Kairos: Zero-knowledge Casper Transaction scaling

Marijan Petricevic, Nick Van den Broeck

Tuesday October 31, 2023

Contents

1. Introduction

The Casper blockchain's ecosystem yearns for a scaling solution to achieve a higher transaction throughput and continue to stay competitive. As a first step towards providing a trustless scaling solution, the goal of the initial version 0.1 of this project, called Kairos, is to build a zero-knowledge (ZK) *validium* for payment transactions in a second layer (L2). This system will both enable a higher transaction throughput and lower gas fees. Here, *validium* refers to the fact that the data, such as account balances, are stored on L2 rather than on L1.

Kairos V0.1 serves two major purposes:

- It is a first step towards a cheap and frictionless NFT minting and transfer system aiding Casper to become *the* blockchain to push the digital art industry forward.
- Its scope allows us to focus the engineering effort and move forward productively. More specifically, it allows exploring the problem space of building an L2 solution that leverages zero-knowledge technology and integrates with Casper's L1. Furthermore, the size and complexity of this project provide an opportunity to get a better understanding of the challenges associated with bringing zero-knowledge into production, as well as allowing the team to collaborate and grow together.

The initial version 0.1 of Kairos will support very few simple interactions and features. Users will be able to deposit and withdraw funds to an L1 contract controlled by Kairos. The L2 is then used to transfer tokens to others. In the remainder of this document, we will detail the requirements of such a system and how we plan to implement and test it.

In Section 2 (Product Overview) we specify the high-level interactions that the proof of concept will implement. Section 3 determines requirements based on the specified interactions to end-to-end test. Next, we provide an abstract architecture in Section 4 (High-Level Design), followed by low-level design considerations in Section 5 and their conclusions in Section 6. After discussing testing concerns in Section 7, we conclude with thread models and a glossary, which clarifies the terminology used throughout this document.

2. Product Overview

To have a common denominator on the scope of the proof of concept, this section describes the high-level features it has to fulfill.

2.1. Features

2.1.1. Deposit money into L2 system

A user should be able to deposit CSPR tokens from the Casper chain to their validium account at any given time through a command line interface (CLI).

2.1.2. Withdraw money from L2 system

A user should be able to withdraw CSPR tokens from their validium account to the Casper chain at any given time through a CLI. This interaction should be made possible without the approval of the validium operator (see Ethereum's validium).

2.1.3. Transfer money within the L2 system

A user should be able to transfer CSPR tokens from their validium account to another user's validium account at any given time through a CLI.

2.1.4. Query account balances

Anyone should be able to query the validium account balances of available CSPR tokens at any given time through a CLI. In particular, users can also query their personal account balance.

2.1.5. Verification

Each transfer must be verified by L1. In addition, at any given time anyone should be able to verify deposits, withdrawals, or transactions. This should be possible through the CLI or application programming interface (API), i.e. a machine-readable way.

2.2. Usage

Version 0.1 can be used by any participants of the Casper network. It will allow any of them to transfer tokens with lower gas fees and significantly higher transaction throughput.

We will offer two user interfaces: A CLI client for users and a L2 API for developers. Thereby, we can serve customers directly while also allowing new projects to build on top of our platform. In addition, this allows customers to generate ZKPs on the client-side by using our L2 API.

Our L2 server itself will be a set of dedicated, powerful machines, including a powerful CPU and GPU, in order to provide GPU accelleration for ZK proving (see Risc0). The machines will run NixOS and require a solid internet connection. The CLI client will run on any Linux distribution.

3. Requirements

Based on the product overview given in the previous section, this section aims to describe testable, functional requirements the validium needs to fulfill.

3.1. Functional requirements

3.1.1. Deposit money into L2 system

- [tag:FRD00] Depositing an amount of `CSPR tokens`, where `CSPR tokens >= min. amount` should be accounted correctly
- [tag:FRD01] Depositing an amount of `CSPR tokens`, where `CSPR tokens < min. amount` should not be executed at all
- [tag:FRD02] A user depositing any valid amount to on its `account` should only succeed if the user has signed the deposit transaction
- [tag:FRD03] A user depositing any valid amount with a proper signature to another users account should fail

3.1.2. Withdraw money from L2 system

- [tag:FRW00] Withdrawing an amount of `CSPR tokens`, where `users account balance >= CSPR tokens > min. amount` should be accounted correctly
- [tag:FRW01] Withdrawing an amount of `CSPR tokens`, where `CSPR tokens < min. amount` should not be executed at all
- [tag:FRW02] Withdrawing an amount of `CSPR tokens`, where `CSPR tokens > users account balance` should not be possible
- [tag:FRW03] Withdrawing a valid amount from the users account should be possible without the intermediary operator of the validium
- [tag:FRW04] Withdrawing a valid amount from the users account should only succeed if the user has signed the withdraw transaction
- [tag:FRW05] Withdrawing a valid amount from another users account should not be possible

3.1.3. Transfer money within the L2 system

- [tag:FRT00] Transfering an amount of `CSPR tokens`, where `users account balance >= CSPR tokens > min. amount` should be accounted correctly
- [tag:FRT01] Transfering an amount of `CSPR tokens`, where `CSPR tokens < min. amount` should not be executed at all
- [tag:FRT02] Transfering an amount of `CSPR tokens`, where `CSPR tokens > users validium account` balance should not be possible
- [tag:FRT03] Transfering a valid amount to another user that does not have a registered validium account yet should be possible.

- [tag:FRT04] Transfering a valid amount to another user sbould only succeed if the user owning the funds has signed the transfer transaction
- [tag:FRT05] When a transfer request is submitted, this request cannot be used to make the transfer happen twice

3.1.4. Query account balances

- [tag:FRA00] A user should be able to see its validium account balance immediately when it's queried through the CLI
- [tag:FRA01] Anyone should be able to see all validium account balances when querying the CLI and API

3.1.5. Verification

- [tag:FRV00] Anyone should be able to query and verify proofs of the validiums state changes caused by deposit/withdraw/transfer interactions at any given time

3.1.6. Storage

- [tag:FRD00] Transaction data should be served read-only to anyone
- [tag:FRD01] Transaction data should be available at any given time
- [tag:FRD02] Transaction data should be written by known, verified entities only
- [tag:FRD03] Transaction data should be written immediately after the successful verification of correct deposit/withdraw/transfer interactions
- [tag:FRD04] Transaction data should not be written if the verification of the proof of the interactions fails
- [tag:FRD04] Transaction data should be stored redundantly

3.2. Non-functional requirements

These are qualitative requirements, such as "it should be fast" and could e.g. be benchmarked.

