Democrit on EVM Chains

Daniel Kraft
March 2023

Abstract

Democrit is a software and protocol for atomic trading between CHI on Xaya Core and assets inside of game states. On Xaya Core, this is based on multisignature transactions similar to CoinJoins on Bitcoin, which do atomic payments for name updates. In their main implementation, the drawback of Democrit trades is that they need interactivity between the trading partners.
On EVM-chains like Polygon, the base chain provides much more flexibility. In particular, names can be owned by smart contracts with verifiable and trusted code. In this document, we describe how this more powerful setting can be used to implement atomic and trustless trades between game assets and on-chain assets (like WCHI or other ERC-20 tokens). The protocol we describe allows market and limit orders for both buying and selling, with very little (or no) interactivity required.
The trading protocol is discussed in terms of trading Soccerverse coins (SMC) for WCHI on the Polygon network, but it can be applied in the same way to other game assets, tokens and smart-contract chains as well.

Background

If Xaya is running on a base chain that supports smart contracts (in particular, is EVM-based), it is possible to give a contract ownership of a name. This implies that any moves sent through that name are predictable and restricted to what is coded into the contract, so that the associated Xaya name and its moves (perhaps with special power in a game) is not a trusted third-party but can be used to build a fully decentralised and trustless protocol.
Unlike full roll-ups, the Xaya GSP model has only a unidirectional flow of information; there is no direct way to prove some game state back onto the base chain (in a trustless and decentralised manner). This radically simplifies Xaya and avoids issues and difficulties seen in full roll-ups, but makes building bidirectional bridges hard. It is, however, possible to implement direct, peer-to-peer trading that is atomic and trustless. The main ingredient for this to work is a requirement that the buyer of some asset has to verify the game state themselves, to ensure that a trade is valid (i.e. the seller actually has the asset they are buying). Each user does this for themselves, so there is no trusted or centralised oracle; but if they fail to properly verify the game state, then they risk losing funds. There is also no direct way to build a bridge or trading pool that is not inherently peer-to-peer, e.g. where a smart contract is one of the parties (like in an AMM).
For Soccerverse, we would like to enable peer-to-peer trading of SMC inside the game state to WCHI or other ERC-20 tokens on the Polygon chain, so that users can cash out funds earned

inside the game if they wish. This should be trustless and without a centralised party (e.g. us) facilitating the trades.

Overview

For the trading protocol, we define a new primitive that is implemented in the GSP: *Trading vaults*. Vaults can be created by "locking" a certain initial amount of SMC inside the game state; they will end up inside a new vault with some ID. The vault is controlled by another name, and only that name is able to then transfer SMC from within the vault to users inside the game state.
We use this feature now by making the vaults owned by a smart contract (which creates and owns a name, perhaps its address in hex as ASCII, in its constructor). That way, once a user has locked coins into a vault owned by the trading contract, the coins only move in predictable ways based on the contract's code. In particular, this allows us to keep information about how many coins are still inside the vault in sync between the contract's on-chain state and the vault's state inside the game.
With this, the contract can implement an order-book exchange:

- Users can create a (limit sell) order, which locks coins into a vault.
- They can cancel orders, which moves all remaining coins back to themselves.
- If another user takes an open order, the contract moves the payment tokens on-chain from buyer to seller, and then sends a move that sends coins from the vault to the buyer. This is guaranteed to be valid, since the amount of coins inside the vault is known in the contract state.

However, there's still one important caveat: **When a vault is initially funded, it is not known on-chain if the user had enough coins, and thus if the vault was successfully created in the game state or not.** This remains the only bit of information that each user (especially buyers) has to verify for themselves. So before accepting an offer tied to a particular vault ID, they need to query the game state to check if the vault with that ID exists. But if it does, then they know that the contract's state matches the game state, so nothing else can go wrong!
For this check, we will need to make sure the "sync" between GSP and the smart contract is protected against reorgs. So in other words, when the user queries their GSP to learn that a vault exists, and then goes on to call the trading contract to execute a trade, the contract must in ensure that the GSP state the user queried for is in the blockchain ancestry of the current block, and is not a branch that got reorged away.

