Blaize.Security

DUSK NETWORKING PROTOCOL: KADCAST SECURITY AUDIT

April 17th, 2024 / V. 1.0

Table of Contents

Executive Summary

During the audit, we examined the security of the Kadcast - Dusk networking protocol. Our task was to find and describe any security issues in the protocol. This report presents the findings of the security audit of the **Kadcast** protocol conducted between **January 30th, 2024 and February 26th, 2024.** The second iteration of the audit connected to the review of the protocol upgrade and consulting with the Dusk team occurred between **March 25th, 2024 and April 10th, 2024.**

Blaize.Security team conducted the audit of the Kadcast network protocol implemented by the Dusk network. The audit's goal was to review the implementation against the original specification (https://eprint.iacr.org/2021/996.pdf) and to check the codebase for inconsistencies, security issues, vulnerabilities, or business logic flaws. The security team worked simultaneously on three main directions: system analysis, review of each module and comparison with the specification; line-by-line code review; and simulation testing of the protocol on the local network (both with Dusk network node for formal verification of the compatibility and with light test clients for sorrow protocol testing).

The audit team examined the system architecture, ensuring the correct relationship between protocol modules, their interfaces, types relations tree, and used data structures. The team researched the implemented messaging protocol (with a main focus on functionality around FIND_NODES messages processing), the message structure (with inspection of marshaling rules and header structure), the implementation of the BinaryID (including all checks regarding the difficulty of the generated ID) and distance calculation. The security team reviewed the available configuration and settings and the initial setup of the protocol.

During the testing stage, the team reviewed all native unit tests, conducted local testing of functions (especially connected to the BinaryIDy processing), and performed a simulation of the internal network with different configurations, watching the messages delivery process within the system and nodes' connection process.

The security team discovered several issues:

- A few deviations from the specification (verified by the team at later stages);
- A lack of pings for the idle nodes and deviation from the peer's eviction process;
- A few local code issues connected with best practices;
- An ambiguous approach with the reserved fields in the message header;
- Local issues with the BinaryID processing and difficulty verification.

The Dusk team verified most of the issues and provided the upgrades for the rest. However, auditors should note that some updates to the protocol will be introduced in the next version (including the new model for the reserved fields for message headers and changes for difficulty verification during BinaryID processing).

The Kadcast protocol successfully passed the security audit, and the security team noted the high code quality of the codebase, high functionality of each component, excellent work with resources, flexible configuration, and settings, and the availability of a certain number of native tests. However, the security team recommends adding more documentation, especially natspec comments. The repository contains several readmes, and the implementation follows the original specification. However, several design and implementation decisions required additional consultations with the Dusk team, thus pointing to the need for extended documentation.

From all points of view, the protocol shows high compliance with the security standards.

Auditing Strategy and Techniques Applied/Procedure

Blaize.Security auditors start the audit by developing an auditing strategy - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

MANUAL AUDIT STAGE

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- System analysis for components/modules co-dependencies;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Order-dependency and time-dependency of operations
- Math operations and calculations analysis, formula modeling;
- Review of dependencies, 3rd parties, and integrations;
- Storage structure and stored data structure review (including read/write operations);
- Review global settings for the usage/mis-using within the protocol (check for necessity, ambiguity and mis-using) and the default values;
- Basic validations of parameters of functions/methods/messages (or their analogs)
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Code quality, documentation, and consistency review.

FOR ADVANCED COMPONENTS:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

TESTING STAGE:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory/simulation tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

Audit Rating

Score: **9.8** /10

	RATING
Security	**9.8**
Logic optimization	9.9
Code quality	10
Testing suite	9.2
Documentation	9.7

Security: General mark for the security of the protocol.
The main mark for the audit qualification.

Logic optimization: Evaluation of how optimal the implementation is, including presence of extra/unused code, uncovered/extra cases, gas (or its analog) optimization, memory management optimization, etc
Code quality: Evaluation of best practices followed, code readability, structure and convenience of further development
Testing suite: Availability of the native tests suite, level of logic coverage, checks of critical areas being covered.
Documentation: Availability and quality of the documentation, coverage of core functionality and user flows: whitepaper, gitbook, readme, specs, natspec, comments in the code and other possible forms of documentation.

SECURITY RATING CALCULATION

Approximate weight of unresolved issues.
Critical: -3 points
High: -2 points
Medium: -1 points
Low: -0.25 points
Informational: -0.1 point

Note: additional concerns, violated checklist items (including standard vulnerabilities), and verified backdoors may influence the final mark and weight of certain issues.

Starting with a perfect score of 10:

Critical issues: 0 issues (0 resolved): 0 points deducted

High issues: 2 issues (2 resolved): 0 points deducted.

Medium issues: 2 issues (2 verified): 0 points deducted

Low issues: 1 issue (1 resoled): 0 points deducted

Informational issues: 9 issues (6 resolved, 1 verified): -0.2 points deducted

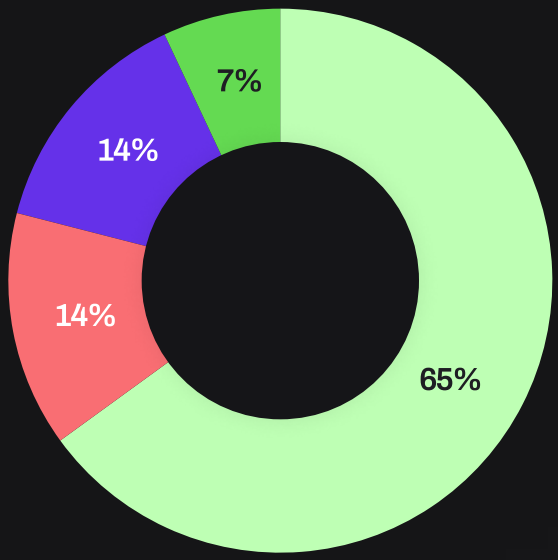
Other: 0 points deducted

Security rating = 10 -0. 2= 9.8

Technical Summary

THE GRAPH OF VULNERABILITIES DISTRIBUTION:

- Critical
- High
- Medium
- Low
- Info



The table below shows the number of the detected issues and their severity. A total of 14 problems were found. 12 issues were fixed or verified by the Customer's team.

	FOUND	FIXED/VERIFIED
Critical	0	0
High	2	2
Medium	2	2
Low	1	1
Info	9	7

SEVERITY DEFINITION

CRITICAL

The system contains several issues ranked as very seriousand dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

HIGH

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

MEDIUM

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

LOW

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

INFO

The issue has no impact on the contract's ability to operate, yet is relevant for best practices. Or this status can be assigned to the issues related to the suspicious activity or substandard business logic decisions which cannot be classified without the comments from the team (and can be re-classified on the later audit stages).

Issues reviewed by the team can get the next statuses:
Resolved: issue is resolved by an appropriate patch or changes in the business logic
Verified: the team provided sufficient evidences that the issue describes desired behavior
Unresolved: neither path nor comments provided by the team, or they are not sufficient to resolve the issue
Acknowledged: the team accepts the misbehavior and connected risks

Audit Scope

Language/Technology: **Rust**

Blockchain: **Dusk**

The scope of the project includes:

- src/handling.rs
- src/kbucket/bucket.rs
- src/transport.rs
- src/lib.rs
- src/kbucket.rs
- src/transport/encoding/raptorq/decoder.rs
- src/kbucket/key.rs
- src/config.rs
- src/maintainer.rs
- src/encoding/payload/nodes.rs
- src/encoding/message.rs
- src/transport/sockets.rs
- src/peer.rs
- src/rwlock.rs
- src/transport/encoding/raptorq/encoder.rs
- src/transport/encoding/raptorq.rs
- src/encoding/header.rs
- src/kbucket/node.rs
- src/encoding/payload/broadcast.rs
- src/transport/encoding.rs
- src/transport/encoding/raptorq.md
- src/transport/encoding/plain_encoder.rs
- src/encoding.rs
- src/encoding/payload.rs

Repository: https://github.com/dusk-network/kadcast

The source code of the smart contract was taken from the branch: **main**

Initial commit:

- 1fd4d87e5666cd002a9756d19b90d7200a08e296

Final commit:

- 5680bffc6946384f54921bd31aa70d9e9eaf0192

Protocol overview

DESCRIPTION

Currently audited codebase represents the Rust implementation of the Kadcast algorithm for the Dusk network. The implementation follows the specification provided in the paper: https://eprint.iacr.org/2021/996.pdf

PROTOCOL SETTINGS / CONFIGURATION

Most protocol settings are located in the config.rs file and they are implemented in a form of few constants and config objects.

Constants

1) Allowed timeframe for the node to be considered “alive”
During this timeframe (from the last node successful ping) the node cannot be requested for eviction.
Implemented as:
BUCKET_DEFAULT_NODE_TTL_MILLIS (default value is 30,000 ms)
Is used as a default value for **BucketConfig.node_ttl**

2) “Cooldown” after the node requested to eviction
During this period (from the timestamp of eviction request) the node cannot be evicted.
Implemented as:
BUCKET_DEFAULT_NODE_EVICT_AFTER_MILLIS (default value 5000 ms)
Is used as a default value for **BucketConfig.node_evict_after**

3) Timeframe, after which the node is considered as idle
Characteristic which partly duplicates the sense of the allowed “alive” timeframe and marks the period of time after which current peer can give up on requesting idle node.
Implemented as:
BUCKET_DEFAULT_TTL_SECS (default value 3600 sec)
Is used as a default value for **BucketConfig.bucket_ttl**

4) Enabling broadcast propagation
Flag to mark the strategy on broadcasting incoming messages further for the stored tree of nodes.
Implemented as:
ENABLE_BROADCAST_PROPAGATION (default value true)
Is used as a default value for **Config.auto_propagate**

5) Capacity of the internal communication channel
Default value for tokio::sync::mpsc::channel primitive - channel capacity
Implemented as:
ENABLE_BROADCAST_PROPAGATION (default value 1000 messages for mpsc::channel)
Is used as a default value for **Config.channel_size**

6) Number of peer ping retries
Implemented as
DEFAULT_SEND_RETRY_COUNT (default value 3 retries)
Is used as a default value for **NetworkConfig.udp_send_retry_count**

7) Interval between retries of the peer
Implemented as
DEFAULT_SEND_RETRY_SLEEP_MILLIS (default value 5 ms)
Is used as a default value for **NetworkConfig.udp_send_retry_interval**

8) Default fetching “cooldown” for the local blocklist update.
Implemented as
DEFAULT_BLOCKLIST_REFRESH_SECS (default value 10 s)
Is used as default value for **NetworkConfig.blocklist_refresh_interval**

CONFIGURATION AND SETTINGS

Protocol Config

The structure which keeps settings for the Peer level:

- **kadcast_id**: supposed to represent the network id of the chain/network, which uses Kadcast as a transport/communication layer and is supposed to be validated outside of the protocol. It is supposed to be used as Node.network_id and match the id of the blockchain network. However, there are no restrictions for it, so it generally separates peers within a network from outer networks. The setting is not set by default.
- **public_address**: public address of the peer. It defaults to localhost, 9000, for testing convenience. It is used as PeerNode.address (PeerInfo.address) as is one of main peer characteristics as is a default address to listen messages from.
- **listen_address**: an additional parameter for the public address to receive incoming messages that override the “public_address.” It can be used in case a peer is not publicly reachable. Not set by default
- **bootstrapping_nodes**: list of default public nodes to receive the initial set of available peers. Thus, the node will send the first FIND_NODES message to available bootstrap nodes. It is empty by default; it is the responsibility of the network/chain to fill it up.
- **auto_propagate**: parameter for the strategy for broadcasting incoming messages. It is defaulted to the constant.
- **channel_size**: The number of messages the node handles in the communication channel (for the tokio::sync::mpsc::channel primitive). It is defaulted to the constant.
- **recursive_discovery**: This parameter handles participation in the recursive discovery of the nodes. Nodes with this parameter turned on will re-propagate the FIND_NODES messages; otherwise, they will just PING nodes from the stored tree to check their status. It is defaulted to the constant.

