Djwebdapp StarkEx Providemr

Thomas Binetruy

2024-11-12

Abstract

In this paper, the StarkEx provider for DjWebDapp is introduced, a new Django application to simplify integrating with the StarkEx Layer-2 blockchain. It describes the various abstraction implemented to deposit, withdraws, transfer and settle trade-orders.

1 Introduction

DjWebDapp [5] is provider-based Django [1] application allowing to easily integrate multiple blockchains. Until now, DjWebDapp provided integrations for Tezos [3] and EVM-compatible blockchains such as Ethereum [10] and Polygon [4].

Thanks to DjWebDapp's provider based architecture, it is simple to integrate new blockchain protocols while mainaining the same abstractions when developping applications.

There are however two user experience (UX) issues when developping blockchain applications:

1. transaction throughput, generally measured in transactions per seconds (TPS);
2. confirmation blocks.

StarkEx [8] is a Layer-2 (L2) scalability engine for the Ethereum Mainnet allowing to address both of these issues. It essentially works by sending transactions to the StarkEx gateway via an API without any TPS constraints other than network latency and without any confirmation blocks. The gateway then batches these transactions, provides a proof that these transactions are coherent, and submits the batch on a Ethereum smart contract that validates the proof. In the event that the proof is rejected by the Ethereum smart contract (ex: double spending), StarkEx then asks for transaction replacements that needs to be resubmitted.

In order for this scalability engine to work however, the gateway cannot be public and only a single entity can submit transactions to it. As such, StarkEx is considered a private blockchain backed-up by a public blockchain (Ethereum).

Some application make this trade-off; such as Sorare [6], a fantasy sports game and DyDx [2], a perpetual trading platform. Indeed, both of these platforms made the trade-off to relax decentralibility in favor of performance and UX.

In this paper, we present the StarkEx DjWebDapp provider, allowing to integrate Django web applications with the StrakEx scalability engine.

2 StarkEx mechanics

Let us first introduce the fundamental operations exposed by StarkEx:

1. **Mint**: allows minting an asset (a fungible or non-fungible token) on StarkEx (L2);
2. **Deposit**: allows depositing an asset from Ethereum (L1) to StarkEx;
3. **Withdraw**: allows withdrawing an asset from StarkEx to the L1;
4. **Transfer**: allows transfering an asset from an L2 account to another L2 account;
5. **Asset Trade Order**: allows for a user to define an order for a quantity q_g of *give* assets a_g in exchange for a quantity q_a of *ask* asset a_a.
6. **Multi asset trade order**: allows for a user to define an order for quantities $(q_{g_1}, ..., q_{g_n})$ of n *give* assets $(a_{g_1}, ..., a_{g_n})$ in exchange for quantities $(q_{a_1}, ..., q_{a_m})$ of m *ask* assets $(a_{a_1}, ..., a_{a_m})$.

Some of these operations interact only with the L2 (*mints*, *transfers* and *trade orders*), whereas others also interact with smart contracts on the L1 (*deposits* and *withdraws*).

Creating transaction on StarkEx target a **gateway**, **G**, which processes these transactions.These are then batched together and submitted to a **validating contract**, **V**, deployed on Ethereum. Finally, *deposits* and *withdraws* interact with a **bridge contract B** deployed on Ethereum. Note that in practice, **G** and **V** are the same contract.

Since StarkEx is a private blockchain, in a typical architecture, only the application backend is authorized to submit transaction to **G**.

2.1 Asset

An asset on StarkEx is represented by an **asset ID**, which is an integer calculated based on parameters including the asset type (mintable, non-mintable, fungible, non-fungible). All StarkEx operations then target this asset using the asset ID.

In order to allow for *deposits* and *withdraws* to and from the L2, the StarkEx asset needs to be paired with a smart contract on the L1. Let a a represent the asset on the L1 and $\bar{a}$ represent the bridged asset on the L2. An asset a's ID will be denoted by a_{ID} and a quantity of asset a by n_a.