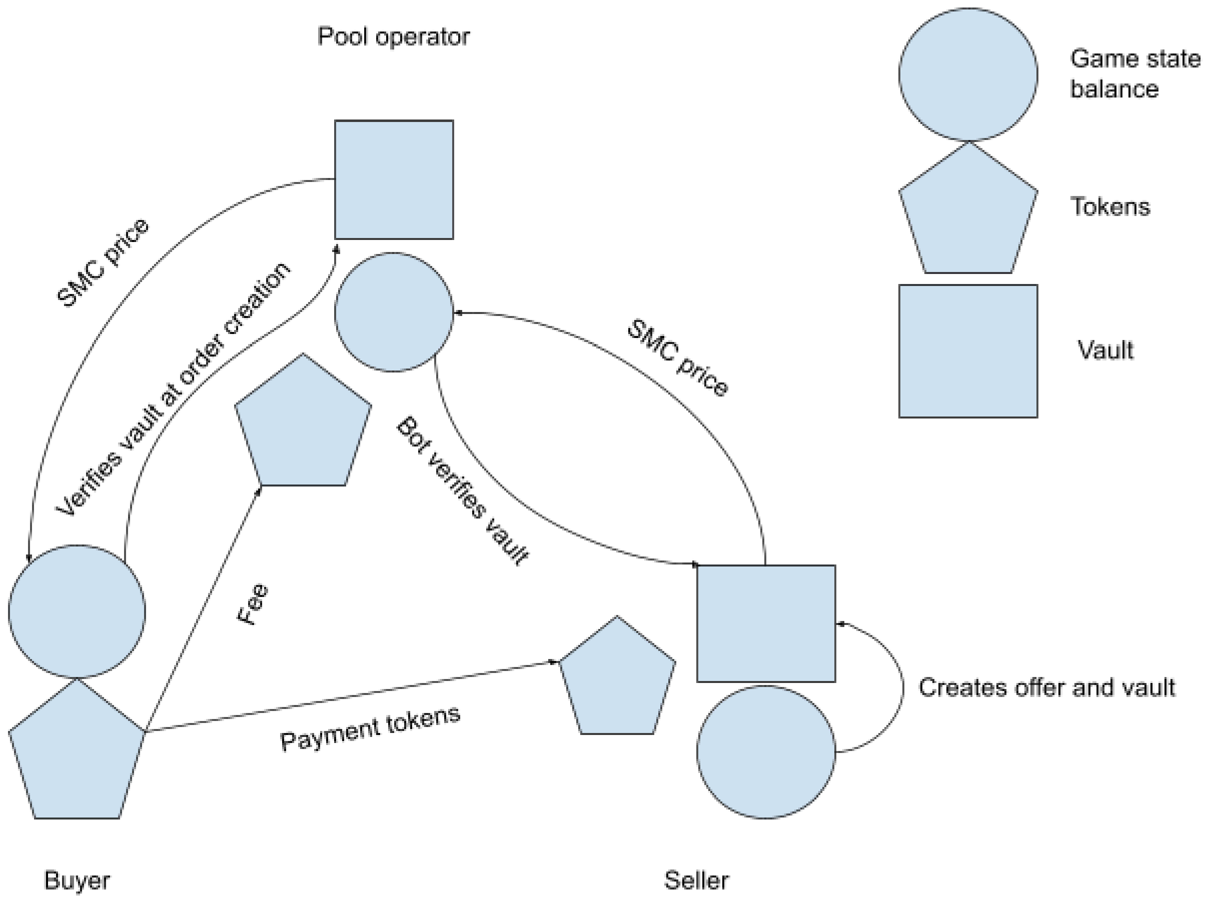
Limit Buy Orders

The basic protocol described above works well to implement limit sell orders: Those can sit in the order book until a buyer takes them, and at that point in time, the buyer can for themselves verify the orders/vaults in the game state. So this works without any further interaction between the two parties. For limit buy orders, it is not that simple, because when creating the order, the buyer does not know yet what vault the seller that ultimately accepts their offer will be using, so they cannot verify its validity. Thus it will be necessary for the buyer to be online when the seller accepts their order, so they can then verify it before proceeding; the trade becomes *interactive*.

