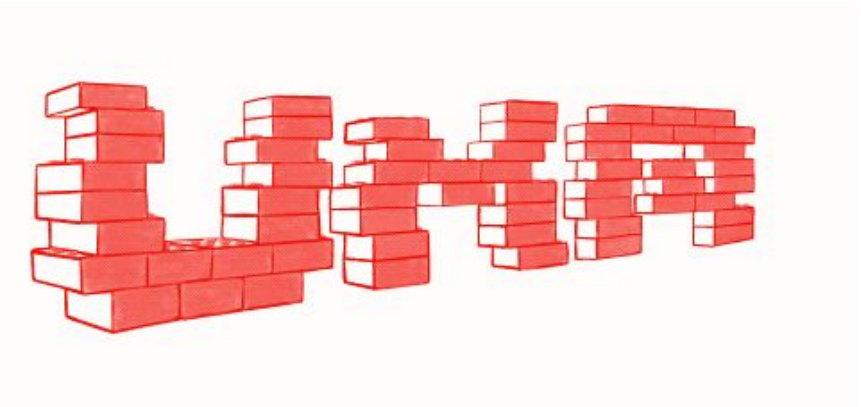

Universal Market Access

Across V2 Learning Session: Contents

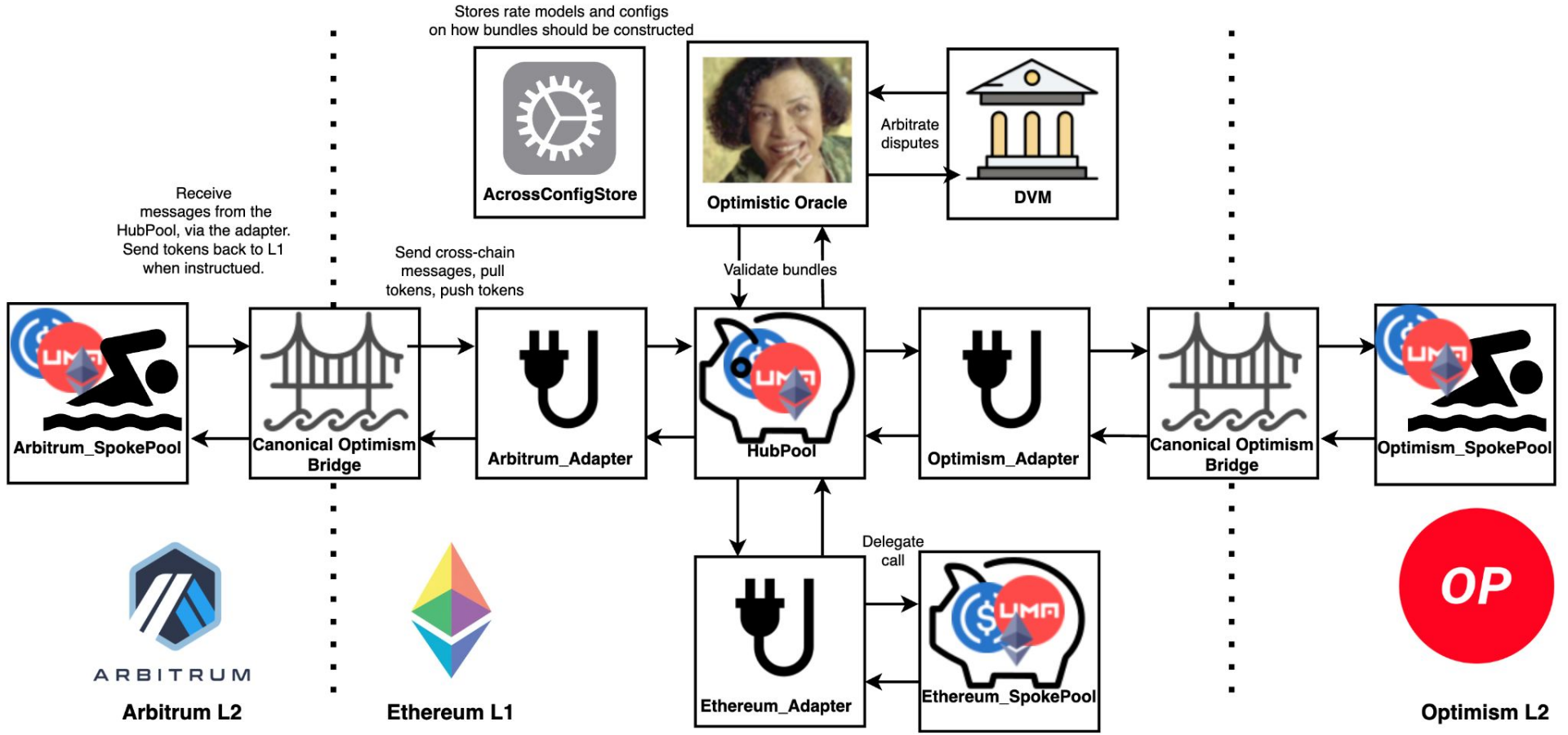
- Features in Across v2
- Across v2 implementation
  - Smart contract structure
  - Example transaction flow
  - Bunde all the things!
  - Routes
  - Fees
- Relayer bot
- Hard problems
- Resources



Across V2 Features

- **Ln <-> Ln bridge. Send funds from anywhere to anywhere.**
- All funds are concentrated on L1 to achieve high capital efficiency.
- Low gas costs for depositors (lower than v1!)
- Low gas costs for relayers(lower than v1!)
- Simple execution of relayer bot (easier than v1!).
- Simple OO integration by using only 1 kind of OO price request on L1.

Smart Contract Architecture



Contract Breakdown: Hub Pool

- Deployed on Ethereum L1 **only.**
- Stores all liquidity provider deposits:
  - LPs deposit/withdraw here.
  - Tracks fees for each enabled token.
- Main interaction point for data worker:
  - Submit and execute L1 bundles.
  - Dispute invalid bundles.
- Cross-chain Administrator of spoke pools deployed on each L2.
  - Can instruct L2 contracts to pull funds back to L1.
  - All governance actions originate here.
- Owned by the Across DAO multisig.
- Implements a fee capture mechanism similar to xSUSHI where a % of LP fees can be captured by a configured address.

Contract Breakdown: Spoke Pool

- Deployed on all enabled destination chains, including Ethereum L1.
  - I.e anywhere you could send money from/to in the system has a spoke pool
- Users *deposit* funds and relayer *fills* deposits.
  - Deposit tokens are locked on the *origin chain*.
  - Relayer fills the deposit on the *destination chain*.
- When instructed by the hub pool can:
  - Send funds back to L1 via canonical bridge due to a pool rebalance.
  - Refund relayers who chose to get refunded on the chain the spoke pool is deployed on.
  - Fill slow relays with funds sent over the canonical bridge from the hubpool.

