Running a Relayer

Technical instructions that someone comfortable with command line can easily follow to run their own Across V2 relayer. All of the code in this repository can be found in this GitHub repository .

Requirements

The Across v2 Relay Bot is implemented in Node.js and is capable of running on a variety of platforms. See the following table for platform recommendations. Resource Recommended CPU 64-bit Dual Core @ 2+ GHz RAM 4GB OS UNIX-like (GNU/Linux, MacOS)

Installation

Clone relayer code with Github CLI or git clone.

git clone https://github.com/across-protocol/relayer-v2.git cd relayer-v2

Establish environment file and restrict filesystem permissions.

touch .env chmod 0600 .env

The private key or seed phrase for the relayer can be stored in a dedicated file. Operators should be especially careful to set the file permissions correctly and to backup any secrets securely. The path to the secret is set via the SECRET env var (optionally specified in .env). The file may be stored anywhere in the file system but must be readable by the user that runs the relayer.

touch .secret chown

< user

: < group

.secret chmod 0600 .secret echo

< private-key-or-mnemonic

.secret chmod 0400 .secret

Install dependencies and build relayer.

Nodejs and yarn are required.

yarn

install yarn build

Run unit tests.

yarn

test

Apply any necessary changes to .env and mark it read-only.

chmod 0400 .env

Updating

A helper script is available to automate updates. This performs the following actions: * Flushes any existing installed dependencies. * Pulls down the latest relayer-v2 commit. * Installs all dependencies and builds the relayer. * Displays the latest commit in the relayer-v2 repository.

Important

This update helper is offered as a convenience. After update, the operator must manually verify that the update succeeded and that the commit shown matches the intended target. yarn update

Configuration

Environment Variables

This section describes the environment variable configuration that the bot requires in order to operate. This is the minimum configuration for running a relayer that operates on all supported tokens for all destination chains. You just need to change the configs at the top. Operators can exclude tokens/destination chains by not having a balance for that token on that destination chain. For customizing tokens/destination chains while having balances, see the advanced section for all supported configs.

Do change the following configs:

SECRET identifies a separate file containing a private key or mnemonic to be used by the relayer. The file must contain only the raw key or mnemonic. Critical: Ensure that the filesystem permissions for this file are properly configured (i.e. owned by one specific user, not world-readable, ...).

SECRET=

SECRET

.secret

Define RPC providers for each chain. One RPC provider is specified per line. Format:

RPC_PROVIDER__=

RPC_PROVIDER_INFURA_1

https://mainnet.infura.io/v3/ .. . RPC_PROVIDER_INFURA_10 = https://optimism-mainnet.infura.io/v3/ .. .

Repeat this for each supported chain (1, 10, 137, 42161, ...)

RPC_PROVIDER_ALCHEMY_1 = https://eth-mainnet.g.alchemy.com/v2/ .. .

Repeat this for each supported chain (1, 10, 137, 42161, ...)

Specify RPC provider preferences. The first provider is always used. Subsequent providers are used as backups in event of a higher priority provider being unavailable, or failing quorum. If NODE_QUORUM is > 1, there must be at least NODE_QUROUM number of providers defined, and some RPC queries will be performed against NODE_QUORUM number of providers in parallel.

RPC_PROVIDERS

INFURA,ALCHEMY

Per-chain overrides are possible. In the example below, LlamaNodes is preferred on Ethereum and Polygon.

RPC_PROVIDERS_1

LLAMANODES,INFURA,ALCHEMY RPC_PROVIDERS_137 = LLAMANODES,INFURA,ALCHEMY

Enable on-chain relayer functionality. This is disabled by default and must be explicitly enabled for the relayer to send transactions. This can be used to run bot in "simulation mode". To turn bot on, set to "true".

SEND_RELAYS

false

Deposit lookback window, specified in seconds. This is subtracted from the current time and is resolved to a block number on each chain, effectively controlling how far back in time the relayer will scan for unfilled deposits.

MAX_RELAYER_DEPOSIT_LOOK_BACK

1800

Gas fees are difficult to estimate correctly, and the strategy for setting gas might depend on the priorities of the relay bot operator. Gas fees can therefore be scaled on each chain. Note that ethers is used for sourcing gas estimates, and it can supply a default value of 1.5 Gwei for priority fees. This is notably seen on Optimism (chainId 10), and can lead to overpriced transactions. Operators are encouraged to tune these scalers to meet their own needs and risk profile.