- [tag:NRB00] The application should not leak any private nor sensitive informations like private keys
- [tag:NRB01] The backend API needs to be designed in a way such that it's easy to swap out a client implementation
- [tag:NRB02] The CLI should load fast
- [tag:NRB03] The CLI should respond on user interactions fast
- [tag:NRB04] The CLI should be designed in a user friendly way

4. High-level design

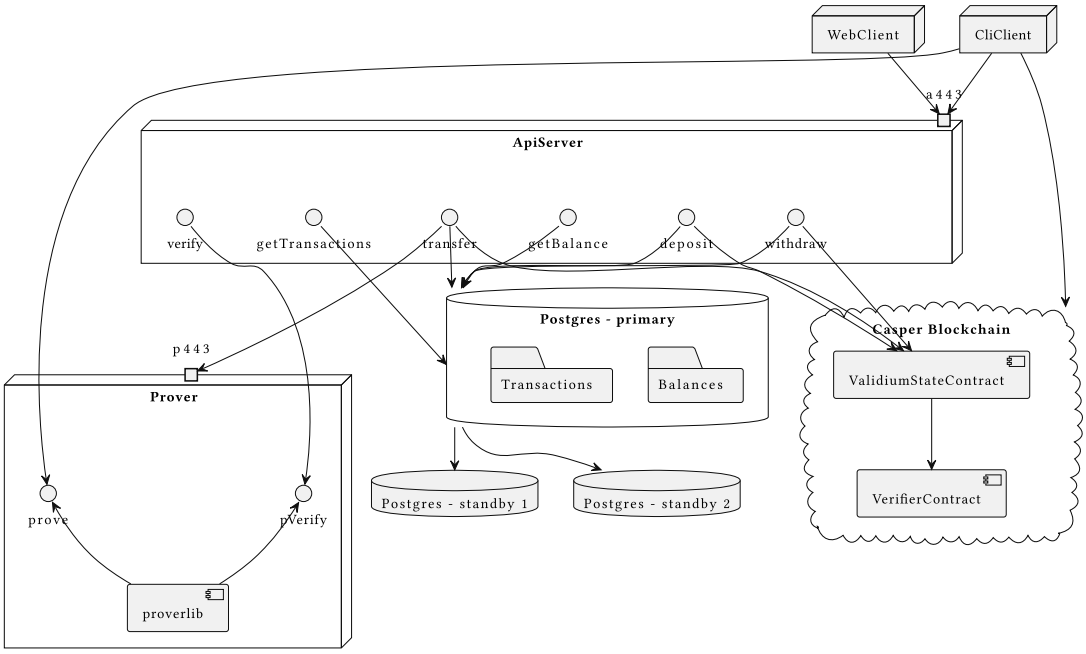

Figure 1: Components diagram

Any ZK validium can be described as a combination of 6 components. For this project's version 0.1, we made the following choices:

- Consensus layer: Casper's L1, which must be able to accept deposits and withdrawals and accept L2 state updates
- Contracts: Simple payments
- ZK prover: Risc0 generates proofs from the L2 simple payment transactions
- Rollup: Risc0 also combines proofs into a compressed ZKR, posted on L1
- L2 nodes: A centralized L2 server connects all other components
- Data availability: The L2 server offers an interface to query public inputs and their associated proofs as well as the validium's current state

From a services perspective, the system consists of four components:

- L1 smart contract: This allows users to deposit, withdraw and transfer tokens
- L2 server: This allows users to post L2 transactions, generates ZKPs and posts the results on Casper's L1, and allows for querying the validium's current state. This is also where the ZKPs and ZKRs are generated.

4.1. L2 server

The L2 server will be ran on three powerful machines running NixOS, as mentioned before. The server code will be written in Rust, to assure performance and simplify interactions with Casper's L1. The database will be postgres, replicated over the three machines.

4.2. Smart contract

The L1 smart contract will be implemented in WASM. Each update of the contract will be accompanied by metadata to confirm the update was made legitimately, such as a ZKP. When a contract endpoint is called, the contract

1. computes the proof's "public inputs", given the contract endpoint call;
2. verifies the update was done correctly, given the public inputs, contract endpoint and metadata, e.g. by running the ZK verifier;
3. if the verification succeeds, the L1 transaction is accepted.

4.3. Prover

There will be a service, running on the same servers as the L2 server, which computes ZK proofs and ZK rollups. This service will be implemented in Rust, using Risc0 to generate ZKPs.

4.4. CLI

4.5. Design decisions

4.5.1. Operating Modes

The CLI should have two operating modes

1. Trusting: the CLI should use the L2 server's endpoints to perform the interactions with the Validium
2. Trustless: the CLI should perform all computations locally except for querying the data availability layer to obtain the Validium's state.

4.5.2. Validium vs. Rollup

As mentioned in the introduction, a ZK rollup is an L2 solution where the state is stored on the L1, while state updates are performed by the L2. A ZK proof, proving that such a state update was performed correctly, is then posted along with the new state on L1. A ZK Validium is similar to a ZK rollup except in that the whole state isn't posted to the L1, but rather a hash of the state. This requires significantly less data resources on L1, and therefore allows further scaling.

We are attempting to create an L2 solution which can scale up to 10,000 transactions per second. However, these transactions need to be independent. The reason for that is that dependent transactions require latter transaction to be aware of the state resulting from the prior transaction. Note: the state is a Merkle-tree root hash of the account balances. Given restrictions such as the time it takes to sign a transaction and send messages around constrained by the speed of light, one is not quick enough to query the prior state. Therefore, in order to reach 10,000 transactions per second you need at least 20,000 people using the L2. This requires 20,000 people's L2

account balances to be stored within the L1 smart contract. However, this amount of data supercedes Casper L1's data limits. In conclusion, any L2 ZK solution on top of Casper must be a Validium.

4.5.3. Centralized L2

Decentralized L2s require many complex problems to be resolved. For example, everyone involved in the L2 must get paid, including the storers and provers. In addition, we must avoid any trust assumptions on individual players, making it difficult to provide reasonable storage options. Instead, this requires a complex solution such as Starknet's Data Availability Committee. Each of these issues takes time to resolve, and doing all this within the project's version 0.1 is likely to prevent the project from ever launching into production. Therefore, a centralized L2 ran by the Casper Association is an attractive initial solution. This poses the question, what are the dangers of centralized L2s?