It is, however, possible to loosen this requirement with a trick: We introduce *trading pools*. They are large vaults created by dedicated "market makers", which could be us or community members. It is important to note that while they "lock" SMC into their vault, they can withdraw them at any time; also, they do not actually market make in the traditional sense (where they buy and sell SMC for another token), but they just facilitate transfers of SMC between trading partners similar to a Lightning hub. They might take a fee for this service, which would be almost risk-free yield on SMC. Operators of trading pools are not trusted by any of the parties either, so the protocol is still fully trustless (and can be decentralised as anyone can act as pool operator).

If someone now wants to create a limit buy order, they select one or more of the trading pools, and verify that *their* vaults are valid. With their order, they specify which trading pools they accept as hub, i.e. which they have verified to be correct. A potential seller who wants to accept such a limit order then creates a vault, and gets the vault checked by one of the operators of the selected trading pools (which can be a bot running on a server). The trading contract then atomically transfers the payment tokens from buyer to seller, an optional fee in tokens from buyer to pool, transfers SMC from seller to the trading pool's operator, and from the trade pool to the buyer. This way, SMC transfers happen only from vaults to users which have verified the respective vaults explicitly. The interactivity requirement is moved from the buyer (which is an ordinary user running Soccerverse in their browser) to the operator of the trading pool, which can be a dedicated server.

Through trading, the vaults of the pools will be slowly depleted over time. When they run out, all orders referencing them will become invalid (impossible to accept). The pool operator will then have to create a new pool, and prospective buyers will have to re-verify the new vault and create fresh orders. But the vaults can be large, so it should be possible to do all this without too much impact on trading activity in practice.



Implementation

Let us now describe the various bits and pieces that make up a full implementation in detail.

Protecting against Reorgs

In the trade protocol outlined, users will have to check a GSP for the existence of a vault, and then, if it exists, proceed with a trade. In case the GSP state and the on-chain state when the transaction gets included diverge, e.g. through a reorg, this could in theory lead to loss of funds, because the vault might not actually exist on the version of reality that leads to the transaction finally confirmed in the trade contract (i.e. if the GSP check was done on a block that was later reorg'ed, or the GSP was stuck on some branch or something like that).
When querying the GSP for a state, the GSP will return the block hash at which the answer is current and correct. So ideally users would then pass on that block to the trading contract, and the trading contract would verify that this block exists in the ancestry of the current block, reverting the transaction if not. Unfortunately, the `blockhash()` function, which would be used

for this, is restricted to the last few blocks and cannot be used to check an arbitrarily-old ancestor.
Thus we propose an alternative protocol for protecting GSP checks against reorgs, based on *checkpointing*. The idea is to explicitly keep a list of known ancestor blocks in the trading contract, with a `mapping(bytes32 => bool)`. Unfortunately, we still can't just register all blocks at which vaults are created, because during creation, the *current* block hash is not known either. We can, however, do a later transaction that records the then-previous block as a "checkpoint", i.e. a block known to be in the ancestry of the current chain tip. When this is done, the vault controller account sends a special move, providing the checkpointed block hash. The GSP will then record the checkpoint hash on all existing vaults by that controller that have been created earlier than the current block and do not yet have a checkpoint.
This ensures that:

- The checkpoint recorded for a vault in the GSP is a block at which the vault was existing already (either just created or for some time already).
- If a record exists for a checkpoint hash in the trading contract, that checkpoint is an ancestor block of the current block in which the execution is done.
- Assuming GSP and contract execution are on the same branch of the blockchain, then all checkpoints recorded for vaults in the GSP will be recorded on the contract.

So with this, it is possible to safely verify GSP-existence of vaults and assert them in the contract, without risks for reorgs:

- Query the GSP for the vault in question.
- If there is not yet a checkpoint attached to it, stop for now, and perhaps manually trigger checkpointing in the contract, or wait.
- If there is a checkpoint, pass that to the contract call.
- The contract receives a checkpoint hash, and verifies that it is recorded in its mapping of known ancestor blocks.