MAX_FEE_PER_GAS_SCALER_10

1.2 PRIORITY_FEE_SCALER_10 = 0.1

Do not change the configs below without checking with the Across team

Or unless you strongly know what you're doing

A Redis in-memory DB can drastically speed up the performance of the bot. This is technically not required, but reduces the instance of repeated network queries and therefore reduces the time and network bandwidth required for successful relay bot operation.

Install redis and then ensure that redis-server is started: https://redis.io/docs/getting-started/

Under the hood, the relayer will cache JSON-rpc request data from requests like eth_getBlock in the Redis DB.

REDIS_URL

"redis://127.0.0.1:6379"

Advanced configurations

Amount of time to wait (in seconds) between bot loops. This can be set to 0 to run once and exit (recommended). Operators can schedule relayer externally (i.e. via cron or another job scheduler).

If set to a non-zero value such as 10, the bot will run through all instructions, sleep for 10 seconds, then run again. This mode is not

recommended.

POLLING_DELAY

0

The NODE_MAX_CONCURRENCY environment variable controls the maximum number of concurrent requests can be issued to a single RPC provider. Per-chain overrides are possible by appending _=. In the event that rate-limiting is occurring (429 responses to RPC requests) then concurrency can be reduced as an alternative to upgrading the RPC provider subscription/quota. In the example below, the global default is set to 25, and is overridden to 40 for Ethereum.

NODE_MAX_CONCURRENCY

25 NODE_MAX_CONCURRENCY_1 = 40

The relayer can be configured to require a minimum return on capital outlaid when filling relays. This minimum return is specified as a multiplier of the amount to be allocated to each fill. Minimum fees can also be configured per token(symbol)/route combination. Examples:

Require 1 bps as the global default (unless overridden).

MIN_RELAYER_FEE_PCT

0.0001

Override: Require at least 1.5 bps on USDC from Arbitrum to Ethereum.

MIN_RELAYER_FEE_PCT_USDC_42161_1

0.00015

Override: Require at least 0.8 bps on WETH from Optimism to Arbitrum.

MIN_RELAYER_FEE_PCT_WETH_10_42161

0.00008

The caching duration for a subset of the queries issued to RPC providers can be configured. The default time-to-live (TTL) of queries is 3600 seconds (60 minutes). This is set conservatively to refresh the cache often, so as to ensure that any incomplete or invalid RPC provider responses are ejected within the short term. Increasing cache TTL may improve the speed of the bot and lead to reduced RPC provider quota utilisation, at the cost of increased

resource usage (i.e. RAM + disk). Increasing cache TTL may also provide additional exposure to invalid or incomplete RPC responses.

PROVIDER_CACHE_TTL

3600

Minimum number of block confirmations (MDC) for a Deposit event before a relay bot will consider the Deposit finalised and valid. The MDC is set based on the total deposit USD volume in a relayer iteration and is specific to an origin chain. For example, if the relayer wants to fill 1000 of deposits originating from mainnet, then it will wait until a deposit is 32 blocks past HEAD before it sends a fill. If the relayer wants to fill 100 of deposits, then it will wait 16 blocks. Users can tweak these settings and take on more finality risk in exchange for reacting to deposits more quickly. The following configuration is equal to the default (i.e. the following config will be used if MIN_DEPOSIT_CONFIRMATIONS is not set. Finality assurances may change over time.

See code comments here for more details about the risks with modifying this configuration: https://github.com/across-protocol/relayer-v2/blob/0dde90a5909ce4ddc0dfb27e1ec8bcc1d75e2e25/src/common/Constants.ts#L25

MIN_DEPOSIT_CONFIRMATIONS

'{"1000": "{ "1": 32, "10": 0, "137": 100, "42161": 0 }", "100": "{ "1": 16, "10": 0, "137": 80, "42161": 0 }" }'

List of tokens that the relayer supports. These are addresses on Ethereum.

For example, if only USDC's address is specified, the relayer would only fill transfers of USDC going to any chains (unless overridden by RELAYER_DESTINATION_CHAINS).

If RELAYER_TOKENS is not set or set to [], the relayer will fill transfers of any token.