2.2 Accounts

A StarkEx account consist of a private key and its associated stark key. Athough the cryptographic algorithm is different than that of Ethereum, the same concept is used. In the rest of this document, an Ethereum account will be denoted by α and a StarkEx account by $\bar{\alpha}$. The public key belonging to α or $\bar{\alpha}$ will be denoted by $k_{\text{pub},\alpha}$ and $k_{\text{pub},\bar{\alpha}}$ respectively. Indices will be used as needed.

2.3 Vault

A StarkEx vault consists of a numbered slot where an asset can be deposited into. A vault can only store assets of the same type belonging to the same account. As an example if an account $\bar{\alpha}_1$ has a quantity n of assets $\bar{a}$, they can all be deposited into the vault $v_{\bar{\alpha}_1,\bar{a}_1}$ if either one of the following conditions is met:

1. the vault is empty;
2. $\bar{\alpha}_1$ has already deposited $\bar{a}$ in the vault.

The vault is owned by the account that first deposits assets in it and freed when all assets are removed from it. The account $\bar{\alpha}_1$ could also decide to deposit assets $\bar{a}$ into multiple vaults (ex: $v_{\bar{\alpha}_1,\bar{a}}$ and $v'_{\bar{\alpha}_1,\bar{a}}$). However, another account $\bar{\alpha}_2$ cannot deposit into $v_{\bar{\alpha}_1,\bar{a}_1}$ until $\bar{\alpha}_1$ has removed all assets from the vault (by transfering them to another vault).

2.4 Minting

Minting an asset $\bar{a}$ on the L2 to account $\bar{\alpha}$ is done by making an API call from the application backend to the gateway with parameters:

1. destination vault, $v_{\bar{\alpha},\bar{a}}$;
2. asset ID, $\bar{a}_{\text{ID}}$;
3. asset quantity, $n_{\bar{a}}$;
4. account public key, $k_{\text{pub},\bar{\alpha}}$;
5. type `MintRequest`.

Where $v_{\bar{\alpha},\bar{a}}$ is either empty or stores a quantity of $\bar{a}$ already owned by $\bar{\alpha}$.

2.5 Transfer

2.6 Deposit

Depositing an asset $\bar{a}$ consists in bridging its associated asset a from the L1 to the L2 through the bridge smart contract **B**. As such, an L1 transaction need to be made followed by an API call to the gateway. As an example, if a user with an Ethereum account α and a StarkEx account $\bar{\alpha}$ bridges an asset a, the following steps need to be taken:

1. a first Ethereum transaction from α to a's smart contract to approve a `transferFrom` call of n_a from $k_{\text{pub},\alpha}$ to **B**.
2. Ethereum transaction from α to **B**'s deposit method with parameters:
   (a) quantity, n_a;
   (b) quantization factor, q;
   (c) destination vault, $v_{\bar{\alpha},\bar{a}}$;
   (d) asset ID, $\bar{a}_{\text{ID}}$.
3. DjWebDapp indexes **B** and wait for confirmation blocks.
4. A StarkEx API call from the application backend to the gateway with parameters retrieved from the indexed deposit transaction on **B**:
   (a) destination stark key, $k_{\text{pub},\bar{\alpha}}$;
   (b) quantity, $n_{\bar{a}}$;
   (c) destination vault, $v_{\bar{\alpha},\bar{a}}$;
   (d) asset ID, $\bar{a}_{\text{ID}}$.
   (e) type `DepositRequest`.

After step 2, n_a will have been transfered from α to **B**, meaning that α's a balance will have been decreased by n_a on Ethereum. Due to the decentralized nature of Ethereum, it is necessary to wait for a sufficient amount of confirmation blocks before submitting the deposit request to **G**. Indeed, since the StarkEx batch b in which the deposit transaction has been included to will be validated by **V** on the L1 up to a few hours in the future, a reorg could lead to **B**'s transaction not being included on the L1. This in turns means that **V** would not validate b and **G** would then request an **alternative transaction** for the deposit call.

