Kairos: Zero-knowledge Casper Transaction scaling

Marijan Petricevic, Nick Van den Broeck

Friday November 3, 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 the Kairos project is to build a zero-knowledge (ZK) *validium* [1] 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 a rollup where the data, such as account balances, are stored on L2 rather than on the Casper blockchain directly (L1).

Additionally, Kairos V0.1 serves two other major purposes:

- It is the first step towards a cheap and frictionless NFT (non-fungible token) minting and transfer system aiding Casper to become *the* blockchain to push the digital art industry forward.
- The conciseness and complexity of its scope allow us to explore the problem space of L2 solutions which leverage zero-knowledge technology and integrate with Casper's L1. Furthermore, it allows the team to collaborate and grow together by building a production-grade system.

Kairos V0.1 will support very few simple interactions and features. Users will be able to deposit and withdraw funds by interacting with an L1 smart contract controlled by Kairos. Transfers of funds to other participants will be serviced by the L2 and verified and stored by the L1. In the remainder of this document, we will detail the requirements of such a system.

In Section 2 (Product Overview) we describe the high-level features that Kairos V0.1 will support. Section 3 specifies the requirements based on the described interactions and features. Next, in Section 4 (Architecture) we propose an architecture of this initial version, together with the component interfaces and their interactions. We conclude the document with threat models and a glossary, which clarifies the terminology used throughout this document. Note that this specification comes with a number of blogposts detailing some of the design considerations in more detail, as listed in the bibliography [2] [3].

2. Product Overview

To have a common denominator on the scope of Kairos V0.1, this section describes the high-level features it has to provide.

2.1. Features

2.1.1. Deposit money into L2 system

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

2.1.2. Withdraw money from L2 system

Users should be able to withdraw CSPR tokens from their Kairos account to the Casper chain at any given time through a CLI. This interaction should not require the approval of the operator (see Ethereum's validium).

2.1.3. Transfer money within the L2 system

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

2.1.4. Query account balances

Anyone should be able to query the Kairos 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

Anyone should be able to verify deposits, withdrawals, or transactions either through a CLI or application programming interface (API), i.e. a machine-readable way.

2.2. Product Application

2.2.1. Target Audience

The target audience comprises users familiar with blockchain technology and applications built on top of the Casper blockchain.

2.2.2. Operating Environment

The product's backend will be deployed on modern powerfull machines equipped with a powerful graphics processing unit (GPU) and a large amount of working memory as well as persistent disk space. The machines will have continuous access to the Internet.

The CLI will be deployed on modern, potentially less powerfull hardware.

3. Requirements

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

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 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
- **[tag:FRD04]** When a user submits a deposit request, the request cannot be used more than one time without the users approval

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 system
- **[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
- **[tag:FRW06]** When a user submits a withdraw request, the request cannot be used more than one time without the users approval

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 account` balance should not be possible
- **[tag:FRT03]** Transfering a valid amount to another user that does not have a registered 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 user submits a transfer request, the request cannot be used more than one time without the users approval

3.1.4. Query account balances

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

3.1.5. Verification

- **[tag:FRV00]** Anyone should be able to query and verify proofs of the system's 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:FRD05]** Transaction data should be stored redundantly [4]

3.2. Non-functional requirements

These are qualitative requirements, such as "it should be fast" and could 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
- **[tag:NRB05]** The L2 should support a high parallel transaction throughput[1]

[1]Read [5] for more insight into parallel vs. sequential transaction throughput.

4. Architecture

Kairos's architecture is a typical client-server architecture, where the server (backend) has access to a blockchain network. The client is a typical CLI application. The backend consists of 6 components, whose roles are described in more detail in Section 4.1. Figure 1 displays the interfaces and interactions between the components of the system.

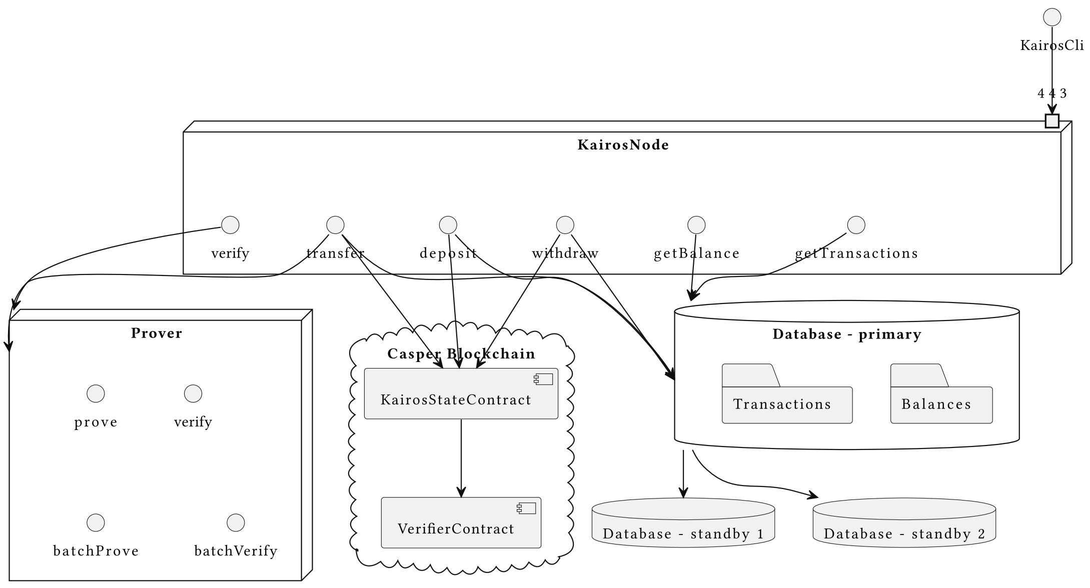


Figure 1: Components diagram

4.1. Architecture Components

4.1.1. Kairos CLI (CLI client)

The Kairos CLI offers a simple user interface (UI) which provides commands to allow a user to deposit, transfer and withdraw funds allocated in their Kairos account. Once a user submits either of the transactions, the client delegates the bulk of the work to the Kairos node (Section 4.1.2.1). It can also be used to verify past state changes and to query account balances.

