

THORChain

Introduction

THORChain

THORChain is a decentralised cross-chain liquidity protocol based on [Tendermint](#) & [Cosmos-SDK](#) and utilising [Threshold Signature Schemes](#) (TSS). It does not peg or wrap assets, it simply determines how to move them in response to user-actions.

THORChain observes incoming user deposits to vaults, executes business logic (swap, add/remove liquidity), and processes outbound transactions. THORChain is primarily a leaderless vault manager, ensuring that every stage of the process is byzantine-fault-tolerant.

THORChain's primary objective is to be resistant to centralisation and capture whilst facilitating cross-chain liquidity. THORChain only secures the assets in its vaults, and has economic guarantees that those assets are safe.

ROLES

There are four key roles in the system:

1. **Liquidity providers** who add liquidity to pools and earn fees and rewards
2. **Swappers** who use the liquidity to swap assets ad-hoc, paying fees
3. **Traders** who monitor pools and rebalance continually, paying fees but with the intent to earn a profit.
4. **Node Operators** who provide a bond and are paid to secure the system

COMPONENTS



Users make signed transactions with a "memo" conveying intent into vaults, which is then picked up by THORChain and executed.

They do not need to hold RUNE, or even care that RUNE was used, or even connect directly with THORChain. This makes THORChain suitable to be integrated into almost any wallet/service/exchange-provider as the "backend" liquidity protocol.



THORChain churns vaults regularly to resist capture. Do not send transactions directly to THORChain vaults without first doing important safety checks.

ASGARDEX (Interface)

ASGARDEX is an interface that allows users to connect to wallets, read balances, query Midgard and broadcast transactions. ASGARDEX can be served from both the web and desktop environment. Anyone can build their own interface, and several wallet libraries have been built to help developers with this.

MIDGARD (API)

Midgard is run by every THORNode and provides a restful API & graphql & websockets that any client can consume to display data. To connect to Midgard, the client (eg wallet) must first challenge a number of nodes to prevent being attacked or phished, since the security model of THORChain is strictly by-consensus.

THORChain (Ledger)

THORChain itself is a ledger that both settles external state, as well as transactions of THOR.RUNE - the network asset. THORChain could be called an app-chain, where the application is a decentralised exchange.

DEVELOPERS

Developers build products that integrate with THORChain, such as wallets, exchanges and other services. Developers simply need to connect to Midgard, they do not need to run a node.

The order of integration is as follows:

1. Add open-source NPM `asgardex` packages to your project
 2. Use `asgardex-midgard` to connect to THORChain via Midgard
 3. Use data provided by Midgard to display in your product
 4. Use `xchain-crypto` to manage keystores for your wallet
 5. Use `xchain-binance` `xchain-ethereum` or `xchain-bitcoin` to sign transactions and broadcast.
-

CONTRIBUTING

THORChain is a public project. Anyone can help contribute to the ecosystem. Start here to learn about the contribution process, as well as upgrading the chain:

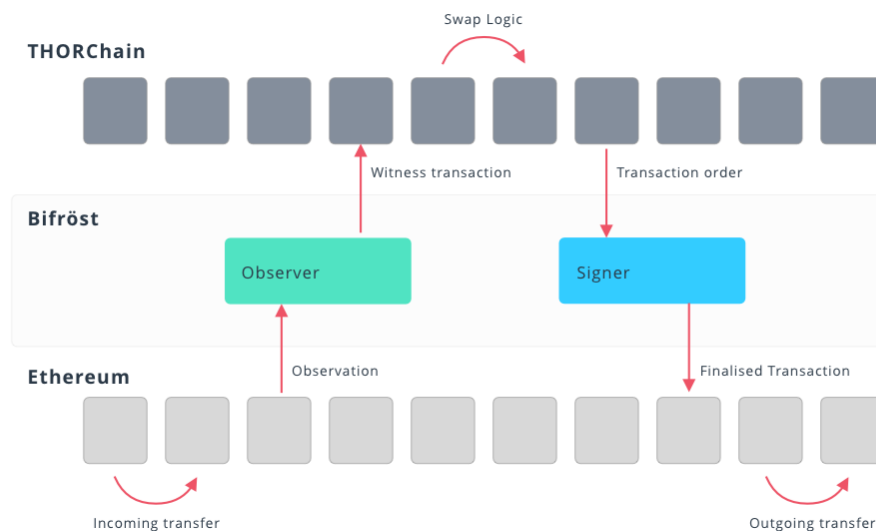
1. Connecting to more chains
2. Adding more business logic to the chain (stablecoins, derivatives etc)
3. Upgrading the chain to be more robust

Technology

Overview

THORChain is a leaderless vault manager:

1. 1-way State Pegs allow syncing state from external chains
2. A State Machine to coordinate asset exchange logic and delegate redemptions
3. Bifröst Chain Client to convert redemptions into chain-specific transactions
4. A TSS protocol to enable distributed threshold key-signing



How THORChain works

The Bifröst Protocol: 1-way State Pegs

Each node has a "Bifröst" service that deals with the nuances of connecting to each chain. Once nodes are synced, they watch vault addresses. If they ever see an inbound transaction, they read it and convert it into a THORChain witness transaction.

The witness transaction has the following parameters that are essentially the same for each chain, no matter the type:



```

1 type Tx struct {
2     ID          TxID    `json:"id"`
3     Chain       Chain    `json:"chain"`
4     FromAddress Address `json:"from_address"`
5     ToAddress   Address `json:"to_address"`
6     Coins       Coins    `json:"coins"`
7     Gas         Gas      `json:"gas"`
8     Memo        string   `json:"memo"`
9 }

```

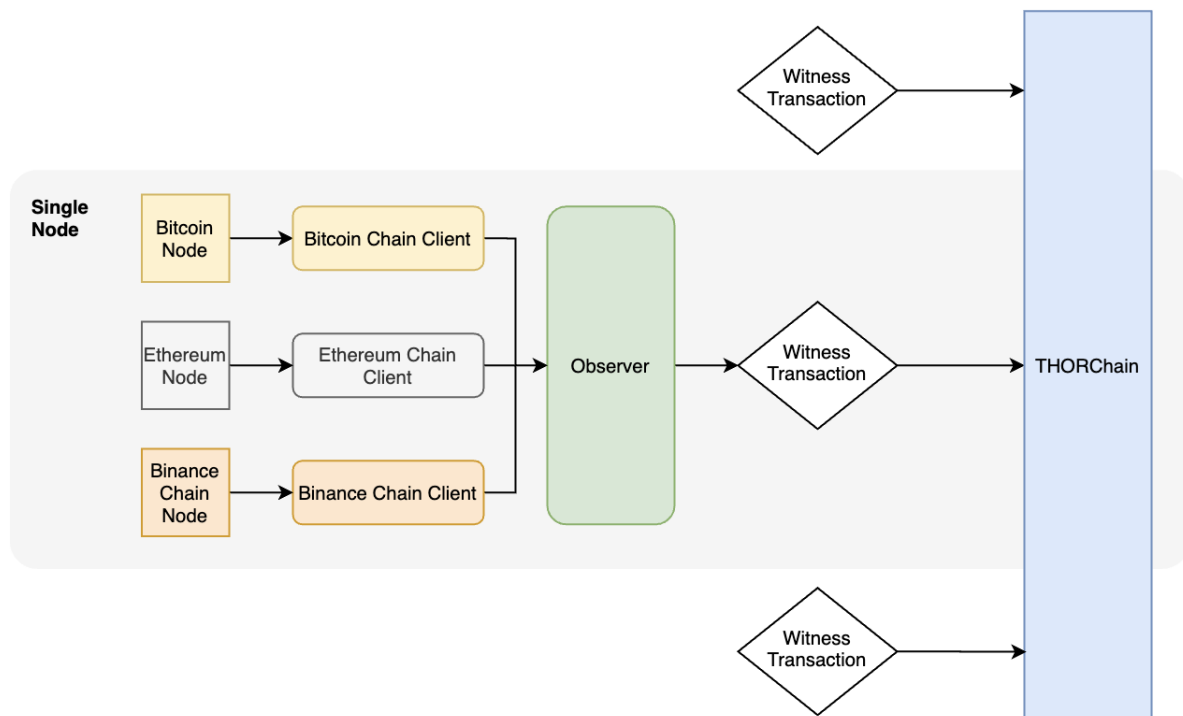
THORChain processes each observed transaction and collects `signers` - essentially the keys of each node that reports a transaction that is 100% identical. Once a super-majority of nodes agree on a particular transaction, it moves from a `pending` state to a finalised state.

```

1 type ObservedTx struct {
2     Tx          common.Tx    `json:"tx"`
3     Status      status      `json:"status"`
4     OutHashes   common.TxIDs `json:"out_hashes"`
5     BlockHeight int64       `json:"block_height"`
6     Signers     []sdk.AccAddress `json:"signers"`
7     ObservedPubKey common.PubKey `json:"observed_pub_key"`
8 }

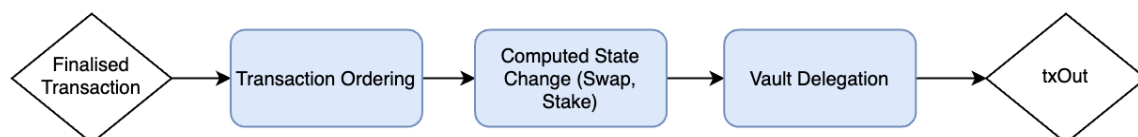
```

Each chain client is quite light-weight, containing only as much logic as is necessary to connect to that particular chain. Most of the logic is in the observer itself.



THORChain State Machine

The state machine processes the finalised transaction and performs logic, such as ordering transactions, computing state changes, and delegating them to a particular outbound vault. Finally, a `txOut` item is created and stored in the Key-Value store.



The `txOut` looks like the following:

```

1 type TxOutItem struct {
2   Chain      common.Chain `json:"chain"`
3   ToAddress  common.Address `json:"to"`

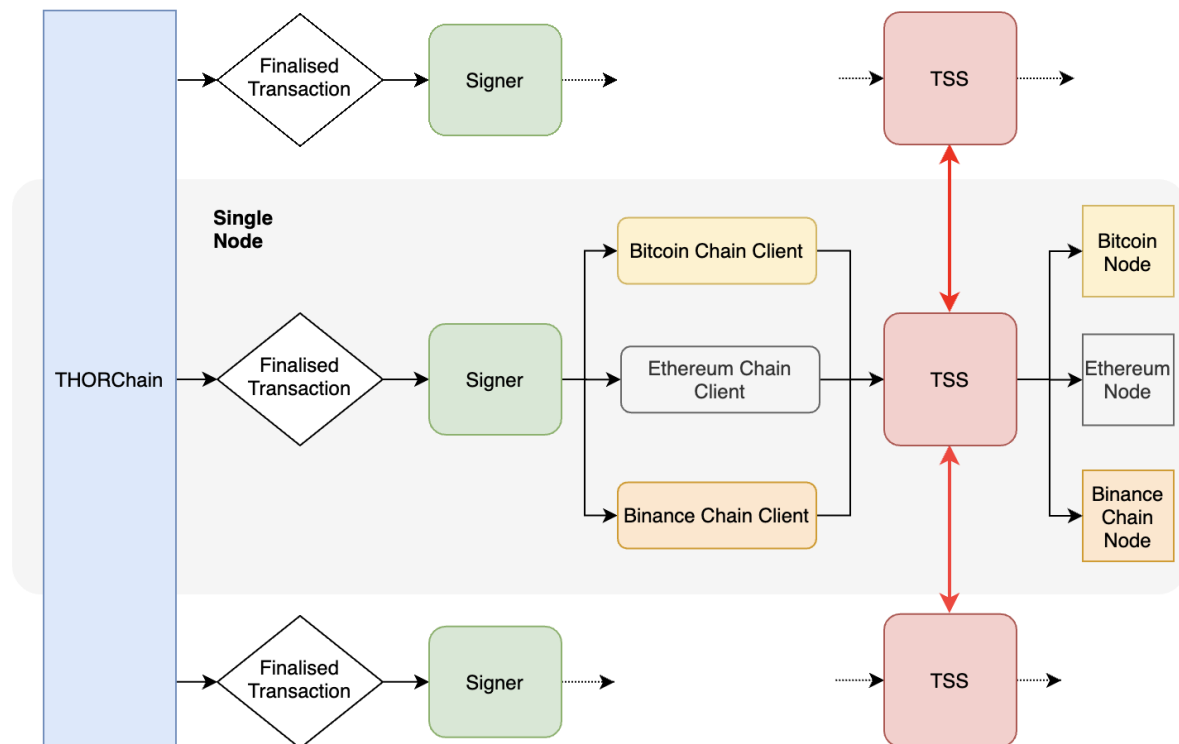
```

```
4   VaultPubKey common.PubKey `json:"vault_pubkey"`
5   Coin         common.Coin  `json:"coin"`
6   Memo         string       `json:"memo"`
7   MaxGas       common.Gas   `json:"max_gas"`
8   InHash       common.TxID  `json:"in_hash"`
9   OutHash      common.TxID  `json:"out_hash"`
10 }
```

The Transaction Out item details which chain it should be sent on, the destination address, the vault it should be sent from, and the maximum gas that should be spent. It also has fields for the transaction that initiated it (the `InHash`) and the transaction that will complete the accounting (the `OutHash`).

Signer (Bifröst)

Once the finalised transaction is created, the Signer loads it from their local copy and serialises it into a correct transaction for the destination chain using the respective chain client. This is then sent to the TSS module which coordinates key-signing. The final signed transaction is then broadcast to the respective chain.



THORChain Vaults

There are two types of vaults in THORChain's system - "inbound vaults" and "outbound vaults":

1. **Asgard** TSS Vaults - inbound vaults with large committees (*24of36*)
2. **Yggdrasil** Vaults - outbound vaults with *1of1* single-signer

This allows the system to use the security of the large vaults to hold the bulk of assets, but delegate to the small, fast outbound vaults the outgoing assets. Every THORNode runs an outbound vault.

i It takes 10-20 seconds to sign 24of36 TSS, so the system would be extremely limited if this vault did all the outbounds. But with each node (36) running an outbound vault, the system can do roughly 2-3 transactions per vault per second, so around 500-1000 times the output.

Multi-realm Asgard Vaults

In order to further increase the node count to beyond 100 nodes, the system shards Asgard vaults into two when it approaches the `MaxNodesForAsgard` constant (and merges them if two ever drop below half of this). As such, with 100 nodes, there would be 3 Asgard vaults, with 100 yggdrasil vaults. The system constantly checks which vault has the highest excess security, and instructs users to send funds there.

Managing Yggdrasil Funding

The state machine constantly monitors and tops up each yggdrasil outbound vault, limited to 25% of the value of its bond in assets. Thus if a node bonded \$4m, then up to \$1m in assets would arrive on its vault. These top up transactions are identified with `yggdrasil+` memos.

When the node churns out, it automatically returns these assets back to Asgard with a `yggdrasil-` memo.

Migrating

When the network churns, it creates new public keys and migrates the funds forward. The churning process is split up in 5 different transactions, 1 per asset (identified by `migrate` memo). It typically takes a few hours to complete. Users are instructed to only send funds to the newest vault, but the retiring vault is monitored. Once the last of the 5 migrations is complete, the previous vault is discarded and no longer monitored.



The previous vault cannot be monitored forever since it can not be guaranteed that all nodes in that vault are still online, and it becomes an attack vector to keep old vaults "around".



If you send funds to a retired vault (likely by caching the address) your funds will be forever lost and is (almost) impossible to be recovered.

Recovery involves significant coordination and is rarely attempted.

RUNE

Overview


RUNE is the asset which powers the THORChain ecosystem and provides the economic incentives required to secure the network. RUNE has four key roles which are described below.

1. Liquidity (As a settlement asset)
2. Security (As a sybil-resistant mechanism, and a means for driving economic behaviour)
3. Governance (Signalling priority on-chain)
4. Incentives (Paying out rewards, charging fees, subsidising gas)

1. Liquidity

Transmitting Purchasing Power

Since RUNE is bonded to assets in its pools, then as the value of those assets increase, then the RUNE value will also increase. This means the system can become "aware" of the value of the assets it is trying to secure. Once it is aware of the value of the assets it is securing, it can use incentives to ensure security of those assets.

 *A rule of thumb is for every \$1m in main-chain assets pooled in liquidity pools, \$1m of RUNE is required to be pooled along side. Due to a mechanism called the Incentive Pendulum, \$2m in RUNE will be driven to be bonded. Thus, \$1m in main-chain assets will cause the total value of RUNE to be \$3m in an equilibrium. Thus liquidity pools have a positive effect on the monetary base of RUNE.*

Providing Liquidity Incentives

Since RUNE is the pooled asset, incentives can be paid directly into each pool. This extra capital is owned by the liquidity providers, and over time, slowly "purchases" the paired asset via arbitrage. Thus RUNE liquidity incentives can drive real yield to LPs.

Solving $O(n^2)$ Problem

Without a native settlement currency, each asset would need to be pooled with every other asset, which would eventually result in hundreds of new pools to be created for just one new asset, diluting liquidity. Having RUNE as the base pair allows any asset to be guaranteed to swap between any other asset.

No. of Assets (eg. BTC, ETH)	No. of Pools (Arbitrary Pairs)	No. of Pools (RUNE Pairs)
n	$pools = (n*(n-1))/2$	$pools = n$
12	66	12
24	276	24
100	4950	100

2. Security

Sybil-resistance

Sybil-resistance refers to the ability to prevent someone masquerading as many identities in order to overcome a network. Bitcoin uses Proof-of-Work (one-cpu-one-vote) to prevent a network take-over. Ethereum 2.0 will use Proof-of-Stake (32-eth-one-vote) to prevent a network take-over.

THORChain could be called Proof of Stake (PoS) network, but there are some differences to warrant it being called a **Proof of Bond** network instead. In THORChain the nodes commit a bond (around 1,000,000 RUNE) in order to be churned in. However, this bond isn't just used to identify a node (give them a voting slot), it is used to underwrite the assets in the pools. If the node attempts to steal assets, then their bond is deducted to the amount of the assets

they stole (1.5x), and the pools are made whole. Additionally, if nodes misbehave their bond is slashed, ensuring reliable service.

Underwriting Assets

The Incentive Pendulum ensures that Nodes are incentivised to continually buy and bond enough Rune each time to maximise their gains - which is a maximum when there is 67% of RUNE bonded and 33% pooled in pools. If the pools are holding \$1m in capital, then the value of RUNE in the aggregate bond is \$2m. Thus all assets can be underwritten.

The bond is extremely liquid - any RUNE holder can immediately enter or exit their position since RUNE is the settlement asset in all pools. Thus, when a node churns in, the cost basis of their bond is known to them and not an arbitrary figure. This means a node bonding \$1m in RUNE will never contemplate making a decision to steal <\$1m in capital from the network, else they will lose overall.

3. Governance

Signalling Priority for Assets

While THORChain strives to be governance-minimal, there are some parts of the protocol that use committed capital to signal priority. This is the case for the asset listing process, where every few days a new asset can be listed as a pool. In a queue of standby assets, the one with the deepest commitment of RUNE is the one that is listed first.

Signalling Priority for Chains

Additionally, if a connected chain is no longer economically valuable, then all liquidity providers in that chain can leave. If a pool is empty, it will be delisted, and this will cause the network to immediately sever ties with that chain in a process call a "ragnarok".

4. Incentives

Fees

RUNE is the native currency of THORChain and is consumed as transaction fees on the network. All swaps are charged both a fixed network fee, as well as a dynamic slip-based fee. This prevents various attack paths such as denial-of-service attacks, as well as sandwich attacks on a pool.

