Transactions and Verification Protocol Design

Optimising Verification and Transactions on the OMNIA SCM through Plasma and Off-Chain Interactions

Josh Fourie
josh@omniaprotocol.com

Draft Copy 0.01 *

August 2018

INTENDED FOR INTERNAL CIRCULATION

Abstract We require a consensus algorithm for the decentralised verification of zero-knowledge proofs to achieve the *trustless*, *verifiable* and *resistant* characteristics of the OMNIA SCM. This document provides an explanation of a verification mechanism implemented on the Plasma model introduced by Poon and Buterin in 2017 and interacting with the Ethereum blockchain for the OMNIA network. The model we provide levers Plasma as an off-chain decentralised marketplace that minimises on-chain processes as a throughput and cost optimisation.

*Note that details in this document are subject to change and there may be errors in this version

Contents

I Transactions on the Plasma Model **3**

1 Introduction **3**

2 Plasma **3**

2.1 Plasma-Chain 4

2.2 Enforcible Transactions on the Root-Chain 5

2.3 Scalability: MapReduce and Child-Chains 5

2.4 Proof of Stake Consensus Controller 7

3 Raiden Implementation **7**

3.1 Netting Channel 7

3.2 Secret Keys and Hash-Locked Deposits 8

3.3 Role of Third Parties 9

3.4 Performance Estimates 9

4 Summarising Diagrams **9**

II Implementing Plasma for OMNIA **12**

5 Basic Functionality for Requirement I **13**

5.1 Stripped Model 13

5.2 Hopping Channels 14

5.3 Incorporating the Proof Function 14

6 Basic Functionality for Requirement II **15**

6.1 Appending Token Generations and Exchanges to the Root-Chain 15

7 Slasher Protocol **16**

7.1 Generalisable Mechanics 16

7.2 Rejected Proofs 16

7.3 Enforcing Valueless Tokens 18

III Related Work **20**

8 Nakamoto Consensus and Proof-of-Work **20**

Part I

Transactions on the Plasma Model

We require that the consensus protocol enables a party (D-O) to prove to the network that they have correctly generated a request for payment through the verification of non-interactive zero-knowledge proofs attached to output of the locally executed computation (ORB). A solution requires both a high-level of security and transactions per second (TPS). In this paper we provide an explanation of our implementation of Plasma and Casper and justify those design decisions. Note that we assume familiarity with the OMNIA protocol and explain Plasma through the Raiden model. [1]

1 Introduction

Plasma operates as a framework of smart contracts layered over and enforced by a root-chain to provide an optimised off-chain platform for exchanging transactions without incurring the monetary or time costs of compensating root-chain nodes to incorporate and verify those same transactions. A working draft of the Plasma framework was proposed by Buterin and Poon in 2017 as a solution to blockchain scalability. A Plasma-based solution achieves a high throughput but requires the submission of fraud proofs wherever a party believes a transaction they are engaged in is invalid. It is therefore useful for a network such as OMNIA where there are transactions of value but there is a lesser risk of double spending or dishonouring of transactions because of the nature of the transacted tokens or services. [2]

2 Plasma

Plasma is frame-worked on the following essential components: [3]

1. A 'Plasma-chain' for transacting.
2. A hierarchical chain structure to enforce transactions and achieve finality.
3. A scalability solution through MapReduce and child-chains.
4. A proof-of-stake consensus algorithm with a novel incentive for block propagation.

We are particularly interested in the transaction functionalities.

2.1 Plasma-Chain

The Plasma-chain is a framework of smart-contracts responsible for managing off-chain interactions between transacting parties. These smart-contracts enables payers to open channels or connect to existing networks, commit tokens to a hash-locked deposit, perform mediated transfers and settle channels. [4]

Registering and Depositing Tokens A payer must commit an amount of registered tokens to a deposit before they are able to message a recipient the key to withdraw funds. The process of registration ensures that the deposited tokens have been validly generated and are backed by a supply of legitimate funds determined through interaction with the root-chain or some other novel mechanism. Once a token is registered, the payer may deposit those tokens in a hash-locked deposit through a commitment of n tokens to the smart-contract controlling the Plasma-chain. [5]

Opening/Connecting Channels A payer is required to open a channel with either a single party or a network of transacting parties before they are able to transfer funds on a Plasma-chain. Once established, a payment channel enables parties to message signed statements to others on the channel either requesting or transferring funds. [6]

Off-Chain Transfers A payer initiates a transfer on the Plasma-chain by sending the recipient a digitally signed message permitting them to use the secret key to withdraw n tokens from the deposit. That signed message is irrefutable and constitutes evidence the recipient can present to a network to prove that they legitimately own those tokens. The transaction concludes when the recipient messages the payer to inform them they have received both the permission message and the secret key for the deposit. [7]

Settling a Channel A payer or recipient can withdraw their tokens from the deposit by messaging the smart-contract controlling the Plasma-chain to settle the channel. The smart-contract allocates the tokens accordingly and records the settlement on the root-chain or a higher-level Plasma-chain. [8]

