VoidPhone Project: Final Report

Group 14
July 2, 2024

Muhammed Orhun Gale
Kerem Kurnaz

Group Name: 14
Number of Team Members: 2
Module: DHT
Implementation: Kademlia

Logical Structure of the DHT-14

Logical Structure of the DHT-14

DHT-14 is composed of several key components, each with defined responsibilities, that collectively form the system's distributed network and data storage. After the midterm report, we refactored the logical structure quite extensively. Below is an updated explanation of the logical structure and its components:

1. Node (Main Process)
The Node class represents the fundamental unit in the network responsible for DHT operations and communication.

1.1. Attributes:

- ID: Unique identifier of the node (generated via Proof of Work).
- IP: Node's IP address.
- Port: Communication port for P2P interactions.
- Ping: Boolean flag to check the node’s status.
- DHT: Reference to the DHT class for routing and storage operations.
- Storage: Reference to the node's storage module for managing local data.
- Network: Reference to the Network component for sending and receiving messages.
- Config: Stores node-specific configurations.
- IsDown: Boolean flag indicating whether the node is down.

1.2. Methods:

- Put(key, value, ttl): Stores data in the node's storage.
- Get(key): Retrieves data from the node’s storage.
- FindNode(targetID): Searches for a node using Kademlia's algorithm.
- FindValue(targetKeyID): Searches for a value across nodes.
- AddPeer(nodeID, ip, port): Adds a peer to the node's routing table.
- GetAllPeers(): Retrieves all peers connected to the node.
- Shutdown(): Gracefully shuts down the node.

2. BootstrapNode (Special Node)
The BootstrapNode class extends the Node class. It acts as a known entry point for other nodes joining the network.

2.1. Additional Attributes:

- KnownPeers: Stores a list of known peer nodes.

2.2. Additional Methods:

- AddKnownPeer(nodeID, ip, port): Adds a peer to the known peer list.
- RemoveKnownPeer(ip, port): Removes a peer from the known peer list.
- GetKnownPeers(): Retrieves the known peer list.

3. DHT (Distributed Hash Table)
The DHT class implements the core logic of Kademlia's distributed hash table.

3.1. Attributes:

- RoutingTable: Manages the node's connections to neighboring nodes.
- Storage: Manages the key-value storage within the DHT.
- Network: Handles the DHT network interactions.

3.2. Methods:

- PUT(key, value, ttl): Stores the key-value pair on the DHT.
- GET(key): Retrieves the value associated with a key from the DHT.
- FindNode(targetID): Finds the closest nodes to a given target ID.
- FindValue(targetKeyID): Looks for a value associated with the target key across the DHT.
- StoreToStorage(key, value, ttl): Stores data in the local storage.
- IterativeFindNode(targetID): Performs an iterative search for a node.
- SendStoreMessage(key, value, targetNode): Sends a message to store data in a target node.

4. Routing Table
The RoutingTable class manages the Kademlia buckets and the node's neighbors.

4.1. Attributes:

- Buckets: Contains K-Buckets which store references to neighboring nodes.
- NodeID: ID of the node that owns this routing table.

4.2. Methods:

- AddNode(targetID): Adds a new node to the appropriate K-Bucket.
- RemoveNode(targetID): Removes a node from the routing table.
- GetClosestNodes(targetID): Returns the closest nodes to the given target ID.
- XORDistance(id1, id2): Calculates the XOR distance between two node IDs.

5. Network
The Network class manages network-level communication between nodes.

5.1. Attributes:

- IP: IP address of the node.
- ID: Unique ID of the node.
- Port: Port used for communication.

5.2. Methods:

- SendMessage(targetIP, targetPort, data): Sends a message to a target node over TLS.
- StartListening(): Starts the TLS server to listen for incoming connections.
- HandleConnection(conn): Processes incoming network connections and handles the received message.

6. Message
The Message class handles the format and content of messages exchanged between nodes.

6.1. Attributes:

- Type: The message type, such as PUT, GET, PING, etc.
- Size: Size of the message header.

6.2. Methods:

- Serialize(): Serializes the message for network transmission.
- Deserialize(): Deserializes received data into a message.
- CreateMessage(): Factory method to create a new message based on its type.

7. Storage
The Storage class is responsible for the encrypted local storage of key-value pairs within a node.

7.1. Attributes:

- data: A map that stores the key-value data.
- cleanup_interval: The interval for cleaning up expired data.
- key: Encryption key used for securing stored data.