- Denial of service: The L2 server could block any user from using the system
- Loss of trust in L2: The L2 server could blacklist someone, thereby locking in their funds. This opens up attacks based on blackmail.
- Loss of data: What if the L2 server loses the data? Then we can no longer confirm who owns what, and the L2 system dies a painful death.

Unfortunately there is nothing we can do about the L2 denying you service within a centralized L2 setting. If the L2 decides to blacklist your public key, you will not have access to its functionality. Of course we should keep in mind two key things here:

1. Withdrawing your current funds from the Validium should always be possible, even without permission from the L2.
2. The centralized L2 will be ran by the Casper Association, which has a significant incentive to aid and stimulate the Casper ecosystem to offer equal access to all.

As mentioned before, we will design the system in such a way that withdrawing validium funds is possible without L2 approval. This eliminates the second danger associated with centralized L2s ZKVs, requiring exclusively that you have access to the current Validium state. Without such access, the L2 would be entirely dead, as no deposits or withdrawals can be made without it.

Finally, what if the L2 loses its data? The Casper Association has a very strong incentive to prevent this, since the entire project would die permanently if this occurs. Therefore, we will build the L2 service in such a way as to include the necessary redundancy, as mentioned above.

4.5.4. Privacy provided by L2

We decided not to provide any increased privacy compared to Casper's L1 within version 0.1, since providing any extra privacy would raise AML-related concerns we wish to stay away from, as seen in the famous TornadoCash example.

4.5.5. The L2 server should get paid

Within version 0.1 this issue is rather simple, given the L2 is centralized. All we need to ensure is that the Casper ecosystem grows and benefits from the existence of the L2, and the Casper Association will receive funds to appropriately maintain and extend the L2. Also note that worst-case scenario, as long as the current Valadium state is known, any user can still withdraw their funds from the Validium.

5. Design considerations

In this section, we will describe in detail how each component works individually. Note that these components are strung together into the bigger project according to the diagrams shown in Section 4.

5.1. L2 server

5.1.1. Sequential throughput

In this section, we want to make a quick note about a fundamental restriction of L2 scaling solutions. Imagine you and I both want to submit a swap on a DEX, and you come in first. My transaction thus dependents on the state that results from your swap. There are now two options:

1. I am aware of your output state. In this case, you have created, signed and submitted your transaction to the L2 server. I then pull your output state from the L2 server, construct, sign and submit my transaction. Given limitations such as how long it takes to sign a transaction and to send data back and forth to the server, this setup leads to a maximal sequential throughput[1] of around 1 Tx/s.
2. I am not aware of your output state. In this case, I have to construct a L2 transaction and sign it without being fully aware of its effects yet. In the example of the DEX swaps, I will sign a transaction which does not fully determine how many tokens I will receive back from the DEX, thereby allowing for sandwich attacks and the like. Finally, not mentioning the full state a given L2 transaction depends on in this transaction, leads to the problem of L2 transaction uniqueness: How can you assure that the DEX swap you just signed, won't be executed twice by the L2 server?

[1] Sequential throughput is defined by the number of transactions which can be posted to the L2 where each transaction depends on the output of the former.

Our solution is to keep all L2 transactions rolled up into the same L1 transaction independent of one another, to avoid such complications. In making this decision, we restrict the sequential throughput of your system.

5.1.2. Ensuring L2 transaction uniqueness

We must ensure that each L2 transaction which is posted to the L2 server, can only be used on the L1 once. This can be accomplished in many ways, which generally fall into two categories:

1. Add something to the L2 transaction about the state of the world.
2. Add something to the L2 transaction about the state of the Validium.

The former option is difficut to accomplish, as there are few real-world concepts which translate into the blockchain world easily. For example, naively speaking, time would be a great option: What if we add a timestamp to each L2 transaction? There are two problems with a suggestion like this:

- Time is a very complex concept in the blockchain world. Which `currentTime` should the timestamp be compared to by a casper-node? There is no well-defined time at which a block is added to the blockchain, as each node does so at a different time.
- We must ensure that the bounds on the timestamp are loose enough such that no transactions meant to go into a given ZKR are refused, while never allowing a transaction which was added to the last ZKR to be added to a new one. This requirement of having no mistakes on either end, is so stringent that timestamps don't offer enough information.

The alternative is to add a piece of information X about the Validium's state to each L2 transaction. The ZKR can then include that piece of information as a public input, and check that all L2 transactions have that same public input. However, we must make sure that X is unique, meaning that even if the Validium reverts back to an old state, no old L2 transactions can be reused against the will of the person who signed them. Therefore, we decided to make use of [logical time](https://en.wikipedia.org/wiki/Logical_clock) analogous to [lamport timestamps](https://en.wikipedia.org/wiki/Lamport_timestamp). Meaning that X will be a counter, initialized at 0 and increasing by 1 every time a ZKR is posted to the L1. As such, the ZKR and Validium smart contract can verify perfectly whether a given L2 transaction fits into its rollup, while also providing a simple and clear user interface.

5.1.3. Two phases of the server

The L2 server accumulates a queue of L2 transactions which can be posted into the same ZKR. Based upon a number of limits[2] the server will start computing a ZKR based on its current queue. Any new transactions must now set their Validium counter to one higher than before, in order to fit into the next ZKR rather than the current one. This will be communicated by the L2 server through a clear error.

5.1.4. How will L2 become aware of deposits and withdrawals?

The casper-node allows for creating a web hook. Therefore, by running a casper-node on each L2 server, we can ensure the servers get notified as soon as a deposit or withdrawal is made on the L1 smart contract.

Once the L2 server gets notified about deposits or withdrawals the L2 server will query a Casper node to obtain the respective information to update the Database and provide consistet data availability.

5.1.5. Data redundancy

Due to the nature of validiums, transaction data will be stored off-chain. To ensure interactions can be proven and verified at any given time by anyone, data needs to be available read-only publicly through an API. To reduce the complexity of the project, the data will be stored by a centralized server that can be trusted. Writing and mutating data should only be possible by selected trusted instances/machines. The storage must be persistent and reliable, i.e. there must be redundancies built-in to avoid data loss.

Because losing the Validium state would lead to a loss of all the funds held by the Validium, there needs to be an appropriate amount of redundancy of the stored data. To meet this requirement, we decided rely on PostgreSQL's streaming replication feature (physical replication). The streaming replication feature comes with two crucial benefits we can make use of:

- Fail-over: Meaning that when the primary server fails, one of the replicating standby servers can take over the role of the primary
- Read-only load balancing: Read-only queries can be distributed among several servers

By configuring the streaming replication to be synchronous, we can additionally achieve reliable freshness of the data across all servers. Moreover, it makes the cluster more resilient if the primary server fails after updating. In an asynchronous setting, data could be written to the primary server, which could afterwards fail

[2]Two examples of such limits would be the number of transactions posted into one ZKR, and the time window which is compiled into one ZKR. The latter limit is necessary in order to allow a sensible sequential throughput as well.

before sending the update to the standby server, leading to loss of data. In a synchronous setting, the update to the primary server would fail and require a retry until the update gets replicated across all instances.