4.1.2. Backend

4.1.2.1. Kairos Node (L2 Server)

As stated in the introduction of this section, Kairos V0.1 has a client-server architecture. Therefore the Kairos node acts as a centralized L2. The reasoning behind this decision, potential dangers, and our methods for dealing with these dangers are explained in [6].

The Kairos node is the backend interface, through which external clients (Section 4.1.1) can submit deposits, transfers, or withdrawals of funds. It is moreover connected to a database (Section 4.1.2.3) to persist the account balances, whose state representation[2] is maintained on-chain. State transitions of the account balances need to be verified and performed on-chain, requiring the node to create the relevant transactions on L1. These transactions call the respective endpoints of the smart contracts described in Section 4.1.2.4 to do so. For performing and batching transfers the node utilizes a proving system provided by the Prover service (Section 4.1.2.2). For deposits and withdrawals, the node creates according Merkle tree updates of the account balances [7].

4.1.2.2. Prover

The Prover is a separate service that exposes a *prove* and a *verify* endpoint, which will mainly be used by the Kairos node (Section 4.1.2.1) to prove batches of transfers. Under the hood, the Prover utilizes a zero-knowledge proving system that computes the account balances resulting from the transfers within a batch and a prove of correct computation.

4.1.2.3. Database

The database is a persistent storage that stores the performed transfers and the account balances whose state is stored on the blockchain. To achieve more failsafe and reliable availability of the data, it is replicated sufficiently often. In the case of failure, standbys (replicas) can be used as new primary stores [4].

4.1.2.4. Kairos State/ Verifier Contract

The Kairos State and Verifier Contract are responsible for verifying and performing updates of the Merkle root of account balances. They can be two separate contracts or a single contract with several endpoints. The important thing is that the state update only happens if the updated state was verified successfully beforehand. The contracts are called by the Kairos node by creating according transactions and submitting them to a Casper node.

In order for the Merkle tree root to have an initial value, the Kairos State Contract will be initialized with an deposit. This initial deposit then becomes the balance of the system.

4.2. APIs

The following sections describe the APIs of the previously described components.

4.2.1. Kairos Node (L2 Server)

[2]The state representation is the Merkle root, see Section 6.

Endpoint	Description
`GET /counter`	Kairos counter, see Section 6
`GET /accounts/:accountID`	Returns a single user’s L2 account balance
`GET /accounts`	Returns the current Kairos state, i.e. all L2 account balances
`POST /transfer`	Takes an L2 transfer in JSON format, and returns a TxID
`GET /transfer/:TxID`	Returns the status of a given transfer: Cancelled, ZKP in progress, batch proof in progress, or “posted on L1 with blockhash X”
`GET /deposit`	Takes a JSON request for an L1 deposit and calculates the new Merkle root and accompanying data needed to verify the new Merkle root, see Section 6
`GET /withdraw`	Takes a JSON request for an L1 withdrawal and calculates the new Merkle root and accompanying data needed to verify the new Merkle root

4.2.2. Kairos State/ Verifier Contract

Endpoint	Description
`POST deposit(sender's public key, token ID, token amount, new Merkle root, metadata needed to verify 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 purse owned by Kairos • Verify the sender’s signature • Verify the new Merkle root given public inputs and metadata • Update the system’s on-chain state
`POST withdrawal(Receiver's public key, token ID, token amount, new Merkle root, metadata needed to verify the merkle root, receiver's signature)`	• Move the appropriate amount of tokens from the purse owned by Kairos to the receiver’s L1 account • Verify the receiver’s signature • Verify the new Merkle root given public inputs and metadata • Update the system’s on-chain state
`POST batch_proof:(batch proof, new Merkle root)`	• Verify batch proof • Update the system’s on-chain state

4.2.3. CLI

The CLI offers the following commands:

Command	Description
`accounts`	Returns all Kairos (L2) account balances
`accounts <accountId>`	Returns a users Kairos account balance
`deposit <from-address> <amount> <key_pair>`	Creates a deposit request for the Kairos node's deposit endpoint
`withdraw <to-address> <amount> <key_pair>`	Creates a withdraw request for the Kairos node's withdraw endpoint
`transfer <from-address> <to-address> <amount> <key_pair>`	Creates a withdraw request for the Kairos node's withdraw endpoint
`verify <Vector of Transfers>`	Verifies batched transfers

4.3. Data

4.3.1. Kairos CLI/ Kairos Node

4.3.1.1. Deposit (L1)

- Depositor's address
- Depositor's signature
- Token amount
- Token ID, i.e. currency

4.3.1.2. Withdraw (L1)

- Withdrawer's address
- Withdrawer's signature
- Token amount
- Token ID, i.e. currency

4.3.1.3. Transfer (L2)

- Sender's address
- Receiver's address
- Token amount
- Token ID, i.e. currency
- Sender's signature
- Kairos counter

4.3.2. Kairos State/ Verifier Contract

- Current Merkle root, representing the Kairos system's state
- Kairos counter

- Its own account balance, which amounts to the total sum of all the Validium account balances

4.3.3. Prover

The zero knowledge proofs for transfer transaction consist of the following:
- Public input: Unsigned L2 transaction
- Private input: L2 transaction signature
- Verify: Signature

For the ZK rollup:
- Public inputs: Old and new Merkle root and the Kairos counter
- Private inputs: The list of L2 transactions and their ZKPs, as well as the full old Merkle tree
- Verify:
  - The ZKPs don't clash, i.e. they all have separate senders and receivers
  - All ZKPs are valid
  - All L2 transactions include the same Kairos counter as the batch proof
  - The old Merkle root is correctly transformed into the new Merkle root through applying all the L2 transactions

4.4. Component Interaction

The following two sequence diagrams show how these individual components interact together to process a user's `deposit` and `transfer` request.