7.2. Methods:

- Put(key, value, ttl): Stores data in the storage with a specified TTL.
- Get(key): Retrieves stored data based on the key.
- CleanupExpired(): Cleans up expired entries from the storage.

- StartCleanup(interval): Starts a background routine to periodically clean up expired data.

8. API
The API class is responsible for providing an interface for clients to interact with the node.

8.1. Attributes:

- tlsAddress: The TLS-enabled address for secure communication.
- nonTLSAddress: The non-secure address (optional).

8.2. Methods:

- StartServer(tlsAddress, nonTLSAddress, nodeInstance): Starts the API server with optional TLS.
- HandleConnection(conn, nodeInstance): Processes incoming client connections and dispatches DHT operations (e.g., PUT, GET).

9. Security
The Security class provides utility functions for securing network communications and data encryption.

9.1. Methods:

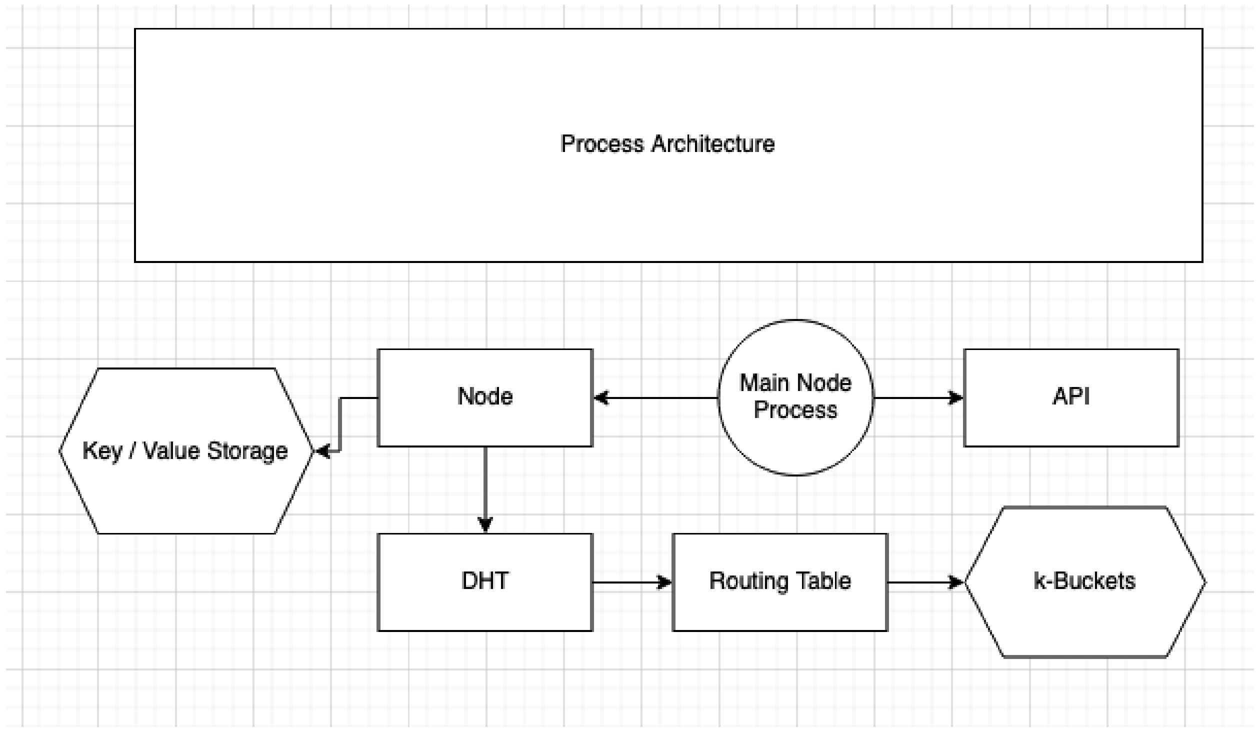
- DialTLS(id, address): Establishes a secure TLS connection to a remote node.
- StartTLSListener(id, address): Starts a TLS listener on the specified address for secure communications.

10. Util
The Util module provides various helper functions for encryption, configuration, logging, and general utilities.

10.1. Methods:

- Encrypt(data, key): Encrypts data using a specified key.
- Decrypt(data, key): Decrypts data using a specified key.
- ParseAddress(address): Parses a network address into its components (IP, port).
- Log(): Returns the logger instance for logging operations.

