KLASTER

Klaster Whitepaper

May 2024

A novel P2P network standard enabling guaranteed conditional execution of transactions across one or more independent blockchains with a single off-chain signature.

Mislav Javor (mislav@polycode.sh)

Filip Dujmušić (filip@polycode.sh)

With notable contributions from:

Ivan Piton, Luka Sučić

Intro & rationale

Blockchain, born out of the desire to create a global decentralized currency - Bitcoin, has evolved a lot over the fifteen years since it's been developed. The first evolution came in the form of smart contract blockchains. Recognizing the limitations of the "single-app" model of Bitcoin, the "world-computer" model of Ethereum was born. This has allowed developers to build trustless & trust-minimized applications and has resulted in the birth of several new industries - most notably DeFi and NFTs.

Scalability challenges

However, while Ethereum has successfully solved for the "programmability" aspect of blockchain, many challenges have remained to be solved. The most notable of those is - scalability. Ethereum blockchain is notoriously unscalable - with a maximum throughput of ~15 transactions per second. In times of market demand, this can raise the fees for a simple "money transfer" action to well over $50. For more complex transactions, these fees can reach into the $100 - $1000 range. This issue has made DeFi the playground of the well-off and has stood as a barrier towards decentralized services reaching mainstream adoption.

Scaling solutions

The problem of blockchain scalability has had many solutions, some more successful than others. This document will not serve to iterate over all of them, but just mention a few meaningful examples. First examples were the various alternative blockchains, which operated with a few masternodes processing the majority of transactions. Some examples include Polygon PoS & Binance Smart Chain.

The second category were blockchains operating on a different model from Ethereum - implementing alternate virtual machines, sharding and many other scaling techniques. Some examples in this category include NEAR Protocol & Solana. The third category, and the one this paper will focus on the most are so-called "rollups".

Rollups are attached to a "parent" blockchain and use some form of cryptography to "inherit" a part of all of the security of the "parent" blockchain. Most rollups today can broadly be split into two categories - **Optimistic Rollups** and **Zero Knowledge Rollups**.

Some notable examples of rollups include Optimism, Base, Arbitrum, zkSync, Polygon zkEVM, Scroll, etc...

With the methods outlined in the text above, blockchain fees can be reduced to <$0.01 per transaction, which unlocks the potential of blockchain to a much wider audience.

Modular Blockchain & Data Availability

Coming back to rollups - in this paper we will focus on Ethereum-aligned rollups ("EVM Rollups"). In order to achieve the functionality promised by those rollups, a lot of changes needed to be done at the protocol level.

Since neither Ethereum nor its associated virtual machine - the EVM - were originally built with rollups in mind, the developers started adapting the core blockchains and building novel solutions to support functionality of rollups. The most notable one of these solutions was the appearance of "Data Availability" layers - protocols which store the data required by rollups to ensure the security of validated transactions. Some of these protocols include Celestia and NEAR. Even Ethereum itself adapted to support roll ups - with the ERC4844 improvement - called "proto danksharding". While technically different, all of these protocols functionally achieve the same thing - they provide a cheap but ephemeral storage for rollup data - a technical prerequisite for rollups to hit the <$0.01/tx goal.

Scaling state of art

With rollups and data availability - blockchains have mostly been scaled for the needs of users today. Most transactions today can be executed on L2/L3 blockchains for $0.01 - $0.1.

Usability challenges

While scalability has been touted as the biggest challenge towards the mainstream adoption of blockchain technology, it is by no means the only one. The usability of many blockchain architectures is substandard. Again, this paper will explore the Ethereum (EVM) ecosystem, but similar conclusions can be applied to other architectures.

Transaction Fees (Gas)

One of the core features of smart contract blockchain architectures is the concept of "gas". Gas is used to pay for transactions and is usually denominated in a coin which is native to the blockchain on which the transaction is being executed. On Ethereum, this coin is Ether (ETH). Gas is a necessary requirement for smart contract blockchains. It prevents malicious users griefing the nodes which validate the transactions by running expensive computation or infinite loops. However, gas introduces complexity when people are interacting with tokens which are not native to the blockchain on which the transaction is being executed.

One example of such a token is *USDC*. This is a token, deployed on all major programmable blockchain networks which is tied 1:1 to the United States Dollar. Many users interact with these tokens, since they're looking to escape from the value volatility of crypto-only tokens, such as ETH or BTC. However, when a user is sending USDC to someone, they need to have not only USDC, but also the native currency. So in the case of sending ETH on Ethereum, the user must have USDC and ETH.

This means that users must continually maintain the balance of ETH and "seed" every new address that they create with an initial ETH balance. For new users, this is another friction point which requires a learning curve and disincentivizes adoption.

This problem has been explored and approached through many standards. On Ethereum, it's mainly solved by Account Abstraction teams working on standards such as ERC4337 & ERC3074. Later in this paper, we will present a novel approach to this problem.

Elliptic curves

While interacting with smart contract blockchains, users must commit signed transactions to the blockchain. These transactions are based on asymmetric cryptography, notably - elliptic curve cryptography. Each blockchain can choose their own curve, but the most popular one is `secp256k` used by Bitcoin, Ethereum and all EVM rollups. For example, Solana uses `curve25519` and thus validates the cryptographic commitments differently.

Elliptic curve cryptography isn't specific to blockchains only, and many interesting solutions exist which implement different elliptic curves. One such example is *Passkey Authentication*, which uses `ES256` as its authentication curve. In native blockchain implementations, these alternate methods are not supported - limiting the usability of blockchains.