The quantization factor q is used to bypass the constraint enforced by StarkEx relative to the maximum quantity of assets that can be transfered on the L2. Indeed, StrakEx encodes token quantities in 64 bits against 256 for Ethereum [7]. q can be used to quantize token quantities and bypass this limitation, which is particularly useful for Ethereum tokens that have a large amount of decimals.

2.7 Withdraw

Withdrawing an asset a consists in bridging its associated asset $\bar{a}$ from the L2 to the L1 through the bridge smart contract **B** from a sender account $\bar{\alpha}$ to a receiver account α. It consists in making:

1. a StarkEx transfer call with the following parameters:
   (a) the asset to transfer, $\bar{a}$;
   (b) the quantity of assets to transfer, $n_{\bar{a}}$;
   (c) the sender's StarkEx public key, $k_{\text{pub},\bar{\alpha}}$;
   (d) the receiver's Ethereum public key, $k_{\text{pub},\alpha}$;
   (e) the sender's vault ID, $v_{\bar{\alpha},\bar{a}}$;
   (f) the receiver's vault ID, $v_{\alpha,\bar{a}}$;
   (g) an expiration timestamp;
   (h) $\bar{\alpha}$'s signature;
   (i) type `TransferRequest`.
2. a StarkEx withdraw request by $\bar{\alpha}$ with parameters:
   (a) receiver's vault ID, $v_{\alpha,\bar{a}}$;
   (b) the token ID, $\bar{a}_{\text{ID}}$;
   (c) the token amount, $n_{\bar{a}}$;
   (d) the receiver account, α;
   (e) type, `WithdrawalRequest`.
3. an Ethereum transaction call by anyone to **B**'s withdraw method to release the funds to α.

2.8 Multi Asset Trade Order

Multi asset trade orders allow for exchanging multiple assets **atomically** on the L2 between multiple parties. It consists of the following parameters:

1. each account α_k specifies give and receive orders:
   (a) give orders, a list of n assets:
      i. an asset ID, $\bar{a}_{i_{\text{ID}}}$ where $i \in [\![0, n]\!]$;
      ii. an amount of asset $\bar{a}_i$, $n_{\bar{a}_i}$;
      iii. the sender's public key, $k_{\text{pub},\bar{\alpha}_k}$;
      iv. the sender's vault id, $v_{\bar{\alpha}_k,\bar{a}_i}$;
   (b) ask orders, a list of m assets:
      i. an asset ID, $\bar{a}_{j_{\text{ID}}}$ where $j \in [\![0, m]\!]$;
      ii. an amount of asset, $\bar{a}_j$, $n_{\bar{a}_j}$;
      iii. the sender's public key, $k_{\text{pub},\bar{\alpha}_k}$;
      iv. the sender's vault id, $v_{\bar{\alpha}_k,\bar{a}_j}$;
   (c) the account's public key, $k_{\text{pub},\bar{\alpha}_k}$;
   (d) $\bar{\alpha}_k$'s signature;
   (e) an expiration timestamp;
   (f) a transaction nonce.
2. type: `MultiAssetTradeRequest`.

The validation smart contract **V** will then accept the multiasset trade call if and only if all orders are satisfied, that is, if all parties receive what they have asked for. It is the application backend's job to ensure that all orders are satisfied before submitting the multi asset trade order to **G**.

2.9 Transaction ordering

Transactions submitted to the StarkEx gateway **must** include a continuously incremented transaction ID chosen by the application backend.

2.10 Alternative transactions

When the validating smart contract **V** rejects a specific transaction, the StarkEx backend will call a webhook on the application backend to request an alternative transaction. Examples of such invalidated transactions include:

1. Making a deposit on **G** of a greater amount than the amount of deposited token on **B**;
2. An account $\bar{\alpha}_1$ transfering tokens from a empty vault;
3. A multi asset trade order than cannot be settled;
4. ...

Due to the decoupling of transaction submission from the application backend to the **G** and from **G** to **V**, many transactions could have been submitted to **G** that depend on an invalid transaction. In such a case, it can be very difficult to recover from such a state divergence between the application backend and StarkEx. Hence, it is important that the application backend minimizes alternative transaction opportunities.