Subsidising Gas

The network continually consumes gas as it makes outgoing transactions (as well as internal transactions). Gas on networks such as Bitcoin and Ethereum becomes complicated fast, so THORChain doesn't make much of an effort to track every minutia of gas consumed. Instead, nodes are free to use at-will the base assets `BNB.BNB` , `ETH.ETH` , `BTC.BTC` , etc in order to pay for gas. These assets are used directly from the vaults.

THORChain then observes outgoing transactions, reports on the gas used, and then pays back the liquidity providers in those pools to the value of twice the amount of gas used (in RUNE).

Paying out Emissions

After fees are charged and gas is subsidised, then THORChain computes the block reward, divides it based on the Incentive Pendulum algorithm, and then pays out to Bonders and Liquidity providers.

This drives Nodes to bond the optimal amount, and pays Liquidity providers for their contribution of liquidity.

Frequently Asked Questions

Why use BFT tendermint?

THORChain uses tendermint which is a classical BFT algorithm. 2/3 of the validators need to agree and the network values safety. The chain has instant finality and this is needed to secure cross-chain bridges.

Why does RUNE need to be the settlement asset in every pool?

If each pool is comprised of two assets eg. BTC:ETH then there will be a scaling problem with $n*(n-1)/2$ possible connections. By having RUNE one side of each pool, \$RUNE becomes a settlement currency allowing swaps between any two other asset. Additionally, having \$RUNE in a pool ensures that the network can become aware of the value of assets it is securing.

Why use 1-way state pegs instead of atomic swaps or 2-way asset pegs?

Simply put, Cross-chain bridges are a better solution than Atomic Swaps. Atomic Swaps involve a complex 6-8 process of signing cryptographic keys between two parties that require interactivity. You also need a counter-party on all trades. Cross-chain bridges, coupled with continuous liquidity pools means you don't need a counter-party, and swaps can happen in less than a second.

1-way state pegs are better than 2-way asset pegs because assets don't need to be wrapped or pegged. Instead of having IOU tokens, real assets can be used instead.

Will there be a pool with stablecoin?

There will be a USDr pool, which will be connect with many different USD stablecoins, derisking them all and allowing traders to arbitrage the price to exactly \$1. USDr will use collateralised debt positions in a novel way with the existing continuous liquidity pools so that it is liquid and safe.

Will there be a lockup for staking the tokens?

There is no minimum or maximum time or amount. Join and leave whenever you wish.

What is the monetary policy?

To goal is to have a fixed supply at all times.

Instead of constantly emitting (infinite supply like Cosmos or Ethereum) or reducing the emission down to zero (Bitcoin) the team elect to match emissions to the difference between current circulating supply and the max supply, as well as burning fees. This means there is 500million progressively emitted to nodes for security and liquidity over time.

Is the team profit oriented?

No - not profit orientated. All fees go back to users. There is no revenue model for the team via the protocol. All swap fees go to liquidity providers, all protocol fees are burned, emissions/block rewards go to validators. The team are incentivised through holding the same RUNE as everyone else.

What is the average rate of return that users can expect when staking in pools?

In the high risk pools, upwards of 10% and in the low risk, deep liquidity pools like BNB & BTC expect 3%-8%.

Is Asgard Wallet built by THORChain or a third-party? Asgard Wallet is developed by the THORChain core team. See the codebase on [GitLab](#).

Does THORChain need external sources for price feeds, like oracles or weighted averages? No. THORChain depends on its continuous liquidity pool design and arbitrageurs to set prices. When pools become imbalanced, arbitrage bots trade to rebalance them. THORChain knows the exchange rates between external asset pairs because RUNE binds all pools together. See [Prices](#).

Roles

Liquidity Providers

Liquidity providers provide assets to the THORChain liquidity pools. They are compensated with swap fees and system rewards. Compensation is affected by a number of factors related to the pool and the state of the network.

! Liquidity providers commit capital to pools which have exposure to underlying assets, thus liquidity providers gain exposure to those assets, which have free-floating market prices.

While they are paid block rewards and liquidity fees, these are dynamic and may not be enough to cover "Impermanent Losses", which occur when price changes happen.

Liquidity providers should not consider they are entitled to receive a specific quantity of their assets back when they deposit, rather that they will receive their fair share of the pool's earnings and final asset balances.

Compensation

Liquidity providers deposit their assets in liquidity pools and earn yield in return. They earn tokens in Rune and the pool's connected asset. For example, someone who has deposited in the BTC/RUNE pool will receive rewards in BTC and RUNE.

Yield is calculated for liquidity providers every block. Yield is paid out to liquidity providers when they remove assets from the pool.

Rewards are calculated according to whether or not the block contains any swap transactions. If the block contains swap transactions then the amount of fees collected per pool sets the amount of rewards. If the block doesn't contain trades then the amount of assets in the pool determines the rewards.

BLOCK WITH SWAPS

How a block with fees splits the reward. In this example, 1000 RUNE is being divided as rewards:

Pool Depth (RUNE)	Fees	Share (of Fees)	Rewards
1,000,000	1000	33%	333
2,000,000	0	0%	0
3,000,000	2000	67%	667
6,000,000	3000	100%	1000

BLOCK WITH NO SWAPS

How a block with no fees splits the rewards. In this example, 1000 RUNE is being divided:

Pool Depth (RUNE)	Fees	Share (of Depth)	Rewards
1,000,000	0	17%	167
2,000,000	0	33%	333
3,000,000	0	50%	500
6,000,000	0	100%	1000

This ensures that yield is being sent to where demand is being experienced - with fees being the proxy. Since fees are proportional to slip, it means the increase in rewards ensure that pools experiencing a lot of slip are being incentivised and will attract more liquidity.

Factors Affecting Yield

Ownership % of Pool – Liquidity providers who own more of a pool receive more of that pool's rewards.

Swap Volume – Higher swap volumes lead to higher fees. Higher fees lead to higher rewards for liquidity providers.

Size of Swaps – Swappers who are in a hurry to exchange assets will tend to make larger swaps. Larger swaps lead to greater price slips and therefore higher fees.

Incentive Pendulum – The Incentive Pendulum balances the amount of capital bonded in the network versus pooled. It does this by changing the amount of rewards given to node operators versus liquidity providers. Sometimes rewards will be higher for liquidity providers to encourage them to deposit assets; sometimes the opposite. [Learn more](#).

Change in Asset Prices -- If the price of the assets change, then liquidity providers will receive more of one and less of the other. This may change yield if yield is being priced in a third asset, ie, USD.

Calculating Pool Ownership

When a liquidity provider commmit capital, the ownership % of the pool is calculated:

$$\text{slipAdjustment} = 1 - \left| \frac{Ra - rA}{(r + R) * (a + A)} \right|$$

$$\text{units} = \frac{P(Ra + rA)}{2RA} * \text{slipAdjustment}$$

- r = rune deposited
- a = asset deposited
- R = Rune Balance (before)
- A = Asset Balance (before)

- P = Existing Pool Units

The liquidity provider is allocated rewards proportional to their ownership of the pool. If they own 2% of the pool, they are allocated 2% of the pool's rewards.

How it Works

Depositing Assets

Depositing assets on THORChain is permissionless and non-custodial.

Liquidity providers can propose new asset pools or add liquidity to existing pools. Anybody can propose a new asset by depositing it. See [asset listing/delisting](#) for details. Once a new asset pool is listed, anybody can add liquidity to it. In this sense, THORChain is permissionless.

The ability to use and withdraw assets is completely non-custodial. Only the original depositor has the ability to withdraw them. Nodes are bound by rules of the network and cannot take control of user-deposited assets.

Process

Liquidity can be added to existing pools to increase depth and attract swappers. The deeper the liquidity, the lower the fee. However, deep pools generally have higher swap volume which generates more fee revenue.

Liquidity providers are incentivised to deposit symmetrically but should deposit asymmetrically if the pool is already imbalanced.

Symmetrical vs Asymmetrical Deposits

Symmetrical deposits is where users deposit an *equal* value of 2 assets to a pool. For example, a user deposits \$1000 of BTC and \$1000 of RUNE to the BTC/RUNE pool.

Asymmetrical deposits is where users deposit *unequal* values of 2 assets to a pool. For example, a user deposits \$2000 of BTC and \$0 of RUNE to the BTC/RUNE pool. Under the hood, the member is given an ownership of the pool that takes into account the slip created in the price. The liquidity provider will end up with <\$1000 in BTC and <\$1000 in RUNE. The deeper the pool, the closer to a total of \$2000 the member will own.

Note: there is no difference between swapping into symmetrical shares, then depositing that, or depositing asymmetrically and being arb'd to be symmetrical. You will still experience the same net slip.

Withdrawing Assets

Liquidity providers can withdraw their assets at any time. The network processes their request and the liquidity provider receives their ownership % of the pool along with the assets they've earned. A network fee is taken whenever assets are taken out of the network. These are placed into [the network reserve](#).

Yield Comes from Fees & Rewards

Liquidity providers earn a yield on the assets they deposit. This yield is made up of fees and rewards.

Fees are paid by swappers and traders. Most swaps cause the ratio of assets in the liquidity pool to diverge from the market rate.



The ratio of assets in a liquidity pool is comparable to an exchange rate.

This change to the ratio of assets is called a 'slip'. A proportion of each slip is kept in the pool. This is allocated to liquidity providers and forms part of their staking yield. Learn more about [swapping](#).

Rewards come from THORChain's own [reward emissions](#). Reward emissions follow a predetermined schedule of release.

Rewards also come from a large token reserve. This token reserve is continuously filled up from [network fees](#). Part of the token reserve is paid out to liquidity providers over the long-term. This provides continuous income even during times of low exchange volume.

Learn about how [factors affecting yield and how yield is calculated](#).



See here for an [interactive example](#) of the staking process.

Strategies

Passive liquidity providers should seek out pools with deep liquidity to minimise risk and maximise returns.

Active liquidity providers can maximise returns by seeking out pools with shallow liquidity but high demand. This demand will cause greater slips and higher fees for liquidity providers.

Requirements, Costs

Liquidity providers must have assets to deposit and their assets must be native to a supported chain. There is no minimum amount to deposit in existing pools. However new assets must win a competition to be listed – larger value deposits will be listed over smaller value deposits.

Liquidity providers must pay for security of their assets, since security is not free. This "payment" is the requirement for liquidity providers to hold RUNE, which acts as a redeemable insurance policy whilst they are in the pool. Holding RUNE allows liquidity providers to retain an ability to economically leverage nodes to ensure security of assets. When the liquidity provider withdraws, they can sell their RUNE back to the asset they desire. H

The only direct cost to liquidity providers is the [network fee](#), charged for withdrawing assets (pays for the compute resources and gas costs in order to process outbound transactions). An indirect cost to liquidity providers comes in the form of impermanent loss. Impermanent loss is common to Constant Function Market Makers like THORChain. It leads to potential loss of liquidity provider purchasing power as a result of price slippage in pools. However, this is minimised by THORChain's [slip-based fee](#).

Liquidity providers are not subject to any direct penalties for misconduct.

Swappers

On THORChain, users can swap their digital assets for other digital assets. The network aims to give users access to:

- A large variety of assets through cross-chain compatibility and simple asset listing
- Superior user experience through open finance protocols and permissionless access
- 1-transaction access to fast chains (Binance Chain), smart chains (Ethereum), censorship-resistant chains (Bitcoin) and private chains (Monero).

How Swaps Work

Available Assets

Users can swap any assets which are on connected chains and which have been added to the network. Users can swap from any connected asset to any other connected asset. They can also swap from any connected asset to [RUNE](#).



Learn more about how chains and assets get added to the network in [the Governance section](#).

To add an asset to THORChain, users simply deposit a new asset to put it in the queue for listing. Swaps can only be made on pools when they have been added to the network and have moved out of the bootstrap phase.

Decentralisation

THORChain manages the swaps in accordance with the rules of the state machine - which is completely autonomous. Every swap that it observes is finalised, ordered and processed. Invalid swaps are refunded, valid swaps ordered in a transparent way that is resistant to

front-running. Validators can not influence the order of trades, and are punished if they fail to observe a valid swap.

Swaps are completed as fast as they can be confirmed, which is around 5-10 seconds.

Continuous Liquidity Pools

Swaps on THORChain are made possible by liquidity pools. These are pools of assets deposited by Liquidity providers, where each pool consists of 1 connected asset, for example Bitcoin, and THORChain's own asset, RUNE. They're called Continuous Liquidity Pools because RUNE, being in each pool, links all pools together in a single, continuous liquidity network.

When a user swaps 2 connected assets on THORChain, they swap between two pools:

1. Swap to RUNE in the first pool,
2. Move that RUNE into the second pool,
3. Swap to the desired asset in the second pool with the RUNE from (2)

The THORChain state machine handles this swap in one go, so the user is never handles RUNE.

See [this example](#) for further detail and the page below for broader detail on Continuous Liquidity Pools.

→ [Continuous Liquidity Pools](#)

</how-it-works/continuous-liquidity-pools>

Calculating Swap Output

The output of a swap can be worked out using the formula

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

where

- x is input asset amount
- X is input asset balance
- y is output asset amount
- Y is output asset balance

Example

The BTC.RUNE pool has 100 BTC and 2.5 million RUNE. A user swaps 1 BTC into the pool. Calculate how much RUNE is output:

$$\frac{1 * 2500000 * 100}{(1 + 100)^2} = 24,507.40$$

This user swaps 1 BTC for 24,507.40 RUNE.



Run through an [interactive tutorial of an asset swap](#).

Costs

The cost of a swap is made up of two parts:

1. Network Fee
2. Price Slippage

All swaps are charged a network fee. The network fee is dynamic – it's calculated by averaging a set of recent gas prices. Learn more about [Network Fees](#).

Note that users who force their swaps through quickly cause large slips and pay larger fees to liquidity providers.

Traders

Prices on THORChain are maintained by profit-seeking traders. Traders find assets that are mispriced between markets. They buy assets on markets with low prices and sell them on markets with high prices. This earns them a profit.

Traders compare the exchange rates on THORChain with the rates on external markets. If they find the price is lower on THORChain they can buy there and sell on an external market. If they find the price is lower on external markets they can buy there and sell on THORChain. This process is repeated at high-frequency. Over time, price information propagates and THORChain settles with external markets.

This is how THORChain avoids the need for oracles and how prices are set. To learn more, see [Prices](#).

How it Works

Process


A swap takes place in the MATIC/RUNE pool, as described in [Prices](#). This leaves the pool unbalanced. The ratio on THORChain is 20:1 MATIC:RUNE, but is 16:1 on external markets. This means that RUNE is undervalued on THORChain.

Traders can now buy cheap RUNE on THORChain and sell it for a profit on external markets. To do so, they swap MATIC into the pool and get RUNE out. They sell this RUNE on external markets and make a profit.

The economics of the [swap formula](#) mean that traders should aim to restore balance to the pool in a single trade. Rebalancing should be done incrementally. If larger rebalancing trades are attempted, arbitrage may not be profitable for traders.

Specifically, each rebalancing trade should be 40–50% the imbalance size. So if the imbalance starts at \$100 in value, the first rebalancing trade should be between \$40–50.

This will leave the imbalance at \$50–60. The next rebalance should be \$25–30. This process repeats until a satisfactory balance is restored.

 This hierarchical cascade of rebalancing trades will create arbitrage opportunities for traders big and small.

Impact of Liquidity

Trading profits are impacted by liquidity on THORChain and on external markets. As an example, if the price of the asset in a THORChain pool is \$1.20, but the same asset on an external market is \$1.00, then someone can buy off that external market and sell into the THORChain pool for profit.

Infinitely Deep Liquidity

If both markets are infinitely deep, then the following will occur:

- Buy on External Market for \$1.00, no price slip.
- Sell on THORChain for \$1.20, no price slip.
- **Total Profit: 20%**

The trader can then continue to arbitrage for a profit of 20% continuously.

Finite, but Uneven Liquidity

If both markets have finite liquidity, but one is much deeper than the other, then the one of the markets will slip in price after the trade. However, the trader will experience a price that is roughly the average of the price before and after the trade:

- Buy on External Market for \$1.00, no price slip.
- Sell on THORChain for \$1.20, realised price of \$1.10, price slip to \$1.00.
- **Total Profit: 10%**

After the trade, there is no more price differential, but the trader made 10% in profit. The trader has made the pool price equal to the secondary market. They have transferred price

information from one market to another.

Low Liquidity

If both markets have low liquidity, then the trader is attempting to make trades that slip each market towards each other:

- Buy on External Market for \$1.00, realised price of \$1.05, price slip to \$1.10.
- Sell on THORChain for \$1.20, realised price of \$1.15, price slip to \$1.10.
- **Total Profit: >10%**

The market now has no more price differential. The trader has made each market equal to each other.

Compensation, Requirements, Costs & Penalties

THORChain does not offer explicit incentives to traders – it does not reward or punish them. Trading profits are determined by the capacity of traders to seek out and capitalise on price differentials between THORChain and external markets.

The majority of arbitrage opportunities will be exercised by software bots. These are under development by 3rd party entities and will be released in due time. They will be open-source and available for anybody to run.

Node Operators

Overview

THORNodes service the THORChain network, of which there is intended to be initially 99. Each THORNode is comprised of several independent servers. All THORNodes communicate and operate in cooperation to create a cross-chain swapping network.

→ [THORNode Overview](#)

[/thornodes/overview](#)

Rewards

Node Operators earn 67% of the system income when it is stable. You can read more here:

→ [THORNode Overview](#)

[/thornodes/overview](#)

Set up Instructions

Follow these setup up instructions

→ [Deploy - K8 Cluster](#)

[/thornodes/kubernetes](#)

Anonymity

Node Operators should stay anonymous and never socially signal that they are running a node. For this reason, no ability to delegate or token-voting is supported at the protocol level.

Tools

There are a variety of tools available in the ecosystem for Node Operators, such as the Telegram Alerts bot:

→ [Alerting](#)

[/thornodes/alerting](#)

Delegation Is Not Permitted

For THORChain to work, it needs to be a neutral amoral platform that has no opinion on the nature of transactions processed on its network, nor the source or destination of funds. THORChain needs to avoid all sources of bias, subjectivity or localised influence. It needs to be a moving target, shuffling funds and avoiding capture. It needs to treat node operators as second-class citizens, paying them for their services and ejecting any node that has been in the system for too long or starts misbehaving. THORChain treats liquidity providers as first-class citizens and does everything it needs to protect their capital.

Delegation is not permitted on the THORChain protocol because delegation undermines both the decentralisation of the project as well as its security.

Decentralisation and Neutrality

If delegation was permitted, then Node Operators would begin running initiatives that would try and attract delegated funds, such as marketing campaigns, or reducing commissions to low or negative amounts. This type of behaviour is wide-spread on existing DPoS networks such as CosmosHub, EOS and Tron. This begins to erode the centralisation of the network since funds would accumulate with large branded nodes that can pay for large campaigns.

As a follow-on, if nodes began social signalling, they would naturally try and build brands to gain public trust. They would buy domain names, and try and promote on-chain identification of their nodes "monikers" to allow users to easily find and delegate them. Over time, this brand will become more and more associated with the protocol and the protocol would start to lose its neutrality. The branded node would try and assert themselves more and more on the network, and this bias would make its way into the source code and eventually affect how the protocol would operate.



Nodes must stay anonymous and be identified only by their public key and IP address, both of which can be obfuscated from their true identity (such as washing funds prior to bonding and using a proxy node to obfuscate their real IP address).

Security

The most important aspect is that Nodes must pay for their Bond, since this is a core assumption to the security of the network. If a node pays \$1m for their bond, then they will never try to steal assets unless if they can access more than \$1m of funds. If they can, then they will make a profit.

If delegation was permitted, a Node Operator that paid \$100k in RUNE, can possibly receive \$900k in delegated funds to qualify and meet the \$1m bond requirement. They will now contemplate stealing assets the moment they get access to more than \$100k in capital, which is likely.



Nodes must pay for their bond to ensure the economic assumptions of the network are upheld.

