Raiden Specification Documentation

Release 0.1

Brainbot

May 16, 2018

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 Balance Proof

Data required by the smart contracts to update the payment channel end of the participant that signed the 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)
```

Fields

Field Name	Field Type	Description
balance_hash	bytes32	Balance data hash
nonce	uint64	Strictly monotonic value used to order transfers. The nonce starts at 1
additional_hash	bytes32	Hash of additional data used on the application layer, e.g.: payment metadata
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
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

1.2.2 Withdraw Proof

Data required by the smart contracts to allow a user to withdraw funds from a channel without closing it. Signature must be valid and is defined as:

```
ecdsa_recoverable(privkey, sha3_keccak(participant_address || total_withdraw ||␣
↪channel_identifier || token_network_address || chain_id)
```

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
participant_address	address	Channel participant, who withdraws the tokens
total_withdraw	uint256	Total amount of tokens that participant_address has withdrawn from the channel
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
signature	bytes	Elliptic Curve 256k1 signature on the above data

1.2.3 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. Signature must be valid and is defined as:

```
ecdsa_recoverable(privkey, sha3_keccak(participant1_address || participant1_balance␣
↪|| participant2_address || participant2_balance || channel_identifier || token_
↪network_address || chain_id)
```

Fields

Field Name	Field Type	Description
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
channel_identifier	uint256	Channel identifier inside the TokenNetwork contract
token_network_address	address	Address of the TokenNetwork contract
chain_id	uint256	Chain identifier as defined in EIP155
signature	bytes	Elliptic Curve 256k1 signature on the above data

1.2.4 Merkle Tree

A binary tree composed of the hash of the locks. The root of the tree is the value used in the *balance proofs*. The tree is changed by the `MediatedTransfer`, `RemoveExpiredLock` and `Unlock` message types.

1.2.5 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	uint64	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 Direct Transfer

A non cancellable, non expirable payment.

Invariants

- Only valid if the *transferred amount* is larger than the previous value and it increased by an amount smaller than the participant's current *capacity*.

Fields

Field Name	Field Type	Description
balance_proof	BalanceProof	Balance proof for this transfer

1.3.2 Mediated 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

- The *balance proof* locksroot must be equal to the previous valid merkle tree with the lock provided in the messaged added into it.
- The transfer is valid only if the lock amount is smaller than the sender's *capacity*.

Fields

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

1.3.3 Secret Request

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

Invariants

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

Fields

Field Name	Field Type	Description
payment_amount	uint256	The amount received by the node once secret is revealed
lock_secrethash	bytes32	Specifies which lock is being unlocked
signature	bytes	Elliptic Curve 256k1 signature

1.3.4 Secret Reveal

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

Fields

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

1.3.5 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. Updated *balance proof*, increases the *transferred amount* and removes the unlocked lock from the merkle tree.

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 SecretReveal message.

Fields

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

1.3.6 RemoveExpiredLock

Removes one lock that has expired. Used to trim the merkle tree and recover the locked capacity. This message is only valid if the corresponding lock expiration is lower than the latest block number for the corresponding blockchain.

Fields

Field Name	Field Type	Description
secrethash	bytes32	The secrethash to remove
balance_proof	BalanceProof	The updated balance proof
signature	bytes	Elliptic Curve 256k1 signature

1.4 Specification

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

1.4.1 Transfers

The protocol supports two types of transfers, direct and mediated. A *Direct transfer* is non cancellable and unexpirable, while a *mediated transfer* may be cancelled and can expire.

A mediated transfer is done in two stages, possibly on a series of channels: - Reserve token *capacity* for a given payment - Use the reserved token amount to complete payments

1.4.2 Message Flow

Nodes may use direct or mediated transfers to send payments.

Direct Transfer