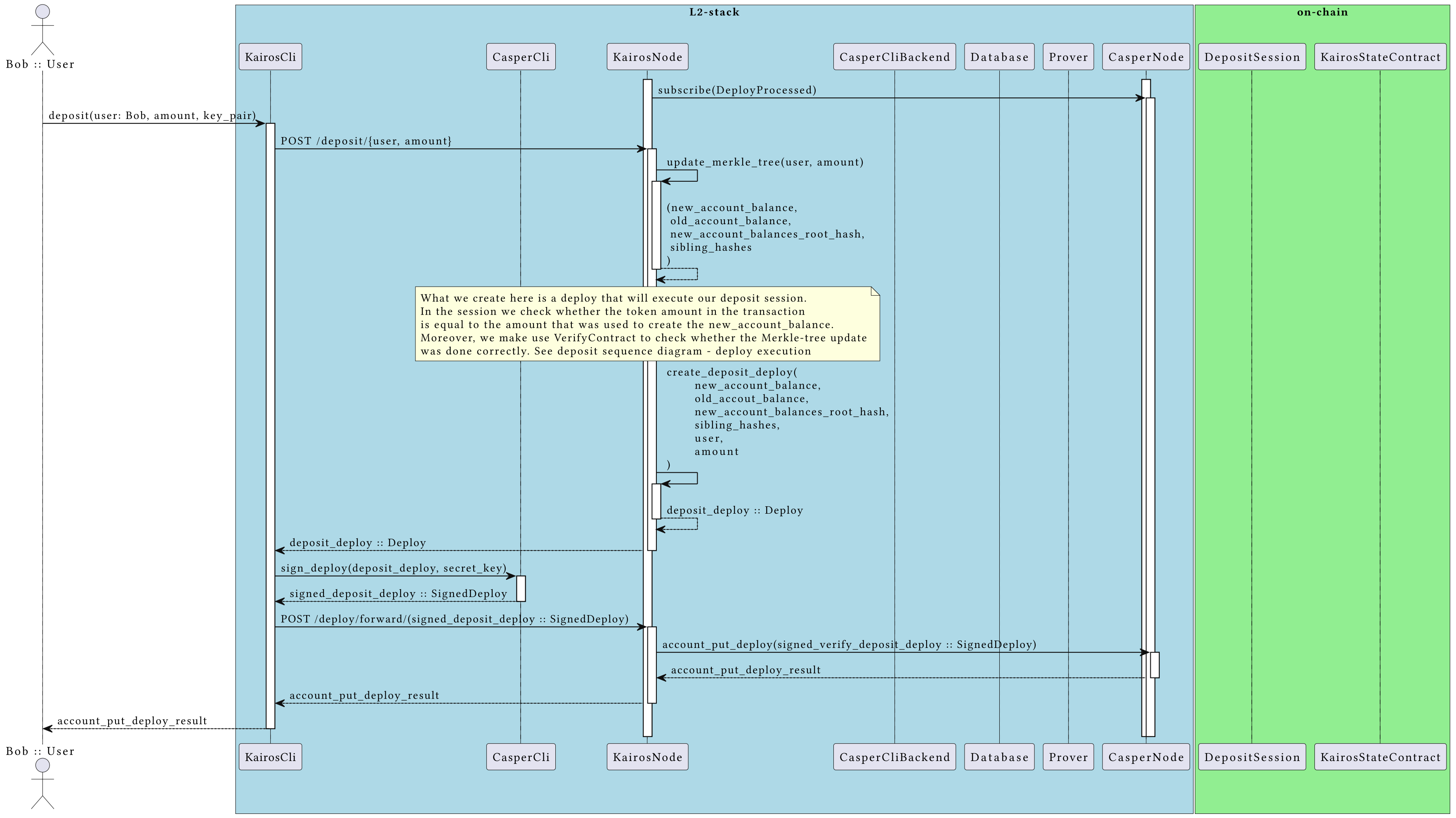
4.4.1. Deposit Sequence Diagram

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

In the first phase (Figure 2) users submit their deposit requests through the Kairos CLI to the Kairos node (L2 server), which updates the Merkle root and creates a *Deploy*. This *Deploy* contains a *Session* to execute the validation of this Merkle tree update, perform the state transition and transfer the funds from the users purse to the Kairos purse. This *Deploy* also needs to be signed by the user before submitting, which can be accomplished by calling the Casper CLI.

After submitting, the L1 smart contracts take care of validating the new Merkle root, updating the Kairos state, and transferring the funds (Figure 3).

Lastly (Figure 4), the Kairos node gets notified after the *Deploy* was processed successfully. The node then commits the updated state to the database. After sufficient time has passed, the user can query its account balance using the Kairos CLI.


L2-stack
on-chain
Bob :: User
KairosCli
CasperCli
KairosNode
CasperCliBackend
Database
Prover
CasperNode
DepositSession
KairosStateContract
subscribe(DeployProcessed)
deposit(user: Bob, amount, key_pair)
POST /deposit/{user, amount}
update_merkle_tree(user, amount)
(new_account_balance,
old_account_balance,
new_account_balances_root_hash,
sibling_hashes
)
create_deposit_deploy(
new_account_balance,
old_accout_balance,
new_account_balances_root_hash,
sibling_hashes,
user,
amount
)
deposit_deploy :: Deploy
deposit_deploy :: Deploy
sign_deploy(deposit_deploy, secret_key)
signed_deposit_deploy :: SignedDeploy
POST /deploy/forward/(signed_deposit_deploy :: SignedDeploy)
account_put_deploy(signed_verify_deposit_deploy :: SignedDeploy)
account_put_deploy_result
account_put_deploy_result
account_put_deploy_result
Bob :: User
KairosCli
CasperCli
KairosNode
CasperCliBackend
Database
Prover
CasperNode
DepositSession
KairosStateContract

What we create here is a deploy that will execute our deposit session.
In the session we check whether the token amount in the transaction
is equal to the amount that was used to create the new_account_balance.
Moreover, we make use VerifyContract to check whether the Merkle-tree update
was done correctly. See deposit sequence diagram - deploy execution

Figure 2: Deposit: User submits a deposit to L2 which gets forwarded to L1.

