CERTIK

Audit Report

Produced by CertiK

for THORCHAIN

April 20^{th}, 2020

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and THORChain (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

As there have been numerous interactions with the Company throughout the entire duration of this audit, and as the codebase target for the audit has evolved over said duration, not all of CertiK's opinions or comments have necessarily made it into this final culmination.

ABOUT CERTIK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that project is checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and verifications on the project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor. CertiK customizes its engineering tool kits, while applying cutting-edge research on smart contracts, for each client on its project to offer a high quality a delivery. As utilizes technologies from blockchain and smart contracts, CertiK team will continue to support the project as a service provider and collaborator.

For more information, visit https://certik.org.

EXECUTIVE SUMMARY

Application Summary	
Name	THORNode
Commit SHA	423b336ea63a6db2425a1e46f67aac0f38fdc30e
Type	Automated Market Maker Liquidty Network
Platforms	Cosmos SDK v0.38

Engagement Summary	
Dates	01/21/2020 — 03/03/2020
Method	Whitebox Analysis
Consultants Engaged	6
Level of Effort	6 calendar weeks

Vulnerability Summary	
Total Addressed Issues	34

ENGAGEMENT OBJECTIVES

Verify the soundness of the implementation, while ensuring its logic meets the specifications and intentions of our client, underlying the linkage between RUNE's utility as a token within THORChain's revenue-generating business model for asynchronous inter-blockchain asset exchange — which is designed to give RUNE intrinsic value in the immediate term while serving the long term sanctity of THORNodes' vaults via a tightly coupled liquidity/security mechanism — in order to:

- provide an estimate of the overall security posture of the system;
- evaluate the difficulty of system compromise from an attacker;
- identify design-level risks to the security of the system;
- identify implementation flaws that illustrate systemic and extrinsic risks;
- provide recommendations for best practices that could improve THORChains' security posture;

- document architectural risks to the system in the form of a threat model and data-flow analysis of the prioritized system components;
- provide a reference architecture that the community may use to evaluate the coverage of the security assessment, and to begin building a baseline of security relevant settings and considerations for the system.

AUDIT METHODOLOGY

The key to building and maintaining highly secure and reliable systems is simplicity — a good system will have nothing to take away, rather than nothing to add. Each component requires securing, updating and debugging in perpetuity, which is an order of magnitude more complicated than building it in the first place. There's a common saying that if you engineer the most clever system you can imagine, you're by definition unqualified to maintain it. Like open source projects which reject code not because it's bad, but because it may not be worth maintaining, the CertiK team carefully examined each piece of technology in the THORChain stack while weighing its benefits against the extra complexity it introduces.

We began by studying THORChain's most recent white papers in order to synthesize its full scope of features, validate their correctness from an economic perspective by reasoning mathematically about their underlying theoretical clauses, and qualify — according to the functional categories listed below — the model's robustness against market manipulations, front-running, and other potential attack vectors plaguing the automated market marker design space that THORChain aims to successfully disrupt:

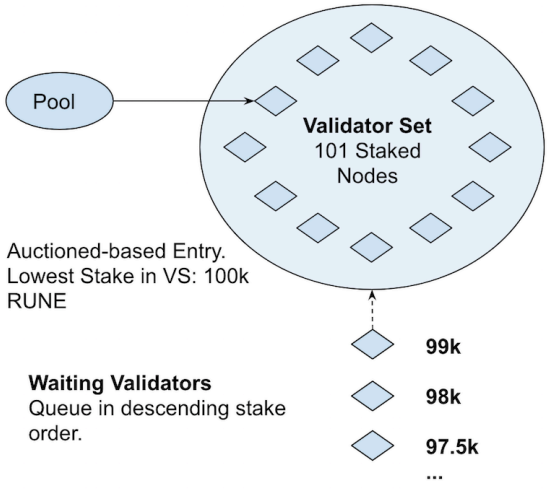
- **Prevention** - properly hardening systems with guardrails (such as THOR's *Ragnarok*) to prevent incidents from happening in the first place, just like sturdy walls and secure locks are the best defense against burglary;
- **Detection** - like with a good home alarm system detecting an incident is the next best thing after preventing it, especially in areas performing validation, decoding, type conversion, or where assumptions are being made;
- **Response** - since `error` is only a type in Go, when it is returned it does not change a program's execution flow on it's own like a `panic` statement would, so properly responding to security incidents if/when they arise through system rollback/lockdown is crucial;
- **Monitoring** - implementation of a perceptive transaction lifecycle monitoring cache to encourage prevention and detection as described above.

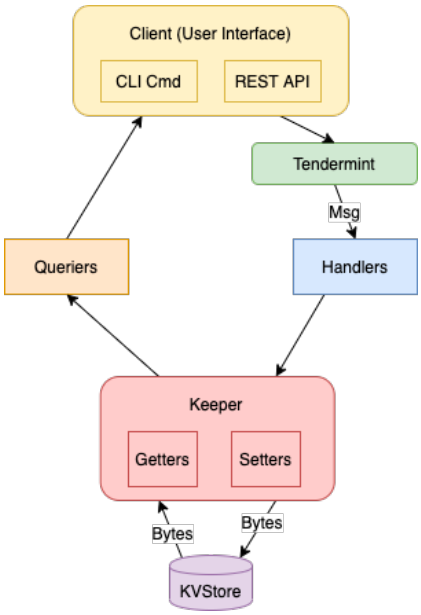
CertiK found THORChain's theoretical model, as well as its `master`-branch Cosmos SDK implementation, to be well-designed and executed cleanly, demonstrating a good command over relevant best practices. While CertiK cannot comment on the final mainnet performance of our clients' end-products, the modeling and mathematical reasoning were determined to be sound overall.

SYSTEM OVERVIEW

Tendermint enables the network to handle 1/3 malicious nodes and still maintain byzantine fault tolerance. This is because validators are required to bond (lock up RUNE, the native currency of THORChain and can have their stake slashed for bad behavior (for example proposing bad blocks). The validators take care of messaging (observation, ordering, and transport) and authentication replicating state on external blockchains via:

- *A* **Client** —> validator runs this for verifying consensus transcripts;
- *its* **Connections** —> driven by the Keeper to track associated state between chains;
- their **Channels** —> *Pools* act as pipes for RUNE to facilitate the flow of assets between vaults.





ARCHITECTURE

NETWORKING LAYER

This layer, handled by the Tendermint library, propagates transactions that happen on one localized state machine across all other state machines (nodes) in the network to be processed in the consensus layer.

CONSENSUS LAYER

This layer comprises the algorithm which is responsible for ensuring that state stored on every state machine is the same after a transaction happens (i.e., machines can't fake transactions that never existed).

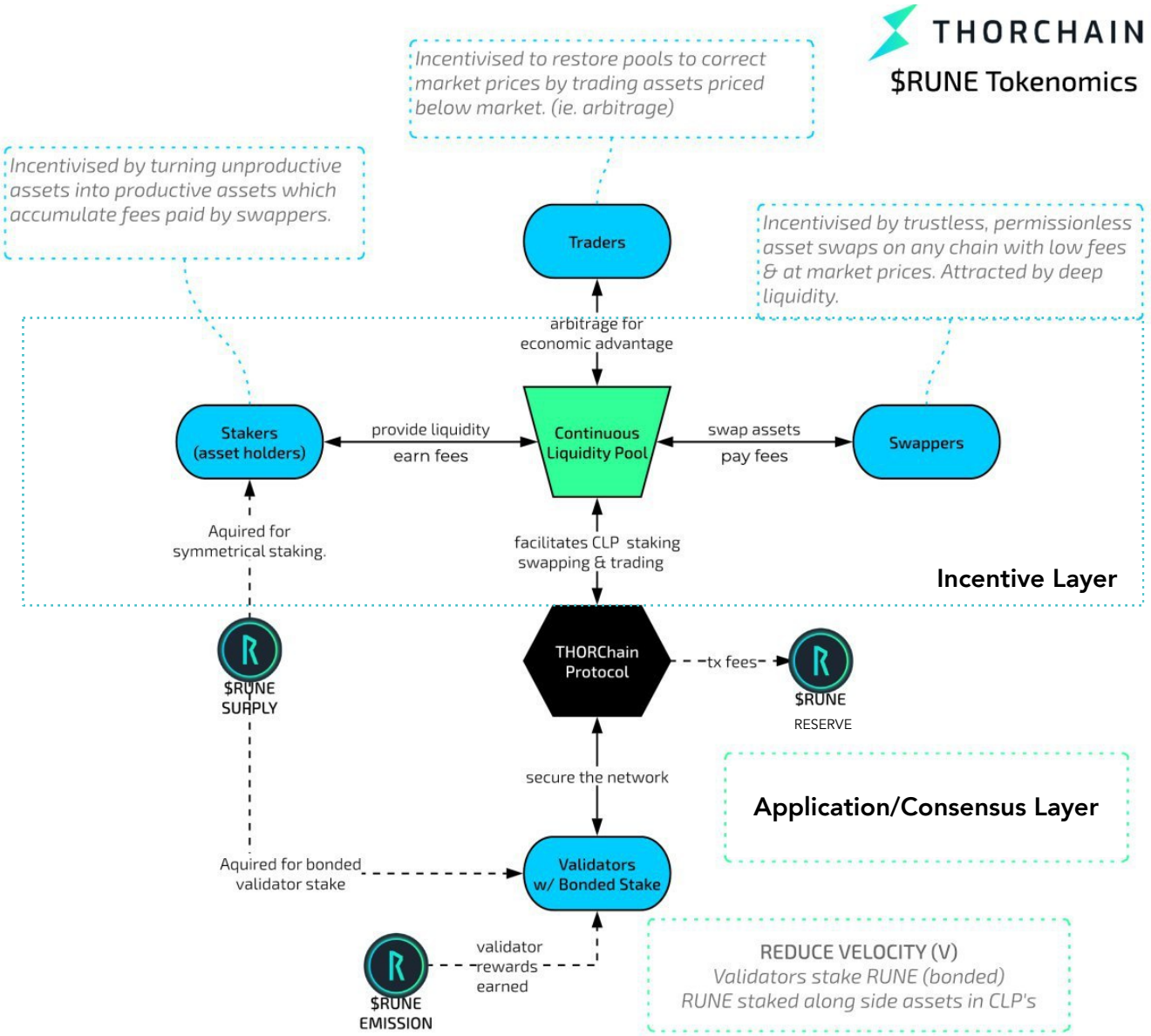
INCENTIVE LAYER

This layer drives consensus, and is liable to suffer from Censorship attacks and transaction withholding (resulting in reduction of perceived uptime, reduction of rewards from transaction feeds).

