Rho Protocol Spec

Version 1.0

Max Wolff
maxcwolff@gmail.com

Summary

Rho is a simple specification for an automated market maker for on-chain interest rate swaps. It allows users to open swaps directly with a smart contract protocol at rates determined by an algorithm. Opening interest rate swaps allow users to speculate or hedge risk on some underlying interest rate.

The protocol pools tokens from liquidity providers to supply the collateral for payouts to the swaps it writes. In return for taking the risk that the protocol loses money on the swaps it writes, the protocol charges extra fees and distributes those fees to the liquidity providers.

This spec is written with Compound interest rates in mind.

Key Terms

Swap Type	There are two types of interest rate swaps. If a user opens a 'payFixed' swap, they pay a fixed rate in return for the protocol paying them a floating rate derived from a Compound market (they effectively go long on the floating rate). If a user opens a 'receiveFixed' swap, they pay the Compound floating rate in return for receiving a fixed rate (effectively shorting the floating rate).
activeCollateral	The amount of collateral that the protocol has locked for active swaps. This sum is how much the protocol would have to pay out for all of its outstanding swaps if the market went against the protocol in each swap. Eg: for a 'payFixed' swap, it's the amount the protocol has to pay if the float rate is at its max for the whole swap.
supplyIndex	The protocol uses Compound's indexing system for accounting for profits and losses for liquidity providers. Unlike Compound v2, liquidity accounting is stored in 'accounts' instead of in token balances.

totalLiquidity	The total cash in the protocol (excluding user collateral). The sum of total liquidity provider deposits, plus their accrued and profits.
notionalReceivingFloat, **notionalPayingFloat**	Sum of notional amounts of the floating legs of all swaps for each 1type. This is the notional amount that floating rates (indices) are applied to. Changes when a swap is opened or closed. Amounts compound every period.
notionalPayingFixed, **notionalReceivingFixed**	The sum of the notional amounts of the fixed leg of all swaps for each type. Inverse of the above, except that these do not compound. This is the amount that the fixed interest rates are applied to. Changes every time a swap is opened or closed.
avgFixedRatePaying, **avgFixedRateReceiving**	The fixed rate to charge or pay swappers. It is the average of all outstanding swaps for each type, weighted by their notional amount. Changes every time a swap is opened or closed.
fixedToPay, **fixedToReceive**	The aggregate amount outstanding of the fixed leg for each swap type. Decreases when time accrues.
notionalDaysPayingFixed, **notionalDaysReceivingFixed**	Used to calculate the active collateral amount for the float leg, and not in interest charges. Think of this as *maxFloatToPay / minFloatToReceive*. Represents the total quantity of float positions (combines amount and duration). Decreases by *notionalPayingFixed* & *notionalReceivingFixed* whenever time accrues, and increases when new swaps are added. No compounding is applied.
liquidityAccount	An object we keep for every liquidity depositor for accounting. We keep *amount*, *lastDepositTime*, *depositSupplyIndex*
lastCheckpointTime	The timestamp of the last time the *accrueProtocolCollateral* function was called.
benchmarkCToken	The address of the Compound cToken that we use for the floating rates. We access the cToken's borrow index to read floating rates, though hypothetically supply rates could also be used.
lastFloatIndex	The Compound interest rate index at the last time any of the core protocol functions was called.

Constants

swapDuration	All swaps are of a single fixed duration, e.g. 90 days. Swaps can be closed later than *swapDuration*, but not before. If they are closed afterward, both the float and fixed rates will be applied for the entire duration.
minPayoutRate, **maxPayoutRate**	Bounds on the maximum and minimum floating interest rates that Rho can expect. Used to calculate collateral requirements. Rho will not pay a floating rate above *maxPayoutRate*, and will collect a minimum of *minPayoutRate* from a trader paying floating.

Protocol Specification

[public] addLiquidity(uint depositAmount)

User supplies assets from her own address to the protocol. In doing so, she agrees to take the aggregate position of the protocol, and collects a portion of fees.

- Accrue profits, updating *supplyIndex* so we know how much the provider has earned
    - $accrueProtocolCashflow()$
- $lastCheckpointTime = now()$
- If the user previously deposited liquidity:
    - Accrue their profits, and update their liquidity deposit amount:
    - $accrued = account.amount * supplyIndex / account.depositSupplyIndex$
- $newAccountLiquidity = accrued + depositAmount$
- Update user's liquidity account
    - Set amount to $newAccountLiquidity$
    - Set *lastDepositTime* to now
    - Set *depositSupplyIndex* to current *supplyIndex*
- Increase *totalLiquidity* by *depositAmount*
- Set *lastFloatIndex* to equal the result of calling $benchmarkCToken_{borrowIndex}$

[public] removeLiquidity(uint withdrawAmount)

Removes liquidity from a user's account. Active collateral must exceed total liquidity after this operation, to prevent the contract from being undercollateralized. Fees based on utilization offset potential concerns of illiquidity for liquidity providers.

