferred_amount || locked_amount || locksroot || reveal_
↪timeout))
```

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

6.5 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 7

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)

7.1 Overview

Specification document for a Raiden Network Mobile Wallet implementation.

7.2 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.

7.3 Terminology

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

7.4 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

7.5 Depends on

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

7.6 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)

7.7 Platforms & Languages

(TODO: pros & cons for each)

7.7.1 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

7.7.2 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

7.7.3 Other wallets

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

7.8 Components

7.8.1 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.

7.8.2 Raiden Full Node

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

7.8.3 On-Chain Client Library

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

7.8.4 Relay Server

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

7.8.5 Ethereum Node

- provides read & write access to the blockchain

7.9 MobileApp

7.9.1 Visuals

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

7.9.2 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)

7.9.3 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

7.9.4 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

7.10 Protocols

7.10.1 Easy onboarding

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

7.10.2 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

7.10.3 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)

7.10.4 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

7.10.5 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

7.11 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)

7.12 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)

7.13 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 8

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. See *Protocol Values and Settlement Algorithm Analysis* for how this is computed.

chain id Chain identifier as defined in EIP155.

Challenge Period

Settlement Window

Settle Timeout The state of a channel after one channel participant closes the channel. During this period the other participant (or any delegate) is able to provide balance proofs by calling *updateNonClosingBalanceProof()*. This phase is limited for a number of blocks, after which the channel can be *settled*. The length of the challenge period can be configured when each channel is opened.

Challenge Period Update Update of the channel state during the *Challenge period*. The state can be updated either by the non-closing participant, 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.

Locked Transfer

Locked Transfer message An offchain Raiden message that reserves an amount of tokens for a specific *Payment*. See *Locked Transfer* for details.

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

Mediator A node that mediates a *Payment*.

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.

Refund Transfer

Refund Transfer message An offchain Raiden message for a *Transfer* seeking a rerouting. When a receiver of a *Locked Transfer* message gives up reaching the target, they return a Refund Transfer message. The Refund Transfer message locks an amount of tokens in the direction opposite from the previous *Locked Transfer* allowing the previous hop to retry with a different path.

Reveal Secret

Reveal Secret message An offchain Raiden message that contains the secret that can open a *Hash Time Lock*. See *Reveal Secret* for details.

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.

Secret Request An offchain Raiden message from the target that asks for the *secret* of the payment. See *Secret Request* for details.

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.

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.

Unlock

Unlock message An offchain Raiden message that contains a new *balance proof* after a *Hash Time Lock* is unlocked. See *Unlock* for details.

withdraw proof

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

Index

S

T

U

W