RELAYER_TOKENS

'["0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2", "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48", "0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599"]'

List of destination chains the relayer supports. If set to a non-empty list, only transfers going to these chains will be filled. In the example below, transfers destined for Ethereum will be ignored.

RELAYER_DESTINATION_CHAINS

'[10,137,42161]'

If true, the relayer will request slow fills for any deposits that it cannot fill due to insufficient balance. There is no direct incentive to the individual relayer for do this and Across functions correctly as long as at least one relayer requests slow fills. Operators are recommend to use the default setting of false.

SEND_SLOW_RELAYS

false

This inventory config will be explained in a separate section but generally this informs the relayer how it should rebalance token inventories across chains. The following simple example tells the bot that it should target holding 8% of its WETH on chain 10, 8% on chain 42161, 8% on chain 137, 0.2% on chain 288, and the remainder on Mainnet. The config can also be used to specify how much ETH versus WETH to hold.

RELAYER_INVENTORY_CONFIG

'{"tokenConfig":{"0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2":{"1":{"targetPct":100,"thresholdPct":100,"unwrapWethThreshold":3.5,"unwrapWethTarget":5},"10":{"targetPct":8,"thresholdPct":5,"unwrapWethThreshold":0.75,"unwrapWethTarget":1.5},"137":{"targetPct":8,"thresholdPct":5},"288":{"targetPct":0.2,"thresholdPct":0.1,"unwrapWethThreshold":0.65,"unwrapWethTarget":1},"42161":{"targetPct":8,"thresholdPct":5,"unwrapWethThreshold":0.75,"unwrapWethTarget":1.5}}}}'

Defaults to 1, must be set >= 1. If set > 1, then for any RPC request, this requires that at least this count of RPCs specified in the NODE_URLS_[ID] list return the exact same response. We recommend setting this to 2 so that the relayer does not accidentally relay a deposit whose properties two different RPCs disagree about. This edge case is rare but could lead to loss of funds.

NODE_QUORUM

1

Notes on requirements to RPC Providers

The relayer is dependent on querying historical blocks on each chain. The RPC provider must therefore support making archive queries. If the RPC provider cannot service archive queries then the relayer will fail with reports of obscure RPC errors.

Using a Redis in-memory database to improve performance

The relayer queries a lot of events from each chain's RPC that Across supports. Therefore, we use an in-memory database to improve performance and cache repeated RPC requests. Installation instructions can be found here . Once installed, run redis-server in one terminal window and then open another one to continue running the relayer from. The redis server is used to cache the responses of RPC-intensive repetitive requests, like eth_getLogs , or internally-computed data likegetBlockForTimestamp . Caching of data is subject to the age of the input response from the RPC provider, such that a minimum block age is required before it will be retained. This provides some protection against caching invalid data, or valid data becoming invalid (or otherwise changing) due to chain forks/re-orgs.

Managing cross chain inventory

The relayer bot is designed to use the same account across each of the supported chains: Ethereum, Optimism, Polygon, Boba and Arbitrum. Therefore, the bot can be told to automatically rebalance its inventory across chains and target some allocation. The best way to demonstrate how rebalancing can be customized is to walk through an example setting of the RELAYER_INVENTORY_CONFIG environment variable: "RELAYER_INVENTORY_CONFIG" :

{ "tokenConfig" :
{ "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2" :
{ "1" :
{
"targetPct" :
100 , "thresholdPct" :
100 , "unwrapWethThreshold" :
3.5 , "unwrapWethTarget" :
5
} , "10" :
{
"targetPct" :
8 , "thresholdPct" :
5 , "unwrapWethThreshold" :
0.75 , "unwrapWethTarget" :
1.5
} , "137" :
{
"targetPct" :
8 , "thresholdPct" :
5
} , "288" :
{
"targetPct" :
0.2 , "thresholdPct" :
0.1 , "unwrapWethThreshold" :
0.65 , "unwrapWethTarget" :
1
} , "42161" :
{
"targetPct" :
8 , "thresholdPct" :
5 , "unwrapWethThreshold" :
0.75 , "unwrapWethTarget" :
1.5
} } , "0x6B175474E89094C44Da98b954EedeAC495271d0F" :
{ "10" :
{
"targetPct" :
8 , "thresholdPct" :
5
} , "137" :
{
"targetPct" :
8 , "thresholdPct" :
5
} , "288" :
{
"targetPct" :
0.2 , "thresholdPct" :
0.1
} , "42161" :
{
"targetPct" :
8 , "thresholdPct" :

