Cross-chain Borrow-lend Model: Centralized Liquidity (Hub & Spoke)

We propose a design and reference implementation for a Cross-chain Borrow Lend protocol, with centralized liquidity and with vaults. Currently, all EVM chains are supported, and this protocol could be extended to enable connectivity with Solana, Algorand, and other non-EVM chains.

Users can deposit collaterals from various different chains into a vault, and borrow against their vault on different chains. In section 1, we provide a philosophical overview of the rationale for building a cross-chain hub-and-spoke borrow-lend protocol. In section 2, we provide the workflow for each of the major operations in this protocol. In sections 3-6, we provide more detail on design of certain components of the protocol.

0. Table of Contents

1. Overview

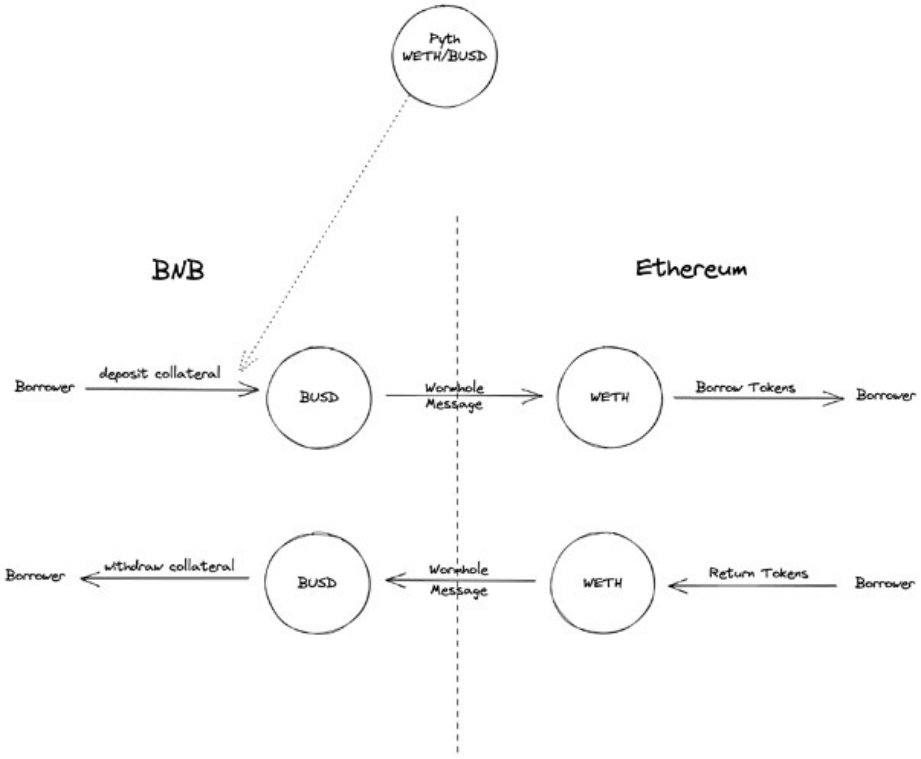
Why Cross-chain?

The rationale for cross-chain borrow-lend in particular is not very different from that for general cross-chain applications. Users want to interact with applications without having to migrate to a particular chain to do so, and many will just choose the easiest application to interact with from their native wallet—as opposed to the best one from a protocol design perspective.

Similar to the famous real-estate saying, the success of a DeFi application is intimately tied to “liquidity, liquidity, liquidity”. Cross-chain applications enable users native to many different chains to pool their liquidity on one application, instead of fragmenting it across multiple applications, each siloed to one chain. Borrow-lend user experience is predominantly a function of liquidity available, so cross-chain makes a lot of sense for this particular use case.

Why Hub-and-spoke instead of Point-to-point?

A point-to-point model of cross-chain borrow-lend could keep every chain identical in function and allow accounting between any pair of chains. Assets would be kept on native chains, without any wrapping, and Wormhole messages would be used to facilitate only accounting between chains. A schematic for this type of design can be seen below:



Meanwhile, a hub-and-spoke model features a single hub chain that functions as either a centralized accounting or liquidity layer, with different spoke chains sending either accounting messages (centralized accounting) or wrapped token transfers (centralized liquidity) upon token movements to the hub. The hub then serves as the authoritative source of state information, and any state stored on spokes is meant to be a copy of the ground truth state stored on the hub. Below, we motivate the hub-and-spoke model by examining some challenges with other models.

The point-to-point model is relatively simple in schema, but interest rate calculation dependencies actually make the pattern a lot more complicated to handle in practice. The interest rate the protocol charges for a loan or pays out for collateral deposits is continually changing as users make deposits and withdrawals, and the math in the aforementioned repo functions is meant to reflect the rebasing that continually updates in the cToken model.

There are 2 relevant interest indices: one for each asset, the borrowed asset on Ethereum and the collateral asset on BNB. The main challenge with this federated XC borrow-lend model is that both elements of state need to be tracked on *both* chains. In the model with more than 2 chains, state needs to be tracked and synchronized on *all*

chains. This is because when an initializeBorrow is successfully called from BNB, that instruction must issue a Wormhole message indicating how much of the borrow asset on Ethereum the user can withdraw. If the user calls completeBorrow on Ethereum with that VAA, they must be able to withdraw that much of the asset. Similarly, when a user pays back debt on Ethereum (either by repaying their own loan or liquidating someone else's vault), they must get back a VAA indicating how much of the collateral of the vault on BNB they can withdraw. They must then complete the repay or liquidate by taking that action on the source chain BNB. Throughout this time, BNB holds the "official" interest index for the collateral asset and its perception of the interest index for the debt asset, while Ethereum holds its perception of the collateral asset interest index and the "official" debt asset interest index.

So far, this seems fine. As long as the user redeems the VAA on the other side as soon as they get it, the states will stay synchronized. The issue comes if a user doesn't redeem a VAA or that VAA gets dropped somehow. In that case, the states will be out of sync, and if the VAA gets dropped it will be impossible for the user to withdraw their collateral until a recovery process takes place. Take the following example: the amount of BNB deposited on BNB is 10, and the amount of ETH withdrawn on Ethereum is 1.5. User X deposits 5 BNB, with the intent of withdrawing a loan of 1 ETH. initiateBorrow increases the amount of BNB deposited to 15 and creates a VAA to be redeemed for 1 ETH on Ethereum. The question then is whether the BNB tracker of the amount of ETH borrowed should be updated when the deposit of collateral is made with the intention of borrowing, or only after the borrow is completed and another VAA is sent back from Ethereum to BNB. If the former, then if the VAA is not redeemed, then the state will be as follows:

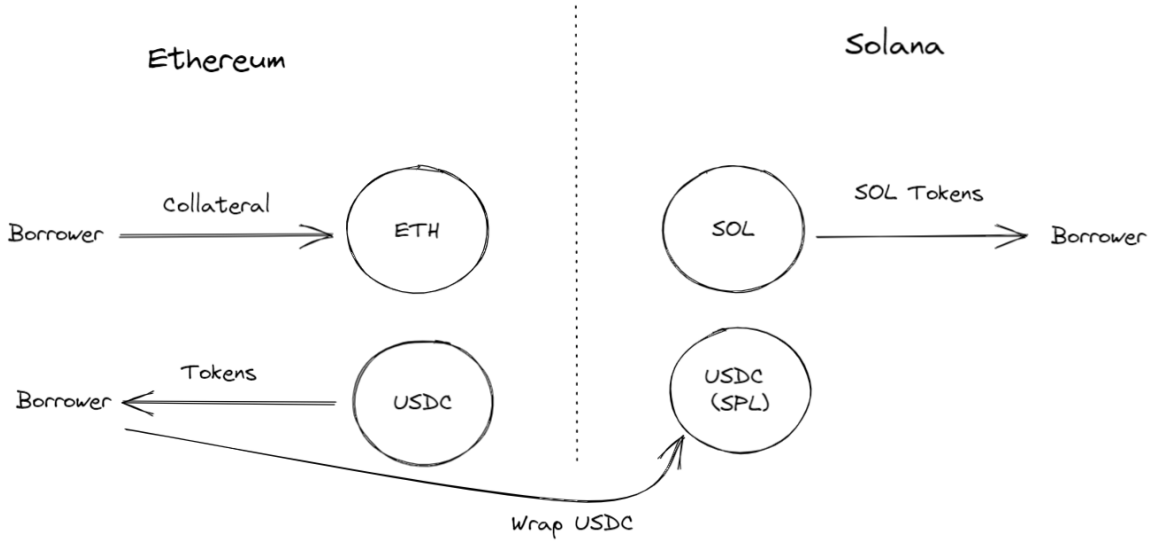
Highlighted in yellow are the "official" indices. Suppose user Y tries to deposit 20 BNB, with the intent of withdrawing a loan of 4 ETH. Their interest indices will not match between BNB and Ethereum, and let us suppose they complete the borrow on Ethereum. Let us suppose they end up draining the ETH reserves on Ethereum; then, once X tries to redeem their VAA to withdraw on Ethereum, they will get a failure.

For this reason and other examples of consequences of state asynchronicity, it would be better to have the latter model of keeping deposited assets in pending escrow until the depositor completes the borrow on Ethereum and a VAA indicating successful or unsuccessful action on the target chain is sent back to BNB. The issue here is that once that VAA is issued, there may be further state asynchronicity between the chains—the pending indices on BNB need to be updated to match what's on target chain. You could create VAA sequence IDs associated with these on-chain contracts, and require sequential processing of the VAAs, but that would require someone to redeem past VAAs in order to get their VAA and/or transaction processed, and could lead to a lot of transaction failures due to the non-atomic synchronization. Moreover, you may not be incentivized to spend gas redeeming your VAAs on-chain, so it may end up falling to the next user to complete the state synchronization before being able to do their transaction—this is potentially okay, but would need to be thought through.

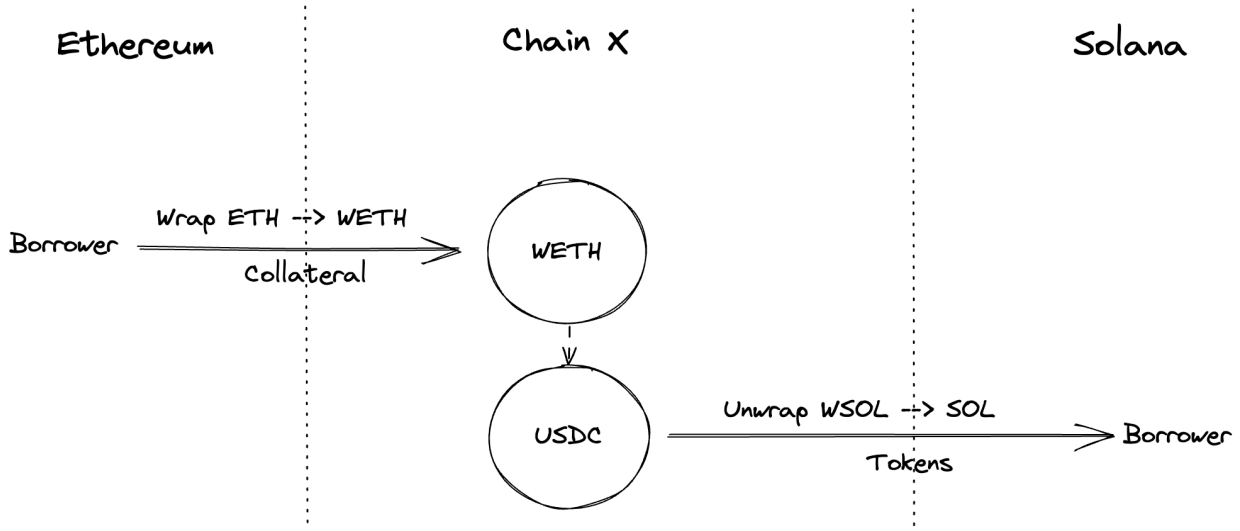
The fundamental issue is that in a single blockchain, state can be atomically updated; in a lending protocol context, both indices are tracked in the same program on the same chain, so they can be updated atomically without any other transactions coming in between. In the case of cross-chain borrow-lend where state is being tracked across multiple chains, synchronization cannot happen atomically, so either there is room for missequencing and error or enforcement of sequential processing of VAAs must be respected.

Instead of only communicating accounting in Wormhole messages, we could try to wrap tokens instead between deposit and withdrawal chains. One potential version of this is depicted in the schematic below, and as can be seen this model suffers from two big problems relating to interacting with two different chains:

1. Capital inefficiency—higher effective collateralization ratio (product of two different min collateralization ratios, one for the USDC borrow and one for the SOL borrow)
2. Liquidation risk—risk of liquidation on both chains' legs of the loan



An alternative to this is to have borrow-lend operations take place on one designated hub chain that serves as the central accounting layer for the whole cross-chain ecosystem. Suppose that layer is on chain X. Consider a user who wants to deposit ETH on an Ethereum spoke and take out a loan of SOL on a Solana spoke. Then, the schematic looks as follows:



This is better than the previous approach of deposits and withdrawals done directly on different chains, because now there is no state asynchronicity problem. The user does need to wrap assets and then deposit those wrapped assets on chain X to make deposits, and a withdrawer needs to take an action on chain X and then unwrap their assets, but all of this is configurable on the frontend. Because all accounting is done on a single blockchain, state is atomically updated, and there is no potential for asynchronicity. In essence, the ability to atomically execute and update makes it far easier to maintain this design of the protocol than a version where borrowing and lending happen on different chains. That type of design in any case does not provide too much for a user base that shouldn't really care all that much about where the borrowing and lending happen, as long as they can initiate the deposit and borrow of assets easily, on their preferred chains, and with good UX.

2. Workflow of Protocol Operations

Register Spoke

The first step in the workflow is to register all of the spoke contracts (one on each chain) on the hub. This enables deposits to be made into a spoke contract and communicated via transfer of wrapped assets to the hub, as well as for borrows, withdrawals, and repays to be initiated from the spoke and tokens received there. Registering a spoke can currently only be done by the owner of the hub contract, to prevent fraudulent spokes from being registered. This could be amended to allow governance to dictate which spokes can be registered.

To register a spoke, the owner must provide the chainId of the chain that the spoke is deployed on and the contract address of the spoke on that chain.

