План завершения системы DHT (Distributed Hash Table)

Автор: Manus AI
Дата: 5 июня 2025 года
Версия: 1.0

Исполнительное резюме

Проект DHT (Distributed Hash Table) представляет собой амбициозную реализацию алгоритма Kademlia на языке Java с применением принципов Clean Architecture и Domain Driven Design. Анализ показал, что проект имеет солидную архитектурную основу с 94 Java файлами, 2706 строками кода и хорошо структурированным разделением на слои. Однако система находится в незавершенном состоянии с 75 TODO комментариями и множеством заглушек в критических компонентах.

Данный документ представляет комплексный план завершения системы, включающий анализ текущего состояния, выявление пробелов в реализации, и детальную дорожную карту для создания полнофункциональной распределенной хеш-таблицы. План структурирован по приоритетам и включает конкретные технические решения, временные оценки и рекомендации по тестированию.

Анализ текущего состояния проекта

Архитектурные достижения

Проект демонстрирует высокий уровень архитектурного мышления, что подтверждается применением современных подходов к проектированию программного обеспечения. Разделение на Domain Layer и Application Layer соответствует принципам Clean Architecture, обеспечивая четкое разграничение бизнес-логики и прикладных сервисов.

Domain Layer содержит 59 файлов и включает все ключевые компоненты DHT системы: управление узлами (node), маршрутизацию (router), хранение данных (storage), протокол взаимодействия (protocol), систему сообщений (messages), управление идентификаторами (id), конфигурацию (config) и обработку

исключений (exception). Такая структуризация создает прочную основу для реализации алгоритма Kademlia.

Application Layer с 24 файлами предоставляет интерфейсы для взаимодействия с внешним миром, включая клиентские и серверные компоненты, менеджеры различных аспектов системы и адаптеры для интеграции. Наличие отдельного слоя для Use Cases демонстрирует понимание принципов гексагональной архитектуры.

Реализация алгоритма Kademlia

Анализ кода показывает, что автор имеет глубокое понимание алгоритма Kademlia. Реализованы основные типы узлов: KademliaRegularNode, KademliaRoutingNode и KademliaStorageNode, что соответствует принципу разделения ответственности в распределенных системах.

Класс KademliaId содержит 66 строк кода и реализует основы XOR метрики - ключевой особенности алгоритма Kademlia. Интерфейс UnionId определяет метрику расстояния, что является правильным абстрагированием для различных типов идентификаторов в системе.

Компонент KademliaRegularNode содержит заготовки основных операций DHT: lookupContentStorages, getContent, putContent и checkResponsibility. Присутствие метода checkResponsibility указывает на понимание концепции ответственности узлов за определенные диапазоны ключей в пространстве идентификаторов.

Выявленные пробелы и незавершенные компоненты

Детальный анализ кода выявил множество критических пробелов, препятствующих функционированию системы. В классе KademliaRegularNode большинство методов содержат заглушки или возвращают null, что делает невозможным выполнение базовых операций DHT.

Метод lookupContentStorages содержит TODO комментарии, указывающие на неопределенность в логике проверки ответственности и поиска ближайших узлов. Метод findClosestNode возвращает null, что критично для функционирования алгоритма поиска в Kademlia.

Система сообщений, несмотря на наличие базовых классов BaseMessage и BaseMessageWithResource, не содержит реализации конкретных типов сообщений Kademlia: PING, STORE, FIND_NODE и FIND_VALUE. Без этих сообщений невозможно обеспечить взаимодействие между узлами сети.

Компоненты маршрутизации и хранения данных также находятся в зачаточном состоянии. Отсутствует реализация k-buckets - основной структуры данных для хранения информации о соседних узлах в Kademlia.

Техническая дорожная карта завершения

Фаза 1: Завершение базовых компонентов (Приоритет: Критический)

Первая фаза должна сосредоточиться на реализации фундаментальных компонентов, без которых система не может функционировать. Эта фаза является критически важной и должна быть выполнена в первую очередь.