Process Architecture of the DHT Project



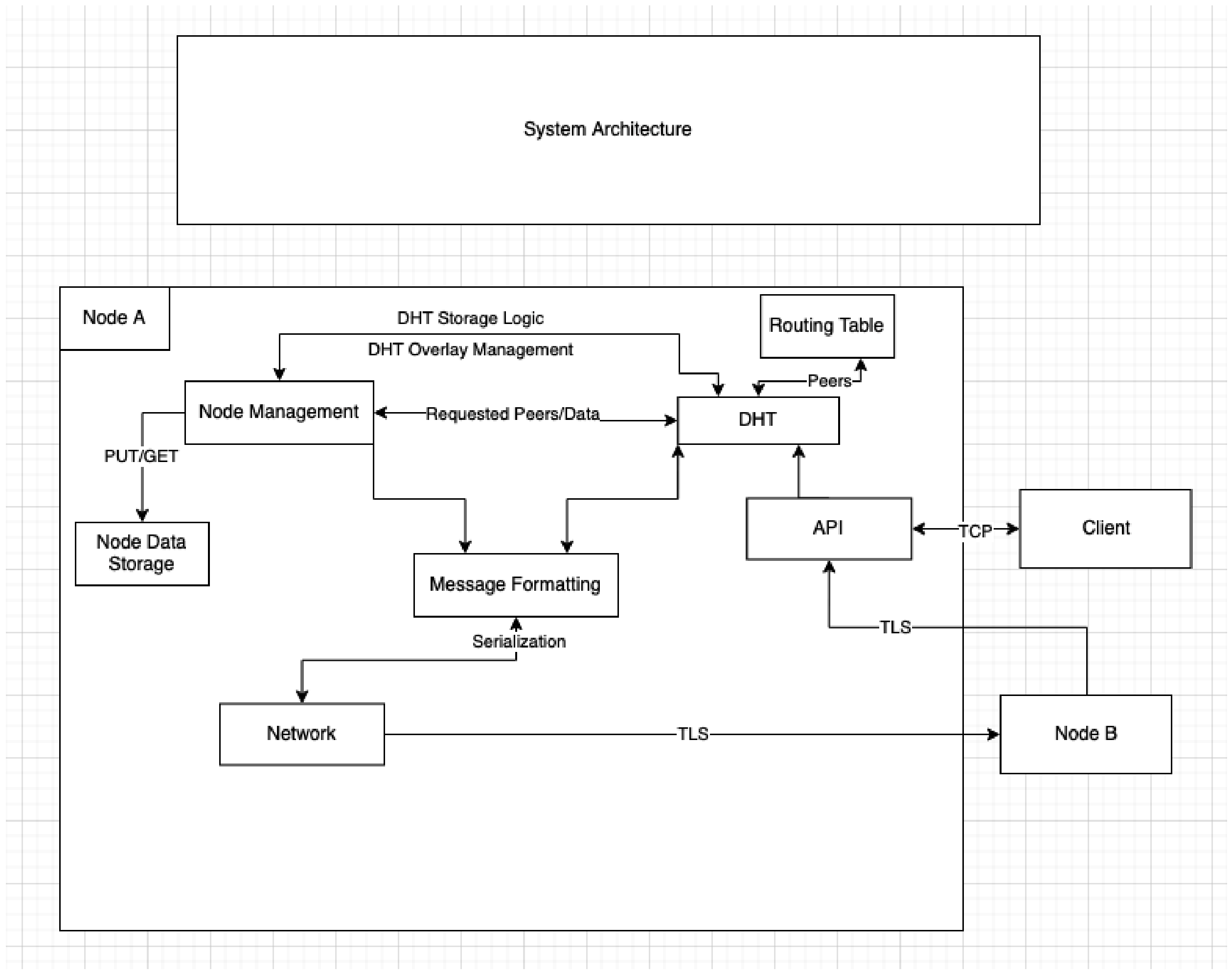
The DHT system operates as a collection of a process and threads, each responsible for handling specific tasks. At the heart of the process architecture is the Node process. Each node represents an independent entity in the network, and it runs as a separate process. This design encapsulates the node's activities and allows for parallel operation with other nodes. The Node process is responsible for managing its own key-value data storage, operations for peer management, and initalizing other components.

