Kademlia DHT Developer Documentation

Table of Contents

Overview

The Kademlia DHT (Distributed Hash Table) implementation provides a complete peer-to-peer distributed storage and routing system based on the Kademlia algorithm. This implementation is designed for the py-libp2p ecosystem and supports both client and server modes.

Key Features

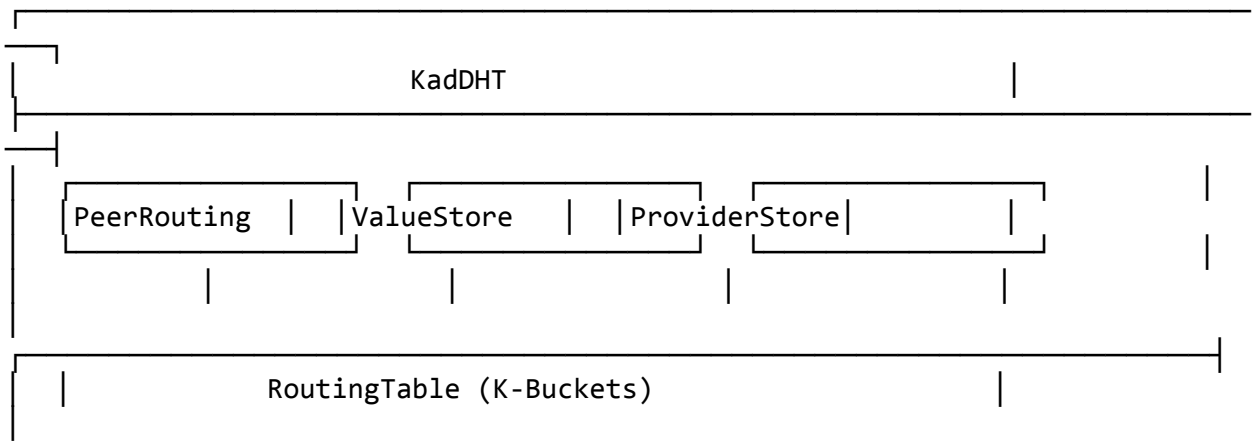
- **Distributed Storage**: Store and retrieve key-value pairs across the network
- **Peer Discovery**: Efficiently find peers in the network using XOR distance metric
- **Content Routing**: Locate content providers for specific keys
- **Fault Tolerance**: Handle peer failures and network partitions
- **Scalability**: Support for large networks with logarithmic lookup complexity
- **Flexible Modes**: Client and server operation modes

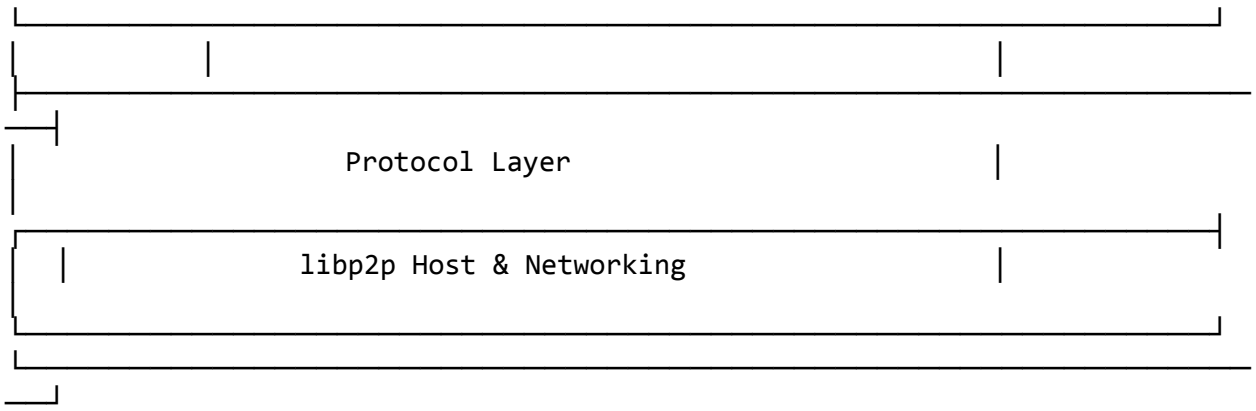
Core Concepts

- **XOR Distance Metric**: Uses XOR operation to calculate distance between keys/peers
- **K-Buckets**: Routing table structure that stores peers by distance ranges
- **Alpha Parameter**: Controls concurrency in network operations (default: 3)
- **Iterative Lookups**: Multi-hop queries to find closest peers or values

Architecture

The Kademlia DHT follows a layered architecture:





Core Components

1. KadDHT (kad_dht.py)

The main orchestrator class that coordinates all DHT operations and extends `Service` for lifecycle management.