The main drawback of this method is that checkpointing needs to be done on vaults as a follow-up step. We can explicitly trigger checkpointing whenever a contract method is called and there are new vaults, but this will then still leave the very newest vaults uncheckpointed for the time being; users will have to manually checkpoint them in case they need the vaults (e.g. for accepting a freshly-created sell order).

Vaults in the GSP

On the GSP side, we need to implement support for trading vaults as a new primitive, with an associated set of moves. In the game state, a vault stores the following bits of information:

- The vault's ID, which is an integer,
- the vault's controller, which is a Xaya account name,
- the vault's founder, which is another Xaya account name,
- the balance of SMC inside the vault,
- the blockheight at which it was created, and
- the checkpoint hash, if one is known already.

Only the balance is mutable, and the checkpoint hash can be set once if it is not yet set. The ID, controller and founder of a vault are fixed upon creation. The ID together with the controller

is the "primary key" identifying the vault; IDs need to be unique per controller, but not globally. The blockheight is recorded so that checkpointing can be done on vaults not created in the current block, and it may be useful in general, e.g. for frontends.
The controller is the name which sent the vault-creation command, and which is able to move the funds inside the vault. In our trading protocol, it will be the smart contract. The founder is the user who deposited the initial SMC upon funding (or is able to while the vault is unfunded). It can be used for instance by frontends to show the balance as part of a user's balance.
Coins inside a vault are counted towards the founding user's *total* and *reserved* balances (this also implies that the *available* balance, which is what matters for trading logic, is not affected).
New RPC methods allow to query for all vaults funded by a given user (so the frontend can display the related data), as well as check the existence of vaults by a given controller and a specified array of IDs.

Creating a Vault

The first new move that is required is for creation of a vault. It can be sent by any name, and specifies the desired vault ID. This creates an unfunded, empty vault, whose controller is set to the account that sent the creation command:

```
      {"tv":{"c":{"id":42,"f":"founder","a":1234}}}
```

Funding a Vault

As a second step after creation, the founding user has to deposit their SMC, by sending a move themselves (proving that they consent to moving their coins into the vault). This can be done only for vaults that are unfunded, if the user's available SMC balance is large enough and by the user specified as founder. If these conditions are all satisfied, the coins (based on the initial balance specified during creation) are moved into the vault.

```
      {"tv":{"f":{"id":42,"c":"controller"}}}
```

Vaults that remain unfunded at the end of a block are removed again.
In the trade protocol, both creating and funding a vault will be done by the smart contract inside a single transaction. This will either succeed and result in a funded vault, or it will fail if the user's balance is not sufficient, in which case the vault will not exist at all afterwards.
Note that all the details of the vault are specified at creation; the funding step is just a confirmation by the user that they consent to moving their coins into the vault. Hence the only thing that matters for the final state, and that users may need to check in the GSP, is really whether or not a vault exists. If it does, then the state can be known and predicted precisely, assuming it was created (i.e. is controlled by) a smart contract with known logic.

Sending from a Vault

The controller of a vault (not the founder) can send coins from inside a vault to another user's normal balance, similar to how normal SMC transfers work. If a vault gets emptied completely, it can be removed from the game state (the balance inside a vault is non-increasing in time).

```
      {"tv":{"s":{"id":42,"u":"domob","a":1000}}}
```

Checkpointing

A special move can be used to request checkpointing of all new vaults:

```
{"tv":{"cp":{"n":1234,"h":"block hash"}}}
```

This move is valid if the provided block hash is a valid uint256 in hex with `0x` prefix, and the block height is a valid, positive int64. It will set this value as checkpoint for all vaults that:

- are controlled by the name sending the move,
- have not yet a checkpoint,
- and have a creation block height smaller than or equal to `n`.

Note that the value passed and stored as checkpoint in the GSP is arbitrary, and also does not necessarily need to match the provided block height. So it only corresponds to a real block hash (at which the vault existed) and matches the height in case the controller is a Democrit trading contract.

Trading Smart Contract

The trading contract is a non-upgradable smart contract, which is controller of the vaults used in trading, holds the order book, and executes the atomic trades as described in the overview.