APPLICATION LAYER

Together the consensus and incentive layer govern this layer, which is responsible for defining the possible state transitions and updating the state of a state machine after a transaction occurs.

BUSINESS LOGIC EVALUATION



WORKFLOW ANALYSIS

We stepped line-by-line through the remaining files, picturing the entire application layer's workflow through the lens of hypothetical scenario-based attacks, while heeding attention toward community-established Cosmos SDK-based best practices. General software best practices around the Go programming language were taken into consideration too.

From a completeness perspective, our expectations were met in seeing a handler for every message type registered, and a test file for every handler. In addition, every chain has an accommodating struct for tracking transactions on it regardless of the nature of the native data structures of that chain. Instrumenting the transaction lifecycle to collect as many useful metrics and metadata as possible is invaluable for debugging, incident response and performance analysis, and generating on-call alerts for when a production system breaks and or throws one-time events that need fixing. From the perspectives of consistency and compatibility, events are fired related to all "movement" within the logical machinery of THORNode. If someone were to replay the events in order, they would arrive at the exact state of THORChain at that historical reference point. Versioning is consistent among all major logical components, and microservices-style wrappers are employed to ensure the provision of backwards-compatible logic. They serve as connectors for versioned (currently all *v1*) sub-components of system-level components such as:

`tx_out_store.go`, `validator_manager.go`, and `versioned_vault_manager.go`.

Safely assuming their lack of mission-critical code, we paid less attention to the */x/client* and */x/query* directories. The default genesis state contains no Pools, NodeAccounts, TxOuts, PoolStakers, StakerPools, Events, Vaults, or ObservedTxVoters. Valid application state throughout the following lifecycle requires the following invariants are satisfied:

- All staking pools have a specified RuneAddress; all pool units of each staking pool have associated with them a valid asset type and non-empty request details.
- All observed Tx votes and OutTx's have a valid TxID; each Tx's BlockHeight and ObservedPubKey are valid;
- Each NodeAccount (Validator) has a non-empty NodeAddress and a non-empty BondAddress.
- Each Vault (primary, secondary, or Yggdrasil) has a non-empty PubKey.

We included some of the major logic flows in the application layer below. The flows are separated according to the different roles of the system:

- Protocol: The protocol mechanism itself as a special type of role.
- Bonder/Validator/Observer: Supporter of the THORChain protocol infrastructure. Observer of transactions.
- Staker: User who provides liquidity to the pool in exchange of liquidity fee share.
- Swapper: User who wants to do asset swap through the THORChain protocol.

PROTOCOL

GENESIS

- An initial set of validators will be set by the `InitChainer` genesis.
- A new validator manager is created (in BeginBlock), triggering the one-off setupValidatorNodes procedure to set up a fixed number of active validators.
- A `DesireValidatorSet` amount of validators with the highest RUNE balances bonded will be the active validator set, and this is recalibrated every `rotatePerBlockHeight` blocks.
- The standby set off validators who are not the top `DesireValidatorSet` are still bonded to the reserve.
- The validator manager creates a new vault manager as well as a TxOutStorage for managing all outgoing transactions.
- An Asgard vault (a TSS vault for asset custody) is generated (in EndBlock) for the new active validator set.
  - Each time a new active node set is generated, there comes an associated Asgard vault. But active node set may generate multiple TSS Keygen and multiple Asgard vaults.

VERSIONING

- THORChain attempts to retrieve the lowest version of the active validators as the main running version for each block. Nodes run backwards-compatible node software, but cannot run software that is lower in version than what 2/3 of the nodes in the active validator set are currently running.
- The version handler allows a node to select the version it wants to run.
- The version follows semantic versioning. The backward-compatible feature of semantic versioning is used for finding new nodes that are ready to be included in the next active node set.
- Each determined version has an associated constant set that determines the behavior of the chain. The protocol achieves chain evolution through the adoption of the dynamic version determination process.

BEGINBLOCK: NODE ROTATION

The BeginBlock procedure is run at the beginning of each block in THORChain. The procedure mainly churns out misbehaving active nodes and churns in new nodes to create a new active node set for transaction observation & processing.

- The protocol determines the version and parameter set to use according to the current active node set.
- Validator manager checks if the chain's Ragnarok (end of the chain) is in progress. If Ragnarok is in progress, no node rotation is needed and the procedure ends.

- If the chain is not under Ragnarok, the protocol checks if the current active nodes satisfies the minimum Byzantine Fault-Tolerance requirement. If the minimum BFT requirement is satisfied, the protocol initiates node rotation process by finding “bad” nodes and “old” nodes.
  - If the current block reaches the BadValidatorRate scan cycle, the protocol finds “bad” nodes which have the highest rate of being slashed (gaining slashpoint through misbehavior during transaction processing) and marks them.
  - If the current block reaches the OldValidatorRate scan cycle, the protocol finds “old” nodes that have been in active mode for the longest time and marks them.
- The protocol scans through all active Asgard vaults (a primary TSS vault for asset custody) to find the last time the node rotation happens.
  - Then it checks if a new `rotatePerBlockHeight` cycle has been reached for a new node rotation. If the `rotatePerBlockHeight` cycle is reached, node rotation procedure is initiated.
  - The protocol first checks if there are any retiring Asgard vaults. If there is an Asgard vault currently in retiring status, node rotation stops as retiring has higher priority.
  - Providing that there is no retiring Asgard vault, the protocol removes “bad” nodes and “old” nodes from the active node set, as well as nodes that request to leave from the active node set (often set in EndBlock as illustrated in the next section), and adds new nodes which are in ready state, as evidenced in the `nextVaultNodeAccounts` function. This completes the next validator node set.
  - The next validator node set is ensured to satisfy the safe BFT requirement.
- The new validator node set is used by the protocol’s vault manager to generate (using the public key of each member) a new TSS Keygen for the block.
  - The TSS Keygen is later used for creating a new Asgard vault (or Yggdrasil vault in the future).
- Note that members of the next validator node set haven’t had their status updated yet. The update is done in the EndBlock procedure described below.

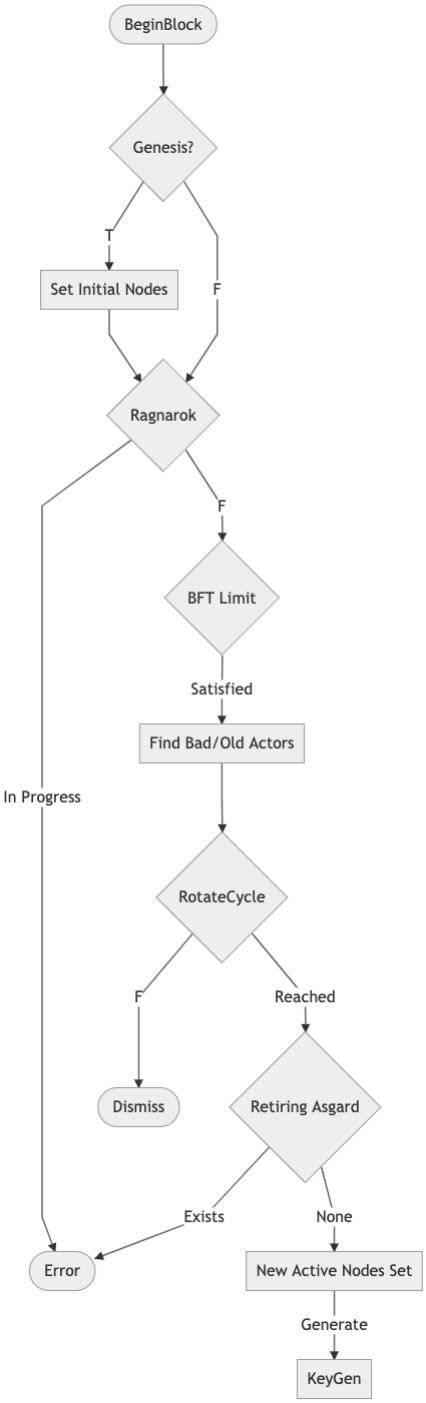
ENDBLOCK: SLASHING, REWARDING, FUNDING, AND NODE UPDATE

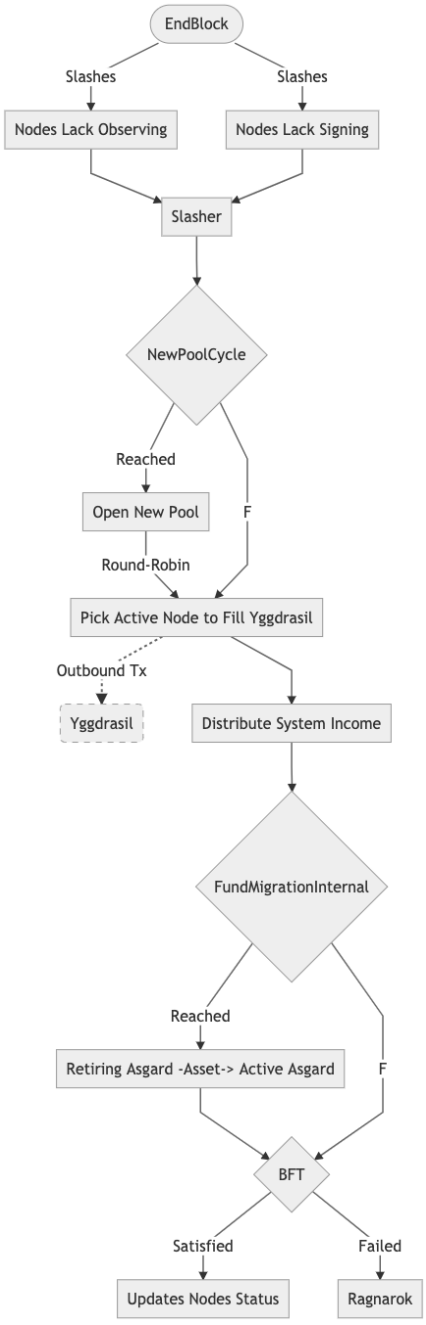
The EndBlock procedure is run at the end of each block in THORChain. The procedure mainly slashes active nodes for bad behaviors, distributes awards among bonders (supporter of the THORChain protocol infrastructure) and stakers (supporters of the liquidity pools), enables a new liquidity pool (with highest pool depth), and migrates assets between vaults.