Key Responsibilities:

- Service lifecycle management with `run()` method
- Protocol message handling via `handle_stream()`
- Coordination between components
- Mode switching (Client/Server)

Initialization Example:

```
class KadDHT(Service):
    def __init__(self, host: IHost, mode: str):
        super().__init__()
        self.host = host
        self.local_peer_id = host.get_id()
        self.mode = mode.upper()

        # Initialize core components
        self.routing_table = RoutingTable(self.local_peer_id, self.host)
        self.peer_routing = PeerRouting(host, self.routing_table)
        self.value_store = ValueStore(host=host,
local_peer_id=self.local_peer_id)
        self.provider_store = ProviderStore(host=host,
peer_routing=self.peer_routing)

        # Set protocol handler
        host.set_stream_handler(PROTOCOL_ID, self.handle_stream)
```

Important Methods with Code Examples

run() - Main Service Loop with Periodic Maintenance

```python
async def run(self) -> None:
    """Run the DHT service."""
    logger.info(f"Starting Kademlia DHT with peer ID {self.local_peer_id}")

    # Main service loop
    while self.manager.is_running:
        # Periodically refresh the routing table
        await self.refresh_routing_table()

        # Check if it's time to republish provider records
        current_time = time.time()
        self._last_provider_republish = current_time

        # Clean up expired values and provider records
        expired_values = self.value_store.cleanup_expired()
        if expired_values > 0:
            logger.debug(f"Cleaned up {expired_values} expired values")

        self.provider_store.cleanup_expired()

        # Wait before next maintenance cycle
        await trio.sleep(ROUTING_TABLE_REFRESH_INTERVAL)
```

What it does: This is the heart of the DHT service that runs continuously. It performs periodic maintenance tasks including refreshing the routing table to discover new peers and remove stale ones, cleaning up expired values from storage, and managing provider record lifecycles. The loop runs every 60 seconds (ROUTING_TABLE_REFRESH_INTERVAL) to maintain network health.

handle_stream() - Process Incoming Protocol Messages

```python
async def handle_stream(self, stream: INetStream) -> None:
    """Handle an incoming DHT stream using varint length prefixes."""
    if self.mode == "CLIENT":
        stream.close()
        return

    peer_id = stream.muxed_conn.peer_id
    logger.debug(f"Received DHT stream from peer {peer_id}")
    await self.add_peer(peer_id)

    try:
        # Read varint-prefixed length for the message
        length_prefix = b""
        while True:
            byte = await stream.read(1)
            if not byte:
```

```
                logger.warning("Stream closed while reading varint length")
                await stream.close()
                return
            length_prefix += byte
            if byte[0] & 0x80 == 0:
                break
        msg_length = varint.decode_bytes(length_prefix)

        # Read the message bytes
        msg_bytes = await stream.read(msg_length)
        if len(msg_bytes) < msg_length:
            logger.warning("Failed to read full message from stream")
            await stream.close()
            return

        # Parse as protobuf
        message = Message()
        message.ParseFromString(msg_bytes)
        logger.debug(f"Received DHT message from {peer_id}, type:
{message.type}")

        # Handle different message types (FIND_NODE, PUT_VALUE, GET_VALUE,
etc.)
        # ...existing message handling code...

    except Exception as e:
        logger.error(f"Error handling DHT stream: {e}")
    finally:
        await stream.close()
```

What it does: This method handles all incoming DHT protocol messages from other peers. It first checks if the node is in CLIENT mode (which rejects incoming streams), then reads varint-prefixed protobuf messages from the stream. It processes different message types like FIND_NODE (peer discovery), PUT_VALUE/GET_VALUE (data storage/retrieval), and ADD_PROVIDER/GET_PROVIDERS (content routing). The method ensures proper stream cleanup and error handling.