2.2 Enforcible Transactions on the Root-Chain

The Plasma-chain interacts with the root-chain in a court-like hierarchy where parties submit fraud proofs impugning transactions that are provably false to the root-chain for a smart-contract to adjudicate upon. Any Plasma-based model relies on that referral system for the enforcement of Plasma-chain relationships and therefore the legitimacy of the network is dependent on the effectiveness of the 'appeal' functionality. [9]

Enforcing Correctness A malicious party may attempt to incorrectly withdraw tokens or deny a withdrawal through an unsynchronised update, forgery or obliteration of their locally accessible copy of the transaction record. The harmed party simply refers the dispute upwards for the smart-contract controlling the Plasma-chain to determine the correct state of the hash-locked deposit and the correct allocation of tokens. Recall that the parties exchanged both a secret key and a set of unforgeable, digitally signed messages expressing permission to withdraw tokens. The smart-contract controlling the Plasma-chain also controls the hash-locked deposit and is capable of determining the correct outcome based on the submissions of the harmed party. Any transacting parties may initiate the dispute process though this incurs costs associated with engaging the root-chain. Note that transactions are not finalised until they are subjected to the dispute mechanism or otherwise recorded on the root-chain. [10]

2.3 Scalability: MapReduce and Child-Chains

MapReduce is a scalability solution where large data sets are parsed by parallel clusters of nodes responsible for executing either a *mapping* or *reducing* function. The effect of a MapReduce program is to break complex data analysis into smaller computations that are allocated to different trusted nodes (*mapping*) who return a simplified output which is coalesced into an output by a master node (*reducing*). [11]

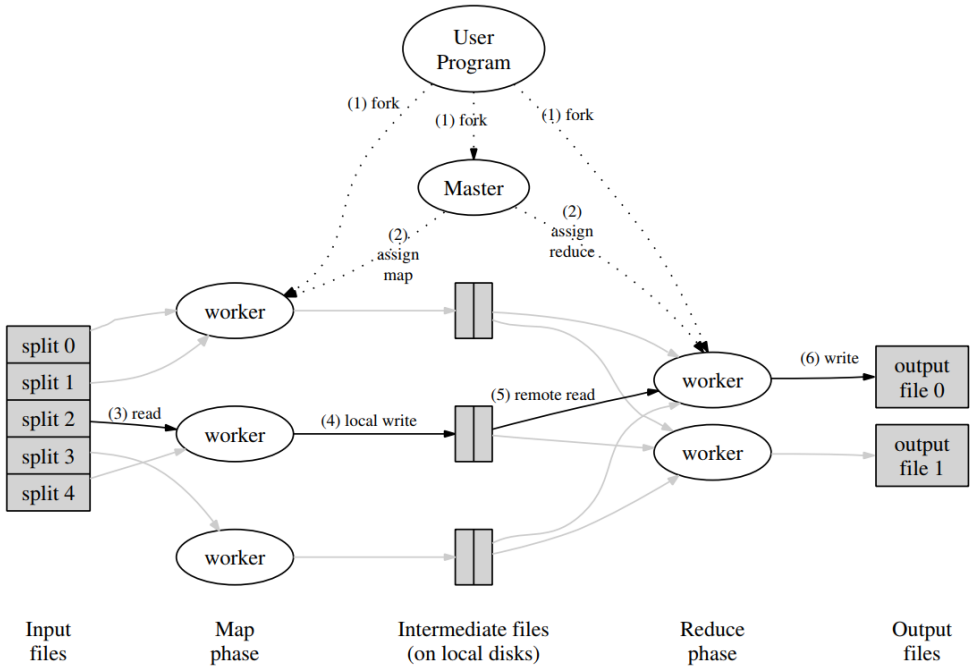

Figure 1: MapReduce execution diagram from Dean and Ghemawat (2004).

Plasma proposes MapReduce as a scalability solution that levers child-chains operating as parallel nodes to minimise the costs of representing transactions on the root chain. A Plasma-chain records a collection of transactions within the Plasma-network and either periodically or when prompted records the current state as a Map-Reduce output on the root-chain. Plasma requires that the finalised MapReduce format holds sufficient information that any node can ensure the legitimacy of the particular token they are transacting. The mapping functionality is responsible for identifying signed messages between parties on a channel then reduced to a single transaction between those parties. [12]

The Plasma model requires the individual parties transacting on child-chains to behave as worker nodes responsible for the MapReduce process. Those workers complete the mapping and reducing task by constructing a private side-chain of transactions recorded on their private ledgers and released to the root-chain in a reduced format upon either the settlement or dispute of the channel. Plasma requires that transacting parties build and monitor child-chains they are interested rather than the entire blockchain to achieve parallelisation and optimise network efficiency rather than verifying every transaction across the blockchain. This incurs a risk of false transactions wherever transacting parties are not monitoring the Plasma-chains relevant to their transaction but those false events have no impact on the broader network and are presumed immaterial. [13]