- The protocol determines the version and parameter set to use according to the current active node set.
- The protocol slashes active nodes that haven’t successfully observe a single inbound transaction (made an observation that reaches super majority consensus) with `LackOfObservationPenalty` slashpoints, achieved through the `LackObserving` function.
- The protocol slashes active nodes that didn’t manage to sign the outbound transaction in time by (*2 * SigningTransactionPeriod*) of the slashpoints.

  - The finding of nodes that failed to sign outbound transactions in time is done through scanning all pending events that have passed the transaction signing period, and extracting the node account associated with the outbound transaction within the pending event.
  - For the missed transaction, the protocol removes it from the original outbound transaction queue, and resets its vault to be a current Active Asgard vault that has the least amount of that asset. The outbound transaction is then re-added to the queue to be sent by the Asgard vault.
- If the current block reaches the `NewPoolCycle`, the protocol enables a new pool that is in the `PoolBootstrap` status and is with the most sufficient amount of RUNE and the asset.
- The protocol fills up the Yggdrasil vaults with the active nodes' bonds.
  - The protocol picks an active node in a round-robin way and gets the Yggdrasil vault the active node is associated with (creates a new Yggdrasil vault if no vault exists).
  - Iterates through each pool to calculate the amount of coin (*targeting bond / 2*) of each pool the Yggdrasil vault should have.
  - For each of the assets that the Yggdrasil vault doesn't have a sufficient amount of ("sufficient amount" refers *50% of the calculated coin amount*), the protocol creates an outbound transaction of type YggdrasilFund which sends the asset coin to this Yggdrasil vault.
- The protocol calculates and distributes rewards for the bonders and the stakers. The reward pendulum is illustrated in the following section.
  - For current block, the reward is coming from the protocol reserve allocation + liquidity pool's transaction fee.
  - The ratio for distributing between bonders and stakers are determined by the pool share factor formula illustrated in the next section as well.
- The protocol initiates the vault manager's EndBlock procedure, which mainly migrates assets from retiring Asgard vaults to active Asgard vaults (as in BeginBlock if there is any retiring Asgard vault to migrate, node rotation is passed).
  - If the current block reaches the `FundMigrationInternal`, fund migration procedure is initiated by the protocol. The protocol gets all retiring Asgard vaults and all active Asgard vaults.
  - For each retiring Asgard vault with funds, the protocol checks each asset coin in the Asgard vault.
  - For each asset coin, the protocol finds the active Asgard vault with the largest amount of that coin, and migrates 20% of the coin amount each time from the retiring Asgard vault to the active Asgard vault.
  - So it takes the protocol 5 times to complete the migration of a retiring Asgard vault.
- The protocol initiates the validator manager's EndBlock procedure, which checks if the new active node set reaches the minimum BFT requirement. If the new set fails to reach the BFT limit, the protocol initiates Ragnarok. Otherwise, the protocol updates the status of the nodes and sends the removed nodes' bonds back.
  - For the case where the new active node set meets the minimum BFT requirement, related nodes' status are updated.
  - The new active node is located by checking if it is a member of any active Asgard vault.
  - If the node is not a member of any active Asgard vault, the node must not be a new active node.

- The protocol initiates the retiring node's "request to leave" procedure, which will be handled in the next block's BeginBlock (as illustrated in previous section).
- The new active node set is then to be participating in the THORChain consensus.

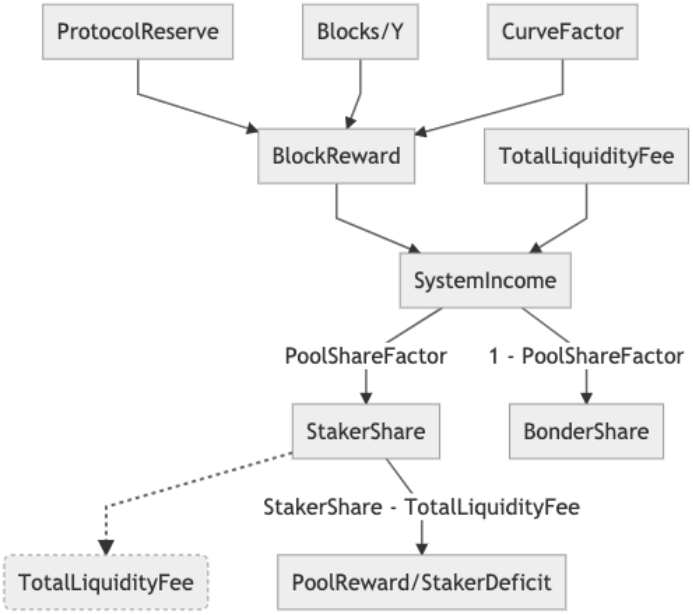




REWARD DISTRIBUTION VARIABILITY (INCENTIVE PENDULUM)

Rewards are the system income gained from the protocol's reserve + liquidity pool's fee. It is distributed between bonders (supporters of the THORChain infrastructure) and stakers (supporters of the THORChain liquidity pools).

- The protocol has 220m RUNE in reserves in order to incentivize security and liquidity. The protocol will emit 1/3rd of the remaining reserves every 12 months, targeting a split of 2/3rds : 1/3rds between THORNodes and Liquidity Providers. The total emission is variable in itself, and also split unequally between incentivizing bonded validators and liquidity providers, potentially increasing the pressure for liquidity providers to withdraw their stake as it becomes less profitable for them not to do so.
  - Increasing the rate of inflation of the staked tokens increases the incentives to bond tokens to validator stakes (directly or via pools).
  - Inflation pivots between a floor of 7% and a ceiling of 20% to drive holders to bond tokens at a desired bond state of ⅔ of the entire circulating supply. The system seeks to emit the Protocol Reserve to participants targeting a 2% annual inflation after 10 years, emitting 1/6th the remaining Protocol Reserves each year.
  - To do this, it expects an average block time of 5 seconds, giving a total of 6,311,390 blocks per year. Thus at any given block, the equation for the block reward is protocolReserve / (6 * blocksPerYear).
- There is more discrepancy between the target and the actual incentives garnered between the two stakeholder categories, the more/less they will be respectively rewarded from the protocol emission.
  - This is also reflected by the percentage discrepancy between assets staked and assets bonded is measured according to the pool share factor formula **(B-S)/(B+S)**, where **B** is total RUNE bonded by validators and **S** is total RUNE staked in liquidity pools. Reserves are transformed into bonding rewards for validators and liquidity providers in the function `UpdateVaultData()`.



RAGNARÖK

Ragnarök is for chain ending, where active nodes are paid back with the bond reward, protocol reserve contributors are refunded, all pools are cleared, and all stakers are refunded.

With ⅔ bonded, the network is optimally secure and resistant to cartels and plutocracy. In the event the network is close to approaching this state, an emergency "escape" hatch procedure called Ragnarok is executed booting out all validators (active and standby set) and liquidity providers, refunding their bonds / stakes to them respectively.

In current Tendermint code, the block proposer decides transaction ordering (irrespective of gas prices). Front running may occur with malicious validators.

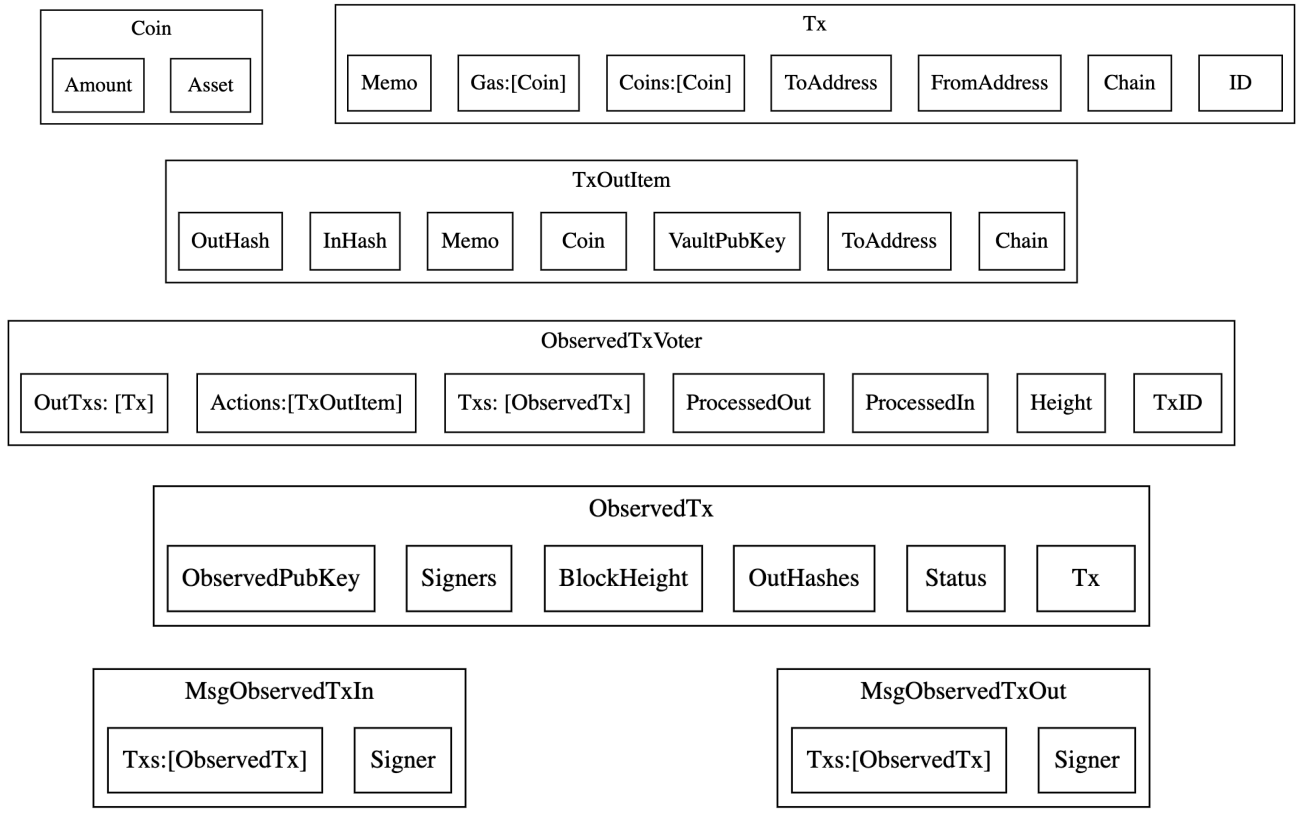
The current Ragnarök implementation of the protocol are split into two stages, with the first stage clearing Yggdrasil vaults and paying back active nodes' bond reward, and the second stage refunding the bond, stake, and the reserve.