put_value() - Store Values in the DHT Network

```
async def put_value(self, key: bytes, value: bytes) -> None:
    """Store a value in the DHT."""
    logger.debug(f"Storing value for key {key.hex()}")

    # 1. Store locally first
    self.value_store.put(key, value)
    logger.debug(f"Stored value locally for key {key.hex()}")

    # 2. Get closest peers, excluding self
    closest_peers = [
        peer for peer in self.routing_table.find_local_closest_peers(key)
```

```
        if peer != self.local_peer_id
    ]
    logger.debug(f"Found {len(closest_peers)} peers to store value at")

    # 3. Store at remote peers in batches of ALPHA, in parallel
    stored_count = 0
    for i in range(0, len(closest_peers), ALPHA):
        batch = closest_peers[i : i + ALPHA]
        batch_results = [False] * len(batch)

        async def store_one(idx: int, peer: ID) -> None:
            try:
                with trio.move_on_after(QUERY_TIMEOUT):
                    success = await self.value_store._store_at_peer(peer,
key, value)
                    batch_results[idx] = success
                    if success:
                        logger.debug(f"Stored value at peer {peer}")
            except Exception as e:
                logger.debug(f"Error storing value at peer {peer}: {e}")

        async with trio.open_nursery() as nursery:
            for idx, peer in enumerate(batch):
                nursery.start_soon(store_one, idx, peer)

        stored_count += sum(batch_results)

    logger.info(f"Successfully stored value at {stored_count} peers")
```

What it does: This method implements the DHT storage protocol by first storing the value locally, then replicating it to the closest peers in the network. It uses batched parallel operations (controlled by ALPHA=3) to avoid overwhelming the network. Each batch of peers is contacted concurrently using trio nurseries, with timeouts to handle unresponsive peers. This ensures data redundancy and availability across the network.

get_value() - Retrieve Values from the DHT Network

```
async def get_value(self, key: bytes) -> bytes | None:
    logger.debug(f"Getting value for key: {key.hex()}")

    # 1. Check local store first
    value = self.value_store.get(key)
    if value:
        logger.debug("Found value locally")
        return value

    # 2. Get closest peers, excluding self
    closest_peers = [
        peer for peer in self.routing_table.find_local_closest_peers(key)
        if peer != self.local_peer_id
```

```
    ]
    logger.debug(f"Searching {len(closest_peers)} peers for value")

    # 3. Query ALPHA peers at a time in parallel
    for i in range(0, len(closest_peers), ALPHA):
        batch = closest_peers[i : i + ALPHA]
        found_value = None

        async def query_one(peer: ID) -> None:
            nonlocal found_value
            try:
                with trio.move_on_after(QUERY_TIMEOUT):
                    value = await self.value_store._get_from_peer(peer, key)
                    if value is not None and found_value is None:
                        found_value = value
                        logger.debug(f"Found value at peer {peer}")
            except Exception as e:
                logger.debug(f"Error querying peer {peer}: {e}")

        async with trio.open_nursery() as nursery:
            for peer in batch:
                nursery.start_soon(query_one, peer)

        if found_value is not None:
            self.value_store.put(key, found_value)  # Cache locally
            logger.info("Successfully retrieved value from network")
            return found_value

    # 4. Not found
    logger.warning(f"Value not found for key {key.hex()}")
    return None
```

What it does: This method retrieves values from the DHT by first checking the local store for performance, then querying closest peers in batches. It uses parallel queries with early termination - as soon as any peer returns the value, it stops querying others and caches the result locally. This optimizes for both speed and network efficiency.

provide() / find_providers() - Content Routing Operations

```
async def provide(self, key: bytes) -> bool:
    """Reference to provider_store.provide for convenience."""
    return await self.provider_store.provide(key)

async def find_providers(self, key: bytes, count: int = 20) ->
list[PeerInfo]:
    """Reference to provider_store.find_providers for convenience."""
    return await self.provider_store.find_providers(key, count)
```

What it does: These methods delegate to the ProviderStore component to handle content routing. `provide()` advertises that this node can provide specific content by sending

ADD_PROVIDER messages to the closest peers to the content key. `find_providers()` locates nodes that have advertised they can provide specific content, essential for content discovery in distributed systems.