2.4 Proof of Stake Consensus Controller

The Plasma-chain is layered over a framework of smart contracts burdened with managing the chain and executed by either a single party or a decentralised network of nodes. Note that the security of the Plasma-chain is contingent on the security of the party or parties controlling the consensus mechanism and a proof-of-stake implementation is therefore preferable. Please refer to the *Raiden case study* or *implementation* part of this document for further information. [14]

3 Raiden Implementation

This section will attempt to explain the key elements of the Plasma model through the Raiden implementation. We can begin with a simplified outline of a transaction on the Plasma blockchain where Alice is sending Bob n tokens through a mediated transfer: [15]

1. Alice connects to an existing token network and registers n tokens with the root-chain.
2. Alice enters Bob's address to open a channel.
3. Alice commits n tokens to a hash-locked deposit for that channel and signs the transfer.
4. Alice messages Bob the signed hash-locked deposit.
5. Bob requests that Alice message the secret key enabling agreed withdrawals from the deposit.
6. Alice messages the key to Bob who then confirms they have received that key.
7. Alice and Bob 'synchronise' their records of the transaction.

3.1 Netting Channel

The netting channel smart contract is responsible for providing the programmatic rules for operating the bi-directional off-chain payment channels and managing interactions with the root-chain. There are four life-cycle stages for a channel on the Raiden network: [16]

1. Deployment
2. Funding / Usage
3. Close
4. Settle

Parties may transact on the Raiden network after deploying a channel through either connecting to an existing network or another party's public address and begin the funding and usage stage where signed messages are exchanged as representations of transactions. A channel must be closed either upon settlement or withdrawal to call the parties to message the balance proof to the network. Note that the transactions on those channels are locked to predetermined tokens whose exchange is only finalised when the root-chain has incorporated the transaction record onto the ledger. [17]

The balance proof acts as both a new transfer and an enforcible record of signed messages which the network uses to determine the correct allocation of tokens between transacting parties upon settlement and is composed of a nonce, the transferred amount, the locksroot and a signature for those elements. Transacting parties derive the locksroot for the balance proof by taking the root of a merkletree constructed by summing the values of the hashes of any pending transfers on the unsettled channel. The merkletree has a constant storage requirement and functions as a proof of containment which is recomputed whenever a transaction completes to incorporate the hashed message as a merkle leaf and update the root with the hash of the most recent signed message. Raiden proposes an optimisation to reduce computational complexity where parties order the merkle leaves chronologically rather than lexicographically to restrict re-computation to the rightmost side of the tree. [18]

3.2 Secret Keys and Hash-Locked Deposits

Steps $4 \rightarrow 6$ of the Raiden transaction process require the transacting parties to exchange a secret key composed of 32 bytes of randomised cryptographically secure data to unlock the 'hash-locked' deposit. The deposit lock is constructed by hashing the randomly generated secret with a keccak function and unlocked whenever a party reveals the pre-image of the hashed secret key to the smart contract controlling the transaction. Payers are further required to specify a lock expiration time which enables refunds after the time prescribed by the payer has overturned. Note that as an alternative implementation, Raiden documentation considers the possibility of using a hashed contract in place of the randomised secret key or a dual-lock system for safe refunds. [19]

Recall that the Raiden network achieves transaction safety by requiring that parties accompany a withdrawal call with the relevant balance proof to ensure the correct allocation of tokens whenever the channel is settled. [20]

3.3 Role of Third Parties

Raiden requires a network of nodes managed by a consensus algorithm to handle off-chain transaction tasks whenever the transacting nodes are offline and a separate network for recording the MapReduced output onto the root-chain ledger. The network levers the Ethereum blockchain as a root-chain for the Raiden platform and a separate cluster of Raiden nodes for the management of netting channels. [21]

3.4 Performance Estimates

A document available on the Raiden GitHub estimates that the platform could process approximately 360.00 transactions per second in the instance where there are three specialised validator nodes operating the network and a vertically long child-chain structure. [22]

4 Summarising Diagrams

We have provided a diagram describing transactions on an open channel in Fig. 2 and another for the settlement process in Fig. 3. [23]