Components and Threads

1. **Key/Value Storage (Data Structure)**
    - This component is responsible for managing the actual data that the node stores. It implements a key-value storage system where data can be put and retrieved.
    - This structure is directly accessed by the Storage class for data operations.
2. **Kademlia k-Buckets (Data Structure)**
    - K-Buckets are a part of the Kademlia protocol implemented in the DHT system. They are used to manage information about neighboring nodes in the network.
    - Each node maintains several k-buckets, which store identifying information (IDs) about other nodes, facilitating efficient routing and lookup operations in the DHT.

3. **DHT**
    - The DHT thread is dedicated to handling DHT-specific tasks such as peer discovery, data lookup, and node management. It ensures that the node can efficiently find and interact with other nodes in the distributed network.
    - This thread manages the RoutingTable with use of k-Buckets and coordinates with other nodes to maintain the DHT network.
4. **API**
    - The API thread is tasked with handling incoming network communications, ensuring that the node can communicate with other nodes, receive data, and maintain network stability.

System Architecture of the DHT Project



The system architecture diagram for the DHT project shows the internal components of a single node (Node A) its interactions with other nodes (such as Node B) and the client and how the internal components interact with each other. Each node in the network performs specific roles including data storage, peer management, message handling, and network communication.

Key Components and Their Roles

1. **Node Data Storage:** Handles PUT and GET operations, allowing data to be stored and retrieved using a key-value storage mechanism. Communicates directly with the DHT component to handle operation requests by the API
2. **Node Management**: Manages peer relationships, initializes the DHT and the Storage.

3. **Routing Table**: Keeps track of known nodes in the network and their corresponding keys. This is used by the DHT to route queries efficiently.
4. **Message Handling:** Manages the formatting and processing of messages for network communication. Serializes messages for transmission and deserializes received messages, ensuring they are in the correct format for processing by the node. Works with the DHT and API components to handle incoming and outgoing messages.
5. **Network:** Facilitates communication to other nodes in the network. Ensures secure communication using TLS. Directly interfaces with message handling to send formatted messages.
6. **API:** Acts as a handler for messages that arrive to the node. Processes requests from both external nodes and client, and routes them to the appropriate handlers within the node. Handles connections using TLS.
7. **DHT:** Manages the DHT logic and operations. Handles tasks related to peer discovery, data lookup, and overall maintenance of the DHT overlay network. Interfaces with the Handler to process requests for peers and data. It also coordinates with the Module Interface to handle tasks from other nodes and the K-Buckets to find the closest nodes to a target ID.

Security Measures

1. TLS
TLS is implemented for secure communication between nodes, ensuring authenticity and preventing eavesdropping or man-in-the-middle attacks.

2. Proof of Work Against Sybil Attacks
The node ID generation process uses Proof of Work to defend against Sybil attacks, limiting the creation of fake identities.

3. Rate Limiting (optional)
Rate limiting is applied to prevent denial-of-service (DoS) attacks, restricting the number of requests a node can make. This feature can be turned off if needed.

4. Encrypted Storage
All stored keys and values are encrypted, ensuring that data remains protected within the node.

1. PUT Message
Purpose:
To store or update a value associated with a given key in the DHT.

Message Format:

- size: Total size of the message.
- type: PUT message type identifier.
- TTL: Time-to-live value for the key-value pair.
- Replication: Replication factor indicating how many nodes should replicate the value.
- Key: The 256-bit (32-byte) key associated with the value.
- Value: The data or value being stored.

Reasoning:
The PUT message allows a node to insert or update key-value pairs in the distributed hash table. It enables the storage and propagation of data across multiple nodes.

2. GET Message
Purpose:
To request the value associated with a given key from the DHT.

Message Format:

- size: Total size of the message.
- type: GET message type identifier.
- Key: The 256-bit (32-byte) key for which the value is being requested.

Reasoning:
The GET message is used by a node to retrieve the value corresponding to a specific key from other nodes in the DHT.

3. PING Message
Purpose:
To check if a node is alive and responsive.

Message Format:

- size: Total size of the message.
- type: PING message type identifier.

Reasoning:
PING messages are sent to verify whether a node is operational and reachable, thus maintaining network health and connectivity.

4. PONG Message
Purpose:
To respond to a PING message, confirming that a node is alive.

Message Format:

- size: Total size of the message.
- type: PONG message type identifier.
- Timestamp: The timestamp of the PONG response.

Reasoning:
PONG messages are replies to PING requests, confirming that a node is responsive. This helps in keeping track of active nodes in the network.