A `DirectTransfer` does not rely on locks to complete. It is automatically completed once the network packet is sent off. Since Raiden runs on top of an asynchronous network that can not guarantee delivery, transfers can not be completed atomically. The main points to consider about direct transfers are the following:

- The messages are not locked, meaning the envelope *transferred amount* is incremented and the message may be used to withdraw the token. This means that a *sender* is unconditionally transferring the token, regardless of getting a service or not. Trust is assumed among the *sender*/*receiver* to complete the goods transaction.
- The sender must assume the transfer is completed once the message is sent to the network, there is no workaround. The acknowledgement in this case is only used as a synchronization primitive, the payer will only know about the transfer once the message is received.

A succesfull direct transfer involves only 2 messages. The direct transfer message and an `ACK`. For an Alice - Bob example:

- Alice wants to transfer `n` tokens to Bob.
- **Alice creates a new transfer with.**
    - transferred_amount = `current_value + n`
    - `locksroot` = `current_locksroot_value`
    - nonce = `current_value + 1`
- Alice signs the transfer and sends it to Bob and at this point should consider the transfer complete.

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 hops in between. The number of hops can also be zero as these transfers can also be sent to a direct partner. Assuming `N` number of hops a mediated transfer will require `6N + 8` messages to complete. These are:

- `N + 1` mediated or refund messages
- `1` secret request
- `N + 1` secret reveal
- `N + 1` secret
- `3N + 4` ACK

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)`
  - 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 `SecretReveal` 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 an `Unlock` 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.

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, communication is via HTTP long polling.

2.3 Use in Raiden

2.3.1 Identity

The identity verification MUST not be tied to Matrix identities. Even though Matrix provides identity server, it is a possible central point of failure. All messages passed between participants MUST be signed using privkey of the ethereum account, using Matrix only as a transport layer. The messages MUST be validated using ecrecover by receiving parties. In order to avoid replay attacks, message format MUST also include identity of the sender.

2.3.2 Discovery

In the above system, clients can search matrix server for any "seen" user whose displayname or userId contains the ethereum address. These are candidates for being the actual user, but it should only be trusted after a challenge is answered with a signed message. 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 then be signed.

2.3.3 Presence

Matrix allows to check for the presence of a user.

2.3.4 Sending transfer messages to other nodes

Direct Message, which is modeled as a room with 2 participants. Channel room may be derived from channel Id / event / hash. Participants may be invited, or voluntarily join upon detecting blockchain events of interest (participant).

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 each token network. Participants can discover peers willing to open more channels.

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.

Blockchain Event Rooms

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

CHAPTER 3

Raiden Network Smart Contracts Specification

3.1 Overview

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

3.2 General Requirements

3.2.1 Secure

- A participant may never receive more tokens than it was paid
- The participants don't need to be anonymous
- The amount transferred don't need to be unknown
- The channel balances don't need to be unknown

3.2.2 Fast

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

3.2.3 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.
- There must be no way for a party to steal funds.
- The proof must be non malleable.
- Losing funds as a penalty is not considered stealing, but must be clearly documented.
- The system must support smart locks.
- Determine if and how different versions of the smart contracts should interoperate.
- Determine if and how channels should be upgraded.

3.4 Project Specification

3.4.1 Expose the network graph

Clients have to collect events in order to derive the network graph.

3.4.2 Functional decomposition

TokenNetworksRegistry Contract

Attributes:

- `address public secret_registry_address`
- `uint256 public chain_id`

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.

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

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 `keccak256(address participant1, address participant2)`, where the two participant addresses are in lexicographic order.

Attributes:

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

Open a channel

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