switch_mode() - Runtime Mode Switching

```python
async def switch_mode(self, new_mode: str) -> str:
    mode = new_mode.upper()
    if mode not in ["CLIENT", "SERVER"]:
        raise ValueError(f"Invalid mode '{mode}'. Must be 'CLIENT' or
'SERVER'.")
    if mode == "CLIENT":
        self.routing_table.cleanup_routing_table()
    self.mode = mode
    logger.info(f"Switched to {mode} mode")
    return self.mode
```

What it does: This method allows runtime switching between CLIENT and SERVER modes. CLIENT mode only makes outgoing requests and rejects incoming streams, while SERVER mode handles both. When switching to CLIENT mode, it cleans up the routing table since client nodes don't need to maintain peer information for serving requests. This flexibility allows nodes to change their participation level in the network dynamically.

2. RoutingTable (routing_table.py)

Implements the Kademlia routing table using k-buckets with XOR distance-based organization.

Key Classes:

- `KBucket`: Stores peers within a specific distance range
- `RoutingTable`: Manages multiple k-buckets and peer lookup

Key Features:

- Peer management with LRU eviction
- Distance-based bucket organization
- Peer liveness checking via ping
- Automatic bucket splitting when full

Distance Calculation Example:

```python
def distance(key1: bytes, key2: bytes) -> int:
    """Calculate XOR distance between two keys."""
    distance = 0
    for b1, b2 in zip(key1, key2):
        distance = (distance << 8) | (b1 ^ b2)
    return distance
```

Important Methods:

- `add_peer()`: Add peer to appropriate bucket
- `find_local_closest_peers()`: Get closest known peers
- `refresh_routing_table()`: Maintain routing table health

3. PeerRouting (peer_routing.py) ----------

Handles peer discovery and network-wide peer lookups using iterative algorithms.

Key Features:

- Iterative peer lookup with convergence detection
- Concurrent queries limited by ALPHA parameter
- Peer information caching in peerstore
- Network topology discovery

Iterative Lookup Implementation:

```
async def find_closest_peers_network(self, target_key: bytes, count: int =
20) -> list[ID]:
    closest_peers = self.routing_table.find_local_closest_peers(target_key,
count)
    queried_peers: set[ID] = set()
    rounds = 0

    while rounds < MAX_PEER_LOOKUP_ROUNDS:
        rounds += 1
        peers_to_query = [p for p in closest_peers if p not in queried_peers]
        if not peers_to_query:
            break

        # Query ALPHA peers in parallel
        peers_batch = peers_to_query[:ALPHA]
        new_peers: list[ID] = []

        async with trio.open_nursery() as nursery:
            for peer in peers_batch:
                nursery.start_soon(
                    self._query_single_peer_for_closest, peer, target_key,
new_peers
                )

        # Update closest peers and check for convergence
        all_candidates = closest_peers + new_peers
        old_closest_peers = closest_peers[:]
        closest_peers = sort_peer_ids_by_distance(target_key,
all_candidates)[:count]

        if closest_peers == old_closest_peers:
            break  # No improvement
```

```
    return closest_peers
```

Important Methods:

- `find_peer()`: Locate specific peer
- `find_closest_peers_network()`: Network-wide closest peer search
- `refresh_routing_table()`: Maintain routing table freshness

4. ValueStore (value_store.py)

Manages key-value storage with TTL-based expiration and peer-to-peer operations.

Key Features:

- TTL-based expiration with automatic cleanup
- Local and remote storage operations
- Varint-prefixed protobuf communication
- Peer-to-peer value propagation

Storage Format:

```
# Store format: {key: (value, validity_timestamp)}
self.store: dict[bytes, tuple[bytes, float]] = {}

def put(self, key: bytes, value: bytes, validity: float = 0.0) -> None:
    if validity == 0.0:
        validity = time.time() + DEFAULT_TTL
    self.store[key] = (value, validity)
```

Remote Storage Example:

```
async def _store_at_peer(self, peer_id: ID, key: bytes, value: bytes) ->
bool:
    stream = await self.host.new_stream(peer_id, [PROTOCOL_ID])

    # Create PUT_VALUE message
    message = Message()
    message.type = Message.MessageType.PUT_VALUE
    message.key = key
    message.record.key = key
    message.record.value = value
    message.record.timeReceived = str(time.time())

    # Send with varint length prefix
    proto_bytes = message.SerializeToString()
    await stream.write(varint.encode(len(proto_bytes)))
    await stream.write(proto_bytes)

    # Read response...
```

Important Methods:

- `put()` / `get()`: Local storage operations
- `_store_at_peer()` / `_get_from_peer()`: Remote operations
- `cleanup_expired()`: Maintenance operations

5. ProviderStore (provider_store.py)

Handles content provider advertisements and lookups with automatic republishing.