Bucket Config

The structure which keeps the settings of the Bucket level:

- **node_ttl**: characteristic that defines the period during which the node (peer stored in the tree) cannot be evicted from the tree (from the moment of the last refresh/ping). Defaulted to the constant
- **node_evict_after**: This setting represents the minimum cooldown period after a node (a peer stored in the tree) eviction request is made before the actual eviction occurs. It is set to a default constant value.
- **bucket_ttl**: characteristic for the period to check if the node is idle. It is used by the maintainer service to ping nodes (peers stored in the tree) periodically. It is defaulted to the constant

Network Config

The structure which keeps the settings of the transport level:

- **udp_recv_buffer_size**: the default size of the buffer for received UDP messages. Value for the standard socket2::SockRef primitive. Defaulted to 5000000
- **udp_send_backoff_timeout**: timeout before sending the UDP message. The setting has no default value.
- **udp_send_retry_interval**: timeout before re-sending the UDP message after the failure, characteristic for the retry. The setting is set to the default constant.
- **udp_send_retry_count**: number of retries to send the UDP message. The setting is set to the default constant.
- **blocklist_refresh_interval**: defines the period to fetch the local blocklist from the library (to prevent self-DDoS or at least to decrease the resources load). The setting is set to the default constant.

FECConfig

The structure which keeps the settings of the transport level:

- **encoder**, **decoder**: placeholders for default encoder/decoder configs for the transport layer

Local Settings

transport.rs

1) MAX_DATAGRAM_SIZE - constant representing the size of the incoming UDP datagram which will be decoded and unmarshalled by the protocol

lib.rs

1) K_ALPHA - the redundancy factor for nodes lookup. However in the Dusk implementation of the Kadcast is used as a number of random nodes to pick from the idle basket. Has default value of 3.
2) K_BETA - redundancy factor for lookup during the broadcast. Has default value of 3.
3) K_K - maximum size of the bucket. Has default value of 20.

sockets.rs

1) MIN_RETRY_COUNT - number of retries of sending the UDP packet

raportq.rs

1) TRANSMISSION_INFO_SIZE - Size of the ObjectTransmissionInformation structure, representing the RaptorQ header. Has default value of 12.
2) UID_SIZE - Size of the hash of the broadcast payload (32).
3) MIN_ENCODING_PACKET_SIZE - Minimum size of the encoding packet. Has default value of 5.
4) MIN_CHUNKED_SIZE - Minimum size for a chunked payload, calculated as the sum of UID size, transmission info size, and min encoding packet size.

encoder.rs

1) DEFAULT_MIN_REPAIR_PACKETS_PER_BLOCK - Value for the minimum number of repair packets per block in RaptorQ encoding.Has default value of 5.
2) DEFAULT_MTU - Value for the maximum transmission unit of the protocol data unit that can be transmitted in a single network layer transaction. Has default value of 1300.
3) DEFAULT_FEQ_REDUNDANCY - Redundancy factor for forward error correction. Has default value of 0.15.

decoder.rs

1) DEFAULT_CACHE_TTL_SECS - Time-to-live for entries in the cache (60 seconds)
2) DEFAULT_CACHE_PRUNE_EVERY_SECS - interval for cache pruning (5 minutes)

PROTOCOL PRIMITIVES: PAYLOAD

nodes.rs

Specifies the PeerEncodedInfo primitive (the minimal broadcasting information on the peer for the Kadcast routing tree) and NodePayload primitive (vector or all peers known to the node). Both structures are used during the messages exchange between peers and implement marshaling crate.

PeerEncodedInfo

Storage:
1) ip: IpInfo stract containing IPv4 or IPv6 info (4 bytes for v4 and 16 bytes for v6)
2) port number (2 bytes number)
3) id: BinaryKey type unique id (derived from Peer)

Marshallable spec:
1) IP type byte (0 for IPv6): 1 byte
2) IP bytes: 4 bytes for IPv4 OR 16 bytes for IPv6
3) port: 2 bytes
4) id: 16 bytes, encoded in constant K_ID_LEN_BYTES

to_socket_address():
- method to return standard Rust SocketAddr object from the peer info

NodePayload

Storage:
1) peers: vector of PeerEncodedInfo objects

Marshallable spec:
1) number of peers stored: 2 bytes
2) consequent peers info serialized based on PeerEncodedInfo marshalling rules

broadcast.rs

Specifies the BroadcastPayload primitive - general frame format for data exchange between peers

BroadcastPayload

Storage:
1) height - Kadcast broadcast height by default, or custom bucket height (if override required)
2) gossip frame - the packet prepared via the RaptorQEncoder

Marshallable spec:
1) height: 1 byte
2) gossip frame length: 4 bytes
3) gossip frame: size specified in the previous field

payload.rs

Re-exports all structures from payload primitives: IpInfo, PeerEncodedInfo, BroadcastPayload and NodePayload

PROTOCOL PRIMITIVES: MESSAGE

header.rs

Specifies the Header primitive - single standard element for every message used within the protocol.

Header

Storage:
1) binary_id: id of the peer in the k-bucket
2) sender_port
3) network_id - network id used within the protocol (derived from the Config.kadcast_id
4) reserved bytes

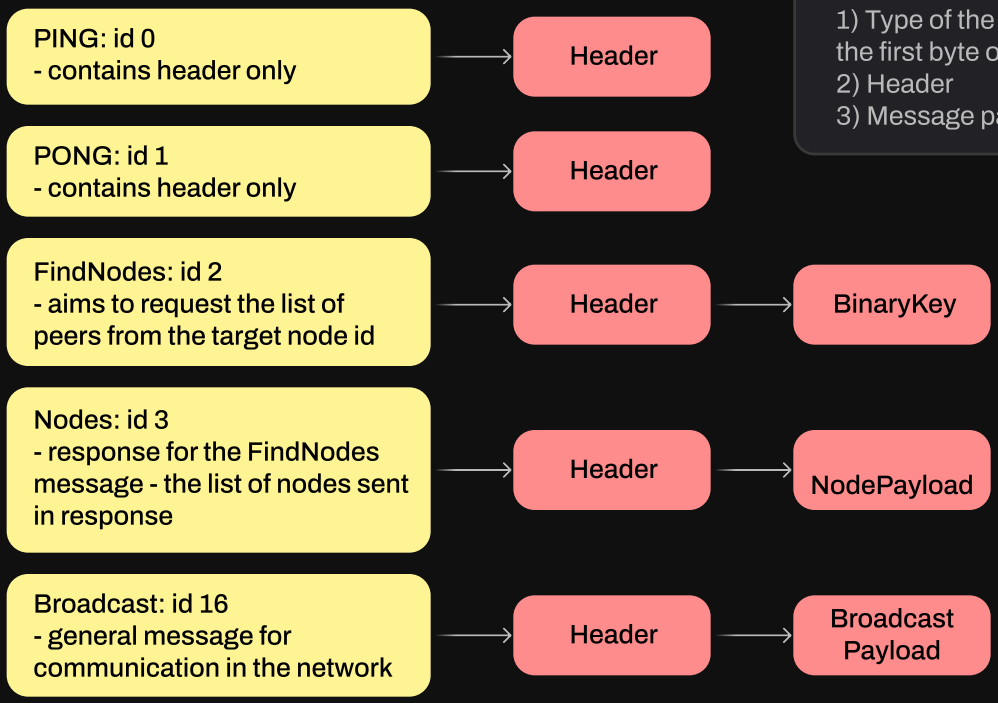
Marshallable spec:
1) ID of the peer: 16 bytes, encoded in constant K_ID_LEN_BYTES
2) Nonce assigned to peer: 4 bytes, encoded in constant K_NONCE_LEN
3) port: 2 bytes
4) network id: 1 byte
5) reserved bytes: 2 bytes

message.rs

Specifies the Message primitive - the format of information exchange between peers in the network. For now, the protocol specifies 5 types of messages

Message

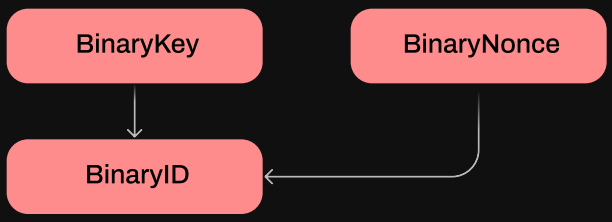
Marshallable spec:
1) Type of the message is encoded in the first byte of the message: 1 byte
2) Header
3) Message payload (by type)



PROTOCOL PRIMITIVES: ID

key.rs

Defines structures for the identifying peers in k-buckets stored in the node.
Contains 2 parts:
1) BinaryKey - the path to the node in the peers-tree (k-bucket). Is specified as an array of bytes with length of 16 bytes (encoded in constant K_ID_LEN_BYTES). Is generated in PeerNode as hash of address+port of the peer
Stores key from the "leaf" to the root.
2) BinaryNonce - the nonce used as proof of validity of the peer - as it is generated via PoW primitive to correspond the network difficulty. Is specified as an array of bytes with length of 4 bytes (encoded in constant K_NONCE_LEN)