5. SUCCESS Message
Purpose:
To indicate the successful completion of a key/value operation (such as GET or PUT).

Message Format:

- size: Total size of the message.
- type: SUCCESS message type identifier.
- Key: The 256-bit (32-byte) key involved in the operation.
- Value: The value associated with the key (if applicable).

Reasoning:
This message confirms that an operation, like storing or retrieving data, was completed successfully. It ensures the requester knows the operation was successful.

6. FAILURE Message
Purpose:
To indicate that a key/value operation (such as GET or PUT) has failed.

Message Format:

- size: Total size of the message.
- type: FAILURE message type identifier.
- Key: The 256-bit (32-byte) key involved in the operation.

Reasoning:
This message is sent when a request, such as retrieving or storing a value, could not be fulfilled. It helps the requester handle errors.

7. FIND_NODE Message
Purpose:
To search for a node with a specific ID in the DHT.

Message Format:

- size: Total size of the message.
- type: FIND_NODE message type identifier.
- Key: The 256-bit (32-byte) ID of the target node.

Reasoning:
The FIND_NODE message is used to discover the location of a node in the DHT based on its unique identifier. This is essential for node discovery and routing.

8. FIND_VALUE Message
Purpose:
To search for a value associated with a specific key across the DHT.

Message Format:

- size: Total size of the message.
- type: FIND_VALUE message type identifier.
- Key: The 256-bit (32-byte) key associated with the value being searched for.

Reasoning:
This message is used to locate the node storing a particular value, enabling data retrieval across the network.

9. STORE Message
Purpose:
To propagate a PUT request across the network and replicate the data in multiple nodes.

Message Format:

- size: Total size of the message.
- type: STORE message type identifier.

- TTL: Time-to-live for the stored value.
- Replication: Number of nodes the data should be replicated to.
- Key: The 256-bit (32-byte) key of the data.
- Value: The data or value being stored.

Reasoning:
The STORE message is used to distribute a PUT request across multiple nodes to ensure redundancy and fault tolerance in the network.

10. BOOTSTRAP Message
Purpose:
To allow a node to join the network by contacting a bootstrap node.

Message Format:

- size: Total size of the message.
- type: BOOTSTRAP message type identifier.
- Address: The network address of the node sending the message.

Reasoning:
The BOOTSTRAP message is used to contact a known bootstrap node to join the DHT network, enabling the node to discover peers and start participating in the DHT.

Future Work

- Kademlia replication logic must be implemented.
- Implement the bootstrapping with DHT node lookup logic.
- Liveness check must be implemented within the DHT module
- Refreshing for outdated buckets must be implemented.

Workload Distribution

Orhun:

1) Designing and implementing the process and system architecture.
2) Implementing the node, message, API modules.
3) Implemented the bootstrap node and bootstrapping process.
4) Implementing the security module and its parts like TLS and PoW.
5) Implementing the utils module to provide the configuration parsing, logging, and encryption.
6) Implementing the CI/CD pipeline, build system, and testing structure.

7) Implementing the Dockerfile and structuring the system to work on distributedly.
8) Deployment and client testing automatization.
9) Documentation on README.md
10) Produced unit and integration tests for the implemented parts.

Kerem:

1) Designing and implementing the DHT logic by using the Kademlia documentation.
2) Implementing K-Buckets, Routing Table, and replication logic.
3) Implemented the bootstrapping process handling and node registration to buckets.
4) Implementing the API handlers and P2P protocol message processing.
5) Producing figures and writing the final report.
6) Produced unit and integration tests for the implemented parts.

Effort Spent for the Project

Both Teammates: We collaboratively designed the logical structure, process architecture, and system architecture. Everyone produced the unit tests for the modules that they implemented.

For Orhun: Orhun focused on designing and implementing the system architecture, including the Node, Message, API, and Security modules, along with their related components such as TLS and PoW. He also developed the utils module for configuration parsing, logging, and encryption. Additionally, he established the CI/CD pipeline, build system, and testing structure, including the Docker setup for distributed deployment. He automated client testing, wrote unit and integration tests, and documentation.

For Kerem: Kerem focused on designing and implementing the DHT logic based on the Kademlia protocol, including the implementation of K-Buckets, the Routing Table, and replication mechanisms. He also developed the API handlers and handled the processing of P2P protocol messages, while writing unit and integration tests for the components he implemented.