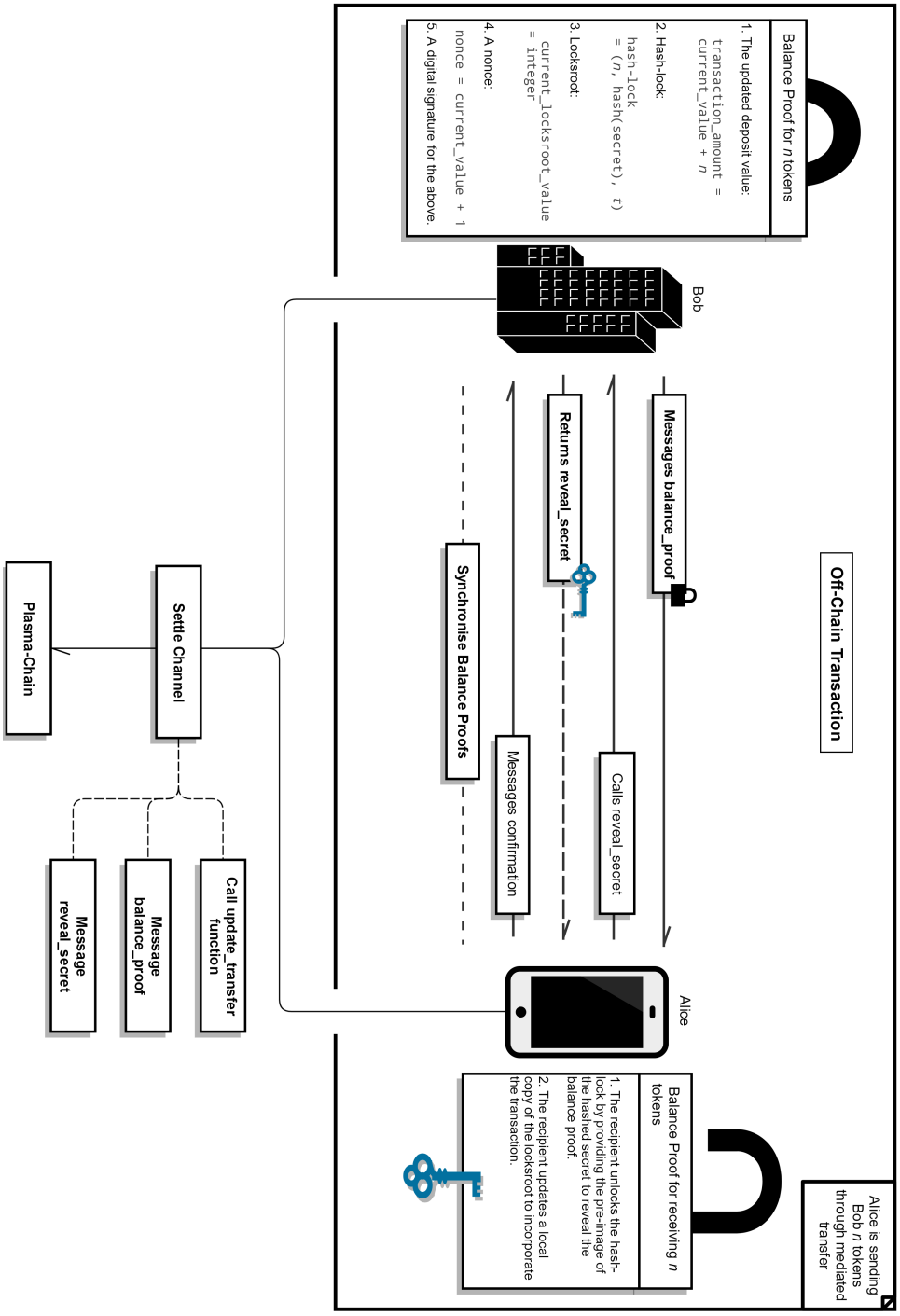

Figure 2: An off-chain transaction where Bob is sending Alice n tokens.

