

Full Node GRPC Streaming

Last updated for:v4.1.3

This feature aims to provide real-time, accurate orderbook updates and fills. Complete orderbook activities and fills are streamed to the client and can be used to construct a full depth L3 orderbook. Streams are implemented using the existing GRPC query service from Cosmos SDK.

The current implementation provides information on orders and fills. Note that by dYdX V4's design, the orderbook can be slightly different across different nodes.

Disclaimer: We recommend you use this exclusively with your own self-hosted node.

Enabling GRPC Streaming

This feature can be enabled via a command line flag (`--grpc-streaming-enabled=true`) when starting your full node. This feature can only be used when grpc is also enabled.

Consuming the gRPC Stream

To follow along with [Google's documentation on gRPC streaming clients](#) (opens in a new tab):

1. Clone the [github.com/dydxprotocol/v4-chain](#) (opens in a new tab)
2. repository at same version as your full node.
3. Generate the protos: `make proto-gen && make proto-export-deps`
4. `.`
5. The generated protos are now in the `proto-export-deps`
6. directory.
7. Use the protobuf compiler (protoc) to [generate stubs](#) (opens in a new tab)
8. in any supported language.
9. Follow the documentation to write a streaming client.

For Python, the corresponding code is already generated in [the v4-proto PyPi package](#) (opens in a new tab).

Request / Response

To subscribe to the stream, the client can send a 'StreamOrderbookUpdatesRequest' specifying the clob pair ids to subscribe to.

// StreamOrderbookUpdatesRequest is a request message for the // StreamOrderbookUpdates method. message StreamOrderbookUpdatesRequest { // Clob pair ids to stream orderbook updates for. repeated uint32 clob_pair_id = 1; } Response will contain a oneof field that contains either:

- StreamOrderbookUpdate
 - Contains one or more OffChainUpdateV1
 - orderbook updates (Add/Remove/Update)
 - boolean field indicating if the updates are coming from a snapshot or not.
- StreamOrderbookFill
 - Contains a singular ClobMatch
 - object describing a fill (order or liquidation). * Represents one taker order matched with 1 or more maker orders.
 - - Matched quantities are provided for each pair in the match.
 - orders
 - field contains full order information at time of matching. Contains all maker and taker orders involved in the ClobMatch
 - object. * Prices within a Match are matched at the maker order price.
 - fill_amounts

- - contains the absolute, total filled quantums of each order as stored in state.* fill_amounts should be zipped together with the orders
- - - field. Both arrays should have the same length.

as well as some debugging metadata about block_height and exec_mode . It is not advised to write business logic around exec_mode , as some of these values are used by the underlying cosmos-sdk and are implementation-specific.

```
// StreamOrderbookUpdatesResponse is a response message for the // StreamOrderbookUpdates method. message
StreamOrderbookUpdatesResponse { // Orderbook updates for the clob pair. repeated StreamUpdate updates = 1 [
(gogoproto.nullable) = false ];

// ---Additional fields used to debug issues--- // Block height of the updates. uint32 block_height = 2;

// Exec mode of the updates. uint32 exec_mode = 3; }

// StreamUpdate is an update that will be pushed through the // GRPC stream. message StreamUpdate { // Contains one of
an StreamOrderbookUpdate, // StreamOrderbookFill. oneof update_message { StreamOrderbookUpdate orderbook_update
= 1; StreamOrderbookFill order_fill = 2; } }

// StreamOrderbookUpdate provides information on an orderbook update. Used in // the full node GRPC stream. message
StreamOrderbookUpdate { // Orderbook updates for the clob pair. Can contain order place, removals, // or updates.
repeated dydxprotocol.indexer.off_chain_updates.OffChainUpdateV1 updates = 1 [ (gogoproto.nullable) = false ];

// Snapshot indicates if the response is from a snapshot of the orderbook. // This is true for the initial response and false for
all subsequent updates. // Note that if the snapshot is true, then all previous entries should be // discarded and the orderbook
should be resynced. bool snapshot = 2; }

// StreamOrderbookFill provides information on an orderbook fill. Used in // the full node GRPC stream. message
StreamOrderbookFill { // Clob match. Provides information on which orders were matched // and the type of order.
ClobMatch clob_match = 1;

// All orders involved in the specified clob match. Used to look up // price of a match through a given maker order id.
repeated Order orders = 2 [ (gogoproto.nullable) = false ];

// Resulting fill amounts for each order in the orders array. repeated uint64 fill_amounts = 3 [ (gogoproto.nullable) = false ]; }
```

Maintaining a local orderbook

After subscribing to the orderbook updates, use the orderbook in the snapshot as the starting orderbook.