3 StarkEx DjWebDapp provider

The DjWebDapp StarkEx provider provides tools to:

- index and normalize **V** deposits;
- create and track vault owner and balances;
- create StarkEx accounts;
- number and submit transaction to **G**;
- index transactions on **G**;
- receives and replace alternative transaction requests.

4 High-level Architecture

Figure 1 provider a high level overview of how the different components of a typical StarkEx-based application interact with each other. Note that the interactions between the application indexer spooler and Ethereum contracts are implemented using DjWebDapp's Ethereum provider [5].

5 StarkEx provider implementation

The StarkEx provider is implemented as a subclass of `djwebdapp.providers.Provider`.

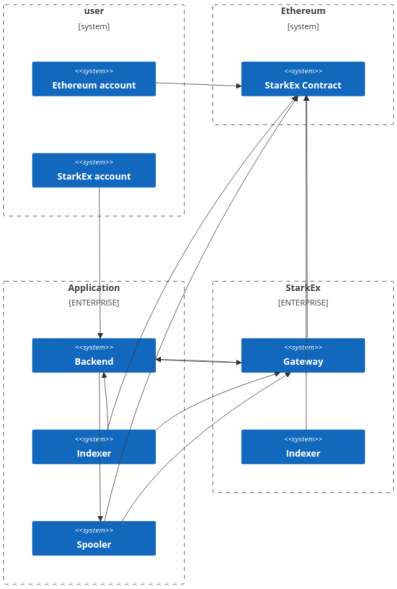

Figure 1: Djwebdapp ↔ StarkEx interactions

5.1 Accounts

A StarkEx account consists of a `djwebdapp.providers.Account` class initialized by the StarkEx provider by overriding the following methods:

1. `generate_secret_key` which returns a pair $(k_{\text{pub},\bar{\alpha}}, \text{priv},\bar{\alpha})$ using StarkEx's key algorithm.

5.2 Spooler

Deploying a transaction is done by overriding the `deploy` method which retrives the transaction arguments and submits an API call to **G**.

As with other blockchain providers, only transactions with state `deploy` are processed by the spooler. It first assigns it a transaction ID by retriving the last deployed transaction ID and incrementing it.

Note that if there are any unprocessed alternative transactions, the spooler will raise an exception. This allows alerting systems such as Sentry to notify application administrators that an alternative transaction was requested by **G**.

5.3 Indexer

The indexer is implemented by overriding the `index_level` provider method to retreive the last accepted batch from **G**, iterate over its transactions, and update the transaction states using the transaction IDs to `confirm`.

An Ethereum provider based indexer is then used to index **V** and update all transactions in the batch to `done` once a configurable amount of confirmation blocks have been indexed above the batch.

6 Stark transaction models

6.1 Vaults

Vaults are a core component of the StarkEx state and must be modeled correctly in order for transactions to be validated by **V**. Should the application vault accounting diverge from that of the StarkEx state, then alternative transactions would be requested by **G** leading to an application hault.

Vaults are implemented as a Django model in `djwebdapp_stark.models.Vault`. It consists in the following columns:

1. `vault_id`: a positive integer;
2. `blockchain`: a foreign key to a DjWebDapp `Blockchain`;
3. `owner`: a foreign key to a DjWebDapp `Account`, the vault owner $\bar{\alpha}$;
4. `token`: a foreign key to a DjWebDapp `StarkToken`, the vault asset $\bar{a}$;
5. `quantity`: a decimal field, the quantity of assets in the vault $n_{\bar{a}}$;

A `Vault` model implements an `update_properties` method to recalculate the balances based on all `StarkTransaction`'s stored in the database.

The vault ID is assigned at the vault creation by incrementing from the previous vault numbering. It does so by locking the database table using `SELECT FOR UPDATE` in order to allow for parallel request processing in the context of multi-processing servers such as uWSGI [9].