Key Features:

- Provider record management with expiration
- Content advertisement to closest peers
- Provider discovery with fallback to closest peers
- Automatic republishing of provider records

Provider Advertisement Example:

```
async def provide(self, key: bytes) -> bool:
    # Store locally
    self.add_provider(key, PeerInfo(self.local_peer_id,
self.host.get_addrs()))

    # Find closest peers to advertise to
    closest_peers = await self.peer_routing.find_closest_peers_network(key)

    # Send ADD_PROVIDER to closest peers
    success_count = 0
    for peer_id in closest_peers[:ALPHA]:
        if await self._advertise_to_peer(peer_id, key):
            success_count += 1

    return success_count > 0
```

Important Methods:

- `provide()`: Advertise content availability
- `find_providers()`: Locate content providers
- `add_provider()` / `get_providers()`: Provider record management

6. Utility Functions (utils.py)

Common utility functions for DHT operations including distance calculations and key generation.

Key Functions:

- `distance()`: XOR distance calculation

- `sort_peer_ids_by_distance()`: Distance-based peer sorting
- `create_key_from_binary()`: Generate DHT keys from binary data
- `bytes_to_base58()`: Encoding utilities

Protocol Implementation

Message Types

The DHT uses protobuf messages defined in `kademlia.proto`:

```
enum MessageType {
    PUT_VALUE = 0;      // Store key-value pair
    GET_VALUE = 1;      // Retrieve value by key
    ADD_PROVIDER = 2;   // Advertise content provider
    GET_PROVIDERS = 3;  // Find content providers
    FIND_NODE = 4;      // Find closest peers
    PING = 5;           // Peer liveness check
}
```

Wire Protocol

- **Transport**: Uses libp2p streams over TCP
- **Encoding**: Protocol Buffers with varint length prefixes
- **Protocol ID**: `/ipfs/kad/1.0.0`

Varint Length Prefixing

All messages use varint encoding for length prefixes:

```
# Sending a message
proto_bytes = message.SerializeToString()
await stream.write(varint.encode(len(proto_bytes)))
await stream.write(proto_bytes)

# Reading a message
length_bytes = b""
while True:
    byte = await stream.read(1)
    if not byte:
        break
    length_bytes += byte
    if byte[0] & 0x80 == 0:
        break
msg_length = varint.decode_bytes(length_bytes)
msg_bytes = await stream.read(msg_length)
```

Message Handling

The `handle_stream()` method in KadDHT processes all incoming protocol messages:

FIND_NODE Message Handling

```
if message.type == Message.MessageType.FIND_NODE:
    target_key = message.key
    closest_peers = self.routing_table.find_local_closest_peers(target_key,
20)

    response = Message()
    response.type = Message.MessageType.FIND_NODE

    for peer in closest_peers:
        if peer == peer_id:  # Skip requester
            continue
        peer_proto = response.closerPeers.add()
        peer_proto.id = peer.to_bytes()
        peer_proto.connection = Message.ConnectionType.CAN_CONNECT

        # Add addresses
        try:
            addrs = self.host.get_peerstore().addrs(peer)
            for addr in addrs:
                peer_proto.addrs.append(addr.to_bytes())
        except Exception:
            pass

    # Send response
    response_bytes = response.SerializeToString()
    await stream.write(varint.encode(len(response_bytes)))
    await stream.write(response_bytes)
```

PUT_VALUE Message Handling

```
elif message.type == Message.MessageType.PUT_VALUE and
message.HasField("record"):
    key = message.record.key
    value = message.record.value

    if key and value:
        self.value_store.put(key, value)

        # Send acknowledgement
        response = Message()
        response.type = Message.MessageType.PUT_VALUE
        response.key = key

        response_bytes = response.SerializeToString()
```

```
        await stream.write(varint.encode(len(response_bytes)))
        await stream.write(response_bytes)
```

Mode-Based Behavior

Client mode nodes reject incoming streams:

```
async def handle_stream(self, stream: INetStream) -> None:
    if self.mode == "CLIENT":
        stream.close()
        return
    # ... handle server mode logic
```