THORNodes

THORNode Overview

Overview

THORNodes service the THORChain network, of which there is intended to be initially 99. Each THORNode is comprised of several independent servers in a cluster. All THORNodes communicate and operate in cooperation to create a cross-chain swapping network.

To set up a node, you have three choices:

1. Set up manually
2. Set up via Kubernetes
3. Set up via Provider (coming soon)

→ [Deploy - Manual](#)

[/thornodes/manual](#)

→ [Deploy - K8 Cluster](#)

[/thornodes/kubernetes](#)

THORNode Stack

Each THORNode is comprised of 5 major components.

1. **thord** - this is a daemon that runs the THORChain chain itself
2. **thor-api** - this daemon runs an HTTP server, that gives a RESTful API to the chain
3. **bifrost** - this daemon creates connections to remote chains (like Bitcoin, Ethereum, Binance, etc) to both observe activity on those chains (incoming/outgoing transactions), and also sign/broadcast outgoing transactions (moving funds on remote chains).
4. **thor-gateway** : THORNode gateway proxy to get a single IP address for multiple deployments

5. **midgard** - this daemon is a layer 2 REST API that provides front-end consumers with semi real-time rolled up data and analytics of the THORChain network. Most requests to the network will come through Midgard. This daemon is here to keep the chain itself from fielding large quantities of requests. You can think of it as a “read-only slave” to the chain. This keeps the resources of the network focused on processing transactions.
6. **Full nodes** - for every chain that is supported by the network, each THORNode operator will run their own full node of each chain (Bitcoin, Ethereum, Binance, etc).

Churning

As new nodes join/leave the network, this triggers a “churning event”. Which means the list of validators that can commit blocks to the chain changes, and also creates a new Asgard vault, while retiring an old one. All funds in this retiring vault are moved to the new Asgard vault.

Normally, a churning event happens every 3 days (50,000 blocks), although it is possible for it to happen more frequently (such as when a node optionally requests to leave the network using the **LEAVE** memo).

Churning Out

On every churn, the network selects one or more nodes to be churned out of the network (which can be typically churned back in later). In a given churning event, multiple nodes may be selected to be churned out, but never more than 1/3rd of the current validator set. The criterion the network will take into account is the following:

1. Requests to leave the network (self-removal)
2. Banned by other nodes (network-removal)
3. How long an active nodes has been committing blocks (oldest gets removed)
4. Bad behavior (accrued slash points for poor node operation)

Churning In

On every churn, the network may select one or more nodes to be churned into the network but never adds more than one to the total. Which nodes that are selected are purely by validator bond size. Larger bond nodes are selected over lower bond nodes.



There is an endpoint on Midgard that has deep analytics in mean and median active & standby bond sizes to drive efficient discovery of the best "bond" size. Whilst 1,000,000 is the minimum, competition to get in will drive it up, and in the long term it is likely to stabilise between 2m and 2.5m RUNE.

The network is safe when it is over-bonded, but it shrewd Node Operators will probably actively manage their bond and pool part of it instead to maximise yield.

Risk of Running a Node

Deciding to run a node should be carefully considered and thought through. While the payoffs/rewards can be significant, there can also be an equally significant costs.

Risks

To run a node, you must obtain a significant amount of Rune, currently 1 million Rune. This Rune is sent into the network as "bond" and held as leverage on each node to ensure they behave in the best interest of the network. Running a malicious or unreliable node results in a slashing of this bond.

Here are the ways in which a validator's bond can get slashed

- **Double Sign** (5% of minimum bond) - if it is observed that a single validator node is committing blocks on multiple chains. To avoid this, never run two nodes with the same node account at the same time.
- **Not Observing** (2 slash pts) - if a node does not observe transactions for all chains, while other nodes do, they get slash points added.
- **Not signing a transaction** (600 slash pts) - if a node does not sign an outbound transaction, as requested by the network, they will get slash points added.
- **Unauthorized transaction** (1.5x transaction value) - if a node sends funds without authorization, the bond is slashed 1.5x the value of the stolen funds. The slashed bond is dumped into the pool(s) where the funds were stolen and added to the reserve.

- **Fail to keygen** (1 hr of revenue) - When the network attempts to churn, and attempts to create a new Asgard pubkey for the network, and fails to do so due to a specific node(s), they will lose 1 hr of revenue from their bond.

Slash points undo profits made on the network. For every 1 slash point a node receives, they lose 1 block of profits. If a node account has more slash points than blocks they have been active, they will lose the equivalent in bond.

Compensation

Bond Rewards

Node Operators receive rewards if they are bonded and active on the network and are paid out in Rune. While revenue is generated every block (every 5 seconds) to each operator, those funds are not available to the operator until they churn out of the network. Each operator makes the same amount of income, no matter how much they bond to the network. They're claimed whenever a node leaves the network. See [Keeping Track of Rewards](#) below for more details.

Nodes receive the same amount of rewards regardless of how much [RUNE](#) they've bonded. This stabilises the amount that nodes need to bond. Over time, this stability increases the median bonded amount and the security of the network.

Rewards are affected by the [Emission Schedule](#) and the [Incentive Pendulum](#). Over time, the Emission Schedule decreases the amount of RUNE allocated to nodes. The Incentive Pendulum increases and decreases the amount of RUNE allocated to nodes according to the security and capital efficiency of the network.

Keeping Track

When a node joins the network the current block height is recorded. The system creates one block unit for every active node for every active block, and has a running total of the block units created. When a node leaves, it cashes in its block units for a portion of the bond rewards. The spent block units are destroyed.

For example, there are 10000 RUNE in bond rewards outstanding. Node A has been active for 30 blocks, and has 33 block units, but accrued 3 slash points. There are 1000 block units in total. Node A leaves the network and cashes in its 30 block units (33 - 3). It receives 300 RUNE $((30/1000) * 10000)$, leaving 9700 RUNE in node rewards. Its 33 block units are destroyed, leaving 967 block units left.

Income

Income for one node can be estimated based on a few inputs:

- Number of active nodes
- Reward emission rate
- % of rewards allocated to nodes, set by the Incentive Pendulum
- Price of RUNE*

These inputs should be plugged into the following formula:

$$\frac{RewardAllocation * EmissionRate}{NumberOfNodes}$$

An example with mainnet day 1 inputs:

- 33 nodes
- 3.06 million RUNE rewards emitted per month
- 67% of rewards allocated to nodes (stable Incentive Pendulum)

$$\frac{0.67 * 3060000}{33} = 62,127$$

In this example, an individual operator would receive 62,127 RUNE over the month.

Costs

Depending on how the node was set up, it will likely cost between \$1000 and \$2000 per month, potentially more as the blockchain scales. The main driver of costs is resource allocation to hosting each THORNode service.

Skillsets

Running a THORNode is no simple task. As an operator, you will need to run/maintain multiple linux servers with extremely high uptime. It is encouraged that only professional systems engineers run nodes to maintain the highest quality reliability and security of the network. The simple question to know if you have the skillsets to run a THORNode is:

Have you used pagerduty before?

If the answer is no, it's probably best that you do not run a node and participate in the network in other ways. The following skill sets are required to be an effective node operator.

- Advanced knowledge of Linux server administration and security
 - Advanced knowledge of Kubernetes
 - Advanced experience running a host of nodes on a hosted platform such as AWS, Google Cloud, Digital Ocean, etc
 - Knowledge of running full nodes for other chains such as Bitcoin, Ethereum, and Binance.
 - Willingness to be “on call” at all times to respond to issues when/if your node becomes unavailable
-

THORNode Details

Node Accounts

When you run a THORNode, each THORNode will have its own node account. An example node account looks like this:



```

1  {
2    "node_address": "thor19h62vypuelrj0pv4jhl26wf79yr5zhxcmd5w85",
3    "status": "active",
4    "pub_key_set": {
5      "secp256k1": "thorpub1addwnpepq0ylvhrqepmsm3rqr4ecuyx42l4y29g2d932zls
6      "ed25519": "thorpub1addwnpepq0ylvhrqepmsm3rqr4ecuyx42l4y29g2d932zlse2
7    },
8    "validator_cons_pub_key": "thorcpub1zcjduepqzknjn39xtkdzr6a2zuzry7f02rn3
9    "bond": "30240000000000",
10   "active_block_height": "180",
11   "bond_address": "tbnb1rqhrnvex4p5zchhu0slgr76dc4cl5dnvzxx2h",
12   "status_since": "123",
13   "signer_membership": [
14     "thorpub1addwnpepqwm3arc5xqgjf8yt70psygcjasfj3c776cux4yxcaacyd9vm859lc
15   ],
16   "requested_to_leave": false,
17   "forced_to_leave": false,
18   "leave_height": "0",
19   "ip_address": "206.189.235.75",
20   "version": "0.1.0",
21   "slash_points": "17",
22   "jail": {
23     "node_address": "thor19h62vypuelrj0pv4jhl26wf79yr5zhxcmd5w85",
24     "release_height": "2393",
25     "reason": "failed to perform keysign"
26   }
27 }

```

Most importantly, this will tell you how many slash points the node account has accrued, their status, and the size of their bond (which is in 1e8 notation, 1 Rune == 100000000).

Node Statuses

Types of node status:

1. **Unknown** - this should never be possible for a valid node account
2. **Whitelisted** - node has been bonded, but hasn't set their keys yet
3. **Standby** - waiting to have minimum requirements verified to become "ready" status.
This check happens on each churn event (3 days on average).
4. **Ready** - node has met minimum requirements to be churned and is ready to do so.
Could be selected to churn into the network. Cannot unbond while in this status.
5. **Active** - node is an active participant of the network, by securing funds and committing new blocks to the THORChain blockchain. Cannot unbond while in this status.

6. **Disabled** - node has been disabled.

To get node account information, make an HTTP call to your `thor-api` port which will look like the following:

```
1 http://<host>:1317/thorchain/nodeaccount/<node address>  
2 http://<host>:1317/thorchain/nodeaccounts
```

□ Emergency Procedures

THORChain is a distributed network. When the network is under attack or a funds-at-risk bug is discovered, Node Operators should react quickly to secure and defend.

❗ Even during emergencies, Node Operators should refrain from doxxing themselves. Staying psuedo-anonymous is critical to ensuring the network is impartial, nuetral and resistant to capture.

Reporting a Bug

If you have discovered a bug, you should immediately DM the team or any other admins. If the bug is deemed to be serious/critical, you will be paid a bounty commensurate to the severity of the issue. Reach out on telegram, twitter, gitlab or discord.

1. Description of the bug
2. Steps to reproduce
3. If funds are at risk

Admin Procedures

Once the bug has been verified, admin should make a decision on the level of response, including any mimic over-rides and announcements:

Critical - Funds At Risk

- **Determine which level of halt is required:**
 - Can a trader siphon out funds by swapping? If yes, `haltTrading` using mimic
 - Can yggdrasil vaults siphon out funds? If yes, `haltInternalTx` using mimic

- Can funds be siphoned out from any vault? If yes, `haltOutboundTx` using `mimir`
- ☐ **Determine what type of announcement is required:**
 - Does the network need coordination from nodes? If yes, announce, with directions.
 - Can the team immediately apply a bug patch with no coordination? If yes, simply announce the halt.
- ☐ **Generate a bug fix**
 - Is there KVStore migration required? If yes, will require migration testing
 - Is there Bifrost/TSS changes? If yes, will require network testing
- ☐ **Test on Mocknet**
- ☐ **Release for Chaosnet**

Major - Funds Not At Risk, but Network At Risk (disruption)

- ☐ **Determine what type of announcement is required:**
 - Does the network need coordination from nodes? If yes, announce, with directions.
 - Can the team immediately apply a bug patch with no coordination? If yes, consider delaying announcement until bug fix ready.
- ☐ **Generate a bug fix**
 - Is there KVStore migration required? If yes, will require migration testing
 - Is there Bifrost/TSS changes? If yes, will require network testing
- ☐ **Test on Mocknet**
- ☐ **Release for Chaosnet**

Minor - Funds Not At Risk, Network Not At Risk

- ☐ **Generate a bug fix**
 - Is there KVStore migration required? If yes, will require migration testing
 - Is there Bifrost/TSS changes? If yes, will require network testing
- ☐ **Deploy to Testnet**
- ☐ **Release for Chaosnet**

Network Upgrades

The network cannot upgrade until 100% of active nodes are on the updated version. This can be accelerated:

1. Naturally, by allowing the network to churn out old nodes
2. Assertive, by waiting until a super-majority has upgraded (demonstrating acceptance of the upgrade) then banning old nodes
3. Forced, by hard-forking out old nodes.

During a natural upgrade cycle, it may take several days to churn out old nodes. If the upgrade is time-critical, the network may elect to ban old nodes. Banning a node will cycle them to be churned, kick them from TSS and eject them from the consensus set. That node will never be able to churn in again, they will need to fully leave, destroy their node, and set up a new one. Hard-forking out old nodes is also a possibility, but comes with significant risk of consensus failures.

Network Recovery

The network will not be able to recover until the upgrade is complete, any mimic over-rides are removed, and TSS is re-synced. Additionally, there may be a backlog of transactions that will need to be processed. This may take some time. If external chain daemons were stopped, re-syncing times may also be a factor.

All wallets and frontends should monitor for any of the halts and automatically go into maintenance mode when invoked.

Deploy - Manual



This guide is incomplete. Refer to [the THORNode GitLab repository](#) for the most current instructions.

First, ensure that you have a recent version of Go installed – at least version `1.13` . [Latest Go versions](#).

Also, be sure to have `GOBIN` in your `PATH` . To ensure this, run—

```
export GOBIN=$GOPATH/bin
```

Next, prepare the local THORNode directory—

```
1 git clone git@gitlab.com:thorchain/thornode.git
2 cd thornode
```

Now there are 2 options for running THORNode—

- [on Linux](#), manually
- [with Docker](#) – for developing with a full local chain

Run THORNode on Linux

Build the binaries—

```
make install
```

Check you've installed `thorcli` and `thord` correctly:

```
1 thorcli help
2 thord help
```

Next, set up the Binance full node. Do this manually [using Binance's documentation](#) or use [a Docker image](#).

Wait until your Binance node is caught up before you continue to the next steps.

Run THORNode with Docker

Use Docker to get a full local mock network for development purposes and to run currently unsupported operating systems, namely Windows.

Go to the Docker directory—

```
cd build/docker
```

Run the mocknet—

```
make reset-mocknet-standalone
```


Run a genesis ceremony with 4 nodes on the mock network—

```
make run-mocknet-genesis
```


Deploy - K8 Cluster

Deploy a Kubernetes cluster

In order to deploy all the different services and provide a high availability environment to operate your node, Kubernetes is the preferred scheduling platform. Any production-grade Kubernetes cluster can be used to run and deploy a THORNode. You need your Kubernetes provider to offer external load balancers services type features. Azure, Digital Ocean, GCE, OpenStack are compatible with external load balancers.

 Terraform is a type of domain-specific language (DSL) used to describe code infrastructure. It is designed to make it easier to create/destroy infrastructure hosted locally or by a provider.

This Terraform deployment will deploy a Kubernetes cluster using your VPS provider credentials and EKS service. The cluster will have autoscaling capabilities, which means you don't have to deal with how many nodes you need to deploy to run your THORNode services.

All the default configurations used in these instructions are for a production environment with enough resources to run your THORNode in good conditions.

Steps

There are three important steps to getting your node set up, deployed and churned in.

1. [Setting up Cluster](#)
2. [Deploying THORNode Services](#)
3. [Joining \("Churning In"\)](#)

Repository Management

Your repository should be organised as follows:

```
1 ./thornode-ops
2   |./cluster-launcher
3   |./node-launcher
```

All of your set up commands are run in `cluster-launcher` and all of your deploying/joining/managing/leaving commands are run from `node-launcher`

Running Two or More Nodes



To prevent a catastrophic mistake in handling multiple nodes, set them up on different machines, or use different user profiles on your machine, or in the least, use different repos:

```
1 ./thornode-ops
2   |./cluster-launcher
3   |./node-launcher
4 ./thornode-ops2
5   |./cluster-launcher
6   |./node-launcher
```

All of your commands can now be run separately.



It is heavily advised to not set up nodes on the same provider. Deploy 1 node on Azure, 1 node on Digital Ocean etc.

Setup - Linode

Deploy a Kubernetes cluster in Linode using LKE service.

Requirements

1. a Linode account
2. `linode-cli` and linode credentials configured
3. `kubectl`

 **LINUX/MAC is the preferred method of setup.**

Windows should choose either:

1. Deploy a THORNode from a Linux VPS.
2. Use Windows Subsystem for Linux - <https://docs.microsoft.com/en-us/windows/wsl/about>****

linode-cli

To install the linode-cli (Linode CLI), follow [these instructions](#).

You need to have pip (python) on your system.

```
pip install linode-cli --upgrade
```

Create a Linode API token for your account with read and write access from your [profile page](#). The token string is only displayed once, so save it in a safe place.

Use the API token to grant linode-cli access to your Linode account. Pass in the token string when prompted by linode-cli.

```
linode-cli
```

kubectl

To install the kubectl (Kubernetes CLI), follow [these instructions](#) or choose a package manager based on your operating system.

MacOS:

Use the package manager [homebrew](#) to install kubectl.

```
brew install kubernetes-cli
```

Windows:

Use the package manager [Chocolatey](#) to install kubectl.

```
choco install kubernetes-cli
```

wget

To install the wget, follow [these instructions](#) or choose a package manager based on your operating system.

MacOS:

Use the package manager [homebrew](#) to install wget.

```
brew install wget
```

Windows:

Use the package manager [Chocolatey](#) to install wget.

```
choco install wget
```

Deploy Kubernetes Cluster

Use the commands below to deploy a Kubernetes cluster.

You can run the make command that automates those command for you like this:

```
make linode
```

Or manually run each commands:

```
1 cd linode/  
2 terraform init  
3 terraform plan # to see the plan  
4 terraform apply
```

Configure kubectl

Now that you've provisioned your Kubernetes cluster, you need to configure kubectl.

To configure authentication from the command line, use the following command, substituting the ID of your cluster.

```
1 # Store it - method #1
2 jq -r ".resources[].instances[].attributes.kubeconfig" linode/terraform.tf
3
4 # Store it - method #2
5 linode-cli lke kubeconfig-view <use_your_cluster_id> > ~/.kube/config-linode
6
7 # Merge it and set current context
8 KUBECONFIG=~/.kube/config:~/.kube/config-linode kubectl config view --flat
9
10 # Or just view it - method #1
11 jq -r ".resources[].instances[].attributes.kubeconfig" linode/terraform.tf
12 # Or just view it - method #2
13 linode-cli lke kubeconfig-view <use_your_cluster_id>
```

Note: If the above `linode-cli` command is broken you can download the file from the web [dashboard](#) for the respective cluster.

This replaces the existing configuration at `~/.kube/config`.

Once done, you can check your cluster is responding correctly by running the command:

```
1 kubectl version
2 kubectl get nodes
```

Clean up your workspace

To destroy and remove previously created resources, you can run the command below.

```
make destroy-linode
```

Or run the commands manually:

```
1 cd linode/  
2 terraform destroy
```


Setup - Azure

Deploy a Kubernetes cluster in Azure using AKS service.

Requirements

1. Azure account
2. `az` and Azure credentials configured
3. `kubectl`

 **LINUX/MAC is the preferred method of setup.**

Windows should choose either:

1. Deploy a THORNode from a Linux VPS.
2. Use Windows Subsystem for Linux - <https://docs.microsoft.com/en-us/windows/wsl/about>****

Steps

Firstly, clone and enter the [cluster-launcher repository](#). All commands in this section are to be run inside this repo.

```
1 git clone https://gitlab.com/thorchain/devops/cluster-launcher
2 cd cluster-launcher
```

Then install the [terraform CLI](#):

LINUX/MAC

Install Terraform:

```
brew install terraform
```

Azure CLI

The [Azure CLI](#) allows you to manage your Azure services.