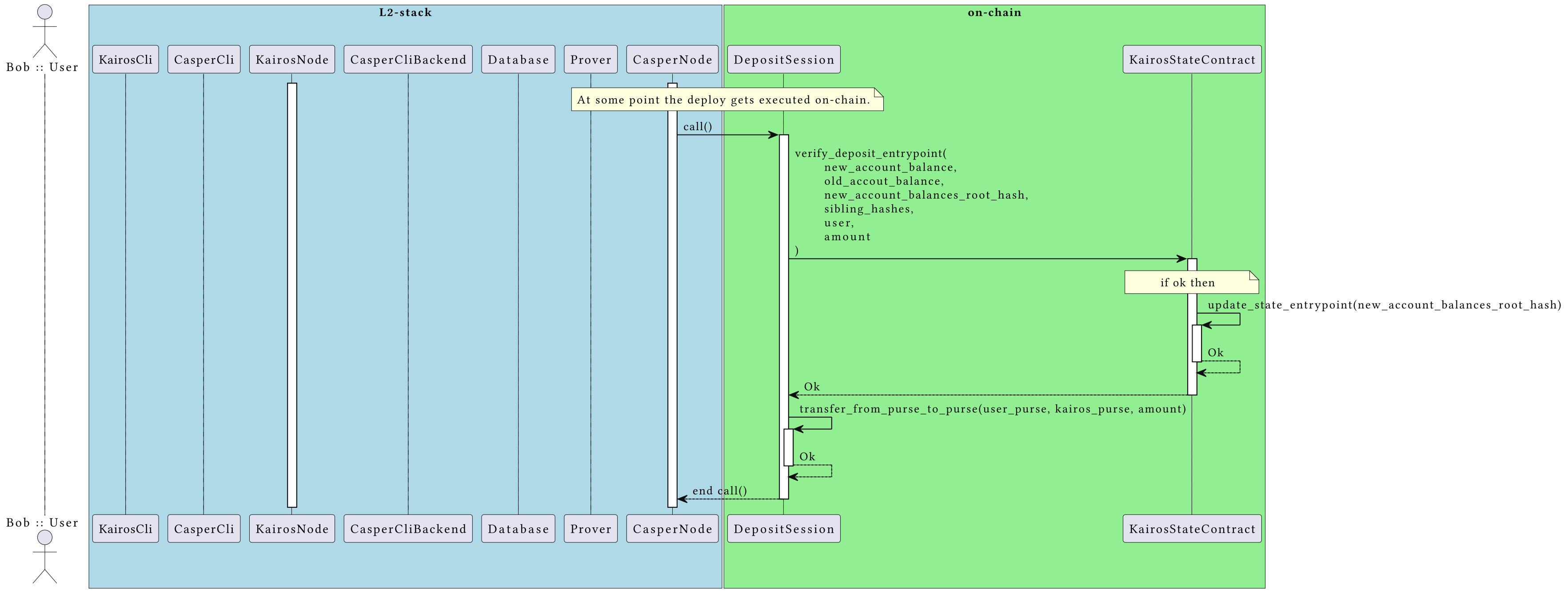


Figure 3: Deposit: Execution of the *Deploy* on L1.