1.1 Реализация XOR метрики и системы идентификаторов

Необходимо завершить реализацию класса KademliaId, обеспечив корректное вычисление XOR расстояния между идентификаторами. Метод computeDistance должен принимать другой KademliaId и возвращать XOR их значений как неотрицательное целое число или массив байтов для больших идентификаторов.

Требуется реализовать методы сравнения идентификаторов, включая equals, hashCode и compareTo для обеспечения корректной работы с коллекциями Java. Метод asBytes должен возвращать байтовое представление идентификатора для сетевой передачи.

Класс должен поддерживать генерацию случайных идентификаторов и создание идентификаторов из хеш-функций SHA-1 или SHA-256. Это критично для обеспечения равномерного распределения узлов в пространстве идентификаторов.

1.2 Создание системы сообщений Kademlia

Необходимо создать конкретные классы сообщений для всех четырех основных операций Kademlia. Класс PingMessage должен содержать идентификатор отправителя и его сетевой адрес. PingResponse должен подтверждать доступность узла.

StoreMessage должен содержать ключ, значение и информацию об отправителе. FindNodeMessage должен содержать целевой идентификатор для поиска. FindNodeResponse должен возвращать список ближайших известных узлов.

FindValueMessage аналогичен FindNodeMessage, но FindValueResponse может содержать либо запрашиваемое значение, либо список ближайших узлов, если значение не найдено.

Все сообщения должны поддерживать сериализацию для сетевой передачи, предпочтительно с использованием Protocol Buffers или JSON для обеспечения межплатформенной совместимости.

1.3 Реализация k-buckets для таблицы маршрутизации

K-bucket является основной структурой данных в Kademlia для хранения информации о соседних узлах. Каждый bucket должен содержать до k узлов (обычно k=20) на определенном XOR расстоянии.

Класс KBucket должен поддерживать операции добавления узла, удаления узла, поиска узла и получения всех узлов в bucket. Узлы в bucket должны быть упорядочены по времени последнего контакта для реализации LRU (Least Recently Used) политики.

Класс RoutingTable должен содержать массив из 160 k-buckets (для 160-битных идентификаторов) и предоставлять методы для поиска ближайших узлов к заданному идентификатору. Метод findClosestNodes должен возвращать k ближайших узлов из соответствующих buckets.

Фаза 2: Реализация основных операций DHT (Приоритет: Высокий)

Вторая фаза сосредоточена на реализации основных операций распределенной хеш-таблицы, которые обеспечивают функциональность системы.

2.1 Завершение реализации KademliaRegularNode

Метод lookupContentStorages должен быть полностью переработан для корректной реализации алгоритма поиска Kademlia. Алгоритм должен начинать с ближайших известных узлов и итеративно приближаться к цели, запрашивая α узлов параллельно на каждом шаге.

Метод checkResponsibility должен определять, отвечает ли текущий узел за хранение данного ключа, основываясь на XOR расстоянии до ключа и знании о других узлах в сети. Узел отвечает за ключ, если он является одним из k ближайших узлов к этому ключу.

Методы getContent и putContent должны реализовывать логику поиска и сохранения данных с использованием алгоритма Kademlia. putContent должен

сохранять данные на k ближайших узлах для обеспечения избыточности и отказоустойчивости.

2.2 Реализация сетевого взаимодействия

Компоненты в пакете protocol должны обеспечивать надежную передачу сообщений между узлами. Необходимо реализовать как UDP, так и TCP транспорт для различных типов сообщений.

UDP подходит для быстрых запросов типа PING и FIND_NODE, где важна скорость, а потеря отдельных пакетов не критична. TCP следует использовать для передачи больших объемов данных в операциях STORE и при передаче значений в FIND_VALUE.

Система должна включать механизмы повторной передачи, тайм-ауты и обработку сетевых ошибок. Каждое сообщение должно иметь уникальный идентификатор для сопоставления запросов и ответов.

2.3 Реализация локального хранилища

Компонент StorageNode должен предоставлять эффективное локальное хранилище для пар ключ-значение. Рекомендуется использовать встроенную базу данных типа H2 или RocksDB для обеспечения персистентности данных.