The number of standby servers can be arbitrarily increased or decreased. For version 0.1 we decided to use one primary server and two replicating standby servers.

Naively, we might want to consider building a failsafe into the Validium smart contract in case the Validium's state gets lost. After all, such a situation would be disasterous. However, building a failsafe would itself create risk and complexity. Therefore, we opt to focus on building data redundancy as mentioned above, including measures such having the three servers spread out geographically.

5.1.6. Load balance for ZK proving

Within version 0.1, the ZKPs themselves will be sufficiently quick to generate that there is little opportunity for speedup through parallelization. When exploring the ZKR, we should look into parallelization opportunities.

Note that we can limit the number of transactions we accept during a single loop of the system in order to provide a feasible version 0.1. After going into production, we can optimize the server(s)' performance to keep up with demand.

5.1.7. What happens when you post an L2 transaction?

- Check if the transaction is posted with the correct Validium counter, see Section 5.1.3. If not, return a clear error to the API caller.
- Create a TxID and return this to the API caller
- Write the transaction and TxID to the database
- Verify the transaction. If this fails, put the transaction status to Cancelled.
  - Signature
  - TokenID and amount are legitimate, given the current Validium's state
  - The transaction is independent of the current queue of transactions, i.e. the sender and recipient aren't included yet
- Generate a ZKP and write it to the database
- During the server's phase 2, generate the ZKR and post it to the L1 smart contract
- Put the status of all transactions included in this ZKR to Success once the L1 transaction is accepted

Notes:

- Anytime the L2 server posts something to its database, this information is sent to the backup servers.

5.2. Smart contract

Each smart contract endpoint will require the new Merkle root and some form of proof that this Merkle root was correctly computed. After verifying this proof, the smart contract's state will be updated and the necessary actions executed on L1. For more details on the smart contract's state and endpoints, see Section 6.

5.3. ZKP

5.3.1. How do Merkle tree updates work?

Transfer transactions don't have a Merkle tree update themselves. Rather, this duty is taken on by the ZKR. The main reason for this is that we want to avoid transfers from depending on the Merkle root, requiring each transfer in progress to be recreated and resigned anytime a deposit or withdrawal is posted on L1. On the other hand, deposit and withdrawal transaction do require the Merkle tree to be updated. Note that these transactions only change one of the leafs. Therefore, in order to verify whether the old Merkle root has been appropriately transformed into the new Merkle root, all we need is the leaves which the updated leaf interacts with.

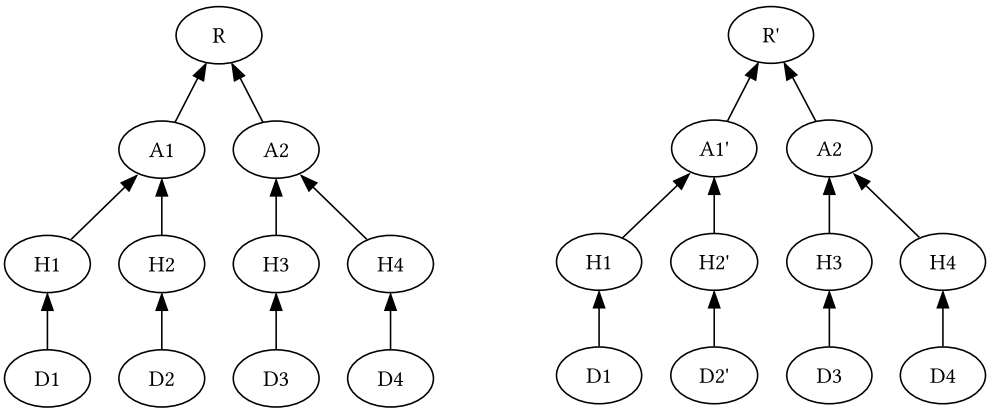

Figure 2: How to update a single leaf of a Merkle tree

Let us look at Figure 2 as an example of a single-leaf Merkle tree update. As we can see, the datum D2 is updated to D2′. As a result, H2, A1 and R each get updated. The deposit transaction itself will include by necessity D2, D2′, R and R', in order to provide the smart contract with all the information necessary in order to execute the right processes. In addition, the smart contract must verify that changing D2 to D2′ does indeed lead to the update of the Merkle tree from root R to root R'. Note now that in order to verify this claim, we don't require the entire Merkle tree. Rather, all we need are values H1 and A2 and the directionality (i.e. the fact that H1 is to the left of H2, whereas A2 is to the right of A1, in the Merkle tree). Given these parameters, we can now check that indeed for

$$\mathrm{H2} = \mathrm{hash}(\mathrm{D2}), \mathrm{A1} = \mathrm{hash}(\mathrm{H1}, \mathrm{H2})$$

$$\mathrm{H2'} = \mathrm{hash}(\mathrm{D2'}), \mathrm{A1'} = \mathrm{hash}(\mathrm{H1}, \mathrm{H2'})$$

it is true that

$$R = \mathrm{hash}(\mathrm{A1}, \mathrm{A2}), \mathrm{R'} = \mathrm{hash}(\mathrm{A1'}, \mathrm{A2}).$$

For a general balanced Merkle tree with N leaves, this requires $\log^2(N)$ hashes, each with their directionality, to be passed along to the Validium smart contract, to allow the verification.

Note: This should be implemented and tested as well for cases where a leaf must be added/removed, rather than updated.

5.3.2. Where does the ZK verification happen?

Within the casper-node, if the ZK verification code doesn't fit into a smart contract? If it does, then within the same smart contract, or a dedicated one?

5.3.3. Comparison of ZK provers

Version 0.1 will be built using Risc0 as a ZK prover system, both for the individual ZKPs and for the rollup. The reason for this is a combination of Risc0′s maturity in comparison to its competitors, and Risc0′s clever combination of STARKs and SNARKs to quickly produce small proofs and verify them. In addition, Risc0 is one of few options which allow for GPU acceleration for the ZKR computation.

6. Low-level design

6.1. L2 server

6.1.1. Deposit Sequence Diagram

Depositing funds to user `Bob`'s account is divided into three phases, which are modelled in the following sequence diagrams.