Storage:
1) bytes: record of BinaryKey
2) nonce: record of BinaryNonce

as_binary(): returns stored binary key
nonce(): returns stored nonce
from_nonce(): creates a new BinaryID from key and nonce

calculate_distance(): returns the 0-based Kadcast distance between two BinaryKeys - the process described in part III, fig. 1 of https://eprint.iacr.org/2021/996.pdf
Returns the height of the bucket of the node relatively to the current one

msb(): utility function used for leading zeros check

1) take 2 BinaryKeys (unique ids of peers)
2) convert to binary form
3) XOR both keys byte to byte
4) calculate leading 0s = msb()
5) relative BucketHeight is 128 - msb() - 1

verify_difficulty()
verify_nonce()
Utility functions to verify that the **hash of the concatenation of BinaryKey and BinaryNonce** meets the difficulty parameter assigned by the network to avoid overload of the network by new Peers. By definition - if the number of trailing zeros is higher than the desired difficulty.
Note: by default the required difficulty is encoded in the constant **K_DIFF_MIN_BIT = 8.**
Note: verify_difficulty expects the "reverted" bytes array - it checks trailing zeros, which are leading for the checked value

generate(): generates nonce for the provided BinaryKey to fulfill the requirement on the difficulty in the network. Starts from nonce 0 and iterates until the requirement is fulfilled. Verifies difficulty against the constant **K_DIFF_PRODUCED_BIT** = 8

PROTOCOL PRIMITIVES: NODE

node.rs

Defines the Node - an abstract entity associated with the BinaryID. It carries own information about the eviction status and the last assessed timestamp. It is implemented as a template, thus the Node can carry any information and is abstract regarding the entity to be used as peer.

Node<T>

Storage:
1) id: record of BinaryID
2) value: abstracted value carried by the Node
3) eviction status: record if the eviction was requested for the record or empty otherwise
4) seen_at: timestamp of the last assessment
5) network_id: derived from PeerNode is a network id used in protocol

Note: Node is an abstract structure which carries no checks for eviction status, proves of availability (e.g. during creation) or pre-checks for refreshing - all checks are delegated to structures in control with nodes (via the available interface)

new(): default constructor
id(), value(): getters

calculate_distance(): wrapper around same function from the id

is_alive(): compares if the node was assessed within certain "duration" period from now

refresh(): delete eviction request (if one was set) and refresh the seen_at timestamp

flag_for_check(): set request for eviction

BUCKET FUNCTIONALITY

bucket.rs

Defines the K-bucket - main structure of the algorithm for storing the routing table (tree). Is implemented as Rust native ArrayVec of nodes with:

1. additional placeholder for the pending node
2. with the most recently assessed node in the end of the vector and the least recently assessed node as the first one
3. functionality for inserting and removing nodes

Bucket<V>

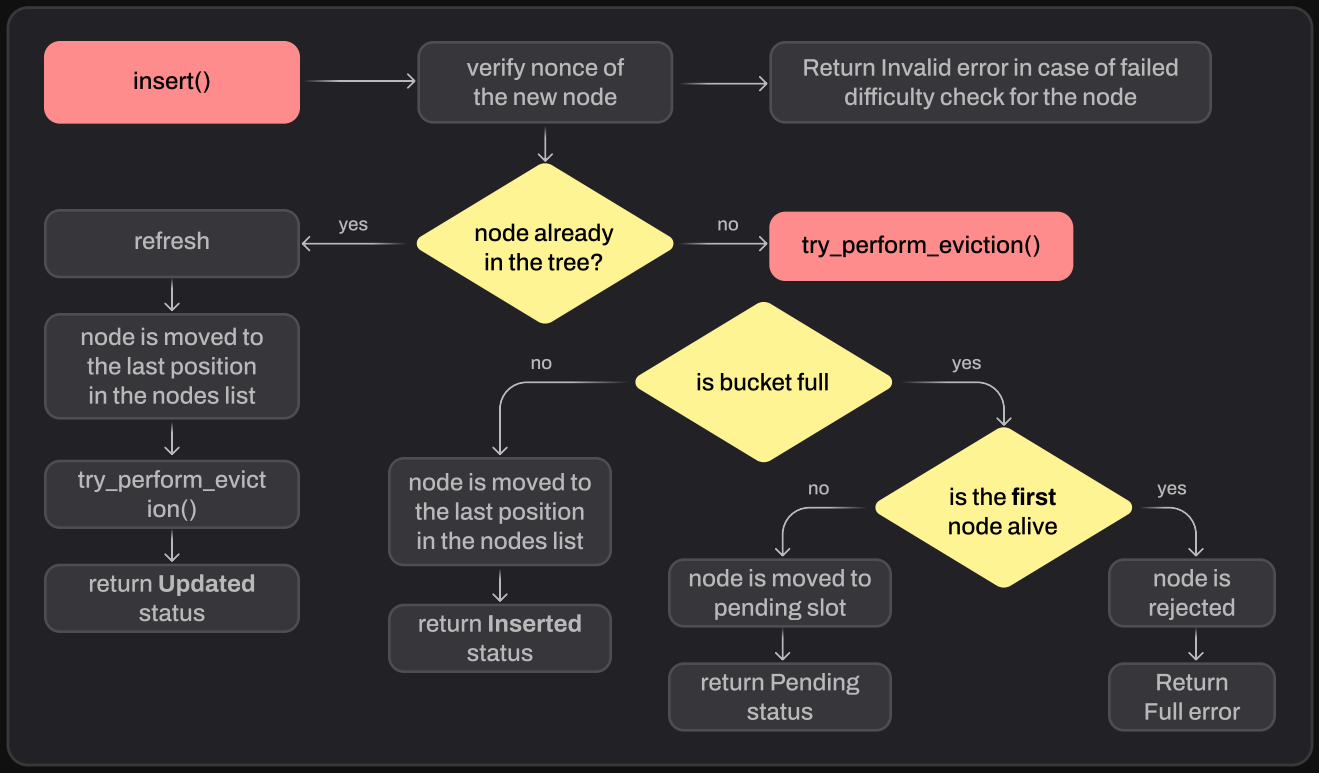
Storage:
1) nodes: ArrayVec of Node objects
2) pending_node: placeholder for the node currently queried to be added to the bucket
3) bucket_config: BucketConfig object (defined in the config)

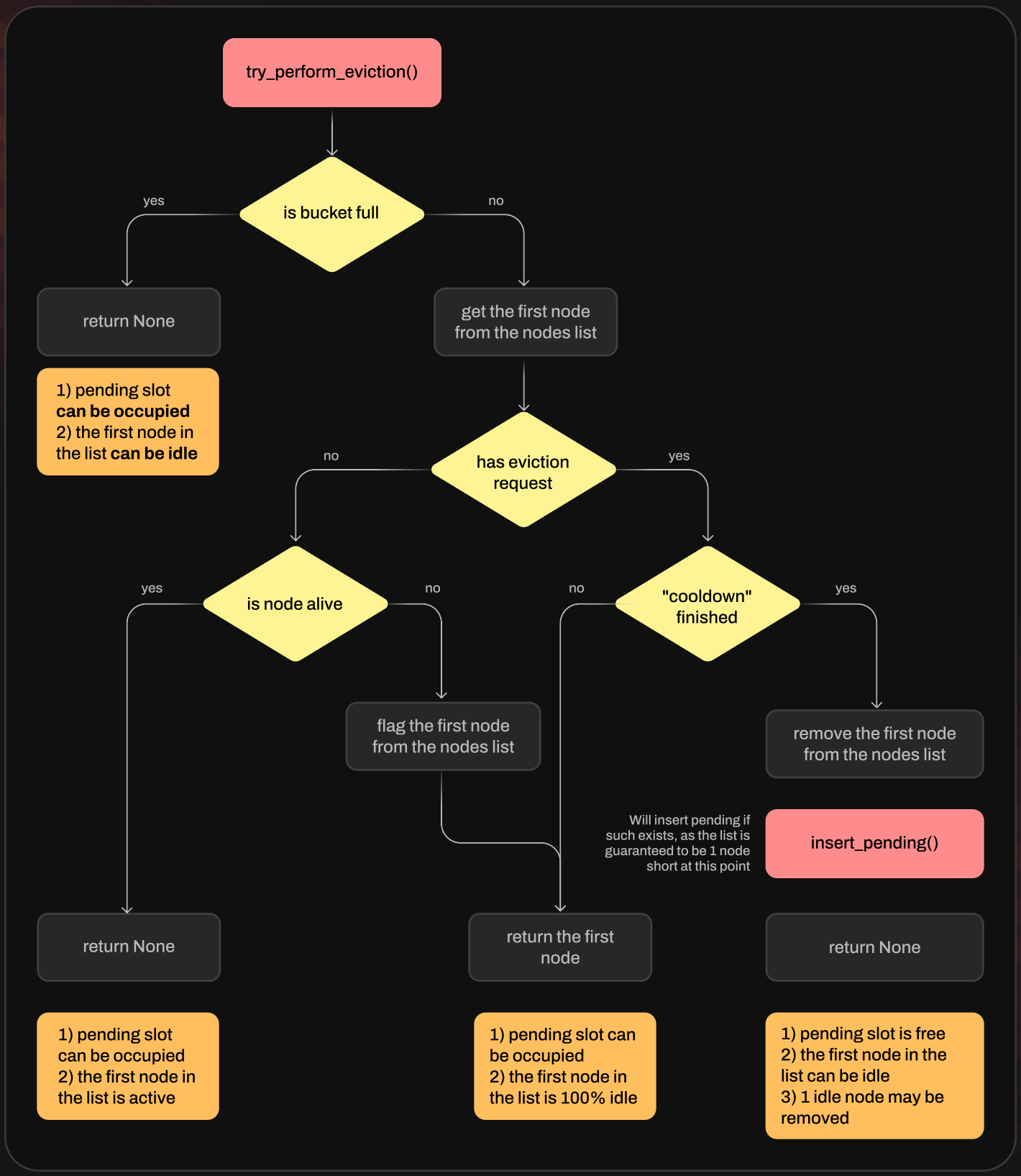
By default (via function new()) is created with empty nodes array and empty pending_node