6.2 StarkToken

A stark token is modeled by a set of two classes:

1. `StarkContract`, a Django model allowing to represent a (non-)fungible token on StarkEx contract with the following rows:
   (a) `blockchain`: a foreign key to a DjWebDapp `Blockchain`;
   (b) `l1_contract`: the associated L1 contract, a;
   (c) `contract_type`: a string encoding the contract type (fungible, non fungible, etc);
   (d) `quantum`: the stark asset's quantum `q`;
   (e) `asset_info`: a string encoding the asset info, needed to calculate asset IDs;
   (f) `asset_type`: a string encoding the asset type, needed to calculate asset IDs;
2. `StarkToken`, a Django model representing individual assets stored by a `StarkContract`. This decoupling of `StarkContract` and `StarkToken` is particularly useful to model non-fungible tokens where each assets is independent, but can be withdrawn on the same L1 contract. Its rows include:

(a) `contract`: a foreign key to a `StarkContract`;

(b) `token_id`: a positive integer encoding the token ID;

(c) `asset_id`: a positive integer encoding the asset ID, needed by stark transactions;

The `StarkContract` also implements a `get_asset_id` method parametered by a `StarkToken.token_id` to automatically calculate a `StarkToken.asset_id` according to the stark contract parameters as specified by the StarkEx documentation.

6.3 StarkTransation

The `StarkTransaction` model is the base model from which all StarkEx transactions are modelled after. It inherits from `djwebdapp.models.Transaction` and includes the following fields:

1. `tx_id`: an positive integer field encoding the transaction ID necessary to order transactions on StarkEx;
2. `contract`: a foreign key to a `StarkContract` instance.

The `StarkTransaction` model exposes Python properties on fields meant to be implemented by subclasses to discriminate deposits, mints, withdraws, etc. These properties are implemented at the `StarkTransaction` level in order for the spooler to be able to cast a `StarkTransaction` to its subclass efficiently (without any joins) and recover the proper transaction arguments needed when submitting the transaction to **G**.

6.4 MovementCall

A movement call is the base class used to encode all StarkEx transaction except for multi-asset trade calls. A `MovementCall` is a model subclass of `StarkTransaction`. It includes all the fields needed by subclasses. Indeed, in order to reduce the number of database joins, it was decided to implement subclasses as Django **proxy** models. As such, the union of all fields neede by subclasses should be defined by this model. They include:

1. `l1_event`: a foreign key to a `djwebdapp_ethereum.models.EthereumEvent`. This field is neede by subclasses implementing L1 $\leftrightarrow$ L2 interactions where the L1 event is to be stored for accounting.
2. `vault_sender`: a foreign key to `Vault` used by some subclasses.
3. `vault_recipient`: a foreign key to `Vault` used by some subclasses.
4. `sender`: a foreign key to `Account` used by some subclasses, from which `vault_sender` is derived.
5. `recipient`: a foreign key to `Account` used by some subclasses, from which `vault_recipient` is derived.
6. `token`: a foreign key to `StarkToken`, the asset being targetted by the transaction.
7. `quantity`: a positive integer encoding the quantity of assets being transfered.
8. `transfer_nonce`: a number used to prevent double-spending inside a batch.
9. `expiration_timestamp`: an expiration timestamp for the transaction.
10. `signature_r_transfer` and `signature_s_transfer`: the transaction sender signature.

Each proxy model subclassing `StarkTransaction` can implements a Django **manager** filtering on a subset of fields to restrict the resulting objects to those part of the proxy model only.

6.4.1 DepositCall

A proxy subclass of `MovementCall` implementing a `get_args` method returning the JSON expected by **G** in a deposit request. Its `save` method recalculates automatically the recipient's vault balance.

The `DepositCallManager` implements a filter on the `MovementCall` table such that only the set of deposit calls, $\mathcal{D}$ is returned:

$$\mathcal{D} = \{m \in \text{MovementCall} \mid m.\text{sender} = \text{None} \wedge m.\text{l1\_event} \neq \text{None}\}$$

6.4.2 MintCall

A proxy subclass of `MovementCall` implementing a `get_args` method returning the JSON expected by **G** in a mint request. Its `save` method recalculates automatically the recipient's vault balance. Its `deploy` method automatically creates a recipient vault if one does not exist for this asset for this user.