In the first phase users submit their deposit requests to the L2 server, which updates the Merkle-tree root and creates a *Deploy* which will execute the validation of this update, do the state transition and transfer the funds. This *Deploy* also needs to be signed by the user before submitting since we transfer funds from the users purse to the Validiums wallet.

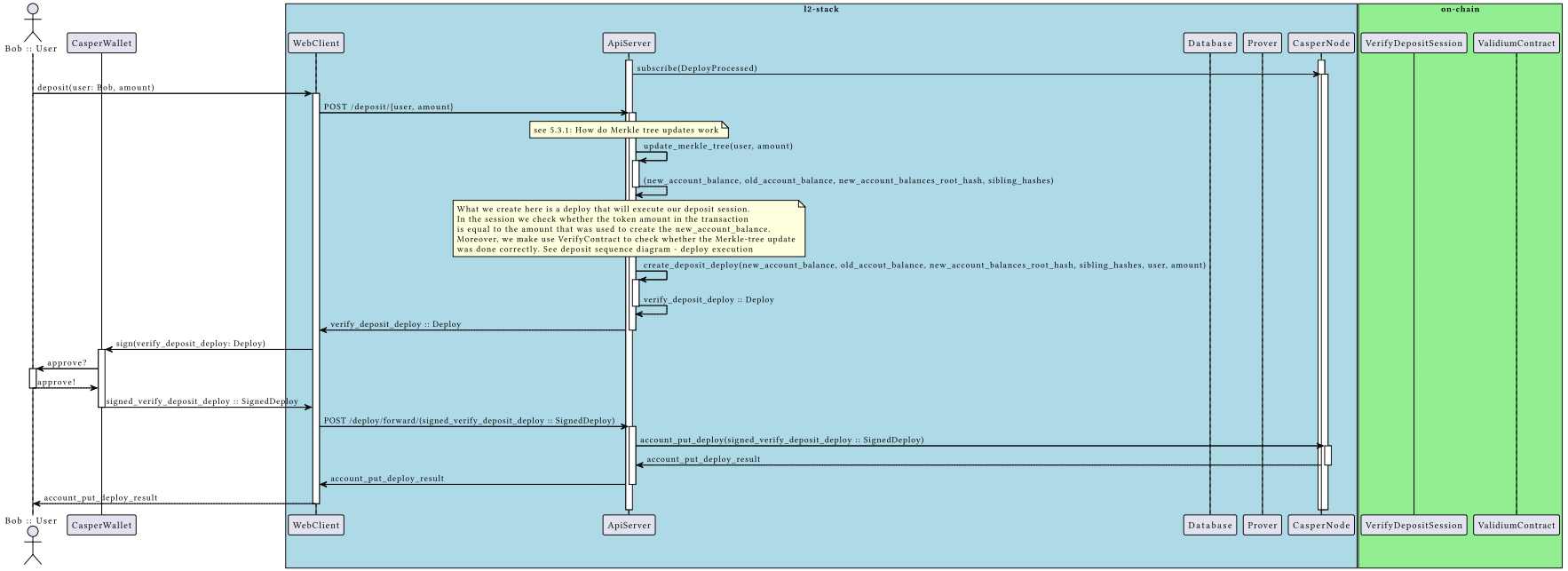

Figure 3: Deposit: User submits a deposit to L2 which gets forwarded to L1.

After submitting, the L1 smart-contracts take care of first validating the new Merkle-tree root hash, updating the validiums state, and transferring the funds.

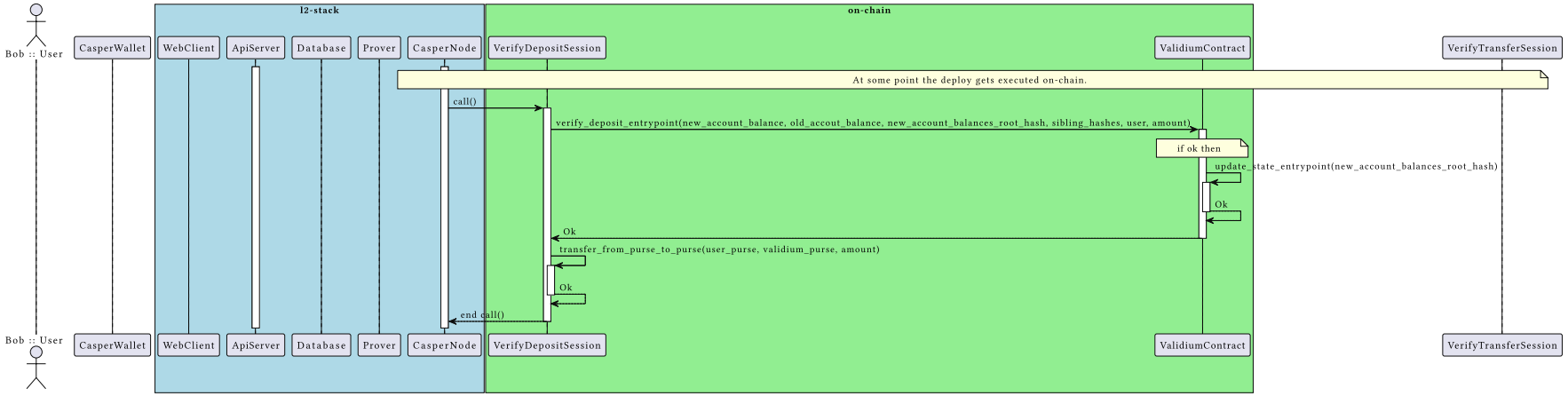

Figure 4: Deposit: Execution of the *Deploy* on L1.

Lastly, the L2 server gets notified when the *Deploy* was processed successfully. The server then commits the updated state to the database and notifies the clients.