OrderPlaceV1

When OrderPlaceV1 is received, add the corresponding order to the end of the price level.

- This message is only used to modify the orderbook data structure (Bids, Asks).
- This message is sent out whenever an order is added to the in-memory orderbook.
- This may occur in various places such as when an order is initially placed, or when an order is replayed during the ProcessCheckState step.
- A Placement message will always be followed by an Update message.

```
func (l * LocalOrderbook) AddOrder (order v1types.IndexerOrder) { l.Lock () defer l.Unlock ()

if _, ok := l.OrderIdToOrder[order.OrderId]; ok { l.Logger. Error ( "order already exists in orderbook" ) }

subticks := order. GetSubticks () if order.Side == v1types.IndexerOrder_SIDE_BUY { if _, ok := l.Bids[subticks]; ! ok {
l.Bids[subticks] =

make ( []v1types.IndexerOrder, 0 ) } l.Bids[subticks] =

append (l.Bids[subticks], order) } else { if _, ok := l.Asks[subticks]; ! ok { l.Asks[subticks] =

make ( []v1types.IndexerOrder, 0 ) } l.Asks[subticks] =

append (l.Asks[subticks], order) }
```

l.OrderIdToOrder[order.OrderId]

```
order.I.OrderRemainingAmount[order.OrderId] =
0 }
```

OrderUpdateV1

WhenOrderUpdateV1 is received, update the order's fill amount to the amount specified.

- This message is only used to update fill amounts (Fill Amounts). It carries information about an order's updated fill amount.
- This message is sent out whenever an order's fill amount changes from an action that isn't aClobMatch
- .
- This includes when deliverState is reset to the checkState from last block, or when branched state is written to and then discarded if there was a matching error.
- An update message will always accompany an order placement message.

```
func (l * LocalOrderbook) SetOrderFillAmount ( orderId * v1types.IndexerOrderId, fillAmount uint64 , ) { l.Lock () defer l.Unlock ()
```

```
if fillAmount ==
```

```
0 { delete (l.FillAmounts, * orderId) } else { l.FillAmounts[ * orderId] = fillAmount } }
```

OrderRemoveV1

WhenOrderRemoveV1 is received, remove the order from the orderbook.

- This message is only used to modify the orderbook data structure (Bids, Asks).
- This message is emitted when an order is removed from the in-memory orderbook.
- Note that this does not mean the fills are removed from state yet.* When fills are removed from state, a separate Update message will be sent with 0 quantum.

```
func (l * LocalOrderbook) RemoveOrder (orderId v1types.IndexerOrderId) { l.Lock () defer l.Unlock ()
```

```
if _, ok := l.OrderIdToOrder[orderId]; ! ok { l.Logger. Error ( "order not found in orderbook" ) }
```

```
order := l.OrderIdToOrder[orderId] subticks := order. GetSubticks ()
```

```
if order.Side == v1types.IndexerOrder_SIDE_BUY { for i, o :=
```

```
range l.Bids[subticks] { if o.OrderId == order.OrderId { l.Bids[subticks] =
```

```
append ( l.Bids[subticks][:i], l.Bids[subticks][i + 1 :] ... , ) break } } if
```

```
len (l.Bids[subticks]) ==
```

```
0 { delete (l.Bids, subticks) } } else { for i, o :=
```

```
range l.Asks[subticks] { if o.OrderId == order.OrderId { l.Asks[subticks] =
```

```
append ( l.Asks[subticks][:i], l.Asks[subticks][i + 1 :] ... , ) break } } if
```

```
len (l.Asks[subticks]) ==
```

```
0 { delete (l.Asks, subticks) } } delete (l.OrderIdToOrder, orderId) }
```

Maintaining Order Fill Amounts

This message is only used to update fill amounts (Fill Amounts), it does not cause any modifications to the orderbook data structure (Bids, Asks).

TheClobMatch data structure exposed contains either aMatchOrders or aMatchPerpetualLiquidation object. Match Deleveraging events are not emitted. Within each Match object, aMakerFill array contains the various maker orders that matched with the singular taker order and the amount of quantum matched.

Note that prices are always matched at the maker order price. Theorders field in theStreamOrderbookFill object allow for price lookups based on order id. It contains all the maker order ids, and in the case of non-liquidation orders, it has the taker order.