Account recovery & protections

Handling crypto in self-custody is off-putting to many users since they must take care of preserving their own accounts and protecting themselves from losses. If the users lose access to their private key, they lose access to their funds. While "crypto natives" have adjusted to this way of thinking - it remains a large hurdle towards adoption of crypto self-custody.

Smart contract accounts

Similar to solving the gas fee problem, the solutions to the usage of elliptic curves and account recovery are *Smart Contract Accounts* (SCAs). These accounts can attach arbitrary execution logic to any transaction and as such they can improve the usability of blockchain systems substantially. Some examples of SCAs include:

- **Gas abstraction & sponsorship**: Allowing the users to cover on-chain transaction fees with non-native assets _or_ allowing blockchain applications to sponsor the gas costs for their users.
- **Account recovery**: The ability for users to recover funds if they lose access to their private key. This can be done through a trusted recovery provider or through methods such as social recovery.
- **Custom elliptic curves**: The ability for users to use passkeys or other cryptography which is not native to the core blockchain.
- **Custom execution logic**: The ability for users to customize the execution logic of their blockchain accounts. This includes setting spending limits, authorizing multiple signers, requiring a form of two-factor authentication, etc...

Problems with rollups & smart accounts

While rollups have solved for scalability & smart accounts have solved for usability - they have introduced a new set of problems - which Klaster was created to solve. These problems can broadly be categorized into two main categories:

- **Asset fragmentation:** When users are interacting with multiple rollups, their assets are fragmented and can't be easily accessed by applications. If the user has supplied e.g. USDC to AAVE on Arbitrum and wants to deposit to

a different lending protocol on e.g. Base - they must first unwind their AAVE position, *bridge* their assets and then they call the deposit transaction on the destination chain. The user is required to interact with three separate applications and sign three separate transactions to get the effect they desire.

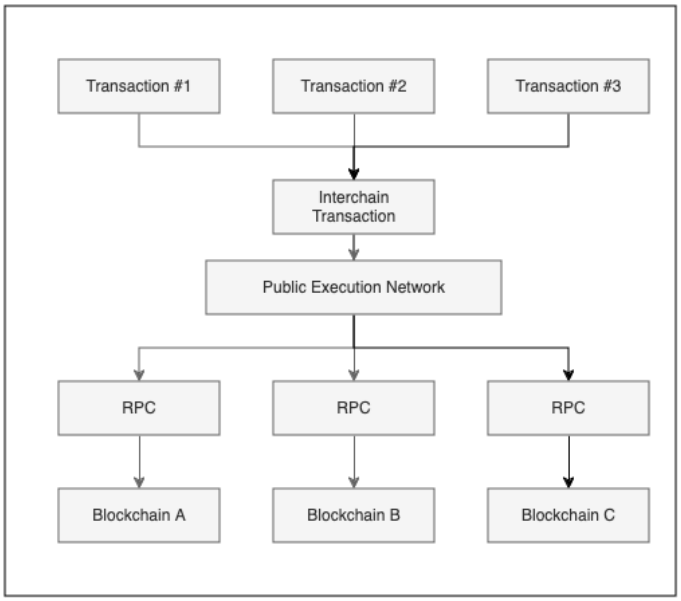
- **Account fragmentation:** If users wish to access the benefits of Smart Contract Accounts that we mentioned earlier, they must deploy separate instances of the same Smart Contract Account on all the blockchains they're using. Thus, the applications which work with these accounts are *not* able to assume that the user has an account available on different blockchains. Beyond this, when the user is changing the authorization or management modules on their smart account - the changes are not distributed to other blockchains. This fragments the accounts which the user has and they have to separately maintain multiple instances of accounts across every rollup they use. If we're moving towards a thousand+ rollup future - this becomes completely unsustainable.

Transaction Commitment Layer

This paper will present a novel mechanism, through which users are able to execute multiple transactions, across multiple independent blockchain networks (L1s & rollups) - with a single off-chain signature.

Separating transaction commitment and execution

To achieve this, we separate *transaction commitment* (sending a signed transaction to the blockchain) from *transaction execution* (blockchain nodes receiving and processing the transaction).



In traditional blockchain execution models, once the user has signed a transaction, the transaction is immediately posted to the blockchain, picked up by the nodes and executed. This design was created for single-chain flows and works quite well - as long as the user is interacting with one blockchain only.

However, when the user/dApp/Wallet wishes to encode actions that traverse multiple blockchains - and those are becoming more and more common as rollups gain market share - they must manually *commit* each transaction to its respective blockchain. *Committing* a transaction includes:

1) Encoding the transaction
2) Signing the transaction with a private key
3) Calling the RPC endpoint of a node for that blockchain network.

What this paper proposes is to create a *separate, non-blockchain* P2P network - which would have the sole purpose of committing transactions to blockchains.