Contract Breakdown: Chain Adapters

- Deployed on Ethereum L1. One contract per destination SpokePool.
  - I.e we have 5 destinations (Ethereum L1, Optimism, Polygon, Boba, Arbitrum) so 5 adapters
- Act to abstract away chain specific interactions by providing a unified interface.
- Can do two primary actions:
  a) Send tokens to associated destination chain.
    - Used for sending funds to refund relayers and fill slow relays.
  b) Send messages to associated destination chain.
    - This is used for executing governance actions and instructing the spoke pools on what to do.

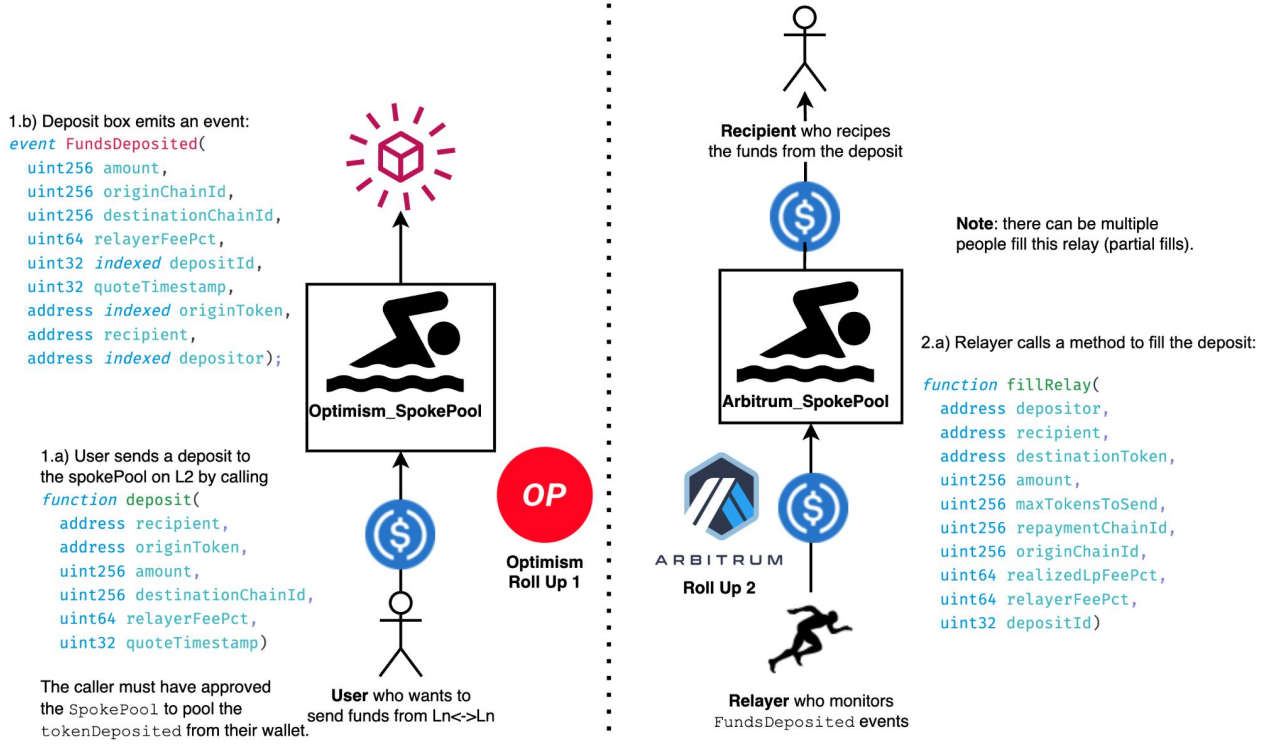
Contract Breakdown: Config Store

- Deployed on Ethereum L1 **only**.
- Contains key system parameters which govern how the protocol behaves:
  - **RateModel:** json file that informs the relationship between hub pool fund utilization and realizedLpFeepct charged to depositors for using the bridge at a commensurate rate.
  - **TokenTransferThreshold:** how often tokens are rebalanced between the hubpool and spoke pools
  - **MaxRefundCountForRelayerRefundLeaf:** maximum number of relayer refunds that should be fitted into one leaf. Prevents leaves that can't be executed due to being too large.
  - **MaxL1TokenCountForPoolRebalanceLeaf:** maximum number of l1 tokens that should be within one pool rebalance leaf. Prevents leaves that can't be executed due to being too large.
- Owned by the Across DAO who controls these settings.

Example Fills

Before going into how the bundling works for pool rebalances, relayer refunds and slow relays let’s go through some sample transactions to build an intuition around how the contracts and users interact with each other.

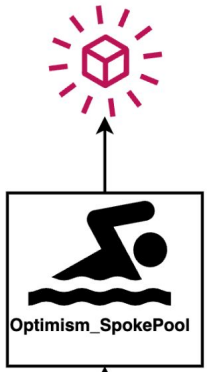
EG 1: Simple fill



EG 2.a: Partial Fill + Slow relay + relayer refund

1.b) Deposit box emits an event:

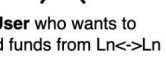
```
event FundsDeposited(
  uint256 amount,
  uint256 originChainId,
  uint256 destinationChainId,
  uint64 relayerFeePct,
  uint32 indexed depositId,
  uint32 quoteTimestamp,
  address indexed originToken,
  address recipient,
  address indexed depositor);
```



1.a) User sends a deposit to the spokePool on L2 by calling

```
function deposit(
  address recipient,
  address originToken,
  uint256 amount,
  uint256 destinationChainId,
  uint64 relayerFeePct,
  uint32 quoteTimestamp)
```

The caller must have approved the `depositBox` to pool the `tokenDeposited` from their wallet.