Construction

The trading contract needs to own a Xaya account itself, which can be used to control the vaults inside the game state. This account will only be able to send moves per the code of the contract, which ensures that it can act as a "middle man" without requiring trust by any user.
After constructing the contract, we will register a name for it (can be the contract's address in hex if still available, for instance, so it can easily be associated back from the GSP) and transfer the name to it. This will "initialise" the contract.
Note that it would be simpler in theory to just auto-register the name in the constructor; but that seems not (easily) possible to do, since the contract needs a WCHI balance for this, and thus we would have to approve the contract's address pre-construction already for WCHI (so it can take the balance). Transferring the name manually after construction seems like the easiest alternative.

Contract State

The state of the trading contract needs to store these bits of data:

- Its Xaya account name,
- a nonce value for creating unique vault IDs,
- a list of all trading vaults that have been created through the contract and are controlled by it, together with a mirror of their state (vault ID, founder, balance),
- a list of all liquidity pools available for trading,
- the order book (list of limit orders that are active),
- the set of checkpointed block hashes, and
- the block height at which a not-yet-checkpointed vault was created, if any, used for triggering checkpoints from contract calls.

Handling of Vaults

The trading contract creates vaults controlled by itself by sending two moves in a single transaction: The vault creation from its own account, and the funding from the user's account who is doing some action (like creating an order). For the latter, it uses the Xaya move delegation contract, and it is required that the user has approved the trading contract for Soccerverse vault moves on it (path `["g", "smc", "tv", "f"]`).

It follows that any vault controlled by the trading contract either does not exist in the game state at all (if the funding user's SMC balance was not sufficient), or it exists and matches the state stored for it inside the contract. Any updates done to the vault in the game state can only be a consequence of code executed by the trading contract, and thus the contract is able to keep the state in sync.

Now as part of order execution (see below), buyers / recipients of in-game SMC need to verify the state of vaults inside the game state using their own GSP, as discussed above. For this, they will pass on the checkpoint retrieved for a vault from the GSP, and the contract will ensure it is a checkpoint it has seen in its own chain ancestry.

Escaping User Names

When the trading contract transfers coins from a vault, it needs to construct the move JSON based on the desired target user name. For this, it needs to escape the user name, which is a user-provided string, into JSON. As per the JSON spec, this means that we need to escale any UTF-8 codepoints that are control characters (value less than 0x20), `\` and `"`. Any other codepoint can be literally copied.

Note that it is even possible to operate in this way on raw bytes (instead of UTF-8 codepoints). All the things we need to escape are represented in UTF-8 by a single byte. If we copy all other bytes literally, this means that we either copy valid codepoints (as we should), or that, if the input is invalid UTF-8, produce invalid UTF-8 as well, which will be rejected by the GSP. So we can do the string escaping without worrying about UTF-8 specifically.

Also note that the trading-vault move format is not specifically susceptible to injection attacks. Since each vault move allows exactly one operation to be performed, even if an attacker would manage to inject arbitrary JSON after the username through a bug, they would not be able to produce a valid vault move that does something bad (like send more coins). Any other type of move besides vaults is not a concern either, because the contract's account name only controls vaults. Also note that the GSP will reject JSON that contains duplicate keys in dicts.

Account Names and Addresses

On-chain identities are addresses (also for holding/receiving/sending ERC-20 tokens). For game assets on the other hand, the identifiers are Xaya account names. The trading contract operates with account names, when it stores the owner of an order, vault or pool.

Any method that modifies assets tied to some account must be called (`_msgSender()`) by an address that is either the owner of or approved for that account name. Token balances transferred to a seller of SMC are sent to the current owner address of the seller's account

name. They are transferred from the `_msgSender()` in a market buy, and from the current owner address of the buyer's account in a market sell.
Another thing to consider is that names can be transferred and sold. In such a case, we need to make sure that orders are invalidated; otherwise this would open up vectors of attack and scamming people, for instance by creating a buy offer with a very high price, transferring the name to someone owning a lot of WCHI, and then accepting the offer. Hence, orders will record in their storage the owner address of the associated account at creation time. Whenever an order is accepted by a taker, the contract verifies that the owner address is still the same; otherwise it reverts. Sell deposits are simply vaults that only the account has any control over, so the same is not necessary for them. Similarly for trading pools, because only someone with a valid signature can use a pool, and the signature is tied to an address anyway (also by using a pool with a valid signature the pool operator can in effect only gain assets and never lose them in any way). For all these cases, if an account associated to any type of vault is transferred, the new owner obtains control over the vault, and will be able to redeem the contained assets.