- **Stage 1:** Transferring Remaining Yggdrasil Assets + Distributing Bonder Reward
  - For each active nodes, the `recallYggFunds` procedure is invoked to request Yggdrasil vault return.
    - The protocol obtains the Yggdrasil vault of the active node and picks the active Asgard vault with the minimum RUNE.
    - For each chain managed by the Asgard vault, the protocol creates a new outbound YggdrasilReturn transaction that sends the Yggdrasil's fund to to the Asgard vault.
  - For each active node, calculates the node's bond units (the amount of time it has been active).
    - The node's bond reward is calculated as the protocol's total bond reward (in Rune) * (node's bond units / total bond units). The bond reward is added to the node's bond directly.

- **Stage 2:** Refunding Bonders & Reserve Contributors & Stakers
  - For each active node with bond, the protocol creates a new outbound transaction that sends out 10% of the bond.
  - For each reserve contributor, the contributor's share is calculated by *protocol reserve x (contributor share x 10% / total share)*
    - The protocol creates an outbound transaction that sends out 10% of the contributor's reserve.
  - The protocol triggers the unstake handling procedure for each staker of each pool. After the pools are cleared, the protocol resets pool status back to Bootstrap mode.

BONDER/VALIDATOR/OBSERVER

The primary round-robin in the system is involved in selecting the validator leader for creating blocks, which is on the consensus level handled by Tendermint. The selection of the validator nodes are done in EndBlock of THORChain, but the leader to generate the block is chosen by Tendermint. If that leader doesn't produce a block then other nodes in the newly selected active nodes set will become the leader and produce the block instead.



Outbound transaction and voting related interfaces are briefly demonstrated below for illustration:

- Coin: the measurement of an asset
- Tx: representation of the protocol transaction, where the asset, source address, and target address are specified.
- TxOutItem: representation of an outbound transaction "task" / "action" to be done.
- ObservedTxVoter: records the voting / observation record toward an inbound / outbound transaction; used for both inbound & outbound transactions.
- ObservedTx: observation wrapper around a transaction; used for both inbound & outbound transactions
- MsgObservedTxIn/Out: The message fired for inbound/outbound transaction observation.

BOND

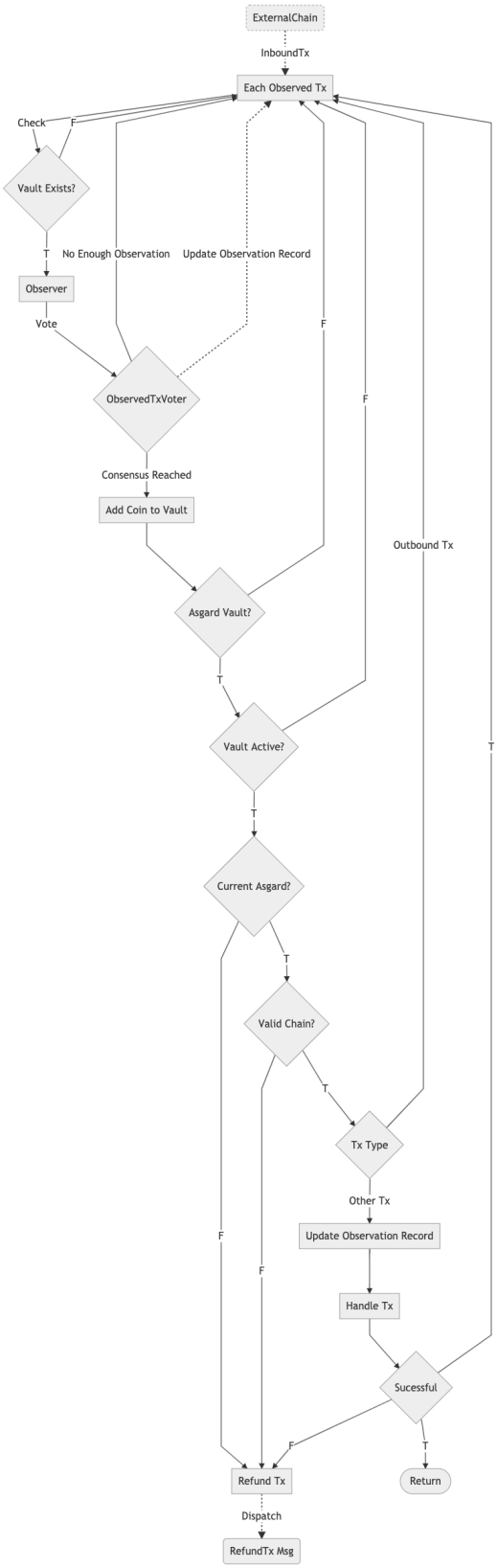
- Validators create one or more transactions on Binance Chain using their RUNE token balance and the corresponding memo indicating their intent to register as a validator candidate (to be whitelisted) on THORChain. This is done by invoking the transaction as a transfer of its balance to the address of the RUNE protocol vault, where the transfer stays as a security deposit.
- Node that bonds more than the `minimumBondInRune` is whitelisted. The bonding record is updated.
- The protocol mints `WhiteListGasAsset` of gas coins to be sent to the newly bonded node.

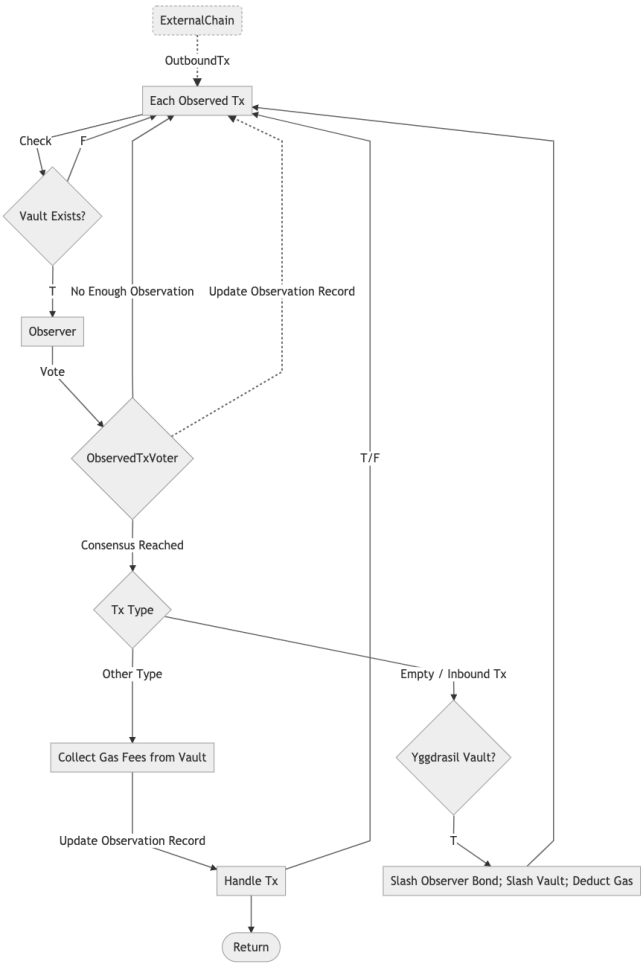
LEAVE

- A validator wishing to leave the set could send out a leave request.
- Upon receiving the leave request, if the validator is in the active validator set, the protocol sets the node's RequestedToLeave.
- In THORChain BeginBlock, every `rotatePerBlockHeight` the protocol scans for nodes that RequestedToLeave, and gives permission to some of them (according to current BFT status).
- In THORChain EndBlock, if the validator's leave request is permitted as of being picked, its bond reward is paid back its bond.

OBSERVE INBOUND TRANSACTION

- When the Bifröst module (which was out of the scope of our investigation) observes a transaction on a supported chain, the `handler_observed_txin.go` tallies the observation as a vote of approval if another validator had previously observed it and otherwise initiates the vote.
  - After being initially observed, the inbound transaction has a time limit for reaching finality. If it passes the time limit without reaching consensus, the observation is considered invalid and the observer is slashed, as illustrated in EndBlock. The "vote" for an inbound transaction is implicitly achieved through observation.
- Each Tx in the inbound transaction is processed individually as follows:
  - The current handler / observer of the transaction gets added to the inbound Tx's ObservedTxVoter record. Each bonded validator has equal rights/weight on observations — one node, one vote.
  - When the inbound Tx gets enough votes (super majority, when 2/3 of the active validator set confirms the observed state), the consensus-ed inbound Tx's coin is added to the inbound Tx's target vault.
    - If the inbound Tx's target vault doesn't exist, the refundTx procedure is triggered, where a refund outbound transaction is added to the outbound transaction queue for processing.
    - The protocol keeps extracting the inbound Tx's msg type and continues processing the transaction.
      - If the transaction processing fails, the refundTx procedure is triggered, where a refund outbound transaction is added to the outbound transaction queue for processing.