Execution Mechanism

We will start by introducing the protocol technical primitives:

- **Interchain Transaction (iTx)**: An interchain transaction is a bundle of one or more blockchain transactions, to be executed on one or more blockchain networks. Thus, each interchain transaction can contain an arbitrary number of “regular” blockchain transactions. iTx is encoded as a Merkle Tree of “child” transactions. The *iTx hash* is the root hash of that Merkle Tree.
- **Interchain Commitment**: An interchain commitment is the root hash of the *Interchain Transaction* Merkle Tree root signed by a Klaster Node. This *commits* the node to execute all the transactions in that iTx or get slashed by the Klaster protocol.
- **Public Execution Network (PEN)**: The Klaster P2P network serves to execute the Interchain Commitments. It uses staked or re-staked tokens to provide economic guarantees of execution. This will be the Klaster Public Execution Network (PEN)
- **Multichain Smart Account**: A Multichain Smart Account is a Smart Contract Account which is available on *multiple blockchains*. It uses deterministic address derivation to provide the dApps which interact with it an address for every blockchain - which they can have certainty is under the control of the same user. Users are able to derive an arbitrary number of Multichain Smart Accounts from a single “master” account by providing the derivation

function with a Salt parameter.

Executing an interchain transaction

When executing an Interchain Transaction, the Klaster PEN must follow a strict process.

1. Encoding the Interchain Transaction

dApp/Wallet creates an `Interchain Transaction` object. This object contains all the required information for the nodes to execute a sequence of transactions across multiple blockchains. The `Interchain Transaction` object contains:

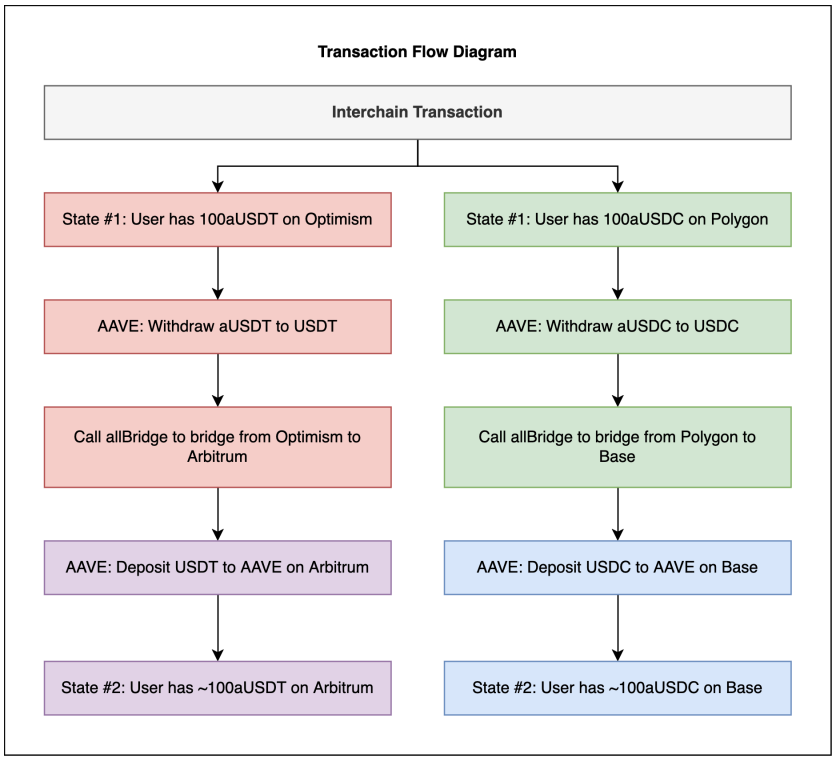
1) **A list of native blockchain transaction objects**.
    a) For EVM, these are ERC4337 `UserOp` objects, with extra information on top. The extra information is:
        i) `ChainID` on which the `UserOp` needs to be executed
        ii) `Salt` parameter, determining which of the derived Multichain Smart Accounts the transaction needs to be executed on.
        iii) `Minimum Block Height` determining the earliest block height on which the node is to execute the transaction.
    b) For every other VM (e.g. Soalana, Near, ...) a transaction encoding standard needs to be defined. This is beyond the scope of this version of the whitepaper.
2) **Payment information**. The user communicates a `Payment Info` object in which they specify how they would like to cover the cost of executing the Interchain Transaction. This can include payments in native gas tokens, payments in some other tokens (ERC20 or otherwise) or even an off-chain payment structure (pay by credit card, prepay transactions with a “gas tank”, subsidized transactions)

Interchain Transaction Example

One example of an interchain transaction would be a transaction which moves a lending position to a blockchain which has the most favorable yield with a single signature.

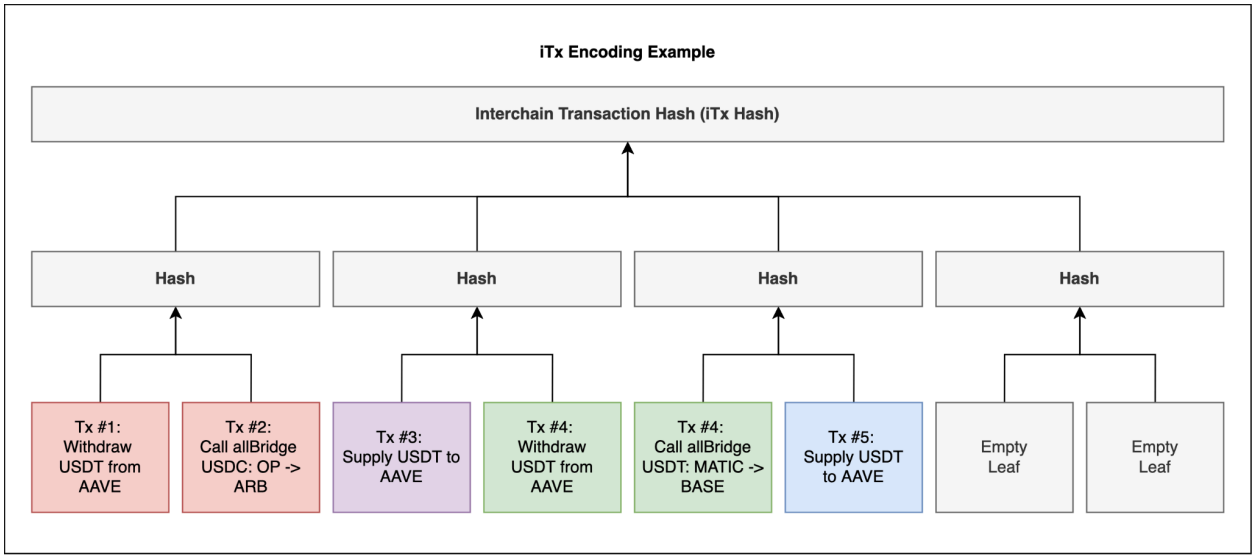
1. Withdraw USDT from AAVE on Optimism
2. Call Across to bridge USDT from Optimism to Arbitrum
3. Supply USDT to AAVE on Arbitrum