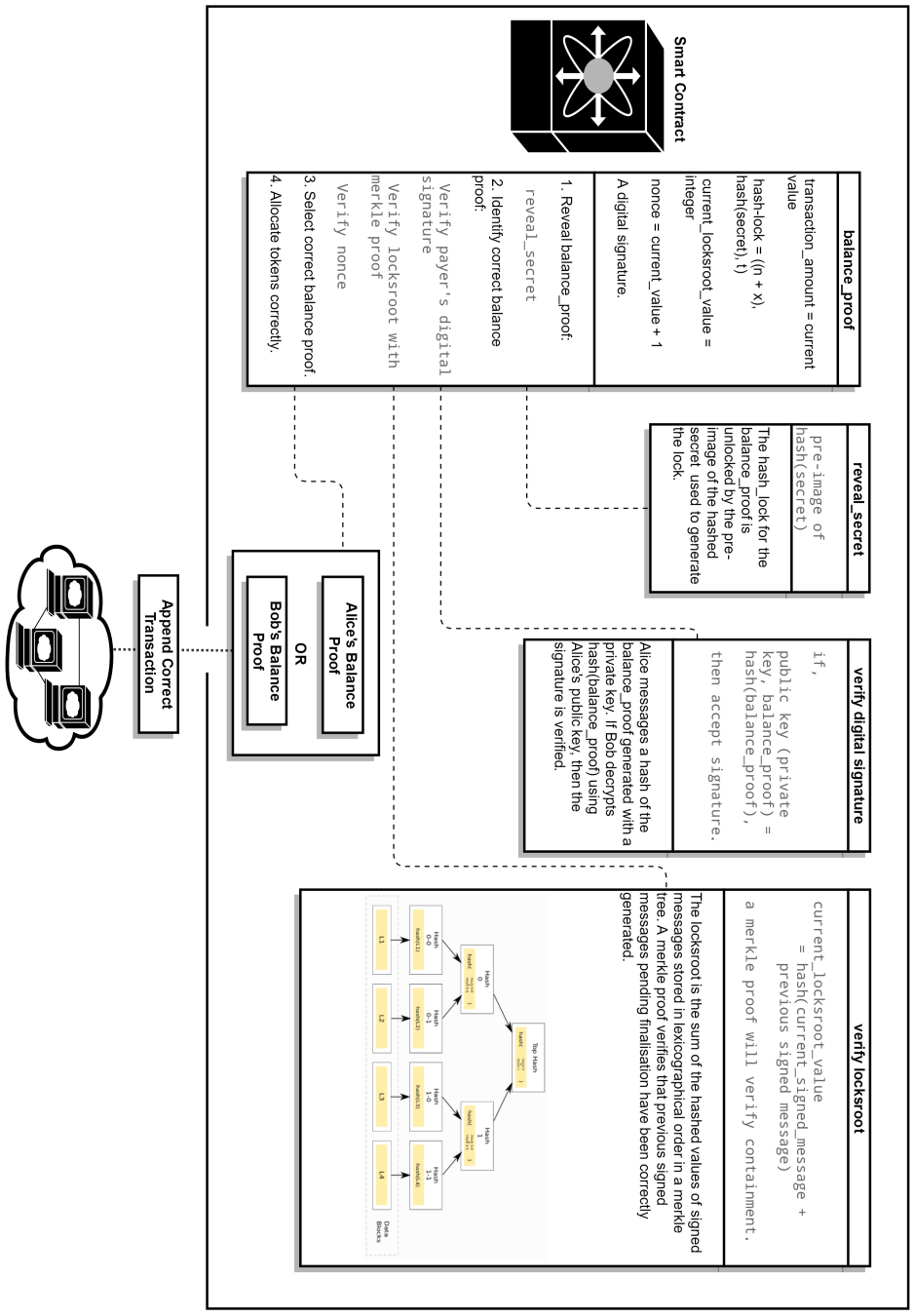

Figure 3: An on-chain transaction where the smart contract settles the open channel between Alice and Bob.

Part II

Implementing Plasma for OMNIA

We intend to leverage the Plasma model for the underlying OMNIA SCM protocol responsible for the verification and compensation functionalities explained in this part. We assume familiarity with zk-SNARKs, ORBS and Plasma and refer the reader to supporting documentation for the explanation of the OMNIA SCM. [24]

Required Functionalities We require the OMNIA plasma protocol to enable two broad functionalities: [25]

I. Compensating D-O with specialised tokens for providing verified ORB COMPUTATIONS.

II. Exchanging specialised tokens for other valuable rewards.

It is imperative that those functions hold the properties required for the OMNIA SCM:

- Requirement I:
  - D-O is not required to trust I-P and can enforce correct behaviour.
  - There is only a negligible chance that D-O will successfully cheat I-P.
- Requirement II:
  - I-P can verify that D-O is exchanging legitimate tokens.
  - D-O can enforce the value of specialised tokens.

This part is structured to explain the motivation of each component and will propose a final model in the last section.

5 Basic Functionality for Requirement I

5.1 Stripped Model

We can trivially achieve the transactional functionality requirement with the Plasma model discussed in Part I and applied in Fig. 4. [26]

Bob initiates the transaction by generating a signed balance proof for n tokens containing the transaction amount, the hash-lock, the locksroot and a nonce: [27]

```
transaction_amount = current_value + n
hash_lock = (n, hash(secret), lock_expiration)
locks_root = recompute_merkle
balance_proof_nonce = current_value + 1
digital_signature = private_key(balance_proof)
```

Alice receives the signed balance proof and returns a request for the secret. Bob may then reveal the secret so that Alice can unlock the balance proof and recompute the locksroot to update their ledger as an enforcible record of the transaction. [28]

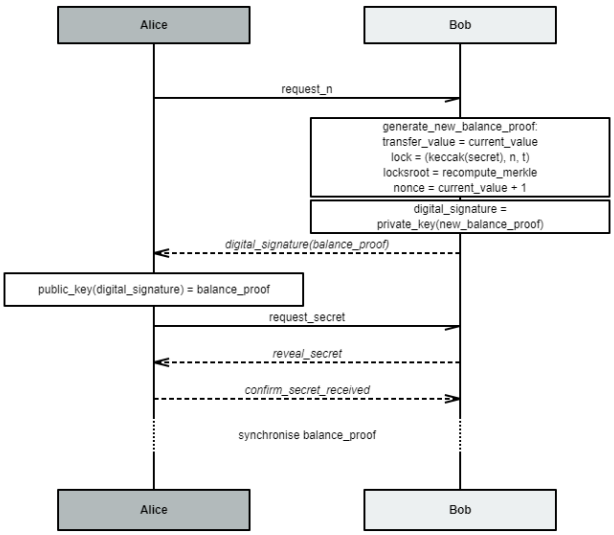

Figure 4: A simple off-chain Plasma transaction between Alice and Bob for n tokens.