LINUX/MAC

Use the package manager [homebrew](#) to install the Azure CLI.

```
1 brew install azure-cli
2 az login
```



You will be asked for your Personal Access Token with read/write privileges (retrieve from API Panel from the Azure web console.)

API -> Tokens/Keys -> Create Token.

Make sure you handle your secrets securely!

Kubernetes Control Tool

You must install and configure the Kubernetes CLI tool (**kubectl**). ****To install kubectl****, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **kubectl**.

```
brew install kubernetes-cli
```

wget && jq

You also need **wget** and **jq**, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **wget** and **jq** *Note: You most likely have these installed already.*

```
1 brew install wget
2 brew install jq
```

Deploy Kubernetes Cluster


Use the commands below to deploy an AKS cluster:

```
make azure
```

During the deploy, you will be asked to enter information about your cluster:

```
1 var.location
2   The location where the Managed Kubernetes Cluster should be created
3
4   Enter a value: eastus2
5
6 var.name
7   The base name used for all resources
8
9   Enter a value: tc-k8s
```

- Location – `az account list-locations -o table`
- Name
- Confirm `yes`

 Deploying a cluster takes ~15 minutes

CONFIGURE

Now that you've provisioned your AKS cluster, you need to configure **kubectl**. Customize the following command with your cluster name and resource group. It will get the access credentials for your cluster and automatically configure kubectl.

```
az aks get-credentials -a -g <resource_group> -n <cluster_name>
```

This replaces the existing configuration at `~/.kube/config`.

Once done, you can check if your cluster is responding correctly by running the following commands.

```
1 kubectl version
2 kubectl get nodes
```

You are now ready to deploy a THORNode.

Setup - Hetzner Bare Metal

Checkout the repository [source](#) to manage a cluster of dedicated servers on Hetzner.

The scripts in this repository will setup and maintain one or more [kubernetes](#) clusters consisting of dedicated [Hetzner](#) servers. Each cluster will also be provisioned to operate as a node in the [THORCHain](#) network.

Executing the scripts in combination with some manual procedures will get you highly available, secure clusters with the following features on bare metal.

- [Kubespray](#) (based)
- Internal NVMe storage ([Ceph/Rook](#))
- Virtual LAN (also over multiple locations) ([Calico](#))
- Load Balancing ([MetalLB](#))

Preparations

Servers

Acquire a couple of [servers](#) as the basis for a cluster (`AX41-NVME` 's are working well for instance). Visit the [admin panel](#) and name the servers appropriately.

```
1 tc-k8s-node1
2 tc-k8s-node2
3 tc-k8s-node3
4 ...
5
6 tc-k8s-master1
7 tc-k8s-master2
8 tc-k8s-worker1
9 tc-k8s-worker2
10 tc-k8s-worker3
11 ...
```

Refer to the [reset procedure](#) to properly initialize them.

vSwitch

Create a [vSwitch](#) and order an appropriate subnet (it may take a while to show up after the order). Give the vSwitch a name (i.e. `tc-k8s-net`) and assign this vSwitch to the servers.

Checkout the [docs](#) for help.

Usage

Clone this repository, `cd` into it and download kubespray.

```
git submodule init && git submodule update
```

Create a Python virtual environment or similar.

```
1 # Optional
2 virtualenv -p python3 venv
```

Install dependencies required by Python and Ansible Galaxy.

```
1 pip install -r requirements.python.txt
2 ansible-galaxy install -r requirements.ansible.yml
```

Note: [Mitogen](#) does not work with ansible collections and the strategy must be changed (i.e. `strategy: linear`).

Provisioning

Create a deployment environment inventory file for each cluster you want to manage.

```
1 cp hosts.example inventory/production.yml
2 cp hosts.example inventory/test.yml
3 cp hosts.example inventory/environment.yml
4 ...
5
6 cp hosts.example inventory/production-01.yml
7 cp hosts.example inventory/production-02.yml
8 ...
9
10 cp hosts.example inventory/production-helsinki.yml
11 cp hosts.example inventory/whatever.yml
```

Edit the inventory file with your server ip's and network information and customize everything to your needs.

```
1 # Manage a cluster
2 ansible-playbook cluster.init.yml -i inventory/environment.yml
3 ansible-playbook --become --become-user=root kubespray/cluster.yml -i inve
4 ansible-playbook cluster.finish.yml -i inventory/environment.yml
5
6 # Run custom playbooks
7 ansible-playbook private-cluster.yml -i inventory/environment.yml
8 ansible-playbook private-test-cluster.yml -i inventory/environment.yml
9 ansible-playbook private-whatever-cluster.yml -i inventory/environment.yml
```

Check [this](#) out for more playbooks on cluster management.

THORChain

In order for the cluster to operate as a node in the THORChain network deploy as instructed [here](#). You can also refer to the [node-launcher repository](#), if necessary, or the THORChain [documentation](#) as a whole.

Resetting the bare metal servers

This will install and use Ubuntu 20.04 on only one of the two internal NVMe drives. The unused ones will be used for persistent storage with ceph/rook. You can check the internal drive setup with `lsblk`. Change it accordingly in the command shown above when necessary.

Manually

Visit the [console](#) and put each server of the cluster into rescue mode. Then execute the following script.

```
installimage -a -r no -i images/Ubuntu-2004-focal-64-minimal.tar.gz -p /:ext
```

Automatically

Create a pristine state by running the playbooks in sequence.

```
1 ansible-playbook server.rescue.yml -i inventory/environment.yml
2 ansible-playbook server.bootstrap.yml -i inventory/environment.yml
```

Instantiation

Instantiate the servers.

```
ansible-playbook server.instantiate.yml -i inventory/environment.yml
```

Setup - Google Cloud

Deploy a Kubernetes cluster in GCP using GKE service.

Requirements

1. GCP account
2. `gcloud` and GCP credentials configured
3. `kubectl`

 **LINUX/MAC is the preferred method of setup.**

Windows should choose either:

1. Deploy a THORNode from a Linux VPS.
2. Use Windows Subsystem for Linux - <https://docs.microsoft.com/en-us/windows/wsl/about>****

Steps

Firstly, clone and enter the [cluster-launcher repository](#). All commands in this section are to be run inside this repo.

```
1 git clone https://gitlab.com/thorchain/devops/cluster-launcher
2 cd cluster-launcher
```

Then install the [terraform CLI](#):

LINUX/MAC

Install Terraform:

```
brew install terraform
```

gcloud CLI

The [gcloud CLI](#) allows you to manage your GCP services.

LINUX/MAC

Use the package manager [homebrew](#) to install the GCP CLI.

```
brew install google-cloud-sdk
```

After the installation perform the steps outlined below. This will authorize the SDK to access GCP using your user account credentials and add the SDK to your PATH. It requires you to login and select the project you want to work in. Then add your account to the Application Default Credentials (ADC). This will allow Terraform to access these credentials to provision resources on GCP. Finally, you need to enable the Compute Engine and Kubernetes Engine API services for your GCP project.

```
1 gcloud init
2 gcloud auth application-default login
3 gcloud services enable compute.googleapis.com
4 gcloud services enable container.googleapis.com
```



You will be asked for your Personal Access Token with read/write privileges (retrieve from API Panel from the GCP web console.)

API -> Tokens/Keys -> Create Token.

Make sure you handle your secrets securely!

Kubernetes Control Tool

You must install and configure the Kubernetes CLI tool (**kubectl**). ****To install kubectl****, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **kubectl**.

```
brew install kubernetes-cli
```

wget & jq

You also need **wget** and **jq**, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **wget** and **jq** *Note: You most likely have these installed already.*

```
1 brew install wget
2 brew install jq
```

Deploy Kubernetes Cluster


Use the commands below to deploy an GKE cluster:

```
make gcp
```

During the deploy, you will be asked to enter information about your cluster:

```
1 var.project_id
2   Project ID
3
4   Enter a value: tc-k8s-123456
5
6 var.zone
7   GCP zone in region
8
9   Enter a value: us-east1-d
```

- Project ID
- Zone – `gcloud compute zones list`
- Confirm `yes`

 Deploying a cluster takes ~15 minutes

CONFIGURE

Now that you've provisioned your GKE cluster, you need to configure **kubect**l. The following command will get the access credentials for your cluster and automatically configure kubect

```
(cd gcp && gcloud container clusters get-credentials $(terraform output clus
```

This replaces the existing configuration at ~/.kube/config.

Once done, you can check if your cluster is responding correctly by running the following commands.

```
1  kubectl version
2  kubectl get nodes
```

You are now ready to deploy a THORNode.

Setup - HCloud

Deploy an unmanaged Kubernetes cluster in hcloud

⚠ This approach is only recommended for experienced operators because the kubernetes control plane among other things needs to be managed manually.

Requirements

1. hcloud account
2. `hcloud` and hcloud credentials configured
3. `kubectl`
4. `ansible`

⚠ **LINUX/MAC is the preferred method of setup.**

Windows should choose either:

1. **Deploy a THORNode from a Linux VPS.**
2. **Use Windows Subsystem for Linux - <https://docs.microsoft.com/en-us/windows/wsl/about>******

Steps

Firstly, clone and enter the [cluster-launcher repository](#). All commands in this section are to be run inside this repo.

```
1 git clone https://gitlab.com/thorchain/devops/cluster-launcher
```

```
2 cd cluster-launcher
```

Then install the [terraform CLI](#):

LINUX/MAC

Install Terraform:

```
brew install terraform
```


hcloud CLI

The [hcloud CLI](#) allows you to manage your hcloud services.

LINUX/MAC

Use the package manager [homebrew](#) to install the hcloud CLI.

```
1 brew install hcloud
2 hcloud context create <project_name>
```

 You will be asked for you Personal Access Token with read/write priveleges (retrieve from API Panel from the hcloud web console.)

API -> Tokens/Keys -> Create Token.

Make sure you handle your secrets securely!

Kubernetes Control Tool

You must install and configure the Kubernetes CLI tool (**kubectl**). ****To install kubectl**** , follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **kubectl**.

```
brew install kubernetes-cli
```

wget & jq

You also need **wget** and **jq**, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **wget** and **jq** *Note: You most likely have these installed already.*

```
1 brew install wget
2 brew install jq
```

Environment

Initialize the git submodule.

```
git submodule update --init
```

Use `direnv`, `venv` or whatever you prefer to manage a python environment inside the `hcloud` folder.

```
1 # Optional
2 (cd hcloud && direnv allow)
3
4 # Optional
5 (cd hcloud && virtualenv -p python3 venv)
```

Install dependencies required by Python and Ansible Galaxy.

```
1 (cd hcloud && pip install -r ansible/requirements.txt)
2 (cd hcloud && ansible-galaxy install -r ansible/requirements.yml)
```

Deploy Kubernetes Cluster


Use the commands below to deploy an hcloud cluster:

```
make hcloud
```

During the deploy, you will be asked to enter information about your cluster:

```
1 var.name
2   The base name used for all resources
3
4   Enter a value: tc-k8s
5
6 var.token
7   Hetzner Cloud API token
8
9   Enter a value: <secret>
10
11 var.user_name
12   The admin user name for the nodes
13
14   Enter a value: admin
```

- Name
- Token
- Confirm

 Deploying a cluster takes ~15 minutes

Quotas

If necessary, request a quota increase [here](#).

CONFIGURE

Now that you've provisioned your hcloud cluster, you need to configure **kubect**l. Customize the following command with your cluster name and resource group. It will get the access credentials for your cluster and automatically configure kubect

```
1 (cd hcloud && scp $(terraform output -raw hcloud_config) ~/.kube/config-hc
2
3 # Merge it and set current context
```

```
4 KUBECONFIG=~/.kube/config:~/.kube/config-hcloud kubectl config view --flat
5
6 kubectl version
```

You are now ready to deploy a THORNode.

Setup - Digital Ocean

Deploy a Kubernetes cluster in DO using EKS service.

Requirements

1. DO account
2. `doctl` and DO credentials configured
3. `kubectl`

 **LINUX/MAC is the preferred method of setup.**

Windows should choose either:

1. Deploy a THORNode from a Linux VPS.
2. Use Windows Subsystem for Linux - <https://docs.microsoft.com/en-us/windows/wsl/about>****

Steps

Firstly, clone and enter the [cluster-launcher repository](#). All commands in this section are to be run inside this repo.

```
1 git clone https://gitlab.com/thorchain/devops/cluster-launcher
2 cd cluster-launcher
```

Then install the [terraform CLI](#):

LINUX/MAC

Install Terraform:

```
brew install terraform
```

DOCLI

The [Digital Ocean Control tool](#) allows you to manage your DO services.

LINUX/MAC

Use the package manager [homebrew](#) to install the DO CTL.

```
1 brew install doctl
2 doctl auth init --context <NAME>
3 doctl auth switch --context <NAME>
4 doctl account get
```



You will be asked for your Personal Access Token with read/write privileges (retrieve from API Panel from the Digital Ocean web console.)

API -> Tokens/Keys -> Create Token.

Make sure you handle your secrets securely!

Kubernetes Control Tool

You must install and configure the Kubernetes CLI tool (**kubectl**). ****To install kubectl****, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **kubectl**.

```
brew install kubernetes-cli
```

wget && jq

You also need **wget** and **jq**, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **wget** and **jq** *Note: You most likely have these installed already.*

```
1 brew install wget
2 brew install jq
```

Deploy Kubernetes Cluster

Use the commands below to deploy a DOKS cluster:

```
make do
```

During the deploy, you will be asked to enter information about your cluster:

```
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
var.cluster_name
    EKS cluster name

Enter a value: █
```

- Name
- DO Region – see valid [List of Regions](#) (use lower-case)
- Confirm


Other Product Availability

Product	NYC1	NYC2	NYC3	AMS2	AMS3	SFO1	SFO2	SFO3	SGP1	LON1	FRA1	TOR1	BLR1
Kubernetes	◆		◆		◆		◆	◆	◆	◆	◆	◆	◆
Volumes	◆		◆		◆		◆	◆	◆	◆	◆	◆	◆
Spaces			◆		◆		◆		◆		◆		
Load Balancers	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆

Kubernetes Availability (note, use lower-case in the terminal)

Final success message:

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

 Deploying a cluster takes ~10 minutes

CONFIGURE

Now that you've provisioned your DOKS cluster, you need to configure **kubectl**. Customize the following command with your cluster name and region.

```
1 doctl kubernetes cluster kubeconfig save <use_your_cluster_name>
2 kubectl version
```

If successful, you will see:

```
1 Notice: Adding cluster credentials to kubeconfig file found in "/home/user
2 Notice: Setting current-context to do-<region_name>-<cluster_name>
```

Test this configuration,

```
1 $ kubectl version
2 Client Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.6",
3 Server Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.6",
```

To verify, run this, and check the status is "Ready":

```
1 kubectl get nodes
2
3 NAME                                STATUS    ROLES    AGE    VERSION
4 <cluster_name>-pool-5xhc1           READY    <none>   6m     v1.18.6
```

You are now ready to deploy a THORNode.

Setup - AWS

Deploy a Kubernetes cluster in AWS using EKS service.

Requirements

1. AWS account
2. CLI and AWS credentials configured
3. AWS IAM Authenticator
4. `kubectl`
5. `wget` (required for EKS module)

 **LINUX/MAC is the preferred method of setup.**

Windows should choose either:

1. Deploy a THORNode from a Linux VPS.
2. Use Windows Subsystem for Linux - <https://docs.microsoft.com/en-us/windows/wsl/about>

Steps

Firstly, clone and enter the [cluster-launcher repository](#). All commands in this section are to be run inside this repo.

```
1 git clone https://gitlab.com/thorchain/devops/cluster-launcher
2 cd cluster-launcher
```

Then install the [terraform CLI](#):

LINUX/MAC

Install Terraform:

```
brew install terraform
```

AWS CLI

In order for Terraform to run operations on your behalf, you must install and configure the AWS CLI tool. To install the AWS CLI, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install the AWS CLI.

```
1 brew install awscli
2 aws configure
```



You will be asked for you AWS access credentials (retrieve from AWS IAM from the AWS web console.)

IAM -> User -> Security Credentials -> Create Access Key.

Make sure you handle your secrets securely!

AWS IAM Authenticator

You also must install and configure the **AWS IAM Authenticator** tool. To install, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install the **AWS IAM Authenticator**.

```
brew install aws-iam-authenticator
```

Kubernetes Control Tool

You must install and configure the Kubernetes CLI tool (**kubectl**). To install **kubectl**, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **kubectl**.

```
brew install kubernetes-cli
```

wget && jq

You also need **wget** and **jq**, follow [these instructions](#), or choose a package manager based on your operating system.

LINUX/MAC

Use the package manager [homebrew](#) to install **wget** and **jq**

Note: You most likely have these installed already.

```
1 brew install wget
2 brew install jq
```

Deploy Kubernetes Cluster

Use the commands below to deploy an AWS EKS cluster. You can run the make command that automates those command for you like this:

```
make aws
```

During the deploy, you will be asked to enter information about your cluster:

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.
```

```
If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

```
var.cluster_name
    EKS cluster name
```

```
Enter a value: █
```

- Name
- AWS Region – see valid [List of Regions](#)
- Confirm

Region Name	Code
US East (Ohio)	us-east-2
US East (N. Virginia)	us-east-1
US West (N. California)	us-west-1
US West (Oregon)	us-west-2
Africa (Cape Town)	af-south-1
Asia Pacific (Hong Kong)	ap-east-1
Asia Pacific (Mumbai)	ap-south-1
Asia Pacific (Osaka-Local)	ap-northeast-3
Asia Pacific (Seoul)	ap-northeast-2
Asia Pacific (Singapore)	ap-southeast-1
Asia Pacific (Sydney)	ap-southeast-2
Asia Pacific (Tokyo)	ap-northeast-1
Canada (Central)	ca-central-1
China (Beijing)	cn-north-1
China (Ningxia)	cn-northwest-1
Europe (Frankfurt)	eu-central-1


Regions

Services Offered:	Northern Virginia	Ohio	Oregon	Northern California	Montreal	São Paulo	AWS GovCloud (US-West)	AWS GovCloud (US-East)
Amazon Elastic Kubernetes Service (EKS)	✓	✓	✓		✓	✓	✓	✓


Note: AWS EKS is not available in some regions

Final success message:

Apply complete! Resources: 30 added, 0 changed, 0 destroyed.

-  If you are a **returning** node operator and you wish to use the same node name, the Cloudwatch log files from your previous session will block this step. You need to manually delete the logs from your console:

Cloudwatch / Cloudwatch Logs / Log Groups -> "delete"

-  Deploying a cluster takes ~10 minutes

CONFIGURE

Now that you've provisioned your EKS cluster, you need to configure **kubect**l. Customize the following command with your cluster name and region.

```
1 aws eks --region <cluster_region> update-kubeconfig --name <cluster_name>
2 kubectl version
```

If successful, you will see:

```
Added new context ..... <details>
```

To verify, run this, and check the status is "Ready":


```
1 kubectl get nodes
2
3 NAME                                STATUS    ROLES    AGE    VERSION
4 ip-10-0-49-192.ec2.internal        Ready    <none>   4m16s  v1.16.12-eks-904af
```

You are now ready to deploy a THORNode.

Deploying

Deploy THORNode services

Now you have a Kubernetes cluster ready to use, you can install the THORNode services.

-  Helm charts are the defacto and currently easiest and simple way to package and deploy Kubernetes application. The team created different Helm charts to help to deploy all the necessary services. Please retrieve the source files from the Git repository here to follow the instructions below:

<https://gitlab.com/thorchain/devops/node-launcher>

Requirements

- Running Kubernetes cluster
- Kubectl configured, ready and connected to running cluster

-  If you came here from the Setup page, you are already good to go.