4. Withdraw USDC from AAVE on Polygon
5. Call Across to bridge from Polygon to Base
6. Deposit USDC to AAVE on Base



An example of an interchain transaction flow

The interchain transaction will be encoded as a Merkle Tree, where all of the child transactions will be the leafs. The *iTx hash* - is the Merkle Tree root hash of all of the child transactions.



Example of a Merkle Tree encoding for an interchain transaction

2. Sending the Interchain Transaction

The dApp/Wallet is connected to a **Light Node.** A light node is a node in the Klaster PEN which does not execute Interchain Transactions nor does it provide Interchain Commitments. The light node would be owned and operated by the dApp or Wallet developers / operators. The role of the light node is to run auxiliary tasks - such as:

- iTX gas cost simulation
- Maintaining a list of full-node peers
- Providing network gossip to non-committing full nodes for slashing purposes.

In this step, the dApp/Wallet will send the unsigned `Interchain Transaction` to the light node.

3. Interchain transaction pre-processing

The interchain transaction will be pre-processed by the light node. The light node will take all the child transactions and simulate the `gasLimit` for those transactions. If `gasLimit` has been provided by the wallet/dApp - no simulation will be done.

After the transaction has been pre-processed, the light node will pass the unsigned `Interchain Transaction` bundle to its **Full Node** peers in the Public Execution Network.

Full Node Definition: A full node is a node which has the ability to generate Interchain Commitments, has staked tokens in the StakeManager contracts on destination chains and which actively executes Interchain Transactions on one or more chains.

4. Receiving transaction quotes

Once the Full Nodes have received the unsigned `Interchain Transaction` object, they can respond with a `Transaction Quote`.

Transaction Quote Definition: The transaction quote is a *quote* by the Full Node, outlining *under which conditions* it is willing to commit the transactions within the `Interchain Transaction` object to the blockchain. The conditions of the quote are:

1) Full payment info: The node checks the `Payment Info` object from the `Interchain Transaction` object and fills the missing data. For example, the `PaymentInfo` object could contain instructions "I want to pay with USDC on Arbitrum". The *Full Payment Info* would then be populated with the *amount* of USDC that the node will require to execute the transaction.

2) Information on which transactions will be committed. The Klaster nodes are *stateless*, so every back-and-forth request contains the full `Interchain Transaction` info. This enables the system to be easily distributed and highly parallelizable.

3) Maximum block height for every `UserOp`. The node returns the max block height on which it will guarantee the execution of a specific `UserOp`. This gives the dApp/Wallet a guaranteed timeframe for transaction execution.

5. Quote selection

The light node will receive the *Transaction Quotes* from the Full Nodes and then it will use some strategy to select the best transaction quote *or* pass the transaction quotes to the dApp/wallet if they have implemented their own selection strategy. Some potential strategies are:

- **Price optimized:** Select the cheapest transaction quote.

- **Speed optimized:** Select the transaction with the lowest Maximum Block height

6. Connecting to the full node

The dApp/Wallet connects directly to the *Full Node* whose transaction quote it selected.

7. Generating the Interchain Commitment

The dApp/Wallet sends the selected Transaction Quote to the Full Node which generated it and requests the creation of an Interchain Commitment.

The Full Node takes the `Interchain Transaction` object and encodes it as a Merkle Tree of transactions. The root hash of that Merkle Tree is the *hash* of the Interchain Transaction. This is called the iTx hash and takes the format of `0x_{64_character_hash}`. The iTx hash describes the `Interchain Transaction` fully.

An example iTx hash would look like

`0x0157c780e5b884bc442f790f2c9e403417fdf48b0fcd67f5976cfdcff85bdbac`

The Full Node then takes the Merkle Tree root hash and signs it with its private key. Once it has signed the root hash - it has *committed* itself to execute the `Interchain Transaction`.

The `Interchain Transaction` data, together with the signed Merkle Tree root hash - is called an `Interchain Commitment`.

At the end of the process, the full node sends the `Interchain Transaction` back to the dApp/Wallet

8. User Signing the Interchain Commitment

Once the dApp/Wallet has received the signed `Interchain Commitment`, it will sign the Merkle Tree root hash with the private key of the end user. This signature is used by the Klaster `EntryPoint` contracts to validate the `UserOp`s which the user has requested.