5.2 Hopping Channels

Verifiers occupy a central role as distributors and purchasers of the specialised tokens and as a scalability solution we implement the novel hopping mechanism introduced by the Raiden network to construct a network rather than a centralised exchange to reduce the computational load on the verifier. [29]

5.3 Incorporating the Proof Function

Recall that we also require that Bob is able to verify that Alice has correctly performed the ORB COMPUTATION before releasing n tokens which we can insert between `request_secret` and `reveal_secret` described in Fig. 5. [30]

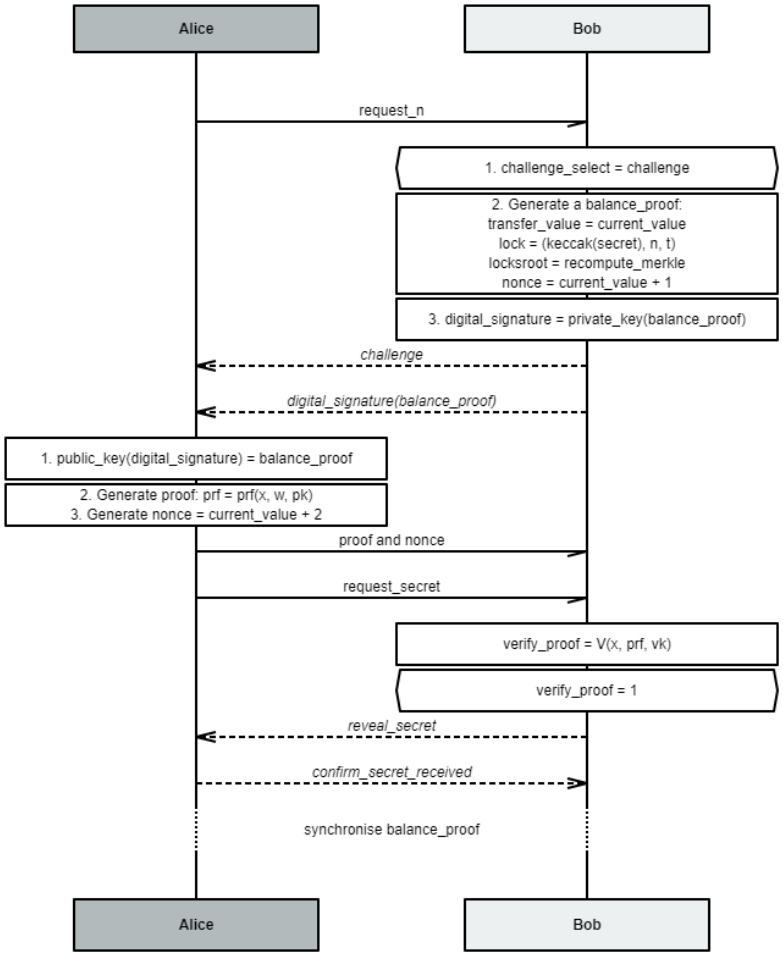

Figure 5: A Plasma transaction where Alice is requesting n tokens from Bob.

Alice is only required to return a `zk_proof` for the ORB COMPUTATION if Bob has chosen to issue a `challenge`. We provide security against a malicious prover who submits an incorrect proof by only requiring that Bob reveal the secret for the hash-lock after the proof has been verified and enforce the generation of another nonce to prevent a malicious prover from recycling correct proofs for outdated ORB COMPUTATIONS to fool the verifier and dishonestly obtain the pending n tokens. [31]

```
proof = prf(orb computation, private data inputs, prover key)
reveal_secret_nonce = current_value + 2
digital_signature - private_key(generate_proof, generate_nonce)
verify_proof = v(orb computation, verifier key, proof)
```

6 Basic Functionality for Requirement II

We can trivially achieve a transaction by extending the model applied in Fig. 4 to enable the prover to transact n tokens on the Plasma model but require additional functionalities explained in this section.

6.1 Appending Token Generations and Exchanges to the Root-Chain

An important utility of the OMNIA platform is the ability for any party to verify that circulated tokens are linked to some valued event and are not generated arbitrarily by unauthorised parties. [32]

We therefore provide the option of an on-chain generation event to introduce a specialised token to the marketplace and broadcast the identities of legitimate tokens available to both the prover and the verifier through the `generate_tokens` protocol. Whenever either a prover or verifier opens a new channel they are required to call the on-chain `deposit` function to lock-up a quantity of tokens on the channel which are later released on-chain when the channel is settled. Calling the root-chain to execute the `deposit` function ensures that tokens exchanged on the channel are backed by a legitimate supply of funds resistant to double-spending or forgery.