```
function openChannel(address participant1, address participant2, uint settle_timeout)␣
↪public returns (uint256 channel_identifier)
```

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

- `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`.
- `channel_identifier`: *Channel identifier* assigned by the current contract.

Fund a channel

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

```
function setDeposit(
    address participant,
    uint256 total_deposit,
    address partner
)
    public
```

```
event ChannelNewDeposit(uint channel_identifier, address participant, uint deposit);
```

- `participant`: Ethereum address of a channel participant whose deposit will be increased.
- `total_deposit`: Total amount of tokens that the `participant` will have as `deposit` in the channel.
- `partner`: Ethereum address of the other channel participant, used for computing `channel_identifier`.
- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `deposit`: The total amount of tokens deposited in a channel by a participant.

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

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

Withdraw tokens from a channel

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 message.

```
function withdraw(
    address participant,
    uint256 total_withdraw,
    address partner,
    bytes participant_signature,
    bytes partner_signature
)
    external
```

```
event ChannelWithdraw(bytes32 channel_identifier, address participant, uint256␣
↪withdrawn_amount);
```

- `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.
- `partner`: Channel partner address.
- `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.

Close a channel

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

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

```
event ChannelClosed(uint channel_identifier, address closing_participant);
```

- `partner`: Channel partner of the participant who calls the function.
- `balance_hash`: Hash of the balance data `keccak256(transferred_amount, locked_amount, locksroot)`
- `nonce`: Strictly monotonic value used to order transfers.
- `additional_hash`: Computed from the message. Used for message authentication.
- `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.
- `signature`: Elliptic Curve 256k1 signature of the channel partner on the *balance proof* data.

- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `closing_participant`: Ethereum address of the channel participant who calls this contract function.

Note: Only a participant may close the channel.

Only a valid signed *balance proof* from the channel partner (the other channel participant) must be accepted. This *balance proof* sets the amount of tokens owed to the participant by the channel partner.

Update non-closing participant balance proof

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

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

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

- `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 message. Used for message authentication.
- `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.
- `channel_identifier`: Channel identifier assigned by the current contract.
- `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.

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.

```
function settleChannel(
    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 channel_identifier, uint256 participant1_amount, uint256␣
↪participant2_amount);
```

- `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`).
- `channel_identifier`: *Channel identifier* assigned by the current contract.

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

Cooperatively close and settle a channel

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

- `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.

Can be called by a third party as long as both participants provide their signatures.

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(
    address participant,
    address partner,
    bytes merkle_tree_leaves
)
    public
```

```
event ChannelUnlocked(uint256 channel_identifier, address participant, uint256␣
↪unlocked_amount, uint256 returned_tokens);
```

- `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)`.
- `channel_identifier`: *Channel identifier* assigned by the current contract.
- `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.

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)
```

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

Getters

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

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

3.4.3 Data types definition

A detailed description of the *balance proof* can be found in the *message definition*. A detailed description of the *withdraw proof* can be found in the *message definition*. A detailed description of the *cooperative settle proof* can be found in the *message definition*.

3.5 Decisions

- **Batch operations should not be supported in Raiden Network smart contracts. They can be done in a smart contract wrap**
    - Provide smart contract to batch operations with the same function names but vectorized types. Example: opening multiple channels in the same transaction.
    - To save on the number of transactions, add optimization functions that do multiple smart contract function calls

CHAPTER 4

Raiden Network Monitoring Service

4.1 Basic requirements for the MS

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

4.2 Usual scenario

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

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

4.3 Economic incentives

4.3.1 Raiden node

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

4.3.2 Monitoring service

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

4.3.3 General requirements

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

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

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

Monitoring services MUST listen in the provided global chat room

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

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

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

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

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

4.3.4 Fees/Rewards structure

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

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

Proposed SC logic

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

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

4.3.5 Notes/observations

The NettingChannelContract/Library as it is now doesn't allow more than one updated BP to be submitted. The contract also doesn't check if the updated BP is newer than the already provided one How will raiden nodes specify/deposit the monitoring fee? How will it be collected?

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

4.4 Appendix A: Interfaces

4.4.1 Broadcast interface

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

4.4.2 Web3 Interface

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

Event filtering

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

4.4.3 REST interface

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

Endpoints

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

4.5 Appendix B: Message format

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

4.5.1 Balance proof

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

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

4.5.2 Example data: Balance proof

‘‘ {

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

CHAPTER 5

Raiden Pathfinding Service Specification

5.1 Overview

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

5.2 Assumptions

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

5.3 High-Level-Description

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

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

5.4 Public Interface

5.4.1 Definitions

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

Channel_Id

- uint: channel_identifier

Balance_Proof

See *balance proof definition*.

Lock

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

5.4.2 Public Endpoints

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

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

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

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

Arguments

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

Returns

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

- Invalid balance proof
- Invalid channel id

Example

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

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

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

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

Arguments

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

Returns

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

- Invalid channel id
- Invalid signature

Example

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

`api/1/<token_network_address>/paths`

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

- Checks if an edge (i.e. a channel) has `capacity > value`, else ignores it.

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

Arguments

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

Returns

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

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

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

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

Example

```
// Request
curl -X GET --data '{
    "from": "0xalice",
    "to": "0xbob",
    "value": 45,
    "num_paths": 10
}'  /api/1/paths
// Request with specific preference
curl -X PUT --data '{
    "from": "0xalice",
    "to": "0xbob",
    "value": 45,
    "num_paths": 10,
    "extra_data": "min-hops"
}'  /api/1/0xtoken_network/paths
// Result for success
{
    "result": [
```

(continues on next page)

(continued from previous page)