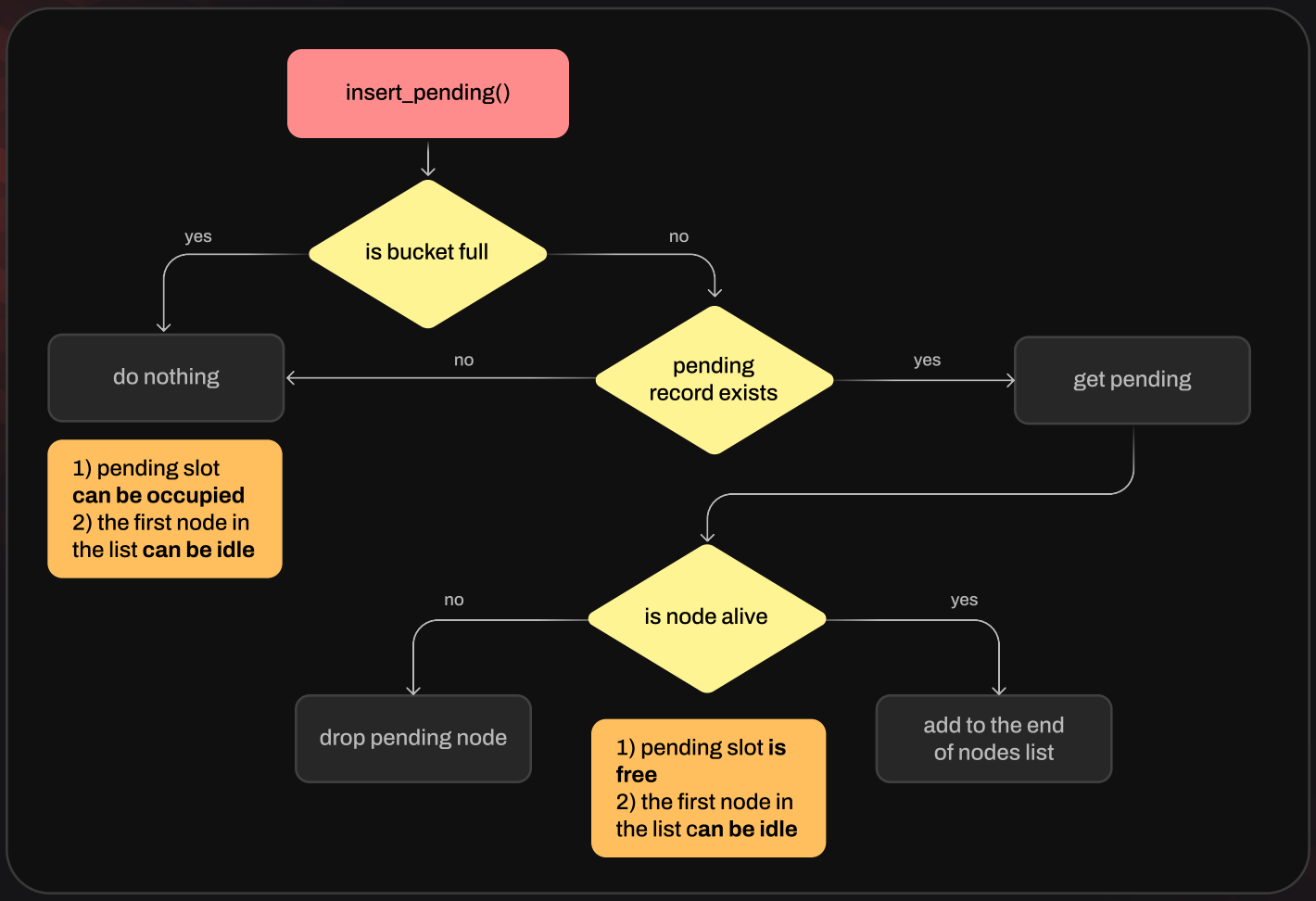
pending_eviction_node(): first node if it has the requested eviction
peers(): iterator over nodes
has_idle(): checks if the first node is idle
alive_nodes(): returns responding nodes
has_node(): checks if there is a node with BinaryKey in the bucket
is_full(): checks if the bucket is full

calculate_distance(): wrapper around same function from the id

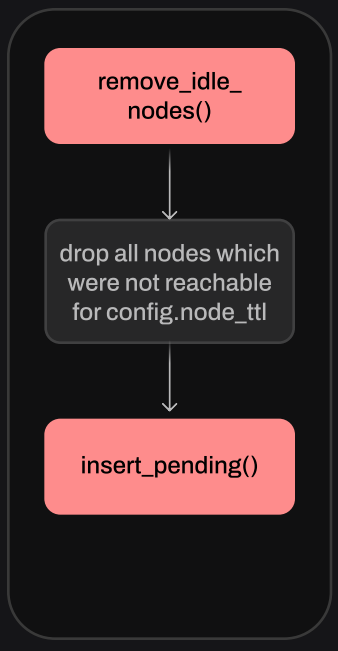
pick(): chooses random set of nodes by specified parameter

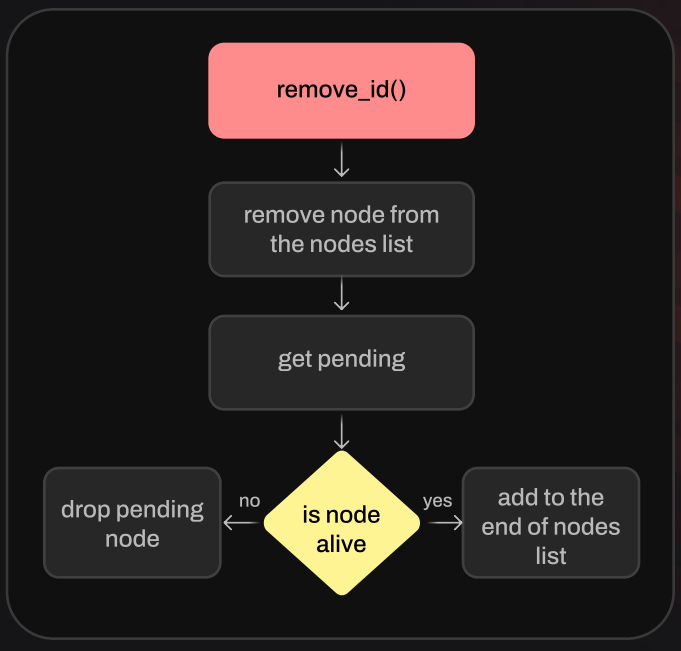



try_perform_eviction()
is bucket full
yes
no
return None
1) pending slot **can be occupied**
2) the first node in the list **can be idle**
get the first node from the nodes list
has eviction request
no
yes
is node alive
yes
no
flag the first node from the nodes list
return None
1) pending slot can be occupied
2) the first node in the list is active
"cooldown" finished
no
yes
remove the first node from the nodes list
Will insert pending if such exists, as the list is guaranteed to be 1 node short at this point
insert_pending()
return the first node
1) pending slot can be occupied
2) the first node in the list is 100% idle
return None
1) pending slot is free
2) the first node in the list can be idle
3) 1 idle node may be removed


insert_pending()
is bucket full
yes
no
do nothing
pending record exists
no
yes
get pending
1) pending slot **can be occupied**
2) the first node in the list **can be idle**
is node alive
no
yes
drop pending node
1) pending slot **is free**
2) the first node in the list **can be idle**
add to the end of nodes list


refresh()
node is moved to the last position in the nodes list


remove_idle_ nodes()
drop all nodes which were not reachable for config.node_ttl
insert_pending()


remove_id()
remove node from the nodes list
get pending
drop pending node
no
is node alive
yes
add to the end of nodes list

BUCKET FUNCTIONALITY: KADCAST TREE

kbucket.rs

Defines the tree (set of K-Buckets)
1. holds buckets from the height 1 to the max bucket height
2. wraps bucket functionality (for each bucket height)
3. implements the Kadcast tree structure

Tree<V>

Storage:
1) root: Node object, the node representing current peer
2) buckets: hashmap of k-buckets
3) bucket_config: BucketConfig object (defined in the config)

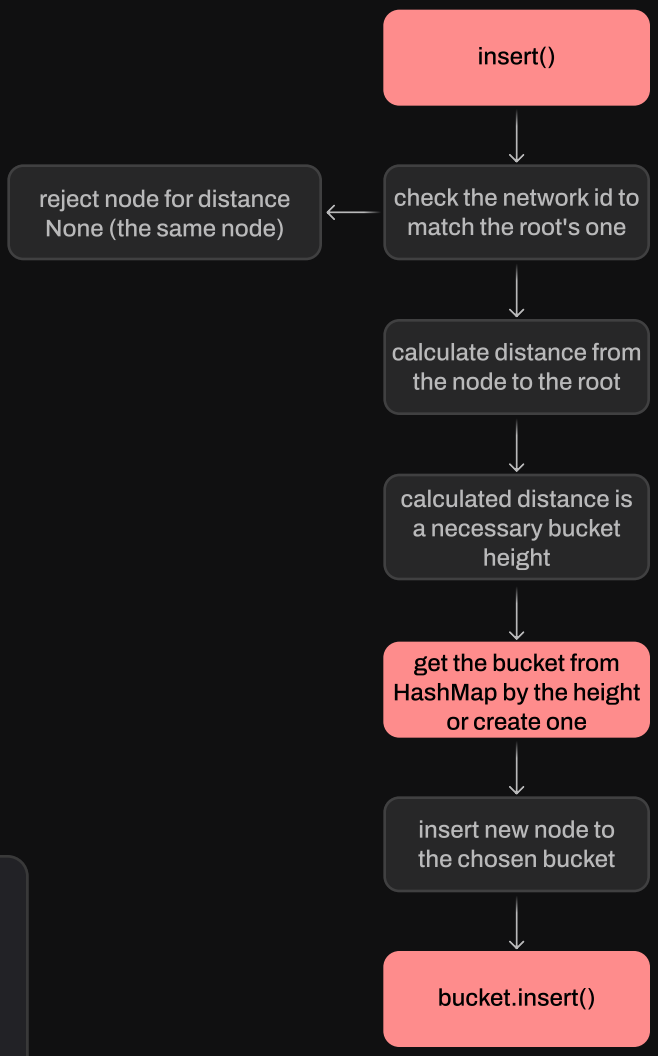
extract(): pick at most k-beta nodes (by constant specified) from each bucket (via bucket.pick())
Note: extracts nodes from buckets with height lower than the parameter max_h

idle_buckets(): pick at most k-alpha nodes (by constant specified) from each bucket which has idle nodes

closest_peers(): get sorted list with closest peers based on the distance between peers in the tree (between BinaryKeys). Array will be limited to ITEM_COUNT length

other public methods:

insert()
remove_idle_nodes(): wrap around bucket.remove_idle() (for all buckets)
remove_peer(): wrap around bucket.remove_id()
has_peer(): wrap around bucket.has_node()
is_bucket_full(height): wrap around bucket.is_full()
alive_nodes(): wrap around bucket.alive_nodes()



