Cluster configuration

caution These cluster definition and cluster lock files are a work in progress. The intention is for the files to be standardised for operating distributed validators via theEIP process when appropriate. This document describes the configuration options for running a charon client or cluster.

A charon cluster is configured in two steps:

- cluster-definition.json
- which defines the intended cluster configuration before keys have been created in a distributed key generation ceremony.
- cluster-lock.json
- which includes and extendscluster-definition.json
- with distributed validator BLS public key shares.

In the case of a solo operator running a cluster, thecharon create cluster command combines both steps into one and just outputs the finalcluster-lock.json without a DKG step.

Cluster Definition File

Thecluster-definition.json is provided as input to the DKG which generates keys and thecluster-lock.json file.

Using the CLI

Thecharon create dkg command is used to create thecluster-definition.json file which is used as input tocharon dkg .

The schema of thecluster-definition.json is defined as:

{ "name" :

"best cluster" ,

// Optional cosmetic identifier "creator" :

{ "address" :

"0x123..abfc" ,

//ETH1 address of the creator "config_signature" :

"0x123654...abcedf"

// EIP712 Signature of config_hash using creator privkey } , "operators" :

[{ "address" :

"0x123..abfc" ,

// ETH1 address of the operator "enr" :

"enr://abcdef...12345" ,

// Charon node ENR "config_signature" :

"0x123456...abcdef" ,

// EIP712 Signature of config_hash by ETH1 address priv key "enr_signature" :

"0x123654...abcedf"

// EIP712 Signature of ENR by ETH1 address priv key }] , "uuid" :

"1234-abcdef-1234-abcdef" ,

// Random unique identifier. "version" :

"v1.2.0" ,

// Schema version "timestamp" :

"2022-01-01T12:00:00+00:00" ,

// Creation timestamp "num_validators" :

2 ,

// Number of distributed validators to be created in cluster-lock.json "threshold" :

3 ,

// Optional threshold required for signature reconstruction "validators" :

[{ "fee_recipient_address" :

"0x123..abfc" ,

// ETH1 fee_recipient address of validator "withdrawal_address" :

"0x123..abfc"

// ETH1 withdrawal address of validator } , { "fee_recipient_address" :

"0x123..abfc" ,

// ETH1 fee_recipient address of validator "withdrawal_address" :

"0x123..abfc"

// ETH1 withdrawal address of validator }] , "dkg_algorithm" :

"foo_dkg_v1" ,

// Optional DKG algorithm for key generation "fork_version" :

"0x00112233" ,

// Chain/Network identifier "config_hash" :

"0xabcfde...acbfed" ,

// Hash of the static (non-changing) fields "definition_hash" :

"0xabcdef...abcedef"

// Final hash of all fields }

Using the DV Launchpad

- Aleader/creator
- , that wishes to coordinate the creation of a new Distributed Validator Cluster navigates to the launchpad and selects "Create new Cluster"
- Theleader/creator
- uses the user interface to configure all of the important details about the cluster including:* TheWithdrawal Address
- - for the created validators
- - TheFee Recipient Address
- - for block proposals if it differs from the withdrawal address
- - The number of distributed validators to create
- - The list of participants in the cluster specified by Ethereum address(/ENS)
- - The threshold of fault tolerance required
- These key pieces of information form the basis of the cluster configuration. These fields (and some technical fields like DKG algorithm to use) are serialized and merklized to produce the definition'scluster_definition_hash
- . This merkle root will be used to confirm that there is no ambiguity or deviation between definitions when they are provided to charon nodes.
- Once theleader/creator
- is satisfied with the configuration they publish it to the launchpad's data availability layer for the other participants to access. (For early development the launchpad will use a centralized backend db to store the cluster configuration. Near production, solutions like IPFS or arweave may be more suitable for the long term decentralization of the launchpad.)

Cluster Lock File

Thecluster-lock.json has the following schema:

{ "cluster_definition" :

{ ... } ,

// Cluster definiition json, identical schema to above, "distributed_validators" :

[

// Length equal to cluster_definition.num_validators. { "distributed_public_key" :

"0x123..abfc" ,

// DV root pubkey "public_shares" :

[

"abc...fed" ,

"cfd...bfe"] ,

// Length equal to cluster_definition.operators "fee_recipient" :

"0x123..abfc"

// Defaults to withdrawal address if not set, can be edited manually }] , "lock_hash" :

"abcdef...abcedef" ,

// Config_hash plus distributed_validators "signature_aggregate" :

"abcdef...abcedef"

// BLS aggregate signature of the lock hash signed by each DV pubkey. }

Cluster Size and Resilience

The cluster size (the number of nodes/operators in the cluster) determines the resilience of the cluster; its ability remain operational under diverse failure scenarios. Larger clusters can tolerate more faulty nodes. However, increased cluster size implies higher operational costs and potential network latency, which may negatively affect performance

Optimal cluster size is therefore trade-off between resilience (larger is better) vs cost-efficiency and performance (smaller is better).

Cluster resilience can be broadly classified into two categories:

- Byzantine Fault Tolerance (BFT)
  - the ability to tolerate nodes that are actively trying to disrupt the cluster.
- Crash Fault Tolerance (CFT)
  - the ability to tolerate nodes that have crashed or are otherwise unavailable.

Different cluster sizes tolerate different counts of byzantine vs crash nodes. In practice, hardware and software crash relatively frequently, while byzantine behaviour is relatively uncommon. However, Byzantine Fault Tolerance is crucial for trust minimised systems like distributed validators. Thus, cluster size can be chosen to optimise for either BFT or CFT.

The table below lists different cluster sizes and their characteristics:

- Cluster Size
  - the number of nodes in the cluster.
- Threshold
  - the minimum number of nodes that must collaborate to reach consensus quorum and to create signatures.
- BFT #
  - the maximum number of byzantine nodes that can be tolerated.

- CFT #
-
    - the maximum number of crashed nodes that can be tolerated.

Cluster Size Threshold BFT # CFT # Note 1 1 0 0 ✖ Invalid: Not CFT nor BFT! 2 2 0 0 ✖ Invalid: Not CFT nor BFT! 3 2 0 1 ⚠ Warning: CFT but not BFT! 4 3 1 1 ✅ CFT and BFT optimal for 1 faulty 5 4 1 1 6 4 1 2 ✅ CFT optimal for 2 crashed 7 5 2 2 ✅ BFT optimal for 2 byzantine 8 6 2 2 9 6 2 3 ✅ CFT optimal for 3 crashed 10 7 3 3 ✅ BFT optimal for 3 byzantine 11 8 3 3 12 8 3 4 ✅ CFT optimal for 4 crashed 13 9 4 4 ✅ BFT optimal for 4 byzantine 14 10 4 4 15 10 4 5 ✅ CFT optimal for 5 crashed 16 11 5 5 ✅ BFT optimal for 5 byzantine 17 12 5 5 18 12 5 6 ✅ CFT optimal for 6 crashed 19 13 6 6 ✅ BFT optimal for 6 byzantine 20 14 6 6 21 14 6 7 ✅ CFT optimal for 7 crashed 22 15 7 7 ✅ BFT optimal for 7 byzantine The table above is determined by the QBFT consensus algorithm with the following formulas fromthis paper:

n = cluster size

Threshold: min number of honest nodes required to reach quorum given size n Quarom(n) = ceiling(2n/3)

BFT #: max number of faulty (byzantine) nodes given size n f(n) = floor((n-1)/3)

CFT #: max number of unavailable (crashed) nodes given size n crashed(n) = n - Quorum(n)Edit this page Previous Distributed Key Generation Next Charon networking