Register Asset

Next, the owner can register an asset on the hub and then communicate that registration to the spokes via Wormhole. Again, only the owner of the hub contract can currently register assets, but this could be amended to leave it in the hands of governance. To register an asset, the owner must provide the asset address, the minimum collateralization ratios for depositing and borrowing (see below), the reserve factor and precision, the pythId mapping to this asset's prices, and the decimals of the asset. The contract then checks that the asset address is not already registered before storing the information in the hub state.

Previously, we had the hub broadcast a Wormhole message intended for the spokes to complete registration of the asset on their side. This was for user safety purposes, to ensure that assets were registered on both the hub and spoke in order to initiate any action with that asset on the spoke. However, this runs into issues with broadcasting past asset registrations to newly registered spokes. One option is to allow any user to permissionlessly reregister each asset on the new spoke, by making cached Wormhole registerAsset messages available to users. Another option is to send as part of the completeRegisterSpoke message the existing allowList. However, we decided to remove the need to complete registration of assets on the spoke, and to pass the checking of an asset being on the official allowList on the hub to frontends.

Deposit

A deposit is initiated from the spoke. A Wormhole payload3 message is sent with the appropriate wrapped token transfer to the hub contract. When this message and the tokens are received, they can be parsed and processed via a call to completeDeposit on the hub contract. This ensures that the deposited asset is valid and registered before updating the accrual indices and normalized token amounts (see below) in this vault and in the protocol. This accounting step is important to enable this vault (identified with a specific user's wallet address) to withdraw or borrow assets in the future. Currently, deposit, borrow, withdraw, and repay all take place with only one asset address at a time.

An important note is that we only allow wrapped assets to be deposited on the hub side. This only affects the native gas token of the hub chain, and we explain in more detail why we make this design choice in the Wormhole Solution Details section. What this entails is that the wrapped version of the native gas token can be registered as an asset and subsequently deposited, borrowed, withdrawn, repaid, and liquidated, but no such pool exists for the native token. However, we do allow native gas tokens on spoke chains to be wrapped and deposited into the hub contract; this extends to any spoke developed on the hub chain, which would mean a gas token deposited into that spoke would be wrapped and then deposited into the hub contract in its wrapped form (e.g., if the hub chain was BNB, we could send native BNB gas token to a BNB spoke, and it would be converted to wBNB before being deposited in the hub contract).

Borrow

A borrow is initiated from the spoke. Instead of a Payload3 message, a general Wormhole message is sent, since no tokens need to be transferred to the hub. The hub contract parses and processes the Wormhole message via a call to completeBorrow, and then it carries out similar validation and accounting measures. Here, though, the hub contract needs to carry out a check of the notional values of the current deposits, current borrows, and new intended borrow of the vault to ensure that the vault is not underwater. More detail on this is provided in the example flow and the price oracle sections below.

If the validation and accounting checks are successful, then the protocol transfers tokens to the user vault on the spoke via the token bridge.

Withdraw

The workflow for withdraw is fairly similar to borrow. A general Wormhole message is sent from the spoke to the hub, since no tokens are transferred to the hub. The hub contract parses and processes the Wormhole message via a call to completeWithdraw, and then it carries out validation and accounting measures, including a check of the notionals to ensure that this withdraw does not take the vault underwater. If the checks are successful, then the protocol transfers the withdrawn tokens to the user vault on the spoke via the token bridge.

Repay

The workflow for repay is fairly similar to deposit. A Wormhole payload3 message is sent with the appropriate wrapped token transfer to the hub contract. The hub contract parses and processes the message via a call to completeRepay, and then it performs the necessary validation and accounting checks and updates.

Unlike deposit, the repay operation must handle reversion. To illustrate, consider the following example—a user close to liquidation tries to repay some of their borrow, but their vault goes underwater and is liquidated partially before their repayment is completed on the hub. Suppose the amount they are repaying is now greater in quantity than their outstanding borrow post-liquidation. Then, they are no longer able to complete their repay, and we need to do something with the tokens bridged over. One option is to just convert their repayment over to a deposit; however, this is bad user experience and complicates the protocol. Another more intuitive option is to repay whatever is outstanding for that asset and bridge the remainder back to the user on the spoke. This is certainly an option, though for simplicity in this reference example, we just bridge back everything to the user on the spoke. To accommodate this, the repay payload has an extra field for the reversionPaymentChainId, which is the chain Id that the repayer would like their tokens to be sent to their address on in case of failed repayment. This can be configured to be user-specified, though in our implementation, we simply set this to the chain Id of the spoke whence the repayment originated.

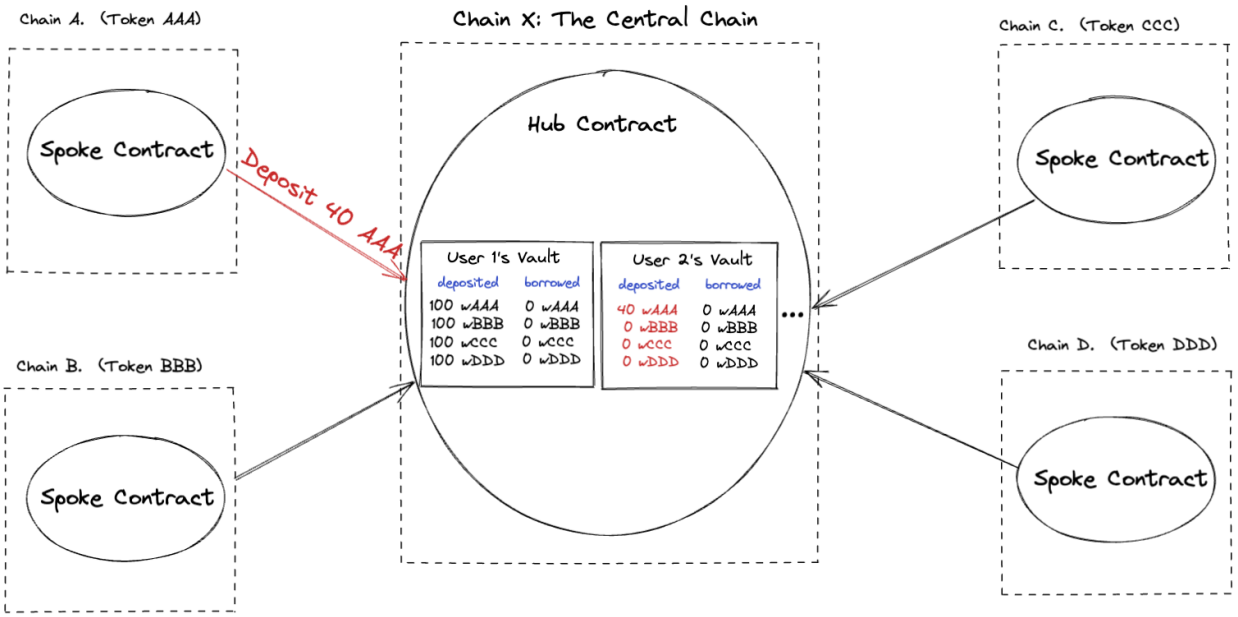
Liquidation

liquidation can only be triggered from the hub, for reasons discussed in the Liquidation Design Choices section below. A liquidator set up on the hub calls liquidation with the address of the vault they wish to liquidate, the addresses and amounts of the assets they wish to repay, and the addresses and amounts of the assets they wish to receive. The hub then checks that each asset address is registered and unique, before checking that the intended liquidation is valid, updating the accrual indices and amounts for each asset, and paying out the tokens to the liquidator on the hub chain. More details on the liquidation process are shared in the Liquidation Design Choices section below.