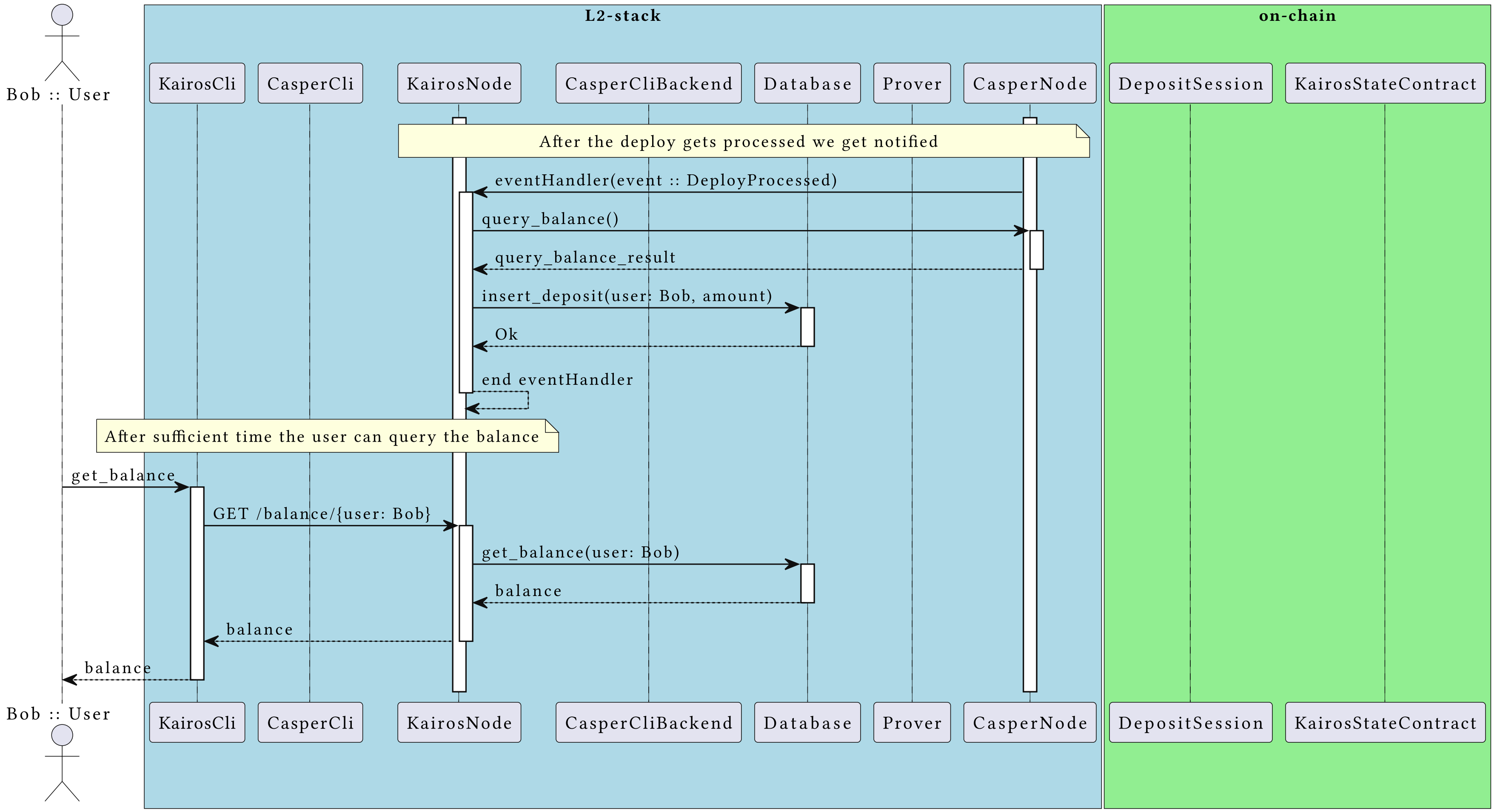


Figure 4: Deposit: Notifying the L2 after succcessfull on-chain execution.

4.4.2. Transaction Sequence Diagram

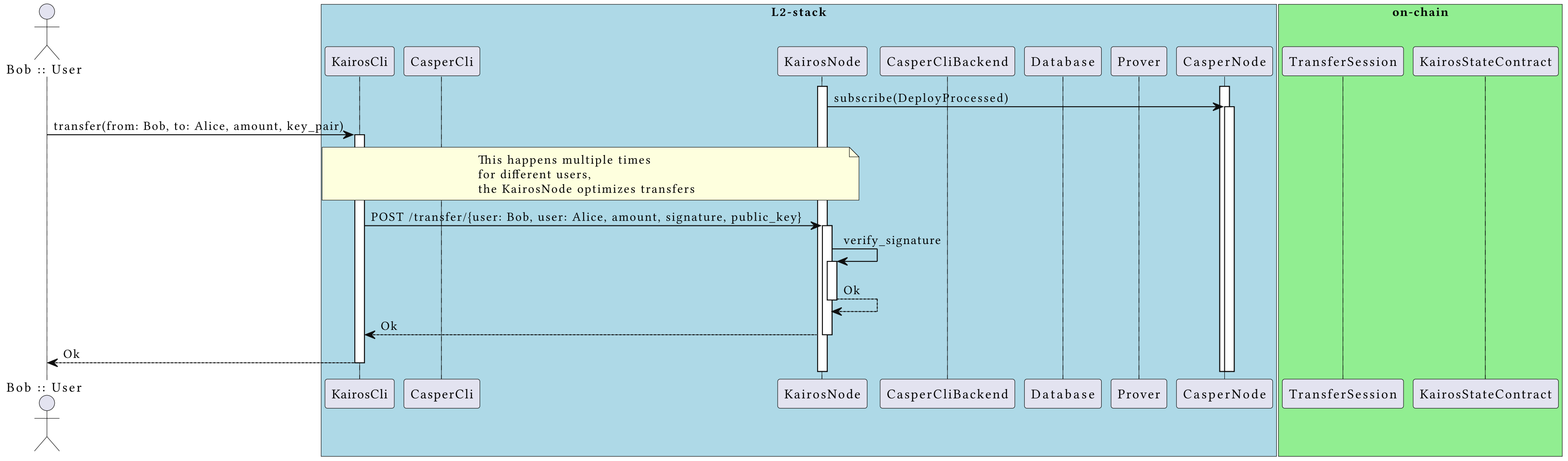
Transfering funds from user `Bob` to a user `Alice` can be divided into four phases, which are modelled in the following sequence diagrams.

In the first phase (Figure 5) users submit their transactions through the Kairos CLI to the Kairos Node (L2 server), which accumulates them and checks for independence. In addition, the Kairos node will check that the batch proof which is going to be computed next, has the same value for the `Kairos counter` as the submitted transaction.

After `t` seconds or `n` transactions (Figure 6), the Kairos node creates a proof and the according *Deploy* which will execute the validation and the state transition on-chain.