User who wants to send funds from Ln<->Ln

Optimism Roll Up 1

Recipient who recipes part the funds from the fill

Roll Up 2

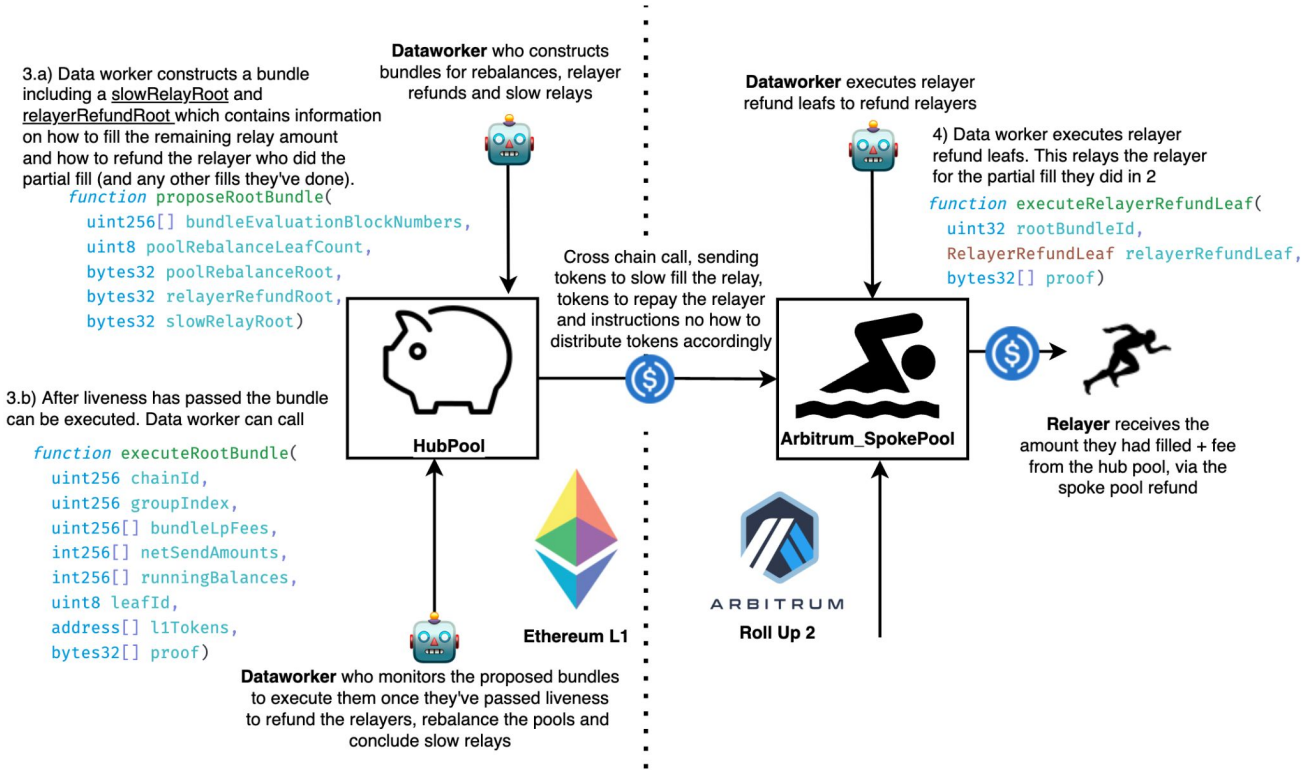
Relayer who monitors L2 1 `FundsDeposited` events to execute a partial fill

Note: there can be multiple people fill this relay (partial fills).

2) Relayer calls a method to <u>partially</u> fill the deposit

```
function fillRelay(
  address depositor,
  address recipient,
  address destinationToken,
  uint256 amount,
  uint256 maxTokensToSend,
  uint256 repaymentChainId,
  uint256 originChainId,
  uint64 realizedLpFeePct,
  uint64 relayerFeePct,
  uint32 depositId)
```

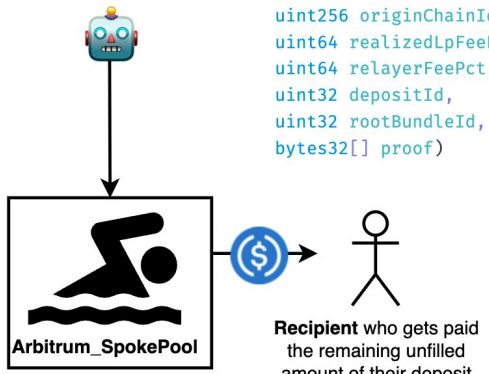
EG 2.b: Partial Fill + Slow relay + relayer refund



EG 2.c: Partial Fill + Slow relay + relayer refund

5) **Dataworker** executes slow relay bundle to pay depositor the unfilled amount

```
function executeSlowRelayLeaf(
  address depositor,
  address recipient,
  address destinationToken,
  uint256 amount,
  uint256 originChainId,
  uint64 realizedLpFeePct,
  uint64 relayerFeePct,
  uint32 depositId,
  uint32 rootBundleId,
  bytes32[] proof)
```

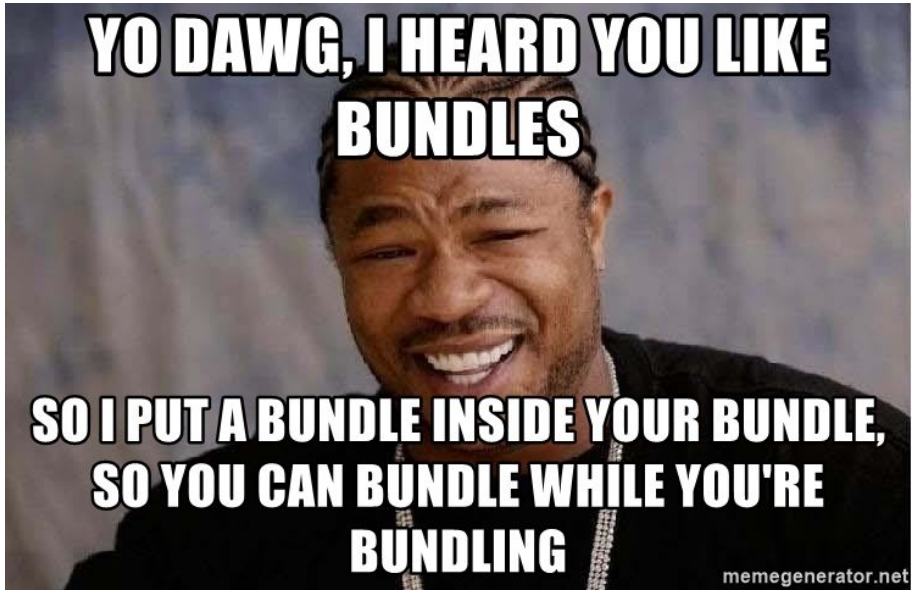


Recipient who gets paid the remaining unfilled amount of their deposit

ARBITRUM

Roll Up 2

Bundles all the things!



Bundles: overview

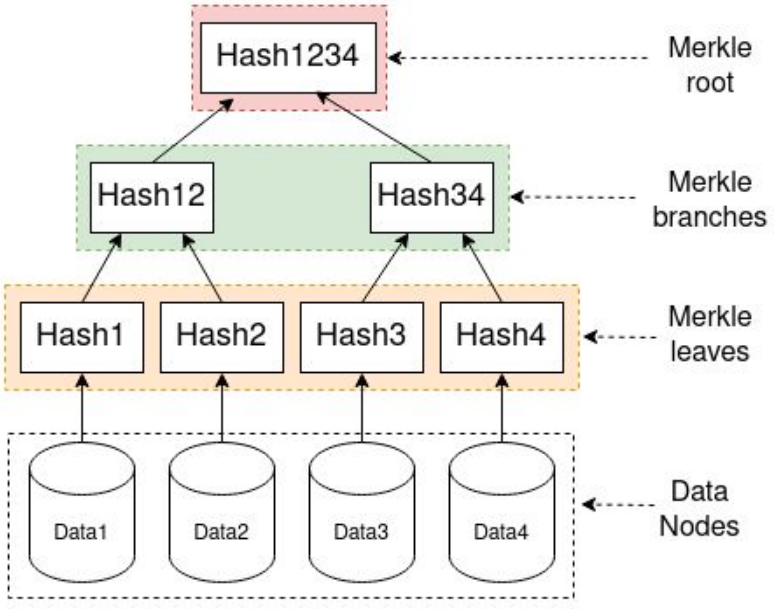
- Across v2s *speed*, *efficiency* and *low cost* cost primarily come from bundling
- Number of different kinds of bundles that are important to distinguish:
  - Bundling *data* in the form of a compressed data structure, called a merkle tree.
  - Bundling *transactions* in the form of sending batch transactions via multicall.
  - Bundling *liquidity* in the form of storing all funds together on L1.