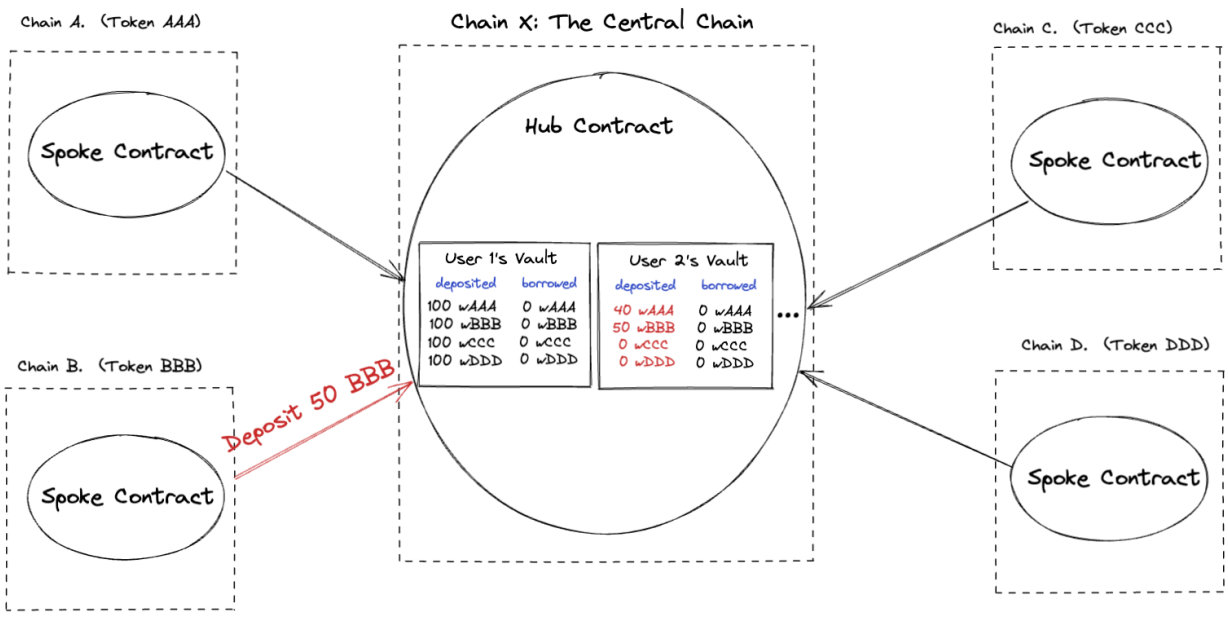
Example Flow

Suppose User 2 wishes to borrow 35 CCC against collateral of 40 AAA and 50 BBB. Currently, User 2's vault on the Hub is empty. User 2 employs the following steps.

1) User 2 calls 'depositCollateral' (with 40 AAA) on the spoke contract on Chain A. This transaction creates a Wormhole Token Bridge VAA with 40 wormhole-wrapped AAA (wAAA). Then, a relayer relays this VAA to the Hub Contract on chain X by calling 'completeDeposit', which retrieves 40 wAAA from the TokenBridge contract on Chain X, and updates the Hub state



2) User 2 calls 'depositCollateral' (with 50 BBB) on the spoke contract on Chain B. This transaction creates a Wormhole Token Bridge VAA with 50 wormhole-wrapped BBB (wBBB). Then, a relayer relays this VAA to the Hub Contract on chain X by calling 'completeDeposit', which retrieves 50 wBBB from the TokenBridge contract on Chain X, and updates the Hub state.



3) User 2 calls 'borrow' (with input '35 CCC') on the spoke contract on Chain C.

This transaction creates a Wormhole Core Bridge VAA. Then, a relayer relays this VAA to the Hub Contract on chain X by calling 'completeBorrow'.

On the Hub, the completeBorrow function first checks to see if this borrow is valid; i.e.

1) does the user have enough value in their vault for this borrow to be valid (we use Pyth prices of the assets in the user's vault to check this). Normally, we would need:

$$40 \cdot (\text{price of wAAA}) + 50 \cdot (\text{price of wBBB}) \geq 35 \cdot (\text{price of wCCC})$$

However, our protocol makes use of minimum **collateralization ratios**.

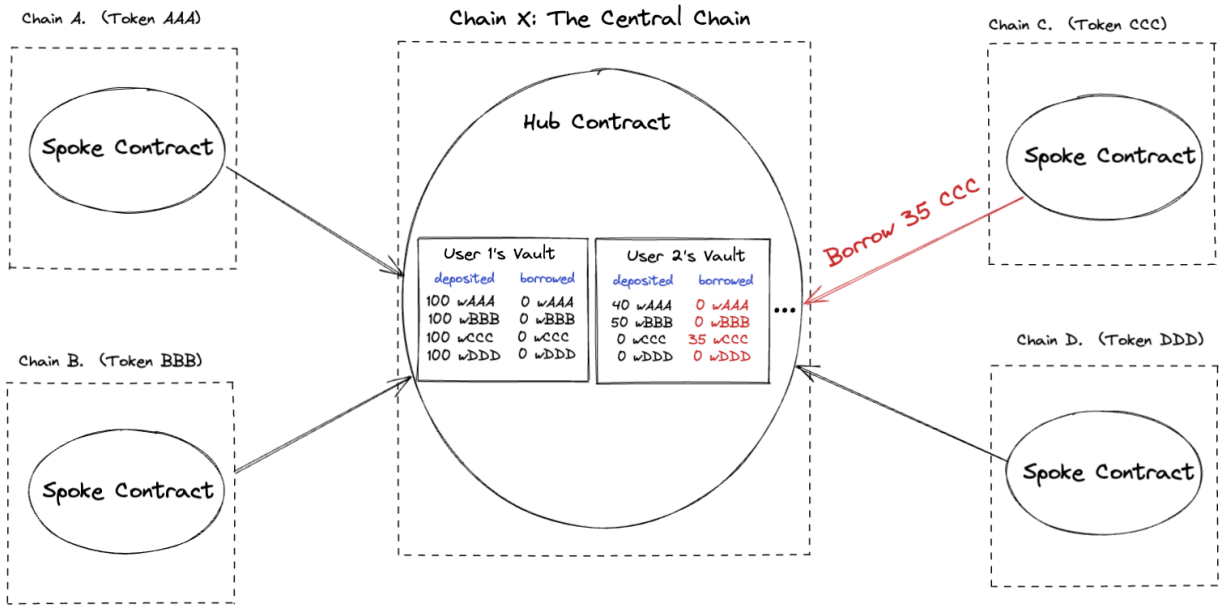
We use two different min collateralization rates for each asset —one for the deposit side and one for the borrow side. These ratios are both greater than 1. This is to guard against both downward movements in the price of collateral assets and upward movements in the price of borrowed assets, and is in line with Euler's two-sided risk-adjusted borrowing capacity feature. If one wishes to default to a simpler protocol without separate deposit and borrow collateralization ratios, they can set the deposit ratio to 1 and vary the borrow collateralization ratio. The actual check we do is (collat. is short for collateralization) :

$$40 \cdot \frac{\text{price of wAAA}}{\text{deposit collat. ratio of AAA}} + 50 \cdot \frac{\text{price of wBBB}}{\text{deposit collat. ratio of BBB}} \geq 35 \cdot (\text{price of wCCC}) \cdot (\text{borrow collat. ratio of CCC})$$

The calculation we conduct is actually a bit more complicated because we use slightly different prices for collateral and debt assets—see the Pyth Oracle section for more detail—but the main idea about using multiple collateralization ratios still stands.