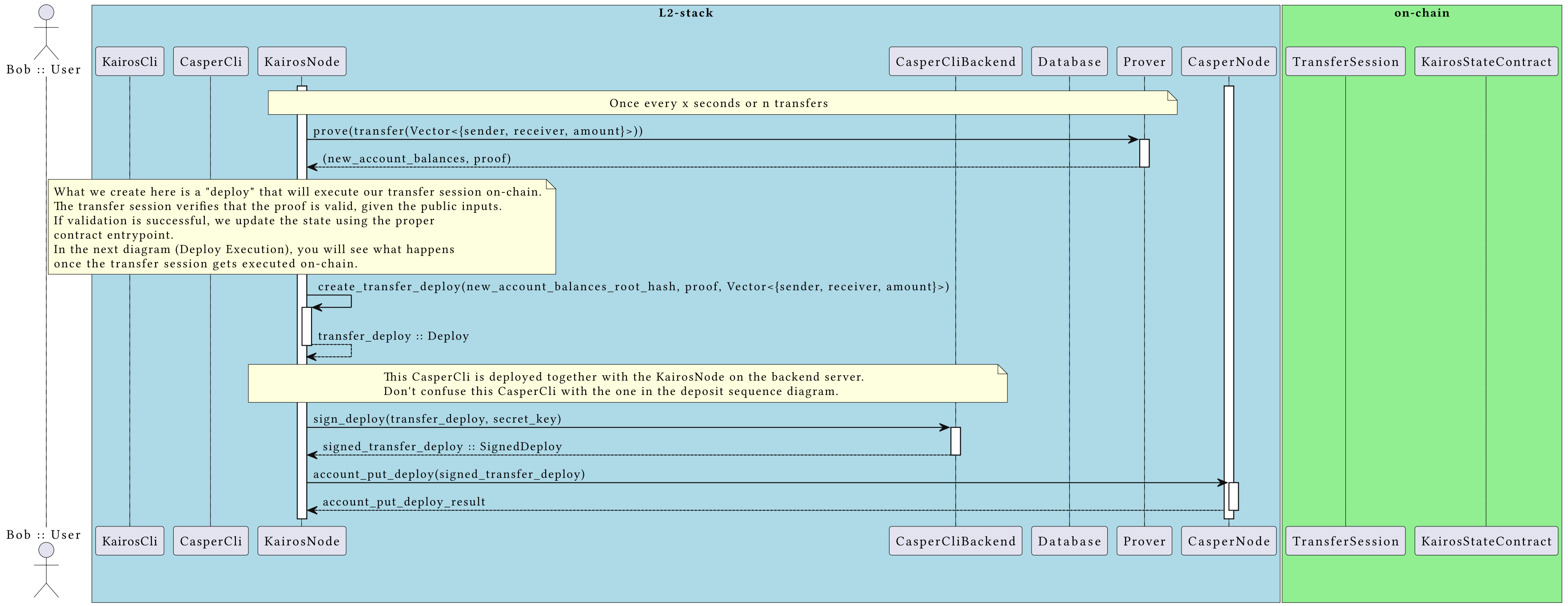
After submitting, the L1 smart contracts take care of first validating the proof and updating the state (Figure 7).

Lastly (Figure 8), the Kairos node gets notified when the *Deploy* was processed successfully. The node then commits the updated state to the database. After sufficient time has passed, the users can query their account balances using the Kairos CLI



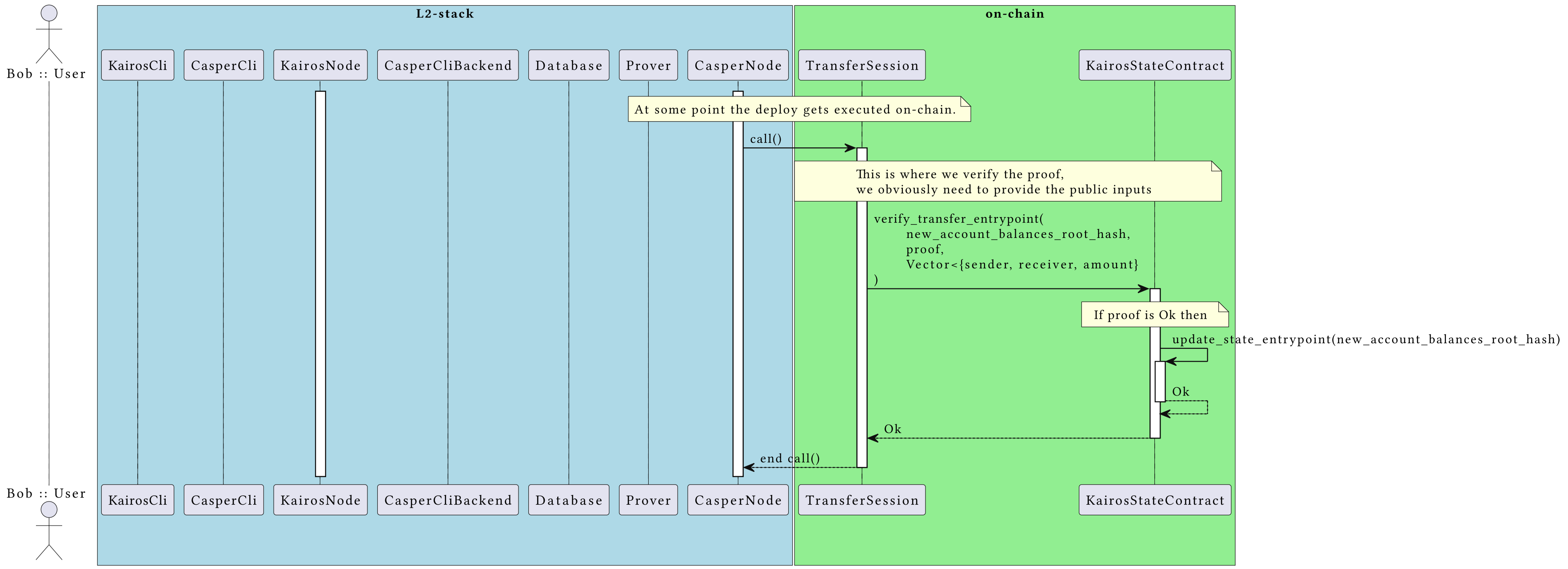
L2-stack
on-chain
Bob :: User
KairosCli
CasperCli
KairosNode
CasperCliBackend
Database
Prover
CasperNode
TransferSession
KairosStateContract
subscribe(DeployProcessed)
transfer(from: Bob, to: Alice, amount, key_pair)
This happens multiple times
for different users,
the KairosNode optimizes transfers
POST /transfer/{user: Bob, user: Alice, amount, signature, public_key}
verify_signature
Ok
Ok
Ok

Figure 5: Transfer: User submits a transaction to L2.