Steps

Clone the `node-launcher` repo. All commands in this section are to be run inside of this repo.

```
1 git clone https://gitlab.com/thorchain/devops/node-launcher
2 cd node-launcher
```


Install Helm 3

Install Helm 3 if not already available on your current machine:

```
make helm
```

Tools

Deploy

To deploy all tools, metrics, logs management, Kubernetes Dashboard, run the command below.

```
make tools
```

Destroy

To destroy all those resources run the command below.

```
make destroy-tools
```

If you are successful, you will see the following message:

```
Installing Helm repos
"stable" has been added to your repositories
"kubernetes-dashboard" has been added to your repositories
Installing Logs Management
Release "elastic" does not exist. Installing it now.
NAME: elastic
LAST DEPLOYED: Thu Jul 16 12:00:39 2020
NAMESPACE: elastic-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
Waiting for services to be ready...
pod/elastic-operator-0 condition met
pod/filebeat-ss9s7 condition met
Installing Metrics
Release "metrics-server" does not exist. Installing it now.
NAME: metrics-server
LAST DEPLOYED: Thu Jul 16 12:03:28 2020
NAMESPACE: prometheus-system
STATUS: deployed
REVISION: 1
NOTES:
The metric server has been deployed.
```


If there are any errors, they are typically fixed by running the command again.

Deploy THORNode

It is important to deploy the tools first before deploying the THORNode services as some services will have metrics configuration that would fail and stop the THORNode deployment.

You have multiple commands available to deploy different configurations of THORNode. You can deploy testnet, chaosnet and mainnet. The commands deploy the umbrella chart `thornode-stack` in the background in the Kubernetes namespace `thornode` (or `thornode-testnet` for testnet) by default.

```
make install
```

 Deploying a THORNode will take 1 day for every 3 months of ledger history, since it will validate every block. THORNodes are "full nodes", not light clients.

If successful, you will see the following:


```
Creating THORNode Namespace
namespace/thornode created
Generating THORNode Mnemonic phrase
secret/thornode-mnemonic created
Installing THORNode
Release "thornode" does not exist. Installing it now.
NAME: thornode
LAST DEPLOYED: Thu Jul 16 12:11:53 2020
NAMESPACE: thornode
STATUS: deployed
REVISION: 1
```

You are now ready to join the network:

→ [Joining](#)

</thornodes/joining>

Debugging

 Set `thornode` to be your default namespace so you don't need to type `-n thornode` each time:

```
kubectl config set-context --current --namespace=thornode
```

Use the following useful commands to view and debug accordingly. You should see everything running and active. Logs can be retrieved to find errors:

```
1 kubectl get pods -n thornode
2 kubectl get pods --all-namespaces
3 kubectl logs -f <pod> -n thornode
```

Kubernetes should automatically restart any service, but you can force a restart by running:

```
kubectl delete pod <pod> -n thornode
```

⚠ Note, to expedite syncing external chains, it is feasible to continually delete the pod that has the slow-syncing chain daemon (eg, binance-daemon-xxx).

Killing it will automatically restart it with free resources and syncing is notably faster. You can check sync status by viewing logs for the client to find the synced chain tip and comparing it with the real-world blockheight, ("xxx" is your unique ID):

```
kubectl logs -f binance-daemon-xxx -n thornode
```

i Get real-world blockheights on the external blockchain explorers, eg:

<https://testnet-explorer.binance.org/>

<https://explorer.binance.org/>

CHART SUMMARY

THORNode full stack umbrella chart

- **thornode:** Umbrella chart packaging all services needed to run a fullnode or validator THORNode.

This should be the only chart used to run THORNode stack unless you know what you are doing and want to run each chart separately (not recommended).

THORNode services:

- **thor-daemon**: THORNode daemon
- **thor-api**: THORNode API
- **thor-gateway**: THORNode gateway proxy to get a single IP address for multiple deployments
- **bifrost**: Bifrost service
- **midgard**: Midgard API service

External services:

- **binance-daemon**: Binance fullnode daemon
- **bitcoin-daemon**: Bitcoin fullnode daemon
- **ethereum-daemon**: Ethereum fullnode daemon
- **chain-daemon**: as required for supported chains

Tools

- **elastic**: ELK stack, deprecated. Use elastic-operator chart
- **elastic-operator**: ELK stack using operator for logs management
- **prometheus**: Prometheus stack for metrics
- **loki**: Loki stack for logs
- **kubernetes-dashboard**: Kubernetes dashboard

Joining

Joining THORChain

Now that you have a THORNode deployed in your Kubernetes cluster, you need to start operating your node to join the network.

There are a couple of steps to follow to do so.

1. Check your current node status

The first step would be to make sure your deployment was successful and your node is running correctly. ******To check the current status of your node, you can run the command status from the `node-launcher` repository on your terminal:

```
make status
```


You will get an output along those lines, the example below is for a testnet node:

```
1  _____ _ _ _ _ _ _ _
2  / _ _ / // / _ \ / _ \ | / _ _ _ _ / / _
3  / / / _ / // / , _ / _ \ _ / _ )
4  / _ / _ // _ \ _ _ / _ \ | _ \ _ \ _ , _ \ _ /
5  ADDRESS      thor13hh6qyj0xgw0gv7qpay8thfucxw8hqkved9vr2
6  IP
7  VERSION      0.0.0
8  STATUS       Unknown
9  BOND         0.00
10 REWARDS      0.00
11 SLASH        0
12 PREFLIGHT    { "status": "Standby", "reason": "node account has invalid reg
13 API          http://:1317/thorchain/doc/
14 RPC          http://:27147
15 MIDGARD      http://:8080/v2/doc
16 CHAIN        SYNC      BLOCKS
17 THOR         55.250%    37,097/67,144
18 BNB          99.915%    155,426,471/155,559,013
```

19	BTC	2.399%	227,986/678,340
20	ETH	7.764%	947,182/12,199,994
21	LTC	0.012%	6,526/1,818,000
22	BCH	2.293%	197,340/682,404

Your node is running but as you can see in the `Preflight` section, your node is not yet ready to be churned in and currently is in standby status, since your node has no IP address setup yet.

But to be able to set up the node IP address, you first need to get it registered in the chain by sending your BOND.


 Before sending the BOND, verify that your THORNode is fully synced with connected chains. Connected chains such as Bitcoin, may take a day to sync. If you join THORChain without fully syncing a connected chain, you will immediately get slashed for missing observations, and lose money.

2 - Send a small BOND (recommend 100-1000)

1) You will do a "Deposit" transaction. There is no destination address – use an appropriate wallet. The Bond is locked in a module controlled by the state machine.

2) Deposit your BOND using the memo `BOND:<thornode-address>` (or use an appropriate GUI that does this memo for you). Start small, the bond will be picked up.

[< BACK](#)



DEPOSIT

INTERACT WITH THORCHAIN

AVAILABLE ACTIONS

[BOND](#)
[UNBOND](#)
[LEAVE](#)
[CUSTOM](#)

THORADDRESS

AMOUNT

MAX R 3,112.18200098

SEND

Bonding using BOND option in ASGARDEX

Give the network 3-5 seconds to pick up your bond. To verify it has received your bond, run the following:

```
curl http://thornode.thorchain.info/thorchain/node/<node-address>
```

If you run `make status` again, you should see this:

```

1  -----  -----  ---  -  --
2  /_  _/  //  /  _  \  _  \  |  /  _  _  /  /  _
3  /  /  _  /  /  /  ,  _  /  _  \  _  /  -  )
4  /_  /  _  /  _  /  _  /  _  /  _  /  _  /  _  /
5  ADDRESS      thor13hh6qyj0xgw0gv7qpay8thfucxw8hqkved9vr2
6  IP
7  VERSION      0.0.0
8  STATUS       Whitelisted
9  BOND         1000.00
10 REWARDS      0.00
11 SLASH        0
12 PREFLIGHT    { "status": "Standby", "reason": "node account has invalid reg
13 API          http://:1317/thorchain/doc/
14 RPC          http://:27147
15 MIDGARD      http://:8080/v2/doc

```


16	CHAIN	SYNC	BLOCKS
17	THOR	55.250%	37,097/67,144
18	BNB	99.915%	155,426,471/155,559,013
19	BTC	2.399%	227,986/678,340
20	ETH	7.764%	947,182/12,199,994
21	LTC	0.012%	6,526/1,818,000
22	BCH	2.293%	197,340/682,404


As you can see, it is in standby but does not have an IP registered yet. This is needed for peer discovery.

3 - Setup Node IP Address

You must tell THORChain your IP-Address for its address book and seed-service to run properly:

```
make set-ip-address
```

If you run the status command again, you should now see a different message for the Preflight section saying you need to set your node keys.

 Once your IP address has been registered for discovery, you can use your own host for queries.

4 - Setup Node keys

Tell THORChain about your public keys for signing sessions:

```
make set-node-keys
```

If you run the status command again, you should now see that your node is in status “ready” and is now ready to be churned in the next rotation.

5 - Set Version

Make sure your node broadcasts its latest version, else you won't churn in since THORChain enforces a version requirement. This version will appear in your `make status`. If you are on `0.0.0` then you haven't set your version:

```
make set-version
```

6 - Send Final Bond

If you followed steps 1-5 above, your preflight will be saying:

```
PREFLIGHT { "status": "Standby", "reason": "node account does not have min
```

To address this, send the remaining bond, that is higher than the minimum bond. You can find that quantity on <https://thornode.thorchain.info/thorchain/constants> and look for `MinimumBondInRune`. If you finally run `make status` you should see this, with keyword **"Ready"**:

```
1  _____ _ _ _ _ _ _ _
2  /_ _/ // / _ \ / | / _ _ _ / _
3  / / / _ / / / , _ / _ \ / _ )
4  /_ / _// _\ _ _ / _ / _ \ _ \ _ \
5  ADDRESS      thor13hh6qyj0xgw0gv7qpay8thfucxw8hqkved9vr2
6  IP           1.2.3.4
7  VERSION      0.38.0
8  STATUS       Standby
9  BOND         1,000,001.00
10 REWARDS      0.00
11 SLASH        0
12 PREFLIGHT    { "status": "Ready", "reason": "OK", "code": 0 }
13 API          http://1.2.3.4:1317/thorchain/doc/
```

```
14 RPC      http://1.2.3.4:27147
15 MIDGARD  http://1.2.3.4:8080/v2/doc
16 CHAIN    SYNC      BLOCKS
17 THOR     100.000%    67,144/67,144
18 BNB      100.000%    155,559,013/155,559,013
19 BTC      100.000%    678,340/678,340
20 ETH      100.000%    12,199,994/12,199,994
21 LTC      100.000%    1,818,000/1,818,000
22 BCH      100.000%    682,404/682,404
```

Bonding The Right Amount

Although your node is ready to be churned in, it doesn't mean it will be the next one to be selected since someone else could have posted a higher bond than you. To maximise chances of a quick entry, monitor Midgard to see what everyone else is bonding and try to outbid them. Keep an eye on `maximumStandbyBond` and make sure you are bonding that amount.

```
1 curl http://52.221.153.64:8080/v1/network | json_pp
2
3 resp:
4   "bondMetrics" : {
5     "minimumActiveBond" : "10001000000000",
6     "medianStandbyBond" : "10100000000000",
7     "medianActiveBond" : "15001000000000",
8     "averageStandbyBond" : "10100000000000",
9     "maximumActiveBond" : "15001000000000",
10    "averageActiveBond" : "12006800000000",
11    "maximumStandbyBond" : "10100000000000",
12    "totalStandbyBond" : "10100000000000",
13    "totalActiveBond" : "60034000000000",
14    "minimumStandbyBond" : "10100000000000"
15  }
```

The endpoint will show data on average, median, total, minimum and maximum bond amounts. For fastest entry, bond higher than the current maximum.



RUNE is always displayed in 1e8 format, 100000000 = 1 RUNE

Bonding More

At any time during standby, you can bond more by making an additional BOND transaction with memo:

```
BOND:<thornode-address>
```

You can also [remove some of your bond](#) whilst you are on standby, using the UNBOND memo.

Managing

THORNode commands

The Makefile provide different commands to help you operate your THORNode.

There are two types of make commands, READ and WRITE.

READ COMMANDS

Read commands simply read your node state and doesn't commit any transactions.

STATUS

To get information about your node on how to connect to services or its IP, run the command below. You will also get your node address and the vault address where you will need to send your bond.

```
make status
```

SHELL

Opens a shell into your `thor-daemon` deployment: From within that shell you have access to the `thorcli` command.

```
make shell
```

LOGS

Display stream of logs of THORNode deployment:

```
make logs
```

MNEMONIC

This will print your node mnemonic (phrase). Use this to ever rescue your node funds if something goes wrong.

Note: This phrase should only be used "in anger". This is your node "hot vault", also referred to as its yggdrasil vault, which allows the network to delegate swaps for faster execution. You will be slashed significantly if any funds are moved from this vault, since it is monitored by the THORChain network. Your bond is held at ransom in order to prevent you from stealing funds from this vault. Your bond will always be more valuable than funds on this vault, so you have no economic reason to touch these funds.

```
make mnemonic
```

PASSWORD

A keystore file that secures your private keys is also stored on the THORNode. The password that is used to decrypt it can be printed by the following command

```
make password
```

RESTART

Restart a THORNode deployment service selected:

```
make restart
```

WRITE COMMANDS

Write commands actually build and write transactions into the underlying statechain. They cost RUNE from your bond, currently 1 RUNE, but you can check this on the `/constants` endpoint "CLICOSTINRUNE". This will post state in the chain which will be now updated globally. The RUNE fee is to prevent DOS attacks.

NODE-KEYS

Send a `set-node-keys` to your node, which will set your node keys automatically for you by retrieving them directly from the `thor-daemon` deployment.

```
make set-node-keys
```

IP ADDRESS

Send a `set-ip-address` to your node, which will set your node ip address automatically for you by retrieving the load balancer deployed directly.

```
make set-ip-address
```


VERSION

In order to update your THORNode to a new version, you will need to update the docker tag image used in your deployments. Depending on your choice of deployment this can be done differently.

For Kubernetes deployments, you can edit the deployments of the different services you want to update using the commands below.

To update your `thor-daemon`, `thor-api` and `bifrost` deployment images to version 0.2.0:

```
1 kubectl set image deployment/thor-daemon thor-daemon=registry.gitlab.com/thor-node/thor-daemon:0.2.0
2 kubectl set image deployment/thor-api thor-api=registry.gitlab.com/thor-node/thor-api:0.2.0
3 kubectl set image deployment/bifrost bifrost=registry.gitlab.com/thor-node/bifrost:0.2.0
```

To update your `midgard` deployment image to version 0.2.0

```
kubectl set image deployment/midgard midgard=registry.gitlab.com/thor-node/midgard:0.2.0
```

You can then follow the deployments restarting status either by checking your Kubernetes dashboard or using the CLI command below:

```
kubectl get deployment/thor-daemon
```

Once the deployments are all in the ready state again, you need to broadcast to the network that you are running a new version using the command below:

```
make set-version
```

Tools

Note, all of these should already be installed from `make tools` . However you can install them separately using the DEPLOY tabs below.

To access the tools, navigate to the ACCESS tabs below.

All of these commands are to be run from `node-launcher`

LOGS MANAGEMENT (KIBANA)

It is recommended to deploy an Elastic Search / Logstash / Filebeat / Kibana to redirect all logs within an elasticsearch database and available through the UI Kibana.

For a reminder on Kubernetes commands, please [visit this page](#).

DEPLOY

You can deploy the log management automatically using the command below:

```
make install-logs
```

This command will deploy the elastic-operator chart. It can take a while to deploy all the services, usually up to 5 minutes depending on resources running your kubernetes cluster.

You can check the services being deployed in your kubernetes namespace

```
elastic-system .
```

ACCESS

You can automate this task to access Kibana from your local workstation:

```
make kibana
```

Open <https://localhost:5601> in your browser. Your browser will show a warning because the self-signed certificate configured by default is not verified by a third party certificate authority and not trusted by your browser. You can temporarily acknowledge the warning for the purposes of this quick start but it is highly recommended that you configure valid certificates for any production deployments.

Login as the elastic user. The password should have been displayed in the previous command (`make kibana`).

To manually access Kibana follow these instructions: A ClusterIP Service is automatically created for Kibana:

```
kubectl -n elastic-system get service elasticsearch-kb-http
```

Use `kubectl port-forward` to access Kibana from your local workstation:

```
kubectl -n elastic-system port-forward service/elasticsearch-kb-http 5601
```

Login as the `elastic` user. The password can be obtained with the following command:

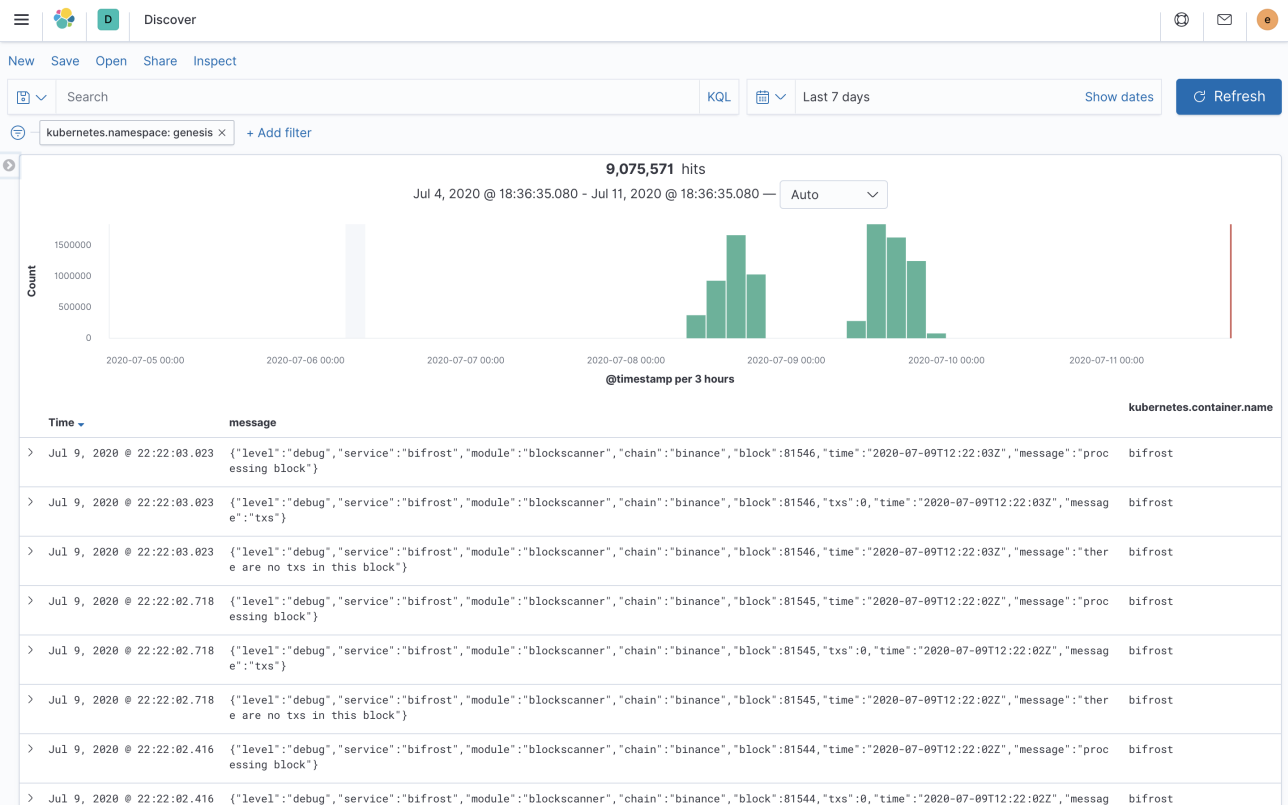
```
kubectl -n elastic-system get secret elasticsearch-es-elastic-user -o
```

To access the logs of the THORNode services running, you can either use directly Kubernetes commands to get logs from different deployments, for example, to get the logs of the service `thor-daemon` :

```
kubectll logs -f deploy/thor-daemon -n thornode
```

From there, you will need to create a new index to get the logs coming from Filebeat. You can search using the regular “Discover” app in Kibana to go through your logs using a filter to get only specific services or keywords, etc.