2) does the protocol have enough total supply of the CCC token to lend out these 35 CCC? The answer to this is yes because User 1 has 100 wCCC in their vault, so the protocol currently has 100 wCCC available to lend out.

If both checks are valid, we initiates a TokenBridge transfer of 35 CCC to User 2's address on Chain C, and updates the Hub state.



Suppose next that user 2 wishes to withdraw 10 AAA. They would call 'withdrawCollateral' on the spoke chain. This transaction creates a Wormhole Core Bridge VAA. Then, a relayer relays this VAA to the Hub Contract on chain X by calling 'completeWithdraw'.

On the Hub, the completeWithdraw function first checks to see if this withdraw is valid. Three checks need to be satisfied:

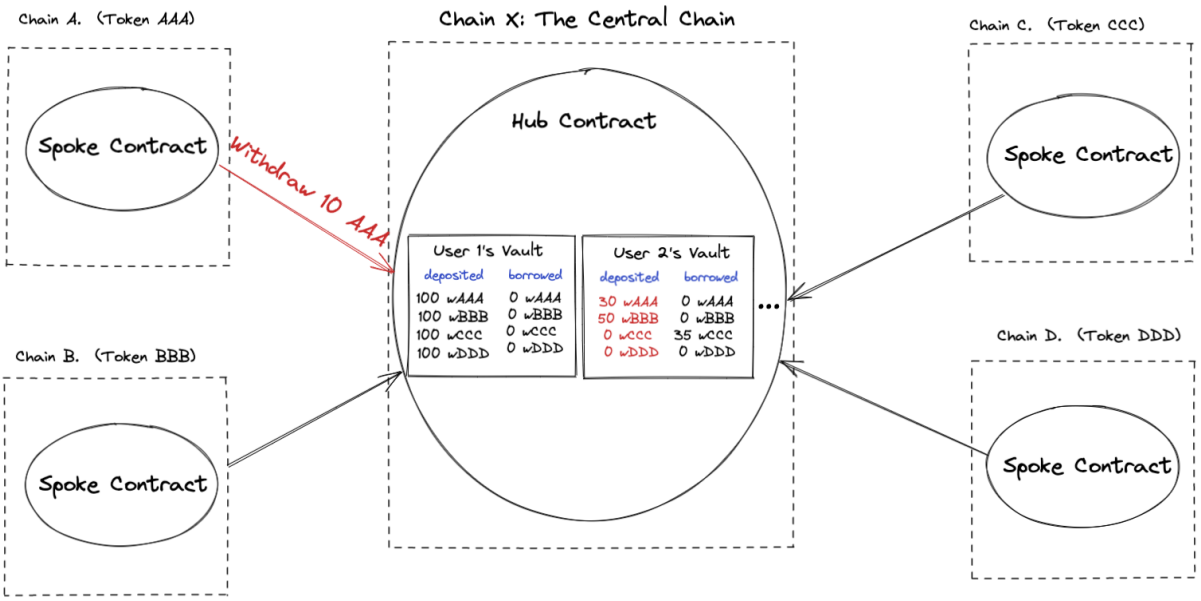
1) does the user have enough value in their vault for this withdraw to be valid (we use Pyth prices of the assets in the user's vault to check this. In fact we use slightly different prices for borrowed vs deposited assets; see details in the Pyth Oracle Integration section).

$$(40-10)\cdot\frac{\text{price of wAAA}}{\text{deposit collat. ratio of AAA}}+50\cdot\frac{\text{price of wBBB}}{\text{deposit collat. ratio of BBB}}\geq 35\cdot(\text{price of wCCC})\cdot(\text{borrow collat. ratio of CCC})$$

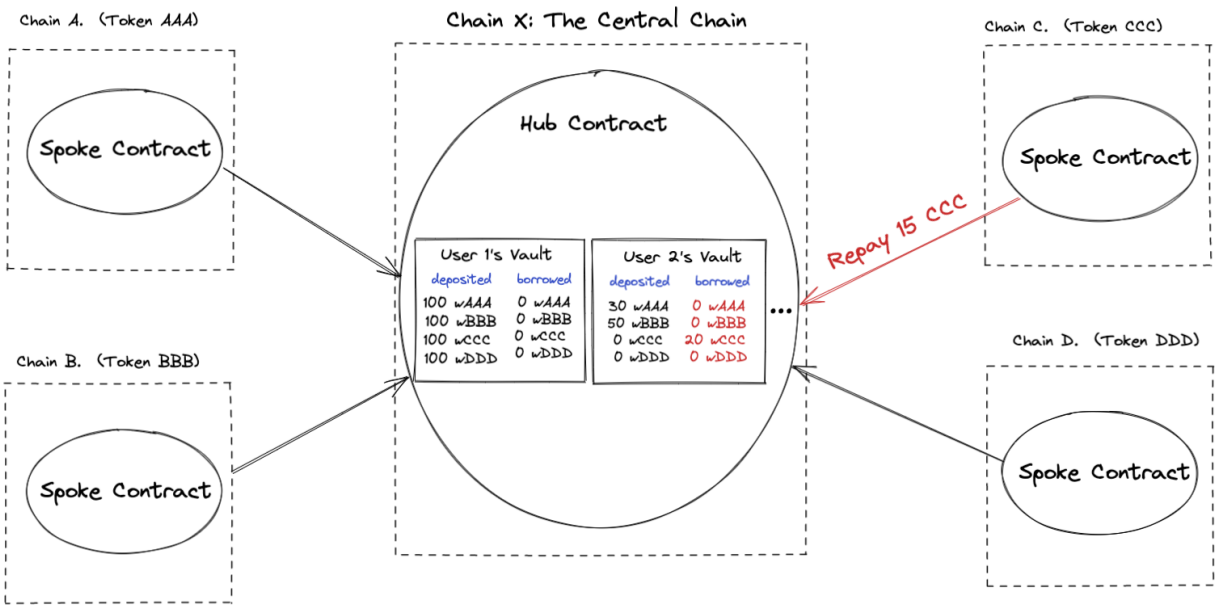
2) does the user have enough of the token in their vault for the withdraw to be valid. Specifically, does the user have ≥ 10 wAAA in their vault. The answer to this is yes since User 2 has 40 wAAA in their vault.

3) does the protocol have enough total supply of the AAA token to execute this withdraw? The answer to this is yes because the total supply of wAAA at the moment is 100 + 40 = 140 wAAA. One might ask why this check is necessary if check 2 exists; the answer is that it is possible that the protocol lent out these tokens to a different user (say, if before this, a user 3 borrowed all 140 wAAA having deposited a large amount of a different token as collateral).