Data Flow

1. Value Storage Flow ----------

```
async def put_value(self, key: bytes, value: bytes) -> None:
    # 1. Store locally first
    self.value_store.put(key, value)

    # 2. Get closest peers, excluding self
    closest_peers = [
        peer for peer in self.routing_table.find_local_closest_peers(key)
        if peer != self.local_peer_id
    ]

    # 3. Store at remote peers in batches of ALPHA
    stored_count = 0
    for i in range(0, len(closest_peers), ALPHA):
        batch = closest_peers[i : i + ALPHA]

        async with trio.open_nursery() as nursery:
            for peer in batch:
                nursery.start_soon(self._store_at_peer_wrapper, peer, key,
value)

        stored_count += successful_stores
```

2. Value Retrieval Flow ----------

```
async def get_value(self, key: bytes) -> bytes | None:
    # 1. Check local store first
    value = self.value_store.get(key)
    if value:
        return value

    # 2. Query closest peers in batches
    closest_peers = [
```

```
        peer for peer in self.routing_table.find_local_closest_peers(key)
        if peer != self.local_peer_id
    ]

    for i in range(0, len(closest_peers), ALPHA):
        batch = closest_peers[i : i + ALPHA]
        found_value = None

        async with trio.open_nursery() as nursery:
            for peer in batch:
                nursery.start_soon(self._query_peer_for_value, peer, key)

        if found_value:
            self.value_store.put(key, found_value)  # Cache locally
            return found_value

    return None
```

3. Service Lifecycle ----------

```
async def run(self) -> None:
    logger.info(f"Starting Kademlia DHT with peer ID {self.local_peer_id}")

    while self.manager.is_running:
        # Periodic maintenance tasks
        await self.refresh_routing_table()

        # Cleanup expired values
        expired_values = self.value_store.cleanup_expired()
        if expired_values > 0:
            logger.debug(f"Cleaned up {expired_values} expired values")

        # Cleanup expired provider records
        self.provider_store.cleanup_expired()

        # Wait before next cycle
        await trio.sleep(ROUTING_TABLE_REFRESH_INTERVAL)  # 60 seconds
```

API Reference

KadDHT Class

Constructor

```
def __init__(self, host: IHost, mode: str):
    """
    Initialize Kademlia DHT.

    Args:
```

```
        host: libp2p host instance
        mode: "CLIENT" or "SERVER"

    Raises:
        ValueError: If mode is not "CLIENT" or "SERVER"
    """
```

Core Methods

Value Operations

```
async def put_value(self, key: bytes, value: bytes) -> None:
    """
    Store a value in the DHT.

    Stores the value locally and replicates to closest peers.
    Uses batched parallel operations with ALPHA concurrency.
    """

async def get_value(self, key: bytes) -> bytes | None:
    """
    Retrieve a value from the DHT.

    Checks local store first, then queries closest peers.
    Caches successful retrievals locally.
    """
```

Content Routing

```
async def provide(self, key: bytes) -> bool:
    """
    Advertise that this node provides content for the key.

    Sends ADD_PROVIDER messages to closest peers.
    Returns True if advertised to at least one peer.
    """

async def find_providers(self, key: bytes, count: int = 20) ->
list[PeerInfo]:
    """
    Find providers for the given content key.

    Searches local provider store first, then queries network.
    Falls back to closest peers if no providers found.
    """
```

Peer Management

```
async def add_peer(self, peer_id: ID) -> bool:
    """
    Add a peer to the routing table.
```

```
    Returns True if peer was added or updated.
    """

async def find_peer(self, peer_id: ID) -> PeerInfo | None:
    """
    Find a specific peer in the network.

    Checks routing table and peerstore first,
    then performs network lookup if needed.
    """
```

Utility Methods

```
def get_routing_table_size(self) -> int:
    """Get number of peers in routing table."""

def get_value_store_size(self) -> int:
    """Get number of stored values."""

async def switch_mode(self, new_mode: str) -> str:
    """
    Switch between CLIENT and SERVER modes.

    CLIENT mode: Only makes outgoing requests
    SERVER mode: Handles incoming requests
    """
```

