

# Orderbook Stream

This feature aims to provide real-time and accurate orderbook updates. Complete orderbook activities 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 initial implementation only contains orders but not trades. Also note that by dYdX V4's design, the orderbook can be slightly different across different nodes.

Disclaimer: It's possible for the full node to block indefinitely when sending a message to an unresponsive client, so right now we recommend you use this exclusively with your own node and that the client always close the gRPC stream before shutting down. This issue will be fixed in the next version.

## 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 on non validating full nodes and when gRPC is also enabled.

## 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 the orderbook updates (Add/Remove/Update), whether the updates are coming from a snapshot,
and a few fields used for debugging issues.

// StreamOrderbookUpdatesResponse is a response message for the // StreamOrderbookUpdates method. message
StreamOrderbookUpdatesResponse { // Orderbook updates for the clob pair. 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 ;

// ---Additional fields used to debug issues--- // Block height of the updates. uint32 block_height =
3 ;

// Exec mode of the updates. uint32 exec_mode =
4 ; }
```

## 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, total filled amount = 1

## Maintaining a local orderbook

Building a local orderbook should be fairly straight forward. Here is a quick [example PR\(opens in a new tab\)](#) for a Go GRPC client that subscribes to the orderbook updates and maintains an orderbook locally.

Specifically after subscribing to the orderbook updates:

- Use the orderbook in the snapshot as the starting orderbook.
- Add the corresponding order to the end of the price level when OrderPlaceV1
- is received.

```
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 l.OrderRemainingAmount[order.OrderId] =
0 } * Update the order remaining size when OrderUpdateV1 * is received
func (l * LocalOrderbook) SetOrderRemainingAmount (orderId v1types.IndexerOrderId, totalFilledQuantums uint64 ) { l. Lock
() defer l. Unlock ()
order := l.OrderIdToOrder[orderId] if totalFilledQuantums
    order.Quantums { l.Logger. Error ( "totalFilledQuantums > order.Quantums" ) } l.OrderRemainingAmount[orderId]
    = order.Quantums - totalFilledQuantums } * Remove the order from the orderbook when OrderRemoveV1 * is
    received.
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.OrderRemainingAmount, orderId) delete (l.OrderIdToOrder, orderId) } Last updated on April 3, 2024 How to interpret
block data for trades FAQs
```