Mapping each order inorders to the corresponding value in thefill_amounts field provides the absolute filled amount of quantum that each order is filled to after the ClobMatch was processed.

```
// fillAmountMap is a map of order ids to fill amounts. // The SetOrderFillAmount code can be found in theOrderUpdateV1`
section. func (c * GrpcClient) ProcessMatchOrders ( matchOrders * clobtypes.MatchOrders, orderMap map
[clobtypes.OrderId]clobtypes.Order, fillAmountMap map [clobtypes.OrderId] uint64 , ) { takerOrderId :=
matchOrders.TakerOrderId clobPairId := takerOrderId. GetClobPairId () localOrderbook := c.Orderbook[clobPairId]

indexerTakerOrder := v1. OrderIdToIndexerOrderId (takerOrderId) localOrderbook. SetOrderFillAmount ( &
indexerTakerOrder, fillAmountMap[takerOrderId])

for _, fill :=

range matchOrders.Fills { makerOrder := orderMap[fill.MakerOrderId] indexerMakerOrder := v1. OrderIdToIndexerOrderId
(makerOrder.OrderId) localOrderbook. SetOrderFillAmount ( & indexerMakerOrder, fillAmountMap[makerOrder.OrderId]) } }

func (c * GrpcClient) ProcessMatchPerpetualLiquidation ( perpLiquidation * clobtypes.MatchPerpetualLiquidation, orderMap
map [clobtypes.OrderId]clobtypes.Order, fillAmountMap map [clobtypes.OrderId] uint64 , ) { localOrderbook :=
c.Orderbook[perpLiquidation.ClobPairId] for _, fill :=

range perpLiquidation. GetFills () { makerOrder := orderMap[fill.MakerOrderId] indexerMakerOrderId := v1.
OrderIdToIndexerOrderId (makerOrder.OrderId) localOrderbook. SetOrderFillAmount ( & indexerMakerOrderId,
fillAmountMap[makerOrder.OrderId]) } }
```

Optimistic Orderbook Execution

By protocol design, each validator has their own version of the orderbook and optimistically processes orderbook matches. As a result, you may see interleaved sequences of order removals, placements, and state fill amount updates when optimistically processed orderbook matches are removed and later replayed on the local orderbook.

Example Scenario

- Trader places a bid at price 100 for size 1* OrderPlace, price = 100, size = 1
- - OrderUpdate, total filled amount = 0
- Trader replaces that original bid to be price 99 at size 2* OrderRemove
- - OrderPlace, price = 99, size = 2
- - OrderUpdate, total filled amount = 0
- Another trader submits an IOC ask at price 100 for size 1.* Full node doesn't see this matching anything so no updates.
- Block is confirmed that there was a fill for the trader's original order at price 100 for size 1 (BP didn't see the order replacement)* OrderUpdate, set total fill amount to be 0 (no-op) from checkState -> deliverState reset
- - MatchOrder emitted for block proposer's original order match, total filled amount = 1

FAQs

Q: Suppose the full node saw the cancellation of order X at t0 before the placement of the order X at t1. What would the updates be like?

- A: No updates because the order was never added to the book

Q: A few questions because it often results in crossed books: In which cases shall we not expect to see OrderRemove message?

- Post only reject? → PO reject won't have a removal since they were never added to the book
- IOC/FOK auto cancel? → IOC/FOK also won't have a removal message for similar reason
- Order expired outside of block window? → expired orders will generate a removal message
- Passive limit order was fully filled → fully filled maker will generate a removal message
- Aggressive limit order was fully filled? → fully filled taker won't have a removal

Q: Why does StreamOrderbookUpdate use IndexerOrderId and StreamOrderbookFill use dydxprotocol.OrderId?

- A: GRPC streaming exposes inner structs of the matching engine and our updates are processed differently from fills. The two data structures have equivalent fields, and a lightweight translation layer to go from Indexer OrderId to Protocol OrderId can be written.

Q: What are the exec modes?

execModeCheck

0

// Check a transaction execModeReCheck =

1

// Recheck a (pending) transaction after a commit execModeSimulate =

2

// Simulate a transaction execModePrepareProposal =

3

// Prepare a block proposal execModeProcessProposal =

4

// Process a block proposal execModeVoteExtension =

5

// Extend or verify a pre-commit vote execModeVerifyVoteExtension =

6

// Verify a vote extension execModeFinalize =

7

// Finalize a block proposal ExecModeBeginBlock =

100 ExecModeEndBlock =

101 ExecModePrepareCheckState =

102 Last updated on May 22, 2024 [Snapshots Types of Upgrades](#)