Configuration Constants

```
# Protocol configuration
PROTOCOL_ID = "/ipfs/kad/1.0.0"
BUCKET_SIZE = 20  # k parameter - peers per bucket
ALPHA = 3  # Concurrency parameter for parallel operations

# Timing configuration
DEFAULT_TTL = 24 * 60 * 60  # 24 hours - value expiration
ROUTING_TABLE_REFRESH_INTERVAL = 60  # 1 minute - maintenance cycle
QUERY_TIMEOUT = 10  # seconds - individual query timeout

# Limits
MAX_PEER_LOOKUP_ROUNDS = 20  # Maximum lookup iterations
MAXIMUM_BUCKETS = 256  # Maximum routing table buckets
```

Usage Examples

Basic Setup with Service Management

```
import trio
from libp2p import new_host
from libp2p.kad_dht import KadDHT
```

```python
async def main():
    # Create libp2p host
    host = new_host()

    # Initialize DHT in server mode
    dht = KadDHT(host, "SERVER")

    # Start services using context manager
    async with host.run():
        async with dht.run():
            # DHT is now running and handling requests
            print(f"DHT started with peer ID: {dht.local_peer_id}")

            # Your application logic here
            await trio.sleep_forever()

trio.run(main)
```

Complete Value Storage Example

```python
async def complete_storage_example():
    host = new_host()
    dht = KadDHT(host, "SERVER")

    async with host.run(), dht.run():
        # Store a value
        key = b"my-important-key"
        value = b"my-important-value"

        print(f"Storing value for key: {key.hex()}")
        await dht.put_value(key, value)
        print(f"Stored at {dht.get_routing_table_size()} peers")

        # Retrieve the value
        print(f"Retrieving value for key: {key.hex()}")
        retrieved_value = await dht.get_value(key)

        if retrieved_value == value:
            print("✓ Value retrieved successfully!")
        else:
            print("✗ Value retrieval failed")
```

Content Provider Example

```python
async def content_provider_example():
    host = new_host()
    dht = KadDHT(host, "SERVER")

    async with host.run(), dht.run():
        # Content hash (could be from IPFS, etc.)
        content_key = hashlib.sha256(b"my-content").digest()
```

```
        # Advertise that we provide this content
        print(f"Advertising content: {content_key.hex()}")
        success = await dht.provide(content_key)

        if success:
            print("✓ Successfully advertised as provider")
        else:
            print("✗ Failed to advertise")

        # Later, find providers for this content
        providers = await dht.find_providers(content_key)
        print(f"Found {len(providers)} providers:")
        for provider in providers:
            print(f"  - {provider.peer_id} at {provider.addrs}")
```

Network Bootstrap Example

```
async def bootstrap_example():
    # Create multiple DHT nodes
    nodes = []
    for i in range(3):
        host = new_host()
        dht = KadDHT(host, "SERVER")
        nodes.append((host, dht))

    # Start all nodes
    async with trio.open_nursery() as nursery:
        for host, dht in nodes:
            nursery.start_soon(host.run)
            nursery.start_soon(dht.run)

        # Connect nodes to each other
        for i, (host1, dht1) in enumerate(nodes):
            for j, (host2, dht2) in enumerate(nodes):
                if i != j:
                    await dht1.add_peer(host2.get_id())

        # Now the network is bootstrapped
        print("DHT network bootstrapped!")
        await trio.sleep_forever()
```

Mode Switching Example

```
async def mode_switching_example():
    host = new_host()
    dht = KadDHT(host, "CLIENT")  # Start in client mode

    async with host.run(), dht.run():
        print(f"Started in {dht.mode} mode")
```

```
        # Client mode - can make requests but won't handle incoming ones
        value = await dht.get_value(b"some-key")

        # Switch to server mode
        new_mode = await dht.switch_mode("SERVER")
        print(f"Switched to {new_mode} mode")

        # Now can handle incoming requests
        await dht.put_value(b"new-key", b"new-value")
```

Error Handling

Network Error Handling

```
async def robust_dht_operations():
    try:
        value = await dht.get_value(key)
        if value is None:
            print("Value not found in DHT")
    except trio.TooSlowError:
        print("Operation timed out")
    except ConnectionError as e:
        print(f"Network connectivity issue: {e}")
    except Exception as e:
        print(f"Unexpected error: {e}")
```