If all three checks hold, then the withdraw is allowed and completed. A token bridge transfer of 10 AAA to user 2's address on chain A is performed.



If user 2 later wishes to repay 15 CCC, they would call 'repay' on the spoke. This call creates a Wormhole Token Bridge VAA with 15 wormhole-wrapped CCC (wCCC). Then, a relayer relays this VAA to the Hub Contract on chain X by calling 'completeRepay', which retrieves 15 wCCC from the TokenBridge contract on Chain X (giving the tokens to the Hub contract), and updates the Hub state



If user 2's vault, now with deposits of 30 wAAA and 50 wBBB and a loan of 20 wCCC goes underwater because of price movements (for example, the price of CCC goes up a lot), then any user can liquidate this vault.

Suppose that user 3 (on Chain X, the Hub Chain, because liquidations are only allowed on the hub chain) attempts to liquidate the vault by repaying 10 wCCC and receiving 15 wAAA and 25 wBBB. They would call liquidation from the hub contract, and the contract would allow the liquidation if the following checks hold:

1) The vault is underwater. Specifically,

$$30 \cdot \frac{\text{price of wAAA}}{\text{deposit collat. ratio of AAA}} + 50 \cdot \frac{\text{price of wBBB}}{\text{deposit collat. ratio of BBB}} < 20 \cdot (\text{price of wCCC}) \cdot (\text{borrow collat. ratio of CCC})$$

2) The liquidator is receiving nonnegative value, i.e. the received notional is ≥ the repaid notional (this isn't strictly necessary for the health of the protocol but serves as a 'safety check' to safeguard liquidators from making bad liquidations).

$$15 \cdot \text{price of wAAA} + 25 \cdot \text{price of wBBB} \geq 10 \cdot \text{price of wCCC}$$

3) a check to make sure that the received notional is no more than some max liquidation bonus (set by the owner/governance) times the repaid notional:

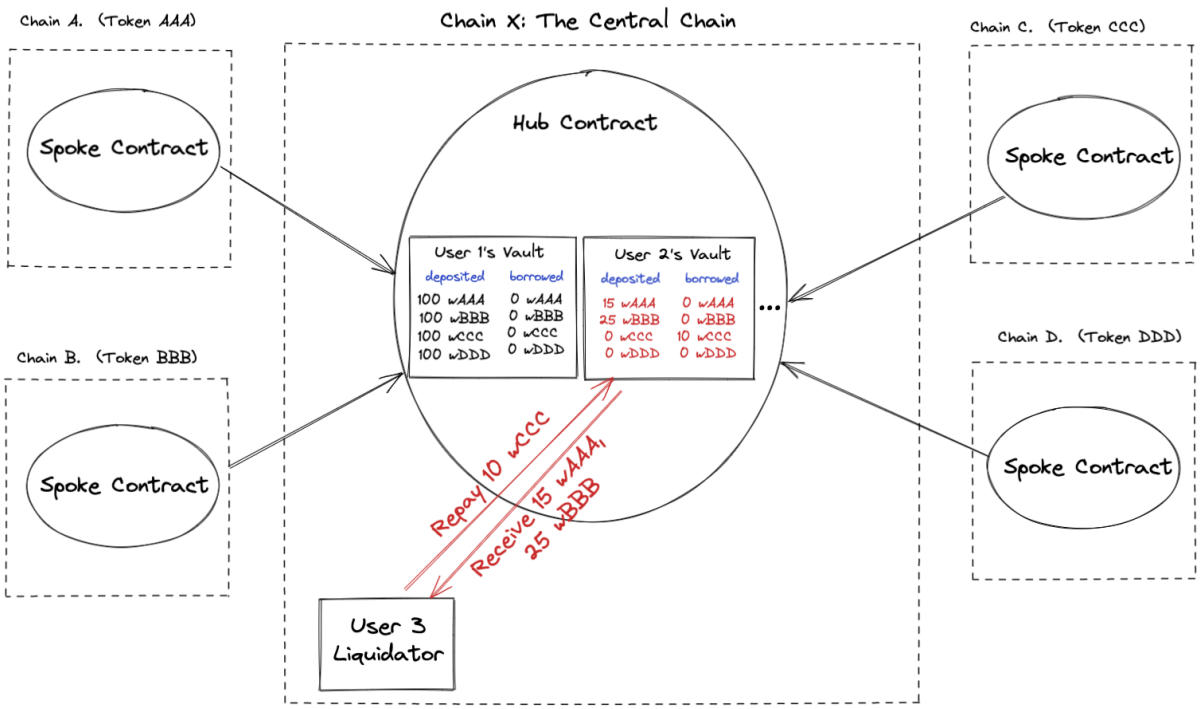
$$15 \cdot \text{price of wAAA} + 25 \cdot \text{price of wBBB} \leq 10 \cdot \text{price of wCCC} \cdot \text{max liquidation bonus}$$

4) Each amount of token being repaid is less than or equal to the outstanding amount of that token owed by the vault being liquidated. Specifically, 10 ≤ 20 We don't allow people to repay more than a vault actually owes.

5) Each amount of token being received is less than or equal to the outstanding amount of the token deposited by the vault being liquidated. Specifically, 15 ≤ 30, and 25 ≤ 50. We don't allow people to receive more than a vault actually has.

6) Each amount of token being received is less than or equal to the current supply of the token overall. Specifically, 15 ≤ 130, and 25 ≤ 150. We don't allow people to receive more than the hub actually has (this is not even possible to allow).

If these six checks pass, then the liquidation is successful, and the repay tokens are transferred to the contract while the receipt tokens are transferred to the liquidator.



3. Indexing and Interest Rate Calculations

We use a similar accounting module as the aforementioned point-to-point repo to keep track of owed interest. This accounting draws on the fact that the amount of interest associated with a deposit or a withdrawal is a function not only of that amount but also of the state of the protocol throughout the deposit or loan. Intuitively, if deposited amounts are high relative to borrowed amounts, then a deposit should not earn much interest, and a borrow should not be charged much interest. On the converse, if borrowed amounts are high (a.k.a. utilization rate is high), then we want to hike up the deposit and borrow APYs to encourage more deposits relative to borrows. This philosophy is at the heart of the variable interest rate model.

Our interest rate model is a simpler version of the model used by Aave. Our model is simpler because we don't use a piecewise function and instead use a single linear function. This model can be replaced with a piecewise linear function, or something more complicated (e.g. Euler uses a control-theory-set rate). The interest owed by a borrower can be written out as

$$t \cdot (a \cdot \frac{N_{borrowed}}{N_{deposited}} + b)$$

where t is the amount of time that has elapsed between updates to state, a is the rate coefficient in the interest rate linear model, b is the rate intercept, and $N_{borrowed}$ and $N_{deposited}$ are the total normalized amounts (see below) borrowed and deposited of the asset, respectively. a and b can be set by the protocol controller or governance, and in general the whole model can be finetuned or altered by governance.