Trading Pools

Trading pools that are available for limit buy orders can be registered with the smart contract. A user wishing to operate a pool can call a contract function `createPool`, which will vault the desired balance, and list the trading pool in the contract state / events, so potential buyers can see it.
Upon listing of a pool, the operator can specify a desired fee, which is then immutable for the existence of the pool (but of course the operator can cancel the pool and create a new one with a modified fee). This ensures that buyers are well aware of the fee, and it cannot be changed unexpectedly.
The pool can also store a contact URI in the contract, which defines a JSON-RPC endpoint at which requests to sign vaults can be sent to the pool operator.
If the vault corresponding to a trading pool gets emptied completely, the pool is removed. Any funds left inside a pool can be withdrawn by the pool's operator through another function, `cancelPool`.

Order Creation and Execution

Limit Sell Orders

Limit sell orders can be created with a contract call `createSellOrder`. The seller specifies the parameters (amount to sell and desired limit price) and their account name.
The contract then vaults the specified amount of SMC, and enters the order into the orderbook. The data stored for it are the vault ID (implying the remaining quantity of SMC to sell) and the amount of WCHI requested for the total (implying the limit price).
The seller themselves can cancel an outstanding order any time through `cancelSellOrder`, which accepts the vault ID and removes the order from the book, empties the vault back to the seller's SMC balance, and removes the vault.

Market Buy Orders

There are no direct "market" orders, but market buying is more or less equivalent to accepting one or more existing limit sell orders. That is what the contract allows through `acceptSellOrder` (with a variant for one order and a variant for an array of orders to accept).
The user passes in the desired order (by vault ID) and amount of coins to buy from it, their account name, as well as the checkpoint for which they have verified the vault in the GSP (see above).
The contract then verifies the GSP check as described, transfers the payment ERC-20 tokens from buyer to seller, and issues a move to send SMC from the vault to the buyer. If the vault is emptied, it gets removed together with the associated order from the book.

Limit Buy Orders

Before creating a limit buy order, the prospective buyer has to choose a liquidity pool to use and verify the associated vault in their GSP. They then call `createBuyOrder` on the contract, passing in the order details (amount and price), their account name, the liquidity pool's vault ID and the checkpointing data. The contract checks the vault verification and then lists the order.
Buy orders get assigned order IDs (since they are not associated to vaults). With the order ID, the owner can `cancelBuyOrder` any later time. The data stored in the contract state are the order ID (as key), the quantity to buy, the limit price, the chosen pool vault ID, and the buyer's account name.
By choosing a liquidity pool to use, the buyer also implicitly agrees to that pool's configured fee.

Market Sell Orders

For accepting a limit buy order, the prospective seller first has to create a vault with `createSellDeposit`, which just takes the account name and desired SMC balance. It locks the SMC balance into a newly-created vault. This vault is neither associated to a liquidity pool nor a limit sell order, like other vaults are (it is a *sell deposit*). It can be emptied with `cancelSellDeposit` at any time.
Once a deposit exists with (at least) the desired balance of SMC to be sold in it and is checkpointed, then the seller will need to contact the liquidity pool chosen by the owner of the buy order. They pass their vault ID, and the liquidity pool checks the vault, and if it is correct, signs the verification data (vault ID and checkpoint) with their private key.
The seller then calls `acceptBuyOrder`, passing the number of SMC to buy, the order ID of the buy order they are filling, their own vault ID (implying their accout name) and the signature obtained from the liquidity provider. The contract verifies the vault check data, verifies the signature (the signing address must be the owner or approved for the pool's Xaya account), and then executes the order:

- Payment ERC-20 tokens are transferred from buyer to seller,
- the pool fee in ERC-20 tokens is transferred from buyer to the pool operator,
- the sold amount of SMC is transferred from the seller's vault to the pool operator, and
- the amount of SMC is transferred from the pool vault to the buyer.