Хранилище должно поддерживать операции put, get, delete и итерацию по всем ключам. Необходимо реализовать механизм истечения срока действия данных (TTL) для автоматической очистки устаревших записей.

Для повышения производительности следует реализовать кеширование часто запрашиваемых данных в памяти с использованием LRU политики вытеснения.

Фаза 3: Обеспечение отказоустойчивости и производительности (Приоритет: Средний)

Третья фаза направлена на повышение надежности и производительности системы, что критично для работы в реальных условиях.

3.1 Реализация механизмов обнаружения и восстановления после сбоев

Система должна регулярно проверять доступность узлов в таблице маршрутизации с помощью PING сообщений. Недоступные узлы должны удаляться из k-buckets и заменяться новыми узлами.

Необходимо реализовать механизм republishing - периодическое перераспределение данных для компенсации ушедших из сети узлов. Каждый узел должен периодически проверять, что данные, за которые он отвечает, все еще реплицированы на достаточном количестве узлов.

Система должна поддерживать graceful shutdown - корректное завершение работы узла с передачей его данных другим узлам сети.

3.2 Оптимизация производительности

Для повышения производительности поиска следует реализовать параллельные запросы к нескольким узлам одновременно. Параметр α (обычно 3) определяет количество параллельных запросов.

Необходимо реализовать кеширование результатов поиска для избежания повторных запросов к сети. Кеш должен иметь ограниченный размер и TTL для предотвращения использования устаревших данных.

Система должна поддерживать адаптивные тайм-ауты, которые увеличиваются для медленных узлов и уменьшаются для быстрых, что позволяет оптимизировать время отклика.

3.3 Реализация мониторинга и метрик

Необходимо добавить сбор метрик производительности: время отклика на запросы, количество успешных и неуспешных операций, размер таблицы маршрутизации, количество хранимых ключей.

Система должна предоставлять API для мониторинга состояния узла, включая информацию о соседних узлах, статистику сетевого трафика и использование ресурсов.

Рекомендуется интегрировать систему логирования с возможностью настройки уровней детализации для различных компонентов системы.

Фаза 4: Интеграция и тестирование (Приоритет: Высокий)

Четвертая фаза сосредоточена на интеграции всех компонентов и обеспечении качества системы через комплексное тестирование.

4.1 Создание интеграционных тестов

Необходимо создать тесты, которые проверяют взаимодействие между различными компонентами системы. Тесты должны включать сценарии

присоединения узлов к сети, поиска данных, сохранения данных и обработки сбоев.

Следует реализовать тесты производительности, которые измеряют время отклика системы при различных нагрузках и размерах сети. Тесты должны проверять масштабируемость системы от небольших сетей (10-100 узлов) до больших (1000+ узлов).

Необходимо создать тесты отказоустойчивости, которые симулируют различные типы сбоев: отключение узлов, сетевые разделения, потеря пакетов, высокая задержка.

4.2 Создание демонстрационного приложения

Для демонстрации возможностей системы следует создать простое приложение, которое показывает основные операции DHT. Приложение может представлять собой распределенную файловую систему или систему обмена сообщениями.

Демонстрационное приложение должно включать веб-интерфейс для визуализации состояния сети, просмотра таблиц маршрутизации узлов и мониторинга операций в реальном времени.

4.3 Документация и примеры использования

Необходимо создать подробную документацию API с примерами кода для всех основных операций. Документация должна включать руководство по развертыванию, конфигурированию и мониторингу системы.

Следует создать tutorial для разработчиков, который показывает, как интегрировать DHT в существующие приложения и как расширять функциональность системы.

Временные оценки и ресурсы

Распределение времени по фазам

Фаза 1 (Базовые компоненты) оценивается в 4-6 недель работы опытного Java разработчика. Эта фаза является наиболее критичной и требует глубокого понимания алгоритма Kademlia и принципов распределенных систем.