5

} } , "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48" :

{ "10" :

{

"targetPct" :

8 , "thresholdPct" :

5

} , "137" :

{

"targetPct" :

8 , "thresholdPct" :

5

} , "288" :

{

"targetPct" :

0.2 , "thresholdPct" :

0.1

} , "42161" :

{

"targetPct" :

8 , "thresholdPct" :

5

} } , "0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599" :

{ "10" :

{

"targetPct" :

8 , "thresholdPct" :

5

} , "137" :

{

"targetPct" :

8 , "thresholdPct" :

5

} , "288" :

{

"targetPct" :

0.2 , "thresholdPct" :

0.1

} , "42161" :

{

"targetPct" :

8 , "thresholdPct" :

5

} } } , "wrapEtherThreshold" :

2 }

Rebalances from L1 to L2

First let's look at the "tokenConfig". This informs the relayer how it should distribute its token balances across the different networks. Each of the keys of the "tokenConfig" object ("0xC02", "0x6B1", "0xA0b", "0x226") represent the Mainnet address of the ERC20 token that we want to automatically control inventory for. So, in order, the tokens in the example config are WETH , DAI , USDC , and WBTC . Diving into WETH's token configuration, notice that it has an object containing "targetPct", "thresholdPct", "unwrapWethThreshold", and "unwrapWethTarget" mapped to each network ID. The "targetPct" and "thresholdPct" instructs the relayer account on Mainnet when to send funds to the network with the associated network ID. The relayer is always aware of its aggregate funds across all chains, so for example if the account has 10 WETH on Mainnet, 5 on Optimism, 4 on Arbitrum, 3 on Polygon, and 2 on BOBA, then it has a total of 24 WETH. The allocations are: 5/24 on Optimism, 4/24 on Arbitrum, etc. When the relayer's allocation for a specific chain drops below the "thresholdPct", then the relayer will send funds from its Mainnet account to the chain with the shortfall so that its post-transfer allocation is increased to the "targetPct". In the example config above, if the allocation percent on Optimism were to drop below 5%, then the relayer would send funds from Mainnet to Optimism to bring its allocation to 8%. All funds are sent through the canonical L1-->L2 bridges and the logic implementing such rebalances can be found here. Note that the "targetPct" and "thresholdPct" are unused for the mainnet token config (i.e. for the row with ID "1") but are included to avoid a compile-time error.

Overriding repayment chain

When Inventory Management is configured, the relayer will tend to request repayment on mainnet. For example, if the relayer makes a USDC fill on Polygon, then it will expect to receive the USDC repayment on mainnet. This means that over time, the relayer's allocation on Polygon will decrease and the balance will shift to mainnet. Therefore, the inventory rebalance logic also provides a function that overrides the repayment chain ID to try to maintain its target allocation. This happens automatically before submitting any fill by this function

.

Unwrapping WETH

The "unwrapWethThreshold" and "unwrapWethTarget" tell the relayer when to unwrap WETH into ETH to keep enough ETH on hand for paying gas costs. These configs are unused on Polygon which does not pay gas in ETH. Moreover, these configs should only be set in the token configuration for WETH (i.e. "0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2"). If the relayer's ETH balance drops below the "unwrapWethThreshold", then it will unwrap enough WETH to increase its ETH balance to the "unwrapWethTarget" value. This logic can be found here . Wrap ETH The "wrapEtherThreshold" is used only for Optimism and Boba, the two OVM networks which requires that ETH, not WETH, is sent over the bridge from L2 to L1. Because of this fact, the relayer can end up holding a lot of ETH on L2 and not enough WETH. If the relayer account's ETH balance on Optimism or Boba drops below this value, then it will wrap any excess ETH above the value into WETH. Its important that this value is set higher than the "unwrapWethTarget" for WETH for Optimism and Boba. In the example above, the "unwrapWethTarget/Threshold" is 1.5/0.75 and the "wrapEtherThreshold" is 2, meaning that the relayer on Optimism and Boba will unwrap WETH if its ETH balance is less than 0.75 and increase its balance to 1.5, and will wrap ETH if its balance is above 2. Logic for wrapping ETH can be found here .