Consequentially, verifiers are able to attach significant value to specialised tokens because they are tracked from the generation event to the most recent finalised block on he root-chain.

7 Slasher Protocol

We provide a general mechanism to enforce correct behaviour amongst provers and verifiers to achieve the trustlessness property of the OMNIA SCM.

7.1 Generalisable Mechanics

A prover calls the `slasher` protocol whenever a verifier behaves incorrectly to refer the malicious party to the smart contract for punishment. The appealing prover must provide the smart contract with the `balance_proof`, the `slasher_proof` and a `slasher_nonce` coalesced in the `slasher_appeal` function to initiate the slashing mechanism. We burden the prover with posting a redeemable `slasher_bond` to compensate the consensus network operating the smart contract and deter recurrent false claims intended to drain a verifiers deposit. The smart contract will slash either the verifiers on-chain deposit or the `slasher_bond` posted by the prover according to whether the `slasher_proof` demonstrates that the verifier has behaved incorrectly. A portion of the slashed funds are redistributed through the `redistribute_funds` function to compensate the root-chain nodes and award the harmed prover a bounty in excess of the posted `slasher_bond`. [33]

```
slasher_nonce = current_value + 3
slasher_bond = integer
```

Note that the `slasher_proof` requirements differ depending on the impugned misbehaviour and the `slasher_bond` is set independently of the `slasher_appeal`.

7.2 Rejected Proofs

There is an incentive for a malicious verifier to reject a correctly generated proof to deny an honest prover the promised n tokens and we require the slashing mechanism to identify when a proof has been incorrectly rejected. The prover must package the digitally signed `balance_proof` and the verifier's `reject_proof` message in the `slasher_appeal` object submitted to the smart contract before the network can determine whether the verifier behaved dishonestly. An honest verifier can trivially refute the appeal by providing a valid `balance_proof` containing a record of correct behaviour. [34]

```
reject_proof == V(verifier key, public inputs, Prf) = 0
slasher_proof = v_digital_signature(balance proof, reject_proof)
```

The smart contract executes a containment proof against the `slasher_proof` which has stored the signed `balance_proof` and `reject_proof` respectively in a merkle tree to confirm that there is an incorrect `reject_proof` message from the verifier. The funds deposited by a verifier will only be slashed where the smart contract determines that the `slasher_proof` validly holds both a signed `reject_proof` message and a correctly generated `zk_proof`. [35]

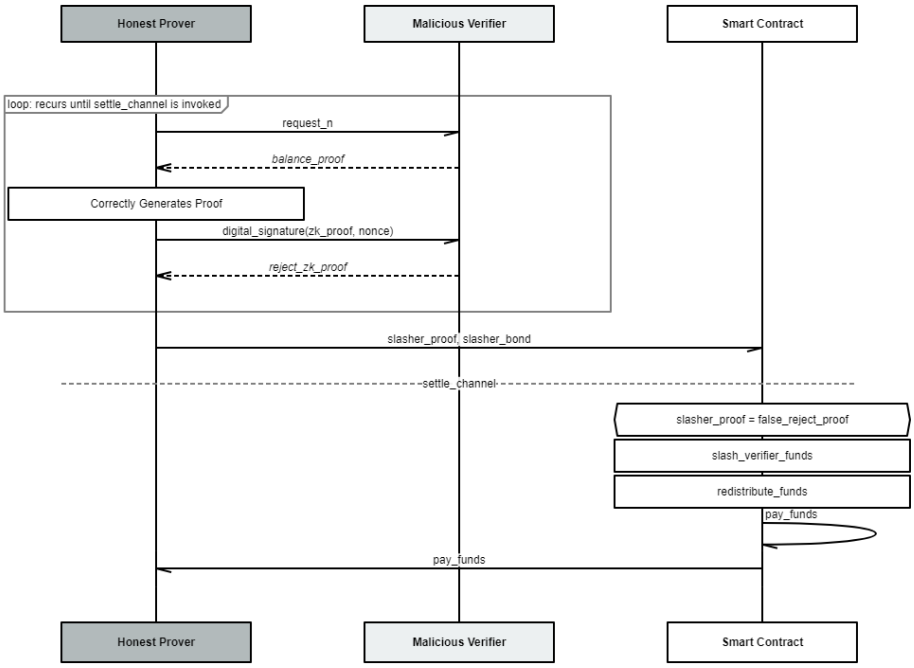

Figure 6: A diagram of the root-chain smart contract slashing a malicious verifier and compensating an honest prover.