```
    {
        "path": ["0xalice", "0xcharlie", "0xbob"],
        "estimated_fees": 3
    },
    {
        "path": ["0xalice", "0xeve", "0xdave", "0xbob"]
        "estimated_fees": 5
    },
    ...
    ]
}
// Result for failure
{
    "error": "No suitable path found."
}
// Result for exceeded rate limit
{
    "error": "Rate limit exceeded, payment required. Please call 'api/1/payment/info'␣
↪to establish a payment channel or wait."
}
```

`api/1/<token_network_address>/payment/info`

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

Arguments

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

Returns

An object consisting of two properties:

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

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

- No suitable path found

Example

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

5.5 Implementation notes

5.5.1 Network topology updates

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

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

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

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

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

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

5.6 Future Work

The methods will be rate-limited in a configurable way. If the rate limit is exceeded, clients can be required to pay the path-finding service with RDN tokens via the Raiden Network. The required path for this payment will be provided by the service for free. This enables a simple user experience for light users without the need for additional on-chain transactions for channel creations or payments, while at the same time monetizing extensive use of the API.

CHAPTER 6

Raiden Mobile Wallet Specification

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

6.1 Overview

Specification document for a Raiden Network Mobile Wallet implementation.

6.2 Goal

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

6.3 Terminology

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

6.4 Requirements

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

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

6.5 Depends on

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

6.6 High Level Features

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

6.7 Platforms & Languages

(TODO: pros & cons for each)

6.7.1 Native vs. Progressive Apps

Progressive Apps

- service workers handle push notifications easier

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

Native

- more efficient, can be compiled to wasm

6.7.2 Languages

RLC

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

6.7.3 Other wallets

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

6.8 Components

6.8.1 Raiden Light Client Library

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

6.8.2 Raiden Full Node

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

6.8.3 On-Chain Client Library

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

6.8.4 Relay Server

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

6.8.5 Ethereum Node

- provides read & write access to the blockchain

6.9 MobileApp

6.9.1 Visuals

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

6.9.2 User Onboarding Flow Example

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

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

6.9.3 Transaction Flow Example

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

6.9.4 UI Features Example

About

- version
- Terms of Use
- License

Settings

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

Account

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

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

Setup

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

Channel info

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

On-chain transaction UI

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

Off-chain transaction UI

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

Hub reputation system (tbd)

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

6.10 Protocols

6.10.1 Easy onboarding

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

6.10.2 Payment Requests

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

6.10.3 Push Notifications

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

6.10.4 Other

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

6.10.5 Existing Tools/Services

Wallet

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

Wallet SC

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

Account identity

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

Event Watching

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

6.11 Roadmap

(purely estimative)

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

6.12 Issues to clarify on

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

6.13 Other wallets:

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

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

CHAPTER 7

Raiden Terminology

Additional Hash

additional_hash Hash of additional data used on the application layer. This can for example be some form of payment metadata. The Raiden Network protocol does not use or enforce the type or content of this additional data.

balance proof

Participant Balance Proof

BP Signed data required by the *Payment Channel* to prove the balance of one of the parties. See the *message 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.

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

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

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

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

Deposit Amount of token locked in the contract.

Direct Transfer A non-refundable non-cancellable off-chain Payment done with a single Transfer.

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 *mediated 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.

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

Mediator A node that mediates a transfer.

merkletree root

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

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

Monitoring Service

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

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

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 *mediated transfer* sent by a node. The node may be a *Mediator* in the path or the *Initiator*.

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

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

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

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

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

Payment Receipt TBD

Raiden Channel The Payment Channel implementation used in Raiden.

Raiden Light Client A client that does not mediate payments.

Raiden Network A collection of Token networks.

Receiver The node that is receiving a Message.

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

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

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

Sender The node that is sending a Message.

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

Settlement Window

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

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

Target The node that receives a *Payment*.

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

Token Swaps Exchange of one token for another.

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

Transferred amount Monotonically increasing amount of token transferred from one node to another.

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

withdraw proof

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

Index