Primer on Merkle trees and proofs

- A merkle tree is a not this

Primer on Merkle trees and proofs

- A merkle tree (or hash tree) is a data structure where:
  - Every *leaf* is hash of a *data block*.
  - Every *node* is a hash of its *children*.

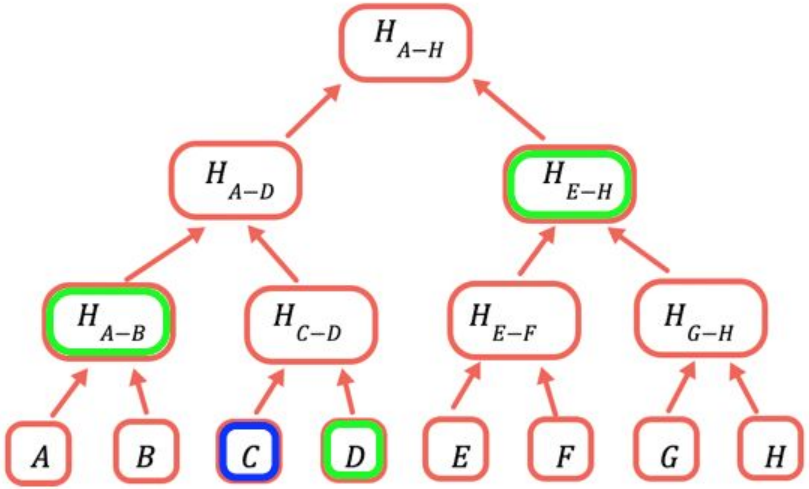


Primer on Merkle trees and proofs

- A merkle proof is an array of bytes32.
- Is used to *prove inclusion of a leaf in the original set of leaves*.
  - I.e an “inclusion proof”.
  - If you have the *root of the merkle tree* and I provide you with *a leaf* and the *proof* you can be convinced that the leaf I showed you was in the original set of leaves.
- Proof size is logarithmic in the size of the tree (base 2).
  - This is important because it means the proof size scales slowly for large data sets.
  - A tree with 8 leaves needs a proof of size 3.
  - But a tree with 100,000,000 leaves needs a proof size of 27.

Primer on Merkle trees and proofs

- The root hash is the only part that needs to be stored on chain to prove inclusion.
  - To prove a certain value, you provide all the hashes that need to be combined with it to obtain the root.
  - For example, to prove C you provide D, H(A-B), and H(E-H).



Primer on Merkle trees and proofs

- The only thing that needs to be stored on-chain for merkle tree is:
  - The root of the merkle tree (used when validating inclusion proofs)
  - A claims bitmap to store which leaves have been claimed (or in the case of across executed)
- Validation of proofs can be easily done using common libraries. OZ has one.
- Example proof:

```
[
   "0xfdc2f9fada1fd2964202b46150f246428cbaffc33dabdd4e2ade6b4bb31892fb",
   "0x834d97d1785331b3630a1316b795c6bba7fe967a36eb44066df777c0e3844447"
]
```

Primer on Merkle trees and proofs

- Merkle trees are used a lot in blockchain system.
  - Ethereum uses a variant called the *merkle patricia tree*
    - These store state, trancations receipts in separate trees
  - Uniswap airdrop used a merkle tree.
    - So did balancer, pie dao and UMA for our TVL tokens last year.

Data Bundles: Overview

- There are three main kinds data of bundles used in across V2:
  - **Pool Rebalance** - Used to push/pull tokens too and from the spoke pools.
  - **Relayer Refund -** Used to refund relayers for *valid* fills against deposits.
  - **Slow relays** - used to instruct spoke pools to slow fill a deposit that was not completely filled.


TELL ME MORE ABOUT ALL YOUR
BUNDLES
memegenerator.net

Data Bundles: Pool Rebalance Leaf

- Pool rebalance leafs indicate how funds should flow to/from the Spoke pools.
- Funds are pulled back from the spoke pools to the hub pool when:
  - The amount of funds in the spoke pool exceeds a configurable % of total system funds.
    - `TokenTransferThreshold` that we saw in the configStore
- Funds are pushed to the spoke pools when:
  - Executing relayer repayments on that chain.
  - Executing slow relays on that chain.
- Pool rebalance leafs also contain information about accumulated LP fees.

Data Bundles: Pool Rebalance Leaf

```
struct PoolRebalanceLeaf {
        // This is used to know which chain to send cross-chain transactions to (and which SpokePool to send to).
        uint256 chainId;
        // Total LP fee amount per token in this bundle, encompassing all associated bundled relays.
        uint256[] bundleLpFees;
        // Represents the amount to push to or pull from the SpokePool. If +, the pool pays the SpokePool. If negative
        // the SpokePool pays the HubPool. There can be arbitrarily complex rebalancing rules defined offchain. This
        // number is only nonzero when the rules indicate that a rebalancing action should occur. When a rebalance does
        // occur, runningBalances must be set to zero for this token and netSendAmounts should be set to the previous
        // runningBalances + relays - deposits in this bundle. If non-zero then it must be set on the SpokePool's
        // RelayerRefundLeaf amountToReturn as -1 * this value to show if funds are being sent from or to the SpokePool.
        int256[] netSendAmounts;
        // This is only here to be emitted in an event to track a running unpaid balance between the L2 pool and the L1
        // pool. A positive number indicates that the HubPool owes the SpokePool funds. A negative number indicates that
        // the SpokePool owes the HubPool funds. See the comment above for the dynamics of this and netSendAmounts.
        int256[] runningBalances;
        // Used by data worker to mark which leaves should relay roots to SpokePools, and to otherwise organize leaves.
        // For example, each leaf should contain all the rebalance information for a single chain, but in the case where
        // the list of l1Tokens is very large such that they all can't fit into a single leaf that can be executed under
        // the block gas limit, then the data worker can use this groupIndex to organize them. Any leaves with
        // a groupIndex equal to 0 will relay roots to the SpokePool, so the data worker should ensure that only one
        // leaf for a specific chainId should have a groupIndex equal to 0.
        uint256 groupIndex;
        // Used as the index in the bitmap to track whether this leaf has been executed or not.
        uint8 leafId;
        // The bundleLpFees, netSendAmounts, and runningBalances are required to be the same length. They are parallel
        // arrays for the given chainId and should be ordered by the l1Tokens field. All whitelisted tokens with nonzero
        // relays on this chain in this bundle in the order of whitelisting.
        address[] l1Tokens;
    }
```

Data Bundles: Relayer Refund Leaf

- Relayer refund bundles are used to refund a relayer for a set of valid fills
- The relayer will receive the sum of payments from the previous refund bundle
  - I.e if they have done 10 fills since the previous bundle they will get 1 refund containing all 10 refunds.
    - This is bundles within bundles! The relayer receives a bundle of refunds, within a bundle!