- Check if liquidity provider has kept their funds in the protocol for the minimum time period. This prevents them by sandwiching a swap between and *addLiqudity* and *removeLiquidity* transaction, and lowering their slippage amount.
    - Require $account_{lastDepositTime} - now() < minDepositTime$

- Accrue time, updating *supplyIndex* so we know how much the provider has earned:
    - $accrueProtocolCashflow()$
- Calculate the user's new liquidity account value:
    - $newAccountValue = account.amount * supplyIndex / account.depositSupplyIndex$
- If $withdrawAmount$ is *uint256(-1)*, set to *newAccountValue* (signals a full withdrawal)
- Update minimum collateral based on cashflows, since we have to check if this *remove* action would undercollateralize the protocol
    - $updateProtocolActiveCollateral()$
    - $lastCheckpointTime = now()$
- Require *withdrawAmount* to be less than *newAccountValue*
- Update the user's liquidity account:
    - Set the user's new account liquidity to $newAccountValue - withdrawAmount$
    - Set *lastDeposit* time to be now
    - Set *depositSupplyIndex* to be the current *supplyIndex*
- Decrease *totalLiquidity* by *withdrawAmount*
- Set *lastFloatIndex* to equal the result of calling $benchmarkCToken_{borrowIndex}$

[public] openPayFixedSwap(uint notionalAmount)

Opens a swap where the user pays the protocol-offered fixed rate and receives a Compound floating rate for $swapDuration$.

- We need to accrue profits to and update collateral to confirm the new swap does make the protocol undercollateralized. Also, fees depend on utilization.
    - *accrueProtocolCashflow()*
    - *updateProtocolActiveCollateral()*
    - $lastCheckpointTime = now()$
- Set *swapFixedRate* to *getRate('payFixed', notionalAmount)*
- Now that we know the rate, require the protocol is adequately collateralized after the swap
    - $activeCollateral + newMaxFloatToPay - newFixedToReceive < totalLiquidity$ where:
        - $newFloatLiability = notionalAmount * maxPayoutRate * swapDuration / 365$
        - $newFixedToReceive = notionalAmount * swapFixedRate * swapDuration / 365$
- Update internal accounting measures for collateral and interest rate accrual:
    - $avgFixedRateReceiving$ is the average of the existing average rate and the new swap's rate, weighted by notional amount
    - $notionalReceivingFixed \mathrel{+}= notionalAmount$
    - $fixedToReceive \mathrel{+}= newFixedToReceive$
    - $notionalDaysPayingFloat \mathrel{+}= notionalAmount * swapDuration$
- Calculate the collateral the user must post, and pull it from them

    - $userCollateral = notionalAmount * swapDuration * (swapFixedRate - minPayoutRate)$
- Save new swap object
    - Set *type = payFixed*
    - Set *notional = notionalAmount*
    - Set *swapRate = swapRate*
    - Set *owner = msg.sender*
    - Set *initIndex = current Compound float Index*
    - Set *userCollateral = userCollateral*
- Set *lastFloatIndex* to equal the result of calling $benchmarkCToken_{borrowIndex}$

[public] openReceiveFixedSwap(uint notionalAmount)

Opens a swap where the user receives the protocol-offered fixed rate and pays a Compound floating rate for $swapDuration$.