After this, two signatures of the Merkle Tree root exist:

- **Signed Node Commitment:** Signed by the node private key, used to slash the node if they don't execute the transactions.

- **Signed User Approval:** Signed by the end-user private key, used to allow the Klaster PEN to commit the transactions to the blockchain(s)

9. Distributing the signatures

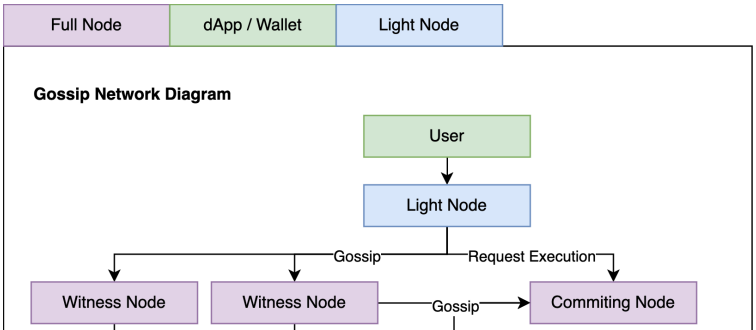
The dApp/Wallet will send both the **Signed Node Commitment** and the **Signed User Approval** to several participants of the Klaster PEN.

- **Committing node:** The user is sending the commitment and user approval to the Full Node which gave them the `Transaction Quote`. When the full node is the one who is committing the transactions - we call them the *Committing Nodes*.

- **Witness nodes:** The guarantee of commitment is provided to the end-user by having the knowledge that if the *Committing Node* fails to execute the transactions, they will be slashed.

However, the Commiting Node could also be exploited by the user. The user could request the `Interchain Commitment`, get the signed commitment from the Full Node and then not sign it. The user would then wait for the `expiry` period of the `Interchain Commitment` to pass, *then* sign the commitment and accuse the Commiting node of maliciously censoring the transaction. This is a variant of the famous *Two Generals Problem*. There is no way for the node to prove that it hasn't received the transaction from the user - and there is no way for the user to prove that they actually *did* send the transaction and the node rejected it or there was a noisy communication channel in the middle.

The Klaster protocol prevents this attack by requiring the end-user to **gossip** the signed transaction to the entire Klaster network. The end-user would send the **Signed Node Commitment** and the **Signed User Approval** to its peers, who would then *gossip* that data between themselves. This would ensure that even if the *Commiting Node* is claiming it hasn't received the signed commitment, the other nodes could tell the protocol that *they* have received the transaction and tried to gossip it to the *Commiting Node*.



Slashing

The security which users have in using the Klaster PEN comes from two guarantees:

1. **Fraudulent transaction protection.** The Klaster PEN is unable to execute any transaction which is not part of the iTx Merkle Tree. These validations are enforced by the Klaster *EntryPoint* contracts - inheriting the security of the destination chains.

   In practice, this means that Klaster users don't need to put any additional trust assumptions into the Klaster PEN itself. As long as the EntryPoint contract works as expected, they are secure.

2. **Censorship protection / liveness guarantees.** The Klaster network needs to provide its users with censorship protection and liveness guarantees in order to be a trusted solution for processing interchain transactions. For censorship protection, the network relies on two key components:

   a. ***Forced transaction execution -*** If the Klaster PEN were to *censor* the user, the user would be able to *force* their own transactions through, by manually calling the `execute` function on their destination Smart Contract Account

   b. ***Crypto-economic guarantees -*** The nodes are incentivized to execute the transactions by taking a *transaction fee* on top of the cost it takes them to execute the transactions. Beyond that, once a node has committed itself to execute the transaction - it *must* execute the entirety of that transaction *or* it will get slashed.

The slashing system in Klaster is split on a per-chain basis.

Slashing the offending node

A Klaster PEN node will be slashed if it fails to execute *any* of the child transactions in the *iTx* bundle. If it fails to execute multiple transactions in the *iTx* bundle, it will be slashed on every chain where it failed to execute the transaction.

Since every transaction in the *iTx* bundle has a `maxBlockHeight` variable, committed to by the node - once this block height has been reached on the chain on which the node failed to execute the transaction - a `Challenge Period` will begin. During the challenge period - the *committing node* can be slashed by the *witness nodes*.

The process is as follows:

1. Detection

The witness nodes hold the non-stale gossiped interchain transactions (together with the signed proofs) in their memory and monitor the chain for block height. Once the block height for the transaction has been reached, the node will query the Klaster EntryPoint contract to see if the transaction has been executed. If the transaction has been executed, the witness node will prune the iTx from its memory. If the transaction *hasn't* been executed - the *witness node* will start the slashing process.

2. Slashing

The witness node will call the `slash` function on the `StakeManager` smart contract with the signed proofs of what the node has committed itself to execute. The `StakeManager` will query the `EntryPoint` contract to see if the hash of the transaction that the node has committed itself to execute has been added to the executed transactions list. If the transaction is in the executed transactions list, the `slash` call will revert. If the transaction is *not* in the executed transactions list, the `slashCommitedStake` variable will be increased by the number of tokens that the *witness node* has staked in the `StakeManager` contract.

If the `slashCommitedStake` has increased above a predefined `threshold` value - the *committing node* will be slashed. The `threshold` value will be a function of the total economic value of the stake on every supported blockchain. For example, if the `threshold` value is set to 25%, the *committing node* will not be slashed, until nodes which hold *at least* 25% of the total stake on that chain - have not called the `slash` function.