Data Bundles: Relayer Refund Leaf

```
struct RelayerRefundLeaf {
        // This is the amount to return to the HubPool. This occurs when there is a PoolRebalanceLeaf netSendAmount that
        // is negative. This is just the negative of this value.
        uint256 amountToReturn;
        // Used to verify that this is being executed on the correct destination chainId.
        uint256 chainId;
        // This array designates how much each of those addresses should be refunded.
        uint256[] refundAmounts;
        // Used as the index in the bitmap to track whether this leaf has been executed or not.
        uint32 leafId;
        // The associated L2TokenAddress that these claims apply to.
        address l2TokenAddress;
        // Must be same length as refundAmounts and designates each address that must be refunded.
        address[] refundAddresses;
    }
```

Data Bundles: Slow Relay Leaf

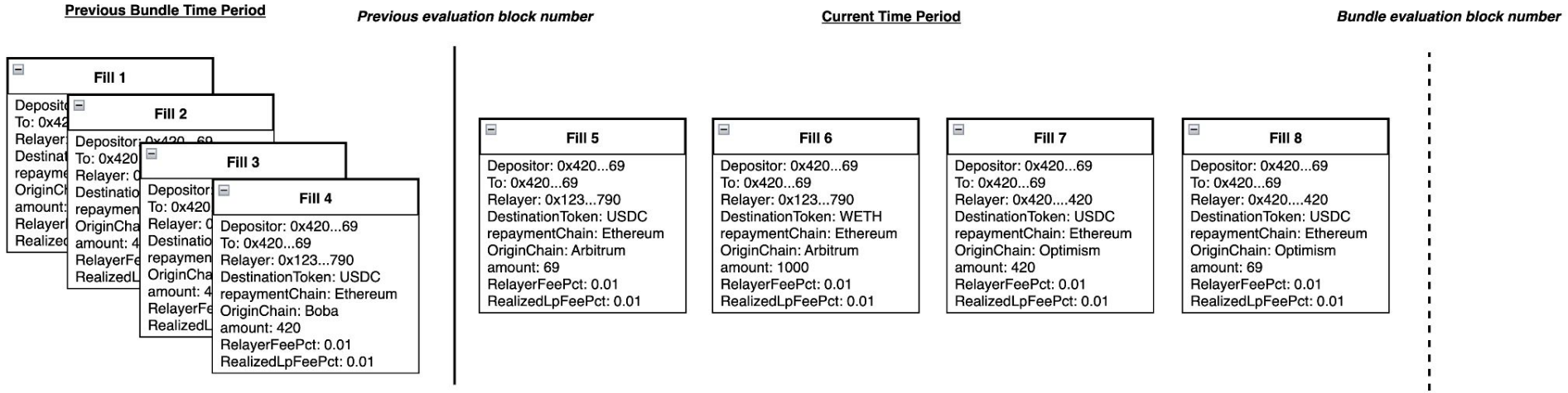
- Slow relay leafs effectively fill the remains of a relay from the hubpool's funds.
  - The leaf contain all fill information needed to conclude a fill.
- Slow relays , when execute, will always fill the remaining funds.
  - I.e if a relayer comes in after a slow relay leaf is created and fills some additional part of the deposit the slow relay will only fill the *remaining* amount.

Data Bundles: Slow Relay Leafs

```
struct RelayData {
        // The address that made the deposit on the origin chain.
        address depositor;
        // The recipient address on the destination chain.
        address recipient;
        // The corresponding token address on the destination chain.
        address destinationToken;
        // The total relay amount before fees are taken out.
        uint256 amount;
        // Origin chain id.
        uint256 originChainId;
        // Destination chain id.
        uint256 destinationChainId;
        // The LP Fee percentage computed by the relayer based on the deposit's quote timestamp
        // and the HubPool's utilization.
        uint64 realizedLpFeePct;
        // The relayer fee percentage specified in the deposit.
        uint64 relayerFeePct;
        // The id uniquely identifying this deposit on the origin chain.
        uint32 depositId;
    }
```

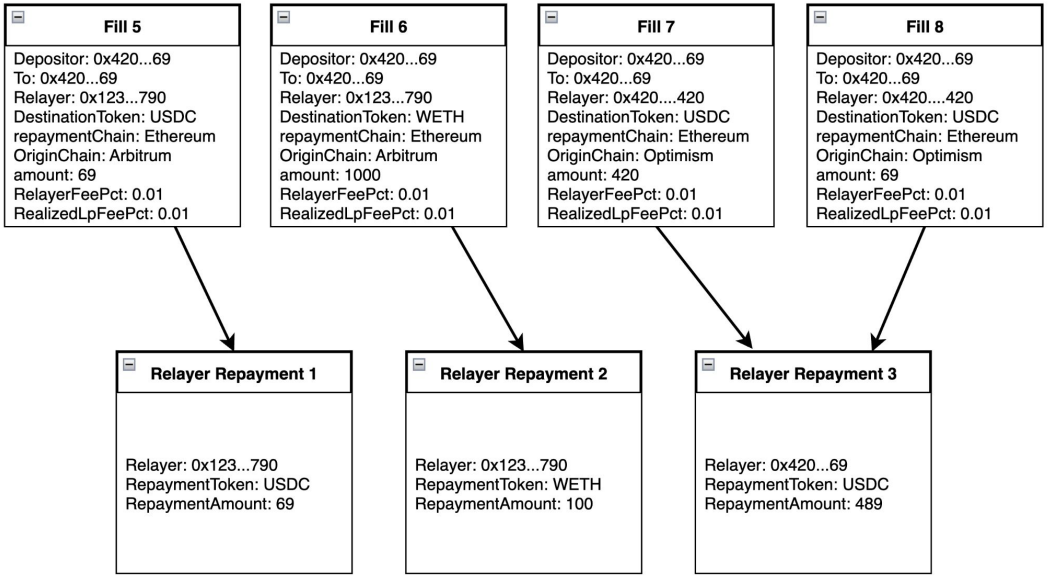
Bundle Construction Example 1

- **Step 1**: the data worker collects all valid fills within the current time period
  - Current time is from the previous evaluation block number to the current block number