PEER

peer.rs

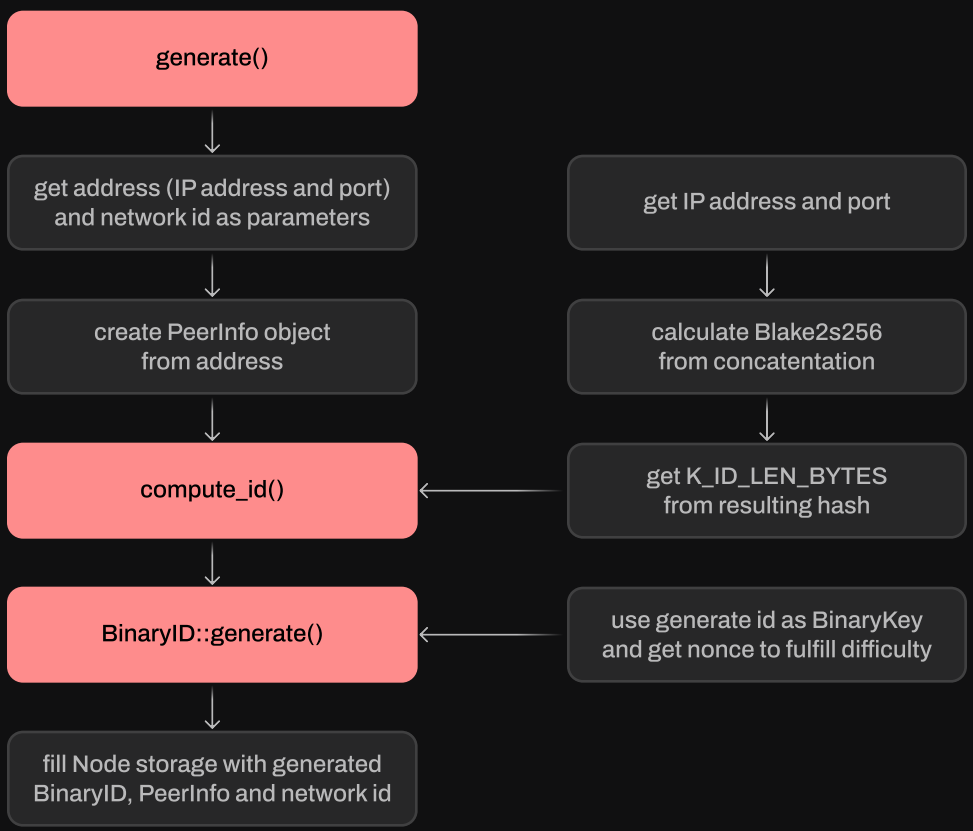
Defines peer - the highest (the most applicable) data structure in the algorithm.
It is the actual representative of the peer (node in the network)
Specifies:

1. generation of message Header from stored info
2. generation of PeerEncodedInfo payload from stored info
3. id (BinaryID) generation from address and port

PeerNode = Node<PeerInfo>

Storage:
1) address: standard SocketAddr object
2) all storage derived from Node<>

Newly created node will contain address (in a form of stored PeerInfo object), network id and binary id generated during creation



MESSAGING PROTOCOL

handling.rs

Defines the high-level data structures for communication between nodes and the messaging protocol itself (based on payload data and messages type primitives)

MessageInfo

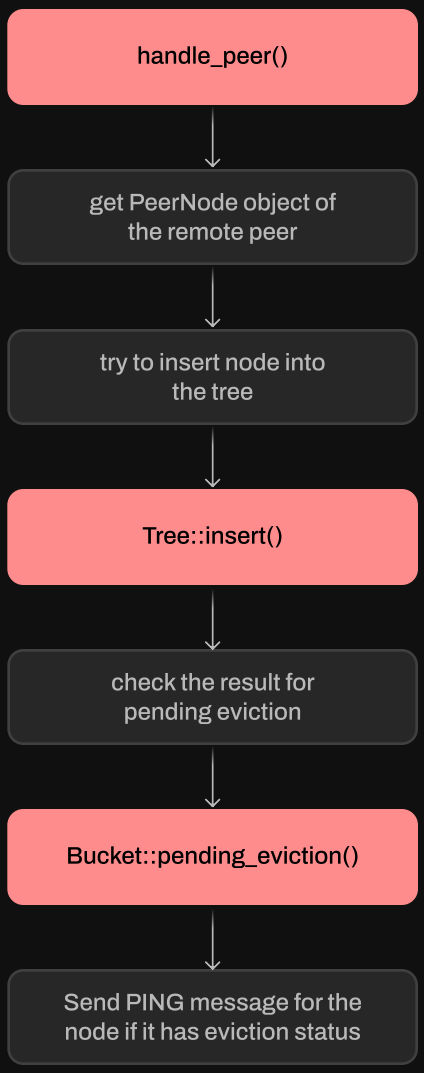
Storage:
1) src: standard SocketAddr object
2) height: Kadcast broadcast height

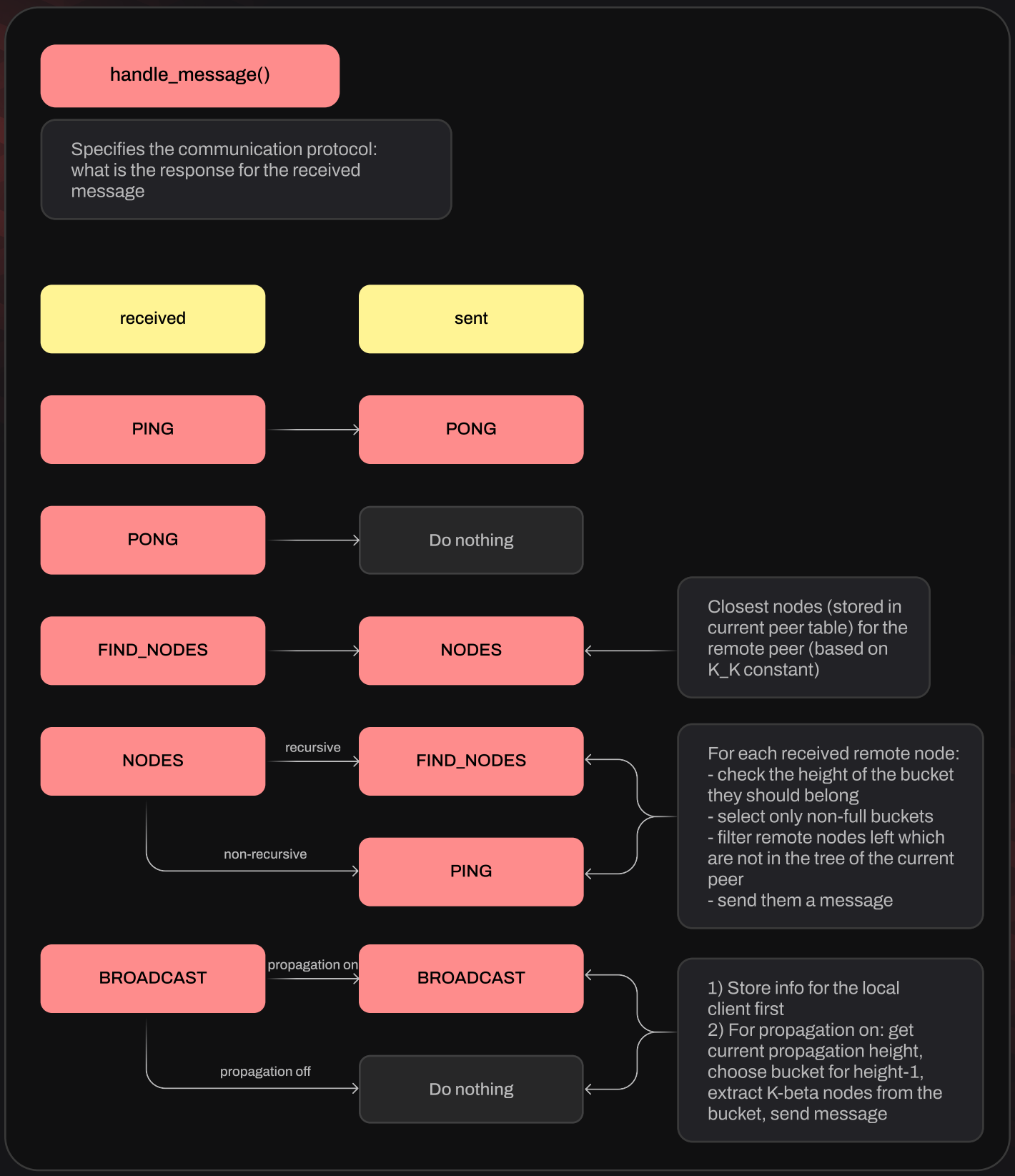
MessageHandler

Storage:
1) my_header: Header object - the representation of header generated from Root node (current peer)
2) ktable: read-write lock object (standard tokio::sync object) around the tree of buckets with nodes (Tree<PeerInfo>)
3) outbound_sender, listener_sender: sender objects (standard tokio::sync::mpsc objects)
4) nodes_reply_fn: callback for received messaged for FindNodes route - either FindNodes (for recursive search) or regular Ping
5) auto_propagate: constant depending on the NetworkConfig

Is created around the NetworkConfig, and the tree of k-buckets.

After being started and instantiated, listens to incoming connections, trues to insert each new peer and handle its message




handle_message()
Specifies the communication protocol: what is the response for the received message
received
sent
PING
PONG
PONG
Do nothing
FIND_NODES
NODES
Closest nodes (stored in current peer table) for the remote peer (based on K_K constant)
NODES
recursive
FIND_NODES
non-recursive
PING
For each received remote node:
- check the height of the bucket they should belong
- select only non-full buckets
- filter remote nodes left which are not in the tree of the current peer
- send them a message
BROADCAST
propagation on
BROADCAST
propagation off
Do nothing
1) Store info for the local client first
2) For propagation on: get current propagation height, choose bucket for height-1, extract K-beta nodes from the bucket, send message

MAINTAINER

maintainer.rs

Defines the peer behavior logic: stores bootstrap nodes for start, defines monitoring logic and handling idle nodes.