ExternalChain
InboundTx
Each Observed Tx
Check
F
Vault Exists?
T
No Enough Observation
Update Observation Record
Observer
Vote
ObservedTxVoter
F
Consensus Reached
Add Coin to Vault
Outbound Tx
Asgard Vault?
T
Vault Active?
T
Current Asgard?
T
Valid Chain?
T
Tx Type
Other Tx
Update Observation Record
Handle Tx
Sucessful
F
T
Refund Tx
Return
Dispatch
RefundTx Msg


ExternalChain
OutboundTx
Each Observed Tx
Check
F
Vault Exists?
T
No Enough Observation
Update Observation Record
Observer
Vote
ObservedTxVoter
T/F
Consensus Reached
Tx Type
Other Type
Empty / Inbound Tx
Collect Gas Fees from Vault
Yggdrasil Vault?
Update Observation Record
T
Handle Tx
Slash Observer Bond; Slash Vault; Deduct Gas
Return

  - The protocol updates the inbound transaction's observation history for slashing later in EndBlock procedure.
- Although the mechanism for this has not yet been implemented, the only transactions to be accepted as valid observations on THORChain are those which have already reached finality on their origin chains (~ 6 confirmations on Bitcoin) or transactions whose value does not exceed that of one block's emission reward.

OBSERVE OUTBOUND TRANSACTION

For each observed outbound transaction, each of its Tx contained in the transaction msg is checked.

- The protocol first ensures that the source vault of the outbound Tx exists.
- The protocol gets the voting status ObservedTxVoter of the outbound Tx, adds the current observer to the ObservedTxVoter to update the voting status.
- If the outbound Tx reaches consensus (super majority, same as the case in observing inbound transaction), then the protocol parses the outbound Tx's memo field for further processing.
  - If the transaction processing fails, the refundTx procedure is triggered, where a refund outbound transaction is added to the outbound transaction queue for processing.
- The outbound Tx's Gas is extracted from the outbound Tx's source vault (and the corresponding pool), and added to the protocol vault.

The protocol updates the inbound transaction's observation history for slashing in the EndBlock procedure later. While facilitating OUTGOING transactions, signers' misbehavior may be double-signing blocks, being offline or not participating in threshold signatures for a consecutive 100 blocks. The amount that a validator is slashed for this depends on the severity of the incident and will likely be finalized with on-chain voting prior to mainnet (though this feature is not yet implemented).

HANDLE OUTBOUND TRANSACTION

The observation of an outbound transaction and the handling of an outbound transaction are separate procedures. An outbound transaction currently is triggered either through `swap`, or asset refund by `unstake`. The handling of these two scenarios has been generalized into `handler_common_outbound`:

- The protocol obtains the ObservedTxVoter (voting record) for the original observed transaction and checks if voting is done (consensus reached).
  - If consensus is reached, this means the outbound transaction has already been processed. The protocol then slashes the current node which "attempts" to send out the extra asset by 1.5x the sent amount from the node's bond as evidence in `slashNodeAccount` procedure.
  - If consensus is not reached, add the current active node to the ObservedTxVoter's voting record.

- The protocol iterates through the outbound transaction queue, and set the `TxOutItem`'s `OutHashes` to indicate finish.
- The protocol re-examines the coin being sent. If more coins are sent than what's specified in the original outbound transaction, then slashes the node by taking 1.5x of the amount from its bond as evidenced in `slashNodeAccount` procedure.

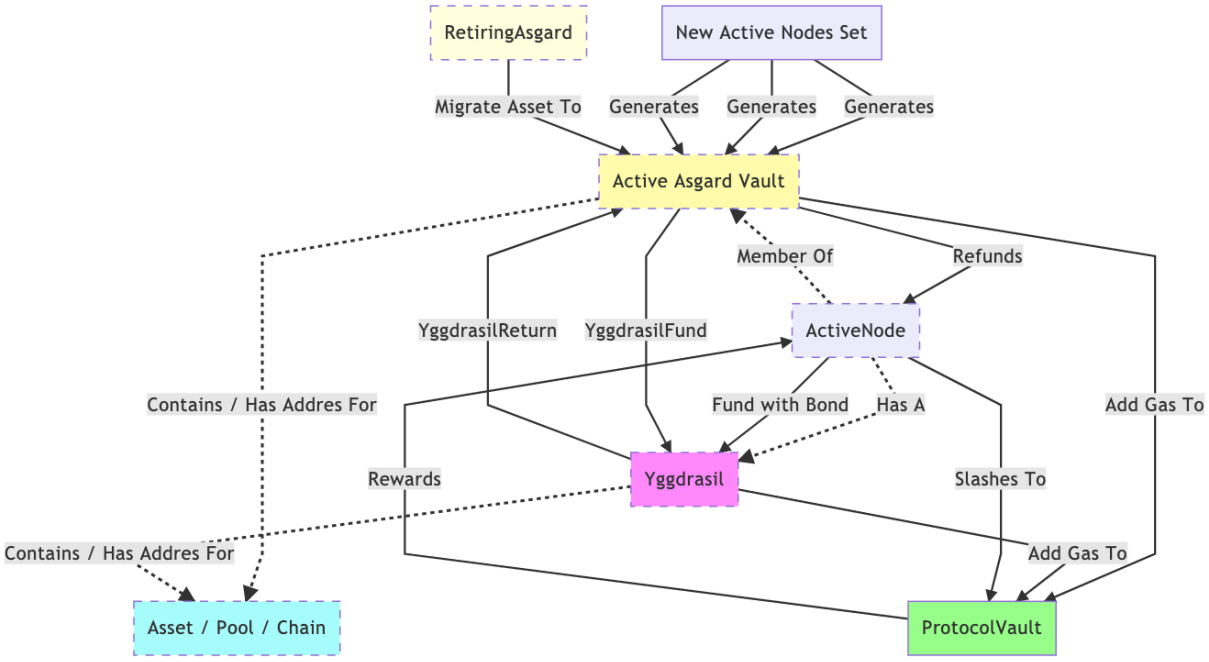
As mentioned above, `slashNodeAccount` handles the situation where a validator attempts to disburse more than the appropriate amount as designated by the protocol, as represented by a "double vote" (more outbound transactions than Actions), and must be penalized for doing so. Additionally, if a node sent some funds with a bad memo, its slashing invocation will be handled through `refundTx`.

The "`Actions`" field of the `ObservedTxVoter` indicates the outbound transaction to process. Once processed, the `outHashes` of the Action item is set, and the action is removed from the `Actions` set to the `OutTxs` set to indicate it is finished. The interfaces is demonstrated above at the beginning of the section. The actions stored in the `ObservedTxVoter` corresponds to the outbound transaction to be processed in the global outbound transaction storage queue.

VAULT TYPES

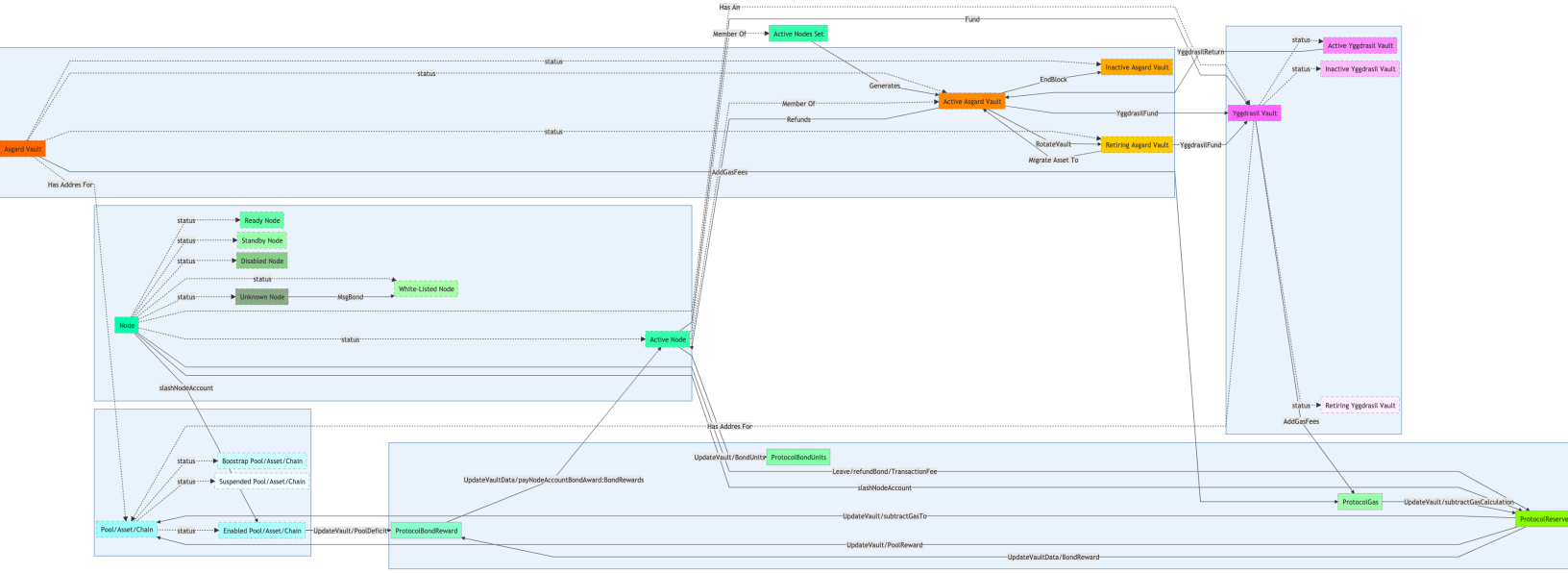
- Vault custodians (bonded validators) for the largest vault of the underlying asset in the pool need to execute TSS on sending the asset being withdrawn to the original depositor's address on the underlying asset's blockchain. Asgards are sharded into sub-asgards to ensure that their size will never exceed 2x the amount bonded by the custodians (TSS-signing validators) securing the vault.
  - While multiple Asgard vaults are supported, the OutBound transactions should percolate through the Asgards with the highest amounts of coins to ensure the least amount of underlying chain transactions necessary to clear the balance.
- The TSS threshold is higher — with an upper bound for both the number of signers and the proportion of total assets custodied by the network — on (primary, "Asgard") vaults that hold custody of assets, with the lowest possible threshold on (secondary, "Yggdrasil") vaults that are in charge of releasing the proceeds from swaps on outgoing transactions.
- The churn schedule for liveness, voluntary departure from the validator set, and good behavior frequently invokes the closing of both types of vaults with an immediate re-opening of replacement vaults having a fresh set of signers.
  - This will get immediately invoked if even just a single signer for a vault is triggered in the current block to be a candidate for being eventually churned out.
  - The migration of funds from closing primary vaults to their replacements is done each consecutive block in increments of 20% over an hour.