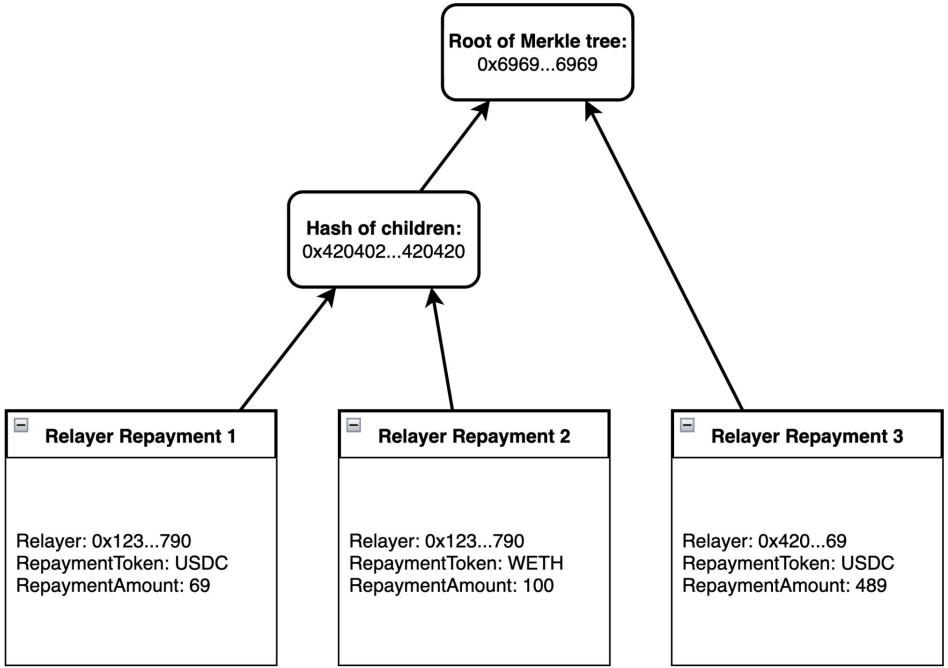
Bundle Construction Example 2

- **Step 2**: the data worker groups repayments by relayer and by token
  - I.e sum all valid fills for each relayer, for each token they've relayed



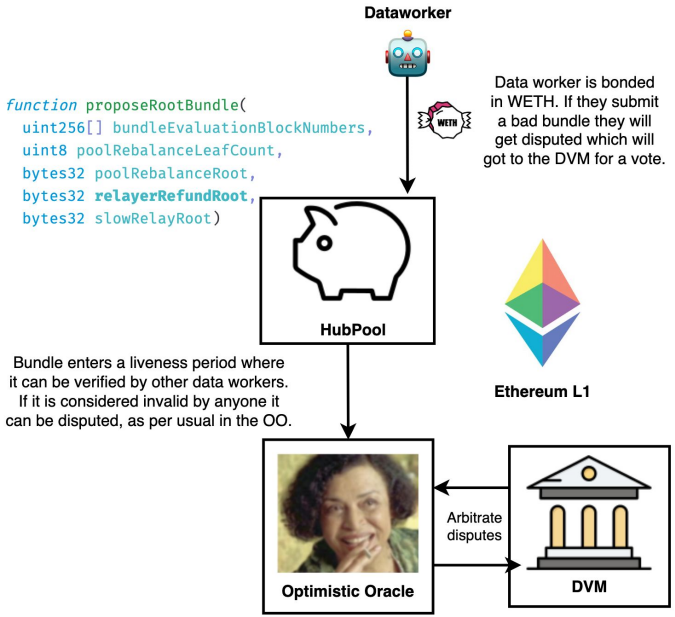
Bundle Construction Example 3

- **Step 3**: The data worker constructs a merkle tree of the relayer relayments
  - Each relayer repayment from the previous step is a leaf in the tree.



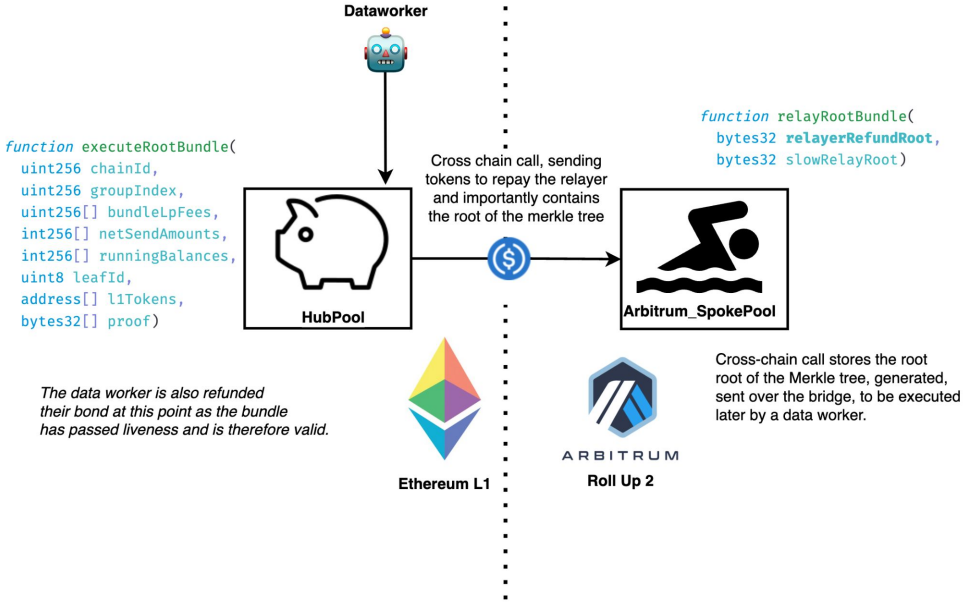
Bundle Construction Example 4

- **Step 4:** Data worker proposes the bundle to the hub pool.
  - In particular, notice the relayer refund root which we constructed in the previous step.