Фаза 2 (Основные операции DHT) потребует 3-4 недели. Основная сложность заключается в корректной реализации алгоритмов поиска и обеспечении надежного сетевого взаимодействия.

Фаза 3 (Отказоустойчивость и производительность) займет 2-3 недели. Эта фаза включает множество оптимизаций и может выполняться параллельно с тестированием.

Фаза 4 (Интеграция и тестирование) потребует 3-4 недели. Значительная часть времени будет потрачена на создание комплексных тестов и отладку выявленных проблем.

Общая оценка проекта составляет 12-17 недель, что соответствует 3-4 месяцам разработки при полной занятости одного разработчика.

Требования к экспертизе

Проект требует разработчика с опытом работы с распределенными системами и глубоким пониманием сетевого программирования. Необходимы знания алгоритмов и структур данных, особенно в области peer-to-peer сетей.

Опыт работы с Java и понимание принципов Clean Architecture будет критически важным для поддержания качества существующего кода. Знание протоколов сериализации и сетевых протоколов также необходимо.

Рекомендуется привлечение консультанта с опытом работы с DHT системами для ревью архитектурных решений и алгоритмов.

Рекомендации по архитектуре и дизайну

Сохранение архитектурных принципов

Существующая архитектура проекта демонстрирует высокое качество дизайна и должна быть сохранена. Принципы Clean Architecture и DDD обеспечивают хорошую тестируемость и расширяемость системы.

Рекомендуется продолжить использование интерфейсов для всех основных компонентов, что обеспечивает возможность создания различных реализаций и упрощает тестирование с использованием mock объектов.

Разделение на слои должно строго соблюдаться: Domain Layer не должен зависеть от Application Layer или внешних библиотек. Все внешние зависимости должны быть абстрагированы через интерфейсы.

Обеспечение расширяемости

Система должна быть спроектирована с учетом возможных будущих расширений. Рекомендуется создать plugin архитектуру для добавления новых типов сообщений и операций без изменения основного кода.

Следует предусмотреть возможность использования различных алгоритмов хеширования и размеров идентификаторов. Это позволит адаптировать систему для различных применений и требований безопасности.

Система должна поддерживать различные стратегии репликации данных и политики кеширования, которые могут быть настроены в зависимости от требований конкретного применения.

Обеспечение безопасности

Хотя безопасность не является приоритетом для базовой реализации, следует заложить основы для будущих улучшений. Все сообщения должны содержать цифровые подписи для предотвращения подделки.

Рекомендуется реализовать базовую защиту от Sybil атак через ограничение количества узлов с одного IP адреса и требование proof-of-work для присоединения к сети.

Система должна поддерживать шифрование данных при передаче и хранении, хотя это может быть реализовано в более поздних версиях.

Заключение

Проект DHT представляет собой амбициозную и технически сложную реализацию алгоритма Kademlia с применением современных принципов архитектуры программного обеспечения. Существующая кодовая база демонстрирует глубокое понимание предметной области и создает прочную основу для завершения системы.

Предложенный план завершения структурирован по приоритетам и обеспечивает поэтапное создание полнофункциональной распределенной хеш-таблицы. Критически важными являются первые две фазы, которые реализуют основную функциональность системы.

Успешное завершение проекта потребует значительных инвестиций времени и экспертизы, но результатом станет высококачественная реализация DHT, которая может служить основой для различных распределенных приложений. Система

будет представлять значительную ценность как для образовательных целей, так и для практического применения в реальных проектах.