Stream Handling Errors

The DHT handles stream errors gracefully:

```
async def handle_stream(self, stream: INetStream) -> None:
    try:
        # Read and process message
        # ...
    except Exception as e:
        logger.error(f"Error handling DHT stream: {e}")
    finally:
        await stream.close()  # Always close streams
```

Validation and Logging

```
# Enable comprehensive logging
import logging
logging.basicConfig(level=logging.DEBUG)

# DHT-specific loggers
logging.getLogger("kademlia-example.kad_dht").setLevel(logging.INFO)
logging.getLogger("kademlia-example.peer_routing").setLevel(logging.DEBUG)
logging.getLogger("kademlia-example.value_store").setLevel(logging.DEBUG)
```

Performance Considerations

Concurrency Control

The implementation uses the ALPHA parameter to control parallelism:

```
# Batch operations by ALPHA to prevent overwhelming the network
for i in range(0, len(peers), ALPHA):
    batch = peers[i : i + ALPHA]
    async with trio.open_nursery() as nursery:
        for peer in batch:
            nursery.start_soon(operation_func, peer)
```

Memory Management

```
# Automatic cleanup prevents memory leaks
async def run(self) -> None:
    while self.manager.is_running:
        # Clean expired values
        expired_values = self.value_store.cleanup_expired()

        # Clean expired provider records
        self.provider_store.cleanup_expired()

        await trio.sleep(ROUTING_TABLE_REFRESH_INTERVAL)
```

Network Optimization

- **Progressive Lookups**: Stop early when sufficient results found
- **Local Caching**: Check local stores before network queries
- **Connection Reuse**: Stream management through libp2p host

Testing and Debugging

Unit Testing Pattern

```
import pytest
import trio
from libp2p import new_host
from libp2p.kad_dht import KadDHT

@pytest.mark.trio
async def test_value_operations():
    # Setup
    host = new_host()
    dht = KadDHT(host, "SERVER")

    # Test within service context
    async with host.run(), dht.run():
        # Test storage
```

```
        key, value = b"test-key", b"test-value"
        await dht.put_value(key, value)

        # Test retrieval
        result = await dht.get_value(key)
        assert result == value

        # Test local storage
        assert dht.get_value_store_size() == 1
```

Debugging Tools

```
# Inspect DHT state
print(f"Routing table size: {dht.get_routing_table_size()}")
print(f"Value store size: {dht.get_value_store_size()}")
print(f"Local peer ID: {dht.local_peer_id}")

# Check specific peer connectivity
peer_info = await dht.find_peer(target_peer_id)
if peer_info:
    print(f"Peer {target_peer_id} is reachable at {peer_info.addrs}")
```

Common Issues and Solutions

1. Empty Routing Table

Problem: DHT operations fail due to no known peers

Solution:

```
# Ensure proper bootstrap
bootstrap_peers = [ID.from_base58("QmBootstrap...")]
for peer_id in bootstrap_peers:
    await dht.add_peer(peer_id)

# Check routing table after bootstrap
if dht.get_routing_table_size() == 0:
    logger.warning("No peers in routing table after bootstrap")
```

2. Values Not Persisting

Problem: Stored values disappear quickly

Solution:

```
# Check TTL configuration
print(f"Default TTL: {DEFAULT_TTL} seconds")

# Manual expiration check
```

```
keys = dht.value_store.get_keys()
print(f"Currently stored keys: {len(keys)}")
```

3. Slow Network Operations ------------------------

Problem: DHT operations are slow

Solution:

```
# Increase parallelism
ALPHA = 5  # More concurrent operations

# Reduce timeout for faster failure detection
QUERY_TIMEOUT = 5  # Faster timeout
```

Conclusion

This Kademlia DHT implementation provides a robust, production-ready foundation for distributed applications. The modular architecture, comprehensive error handling, and flexible configuration options make it suitable for a wide range of peer-to-peer applications.

The implementation leverages modern Python async/await patterns with trio for high-performance concurrent operations, while maintaining compatibility with the broader libp2p ecosystem.

Key strengths:

- **Modular Design**: Easy to extend and customize
- **Production Ready**: Comprehensive error handling and logging
- **Performance Focused**: Efficient algorithms and concurrency control
- **Standards Compliant**: Compatible with other Kademlia implementations

For additional support, contributions, or questions, refer to the project's documentation and issue tracker.