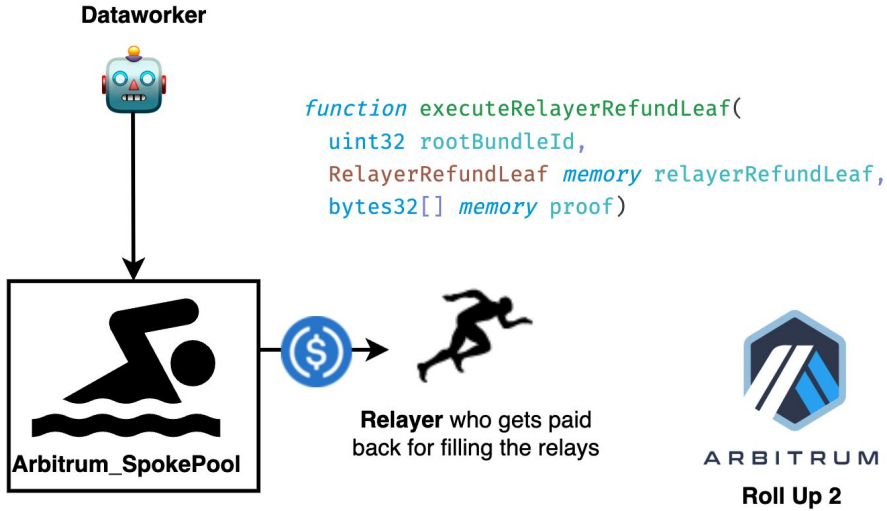
Bundle Construction Example 5

- **Step 5:** After liveness has passed the bundle can be executed.
  - Data worker will execute the associated leafId in the PoolRebalanceLeaf.
  - This sends a cross-chain message to the spoke pool containing the relayerRefundRoot



Bundle Construction Example 6

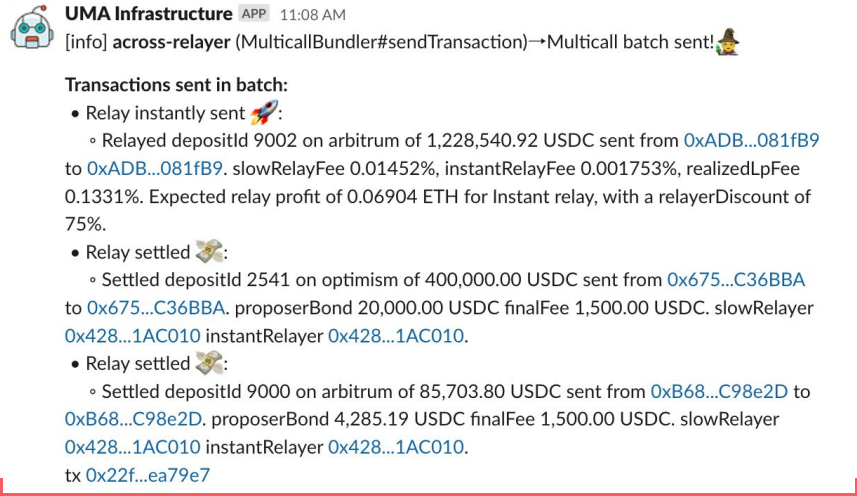
- **Step 6:** The spoke pool now contains the relayer refund root which can be executed
  - Data worker presents the root bundle Id, the whole contents of the leaf and the proof when executing.
  - Relayer is paid back.




YOU GET A BUNDLE! SHE GETS A BUNDLE! HE GETS A BUNDLE!
EVERYBODY GETS A BUNDLE!!!
memegenerator.net

Bundling of Transactions

- Across v2 smart contracts implement multicall
  - This enables a caller to send multiple transactions within one standard transaction.
  - This saves on gas by only spending the transaction overhead once as well as some EVM refunds you get for multi-reads of the same storage slots within the same transaction.
- This is equivalent to across v1 except we can bundle even more aggressively as all tokens go to the spokePool, rather than separate hub pools, as in v1.



UMA Infrastructure APP 11:08 AM

[info] **across-relayer** (MulticallBundler#sendTransaction)→Multicall batch sent!

Transactions sent in batch:

- Relay instantly sent 🚀:
  - Relayed depositId 9002 on arbitrum of 1,228,540.92 USDC sent from 0xADB...081fB9 to 0xADB...081fB9. slowRelayFee 0.01452%, instantRelayFee 0.001753%, realizedLpFee 0.1331%. Expected relay profit of 0.06904 ETH for Instant relay, with a relayerDiscount of 75%.
- Relay settled 🛰️:
  - Settled depositId 2541 on optimism of 400,000.00 USDC sent from 0x675...C36BBA to 0x675...C36BBA. proposerBond 20,000.00 USDC finalFee 1,500.00 USDC. slowRelayer 0x428...1AC010 instantRelayer 0x428...1AC010.
- Relay settled 🛰️:
  - Settled depositId 9000 on arbitrum of 85,703.80 USDC sent from 0xB68...C98e2D to 0xB68...C98e2D. proposerBond 4,285.19 USDC finalFee 1,500.00 USDC. slowRelayer 0x428...1AC010 instantRelayer 0x428...1AC010.

tx 0x22f...ea79e7


BUNDLES!
BUNDLES EVERYWHERE
makeameme.org

Bundling of Funds on L1

- One of the core design goals of L1 is to aggregate liquidity together on L1.
  - LPs add all their funds in one place and we only deploy funds to the L2s as and when required.
- The big advantage of this is we avoid fragmentation which results in lower fees.
- The pool rebalance root instructs to the spoke pool to send money back from L2 to L1.
  - This is done when `TokenTransferThreshold` amount of funds are on the L2.
  - This number is set by the Across DAO and can be changed to invoice how capital flows in the protocol.

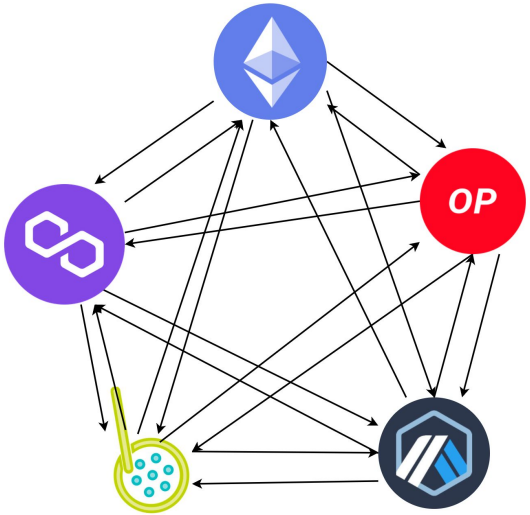
A word on “netting”

- In a perfect world, there are symmetric capital flows to and from all chains.
  - If flows are balanced, and if relayers choose to be refunded on the destination chain, then we don’t need to do any rebalances between any of the chains.
- In reality, the token flows won’t be balanced and are likely to be bursty.
  - A new yield farm is deployed on some Arbitrum and all liquidity flows there.
- In general it’s hard to model how the system will evolve over time.
  - We’ve tried to to design things to accommodate both symmetric and asymmetric flows through the pool rebalance mechanism.
  - As we run the protocol over a period of time we’ll be able to see what “normal” flows look like and use this to update the system parameters that govern how the pools are rebalanced.