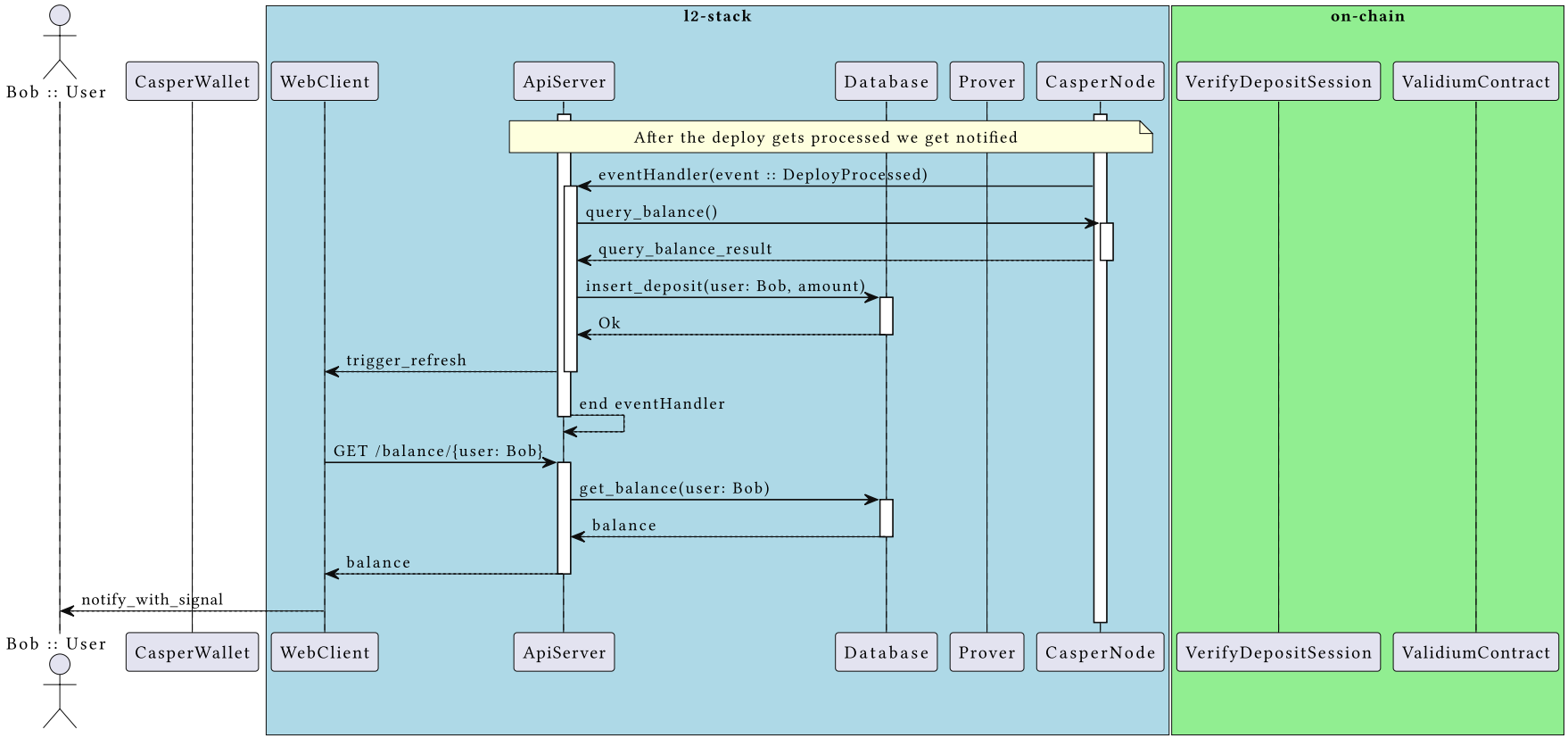

Figure 5: Deposit: Notifying the L2 after succcessfull on-chain execution.

6.1.2. L2 transactions

The content of an L2 transaction is:

- Sender's address
- Receiver's address
- Token ID, i.e. currency
- Token amount
- Sender's signature
- Validium's counter, as discussed in Section 5.1.2

6.1.3. What does the L2 API look like?

- GET /counter returns the Validium counter which new L2 transactions should use in order to be accepted by the server
- GET /accounts/:accountID returns a single user's L2 account balance
- GET /accounts returns the current Validium state, i.e. all L2 account balances
- POST /transfer takes in an L2 transaction in JSON format, and returns a TxID
- GET /transfer/:TxID shows the status of a given transaction: Cancelled, ZKP in progress, ZKR in progress, or "posted in L1 block with blockhash X"
- GET /deposit takes in a JSON request for an L1 deposit and calculates the new Merkle root as well as generating a ZKP for it
- GET /withdraw takes in a JSON request for an L1 withdrawal and calculates the new Merkle root as well as generating a ZKP for it

Note that through the CLI, any user can decide to compute the ZKP necessary for depositing/withdrawing money locally, thereby relying less on the L2 server. This cuts down the dependency on the L2 server to nothing but requesting the current Merkle tree. However, this does require the L2 server to accept ZKPs directly, rather than only L2 transactions, which is a feature for a later version of the project.

6.1.4. L2 Transaction Sequence Diagram

Transfering funds from user `Bob` to a user `Alice` can be divided into four phases, which are modelled in the following sequence diagrams.

In the first phase users submit their transactions to the L2 server, which accumulates them and checks for independence.

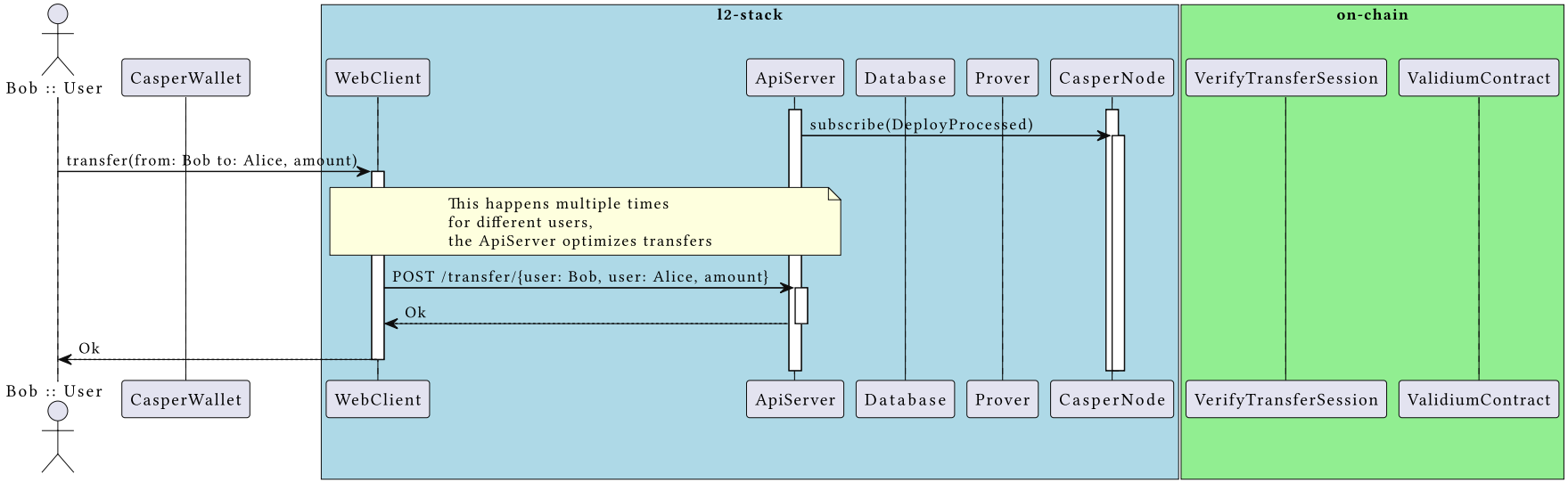

Figure 6: Transfer: User submits a transaction to L2.