- When the transaction is finding the Yggdrasil pool that has sufficient funding, there is a `tx.HasSigned(addr)` check for checking that the vault has signed the transaction: "*if the ygg pool didn't observe the TxIn, and didn't sign the TxIn, THORNode is not going to choose them to send out funds, because they might be offline.*"
  - There is a `mutex` lock to ensure a new outbound transaction is non-concurrently appended to the tail of the queue without simultaneously dropping items from the queue.
  - The actual process happens in the `handle` function of `handler_common_outbound`, where all observed PubKeys are slashed.
- In `txout_store.go`, when there's a transaction going out, there will be a fee charged as reserve. It's the instantaneous equivalent. Ie, if 1 Bolt is paid in fees, and 1 Bolt = 1 Rune, then 1 Rune is then sent to the reserve. This ensures the dues are paid. It is paid by deducting 1 Rune and adding it to the RUNE protocol reserve balance (both of which are virtual).
  - We can find related code in the function `handleV1` in file `handler_reserve_contrib.go`: `vault.TotalReserve = vault.TotalReserve.Add(msg.Contributor.Amount)`
  - The amount of gas paid for processing the outbound transaction on the underlying chain is recorded, the gas fee is handled in `AddGasFee` of `helpers.go`
    - For instance if the chain was Ethereum, we take RUNE from the reserve and reimburse ETH pool for gas, causing it to be RUNE heavy. The community will send in ETH and get back a little extra RUNE, ensuring that gas is always reimbursed because it's economically incentivized to.



A brief illustration on the general relationships between nodes, pools, and vaults are shown as above.

A detailed illustration follows where the status of the nodes, pools, and vaults are expanded below and in the appendix (link).



NEW CHAIN REQUEST

- Anyone in the active validator set may submit a transaction which indicates they are currently observing a chain and are willing to support it.
- The feature is not yet implemented, but similar to the consensus about software versioning, as long as 2/3 of the active validator set is observing a chain it is permanently supported until 100% of the validators indicate their intent to not support it via a special transaction. In this case, prior to being delisted, the chain question is subject to a Ragnarok procedure.

STAKER

POOL CREATION

- When someone creates a pool it's in "bootstrap" mode which means people can stake and unstake but not swap.
  - Every three days the network elects a bootstrap pool to become "enabled" based on the amount of rune staked in the pool.
  - The pool's underlying asset's symbol must be recognized by a supported chain.

STAKE

Stake lets stakers provide liquidity to a certain pool and earn liquidity fees in return. When a liquidity provider deposits only one asset of a pool (RUNE or the underlying), then it's expected that arbitrageurs will exploit the situation to yield a result identical to the corresponding opposite asset being immediately bought in exchange for an equivalent value in the asset being deposited.

- The protocol initially checks that the total staked RUNE will not exceed the MaximumStake. It also ensures that the total stake RUNE will not exceed the total bond of the active nodes to maintain the bonder-staker security and avoid collusion.
- If the staker's stake satisfies the condition, then the corresponding pool is found / created by the protocol for adding the asset.
- The protocol updates the staker's pool ownership according to the pool ownership formula (implemented by the `calculatePoolUnits`) procedure and denoted by a pool unit quantity.
- Finally, the staked asset is added to the pool.

UNSTAKE

Unstake lets staker to retrieve its liquidity fee share and get the stake back. The share's worth is much appreciated by the non-share value in the pool — liquidity fees collected on every swap since the time they deposited.

- Liquidity providers may withdraw the liquidity contributed from the pool they contributed.
- The amount to withdraw is specified in terms of basis point of its pool units.
- The actual withdrew amounts is calculated by the `calculateUnstake` procedure.
  - The claimed units is computed as: (withdraw basis point / 10000) * (staker's pool units).
  - The withdrawn RUNE amount is thus: (pool's RUNE balance) * (claimed units / pool's total units).
  - The withdrawn RUNE amount is then: (pool's Asset balance) * (claimed units / pool's total units).
  - The proportion of the two assets of the pool remains invariant after the withdraw.

Finally, the RUNE and Asset amount is deducted from the pool, and the pool units and the staker's units get updated.

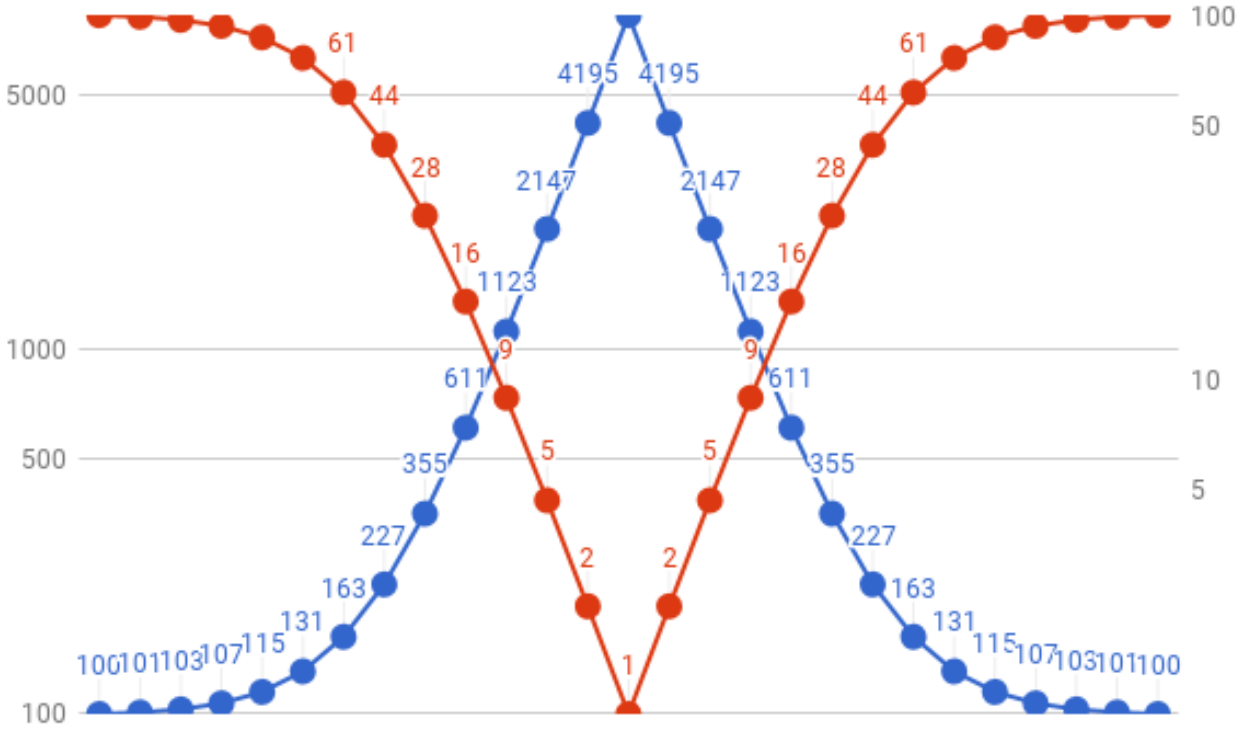
- The protocol creates a new outbound transaction to be appended to the outbound Tx queue.
- The distribution of the fee is done in the `UpdateVaultData` procedure triggered during THORChain's EndBlock. The depth determines the reward size in the `calcPoolReward` procedure where the contributed part is compared against the total amount to get this "depth". Note that when liquidity fee exceeds block reward, it's safe-subtracted to keep the desired equilibrium. The liquidity fee is added to the pool liquidity fee storage directly in the `AddToLiquidityFees` procedure.

- There is an artificial delay of 24hr (*17280 blocks*) wait on withdrawals to prevent double-spend attacks on PoW chains. It requires two transactions to be created, and there is a standard 1 RUNE "network fee" charged on any outgoing RUNE transaction and put into the protocol reserve.

ENDPOOL

- In the EndPool scenario, the protocol triggers “unstake” for each staker of a pool, and the pool gets reset to Bootstrap phase again.

For the sake of visual aid prior to the following section examining the mathematics behind the swap mechanism, we provide a simulation where the y-axis is price and the x-axis is quantity (as in any standard economics graph), for two curves each representing the behavior of one asset in a pool, and display their relationship to each other as governed by the mechanics of the pool:



SWAPPER

According to the CLP formula, here is the expected amount y to be received in exchange for x, where X and Y indicate the depth of each asset in the pool:

$$y = \frac{xYX}{(x+X)^2}$$

The new price of y:

$$P^y_{new} = \frac{x}{y} = \frac{(x+X)^2}{YX}$$

Let there be a transaction x_0 that yields a token output y_0

Let there be a transaction $x_1 > x_0$ that yields a token output y_1

Therefore: $P^y_{new_0} = \frac{(x_0+X)^2}{YX}$ $P^y_{new_1} = \frac{(x_1+X)^2}{YX}$

Since $x_1 > x_0$, it follows that $P^y_{new_1} > P^y_{new_0}$.