The `MintCallManager` implements a filter on the `MovementCall` table such that only the set of deposit calls, $\mathcal{M}$ is returned:

$$\mathcal{M} = \{m \in \text{MovementCall} \mid m.\text{sender} = \text{None} \wedge m.\text{l1\_event} = \text{None}\}$$

6.4.3 TransferCall

A proxy subclass of `MovementCall` implementing a `get_args` method returning the JSON expected by **G** in a transfer request. Its `save` method recalculates automatically the sender and recipient's vault balances. It implements a `get_sender_signature` which returns the datastructure needed to be signed by the sender. The `save` method automatically retrieves or creates

the sender and recipient's vaults based on the database state.

The `TransferCallManager` implements a filter on the `MovementCall` table such that only the set of deposit calls, $\mathcal{T}$ is returned:

$$\mathcal{T} = \{m \in \text{MovementCall} \mid m.\text{withdraw\_account} = \text{None} \wedge m.\text{sender} \neq \text{None}\}$$

6.4.4 WithdrawCall

A proxy subclass of `MovementCall` implementing a `get_args` method returning the JSON expected by **G** in a transfer request. Its `save` method recalculates automatically the recipient's vault balance.

The `WithdrawCallManager` implements a filter on the `MovementCall` table such that only the set of deposit calls, $\mathcal{W}$ is returned:

$$\mathcal{W} = \{m \in \text{MovementCall} \mid m.\text{vault\_sender} = \text{None} \wedge m.\text{withdraw\_account} \neq \text{None}\}$$

6.4.5 AlternativeTransactionWebhook

When **V** does not validate a transaction, **G** sends a webhook to a Django view which creates an `AlternativeTransactionWebhook` object including the following rows:

1. `reason_code`: a string storing the code;
2. `reason_msg`: a string storing the validation error message;
3. `tx_id`: the rejected transaction ID;
4. `tx`: the rejected transaction JSON;
5. `alternate_txs`: the transaction JSON to replace the rejected transaction with;
6. `is_processed`: a boolean field storing whether the transaction was processed or not.

The DjWebDapp StarkEx application also implements a permissioned route `/stark` which **G** can submit altrnative transaction requests to. When the application administrator sets `AlternativeTransactionWebhook.alternate_txs`, the next call by **G** to `/stark` will respond `alternate_txs`. If the alternative transactions are accepted by **V**, then the `is_processed` flag will be set to `True`.

7 Conclusion

In this document, the DjWebDapp StarkEx integration was presented. It was shown that this integration allows to easily create StarkEx accounts, handles vault accounting automatically without the application developper needing to manually keep track of ownership and balances, and uses all the usual DjWebDapp patterns.

It was also shown that the DjWebDapp Ethereum provider previously implemented allowed for indexing deposit calls and the spooler allowed for withdraws.

References

[1] Django Software Foundation. *Django*. Version 4.1. June 20, 2023. URL: `https://djangoproject.com`.

[2] DyDx. *DyDx*. Version 1.0. Oct. 8, 2024. URL: `https://dydx.exchange/`.

[3] L.M Goodman. "Tezos — a self-amending crypto-ledger White paper". In: ().

[4] Polygon. *Polygon*. Version 1.0. Oct. 8, 2024. URL: `https://polygon.technology/`.

[5] Pyratzlabs. *DjWebDapp*. Version 0.5. Oct. 8, 2024. URL: `https://djwebdapp.readthedocs.io/`.

[6] Sorare. *Sorare*. Version 1.0. Oct. 8, 2024. URL: `https://sorare.com/`.

[7] StarkEx. "Asset quantities in StarkEx". In: ().

[8] StrakWare. *StarkEx*. Version 5.0. Oct. 8, 2024. URL: `https://starkware.co/starkex/`.

[9] uWSGI. *uWSGI*. Version 2.0. Oct. 8, 2024. URL: `https://uwsgi-docs.readthedocs.io/`.

[10] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: ().