You can follow a more in-depth [introduction to Kibana here](#).



Overview of Kibana Logs

METRICS MANAGEMENT (Prometheus)

It is also recommended to deploy a Prometheus stack to monitor your cluster and your running services.

DEPLOY

You can deploy the metrics management automatically using the command below:

```
make install-metrics
```

This command will deploy the prometheus chart. It can take a while to deploy all the services, usually up to 5 minutes depending on resources running your kubernetes cluster.

You can check the services being deployed in your kubernetes namespace

```
prometheus-system .
```

ACCESS

We have created a make command to automate this task to access Grafana from your local workstation:

```
make grafana
```

Open <http://localhost:3000> in your browser.

Login as the `admin` user. The default password should have been displayed in the previous command (`make grafana`).

```
[> make grafana
User: admin
Password: prom-operator
Open your browser at http://localhost:3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Access Prometheus admin UI

We have created a make command to automate this task to access Prometheus from your local workstation:


```
make prometheus
```

Open <http://localhost:9090> in your browser.

As part of the tools command deployment, you also have deployed a Prometheus stack in addition to the Elasticsearch in your Kubernetes cluster. All CPU, memory, disk space, and THORNode / THORChain custom metrics are automatically being sent to the Prometheus database backend deployed in your cluster.

You should have available different dashboards to see the metrics across your cluster by nodes, deployments, etc, and also a specific **THORNode / THORChain** dashboard to see the THORChain status, with current block height, how many validators are currently active and other chain related information.



Click the  SEARCH ICON to find the list of dashboards



Example of Grafana Dashboard

For a more in-depth introduction of Grafana, please [follow the documentation here](#).

Kubernetes Dashboard

You can also deploy the Kubernetes dashboard to monitor your cluster resources.

DEPLOY

```
make install-dashboard
```

This command will deploy the Kubernetes dashboard chart. It can take a while to deploy all the services, usually up to 5 minutes depending on resources running your kubernetes cluster.

ACCESS

We have created a make command to automate this task to access the Dashboard from your local workstation:

```
make dashboard
```

Open <http://localhost:8000> in your browser.

View your kubernetes dashboard by running the following:

```
make dashboard
```

Kubernetes					
genesis					
Search					
Workloads > Deployments					
Cluster					
Cluster Roles					
Namespaces					
Nodes					
Persistent Volumes					
Service Accounts					
Storage Classes					
Workloads					
Cron Jobs					
Daemon Sets					
Deployments					
Jobs					
Pods					
Replica Sets					
Replication Controllers					
Stateful Sets					
Service					
Ingresses					
Services					
Config and Storage					
Config Maps					
Persistent Volume Claims					
Deployments					
Name	Labels	Pods	Created	Images	
midgard	app.kubernetes.io/component: midgard app.kubernetes.io/instance: genesis	1 / 1	a day ago	registry.gitlab.com/thorchain/midgard:mocknet	
midgard-timescaledb	app.kubernetes.io/component: timescaledb app.kubernetes.io/instance: genesis	1 / 1	a day ago	timescale/timescaledb:1.7.1-pg12	
thor-api	app.kubernetes.io/instance: genesis app.kubernetes.io/managed-by: Helm	1 / 1	a day ago	registry.gitlab.com/thorchain/thorchain-de:mocknet-30	
bifrost	app.kubernetes.io/instance: genesis app.kubernetes.io/managed-by: Helm	1 / 1	a day ago	registry.gitlab.com/thorchain/thorchain-de:mocknet-30	
binance-daemon	app.kubernetes.io/instance: genesis app.kubernetes.io/managed-by: Helm	1 / 1	a day ago	registry.gitlab.com/thorchain/bepswap/mock-binance:latest	
thor-daemon	app.kubernetes.io/instance: genesis app.kubernetes.io/managed-by: Helm	1 / 1	a day ago	registry.gitlab.com/thorchain/thorchain-de:mocknet-30	

Kubernetes Dashboard

Backing up a THORNode

You should backup your THORNode in case of failures. By default, if you are using the Kubernetes deployments solution, all the the deployments are automatically backed up by persistent volume disks.

Depending on your provider, the volumes are usually available in the provider administration UI, for example in AWS, you can find those volumes in your regular console, in the region you chose to deploy your Kubernetes cluster.

Again by default, with Kubernetes, by using persistent volumes used in the default configuration, you are already protected again restart failures at container level, or node failures. As long as you don't specifically use the destroy commands from the Makefile or manually delete your Kubernetes deployments, your volumes will NOT be deleted at any time.

It is still recommended, as any project, to have different backups on top of those volumes to make sure you can recover in admin error deleting those volumes or other Kubernetes

resources that would imply deleting those volumes.

For AWS, you can easily setup in your console to have snapshots of your cluster volumes be taken every day. For other provider there can be different ways to achieve this as well either manually or automatically.

It is up to the node operator to setup those extra backups of the core volumes to be able to recover in any kind of failures or human errors.

Some volumes would be more critical than others, for example Midgard deployment are also by default backed up by persistent volumes, so it can recover in case of container restarts, failures or node failures and the deployment being automatically scheduled to a different node, but if you were to delete the Midgard volume, it would reconstruct its data from your THORNode API and events from scratch. For that specific service having extra backups might not be critical, at the time of the writing of that document, Midgard implementation might change in the future.

Node Security

The following are attack vectors:

1. If anyone accesses your AWS credentials, they can log in and steal your funds
2. If anyone accesses the device you used to log into kubernetes, they can log in and steal your funds
3. If anyone accesses your hardware device used to bond, they can sign a LEAVE transaction and steal your bond once it is returned



RUNNING A NODE IS SERIOUS BUSINESS


DO SO AT YOUR OWN RISK, YOU CAN LOSE A SIGNIFICANT QUANTITY OF FUNDS IF AN ERROR IS MADE

THORNODE SOFTWARE IS PROVIDED AS IS - YOU ARE SOLELY RESPONSIBLE FOR USING IT

Alerting

thornode-telegram-bot ⚡

A telegram bot to monitor the status of THORNodes.

 This bot can be run in a self-sovereign manner (a personal bot that you run that does not collect any information about you) or by using the block42 bot: https://t.me/thornode_bot on telegram.

If you have questions feel free to open a github issue or contact Block42 in their Telegram Channel https://t.me/block42_crypto

<https://github.com/block42-blockchain-company/thornode-telegram-bot>

Requirements

- Telegram
- Docker (if you want to run with docker or docker-compose)
- Docker Compose (if you want to run with docker-compose)
- Python3 (if you want to run without docker)

Quickstart

Open `variables.env` file and set

- `TELEGRAM_BOT_TOKEN` to your Telegram Bot Token obtained from BotFather.
- `THORCHAIN_NODE_IP` to any THORNode IP you want to monitor (or `localhost`). Leave it empty or remove it to use testnet seed Node IPs.
- `BINANCE_NODE_IP` to any Binance Node IP you want to monitor (or `localhost`). Leave it empty or remove it to not monitor a Binance Node.
- `ADMIN_USER_IDS` to a list of Telegram User IDs that are permissioned to access the Admin Area.

Install `docker` and `docker-compose` and run:

```
docker-compose up -d
```

Steps to run everything yourself

Install dependencies

Install all required dependencies via: `pip install -r requirements.txt`

Create Telegram bot token via BotFather

Start a Telegram chat with [BotFather](#) and click `start` .

Then send `/newbot` in the chat, and follow the given steps to create a new telegram token. Save this token, you will need it in a second.

Set environment variables

Set the telegram bot token you just created as an environment variable:

```
TELEGRAM_BOT_TOKEN
```

```
export TELEGRAM_BOT_TOKEN=XXX
```

Next you can specify the IP of the Thornode that you want to watch in the `THORCHAIN_NODE_IP` environment variable.

Set it to `localhost` to listen a node on your own machine.

Leave this environment variable empty or don't even set it to use IPs of the testnet seed nodes from <https://testnet-seed.thorchain.info>.



Please note, if you leave `THORCHAIN_NODE_IP` empty, IP specific monitoring won't take effect (no check for increasing block height, midgard API and catch up status). Because we rotate through available testnet seed IPs, we would compare data of different nodes and send incorrect alerts.

```
export THORCHAIN_NODE_IP=3.228.22.197
```

If you have a Binance Node IP that you want to monitor, you can set `BINANCE_NODE_IP` to this IP. Set it to `localhost` if the Binance Node runs on the same machine as the Telegram Bot.

Leave this environment variable empty or don't even set it to not do any Binance Node monitoring.

```
export BINANCE_NODE_IP=3.228.22.197
```

Next set Telegram User IDs that are permissioned to access the Admin Area in the `ADMIN_USER_IDS` environment variable. Leave the default dummy values if you do not intend to use the Admin Area.

To find out your Telegram, open your Telegram Client, and search for the Telegram Bot `@userinfobot`. Ensure that this is a Bot and not a Channel and has exactly the handle `@userinfobot`, as there are a lot of channels and bots with similar names. Start this Bot and it returns you your User ID that you need to export in `ADMIN_USER_IDS`.

If you enter multiple User IDs, make sure to separate the IDs with `,` i.e. a comma.

```
export ADMIN_USER_IDS=12345,56789,42424
```

Finally, if you want test the Thornode Telegram Bot with data from your local machine, you need to set the debug environment variable:

```
export DEBUG=True
```

The DEBUG flag set to True will run a local web server as a separate process. This way the telegram bot can access the local files `nodeaccounts.json` und `status.json` in the `test/` folder.

To test whether the bot actually notifies you about changes, the data the bot is querying needs to change. You can simulate that by manually editing `test/nodeaccounts.json` , `test/status.json` and `test/midgard.json` .

Furthermore in DEBUG mode a separate process runs `increase_block_height.py` which artificially increases the block height so that there are no notifications that the block height got stuck.

If you are using a JetBrains IDE (e.g. Pycharm), you can set these environment variables for your run configuration which is very convenient for development (see:

<https://stackoverflow.com/questions/42708389/how-to-set-environment-variables-in-pycharm>).

Start the bot

Start the bot via:

```
python3 thornode_bot.py
```

Make sure that you see a message in the console which indicates that the bot is running.

Run and test the bot

When you created the telegram bot token via BotFather, you gave your bot a certain name (e.g. `thornode_bot`). Now search for this name in Telegram, open the chat and hit start!

At this point, you can play with the bot, see what it does and check that everything works fine!

The bot persists all data, which means it stores its chat data in the file `storage/session.data` . Once you stop and restart the bot, everything should continue as if the bot was never stopped.

If you want to reset your bot's data, simply delete the file `session.data` in the `storage` directory before startup.

Production

In production you do not want to use mock data from the local endpoint but real network data. To get real data just set `DEBUG=False` and all other environment variables as described in the 'Set environment variables' section. If you're using docker-compose to run this Bot, modify the existing variables in `variables.env` file (No need to set DEBUG as there's no DEBUG mode in the docker version).

Docker Standalone

To run the bot as a docker container, make sure you have docker installed (see: <https://docs.docker.com/get-docker>).

Navigate to the root directory of this repository and execute the following commands:

Build the docker image as described in the `Dockerfile` :

```
docker build -t thornode-bot .
```

To make the bot's data persistent, you need to create a docker volume. If the bot crashes or restarts the volume won't be affected and keeps all the session data:

```
docker volume create thornode-bot-volume
```

Finally run the docker container:

```
docker run --env TELEGRAM_BOT_TOKEN=XXX --env THORCHAIN_NODE_IP=XXX --env BI
```

Set the `--env TELEGRAM_BOT_TOKEN` flag to your telegram bot token.

Set the `--env THORCHAIN_NODE_IP` flag to an IP of a running THORNode, or to `localhost` if the THORNode runs on the same machine as the Telegram Bot. If you don't know any IP leave this empty i.e. `--env THORCHAIN_NODE_IP=` or remove it completely - then the Telegram Bot works with testnet seed node IPs from <https://testnet-seed.thorchain.info>.

Set the `--env BINANCE_NODE_IP` flag to an IP of a running Binance Node, or to `localhost` if Telegram Bot and Binance Node run on the same machine. Leave this empty i.e. `--env BINANCE_NODE_IP=` or remove it to not do any Binance monitoring.

The `-v` argument passes the dockersocket to the container so that we can restart docker containers from inside the Telegram Bot.

Finally, the `--mount` flag tells docker to mount our previously created volume in the directory `storage`. This is the directory where your bot saves and retrieves the `session.data` file.



Please note that as docker is intended for production, there is not the possibility for the `DEBUG` mode when using docker.

Healthcheck

There is a health check in the Dockerfile that runs the `healthcheck.py` file. The script assures that `thornode_bot.py` is periodically updating the `health.check` file.

If the docker health check fails, the docker container is marked as "unhealthy". However, when using docker standalone (without docker compose or docker swarm) this doesn't do anything. To restart unhealthy containers, we have to use the autoheal image <https://hub.docker.com/r/willfarrell/autoheal/>.

To make sure a potentially unhealthy thorbot_node container is restarted, run the autoheal container alongside the thornode_bot container:

```
docker run -d --name autoheal --restart=always -v /var/run/docker.sock:/var/
```

Docker Compose

The explained steps in the Docker Standalone section are conveniently bundled into a `docker-compose.yaml` file.

First, as before, you need to set the right values in the `variables.env` file for `TELEGRAM_BOT_TOKEN`, `THORCHAIN_NODE_IP`, `BINANCE_NODE_IP` and `ADMIN_USER_IDS`

If you don't want to spin up the official docker image from our dockerhub, open `docker-compose.yaml` and comment out the line `image: "block42blockchaincompany/thornode_bot:latest"` and comment in the line `build: .`

Finally, start the Thornode Telegram Bot with:

```
docker-compose up -d
```

If you have problems running 'docker-compose up' while using a VPN, try to this:

- First run in your console

```
docker network create vpnworkaround --subnet 10.0.1.0/24
```

- Then comment in the networks configuration in `docker-compose.yml`

```
1 networks:
2   default:
3     external:
4       name: vpnworkaround
```

- Run again in your terminal

```
docker-compose up -d
```

This solution is taken from <https://github.com/docker/for-linux/issues/418#issuecomment-491323611>

Testing

To test the Thornode Bot, you need to impersonate your own Telegram Client programmatically.

To do that, you need to obtain your API ID and API hash by creating a telegram application that uses your user identity on <https://my.telegram.org> . Simply login in with your phone number that is registered on telegram, then choose any application (we chose Android) and follow the steps.

Once you get access to `api_id` and `api_hash`, save them in the Environment variables `TELEGRAM_API_ID` and `TELEGRAM_API_HASH` respectively. Also save the name of your Telegram Bot without the preceding `@` in the `TELEGRAM_BOT_ID` environment variable (e.g. if your bot is named `@thornode_test_bot` , save `thornode_test_bot` in `TELEGRAM_BOT_ID`).

You also need to have set the `TELEGRAM_BOT_TOKEN` environment variable with your telegram bot token, set `ADMIN_USER_IDS` with permissioned IDs and set `DEBUG=True` as explained in previous sections. If you want to test the restarting of docker containers, do not forget to start at least one container on your system.

Keep in mind that the test always deletes the `session.data` file inside `storage/` in order to have fresh starts for every integration test. If you wish to keep your persistent data, don't run the integration test or comment out the line `os.remove("../storage/session.data")` in `integration_test.py`

To run the test open the `test/` folder in your terminal and run

```
python3 integration_test.py
```

The test should endure several minutes. Every command succeeded if you see

```
-----ALL TESTS PASSED-----
```

 at the end.

Leaving


Overview

Every 50,000 Blocks (3 days) the system will churn its nodes:


1. The oldest node, or
2. The most unreliable node

Churned nodes will be put in standby, but their bond will not automatically be returned. They will be credited any earned rewards in their last session. If they do nothing but keep their cluster online, they will be eventually churned back in.

Alternatively, a node can leave the system voluntarily, in which case they are typically churned out 6 hours later. Leaving is considered permanent, and the node-address is permanently jailed. This prevents abuse of the **LEAVE** system since leaving at short notice is disruptive.

 It is assumed nodes that wish to **LEAVE** will be away for a significant period of time, so by permanently jailing their address, it forces them to completely destroy and re-build before re-entering. This also ensures they are running the latest software.

Unbonding


 You can only unbond when your Node is on "standby", ie, just before it is selected to churn in, or after it is churned out.


If a Node Operator wants to retrieve part of their bond & rewards (such as deciding to take profits and contribute as a liquidity provider in order to maximise yield), they can simply Unbond. This keeps their Node on standby, ready to be churned back in.


To unbond from the system, simply send an **UNBOND** transaction to the Vault Address with at least 1 satoshi in funds. The amount and type of asset you use to send to THORChain is actually irrelevant, you are simply passing transaction intent to THORChain and proving you own your bond address.

Example, this will draw out 10k in RUNE from the bond, as long as the remaining amount is higher than the minimum bond.

```
UNBOND:thor1ryr5eancepk1ax5am8mdpkx6mr0rg4xjnjx6zz:10000000000000
```

 THORChain always treats assets in 1e8 "base format" ie, 1.0 RUNE = 100,000,000 units. To get from one to the other, simply multiply by 100m.

 You can get your node address, as well as the current vault address by running `make status`

 Only the address that originally bonded the funds can **UNBOND** or **LEAVE**. This ensures you can safely leave this system if you no longer have access to your node (but it is still running).

Leaving

Leaving is considered permanent. There are two steps.

1. If you are **active**, send a LEAVE transaction to start a churn-out process. This could take several hours.
2. If you are **standby**, send a LEAVE transaction to get your bond back and be permanently jailed.

To leave the system, send the following transaction from your original bond address to the Vault Address: `LEAVE:<ADDRESS>` with at least 1 satoshi in funds.

Example:

```
LEAVE:thor1ryr5eancepk1ax5am8mdpkx6mr0rg4xjnjx6zz
```

□ *Wait a few hours, verify on the /nodeaccount endpoint that you are now **disabled*** □

```
LEAVE:thor1ryr5eancepk1ax5am8mdpkx6mr0rg4xjnjx6zz
```

□ *Wait a few minutes, verify you have received your bond back* □

□ *Commence destroying your node* □



If your node is both offline and inaccessible, then it will be unable to return any assets in its yggdrasil vaults and it will be slashed 1.5x the value of those assets.

Example: If your node has a \$500k bond (in RUNE), but has \$100k in assets in its vaults it can't return, it will lose \$150k in RUNE from its bond. The Node will get back \$350k in its bond.

Confirming you have left

You should complete this checklist before you do the next step:

1. Have you sent a final **LEAVE** transaction and have you received your BOND back - ie 1,000,000 RUNE, and can your account for any slash points or rewards?

If yes, then proceed:

DESTROY

To destroy and remove previously created resources, you can run the command below.



Destroying your cluster will completely destroy your node, including purging all keys on it.

DO NOT DO THIS UNTIL YOUR NODE HAS CHURNED OUT AND YOU HAVE VERIFIED YOUR BOND IS COMPLETELY RETURNED

IF YOU DESTROY A NODE PREMATURELY, YOU MAY LOSE A SIGNIFICANT AMOUNT OF FUNDS

1) Destroying the Node and Tools

First, destroy the node and tools, this will delete your node then your tooling 1-by-1. Do this from the `node-launcher` repo:

```
make destroy destroy-tools
```

```
[> make destroy destroy-tools
Deleting THORNode
release "thornode" uninstalled
namespace "thornode" deleted
Deleting Logs Management
release "elastic" uninstalled
namespace "elastic-system" deleted
Deleting Metrics
release "metrics-server" uninstalled
release "prometheus" uninstalled
namespace "prometheus-system" deleted
Deleting Kubernetes Dashboard
release "kubernetes-dashboard" uninstalled
```

Destroying the Tooling

2) Destroy the cluster

Then destroy the cluster from the `cluster-launcher` repo:

AWS

You will be asked to enter your cluster name and region (the same as what you [put in when you first deployed](#)).