TableMaintainer

Storage:
1) bootstrapping_nodes: vector of addresses of boorstrap nodes (defined in config)
2) ktable: instance of Tree with k-buckets
3) outbound_sender: sender object (standard tokio::sync::mpsc object)
4) my_ip: address of current peer (root node)
5) header: Header object constructed from root node

need_bootstrappers(): picks at most 10 closest peers for the current peer (root node of the tree) and checks if there are less than 3 available

contact_bootstrappers(): sends FIND_NODES message to bootstrap nodes **until need_bootstrappers()** conditions are satisfied.

ping_idle_buckets(): picks k-alpha random peers from the buckets with idles nodes and sends FindNodes message to them

monitor_buckets(): perform the maintenance loop:
- checks if more bootstrap nodes are needed
- sleeps for config.bucket.bucket_ttl value
- calls **ping_idle_buckets()**
- removes idle nodes

NETWORK LAYER

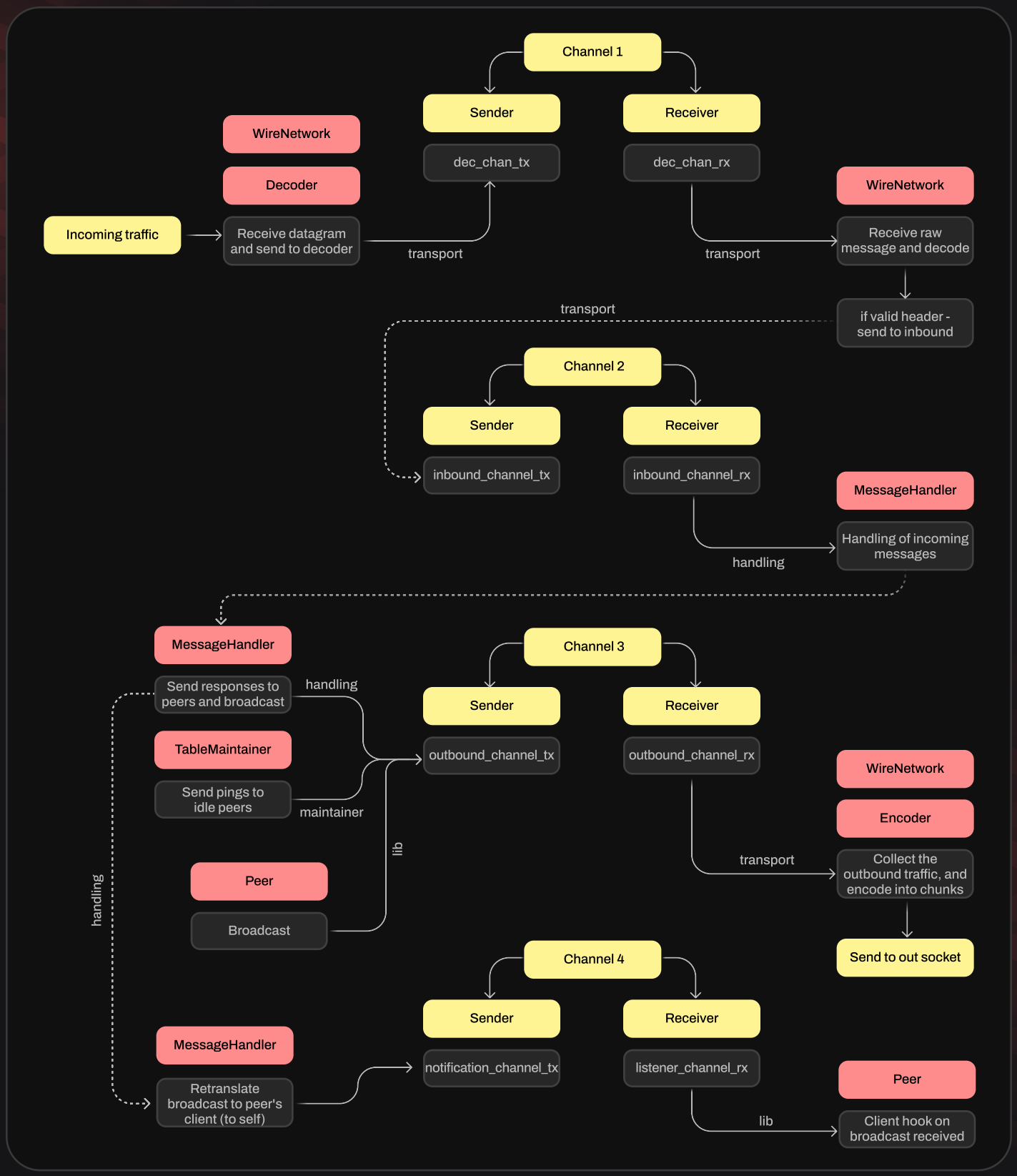
transport.rs

Defines the network (transport) layer of the protocol: encoder/decoder, communication channel, UDP socket and peers' connections handling

WireNetwork

On start:
1) configures the UDP socket
2) configures encoder/decoder
3) spawns sender instance
4) spawns decoder (for unmarshalling incoming messages)
5) spawns receiver instance

INTERNAL COMMUNICATION



TYPES DEPENDENCIES

TRANSPORT LAYER

decoder.rs

Module implements the RaptorQDecoder which is responsible for decoding broadcast messages using the RaptorQ decoding scheme.

RaptorQDecoder

Storage:
1) cache: HashMap storing decoding status for each unique identifier of broadcast messages.
2) last_pruned: Instant representing the time when the cache was last pruned.
3) conf: storing configuration parameters for the decoder that initalized by RaptorQDecoderConf struct.

Configurable spec:
1) TConf type: RaptorQDecoderConf
2) default_configuration(): Using default configuration with a cache prune interval of 5 minutes and a cache time-to-live of 60 seconds.
3) configure(): using provided data to initialize struct

decode():
Decodes a given broadcast message using RaptorQ decoding. The method handles cache management, expiration, and propagation of decoded messages. Checks if the incoming message is a Broadcast, then converts the payload into a ChunkedPayload and retrieves its uid. If the cache entry exists, returns a mutable reference to its status. If the cache entry does not exist, inserts a new entry with CacheStatus::Receiving. If the status is Processed, returns None to avoid repropagation. If the status is Receiving, decodes the encoded chunk using the stored decoder. Compares Kadcast heights to preserve the highest one for propagation. Performs an integrity check and creates a new BroadcastPayload if successful. Updates the cache status to Processed to avoid reprocessing and repropagation. Prunes the cache periodically based on the configured intervals.

RaptorQDecoderConf

Storage:
cache_ttl: Time-to-live for a message in the cache. Messages that have not been processed within this duration will be considered expired and removed from the cache.
cache_prune_every: The interval at which the cache is pruned. Pruning involves removing expired messages from the cache.

TRANSPORT LAYER

encoder.rs

Module implements the RaptorQEncoder, which is responsible for encoding broadcast messages using the RaptorQ encoding scheme. Implements the Configurable and Encoder traits.

RaptorQEncoder

Storage:
1) conf: storing configuration parameters for the encoder that initalized by RaptorQEncoderConf struct.

Configurable spec:
1)TConf type: RaptorQEncoderConf
2) default_configuration(): Using default values to configure struct.
3) configure(conf: &Self::TConf): Using provided data to initialize struct.

decode():
Is responsible for taking a Message::Broadcast and applying RaptorQ encoding, which adds redundancy to transmitted data to recover from potential losses or corruption, and as a results, encode method generates a vector of encoded Messages. If the input message is not a Message::Broadcast, it returns a vector containing the original message.

RaptorQEncoderConf

Storage:
1) min_repair_packets_per_block: The minimum number of repair packets generated per encoding block
2) mtu: Maximum Transmission Unit (MTU) is the maximum size of a protocol data unit (PDU) that can be transmitted in a single network layer transaction.
3) fec_redundancy: Forward Error Correction (FEC) redundancy represents the ratio of redundant (repair) data to the original data. In RaptorQ, this refers to the amount of additional data generated to recover from errors.

TRANSPORT LAYER

socket.rs

Specifies the MultipleOutSocket primitive - element used to managing multiple UDP sockets, providing reliable message sending with configurable retry and backoff strategies for outgoing messages.

MultipleOutSocket

Storage:
1) ipv4:
2) ipv6:
3) udp_backoff_timeout:
4) retry_count:
5) udp_send_retry_interval:

Configurable spec:
1)TConf type: NetworkConfig
2) default_configuration(): NetworkConfig::default()
3) configure(conf: &NetworkConfig): Configures the struct based on the provided NetworkConfig, setting up UDP sockets, backoff timeout, retry count, and retry interval.

send():
Takes a reference to a slice of bytes representing the data to be sent and a reference to the SocketAddr structure indicating the remote address to which the data should be sent.
The method sends data by entering a loop, attempting to send the data up to the specified retry_count times. For each attempt, it chooses the appropriate UDP socket based on whether the remote address is an IPv4 or IPv6 address.

raptorQ.rs

Defines functionality related to encoding and decoding RaptorQ packets within the context of a broadcast payload. It utilizes the blake2 hashing algorithm for generating unique identifiers (UIDs) and the raptorq crate for encoding and decoding RaptorQ packets.

ChunkedPayload

ChunkedPayload<'a>:
Represents a chunk of a broadcast payload for RaptorQ processing.
Wraps a reference to a BroadcastPayload.
Implements TryFrom<&'a BroadcastPayload> to create a ChunkedPayload from a BroadcastPayload.
Checks if the payload length is sufficient.

uid():
Retrieve the UID from the ChunkedPayload. The UID is assumed to be located at the beginning of the gossip_frame.

TRANSPORT LAYER

transmission_info():
Retrieve the ObjectTransmissionInformation from the ChunkedPayload, a slice from the gossip_frame containing the transmission information of size TRANSMISSION_INFO_SIZE and converts it into an array of bytes. Then deserializes this byte array into an ObjectTransmissionInformation struct and returns it.

transmission_info():
Retrieve the encoded chunk from the ChunkedPayload. Extracts this portion from the gossip_frame starting from the index UID_SIZE + TRANSMISSION_INFO_SIZE to the end.

safe_uid():
This method calculates a UID using the Blake2s256 hashing algorithm. It updates the hasher with the UID bytes extracted from the gossip_frame and also includes the transmission information. The rationale for including transmission: it should be sent over a reliable channel because, if corrupted, decoding packets may become impossible, especially if the corrupted info is part of the first received chunk.

BroadcastPayload

bytes():
Marshals the BroadcastPayload into a binary representation. It invokes the marshal_binary method for the BroadcastPayload.

generate_uid():
Calculates a unique identifier (UID) for the BroadcastPayload using the Blake2s256 hashing algorithm. To generate a consistent UID, excludes the first byte of the serialized payload before updating the hasher. The resulting hash is then converted into a fixed-size array of 32 bytes.