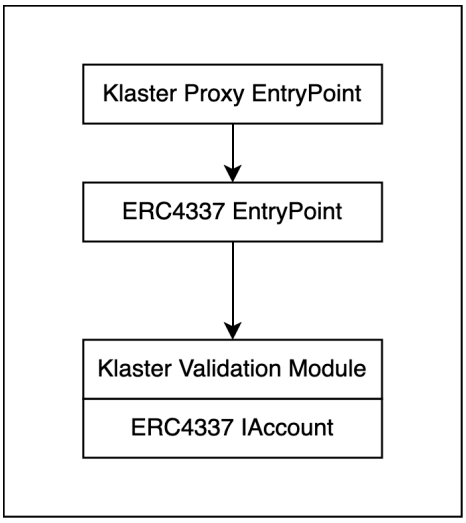
Slashing proof format

In order to call the `slash` function, the proof of action has to be committed in a certain format. The *Witness Node* will call the slash function which has a following form:

```
slash(userOp, signedItxRootHash)
```

The `userOp` is the UserOp object which the node didn't execute and the `signedItxRootHash` is the signed Merkle Tree root hash of the iTx transaction of which the non-executed `userOp` was the leaf.

Smart Contract Stack (ERC4337 + Klaster)



The Klaster PEN is a lightweight execution layer. In order to reduce the security profile of the system, all validation and verification is done on the smart contract levels. For this, Klaster reuses a lot of battle-tested ERC4337 architecture.

Klaster validation happens through a custom ERC-7579 validation module, which verifies the validity of the Merkle Tree Root hash for the given `UserOp`.

This makes the Klaster stack compatible with most ERC-4337 and ERC-7579 infrastructure.
Some very interesting multichain use-cases are unlocked through this modular architecture. To name a few:

- **Passkey authentication:** All passkey authentication systems today are single-chain. Klaster can enable the first, true multichain passkey authentication system.

- **Multi-chain multi-sig:** Klaster can enable developers to build multi-chain multi-sig accounts. It can reuse the Klaster PEN and interchain commitments to use a single signature to update the signers of all smart accounts on all blockchains.

- **Global on-chain spending limits:** Klaster can enforce global spending limits for all accounts on all chains. Each transaction would include a

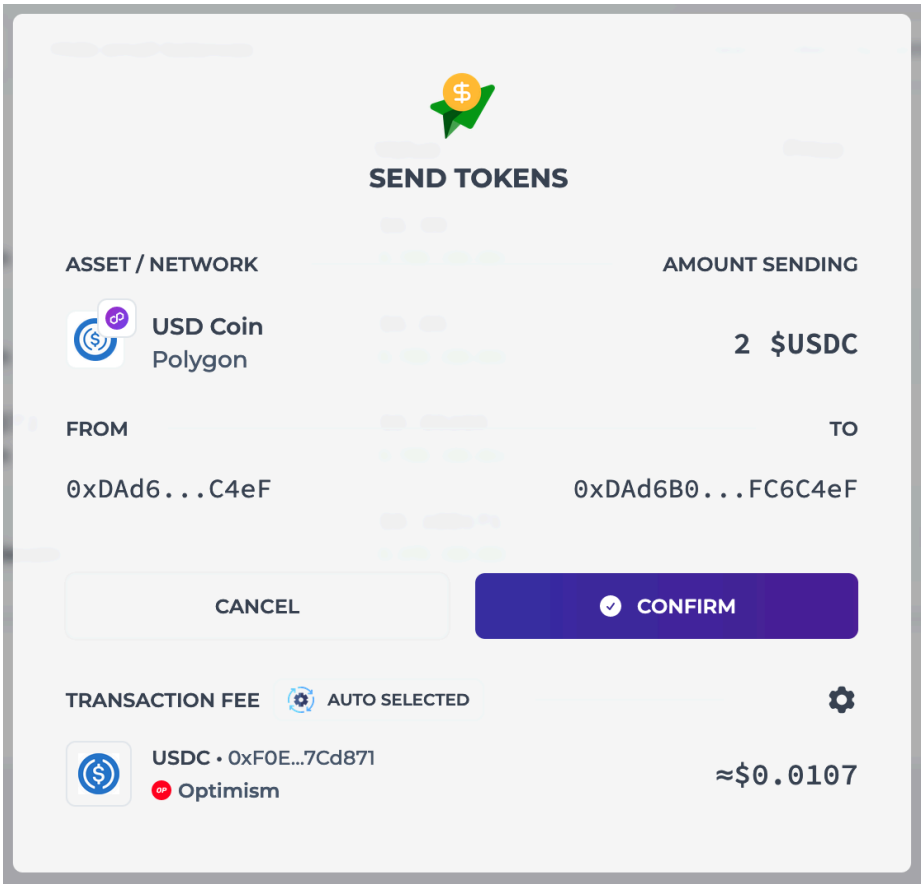
transaction for executing an action and another for globally propagating the spending limit.

Implementation Examples

Gas-abstracted wallets

Interchain Transactions allow wallet developers to create *gas abstracted* wallets. These wallets are able to offer significantly better user flows. As long as a user has *at least one* of the supported payment tokens on *at least one* of the supported blockchains, the user is able to execute their transaction on *any* of the supported blockchains. For users, this means no more “bridging” gas to new blockchains, just to be able to execute transactions.

For wallet developers, this can be a differentiating feature, enabling them to reduce the complexity of usage for new and experienced blockchain users.