Frontend

Finally, let us discuss some details about how a frontend implementation of a market place based on this protocol will look like.

Approvals

As is usual with applications interacting with smart contracts, the user will need to give some approvals to contracts before they can meaningfully use the application. Thus the frontend should check for, and if missing request, these approvals:

- WCHI (or other currency used for trading) for the trading contract,
- the delegation contract is approved (ideally for all by the user's address) on the Xaya account, and
- the trading contract has `["g", "smc", "tv", "f"]` permission on the delegation contract for the accout name used.

Retrieving Data from the Blockchain

We can use The Graph as an indexing service for the data queried from the smart contract by the frontend (such as orderbooks). We will define a subgraph that can be queried for:

- All (or the best) active limit orders for either buying or selling,
- all (or the most liquid) pools for trading,
- all active orders by a user,
- all liquidity pools by a user, and
- a history of all completed trades.

Note that all vaults owned by a user (to be shown as part of their balance for instance, and to let them close them if desired) can be found from the GSP instead.

Also note that the lists of orders and pools need further verification; their associated vaults may not exist at all, buy orders may have become invalid if their associated pools have run out or the user's WCHI balance is too low, and so on.

We can deploy another smart contract (or include that in the trading contract) with logic to query for the status of individual buy orders: The approved WCHI balances associated with a given array of account names (which would be the buyer accounts) and the balances of the liquidity pools associated with an array of order IDs. This data is needed so the frontend can accurately reflect if those orders are valid and if so, for what maximum trading amount.

Orderbook Display

There will be a page where the frontend displays the orderbook, based on the data retrieved from The Graph and verified against the GSP. It will also show the current user's active orders, and allow the user to cancel them.

The orderbook should be filtered by orders that can actually be taken (and the displayed quantity reduced if necessary):

- For sell orders: The vault of the seller is valid in the GSP. Vaults that are valid but have no checkpoint yet should be highlighted somehow, and a button should allow to manually checkpoint if desired.
- For buy orders: The buyer's WCHI balance is sufficient, the selected liquidity pool is known (see below) and has sufficient balance.

Trade History

There will also be a page / area where the trade history is displayed in the usual form (e.g. candle sticks), based on data from The Graph.

Liquidity Pools

Support for liquidity pools in the frontend can be limited. It is expected that operators of pools run their own bot software anyway, so there need not be any support for them. The only thing the frontend needs to do is have a list of pools by known operators available (it can be hardcoded initially to just a pool run by us or something like that). This list can be displayed to the user to choose from when creating a limit buy order, displaying the remaining balance of the pool and the fee.
When a user accepts a limit buy order, the frontend needs to make sure it "knows" the pool used in the order (i.e. it is from the list), so that it can contact it for the signature on the user's vault.

Creating Limit Orders

There will be a form where users can create new limit orders. For limit sell orders, the frontend just needs to call the corresponding contract function. For limit buy orders, the frontend should also ask the user for the liquidity pool to use, displaying a list of "known" pools (see above), and verify that the user's current WCHI balance is sufficient at time of order creation. With this data, it can then create the order by calling into the contract function.

Accepting Limit Orders / Market Trades

There should also be a form to execute market trades, perhaps just called "simple" or something, where the users can enter the amount of SMC to buy/sell and just click a button to do it. The frontend does the matching, i.e. query for the list of best limit orders that are needed to fulfill the order completely, and display the resulting cost or gain in WCHI.
For market buying, the frontend will have already queried the GSP for the corresponding vaults while filtering the orderbook, so it can then just pass that information onward to the smart-contract function for accepting an array of limit sell orders.
For market selling, the frontend will then call the `createVault` function with the total SMC being sold; after confirmation of the transaction and checkpointing, it will get the vault signed by the liquidity pool(s) in the orders, and finally call `acceptBuyOrder` on the array of orders. In case something goes wrong after the vault is already created, it should display a list of vaults somewhere, so the user can cancel it (or just check if there are any orphan vaults and prompt the user to cancel them "automatically" on page load or something like that).