Security Considerations

This description describes a basic set of security considerations that relay bot operators should be aware of. See Recommendations for suggestions on how to improve the security of relay bot instances.

Disclaimer

This is not a complete security guide. Relay bot operators solely assume the risk of loss of fu

Host Attack Surface Area

The Across v2 relay bot communicates with various public RPC endpoints, and thus requires outbound network access. Unknown/untrusted network actors may be able to communicate with the relay bot host. It's important to reduce the attack surface area that the host environment exposes, in terms of: * Network services listening on public interfaces. * Interfaces opened for remote administration, and their permitted authentication mechanisms. * Third-party installations that may autonomously communicate (i.e. phone home) over the network.

Handling Secret Keying Material

The Across relay bot requires an in-memory copy of an Ethereum private key in order to sign relay transactions. This in-memory copy is typically loaded from the execution environment, via the following environment variables: * MNEMONIC (when run with --wallet mnemonic) * PRIVATE_KEY (when run with --wallet privateKey) When not specified directly via environment variables, these configuration items can be saved in the filesystem in a .env file, in the relayer-v2 working directory. Relay bot operators should be aware of at least the following: 1. 1. 2. Depending on the .env mode flags, users or programs with filesystem access may be able to read secrets from the .env file. 3. 2. 4. When storing secrets on disk, anyone with raw disk access may be able to override filesystem permissions and recover file contents. This includes: 5. 1. 1. 6. 2. People with system administrative privileges. 7. 3. 2. 8. 4. People with physical disk/hardware access. 9. 5. 3. 10. 6. Vendors of cloud-based execution environments (i.e. VM hosts). 11. 3. 12. During operation, the relayer-v2 bot retains secret keying information in-memory. Anyone with the ability to dump application or system memory may be able to retrieve secret keying material.

Outsourcing Secret Keying Material Storage

The Across relay bot supports a bespoke Google Cloud key management interface (gckms), whereby the bot retrieves keying material from a secured, trusted key management. Keying material retrieved over the gckms interface is stored locally in-memory, but is not saved to disk. The gckms interface is tailored for use by Risk Labs and is not currently intended for use by third-party bot operators. Support for generic third-party key management systems may be added to the relay bot in future.

Recommendations

1. 1. 2. Deploy relay bot instances in isolated environments (i.e. dedicated VM/container environment, or dedicated hardware). 3. 2. 4. Adhere to basic system administration and hardening practices. 5. NIST 800-123 Guide to General Server Security 6. may be useful. 7. 3. 8. Never install software from untrusted sources, and verify software packages before installation/execution. 9. 4. 10. Limit the ability for malicious network actors to gain system access by restricting inbound network traffic to trusted hosts and/or services. Drop all other traffic. 11. 5. 12. Avoid the use of password-based authentication schemes for remote login services. Use key-based authentication (i.e. SSH keys) instead. 13. 6. 14. Ensure that filesystem ownership and permission flags are set appropriately at all times. These attributes should be periodically reviewed for correctness. 15. 7. 16. Ensure that parties with raw disk access are trusted. 17. 1. 1. 18. 2. Note: Where untrusted or unknown parties may have raw disk access, filesystem encryption schemes 19. 3. may 20. 4. be useful in reducing the opportunity for theft of secret keying material. 21. 8. 22. Ensure that parties with the ability to dump system and/or application memory are trusted. 23. 1. 1. 24. 2. Note: This applies primarily to vendors providing virtualised execution environments - i.e. cloud/VM hosts. 25. 9. 26. Maintain at least one secure offline copy (backup) of the secret keying material. Backups should be periodically reviewed for correctness.

Running the Relayer for the first time

Once you've installed and built the relayer code and set your desired environment variables, you're all set to run the relayer code. The entry point to run the code is the command (choose one of the following):

Run the relayer, deriving private key from the SECRET env var (default)

SEND_RELAYS

false yarn relay

Run the relayer, overriding the private key source.

Sub in the desired key source (secret, mnemonic, privateKey, gckms).