After `t` seconds or `n` transactions, the L2 server creates a proof and the according *Deploy* which will execute the validation and the state transition on-chain.

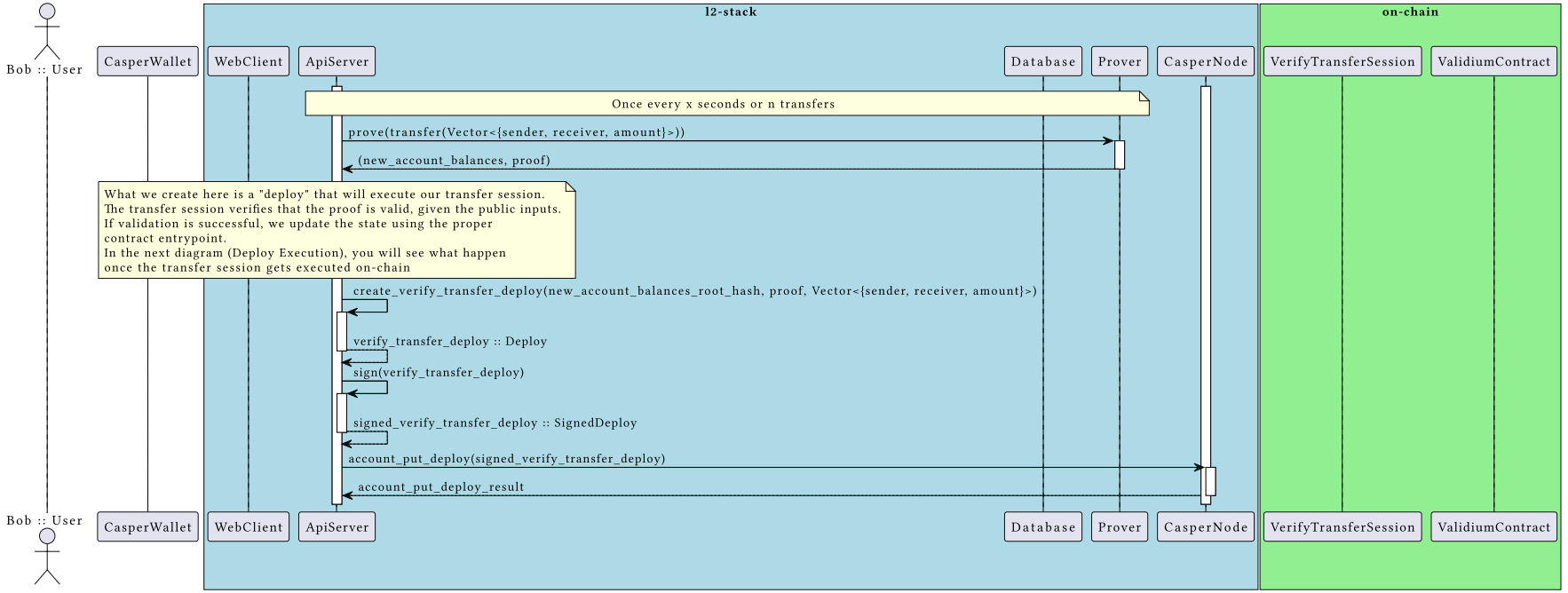

Figure 7: Transfer: Proving and submitting the proof.

After submitting, the L1 smart-contracts take care of first validating the proof and updating the validiums state.

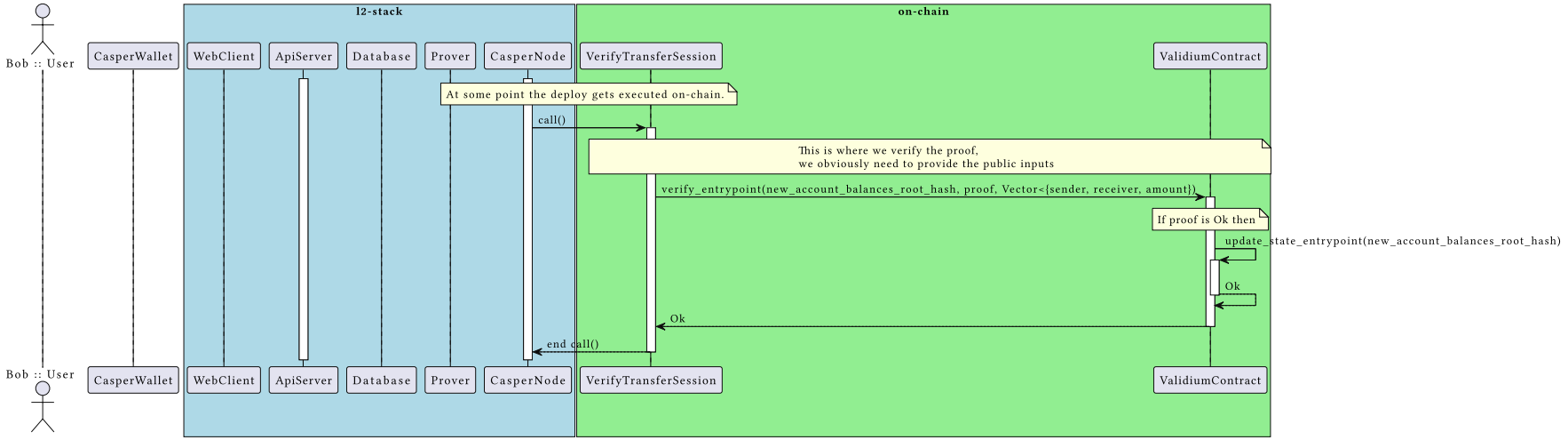

Figure 8: Transfer: Execution of the Deploy on L1.

Lastly, the L2 server gets notified when the *Deploy* was processed successfully. The server then commits the updated state to the database and notifies the clients.