TRANSPORT LAYER

plain_encoder.rs

Implements the PlainEncoder struct for encoding and decoding messages using a simple and straightforward method. This module might be used for testing or scenarios where no complex encoding/decoding logic is required.

PlainEncoder

encode()
Encodes a given Message and returns a vector containing the encoded message. In this implementation, the encoding is simplified, and it just wraps the original message into a vector.

Configurable spec:
1) TConf type: ()
2) default_configuration(): Returns an empty tuple, as there is no configuration needed.
3) configure(conf: &Self::TConf): Configures the struct based on the provided empty configuration, creating a new instance of PlainEncoder.

decode()
Decodes a given Message and returns an Option containing the decoded message. In this implementation, the decoding is also simplified, and it always returns the input message wrapped in Some.

encoding.rs

Provides a flexible mechanism for encoding and decoding messages, allowing for different implementations based on features enabled during compilation.

The module structure is organized based on compilation features:
When the raptorq feature is disabled:
It includes the plain_encoder module, providing a simple encoding/decoding mechanism.
Aliases PlainEncoder as TransportEncoder and TransportDecoder for consistency.
When the raptorq feature is enabled:
It includes the raptorq module, which contains more complex and applicable encoding/decoding logic.
Aliases RaptorQEncoder as TransportEncoder and RaptorQDecoder as TransportDecoder for consistency.

Complete Analysis

STANDARD CHECKLIST / VULNERABLE AREAS

Storage structure and data modification flow	Pass
Access control structure, roles existing in the system	Pass
Public interface and restrictions based on the roles system	Pass
General Denial Of Service (DOS)	Pass
Entropy Illusion (Lack of Randomness)	N/A
Order-dependency and time-dependency of operations	Pass
Accuracy loss, incorrect math/formulas other violated operations with numbers	Pass
Validation of function parameters, inputs validation	Pass
Asset management, funds flow and assets conversions	N/A
Signatures replay and multisig schemes security	N/A
Asset Security (backdoors connected to underlying assets)	N/A
Incorrect minting, initial supply or other conditions for assets issuance	N/A
Global settings mis-using, incorrect default values	Pass
Violated communication between components/modules, broken co-dependencies	Pass
3rd party dependencies, used libraries and packages structure	Pass
Single point of failure	Pass
Centralization risk	Pass
General code structure checks and correspondence to best practices	Pass
Language-specific checks	Pass

PROTOCOL RELATED CHECKS

review of all marshaling rules and checked them during simulation testing as well	Pass
review of messages types, messaging protocol (request/response rules)	Pass
review of the traffic flow (input, inbound traffic, encoding/decoding, marshaling rules, outbound traffic)	Pass
binary key review (as the unique node id) - generation rules, verification (including difficulty checks)	Pass
nodes storing: insertion, pending nodes, eviction process	Pass
set of simulation checks	Pass
tree structure (data structure foreach bucket)	Pass
buckets, height of the bucket, positions of nodes in buckets by ids, id (binary key) generation, distance calculation between nodes	Pass
difficulty parameter and nonce validation	Pass
broadcasting rules (nodes picking, working with height parameter, etc) and general message propagation mechanisms	Pass
“find nodes” functionality	Pass
eviction process and keeping old nodes principles	Pass
general correspondence to the Kadcast specification	Pass
messaging protocol and message structure	Pass
general flow of messages delivery and distribution in the network (via simulations)	Pass

DISCOVERED ISSUES

HIGH-1 Verified

NEWLY ADDED NODE IS NOT REFRESHED, AND LACK OF REFRESHING MECHANICS

bucket.rs, insert_pending()
The pending node is checked for being alive but is not refreshed after being added to the array of nodes in the bucket—in spite of being added to the end of the array (as the most recently assessed) and in spite of having the is_alive() check passed.
This issue becomes especially crucial since there is no publicly available interface for refreshing the node in the K-bucket (available for the outer data structures). So, for now, the node cannot be refreshed at any moment except its insertion. Therefore, it leads to scenarios where idle nodes will appear in the middle of the nodes array—despite the necessity to have less recently accessed nodes in front of the array.
The issue is marked as High, as it disrupts the business logic and leads to scenarios of violation of the Kadcast algorithm.

RECOMMENDATION:

Add public interface/wrap around Node.refresh() to have the ability to refresh nodes while calling from outer data structures (as Maintainer or MessageHandler). Add node refreshing after inserting of the pending node.

POST-AUDIT.

Based on the comments from the Dusk team and after the discussion with the security team, the issue was marked as verified. The pending node doesn't require explicit updating after being added to the tree from the pending slot. The node is kept in the pending slot only in the "alive" state - otherwise, it is either removed from the slot or is overridden with the fresh seen_at timestamp due to the mechanism described in the Medium-1 issue (in case the bucket is still full, but the node is still alive). After being added to the tree, the node will be refreshed every time a message is processed.

HIGH-2 Verified

INCONSISTENCY OF THE PROTOCOL WITH THE SPECIFICATION

The current implementation contains inconsistency with the specification defined in https://eprint.iacr.org/2021/996.pdf

1) lib.rs, K_ALPHA
According to the specifications, k-alpha should be a lookup redundancy factor that regulates the number of nodes picked up for the initial bootstrapping procedure. However, the current implementation uses K_ALPHA as a number of random nodes from idle bucket to ping (Tree.idle_buckets()).

2) As for the lookup redundancy factor during the bootstrapping procedure (maintainer.rs, need_bootstrappers()), the protocol uses the hardcoded value 10 (despite the recommendation of 3)

The issue is marked as High, as it disrupts the business logic and leads to scenarios of violating the Kadcast algorithm.

RECOMMENDATION:

Verify the alignment of the current implementation with the specification for the k-alpha parameter.

POST-AUDIT.

After receiving comments from the Dusk team and a discussion with the security team, auditors verified the issue. The Dusk team verified the deviation from the spec for the bootstrapping procedure. Previously, before the introduction of 'need_bootstrappers,' a node would be bootstrapped, and if it didn't receive any response from the initial FIND_NODES (for any reason, be it delays from bootstrapper nodes or network issues), there was no mechanism to continue contacting the bootstrappers. As a result, the routing table remained empty until the next cycle of maintainers. So, the change was introduced via https://github.com/dusk-network/kadcast/issues/112

POST-AUDIT.

Therefore, the Dusk team introduced a retry during bootstrappers to determine the minimum number of nodes. While the 'closest_peers' function was used, despite being used with K_ALPHA during the protocol's lookup phase, - it is used with an arbitrary value here. Thus, the behavior is verified, and arbitrary parameters are covered in the Info-2 issue.

To bring the behavior closer to the spec, the Dusk team will modify the 'need_bootstrappers' function to use 'alive_nodes' instead of 'closest_peers' to avoid the use of constants and allow the library user to choose the threshold https://github.com/dusk-network/kadcast/issues/135

2) The security team verified that the parameter K_ALPHA is correctly utilized within a single bucket to extract K_ALPHA nodes for refreshing the routing table. Thus, the 'need_bootstrappers' function was verified as an addition that deviates from the protocol to support the bootstrapping phase.

MEDIUM-1 Verified

CURRENT PENDING NODE CAN BE OVERRIDDEN AND LOST

bucket.rs
insert(), line 178
The function overrides the pending_node slot, though there are situations when pending_node is set to some value (refer to the protocol overview section). Thus, the previous value will be lost, which may lead to a lost peer.

The issue is marked as Medium, as although it represents a low probability or low-impact risk, it leads to the scenario of disruption of one of the protocol flows (lost peer).

RECOMMENDATION:

Verify that a pending node override despite the already pending peer is the desired behavior, or reject the new node in favor of the currently pending one (if such exists and passes the is_alive() check).

POST-AUDIT.

The Dusk team is aware of the behavior and chose this approach during the development phase. The team explored the approach with the pending nodes enqueuing, though the approach led to inefficient memory usage. Thus, the team opted not to pursue the queue-based approach and left a single-slot-based mechanism.
The team also reviews the capability of having a fixed capacity queue of pending nodes. However, the security team does not recommend such, as it will just extend the number of slots that can be overridden, thus not changing the issue itself. In such circumstances, the flexible approach with override will work better, and the chances of losing a potentially stable connection via the overridden pending node are not worth the increased resource overload. Given the dynamic nature of the protocol, the overridden pending node will return during later lookup procedures if it has the necessary performance.

MEDIUM-2 Verified

DIFFERENT CONSTANTS USED FOR THE SAME PROPERTY

bucket.rs
has_idle(), remove_idle()
Functions use different constants for detecting idle nodes (bucket_ttl -> BUCKET_DEFAULT_TTL_SECS and node_ttl -> BUCKET_DEFAULT_NODE_TTL_MILLIS)
So, functions will have different conditions of idle nodes detection.

The issue is marked as Medium, as although it represents low-probability or low-impact risk, it leads to the scenario of disruption of one of protocol flows (lost peer).

RECOMMENDATION:

Verify the logic and use the necessary constant.

POST-AUDIT.

The Dusk team agreed with the observation. However, the Dusk team noted that BUCKET_DEFAULT_NODE_TTL_MILLIS is specifically used to identify if a node is alive, preventing the unnecessary eviction of a node. It plays a crucial role in all node insertion/refresh operations within the bucket. On the other hand, BUCKET_DEFAULT_TTL_SECS is designed to identify buckets in the idle state (a distinct concept from alive). If there is at least one idle node within a bucket, the system actively proceeds to request a FIND_NODES operation from the nodes within the bucket to update it. Therefore, parameters describe different characteristics (for passive and active maintenance), and the security team verified that those parameters are checked in different modules. Thus, the issue was narrowed down to the indistinctive naming and verified. However, auditors insist on having more distinctive names for the functions (e.g., is_bucket_idle() and remove_outdated_nodes()).

LOW-1 Resolved

KEEP NONCE VALIDATION DURING THE KEY CREATION

key.rs, from_nonce
The function creates a binary key from arbitrary nonce without compliance with the difficulty. Although there is a public function to verify nonce after the key creation, it is against best practices to delegate the verification process out of the creation. For now, only the Header struct uses BinaryID.from_nonce() and calls the validation after that. Nevertheless, such an approach may lead to issues during further development.

The issue is marked as Low, as despite the fact there is no direct security threat, it violates best practices, creates unhandled cases for the business logic, and may cause issues during further development.