SEND_RELAYS

false yarn relay --wallet

< secret | mnemonic | privateKey | gckms

> This will run the relayer in "simulation mode" meaning that it will simulate the transactions that would fill deposits, but will not submit them. This will give you a chance to review transactions before funds are sent. On the first run, the bot should approve various SpokePool contracts to withdraw ERC20's from it. This is required to fulfill relays. These approval transactions are not "simulated" currently and will still be sent even when running in simulation mode. Approvals will only be sent for tokens with nonzero balances in the relayer account. If the bot successfully completes a run, you will see this log before it exits: [debug] :

{ "at" :

"Relayer#index" , "message" :

"End of serverless execution loop - terminating process" } When you feel ready to run the relayer and send your first relay, set SEND_RELAYS=true to exit simulation mode!

Which account will be used to send transactions?

When running with a MNEMONIC configured, the first account associated with the MNEMONIC set in the environment will be used. Be sure to add ETH and any token balances to its account so that it can send relays.

Which tokens can be relayed?

WETH, USDC, DAI, USDT, WBTC, UMA, ACX, BAL, SNX & POOL. This list is likely to require updates in the future. Work is being done to automate the latest token list.

How can I learn more about the code behind the bot's logic?

The relayer's entry point file is Relayer/index.ts . The relayer bot first identifies all unfilled deposits across each of the chains. Deposit events are fetchedhere by the SpokePoolClient for each chain. Using these unfilled deposits as input, the relayer will attempt to fill them based if it has the token balance on the destination chain to do so.

Auxiliary Topics:

Across V2 smart contracts

The smart contracts can be found at this repository. They have been audited by OpenZeppelin . A high level video overview of the contract architecture can be foundhere .

Across V2 UMIP

UMIP-179 explains how "valid" relays are identified, which are relays that correctly filled a deposit and paid the correct fees and are due to be returned their relayed amount plus a relay fee. The Across relayer, proposer, disputer and executor instances are the UMIP-179 reference implementations.

Additional relayer FAQs:

How do Relayers get refunded?

Relayers are paid back after a root bundle containing the relayer's fulfillment is published to mainnet and passes a challenge period. Relayers can choose which chains they are paid back on. When running the example relayer code, they are by default repaid on the chain that they fulfill deposits on. The relayer can override this setting.

What is a root bundle?

A root bundle for a block range is a set of three Merkle roots that contains all of the information necessary to refund relayers who fulfilled a deposit during the block range. A root bundle is valid only if it contains all of the expected information for a block range. This UMIP explains at length exactly how to construct a valid root bundle.

How often do new root bundles get published and executed?

The current liveness period is 5,400 seconds (1.5 hours). You can always query it on the HubPool by calling the read-only method liveness() , and only one root bundle can be proposed at a time. So, the fastest cadence for root bundles being proposed is every 1.5 hours. Each root bundle contains approximately all of the relayer refunds up to the current time (as of the proposal) and as old as right after the preceding root bundle proposal time. A 'dataworker' is what we refer to as the agent who gathers all Fill and Deposit events from all of the chains that Across V2 supports bridging to and from in order to construct these Merkle root bundles. A dataworker could choose to propose a new root bundle right after the current one passes liveness, or it can choose to wait. Realistically, we expect that a dataworker will only submit a new root bundle after it contains a certain volume of refunds, for capital efficiency reasons. So to summarize: * Relayer refunds are contained in root bundles that are optimistically published to the HubPool * Once the root bundle proposal passes liveness, refunds can be sent to relayers. At this point, relayers are made whole and receive an additional relayer fee * The fastest cadence of root bundle proposals is once every two hours. In the worst case, root bundles can take much longer if volume is low across the system * A dataworker can propose a bundle at any time. However, if a current bundle is in liveness - it is required that either that bundle be disputed or the bundle passes liveness and is executed before a new bundle is proposed. Reference -Previous Tracking Events Next- Resources Release Notes Last modified24d ago On this page Requirements Installation Updating Configuration Notes on requirements to RPC Providers Using a Redis in-memory database to improve performance Managing cross chain inventory Security Considerations Running the Relayer for the first time Which account will be used to send transactions? Which tokens can be relayed? How can I learn more about the code behind the bot's logic? Auxiliary Topics: Across V2 smart contracts Across V2 UMIP Additional relayer FAQs: How do Relayers get refunded? What is a root bundle? How often do new root bundles get published and executed?