The interest paid into the protocol by borrowers is distributed to two sets of parties. A part of the interest paid is siphoned off for the protocol reserves, to serve as a layer of defense against bad debt accumulation and protocol insolvency. The rest of it (the vast majority) is distributed amongst depositors pro rata to their computed interest factors. Thus, once the reserve portion $r \in [0, 1]$ is partitioned off for the reserves, the remainder

$$(1 - r) \cdot t \cdot (a \cdot \frac{N_{borrowed}}{N_{deposited}} + b)$$

is distributed amongst depositors of that asset. To reiterate, the amount claimable by a depositor is not a function of the amount deposited alone, but rather of the normalized amounts. The normalized amount deposited is computed according to a monotonically non-decreasing interest accrual index that serves as a piecewise linear measurement of how much interest has been accrued.

For illustration, consider the following example. At time 0 the interest accrual index is 1 and a depositor D_1 deposits X tokens; at time t_1 another depositor D_2 deposits Y tokens. Suppose for simplicity that at time 0 a borrower B_0 withdraws all of the X tokens deposited by D_1. Then, from time 0 through t_1, D_1 collects all but r of the paid out interest by B_1, and from time t_1 through $2 \cdot t_1$, D_1 and D_2 split (all but r of) the interest paid out by B_1 pro rata according to X and Y. Do note that the interest paid out by B_1 between t_1 and $2 \cdot t_1$ will be lower than that paid out between 0 and t_1, since the utilization rate is lower.

4. Pyth Oracle Integration

Our protocol leverages the Pyth price oracle model for providing high-fidelity, precise prices for different price feeds. More details on how to integrate Pyth can be found here. A list of available price feeds on different chains is accessible here.

One feature of Pyth is its provision of a confidence interval in addition to the aggregate price point estimate. These two parameters parameterize a distribution for the price, with the aggregate price serving as the mean of a Laplace distribution with standard deviation equal to the confidence. We leverage the fact that ~95% of the probability mass is contained within 4.24 standard deviations of the mean and use conservative prices in the protocol’s favor when doing most collateral and debt notional evaluations. Thus, for collateral assets, we use a ”lower bound” of the price by subtracting 4.24 * confidence from the aggregate price, and for debt assets, we use an “upper bound” of the price by adding 4.24 * confidence to the aggregate price.

Referring to the aforementioned example with assets AAA, BBB, and CCC, the revised computation is

$$\begin{aligned} 40 \cdot \frac{\text{price of wAAA} - 4.24 \cdot \text{confidence of wAAA}}{\text{deposit collat. ratio of AAA}} + 50 \cdot \frac{\text{price of wBBB} - 4.24 \cdot \text{confidence of wBBB}}{\text{deposit collat. ratio of BBB}} \\ \geq 35 \cdot (\text{price of wCCC} + 4.24 \cdot \text{confidence of wCCC}) \cdot (\text{borrow collat. ratio of CCC}) \end{aligned}$$

Using these adjusted prices, we undervalue collateral and overvalue debt, so as to be conservative and minimize the chance of the protocol taking on bad debt.

We use “lower bound” prices for evaluating total notional of collateral assets and “upper bound” prices for evaluating total notional of debt assets in `withdraw` and `borrow`, but we do not do this for `liquidation`. We can afford to be cautious with letting users borrow and withdraw assets, and this only serves to add another buffer to keep their health ratio high. While it does limit user experience a bit, it will not penalize users falsely and create erroneous losses for them. However, while liquidations taking place at these adjusted values would keep the protocol extra safe, it would also excessively punish users. Vaults that are healthy at the aggregate prices could be deemed underwater when the value of collateral is reduced and the value of debt is increased, and the result would be earlier, more lucrative liquidation opportunities for liquidation bots. That would mean losses for vaults that may not actually be underwater, since the aggregate prices are the “most likely” values for the prices of assets.

To put it another way, the price values adjusted by confidence should be used to prevent potentially risky (for the protocol) actions; they should not be used to take aggressive action against vaults that according to the aggregate price point estimates are healthy. This design choice is supported by the explanation in the Pyth docs.

To install the pythnetwork requirements for testing and deployment, you should run `npm install @pythnetwork/pyth-sdk-solidity`.

5. Wormhole Solution Details

Wormhole provides the cross-chain messaging necessary to make this protocol function. With the exception of liquidation and registerSpoke, each of the operations discussed in the workflow section uses Wormhole messaging to ensure cross-chain fulfillment of the operation in question. deposit and repay in particular utilize Wormhole’s Payload 3 feature, which enables a user to send a cross-chain message to a contract on another chain with wrapped tokens included via the token bridge. This empowers a contract to send a token transfer with instructions on what the receiving contract should do upon receipt of those tokens; in the case of deposit and repay, those instructions entail updating state on the hub to reflect changes to the token balances of the vault and the protocol.

To create and send a Payload 3 message, the spoke contract calls to approve transfer of the ERC20 token from the user’s address on source chain to the token bridge contract address on source chain. Then, it calls the transferTokenswithPayload function of the tokenBridge contract specifying the hub chain ID, the address of the hub contract, and the arbitrary payload bytes that represent the instructions for the hub contract upon receipt of the tokens. In the case of both deposit and repay, those arbitrary payload bytes constitute a flag for the message type and the address and amount of the asset transferred to the hub contract. Calling transferTokenswithPayload initiates a token transfer to the hub contract and then calls logTransferWithPayload, which publishes a Wormhole message with the token transfer details and the payload.

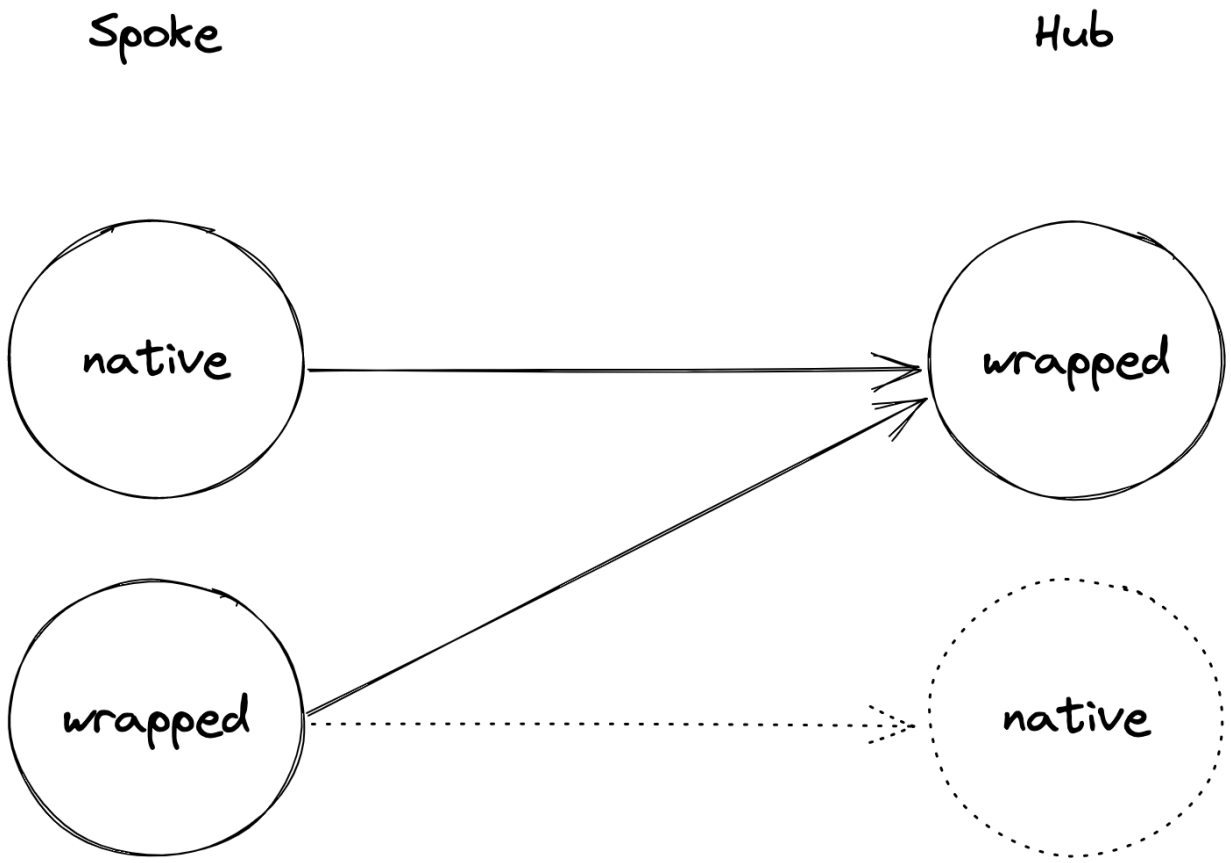
All other operations in the spoke contract use general Wormhole messages without going through the token bridge. This is because borrow and withdraw do not entail upfront token transfers to process the changes on hub state. For these operations, a Wormhole message with payload consisting of a flag for the message type and the address and amount of the asset intended to be borrowed/withdrawn is created and sent via the publishMessage function of the wormhole core contract. Then, the message is processed and parsed on the hub side, and if state is successfully updated, a simple token transfer from the hub contract back to the user address on the source chain is performed via calls to approve the token bridge and the transferTokens function of the tokenBridge contract.