7.3 Enforcing Valueless Tokens

The OMNIA SCM permits the the exchange of specialised tokens which hold no inherent value outside of the prover-verifier relationship. A malicious prover might therefore dishonour the token at the point of exchange which we resolve by implementing an optional slasher mechanism as both a deterrent for incorrect behaviour and as minimum compensation for the computational capacity expended by the prover. [36]

Recall that there is a public record of token transactions stored on the root-chain in a MapReduced format tracking exchanges extending from the token generation event to the most recent finalised block explained at 6.1. We can enforce correctness by requiring a harmed party to call the `slasher` function and provide the `balance_proof` and `reject_tokens` contained in the `slasher_proof` whose merkle tree the smart contract runs an audit proof for the signed rejection message against in addition to finalising the `balance_proof`. Note that a verifier cannot be punished in the instance that the prover does not provide the `reject_proof` object as we cannot guarantee message delivery. [37]

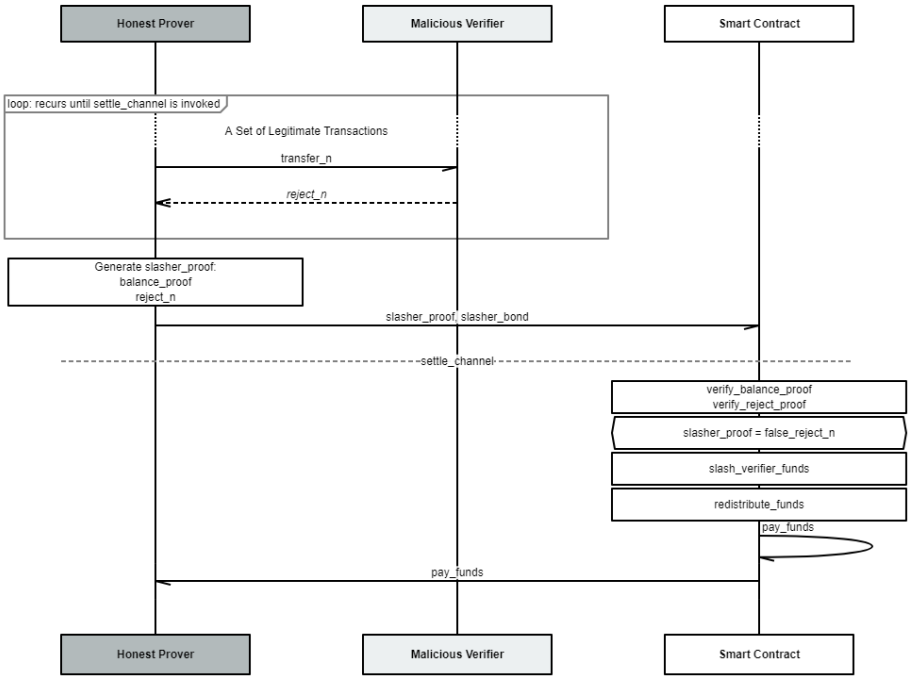

Figure 7: A diagram outlining the smart contract slashing a malicious verifier's funds for incorrectly rejecting n tokens.

An honest prover whose tokens are dishonoured will receive a portion of the slashed funds to recoup the `slasher_bond` and a small bounty set at the discretion of the verifier. We preference a system where verifiers publicly declare the value of funds the smart contract will slash for that bounty if specialised tokens are dishonoured rather than imposing an indiscriminate redistribution function to avoid over-regulating the exchange. This incentive-compatible model encourages honest verifiers to post large bounties for incorrect token rejection events to foster trust with provers but does not punish verifiers who opt against this mechanism. The smart contract derives the correct value to slash from the most recent `slasher_bounty` setting the verifier has appended to the root-chain. [38]

```
slasher_proof = (balance_proof, reject_tokens)
slasher_bounty = (slasher_bond + bounty)
```

Note that an honest prover will retain their ownership of n tokens recorded in the validated `balance_proof`.

Part III

Related Work

In this part we consider alternative solutions that we have rejected for the OMNIA implementation.

8 Nakamoto Consensus and Proof-of-Work

A primary contribution of Bitcoin was the proof-of-work solution (PoW) to the Byzantine Generals problem which enabled a collection of decentralised nodes to reach consensus on a proposed query. The Nakamoto Consensus uses proof-of-work to probabilistically elect a leader node who proposes a block containing transaction records for the remaining nodes to then either accept or reject as the next canonical block in the chain based on the length of the relevant proof-of-work (assuming the block transactions are correct). [39]

A key drawback of that approach is the reliance on computational power as deterministic in the probabilistic leader elections which has both hindered network efficiency from wasted computational capacity invested in the production of stale blocks and enabled nodes with more computational capabilities to improve their chances of proposing a block as leader. The latter has further motivated the development of techniques that increase computational capacity such as Application Specific Integrated Circuits.

We reject the Nakamoto Consensus for the OMNIA SCM as we intend to leverage smart-phones for decentralised verification which are computationally inferior to the computational capacity available to malicious parties. An implementation of proof-of-work within that model would expose the network to '51% attacks and other undue problems.