Технические детали реализации

Пример реализации XOR метрики

Ключевым компонентом системы является корректная реализация XOR метрики в классе KademliaId. Текущая реализация требует существенного расширения для обеспечения всех необходимых операций.

```
public class KademliaId implements BaseId {
    private final byte[] id;
    private static final int ID_LENGTH = 20; // 160 bits = 20
bytes

    public KademliaId(byte[] id) {
        if (id.length != ID_LENGTH) {
            throw new IllegalArgumentException("ID must be
exactly " + ID_LENGTH + " bytes");
        }
        this.id = Arrays.copyOf(id, ID_LENGTH);
    }

    public BigInteger computeDistance(KademliaId other) {
        byte[] result = new byte[ID_LENGTH];
        for (int i = 0; i < ID_LENGTH; i++) {
            result[i] = (byte) (this.id[i] ^ other.id[i]);
        }
        return new BigInteger(1, result); // Unsigned
interpretation
    }

    public int getCommonPrefixLength(KademliaId other) {
        for (int i = 0; i < ID_LENGTH; i++) {
            byte xor = (byte) (this.id[i] ^ other.id[i]);
            if (xor != 0) {
                return i * 8 + Integer.numberOfLeadingZeros(xor
& 0xFF) - 24;
            }
        }
        return ID_LENGTH * 8; // Identical IDs
    }
}
```

Метод getCommonPrefixLength критически важен для определения, в какой k-bucket должен быть помещен узел. Он вычисляет количество общих ведущих

битов между двумя идентификаторами, что определяет уровень в бинарном дереве Kademlia.

Реализация k-buckets и таблицы маршрутизации

K-bucket является основной структурой данных для хранения информации о соседних узлах. Каждый bucket содержит узлы на определенном XOR расстоянии и реализует LRU политику для управления переполнением.

```
public class KBucket {
    private final int k;
    private final LinkedList<NodeInfo> nodes;
    private final Object lock = new Object();

    public KBucket(int k) {
        this.k = k;
        this.nodes = new LinkedList<>();
    }

    public boolean addNode(NodeInfo node) {
        synchronized (lock) {
            if (nodes.contains(node)) {
                // Move to tail (most recently seen)
                nodes.remove(node);
                nodes.addLast(node);
                return true;
            }

            if (nodes.size() < k) {
                nodes.addLast(node);
                return true;
            }

            // Bucket is full, ping least recently seen node
            NodeInfo leastRecent = nodes.getFirst();
            if (pingNode(leastRecent)) {
                // LRU node is still alive, move to tail
                nodes.removeFirst();
                nodes.addLast(leastRecent);
                return false; // New node not added
            } else {
                // LRU node is dead, replace with new node
                nodes.removeFirst();
                nodes.addLast(node);
                return true;
            }
        }
    }
```

```
    public List<NodeInfo> getNodes() {
        synchronized (lock) {
            return new ArrayList<>(nodes);
        }
    }
}
```

Таблица маршрутизации объединяет все k-buckets и предоставляет методы для поиска ближайших узлов.

```
public class RoutingTable {
    private final KademliaId selfId;
    private final KBucket[] buckets;
    private static final int BUCKET_COUNT = 160; // For 160-bit
IDs
    private static final int K = 20;

    public RoutingTable(KademliaId selfId) {
        this.selfId = selfId;
        this.buckets = new KBucket[BUCKET_COUNT];
        for (int i = 0; i < BUCKET_COUNT; i++) {
            buckets[i] = new KBucket(K);
        }
    }

    public void addNode(NodeInfo node) {
        int bucketIndex = getBucketIndex(node.getId());
        if (bucketIndex >= 0 && bucketIndex < BUCKET_COUNT) {
            buckets[bucketIndex].addNode(node);
        }
    }

    public List<NodeInfo> findClosestNodes(KademliaId targetId,
int count) {
        PriorityQueue<NodeInfo> closest = new PriorityQueue<>(
            (a, b) -> a.getId().computeDistance(targetId)
                      .compareTo(b.getId().computeDistance(targetId))
        );

        for (KBucket bucket : buckets) {
            for (NodeInfo node : bucket.getNodes()) {
                closest.offer(node);
                if (closest.size() > count) {
                    closest.poll();
                }
            }
        }

        List<NodeInfo> result = new ArrayList<>(closest);
        Collections.reverse(result); // Closest first
```

```
        return result;
    }

    private int getBucketIndex(KademliaId nodeId) {
        int prefixLength = selfId.getCommonPrefixLength(nodeId);
        return BUCKET_COUNT - 1 - prefixLength;
    }
}
```