- We need to accrue profits to and update collateral to confirm the new swap does not make the protocol undercollateralized. Also, fees depend on utilization.
    - *accrueProtocolCashflow()*
    - *updateProtocolActiveCollateral()*
    - $lastCheckpointTime = now()$
- Set *swapFixedRate* to *getRate('receiveFixed', notionalAmount)*
- Now that we know the rate, require the protocol is adequately collateralized after the swap
    - $activeCollateral + newFixedToPay - newMinFloatToReceive < totalLiquidity$
      where:
        - $newFixedToPay = swapFixedRate * swapDuration * notionalAmount$
        - $newMinFloatAsset = minPayoutRate * swapDuration * notionalAmount / 365$
- Update internal accounting measures for collateral and interest rate accrual
    - $avgFixedRatePaying$ is the average of the existing average and the new swap's rate, weighted by notional amount
    - $notionalPayingFixed \mathrel{+}= notionalAmount$
    - $fixedToPay \mathrel{+}= newFixedToPay$
    - $notionalDaysReceivingFloat \mathrel{+}= notionalAmount * swapDuration$
- Calculate the collateral the user must post, and pull it from them
    - $userCollateral = notionalAmount * swapDuration * (maxPayoutRate - swapFixedRate) / 365$
- Save new swapObject
    - Set *type = receiveFixed*
    - Set *notional = notionalAmount*
    - Set *swapRate = swapRate*
    - Set *owner = msg.sender*
    - Set *initIndex =* $benchmarkCToken_{borrowIndex}$
    - Set *userCollateral = userCollateral*

- Set *lastFloatIndex* to equal the result of calling $benchmarkCToken_{borrowIndex}$

[public] closeSwap(uint orderNumber)

Closes both swap types, after it has expired.

- Accrue time:[1]
  - *accrueProtocolCashflow()*
  - *updateProtocolActiveCollateral()*
  - $lastCheckpointTime = now()$
- Fetch the swap at orderNumber
  - $swap = swaps[orderNumber]$
- Figure out how long the swap has been on for:
  - $swapLength = now() - swap.initTime$
- Require swap is not being closed prematurely:
  - I.e. require $swapLength >= swapDuration$
- Compute late days. If the swap is closed late, we'll have to undo incorrect accruals to protocol profit and collateral numbers:
  - $lateDays = swapLength - swapDuration$
- Adjust internal accounting to take the swap off the books, and apply $lateDay$ adjustments.
- *swap type is payFixed*:
  - Set *avgFixedRateReceiving*, to the new weighted average without this swap
  - $notionalReceivingFixed -= swap.notional$
  - $notionalPayingFloat -= swap.notional * benchmarkCToken_{borrowIndex} / swap.initIndex$
    - $notionalPayingFloat$ compounded during the term of the swap, it is now greater than $swap.notional$. We must remove the entire compounded amount
  - $fixedToReceive = swap.notional * swapRate * lateDays / 365$
    - We accidentally reduced the collateral
  - $notionalDaysPayingFloat += swap.notional * lateDays$
    - We decremented days paying float in *updateProtocolActiveCollateral*
- Otherwise (when *swap type is receiveFixed):*
  - Set *avgFixedRatePaying* to the new weighted average without this swap
  - $notionalPayingFixed -= swap.notional$
  - $notionalReceivingFloat -= swap.notional * benchmarkCToken_{borrowIndex} / swap.initIndex$
    - $notionalReceivingFloat$ compounded during the term of the swap, it is now greater than $swap.notional$. We must remove the entire compounded amount

[1] Some of the updates applied in the accrue functions are overwritten when we do late day calcs (notionalDays, fixedToReceive) later in the *closeSwap()*. Not refactored here for clarity, but may want to edit this in production.

    - $notionalDaysReceivingFloat \mathrel{+}= swap.notional * lateDays$
    - $fixedToPay \mathrel{+}= swap.notional * swapRate * lateDays / 365$
- Calculate the user's profit from the swap, and send it to them
    - $fixedLeg = swap.notional * swap.swapRate * swapLength / 365$
    - $floatLeg = swap.notional * (benchmarkCToken_{borrowIndex} / swap.initIndex - 1)$
    - When swap type is *payFixed*:
        - $userProfit = floatLeg - fixedLeg$
    - Otherwise when swap type is *receiveFixed*
        - $userProfit = fixedLeg - floatLeg$
    - Send user $userProfit + userCollateral$ tokens
- Delete $swap$
- Set *lastFloatIndex* to equal the result of calling $benchmarkCToken_{borrowIndex}$

[public] accrueProtocolCashflow()

Called whenever time passes. Accounts for payments to swappers (even though profit / loss is only realized upon closing a swap). We account continuously so that accrued profits can be applied to $totalLiquidity$ and enable us to make larger swaps without having to wait for current ones to close.