Therefore: larger transactions are more expensive and have higher slippage than smaller transactions

SWAP & DOUBLE SWAP

Swaps are handled by the XYK formula with a custom fee calculation that is proportional to the slippage caused by the swap. Due to the swap-based fees it is not necessary to limit swap transactions to some flexible portion of total available liquidity in the pool (which, anyway, would have no net effect on usability, since transactions can be broken up) in order to discourage the possibility of a severe sandwich attack — it simply becomes sufficient cost-prohibitive.

- General user / swapper indicates the types and addresses of the assets they want to swap through the transaction memo field and sends the transaction to THORChain for swap.
- The emitted amount, liquidity fee of a swap is calculated as indicated by the CLP formula in the THORChain white-paper. The price is determined internally based on the balances of the assets in the pool.
  - A slippage based liquidity fee is deduced based on pool depth and how much the transaction size affected it.
- There is a fixed amount of TransactionFee used for validating the swap. If the swapped output is not enough to pay for the transaction fee, the swap is cancelled.
- Double swap, as denoted by A → B, is achieved by two consecutive single swaps, denoted by A → RUNE + RUNE → B.
  - The liquidity fee is charged twice for double swap.

FINDINGS & RECOMMENDATIONS

ADDRESSED (COMMITS)

- https://gitlab.com/THORChain/thornode/-/merge_requests/663/diffs
  - `validator_mgr_v1.go/ragnarokReserve`
    - The order of `totalReserve` and `totalContributions` needs to be swapped.

- https://gitlab.com/THORChain/thornode/-/merge_requests/664/diffs
  - `handler_tss.go/handleV1`
    - When `voter.PoolPubKey` is empty, it is reset, but why `voter.PubKeys` is reset as well? Should there be an independent check to see if `voter.PubKeys` exists?
      - **[THORChain]** when `PoolPubKey` is empty , which means `TssVoter` with id(`msg.ID`) doesn't exist before, this is the first time to create it and thus set the `PoolPubKey` to the one in msg. There is no reason `voter.PubKeys` have anything in it either, thus override it with `msg.PubKeys` as well.
  - `types/querier.go`
    - `var result []string`: `result` should be preallocated.
  - `yggdrasil.go/calculateTargetYggCoins`
    - In the pool iteration section (to ensure no extra amount will be sent to Yggdrasil), the `runeAmt` is added to the counter to indicate the amount of the asset with the comment "*add rune amt (not asset since the two are considered to be equal)*". The comment should clarify why the amounts considered equal, otherwise this is liable to be a functional flaw.
      - **[THORChain]** In a single pool X, the value of 1% asset X in RUNE ,equals the 1% RUNE in the same pool.

- https://gitlab.com/THORChain/thornode/-/merge_requests/671/diffs
  - `handler.go/getMsgStakeFromMemo`
    - For `if !runeAddr.IsChain(common.BNBChain)`, it would be helpful to have more comments about multi-chain interaction logic.
  - `handler_common_outbound.go/handle`
    - Redundant `tx.Tx.Coins.Contains(txOutItem.Coin)` check when setting the `outHash`.

- https://gitlab.com/THORChain/thornode/-/merge_requests/675/diffs
  - `validator_mgr_v1.go/findBadActors`
    - `var tracker []badTracker`: `tracker` should be preallocated.
  - `types/type_event_status.go`

    - `var result EventStatuses`: `result` should be preallocated.
  - `handler_stake.go/processStakeEvent`
    - The `if eventStatus == EventFail` can be removed as the function is always invoked with `eventStatus` set as `EventSuccess`, unless it's deliberately left here for future usage.
  - `validator_mgr_v1.go/ragnarokProtocolStage2`
    - Failing test case for `HandlerEndPoolSuite.TestHandle`:
      - `obtained types.PoolStatus = 1 ("Bootstrap")`
      - `expected types.PoolStatus = 0 ("Enabled")`
  - `handler_observed_txin.go/validateV1`
    - Line 63 loops through all signers of the message. Is it guaranteed that there's exactly one signer? If so we should probably check that and avoid the loop? If not, the validation will succeed if the first signer is authorized but the second is not. Not all of their states are updated.

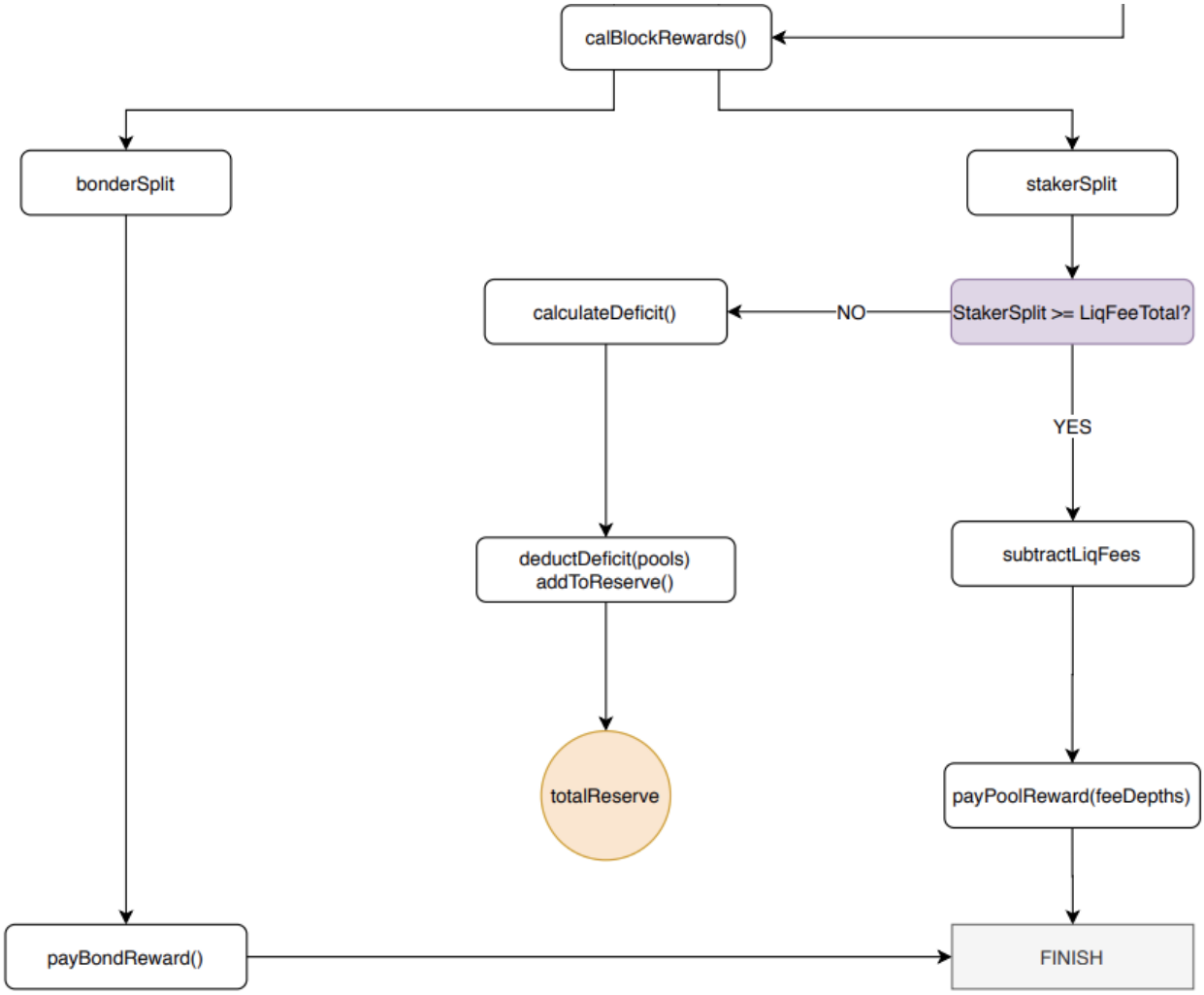
- https://gitlab.com/THORChain/thornode/-/merge_requests/680/diffs
  - `vault_data.go/calcBlockRewards`
    - `var amts []sdk.Uint`: `amts` should be preallocated.
    - Typo "`blocksOerYear`" should be changed to "`blocksPerYear`".
  - `unstake.go/unstake`
    - The lock up block number `17280` should be moved into `constants.go`.
  - `unstake.go/validateUnstake`
    - The check `msg.UnstakeBasisPoints.GT(sdk.ZeroUint()) && msg.UnstakeBasisPoints.GT(…)` can be reformatted as `! msg.UnstakeBasisPoints.GT(sdk.ZeroUint()) || msg.UnstakeBasisPoints.GT(…)`

- https://gitlab.com/THORChain/thornode/-/issues/398
  - The `if emitAssets.GT(Y)` could be `if emitAssets.GTE(Y)` for supporting "just-enough" swap that would round down the pool into bootstrap mode if desired.

- https://gitlab.com/THORChain/thornode/-/issues/397
  - `handler_tss_keysign_fail.go`
    - This handler could be merged with the more generalized `handler_tss_keysign`.

- https://gitlab.com/THORChain/thornode/-/issues/396
  - `handler_pool_data`
    - Recommend double check the usage of this handler for production code at later stage.