"BUNDLES"
memegenerator.net

Token Routes

- In Across v2 any path a token can flow along is defined as a *route*
  - I.e USDC from Optimism to Arbitrum is a *route*.
- The total number of routes, per token, (assuming a token is on all chains) is n x (n-1)
  - I.e if we have 5 routes (Ethereum, Optimism, Polygon, Boba, Arbitrum) there are a total of 5*4=20 routes

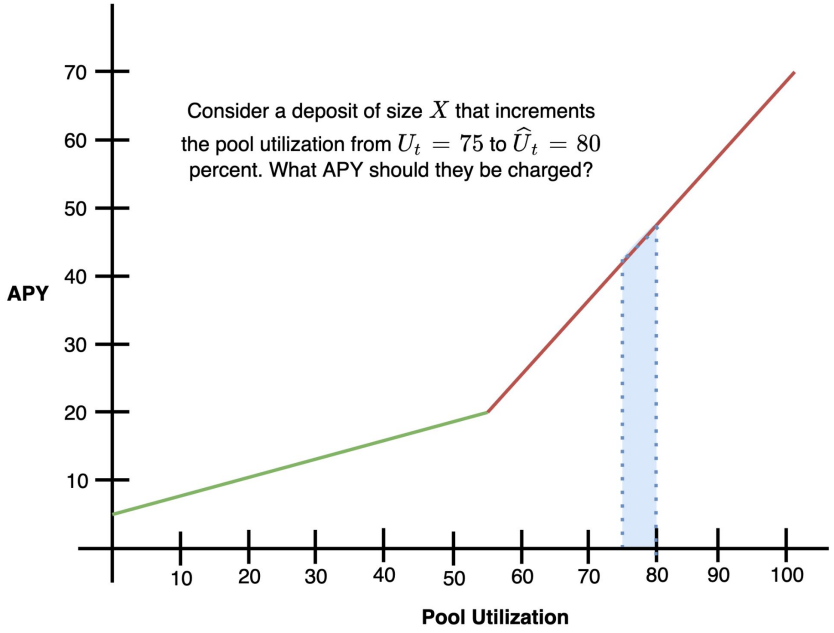


Fees 1: Overview

- Fees in Across v2 are a simplified version of that in v1, with some changes.
  - In Across v1 the depositor had to specify a `slowRelayFeePct` and a `instantRelayFeePct`
  - In Across v2 the depositor only specifies a `relayerFeePct`
- A “slow relay” is now done by the protocol and so the fee percentage is not needed.
- The depositor is refunded for any relayer fee that is sent via a slow relay.
  - Eg consider a user who deposits 100 000 USDC, setting their relayerFeePct to 1%.
  - Say half is relayed by a relayer and the other half is slow relayed.
  - For the half sent by the relayer:
    - The relayer will get 100 000 * 0.5 * 0.01 = 500 in fees,
    - The depositor will get 100 000 * 0.5 * (1 - 0.01) = 49 500 (half minus the 500 in fees).
  - For the second half sent by the slow relay the depositor will get the full 50 000
    - There is no relayer fee applied as the “protocol” did the remaining fill.

Fees 1: LP fee 1

- Liquidity provider Fees in across v2 are mostly the same as in v1.
- Use of a rate model to relate pool utilization to fee percentage paid by depositor.



Fees 1: LP fee 2

- V2 LP fees contains logic for "*protocol fee capture*" wherein the protocol will receive some amount of the realized LP fee percentage.
    - This is functionally equivalent to an xSUSHI like mechanism where the protocol can extract some % of all the LP fees generated.
- In the beginning this is set to 0% but can be changed by the Across DAO.

Fees 2: relayerFeePct with in cost of capital

- One difference vs V1 is that we want to add a larger incentive for relayers.
- In v1 the UI would set the relayer fee to simply cover the cost of gas of the relay
  - This meant that you'd earn the same amount for a 1000 USDC relay as you would for 100,000 USDC
    - Always equal to the gas price of the L1 transaction
- In v2 we want to compensate relayers for the cost of capital. Relayer fees should be:

$$relayerFeePct \propto gasSpent + relaySize * costOfCapital$$

- In practice, this means scaling the relayer fee percentage by the relay size and some proxy for the cost of capital.
  - The proxy we use right now is the L1 realizedLpFeePct, scaled by some time factor.
  - The realizedLpFeePct is the cost of capital LPs are receiving for a 1 week loan. The time scaling factor bringings this down to a 4 hour loan (time the relayer needs to wait to get refunded).
  - This is an imperfect measure but is a "good enough" proxy to get some idea of cost.

Relayer bot

HubPoolClient 🏛️

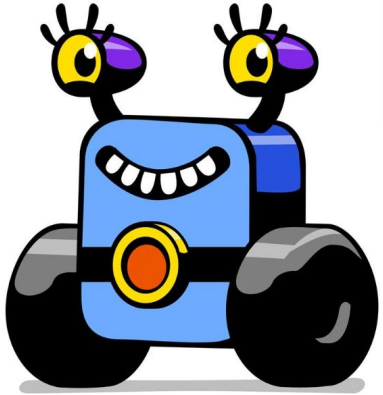
Fetches all LP information
Routing between tokens
Token L1 information
Used extensively by the data worker in constructing bundle information.

SpokePoolClient 🚤

One instance of the client per spoke pool (relayer has 5, 1 for each chain)
Stores and processes all deposits and all fill information.
Validates if a fill is correct or not
Stores all speedup events

TokenClient 🤑

Stores all token balance information.
Used to work out when relayer has a shortfall and cant fill a relay.
Manages approvals of tokens.



BotConfig 🔪

Controls the settings for the bot such as slack channels to post messages to, and config settings for all other clients.
Configured via env variables set in serverless or GCP.

ProfitClient 🛰️

Stores the token prices of each enabled token.
For each deposit works out if filling would be profitable.
Handles situation when deposits are not profitable.

ConfigStoreClient 💻

Handles all reading from the Across Config store which governs how the bot computes realised LP fees and how the data worker processes info.

InventoryManagementClient 👮

Controls how the bot allocates its funds over the chains it operates on. I.e decides when to push funds from L1 to the L2 (token shortfall) and on what chain the bot should get repaid on after doing a relay

Hard problems

- The primary hard problem we will run into is relayers running out of funds.
  - As soon as relayers funds are depleted we default to slow relays which is poor UX.
  - This should be solvable with incentives: pay a high enough APY and relayers will come
  - Another solution to this problem is the creation of a “community relayer”
    - Users can deposit funds into a communal centralized relayer.
    - This is like LIDO (staked eth provider) but as an Across relayer.
    - The main challenge with this solution is requires a trusted relayer to run the relayer bot.
      - We don’t want this to be Risk labs due to liability and associated risks.
- Overall system complexity.
  - All elements are very complex and somewhat difficult to reason about.
  - Upgradability is challenging due to some of the complexity.

Resources

- Across.to front end UI that will be updated after the launch
- Across v2 front end repo
- Across v2 contracts repo
- Across v2 relayer, data worker and monitor repo
- Across v2 UMIP with technical details on the protocol
- Diagrams used in this presentation