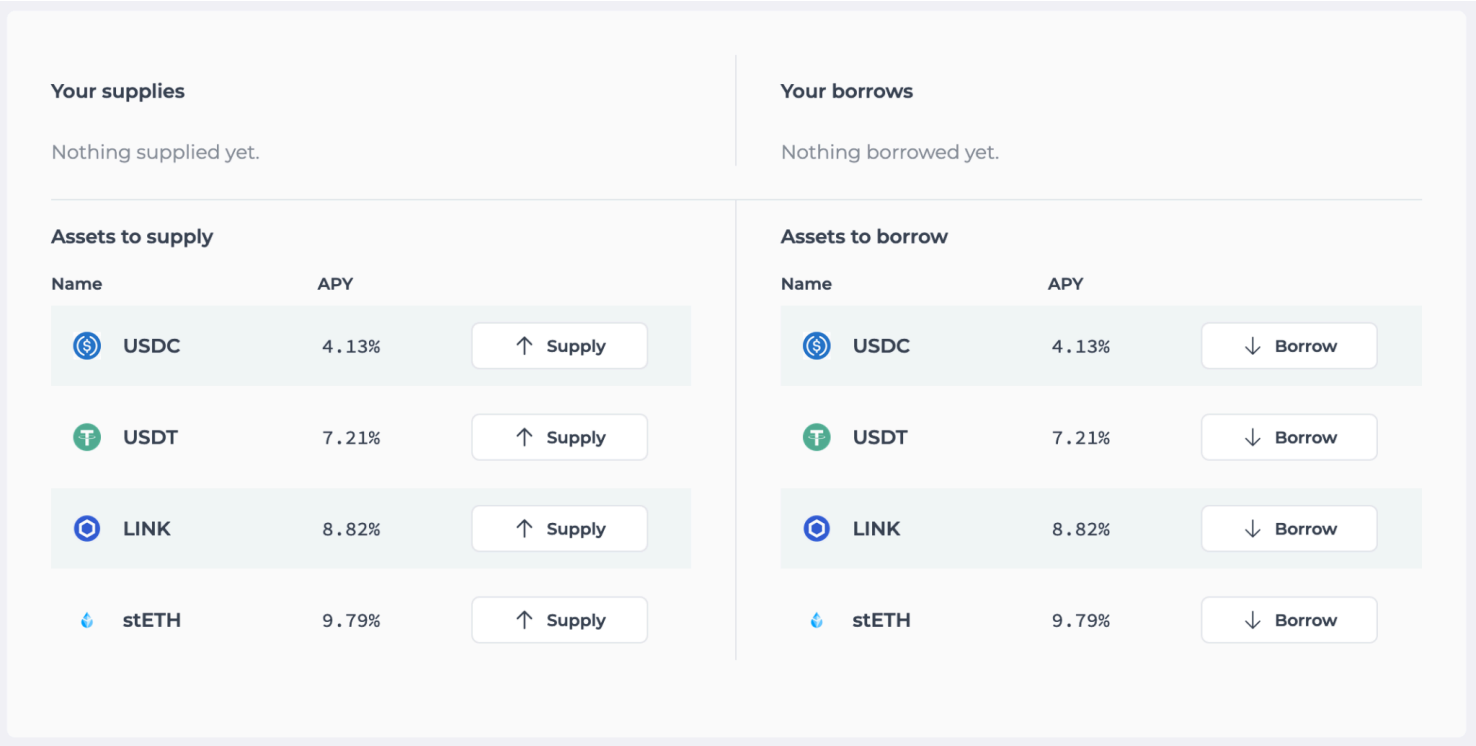
Example of a chain abstracted wallet

Chain-abstracted dApps

A “chain-abstracted” dApp would be a blockchain application which has no mention on which chain it is deployed. It’s able to access user funds across multiple independent blockchains and blend them into a “unified balance”. One of the examples of a chain-abstracted dApp would be the example from the beginning of this paper:

A lending & borrowing aggregator which uses multiple lending & borrowing markets, across multiple blockchains - to find the best yield for supplying and the lowest interest rate for borrowing.