- https://gitlab.com/THORChain/thornode/-/issues/395
  https://gitlab.com/thorchain/thornode/-/merge_requests/753/diffs
  https://gitlab.com/thorchain/thornode/-/merge_requests/756/diffs
  - `keeper_vault_data.go/UpdateVaultData`
    - There were multiple gas subtraction operations after the gas is subtracted from the total reserve. The conditional check `vault.TotalReserve.LT(totalPoolRewards)` was slightly confusing, as the totalPoolRewards came from the total liquidity fee, though the gas had already been subtracted from the total liquidity fee earlier. Since this logical flow was intended to be a failsafe that by definition will not happen, and given the amount of complexity that it involves, it was removed.
  - If there are no fees in a block, block rewards are paid out based on historical fees (not poolDepths), while retaining logic around systemIncome and poolDeficit. The new, simplified business logic flow is below:



- https://gitlab.com/THORChain/thornode/-/issues/394
  - `swap.go/swapOne`
    - The passed-in argument `pool` is unused. Recommend removing the argument.

- https://gitlab.com/THORChain/thornode/-/issues/392
  - `handler_observed_txin.go/validateV1`
    - New observers are marked as active in `validateV1` of `handler_observed_txin.go` but are not `validateV1` of `handler_observed_txout.go`. Should the behaviors of the two functions be unified?

ADDRESSED (DISCUSSIONS)

- `slashing.go/slashNodeAccount`.
  - When the asset is RUNE, 1.5x of the slashed amount is deducted from the node account, with 0.5x of the slashed amount added back to the protocol reserve. But the other 1.0x seems just being burnt without being transferred to anywhere else.
    - **[THORChain]** We take 0.5x and put it into the reserve, then just leave the pool balances the same, and therefore the 1.0x stays in the pools (never actually leaves).

- `vault_manager.go/EndBlock`
  - Should bad actors have precedence in the churn out queue over well-behaved nodes exercising their right to leave voluntarily, in order to prevent the possibility of both occurring at a sufficient magnitude to unnecessarily spur a Ragnarok? Or should there be a check to see that there are sufficient `Ready` standby nodes who will immediately take their place prior to allowing well-behaved nodes into the churn out queue at all?
    - **[THORChain]** In our mind, the priority is
      - 1) people who want to leave
      - 2) people who don't want to leave
    - **[THORChain]** We cannot prioritize bad actors vs good actors, because good actors may stay in the network too long which may block our ability to upgrade the network.

- `validator_mgr_v1.go/ragnarokProtocolStage2`
  - The pool cleanup in Ragnarok stage II could be restructured to invoke `EndPool` instead of issuing numerous individual `unstake` messages.
    - **[THORChain]** No, because Ragnarok will unstake stakers in 10 rounds, it is different as EndPool will unstake the full amount in one go.

- `vault_manager.go/EndBlock`
  - The migration of retiring vaults in `EndBlock` shares a similar pattern with `ragnarokBond` in `validator_mgr_v1.go`. However, in `ragnarokBond` the nth limit the amount is deducted directly from the bond, and the created `TxOutItem` is using the amount to be migrated, instead of the total expected amount used by the migrate transaction. Should the implementation of the migration in these two functions be unified?
    - **[THORChain]** The reason why they are different is because the gas requirements are different. Migration will only have 1 transaction per coin, per round. A Ragnarok may have 100's or 1,000's of transaction per coin, per round. We wanted to make sure that in a Ragnarok scenario, if we somehow don't have enough gas, the last round is only 0.003% of each person's holdings. This isn't a concern for migration, and if it is... its easy for anyone in the world to throw $1 of BNB at the vault and it will recover. For Ragnarok, they nodes may no longer be able to sign the transaction.

- `swap.go`
  - Limiting swap transactions to some flexible portion of total available liquidity in the pool may have no net effect on usability, since transactions can be broken up anyway, but discourage the possibility of a severe sandwich attack. Otherwise, it could be useful to overlaying onto the end-user swap UX an exponential moving-average for prices (provided Midgard can retrieve an accurate historical record) with users choosing how long of a moving average they want for the transaction.
    - **[THORChain]** We used to have a global slip limit of 30% of the pool. But it was removed because it created a limitation without purpose. Its perfectly ok if someone wants to make a massive swap. It won't "sandwich attack" the pool because the fees would be too high.Recommend double check the usage of this handler for production code at later stage.

- `handler_add`
  - Recommend double check the usage of this handler for production code at later stage. in the event of losing funds.
    - **[THORChain]** handler_add is used to top the asset in a pool without affecting existing staker's stake unit, it could be used to reimburse stakers

- `handler_observed_txin.go/handleV1`
  - The check for `tx.Tx.Chain.IsEmpty()` can be moved upstream into `validate` functions instead of coupling it in the `handleV1` with unnecessary memo parsing if the prerequisite is not satisfied.
    - **[THORChain]** We should not do it in the Validate function: potentially there might have multiple transactions getting bundled into one MsgObservedTxIn, if one transaction's chain is empty, we still need to process the rest, thus tx.Tx.Chain.IsEmpty check in handleV1 is fine, also we should have ⅔ majority

observed the tx before we refund it, thus the check is in the right place.

- `handler_observed_txout.go/handleV1`
  - `vault.SubFunds(tx.Tx.Coins)` happens both in `handleV1` and `AddGasFees`. Is the fund doubly charged?
    - **[THORChain]** No, vault.SubFunds(tx.Tx.Coins) is to take the actual coins out of vault, while the one in AddGasFees is take out the gas spent in the transaction from vault, these two are different fields for different purposes, it is not double spending.

- `Yggdrasil`
  - Though it may be considered somewhat redundant or labeled as a mere sanity check, there should be a check to ensure a Yggdrasil is never permitted to process an outgoing transaction where 1.5x the value of the transaction ≥ the amount bonded by the signing node. THORChain should always ensure provisioning for a potential slash, and guarantee that there is a large enough balance in a validator's bond relative to the size of any outbound transaction which may pass through the validator's Yggdrasil.
    - **[THORChain]** Yggdrasil pool will only be funded with half of the value of their bond, and when we choose which Yggdrasil to send out transaction, it always ensures the Yggdrasil pool has enough balance to process the transaction.

- `handler_observed_txin.go`
  - There could be a sanity check for whether the RUNE value of incoming transactions would exceed the value of bonded Rune by all validators.
    - **[THORChain]** We already have protection for staking scenario, swap is not practical, because you will lose money.

- `keeper_events.go/UpsertEvent`
  - There is no check for if the event ID matches with the transaction hash for an event ID which already exists. Recommend adding the else branch check for `if event.ID == 0`.
    - **[THORChain]** I don't think we should do that as some of the events are generated by THORChain itself, for example migration, Yggdrasil fund -- the transaction hash will be the same because there is no inbound transaction.

BOOTSTRAPPING POOLS

- Since only one of the pools will be selected once every three days, the opportunity cost of staking to a pool that consistently doesn't get selected can become an unnecessarily deterring factor. Instead of allowing this occurrence, we propose to borrow a solution from the Cosmos SDK called *commitment tokens*.

- The Tendermint problem addressed by this solution was related to vanquishing staking rewards from active validators by needing to delegate to standby validators. A similar mechanism can be created for the liquidity pools, allowing people to issue commitments that will automatically switch their liquidity over to the new pool once it receives enough commitments.
  - **[THORChain]** Its an interesting concept. Our first thought is that I like the simplicity of the current design. It doesn't anyone to learn any new "concepts" and its completely clear and transparent system. We would be a bit nervous about adding more complexity to the system when we have a via solution that feels pretty intuitive.
- When staking RUNE there is a check to make sure the bonded RUNE is not exceeded, but it doesn't happen when staking non-RUNE. From first appearance this would be a beneficial situation because it is guaranteed to create a temporary arbitrage opportunity and increase the demand for RUNE (potentially the price of RUNE also). This is concerning because a price fluctuation of the non-RUNE asset could temporarily put the total value of the staked assets higher than the value of RUNE bonded.
  - **[THORChain]** Assumptions: no 24hr limit on withdrawals, starting pools: 0

    Alice: 1000 Rune & 100 BNB,

    Malice: 1000 Rune & 100 BNB

    Real world price is 10, Alice & Malice both have 200 BNB total value

Alice	1000		100	200				
ce (after)	**1000**	**10.00**	**100**	**Balance (after)**		550		550
Malice	0		100	200				
ce (after)	**1000**	**5.00**	**200**	**Balance (after)**		550	150	700
						78.57%	21.43%	100.00%
lance (X)	**1,000.00**	**5**	**200**	**Balance (Y)**				
Input (x)	1,000.00		100	Output (y)	(x * Y) / (x + X)	**50.00**	**Tokens Emitted**	X * Y) / (x + X)^2
(during)	**2,000.00**	**13.33333333**	**150**	**Balance (during)**				
Alice	1571.428571		117.8571429	275	<- Total Value			
Malice	428.5714286		32.14285714					
			50.00					
	428.5714286		82.14	125.00	<- Total Value			
lance (X)	**2000**	**13.33333333**	**150.00**	**Balance (Y)**				
Output(y)	268		23.20	**Input (x)**	(x * Y) / (x + X)	**232.01**	**Tokens Emitted**	X * Y) / (x + X)^2
ce AFter	**1732**	**10.0005867**	**173.20**	**Balance After**				
Alice	1360.936984		136.0857143	272.1714286	<- Total Value			
Malice	371		37.11					
			50.00					
			87.11	124.23	<- Total Value			

Alice stakes: 1000:100 (price is 10) --> pool ownership 100%
Malice stakes: 0:100 --> pool ownership 21% (alice has 79%)
Pool: 1000:200 (price is 5)

Malice swaps 1000 RUNE into 50 BNB (price of 20)
Pool: 2000:150

Alice withdraws: 1571 RUNE & 117 BNB
Malice withdraws: 428 RUNE & 32 BNB (+50 BNB from her swap)

Assuming real market price of 10:
Alice has 275 BNB value,
Malice has 125 BNB value

Even if you allowed an arb to step in prior to withdrawing:

Pool: 2000:150 (price is 13)
Arb swaps 23 BNB for 268 RUNE
Pool: 1732:173 (price of 10)

Alice withdraws: 1360 RUNE & 136 BNB
Malice withdraws: 371 RUNE & 37 BNB (plus 50 BNB from her swap)

Alice Total: 272 BNB Value
Malice Total: 124 BNB value