L2-stack
on-chain
Bob :: User
KairosCli
CasperCli
KairosNode
CasperCliBackend
Database
Prover
CasperNode
TransferSession
KairosStateContract
Once every x seconds or n transfers
prove(transfer(Vector<{sender, receiver, amount}>))
(new_account_balances, proof)
What we create here is a "deploy" that will execute our transfer session on-chain.
The transfer session verifies that the proof is valid, given the public inputs.
If validation is successful, we update the state using the proper
contract entrypoint.
In the next diagram (Deploy Execution), you will see what happens
once the transfer session gets executed on-chain.
create_transfer_deploy(new_account_balances_root_hash, proof, Vector<{sender, receiver, amount}>)
transfer_deploy :: Deploy
This CasperCli is deployed together with the KairosNode on the backend server.
Don't confuse this CasperCli with the one in the deposit sequence diagram.
sign_deploy(transfer_deploy, secret_key)
signed_transfer_deploy :: SignedDeploy
account_put_deploy(signed_transfer_deploy)
account_put_deploy_result

Figure 6: Transfer: Proving and submitting the proof.



L2-stack
on-chain
Bob :: User
KairosCli
CasperCli
KairosNode
CasperCliBackend
Database
Prover
CasperNode
TransferSession
KairosStateContract
At some point the deploy gets executed on-chain.
call()
This is where we verify the proof,
we obviously need to provide the public inputs
verify_transfer_entrypoint(
new_account_balances_root_hash,
proof,
Vector<{sender, receiver, amount}
)
If proof is Ok then
update_state_entrypoint(new_account_balances_root_hash)
Ok
Ok
end call()

Figure 7: Transfer: Execution of the *Deploy* on L1.

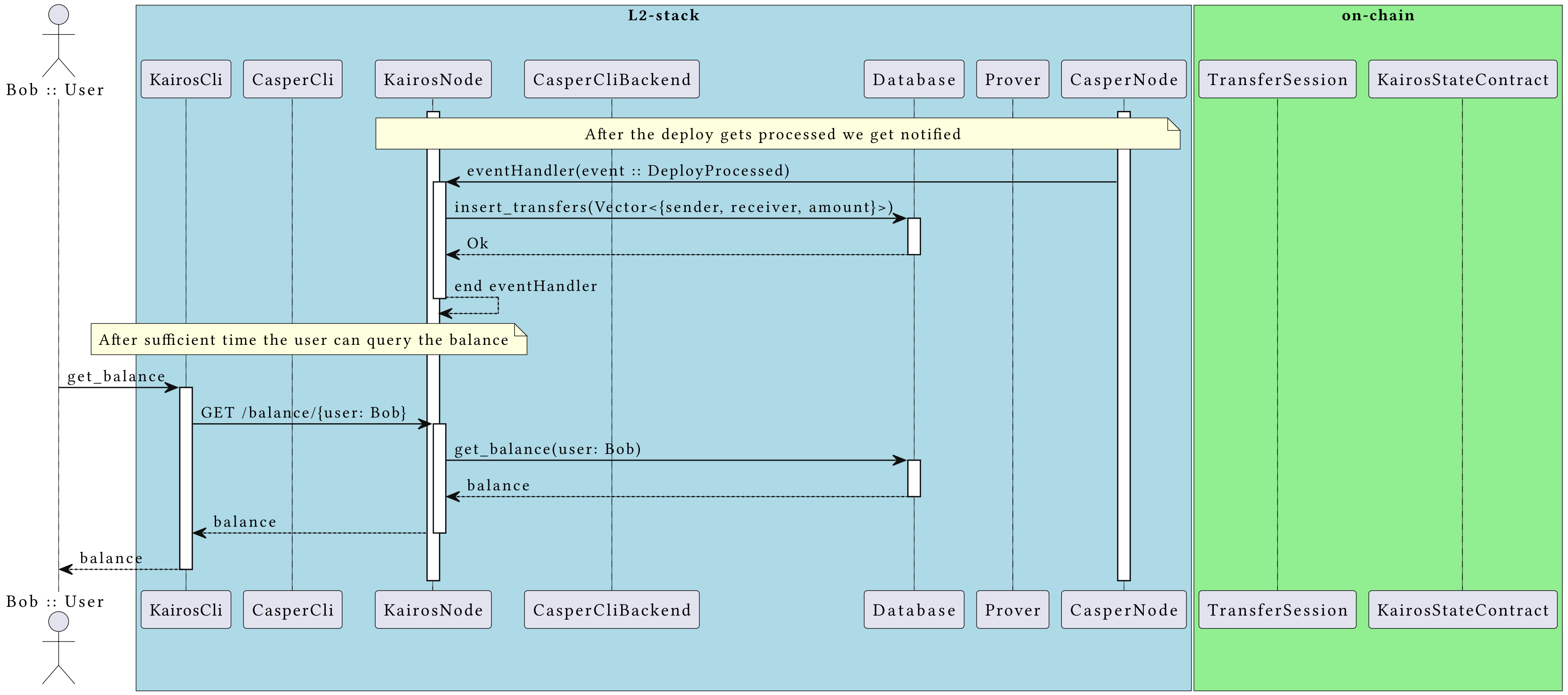


Figure 8: Transfer: Notifying the L2 after succcessfull on-chain execution.

5. Threat model

6. Glossary

Brief descriptions:

- L1: The Casper blockchain as it currently runs.
- L2: A layer built on top of the Casper blockchain, which leverages Casper's consensus algorithm and existing infrastructure for security purposes while adding scaling and/or privacy benefits
- Kairos counter: A mechanism that prevents the usage of L2 transactions more than once without the user's permission. It is added to each L2 transaction, which is verified by the batch proof and L1 smart contract. For an in-depth explanation, see [8].