registerAsset is initiated on the hub side, and the Wormhole component is very similar to those of borrow and withdraw. A Wormhole core message is constructed on the hub side and sent via the publishMessage function of the wormhole core contract.

To handle transfer of native gas tokens to the hub (e.g. transfer native ETH from Ethereum spoke to BNB hub), we support a modified version of the depositCollateral function on the spoke called depositCollateralNative. This function creates a similar Payload 3 payload with the asset address corresponding to the address of wETH on Ethereum but calls wrapAndTransferETHWithPayload instead of transferTokensWithPayload as is done normally in depositCollateral. Thus, the function first wraps the native gas token into its wrapped version via the token bridge, and then conducts the cross-chain transfer of that wrapped token. We make a similar modification to support repayNative.

Currently, we only support wrapped assets being held by the hub; as mentioned previously, this only affects the native gas token of the hub chain, as all other native gas assets would anyways need to be wrapped. There are two big reasons we require all native gas tokens on the hub contract to be wrapped.

For one, as referenced in the prior paragraph, wrapping and unwrapping gas tokens is a special case of the transfer functions of the token bridge, with an extra step needed to wrap the native token into a wrapped form on the same chain before handling the cross-chain transfer via token bridge. Using the wrapped version of the gas token significantly lessens the complexity of handling transfers. As the graphic below shows, we currently support native-to-wrapped conversion and wrapped-to-wrapped conversion from spoke to hub. If we were to support wrapped-to-native from the hub to the spoke, then we would need to know for which asset to call completeTransferAndUnwrapETHWithPayload (i.e. native gas token) as opposed to the usual completeTransferWithPayload. This would entail storing a mapping in the application state that would alert us to the native token being the one that is being unwrapped. For the effective functioning of the hub contract, this is not necessary, since we just need proper accounting of the tokens; thus, we can simply use the wrapped token version on the hub.



Moreover, wrapping the native gas token reduces the registerAsset complexity, for with wrapped assets, we only need to keep a map of address to AssetInfo. If we were to allow the native gas token to be deposited, we would need to configure support for this token; this could be as easy as keeping a separate unused address for mappings for the native gas token, but just using the wrapped version saves us the trouble of having to do this.

A more relevant question is whether to unwrap wrapped gas tokens to their native format when they are transferred back to their chain. This is more valuable from the UX perspective than unwrapping gas tokens on the hub, because the end user presumably wants to receive the native gas token as opposed to a wrapped version thereof. Currently, we only return the wrapped version for simplicity of the reference example, but we could extend this protocol to support unwrapping to native token. To do this, we could keep a mapping storing wrapped token address to native token on different chains, and call completeTransferAndUnwrapETH as opposed to the normal completeTransfer.

Relaying a Wormhole message between the source chain and the destination chain can be done by the user themselves, but often a user is engaging with a Wormhole application precisely because they do not have connectivity to the destination chain. Thus, Wormhole introduces the concept of relayers. Relayers are separate entities that handle the off-chain element of picking up the Wormhole message and broadcasting it to the intended contract on the intended chain. In the case of this protocol, a relayer could pick up a message and call the appropriate complete function on the destination hub or spoke contract. Wormhole supports application-specific

(specialized) relayers, and generic relayer development is on the roadmap. We do not develop a relayer ourselves, but anyone can permissionlessly develop and deploy an application-specific relayer in coordination with deploying a set of contracts.

6. Liquidation Design Choices

Where to allow liquidations from

We needed to decide whether to allow initiation of liquidations on spokes or to restrict them to the hub altogether. The main perk of allowing liquidations on spokes is that liquidation bots no longer need to be configured to work on any particular chain, which may open up the protocol to a greater number and diversity of liquidation bots. However, the downsides of opening up liquidations to all chains are numerous:

1. Asynchronicity of price updates—a liquidation could be valid at one time and invalid shortly after because of fluctuating price movements.
2. Reversions—in the race to liquidate, many users could be caught in the middle and have to revert the repayment leg of their liquidation attempt. This is true for repayments and borrows, but the race atmosphere is less apparent than in the liquidation case, where UX could suffer quite significantly.
3. No batch VAAs currently —> difficulty transferring multiple token types via Payload3—though we could wait to build out with batch VAAs to enable sending multiple tokens at the same time in the repayment leg of a liquidation, we didn't think this was worth the trouble given the other challenges.

What percentage of a vault to make liquidatable

Another decision point is how much of a vault's debt to allow a liquidator to pay back. Too little, and many liquidations will be needed to bring the vault back to a suitable health ratio. Too much, and a vault could lose a ton of money in one swoop. Examples of what other lending protocols do:

- "In a liquidation, up to 50% of a borrower's debt is repaid and that value + liquidation fee is taken from the collateral available, so after a liquidation that amount liquidated from your debt is repaid."—Aave
- "When your account is open to being liquidated, our Third Party Liquidators will repay **20%** of your loans by selling the equivalent amount in Collateral."—Solend
- "On Euler, we therefore use a dynamic close factor to try to `soft liquidate' borrowers. Specifically, we allow liquidators to repay up to the amount needed to bring a violator back out of violation (plus an additional safety factor)."—Euler

We do not choose a particular value, but allow the owner of the contract to determine what this value is at initialization. Optionally setting this value at later times than initialization can be added to the functionality of the contract. In the long run, we'd recommend moving toward something like what Euler does to avoid making unnecessarily large liquidations.