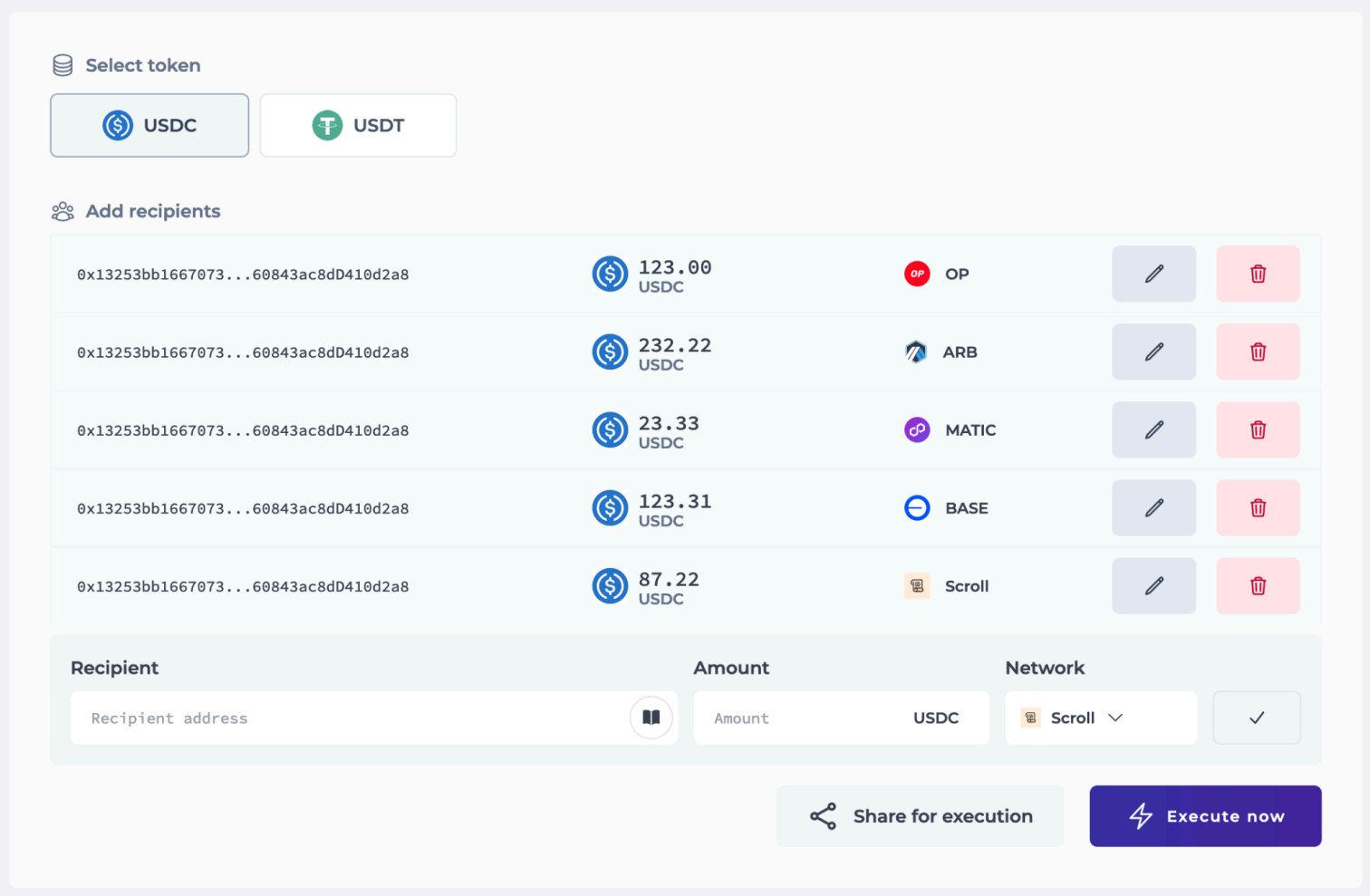
The app would never mention any blockchain or underlying infrastructure. It would simply show the available assets and their respective yields.



Example of chain-abstracted Lending & Borrowing market

Multichain payment flows

Users could use a *single* signature to enable the execution of transactions across multiple blockchains. Imagine a flow of paying out contractors or handling salaries on-chain. The users could be on *any* of the supported chains and the operator could pay them with *no explicit bridging*.



Example of a multichain payment flow frontend.

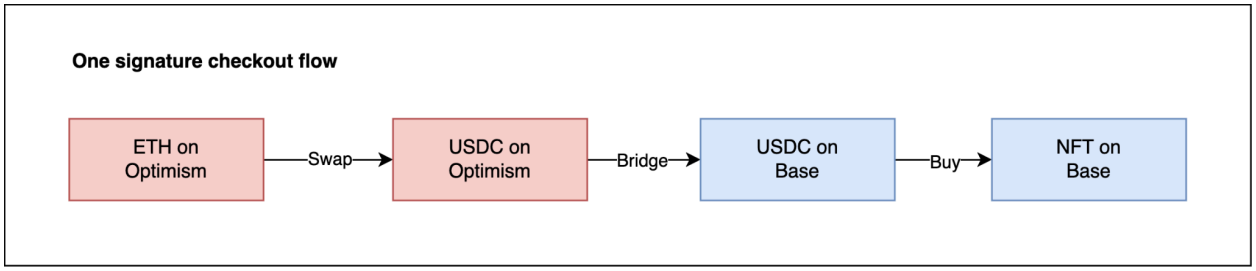
One-signature checkout flows

The developer wishes to implement a checkout flow where a user can buy an NFT on Base chain, with assets on any other chain. The developer wishes to accept only USDC to their account.

With Klaster, a simple interchain transaction can be encoded which:

1. Swaps ETH for USDC on Optimism
2. Bridges USDC from Optimism to Base
3. Buys the NFT on Base

For the end-user, this would be a single-transaction flow.



Flowchart of a one-signature checkout flow

The future of Klaster protocol (State/Storage Proofs, Keystore Rollup support, ...)

This implementation of the Klaster protocol has been optimized to work with solutions available on the market today. However, the architecture of the protocol and smart contracts has been devised in such a way that future upgrades are *expected*.

Storage Proofs

Storage proof support would unlock the ability to have *unified* staking & slashing across all supported blockchains. All node operators would post their stake to Ethereum and this would enable them to process transactions on all blockchains.

The slashing of the stake would happen on Ethereum, with the slashing proof including a state proof of the commitment and non-execution of the desired UserOp on the destination blockchain.

Keystore Rollup Support

Klaster protocol could integrate the Minimal Keystore Rollup specification to enable the synchronization of state across multiple blockchains. This would move the security of the synchronization of the state from crypto-economic security (staking and slashing) to cryptographic proof security (validating the roots of Keystore rollups).