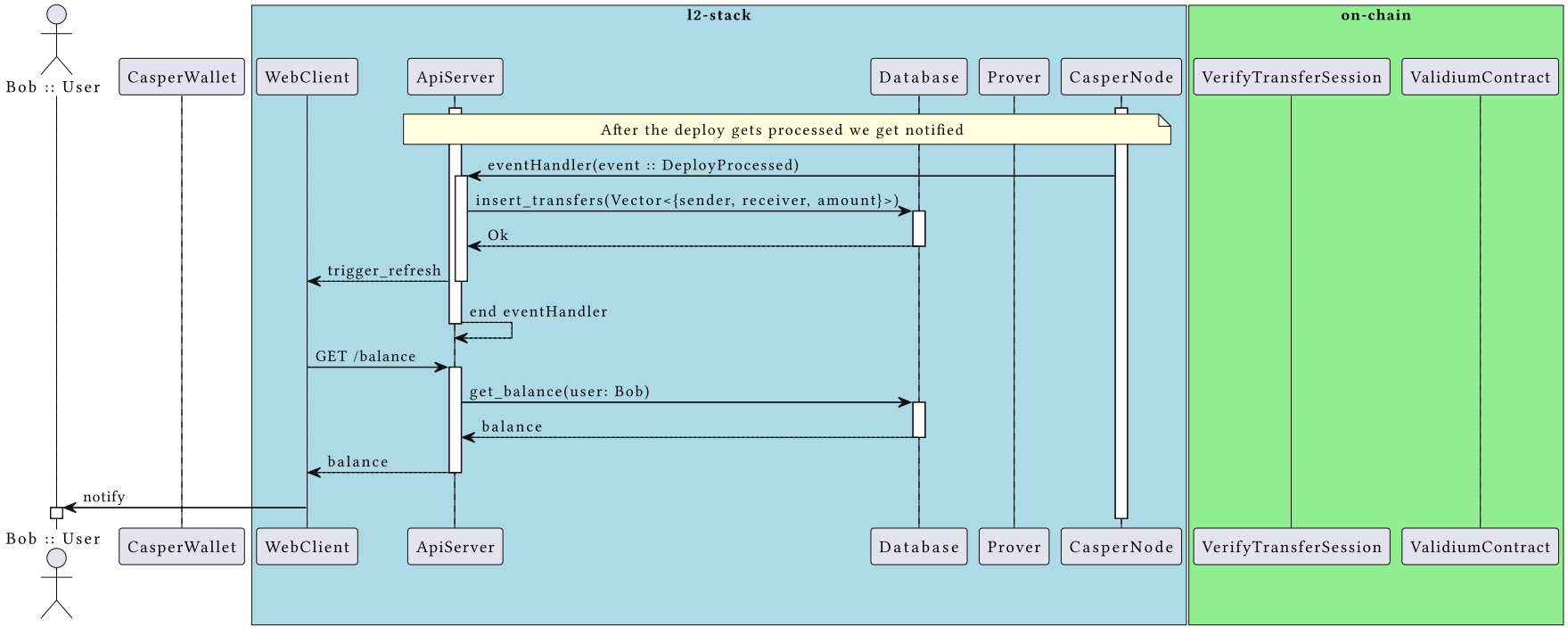

Figure 9: Transfer: Notifying the L2 after succcessfull on-chain execution.

6.2. Smart contract

6.2.1. Smart contract data

The smart contract stores the following data:

- Current Merkle root, representing the Validium's state;
- The Validium counter, see Section 5.1.2
- Its own account balance, which amounts to the total sum of all the Validium account balances.

6.2.2. Smart contract API

The smart contract offers the following API, with each endpoint verifying and acting as described:

- POST deposit: sender's public key, token ID, token amount, new Merkle root, metadata verifying the Merkle root, sender's signature
  - Verify that this amount of tokens can be sent, and move it from the sender's L1 account to the Validium smart contract
  - Verify the sender's signature
  - Verify the new Merkle root given public inputs and metadata
  - Update the smart contract's state
- POST withdrawal: Receiver's public key, token ID, token amount, new Merkle root, metadata verifying the merkle root, receiver's signature
  - Move the appropriate amount of tokens from the Validium smart contract to the receiver's L1 account
  - Verify the receiver's signature
  - Verify the new Merkle root given public inputs and metadata
  - Update the smart contract's state
- POST ZKV: ZKV, new Merkle root
  - Verify ZKV
  - Update the smart contract's state

6.2.3. Smart contract initialization

Initiating the L1 smart contract requires putting money onto the Validium. The balance of the smart contract will then be equal to the initial deposit, and the Merkle roots (current and "current minus deposits/withdrawals") will be equal to the Merkle root for a Merkle tree with only one leaf, namely the single initial account.

6.3. Prover

The zero knowledge proofs for transfer transaction consist of the following:

- Public input: Unsigned L2 transaction

- Private input: L2 transaction signature
- Verify: Signature

The ZK rollup verifies the following:
- The ZKPs don't clash, i.e. they all have separate senders and receivers
- All ZKPs are valid
- All L2 transactions include as a public input the last Merkle root posted on L1 by L2. This Merkle root itself is taken as a public input to the ZKR.
- The old Merkle root is correctly transformed into the new Merkle root through applying all the L2 transactions

Its public inputs are the old and new Merkle root. The private inputs are the list of L2 transactions and their ZKPs, as well as the full old Merkle tree.

6.4. CLI

The CLI offers the following interactions:
- Connect to Casper wallet
- Sign L2 Tx
- Query Validium balance
- Query Casper L1 balance
- Deposit on and withdraw from Validium: Create, sign & submit L1 transaction, check its status on casper-node. This will be possible in two modes: Trusted, where the L2 server does the necessary computations, and trustless, where the L2 is only needed in order to read the Validium state, and the computations are performed locally.
- Transfer within Validium: Query Validium state, create, sign & submit L2 transaction, check its status on L2 server
- Query last N ZKP/ZKRs posted to L1
- Verify ZKP/ZKRs

7. Testing

7.1. E2E testing

7.2. Integration testing

7.3. Testing the smart contract

7.4. Attack testing

7.5. Property testing

- Test our assumption that we don't need Merkle tree rebalancing. If this fails, research and implement Merkle tree rebalancing, generate ZKPs for it and include this as an endpoint to the Validium smart contract.

7.6. Whatever else Syd can come up with

7.7. Thread model

7.8. Glossary

- L1
- L2
- ZKP
- Merkle tree
- Validium
- ZK rollup
- Batch proof