```
make destroy-aws
```

DO

You will be asked to enter your cluster name and region, as well as your Personal Token (the same as what you [put in when you first deployed](#)).

```
make destroy-do
```

You will be asked to confirm:

```
Plan: 0 to add, 0 to change, 50 to destroy.
```

```
Do you really want to destroy all resources?
```

```
Terraform will destroy all your managed infrastructure, as shown above.  
There is no undo. Only 'yes' will be accepted to confirm.
```

```
Enter a value: █
```



DO NOT DESTROY YOUR NODE UNTIL YOU HAVE CHURNED OUT AND HAVE RECEIVED YOUR FULL BOND BACK IN YOUR CUSTODY

IF YOU DESTROY YOUR NODE WITH FUNDS LOCKED UP - YOU WILL LOSE A SIGNIFICANT QUANTITY OF FUNDS

```
-----  
module.eks.aws_security_group.workers[0]: Destroying... [id=sg-007eabbdd46a7]  
module.eks.aws_iam_role.cluster[0]: Destruction complete after 4s  
module.eks.aws_security_group.workers[0]: Destruction complete after 2s  
module.eks.aws_security_group.cluster[0]: Destruction complete after 2s  
module.vpc.aws_vpc.this[0]: Destroying... [id=vpc-0b587e4e5e741238f]  
module.vpc.aws_vpc.this[0]: Destruction complete after 1s
```

Destroy complete! Resources: 50 destroyed.

Final destroy complete

✓ CHECKLIST

Deploying

- ☐ I have set up a local `thornode` repository with a `cluster-launcher` and `node-launcher` git cloned into the right sub-folders
- ☐ I have installed all the pre-requisites
- ☐ I have a AWS/DO account ready with credentials if using a service provider
- ☐ I have deployed a cluster and saved my node details (name of node, region)
- ☐ I have installed tools via `make tools`
- ☐ I have deployed a node by running `make chaosnet-validator`
- ☐ I have securely saved my THORNode Password and THORNode Mnemonic

→ [Deploy - K8 Cluster](#)


/thornodes/kubernetes

→ [Deploying](#)

/thornodes/deploying

Joining

- ☐ I have retrieved the latest VAULT address and sent a small test bond
 - ☐ I have confirmed my node has been credited the small bond
 - ☐ (Optional) I have confirmed I can unbond a small amount
 - ☐ I have sent the final bond, higher than the MinimumBond
 - ☐ I have run `make set-ip-address` to set my Node's IP address
 - ☐ I have run `make set-node-keys` to set my Node's public keys
 - ☐ I have run `make set-version` to set my Node's version
 - ☐ I have verified in `make status` that my node is ready to churn in
 - ☐ (Optional) I have added my node key to the Telegram Bot for notifications
-


 The BOND memo is `BOND:<node-address>`

Upgrading

- ☐ I have run `make status` to verify the current state of my node
- ☐ I have run `kubectl get pods -n thornode` to verify all my node's services are running properly, resetting any that are faulty
- ☐ I have run the upgrade command
- ☐ I have run `make set-version` to set my new Node version
- ☐ I have run `make status` to verify the final state of my node

Unbonding

- ☐ I have waited until my node is churned out and is in `standby`
- ☐ I have sent an UNBOND transaction from the same wallet registered to my node
- ☐ I have verified that I have received my BOND back


 The UNBOND memo is `UNBOND:<node-address>:<amount>`

Leaving Whilst Active

- ☐ I have confirmed my node is active and I wish to leave before waiting to churn naturally
- ☐ I have sent the first LEAVE transaction and verified my node has "requested to leave".
- ☐ I have verified that a churn has taken place and my node is in STANDBY
- ☐ I have verified that my node has returned all hot funds by checking my node's yggdrasil vault on the explorer
- ☐ I have sent the final LEAVE transaction and received my bond back

→ Leaving

/thornodes/leaving

 The LEAVE memo is LEAVE:<node-address>

Leaving Whilst Standby

- ☐ I have verified that a churn has taken place and my node is in STANDBY
- ☐ I have verified that my node has returned all hot funds by checking my node's yggdrasil vault on the explorer
- ☐ I have sent the final LEAVE transaction and received my bond back

→ Leaving

/thornodes/leaving

Destroying a Node

- ☐ I have verified that I have either UNBONDED my entire BOND or LEFT and received my BOND back.
- ☐

I have run `make destroy destroy-tools` to destroy my node from `node-launcher`

☐ I have run `make destroy-aws` to destroy my cluster from `cluster-launcher`

→ Leaving

/thornodes/leaving

Developers

THORCLI

First, clone THORNode.

Then `make install`

Connecting to THORChain

The active node IP addresses can be queried from this endpoint:

TESTNET

<https://testnet-seed.thorchain.info>

CHAOSNET

<https://chaosnet-seed.thorchain.info>

MAINNET

<https://seed.thorchain.info>

The Network Information comes from three sources:

1. MIDGARD: Consumer information relating to swaps, pools, volume. DeFi dashboards, Wallets, Exchanges will primarily interact with Midgard.
2. THORNODE: Raw blockchain data relating to the THORChain state machine. THORChain block explorers will query THORChain-specific information here.
3. TENDERMINT: Tendermint standard data, used by all block explorers to query for base information.

MIDGARD

Midgard returns time-series information regarding the THORChain network, such as volume, pool information, users, liquidity providers and more.

Port: 8080

RPC Guide:

<http://<host>:8080/v1/doc>

Port: 8080

RPC Guide:

<http://:8080/v1/doc>

Example:

<http://:8080/v1/stats>

THORNODE

THORNode returns application-specific information regarding the THORChain state machine, such as balances, transactions and more.

Port: 1317

RPC Guide:

<https://gitlab.com/thorchain/thornode/-/blob/master/x/thorchain/query/query.go>

Example:

<http://:1317/thorchain/constants>

RPC

RPC allows base blockchain information to be returned.

TESTNET Port: 26657

MAINNET Port: 27147

RPC Guide:

<https://docs.tendermint.com/master/rpc/>

Example:

<http://:26657/genesis>

P2P

P2P is the network layer between nodes, useful for network debugging.

TESTNET Port: 26656

MAINNET Port: 27146

P2P Guide

<https://docs.tendermint.com/master/spec/p2p/>

Transaction Memos

Overview

THORChain processes all transactions made to the vault address that it monitors. The address is discovered by clients by querying THORChain (via Midgard).

Transactions to THORChain pass user-intent with the `MEMO` field on their respective chains. The THORChain inspects the transaction object, as well as the `MEMO` in order to process the transaction, so care must be taken to ensure the `MEMO` and the transaction is valid. If not, THORChain will automatically refund.

Mechanism for Transaction Intent

Each chain will have a unique way of adding state to a transaction:

Chain	Mechanism	Notes
Bitcoin	OP_RETURN	Limited to 80 bytes, long memos need to use two OP_RETURN outputs.
Ethereum	Smart Contract Input	The user can pass in the memo in the <code>deposit(asset, value, memo)</code> function, where <code>memo</code> is bytes32. This is emitted as an event.
Binance Chain	MEMO	Each transaction has an optional memo, limited to 128 bytes.
Monero	Extra Data	Each transaction can have attached <code>extra data</code> field, that has no limits.

Transactions

The following transactions are permitted:

SINGLECHAIN

Type	Payload	MEMO	Expected Outcome
STAKE	<div>RUNE & Token</div> <p>Can be either, or just one side.</p>	STAKE:ASSET	Stakes into the specified pool.
WITHDRAW	<div>0.00000001 BNB</div> <p>A non-zero transaction.</p>	<div>WITHDRAW:ASSET:PERCENT</div> <p>Percent is in basis points (0-10000, where 10000=100%)</p>	Withdraws from a pool
SWAP	<div>RUNE Token</div> <p>Either/Or</p>	<div>SWAP:ASSET:DESTADDR:LIM</div> <p>Set a destination address to swap and send to someone. If DESTADDR is blank, then it sends back to self:</p> <div>SWAP:ASSET::LIM</div>	Swaps to token.
		<p>Set price protection. If the value isn't achieved then it is refunded. ie, set 10000000 to be guaranteed a minimum of 1 full asset.</p> <p>If LIM is omitted, then there is no price protection:</p> <div>SWAP:ASSET:DESTADDR</div>	
		<p>If both are omitted then the format is:</p> <div>SWAP:ASSET</div>	

ADD Assets

RUNE &|
Token

Can be either,
or just one
side.

ADD:ASSET

Adds to the
pool
balances
without
being
credited.

MULTICHAIN

Type	Payload	MEMO	Expected Outcome
ADD LIQUIDITY	AssetChain: ASSET	AssetChain: ADD:ASSET:THOR-ADDRESS	Adds into the specified pool.
	THORChain: RUNE	THORChain: ADD:ASSET:ASSET-ADDRESS	
WITHDRAW	AssetChain: lowest possible	AssetChain: WITHDRAW:ASSET:PERCENT	Withdraws from a pool
	THORChain: 0	Percent is in basis points (0-10000, where 10000=100%)	
SWAP	Amount to swap	SWAP:ASSET:DESTADDR:LIM	Swaps to token.
		Set a destination address to swap and send to someone. If DESTADDR is blank, then it sends back to self:	
		SWAP:ASSET::LIM	
		Set trade protection. If the value isn't achieved then it is refunded. ie, set 10000000 to be guaranteed a minimum of 1 full asset.	
		If LIM is ommitted, then there is no price protection:	
		SWAP:ASSET:DESTADDR:	
		If both are ommitted then the format is:	
		SWAP:ASSET	

DONATE
Assets

RUNE &|
Token
Can be
either.

DONATE:ASSET

Adds to the
pool
balances
without being
credited.

Refunds

The following are the conditions for refunds:

Condition	Notes
Invalid MEMO	If the MEMO is incorrect the user will be refunded.
Invalid Assets	If the asset for the transaction is incorrect (adding an asset into a wrong pool) the user will be refunded.
Invalid Transaction Type	If the user is performing a multi-send vs a send for a particular transaction, they are refunded.
Exceeding Price Limit	If the final value achieved in a trade differs to expected, they are refunded.

Refunds cost fees to prevent Denial of Service attacks:

Asset	Amount
RUNE	1 RUNE

Non-RUNE Asset	1 RUNE equivalent
----------------	-------------------

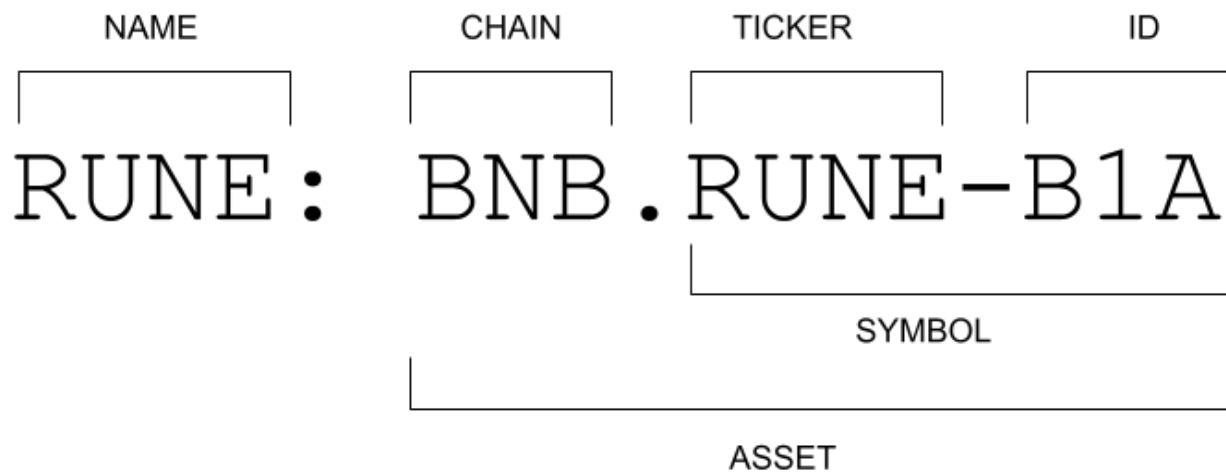
MEMO Alternatives

Alternative MEMO types are available to clients:

Transaction	Long-form MEMO	Short-form MEMO	Symbol MEMO
ADD	ADD	a	+
WITHDRAW	WITHDRAW	wd	-
SWAP	SWAP	s	=
DONATE	DONATE	d	%

Asset Notation

The following is the notation for Assets in THORChain's system:



Examples

Asset	Notation
Bitcoin	BTC.BTC

Ethereum	ETH.ETH
USDT	ETH.USDT-0xdac17f958d2ee523a2206206994597c13d831ec7
BNB	BNB.BNB
RUNE (BEP2)	BNB.RUNE-B1A
RUNE (NATIVE)	THOR.RUNE

ASGARDEX Modules

Midgard API

Overview

Midgard is a layer 2 REST API that provides front-end consumers with semi real-time rolled up data and analytics of the THORChain network. Most requests to the network will come through Midgard. This daemon is here to keep the chain itself from fielding large quantities of requests. You can think of it as a “read-only slave” to the chain. This keeps the resources of the network focused on processing transactions.

Documentation

Midgard documentation is available at `http://<host>:8080/v1/doc`

Chain Proxies

Midgard also serves to proxy to various chains. This is useful for broadcasting signed transactions back to connected networks.

Instead of running a Bitcoin full node, a wallet can send a signed transaction or make RPC queries direct to a THORNode:

```
http://<host>:8332
```

Seed Service

The team maintain a simple Seed Service that crawls for active node accounts and their IP addresses. This is then hosted on an endpoint and can be queried.



The Seed Service only shows active nodes, but it does not proof them. Clients should proof THORNodes by using the `asgardex-midgard` module.

The active node IP addresses can be queried from this endpoint:

TESTNET

<https://testnet-seed.thorchain.info>

MAINNET

<https://seed.thorchain.info>

Proofing THORNodes

The security model is always by consensus - the truth is what everyone agrees it is. Any THORNode can attempt to spoof downstream clients and send a fake vault address, but a client can protect themselves against this by checking that at least 1/3rd of Nodes agree.

The process for this is:

1. Get the full list of active THORNode IPs

2. Retrieve the vault address off at least 1/3rd of them
3. Check for 100% correctness, if incorrect, repeat Step (2)
4. If correct, use one of the proofed THORNodes for the next session
5. Repeat Step (1) regularly



Wallets and services that are very paranoid should instead run their own THORNode as a non-consensus node and retrieve the vault address directly

Decentralisation Plan

It is possible to host the Seed Service entirely in a web3 smart contract on a censorship-resistant blockchain and clients can use Web3 to query.

How it Works

Emission Schedule

Block rewards are calculated as such:

$$blockReward = \frac{\frac{reserve}{emissionCurve}}{blocksPerYear} = \frac{\frac{220,447,472}{6}}{6311390} = 5.8096$$

So if the reserve has 220m rune, a single block will emit ~5.8 Rune from the reserve, which means 2/3rds to that is awarded to the node operators (~3.8) and is divided up between each operator. The rest is paid to Liquidity providers.

The emission curve is designed to start at around 30% APR and target 2% after 10 years. At that point, the majority of the revenue will come from fees.

Incentive Pendulum

The capital on THORChain can lose its balance over time. Sometimes there will be too much capital in liquidity pools; sometimes there will be too much bonded by nodes. If there is too much capital in liquidity pools, the network is unsafe. If there is too much capital bonded by nodes, the network is inefficient.

If the network becomes unsafe, it increases rewards for node operators and reduces rewards for liquidity providers. If the network becomes inefficient, it boosts rewards for liquidity providers and reduces rewards for node operators.

Balancing System States

THORChain can be in 1 of 5 main states—

- Unsafe
- Under-Bonded
- Optimal
- Over-Bonded
- Inefficient

These different states can be seen in the relationship between bonded Rune and pooled Rune. The amount of Rune which has been bonded by node operators, and the amount which has been added to liquidity pools by liquidity providers.

Optimal State

Optimal State Safe & Efficient

INCENTIVE PENDULUM

50% Capital Bonded



50% Capital Pooled



Rewards

NODE OPERATORS
No change

STAKERS
No change



In the optimal state, bonded capital is roughly equal to pooled capital. Bonded capital is 100% Rune; pooled capital half Rune and half external assets.

67% of Rune in the system is bonded and 33% is pooled. This is the desired state. The system makes no changes to the incentives for node operators or liquidity providers.

Unsafe State

Unsafe State

INCENTIVE PENDULUM

Low Capital Bonded



High Capital Pooled



Rewards

NODE OPERATORS
Increase

STAKERS
Decrease

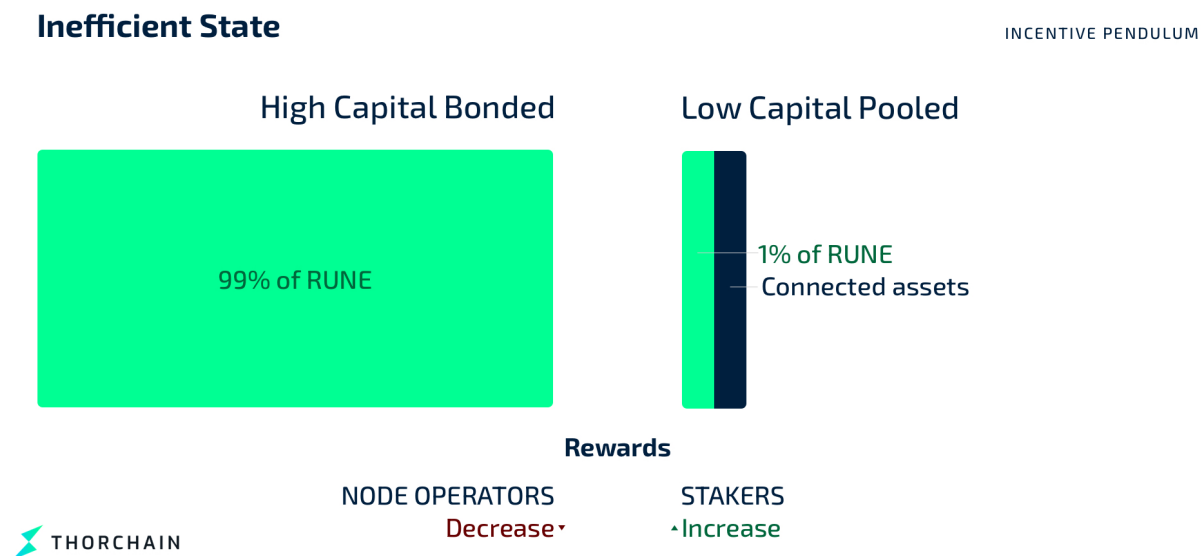


The system may become unsafe. In this case, pooled capital is higher than bonded capital. Pooled Rune is now equal to bonded Rune – a 50/50 split.

This is undesirable because it means that it's become profitable for node operators to work together to steal assets.

To fix this, the system increases the amount of rewards going to node operators and lowers the rewards going to liquidity providers. This leads to more node operators getting involved, bonding more Rune and increasing bonded capital. This also disincentivises liquidity providers from taking part. They receive less return on their investment, so they pull assets out and reduce the amount of pooled capital. With time, this restores balance and the system moves back towards the optimal state.

Inefficient State



The system can also become inefficient. In this case, pooled capital would be much lower in value than bonded capital. This is a problem because it means that much more capital is being put into securing pooled assets than those assets are actually worth.

To fix this, the system increases rewards for liquidity providers and decreases rewards for node operators. This attracts more liquidity providers to the system, and fewer node operators. Liquidity providers add more capital to receive more rewards, increasing pooled capital. Some node operators remove their bonded Rune, seeking more profitable places to

put their capital. Bonded capital falls.
In this way, the system returns to the optimal state.

Under and Over-Bonded States

The under- and over-bonded states are less severe intermediary states. Being under-bonded is not a threat in itself because it is not yet profitable for node operators to steal. Being over-bonded is not a problem in itself because the system is still operating quite well.

The THORChain team does not expect the unsafe or inefficient states to come up often. The system will be in the over-bonded state most of the time, particularly as it gets easier for people to run nodes.

 Try this [interactive model](#) of the Incentive Pendulum.

Algorithm

The algorithm that controls the Incentive Pendulum is as follows:

$$shareFactor = \frac{b + s}{b - s}$$

$$b = totalBonded, s = totalPooled$$

In the stable situation of 67m RUNE bonded and 33m RUNE pooled:

$$shareFactor = \frac{67 + 33}{67 - 33} = 3$$

The state machine then takes the inverse of this to determine how much to send to Liquidity providers:

Thus, 33% of the rewards will be sent to Liquidity providers in the stable scenario.

Driving Capital Allocation

As a by-product of the Incentive Pendulum's aggressive re-targeting of 67:33 split of BONDED:POOLED RUNE, it means that in an equilibrium, the value of BONDED RUNE will always be double the value of POOLED RUNE. Since POOLED RUNE is 1:1 bonded with POOLED Capital (due to liquidity pools), then the total market value of RUNE is targeted to be 3 times the value of pooled assets.

If there is any disruption to this, then it means capital will be re-allocated by Nodes and Liquidity providers to pursue maximum yield, and thus correct the imbalance.

Fees

Fees

Conceptually, fees are both value-capture, access-control and resource-subsidisation mechanisms.

Value Capture

The fees need to capture value from those accessing the resource, and pay it to those providing the resource, and in this case the resource is liquidity. However liquidity is relative to the size of the transaction that demands it over the depth of the market that will service it. A small transaction in a deep pool has less demand for liquidity than a large transaction in a small pool.

Access-control

The other reason for fees is access-control; a way to throttle demand for a fixed resource and let natural market forces take over. If there is too much demand for a resource, fees must rise commensurately. The resource in this case is liquidity, not market depth, thus fees must be proportional to liquidity.

Resource Subsidisation

Every swap on THORChain consumes resources (Disk, CPU, Network and Memory resources from validators). These costs are fixed in nature. In addition, every outgoing transaction demands resources on connected chains, such as paying the Bitcoin mining fee or Ethereum gas cost. As such, THORChain charges a single flat fee on every transaction that pays for internal and external resources.

Other Benefits

In addition to the above, fees also create the following benefits:

1. Avoid dust attacks
2. Store up income after the initial Emission Schedule reduces

3. Give the user a stable fee, rather than a dynamic one which changes with the external network's fees

Fee Process

THORChain maintains an awareness of the trailing gas price for each connected chain, saving both gas price as well as gas cost (inferring transaction weight). Nodes are instructed to pay for outgoing transactions using a gas price that is a multiple of the stored value.

The gas is consumed from each chain's base asset pool - the BTC pool pays for Bitcoin fees, the ETH pool for Ethereum fees etc.

The network then observes an outgoing transaction and records how much it cost in gas in the external asset. The final gas cost is then subsidised back into each pool by paying RUNE from the reserve.

Network Fee

The user is charged an amount that is **three times** the stored gas cost for each chain. The Node can then pay a gas price that is **1.5 times** the gas price, and the pool is subsidised a value that is **twice** what was observed.

Example:

Chain	Typical	Fee	Max Gas	Pool Subsidisation
Bitcoin	\$1	\$3	\$1.50	\$3
Ethereum	\$0.20	\$0.60	\$0.30	\$0.60
Binance Chain	\$0.03	\$0.09	\$0.045	\$0.09

The Network Fee is collected in RUNE and sent to the Protocol Reserve. If the transaction involves an asset which is not RUNE the user pays the Network Fee in the external asset. Then the equivalent is taken from that pool's RUNE supply and added to the Protocol Reserve.

If the transaction is in RUNE then the amount is directly taken in RUNE.

Slip-Based Fee

The CLP algorithm includes a slip-based fee which is liquidity-sensitive. Since demand for liquidity is defined as the size of the transaction over the depth of the market that will service it, then a fee which is proportional to liquidity solves key problems.

Firstly it has better **value-capture** when demand for liquidity is high, no matter the size of the transaction or the depth of the market. This means that over time, pool depths will settle to an equilibrium that is relative to the sizes of transactions that are passed over it. This solves the bootstrapping problem, because low-depth pools may turn out to be more profitable than high-depth pools to liquidity providers.


Secondly it has better **access-control**, since the more a trader (or attacker) demands liquidity, the more they have to pay for it. This makes sandwich attacks prohibitively expensive allowing pools to become reliable price feeds.

Additionally a slip-based fee is stateless and non-opinionated. The final fee paid is always commensurate to the demand of resources (both internal and external) no matter what it is.

$$slip = \frac{x}{x + X}$$

$$fee = slip * output = \frac{x}{x + X} * \frac{xY}{x + X} = \frac{x^2Y}{(x + X)^2}$$

This fee is retained on the output side of the pool, ensuring it counters the trade direction.

-  In an Asset-Asset swap, the fee is applied twice since two pools are involved, however the user only sees it as a single fee and a single slip value.

Continuous Liquidity Pools

Instead of limit-order books, THORChain uses continuous liquidity pools (CLP). The CLP is arguably one of the most important features of THORChain, with the following benefits:

- Provides “always-on” liquidity to all assets in its system.
- Allows users to trade assets at transparent, fair prices, without relying on centralised third-parties.
- Functions as source of trustless on-chain price feeds for internal and external use.
- Democratises arbitrage opportunities.
- Allows pools prices to converge to true market prices, since the fee asymptotes to zero.
- Collects fee revenue for liquidity providers in a fair way.
- Responds to fluctuating demands of liquidity.

CLP Derivation

Element	Description	Element	Description
x	input amount	X	Input Balance
y	output amount	Y	Output Balance

Start with the fixed-product formula:

$$Eqn1 : X * Y = K$$

Derive the raw "XYK" output:

$$Eqn2 : \frac{y}{Y} = \frac{x}{x + X} \rightarrow y = \frac{xY}{x + X}$$

Establish the basis of Value (the spot purchasing power of x in terms of y) and slip, the difference between the spot and the final realised y :

$$Eqn3 : Value_y = \frac{xY}{X}$$

$$Eqn4 : slip = \frac{Value_y - y}{Value_y} = \frac{\left(\frac{xY}{X}\right) - y}{\frac{xY}{X}} = \frac{x}{x + X}$$

Derive the slip-based fee:

$$Eqn5 : fee = slip * output = \frac{x}{x + X} * \frac{xY}{x + X} = \frac{x^2Y}{(x + X)^2}$$

Deduct it from the output, to give the final CLP algorithm:

$$Eqn6 : y = \frac{xY}{x + X} - \frac{x^2Y}{(x + X)^2} \rightarrow y = \frac{xYX}{(x + X)^2}$$

Comparing the two equations (Equation 2 & 6), it can be seen that the Base XYK is simply being multiplied by the inverse of Slip (ie, if slip is 1%, then the output is being multiplied by 99%).

Evolution of the CLP Model

Pegged Model

The simplest method to exchange assets is the pegged model, (1:1) where one asset is exchanged one for another. If there is a liquidity pool, then it can go insolvent, and there is no ability to dynamically price the assets, and no ability to intrinsically charge fees:

$$Eqn8 : y = x$$

Fixed Price Model

The fixed-sum model allows pricing to be built-in, but the pool can go insolvent (run out of money). The amount of assets exchanged is simply the spot price at any given time:

$$Eqn9 : y = \frac{xY}{X}$$

Fixed Product Model

The fixed-product model (Base XYK above), instead bonds the tokens together which prevents the pool ever going insolvent, as well as allowing dynamic pricing. However, there is no intrinsic fee collection:

$$Eqn10 : y = \frac{xY}{x + X}$$

Fixed-Rate Fee Model

The Fixed-Rate Fee Model adds a 30 Basis Point (0.003) (or less) fee to the model. This allows fee retention, but the fee is not liquidity-sensitive:

$$Eqn11 : y = 0.997 * \frac{xY}{x + X}$$

Slip-based Fee Model (CLP)

The Slip-based Fee Model adds liquidity-sensitive fee to the model. This ensures the fee paid is commensurate to the demand of the pool's liquidity, and is the one THORChain uses. The fee equation is shown separate (12b), but it is actually embedded in 12a, so is not computed separately.

$$Eqn12a : y = \frac{xYX}{(x + X)^2}$$

$$Eqn12b : fee = \frac{x^2Y}{(x + X)^2}$$

⚠ The slip-based fee model breaks path-independence and incentivises traders to break up their trade in small amounts.

For protocols that can't order trades (such as anything built on Ethereum), this causes issues because traders will compete with each other in Ethereum Block Space and pay fees to miners, instead of paying fees to the protocol.

It is possible to build primitive trade ordering in an Ethereum Smart Contract in order to counter this and make traders compete with each other on trade size again.

THORChain is able to order trades based on fee & slip size, known as the Swap Queue. This ensures fees collected are maximal and prevents low-value trades.

Benefits of the CLP Model

Assuming a working Swap Queue, the CLP Model has the following benefits:

- The fee paid asymptotes to zero as demand subsides, so price delta between the pool price and reference market price can also go to zero.
- Traders will compete for trade opportunities and pay maximally to liquidity providers.
- The fee paid for any trade is responsive to the demand for liquidity by market-takers.
- Prices inherit an "inertia" since large fast changes cause high fee revenue
- Arbitrage opportunities are democratised as there is a diminishing return to arbitrage as the price approaches parity with reference
- Traders are forced to consider the "time domain" (how impatient they want to be) for each trade.

The salient point is the last one - that a liquidity-sensitive fee penalises traders for being impatient. This is an important quality in markets, since it allows time for market-changing information to be propagated to all market participants, rather than a narrow few having an edge.

Virtual Depths


Balances of the pool (X and Y), are used as inputs for the CLP model. An amplification factor can be applied (to both, or either) in order to change the "weights" of the balances:

Element	Description
a	Input Balance Weight
b	Output Balance Weight

$$Eqn7 : y = \frac{xYbXa}{(x + Xa)^2}$$

If $a = b = 2$ then the pool behaves as if the depth is twice as deep, the slip is thus half as much, and the price the swapper receives is better. This is akin to smoothing the bonding curve, but it does not affect pool solvency in any way. Virtual depths are currently not implemented

If $a = 2$, $b = 1$ then the y asset will behave as though it is twice as deep as the x asset, or, that the pool is no longer 1:1 bonded. Instead the pool can be said to have 67:33 balance, where the liquidity providers are twice as exposed to one asset over the other.

 Virtual Depths and Dissimilar Weighting have not been added to THORChain, because their impact on the **Incentive Pendulum** as well as the loss of revenue to Liquidity Providers has not yet been investigated.

THORChain ruthlessly maximises revenue for itself, taking the perspective that liquidity pools are an incentive **race-to-the-top** as opposed to a fee **race-to-the-bottom**. In typical markets, market-takers are value-extractive from market-makers, whilst in THORChain, market-takers pay handsomely for the privilege of access to liquidity.

Prices

THORChain keeps exchange rates accurate using its CLP design and external arbitrageurs. It does this without fragile external sources like oracles and weighted averages.

This document first explains how THORChain maintains the prices for a single pool. Then it explains how THORChain sets the exchange rate of 2 external assets.

Converging to Reference Prices


This pool has 16,000 MATIC in it and 1,000 RUNE. The price of MATIC on external exchanges is \$0.02. The price of RUNE on external exchanges is \$0.32. The ratio of MATIC to RUNE in the pool is 16:1, and accurately reflects how the world values the two assets.

When a user swaps some MATIC for RUNE. They swap in 2,000 MATIC. Here are those numbers plugged into the swap formula:

$$\frac{2000 * 1000 * 16000}{(2000 + 16000)^2} = 98.76$$

This swapper takes 98.76 RUNE out of the pool. Now the pool has 18,000 MATIC and 901.24 RUNE. This is a ratio of roughly 20:1. But the ratio according to the wider market is still 16:1. This means that swapping MATIC for RUNE on THORChain right now will not be economically rational, since swapping now would yield less RUNE than if swapping on an external exchange. But arbitrageurs will quickly fix the issue with their bots.

Arbitrage bots fix the situation by swapping RUNE for MATIC. When they do this they get MATIC at a lower cost, and can sell it on external markets for a profit. As the amount of RUNE in the pool increases and MATIC decreases, restoring the ratio to 16:1. This reflects the externally-accepted price for the 2 assets.

 In reality, arbitrageurs don't fix the imbalance in one single transaction. This wouldn't be economical. They split the arbitrage swaps into smaller and smaller transactions. This keeps their activity profitable. See [Trading](#) for more.

Getting the Prices for Any Asset

THORChain is also able to determine how much any asset is worth in any other asset simply by using pool balances.

In an example the MATIC:RUNE pool has 16,000 MATIC and 1,000 RUNE. The TUSD:RUNE has 3,300 TUSD and 10,000 RUNE. Based on this information alone, THORChain can know what 1 MATIC is worth in TUSD.

Here's the formula to work out the MATIC price (\$/MATIC):

$$\frac{TUSD_{inPool2}}{RUNE_{inPool2}} * \frac{RUNE_{inPool1}}{MATIC_{inPool1}}$$

Here's that formula with the actual numbers from the pool input:

$$\frac{3300}{10000} * \frac{1000}{16000} = 0.021$$

This information is available within the THORChain network. It doesn't rely on direct external price sources - it works through arbitrageurs and simple market forces.

Here is the formula for working out the exchange rate between any 2 external assets:

$$\frac{AssetInPool1}{RUNEInPool1} * \frac{RUNEInPool2}{AssetInPool2}$$

Advantage Over Alternative Reference Price Designs

THORChain's reference price model differs to other decentralised exchanges. Other exchanges use external oracles and weighted averages to set their prices. This has proven problematic as these external sources become attack vectors.

These external sources are problematic because people can manipulate them. Technical and ecosystem factors can also affect them. For example, the network underlying oracles can become congested and affect their performance. Weighted averages are better than taking direct price references, but large actors can still manipulate them.

THORChain is able to sense both the instantaneous price of an asset, as well as its purchasing power (how much would the asset purchase of another asset if it was instantly sold). The latter is important when it comes to collateralising debt, because the spot price is irrelevant - the purchasing power is needed.

Governance

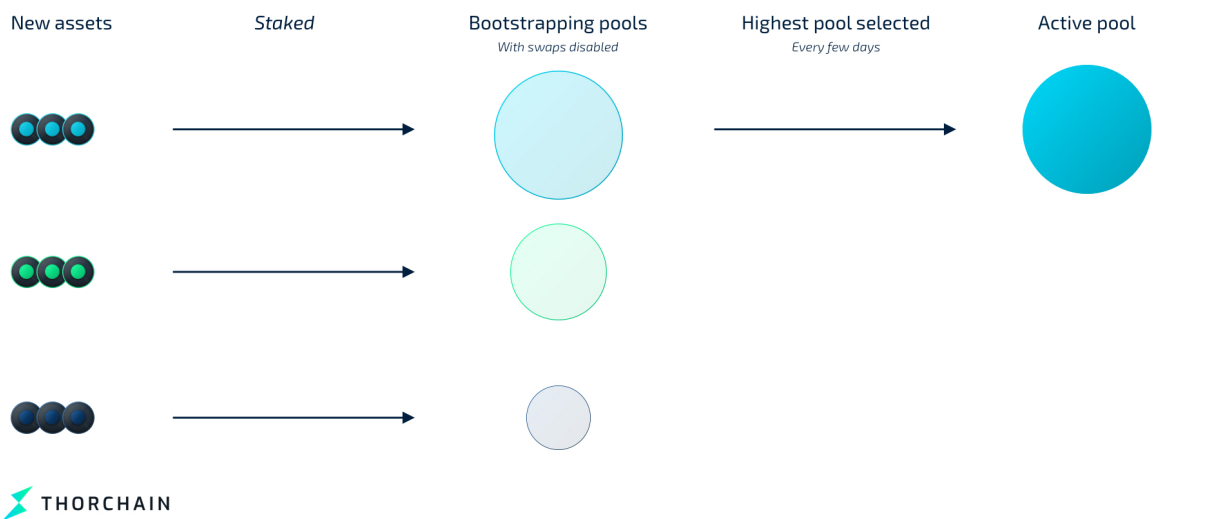
THORChain aims to have as little governance baked into the system as possible. This is done so that nodes don't communicate or learn who one another are. This is important for security because it makes sure that nodes don't act together to take control.

THORChain governance decides:

- which assets are listed/delisted
- which chains are listed/delisted
- when the protocol gets upgraded
- the economic limit – how many nodes can participate

Asset Listing

Listing New Assets



Users signal which assets they want on the network by staking in a new pool. THORChain will realise it is a new asset it hasn't been seen before and create a new pool and place it in bootstrap mode. This is a normal asset pool except swapping is disabled on it. Every few days the networks looks at all the bootstrapping pools and lists the one with the highest value.

Asset Delisting

Assets are delisted when all liquidity providers have taken all their assets out of it, or its pool depth drops too low. The logic is:

1. When a new bootstrap pool is enabled, its depth is compared with the depth of the smallest active pools.
2. If it is deeper, the smallest active pool is placed back into bootstrap mode, and the new pool replaces it.

The process is repeated to re-list an asset.

Chain Listing/Delisting

When the community wants to support a new chain

1. Community developers write a new Bifröst module and propose it via a [THORChain Improvement Proposal \(TIP\)](#)
2. THORChain developer community decides whether or not to approve it
3. If approved, the code gets tested and validated by core developers
4. If accepted, it gets added to THORNode software
5. Nodes upgrade their software as they are cycled off and back onto the network
6. When 67% of nodes are running the new software, the new chain is connected

To delist, nodes stop watching a chain. When 67% are no longer watching, it gets removed. A process begins and the assets of that chain are returned to their owners.

Protocol Upgrades & THORChain Improvement Proposals (TIPs)

Developers from the community submit THORChain Improvement Proposals (TIPs) to improve the network. The community discusses, tests and validates the software. If they decide that the change is beneficial, it's merged into the THORNode software.

The protocol is made up of 3 main pieces, run by the nodes:

- application logic – runs the blockchain
- schema – stores key values of vaults
- network software – keeps the TSS protocol key generation and signing

When nodes are churned off the network they can choose to update their software version. Over time more and more nodes will run the latest version. When 67% of nodes are running the new software, the network is automatically updated. This is how application logic, chain connections and schema are updated.

When upgrading the network software, a certain block number in the future is set when the upgrade will happen. When the network reaches that point, the whole chain stops running and a genesis import to a new network occurs and operations continue normally.

Emergency Changes

Emergency changes are difficult to coordinate because nodes cannot communicate. To handle an emergency, nodes should leave the system. When the number of nodes falls below 4, funds are paid out and the system can be shut-down. This process is called Ragnarök.

Economic Limit

There are only so many nodes who can participate on the network. This is because there's a minimum bond amount and a fixed supply of Rune. If the system is ever found to be always under-bonded or over-bonded, the minimum bond limit can be changed.

Mimir

Mimir is a feature to allow admins to change constants in the chain, such as MinimumBond, ChurnSpeed and more during Chaosnet. When Mimir is destroyed, the chain will be uncapped and in Mainnet.