6.1. Zero Knowledge Proof

In recent decades, a new industry has evolved around the concept of zero knowledge proofs (ZKPs). In essence, the goal of this industry is to allow party A to prove to party B that they are in possession of information X without revealing this information to party B. In practice, this is accomplished by party A generating some proof, called a zero knowledge proof, based on information X, in such a way as to allow party B to verify that the proof (that party A possesses information X). In addition, this zero knowledge proof cannot be used ,in order to gain any information about X other than party A's possession of the information, hence the term "zero knowledge".

This general concept has many applications for two specific reasons: Privacy and scaling. Firstly, zero knowledge proofs allow you to share partial information, retaining your privacy. For example, I could share my birthday in an encoded way (e.g. hashed) and generate a zero knowledge proof that my age is higher than 21, thereby only revealing to you the fact that I am older than 21, and not what my actual age is. Similarly, I could prove to you that I have the password associated to a given Facebook account without actually having to reveal my password.

The second feature of zero knowledge proofs is scalability. Imagine information X is a large amount of data, then it is possible to generate a ZKP proving party A possesses data X, such that the ZKP itself is much smaller than the data X. This feature of ZKPs is particularly interesting to blockchains, as they experience an accute problem: Each transaction posted to a blockchain must, for most blockchains, be verified by each node. This provides a lot of duplicate work and thereby prevents most blockchains from scaling. One solution to this problem is to leverage ZKPs, where one server collects a set of transactions, generates proofs for them and then batches these proofs into one so-called batch proof. This batch proof can then be posted on the blockchain itself ("L1"), together with the related blockchain's state change. This concept constitutes an L2 scaling solution blockchains.

6.2. Merkle tree

A Merkle tree is a cryptographic concept to generate a hash for a set of data. It allows for efficient and secure verification of the contents of large data structures. In addition, Merkle trees allow to quickly recompute the hash (called a "Merkle root") when the data changes locally, e.g. if only one element of a list of data points changes.

We will now briefly explain how to construct a Merkle tree and compute the Merkle root (the "hash" of the data) given a list of data points, as shown in figure Figure 9. First, for each data point, we compute the hash and note that down. These hashes form the leafs of the Merkle tree. Then, in each layer of the tree, two neighboring hashes are combined and hashed again, assigning the resulting value to this node. Eventually the tree ends in one node, the value of which is named the Merkle root.

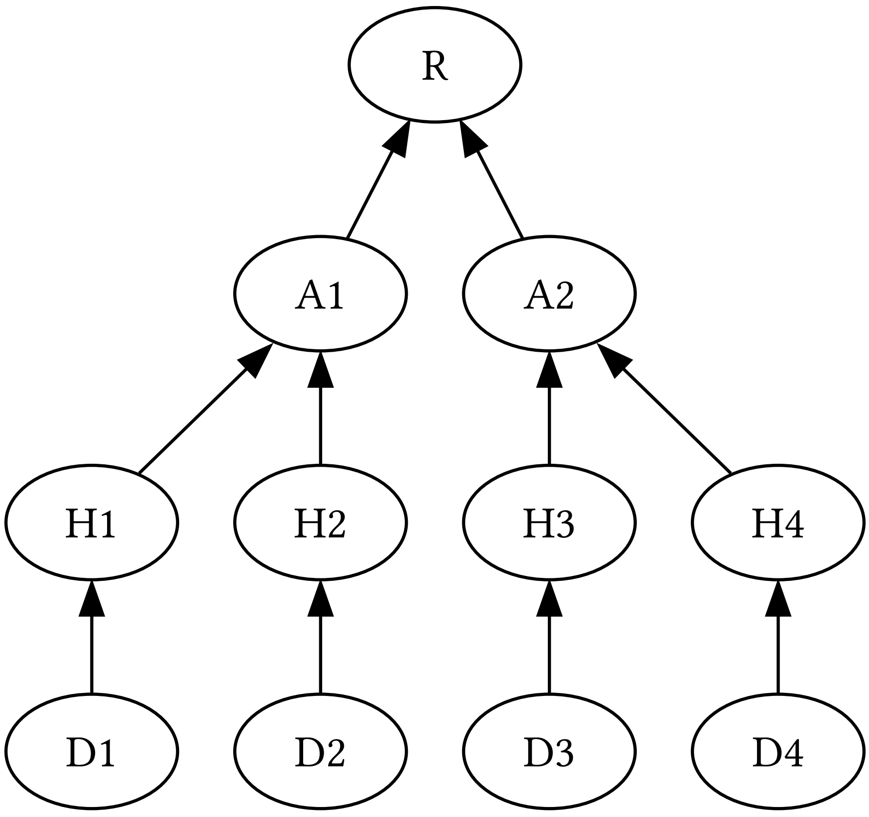


Figure 9: Merkle tree

Bibliography

[1] Nick Van den Broeck, "ZK validium vs. Rollup." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/validium-vs-rollup.md

[2] Nick Van den Broeck, "ZK provers: A comparison." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/compare-zk-provers.md

[3] Nick Van den Broeck, "CLI: Adding a trustless mode." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/trustless-cli.md

[4] Nick Van den Broeck, "Data redundancy." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/data-redundancy.md

[5] Nick Van den Broeck, "Sequential through of l2 ZK validia." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/sequential-throughput.md

[6] Nick Van den Broeck, "Why build a centralized l2?." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/centralized-l2.md

[7] Nick Van den Broeck, "Merkle trees: How to update without a zero-knowledge proof?." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/merkle-tree-updates.md

[8] Nick Van den Broeck, "L2 transaction uniqueness." [Online]. Available: https://github.com/cspr-rad/kairos-spec/blob/main/blogposts/L2-Tx-uniqueness.md