RECOMMENDATION:

Add nonce validation against the difficulty into the function

POST-AUDIT:

Fix was presented via the newly created issues
https://github.com/dusk-network/kadcast/issues/136

INFO-1 Resolved

RESERVED FIELD AND VERSIONING NECESSITY

Header.rs
Header.reserved data field
The data field represents 2 bytes reserved for future use. Although “future use” by definition is an undefined concept and it is okay to have reserve for the future, though, auditors have several concerns regarding this field:

- 2 bytes size is quite uncommon as it is neither used for allocated memory alignment nor seems to be enough for some informational fields to be added in the future;
- The header object is the only one having reserved space for future use, so it seems unusual since all other marshaled objects in the system do not have reserved space;
- Since there is a reservation for future use, it is an indirect sign of future upgrades to the protocol. Thus, it will be reasonable to have the protocol version as one of the fields to prevent future conflicts. In the case of simultaneous usage of the protocol version with unused reserved bytes and the upgraded one, there is a possibility of mistreating the headers of messages, leading to conflicts and decreased network sustainability. Although Kadcast is integrated into the Dusk node, a custom version can still be used.

The issue is marked as Info, as it refers to the business logic decision and cannot be classified as a security issue without confirmation from the team. However, auditors see it as a sub-standard behavior and require a double confirmation from the team.

RECOMMENDATION:

Verify the necessity of reserved space in the header, verify the desired reserved space size and consider its increasing, consider adding versioning to the Kadcast protocol - or verify the desired structure of the reserved storage.

POST-AUDIT:

The Dusk team recognized that the current approach, which allocated 2 bytes for the 'reserved' field in the Header object, was not providing the necessary flexibility. The initial intent was to allow users to parse the second byte and adapt accordingly by altering the first byte. However, this approach proved to be insufficient. To address this, the Dusk team has decided to reduce the 'reserved' field to 1 byte. The value of this byte will be used to specify the length of the next N bytes to parse (where N is specified by the first byte). This modification ensures that even during an upgrade, older versions will be able to parse the header, providing compatibility without fully understanding the content of reserved fields. This adjustment is aimed at enhancing the adaptability and sustainability of the network during protocol upgrades.

The adjustment will be made under the issue: https://github.com/dusk-network/kadcast/issues/137 and will be included into the next major Kadcast release, as it was classified as hardfork changes by the Dusk team.

INFO-2 Resolved

MAGIC NUMBERS USED

key.rs, verify_difficuty: 8 - for the number of bits
maintaining.rs, need_bootstrappers: 10 - for number of picked nodes, 3 - for minimum number of peers required
maintaining.rs, contact_bootstrappers: 30 sec - rate for re-pinging nodes.

The issue is marked as Info, as it has no direct security threat, though it violates best practices and decreases the code readability.

RECOMMENDATION:

Consider using constant or additional commets/natspec clarifying the nature of numbers. Also consider moving some constants (like characteristics from maintaining.rs) to the config (e.g. for the minimum number of peers required, cooldown until re-check of bootstrap requirements, etc).

POST-AUDIT:

The Dusk team introduced the changes for the bootstrapping procedure (also see post-audit for High-2) under the issue https://github.com/dusk-network/kadcast/issues/135

INFO-3 Resolved

UNDOCUMENTED EXPECTATION OF CERTAIN INPUT IN PUBLIC

key.rs, verify_difficuty
Based on the function usage in the code, auditors concluded that the function always expects to get the reverse iterator to the byte array. However, this behavior is not documented, and the reverse iterator extraction is not included in the function. The issue is raised as the function is part of the public interface of the BinaryID struct.
The issue is marked as Info, as it has no direct security threat, though it violates best practices and decreases the code readability.

RECOMMENDATION:

Consider including necessary iterator extraction into the function (instead of relying on correct parameters passed) or add clear documentation to the function.

POST-AUDIT:

The adjustment will be made under the issue: https://github.com/dusk-network/kadcast/issues/139 and will be included into the next major Kadcast release, as it was classified as hardfork changes by the Dusk team.

The Dusk team will remove the reversing of the key byte array for the difficulty verification. Thus, the verify_difficulty() will actually check not the trailing zeros of the whole number, but a hybrid complexity scheme (leading zeros + trailing is the first non-zero byte). Despite the change, the function serves its protection purpose (to have some work done actually to disallow unchecked node connection) and will remain internal (the Dusk team verified that it is not supposed to be a part of the public interface in the future). Though auditors still recommend to add natspec/readme with the explicit algorithm description.

INFO-4 Resolved

OVERLAPPING RESPONSIBILITY OF PARAMETERS FOR DIFFICULTY

config.rs, K_DIFF_MIN_BIT, K_DIFF_PRODUCED_BIT
Used in: key.rs
Two constants represent the difficulty of achieving during the nonce generation for the BinaryID. Although two parameters represent the minimum difficulty to check against and the difficulty to use during key generation, they have overlapping meanings. For now, the minimum number is checked for each peer, and both constants have the same default value. Also, as for the peers in the network, there is no profit from using the higher difficulty since the minimum one is checked.

The issue is marked as Info, as it refers to the business logic decision and cannot be classified as a security issue without confirmation from the team. However, auditors see it as a sub-standard behavior and require double confirmation from the team.

RECOMMENDATION:

Verify the necessity of having 2 constants and consider removing the K_DIFF_PRODUCED_BIT

POST-AUDIT:

The Dusk team provided an extended explanation of the necessity of 2 parameters. The distinction between the 'produced' and 'minimum acceptable' verification difficulty facilitates a delayed library upgrade. A timeframe is established during which not all peers in the cluster have updated to the new minimum verification.

Consider a scenario where there's a need to increase the difficulty from 8 to 10. Without this mechanism, it would be impossible to accept messages from peers that haven't immediately updated, potentially leading to a partitioned network. The proposed approach involves having an initial period (T0) where both MIN and PRODUCED are set to 8. Subsequently, there is a defined timeframe from T1 to T2 during which MIN remains at 8, but PRODUCED is set to 10. This allows for a smooth transition, ensuring compatibility with peers who might not have upgraded immediately. Finally, at T2, both MIN and PRODUCED are set to 10, signaling the completion of the upgrade. This way, all peers have a designated period (T1 to T2) to perform the necessary upgrades.

Therefore, both parameters are necessary for the potential difficulty changes, though auditors should notice that the current implementation of the protocol expects outer control for the timeframe and the transition from one difficulty to another by nodes themselves.

INFO-5 Acknowledged

DUAL RESPONSIBILITY OF THE FUNCTION

bucket.rs
insert(), line 153
In this context, the refresh() function is used for both existence check and refreshing. While it works well for the context, it decreases code readability and goes against best practices, which may influence further development.

The issue is marked as Info, as it has no direct security threat, though it violates best practices and decreases the code readability.

RECOMMENDATION:

Consider using the existing Bucket.has_node() function to add more context and clearing the conditioning under the logic branch.

INFO-6 Resolved

OVERRIDE OF THE EVICTION CHECKS

bucket.rs, remove_idle_nodes(), remove_id()
Functions delete idle nodes with full override of the eviction process (eviction requests, node_evict_after check). Although this functionality is dedicated to the buckets clearing by the maintainer service, it is unusual to override the mechanism for the standard flow.

The issue is marked as Info, as it refers to the business logic decision and cannot be classified as a security issue without confirmation from the team. However, auditors see it as a sub-standard behavior and require a double confirmation from the team.

RECOMMENDATION:

Verify that nodes removing with an override of the eviction process logic are a desired behavior or consider the eviction mechanism integration.

POST-AUDIT:

Fix was introduced within the PR #141 for issue 138. Now before removal, a PING is sent due to pending_eviction mechanism

INFO-7 Acknowledged

MISSING INFORMATION ABOUT INSERTION

bucket.rs
insert_pending() may end up inserting a new node into the array. The insert() function may end in inserting 2 new nodes into the array - the pending one and the new one, as there are no restrictions against it. However, only the insert() function can report the Ok(Inserted()) result, and only about the single node. Therefore, the information about pending node insertion will be lost. The same flaw (absent info about insertion) is connected to the try_perform_eviction() (due to the internal call to insert_pending()).

The issue is marked as Info, as it refers to the business logic decision and cannot be classified as a security issue without confirmation from the team. However, auditors see it as a sub-standard behavior and require a double confirmation from the team.

RECOMMENDATION:

Consider refactoring the results of insert/insert_pending functions to allow several nodes reporting, or at least consider adding informational logging of the insertion.

INFO-8 Resolved

FUNCTION BEHAVES DIFFERENTLY FROM ITS NAME

maintainer.rs, ping_idle_buckets()
The function name, natspec, and documentation state that the functions should send PING messages to idle nodes. However, the function sends a FIND_NODES message.

The issue is marked as Info, as it refers to the business logic decision and cannot be classified as a security issue without confirmation from the team. However, auditors see it as a sub-standard behavior and require a double confirmation from the team.

RECOMMENDATION:

Verify that FIND_NODES is the correct message to send and correct the documentation/natspec/function naming. Or adjust the code to send a PING message.

POST-AUDIT:

The function was renamed to find_new_nodes(). Also, the fix introduced ping_and_remove_idles() function, which now sends Ping before removal of idle nodes. Fix was introduced within the PR #141 for issue 138,

INFO-9 Resolved

IDLE NODES REMOVAL CAN MISS THE TRUE-NEGATIVE CASE

kbucket.rs, idle_buckets()
maintained.rs, ping_idle_buckets()

By implementation, the maintainer periodically checks buckets to be idle. And it picks several nodes from idle buckets to re-ping them. However, the function uses the Tree.idle_buckets() function, which calls Bucket.pick() to get K-alpha random nodes from the bucket. Such an approach may lead to 2 scenarios:

- Tree.idle_buckets() (while having a random pick) may return only active nodes, so truly idle nodes will not be pinged. So they will be removed at the next step, though they could be alive, just not refreshed yet. Such an approach contradicts the “keep old nodes” principle, creating a situation where a random pick is preferred.

Also, the random choice of nodes to ping is not aligned with the chosen algorithm, where idle nodes are located at the beginning of the nodes array in the bucket, and those are the first candidates for rechecks and updates.

The issue is marked as Info, as it refers to the business logic decision and cannot be classified as a security issue without confirmation from the team. However, auditors see it as a sub-standard behavior and require a double confirmation from the team.

RECOMMENDATION:

Review the ping of idle nodes mechanics, and consider changing the mechanism from picking a random node to picking the less recently accessed (the first one in the idle bucket array).

POST-AUDIT:

Ehe update for the Info-6 and Info-8 covers the scenario, as removed nodes now get the PING message first. Fix was introduced within the PR #141 for issue 138. Now before removal, a PING is sent due to pending_eviction mechanism

Disclaimer

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.