- $accruedDays = now() - lastCheckpointTime$
- Calculate how profit the protocol has accrued:
    - $fixedReceived = notionalReceivingFixed * avgFixedRateReceiving * accruedDays / 365$
    - $fixedPaid = notionalPayingFixed * avgFixedRatePaying * accruedDays / 365$
    - $floatPaid = notionalPayingFloat * (benchmarkCToken_{borrowIndex} / lastFloatIndex - 1)$
    - $floatReceived = notionalReceivingFloat * (benchmarkCToken_{borrowIndex} / lastFloatIndex - 1)$
    - $profitAccrued = fixedReceived + floatReceived - fixedPaid - floatPaid$
- Distribute the profits to liquidity providers by increasing (or decreasing) the supply index. Multiply index by the percent profit that each unit of liquidity earned
    - $supplyIndex \mathrel{*}= 1 + profitAccrued / totalLiquidity$
- Decrease collateral requirements, now that we've realized time and the worst case didn't happen
    - $fixedToPay \mathrel{-}= fixedPaid$
    - $fixedToReceive \mathrel{-}= fixedReceived$
- Compound notional amount earning / paying float
    - $notionalPayingFloat \mathrel{*}= benchmarkCToken_{borrowIndex} / lastFloatIndex$
    - $notionalReceivingFloat \mathrel{*}= benchmarkCToken_{borrowIndex} / lastFloatIndex$

[public] updateProtocolActiveCollateral()

Recognizes time being accrued, and decreases the amount of collateral the protocol requires to be locked up (assuming the float rate was not at its max or min).

- $accruedDays = now() - lastCheckpointTime$
- Updates notional days
    - $notionalDaysPayingFloat -= notionalReceivingFixed * accruedDays$
    - $notionalDaysReceivingFloat -= notionalPayingFixed * accruedDays$
- $minFloatToReceive = minPayoutRate * notionalDaysReceivingFloat / 365$
- $maxFloatToPay = maxPayoutRate * notionalDaysPayingFloat / 365$
- $activeCollateral = fixedToPay + maxFloatToPay - fixedToReceive - minFloatToReceive$

[public view] getRate(enum swapType, uint orderNotional)

Return the protocol's offered interest rate. See more docs on this here.

- Compute how much to move the rate factor
    - $rateFactorDelta = \frac{rateFactorSensitivity * orderNotional}{totalLiquidity}$
- Set *fee = getFee()*
- If the incoming swap is 'payFixed', increase the rate the user is paying the protocol
    - $rateFactor += rateFactorDelta$
- If the incoming swap is 'receiveFixed', decrease the rate the protocol is paying the user
    - $rateFactor -= rateFactorDelta$
    - Set *fee = -1 * fee*
- Apply the s-curve to the rateFactor
    - Return $\frac{range * rateFactor}{\sqrt{rateFactor^2 + slopeFactor}} + yOffset + fee$

[public view] getFee()

A fee scheme based on utilization %. We charge a higher fee when there is more utilization, because it means liquidity providers may not be able to withdraw.

- Return $\frac{activeCollateral * feeSensitivity}{totalLiquidity} + feeBase$ (if total liquidity == 0, return $feeBase$)

[public] updateModel(uint _yOffset, uint _slopeFactor, uint rateFactorSensitivity, uint feeBase, uint feeSensitivity)

An admin action that modifies the parameters of interest rate model. Intended to be called when the underlying cToken's interest rate has moved significantly.

- Set *yOffset* = _yOffset
- Set *slopeFactor* = _slopeFactor
- Set *rateFactorSensitivity* = _rateFactorSensitivity

- Set *feeBase* = _feeBase
- Set *feeSensitivity* = _feeSensitivity

[public] updateCollateralRequirements(uint _minPayoutRate, uint _maxPayoutRate)

An admin action that modifies the collateral requirements for Rho. This impacts both users and the protocol itself.

- Set *minPayoutRate* = _minPayoutRate
- Set *maxPayoutRate* = _maxPayoutRate

Notes

- Keep the collateral for all swaps in cTokens, as to minimize opportunity cost for traders and liquidity providers. The spec omits functions that transform cToken into underlying tokens for brevity.
    - The value of all transfers should be calculated in underlying tokens, but should occur in cTokens.
    - For example, a user's required collateral is calculated in underlying tokens, then pulled in cTokens. User collateral is stored in cTokens units. Upon closing a swap, return cTokens to the user, less their liability or in addition to their profit (in underlying tokens).
    - Liquidity providers would add liquidity in cTokens, and the protocol would store *activeCollateral* in cTokens.
- Make swaps ERC721 tokens, so that they may be traded. Omitted here for brevity, should be relatively straight forward. This allows the owner of a swap to trade it to someone else.

Appendix

- Worked Example Spreadsheet
- Interest Rate Model
- Whitepaper
- Reference Implementation