Реализация основных сообщений Kademlia

Система сообщений должна поддерживать все четыре основные операции Kademlia. Каждое сообщение должно содержать необходимую информацию для выполнения операции и поддерживать сериализацию.

```
public abstract class KademliaMessage extends BaseMessage {
    protected final KademliaId senderId;
    protected final InetSocketAddress senderAddress;
    protected final String messageId;

    protected KademliaMessage(KademliaId senderId,
InetSocketAddress senderAddress) {
        this.senderId = senderId;
        this.senderAddress = senderAddress;
        this.messageId = UUID.randomUUID().toString();
    }
}

public class FindNodeMessage extends KademliaMessage {
    private final KademliaId targetId;

    public FindNodeMessage(KademliaId senderId,
InetSocketAddress senderAddress,
                           KademliaId targetId) {
        super(senderId, senderAddress);
        this.targetId = targetId;
    }

    public KademliaId getTargetId() {
        return targetId;
    }
}

public class FindNodeResponse extends KademliaMessage {
    private final List<NodeInfo> closestNodes;

    public FindNodeResponse(KademliaId senderId,
InetSocketAddress senderAddress,
                            List<NodeInfo> closestNodes) {
```

```
        super(senderId, senderAddress);
        this.closestNodes = new ArrayList<>(closestNodes);
    }

    public List<NodeInfo> getClosestNodes() {
        return new ArrayList<>(closestNodes);
    }
}
```

Алгоритм итеративного поиска

Итеративный поиск является сердцем алгоритма Kademlia. Он начинает с ближайших известных узлов и постепенно приближается к цели, запрашивая информацию у все более близких узлов.

```
public class IterativeFinder {
    private final RoutingTable routingTable;
    private final MessageSender messageSender;
    private static final int ALPHA = 3; // Parallelism factor
    private static final int K = 20; // Number of closest nodes
to return

    public CompletableFuture<List<NodeInfo>>
findNode(KademliaId targetId) {
        Set<NodeInfo> contacted = new HashSet<>();
        Set<NodeInfo> toContact = new HashSet<>();
        PriorityQueue<NodeInfo> closest = new PriorityQueue<>(
            Comparator.comparing(node ->
node.getId().computeDistance(targetId))
        );

        // Start with closest known nodes
        List<NodeInfo> initial =
routingTable.findClosestNodes(targetId, K);
        toContact.addAll(initial);
        closest.addAll(initial);

        return findNodeIterative(targetId, contacted,
toContact, closest);
    }

    private CompletableFuture<List<NodeInfo>> findNodeIterative(
            KademliaId targetId, Set<NodeInfo> contacted,
            Set<NodeInfo> toContact, PriorityQueue<NodeInfo>
closest) {

        if (toContact.isEmpty()) {
            return CompletableFuture.completedFuture(
```

```
closest.stream().limit(K).collect(Collectors.toList())
            );
        }

        // Select up to ALPHA nodes to contact
        List<NodeInfo> batch = toContact.stream()
            .limit(ALPHA)
            .collect(Collectors.toList());

        toContact.removeAll(batch);
        contacted.addAll(batch);

        // Send parallel requests
        List<CompletableFuture<FindNodeResponse>> futures =
batch.stream()
            .map(node -> messageSender.sendFindNode(node,
targetId))
            .collect(Collectors.toList());

        return CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0]))
            .thenCompose(v -> {
                // Process responses
                for (CompletableFuture<FindNodeResponse>
future : futures) {
                    try {
                        FindNodeResponse response =
future.get();
                        for (NodeInfo node :
response.getClosestNodes()) {
                            if (!contacted.contains(node)) {
                                toContact.add(node);
                                closest.offer(node);
                            }
                        }
                    } catch (Exception e) {
                        // Log error and continue
                    }
                }

                // Continue iteration
                return findNodeIterative(targetId, contacted,
toContact, closest);
            });
    }
}
```

Реализация операций хранения и поиска данных

Операции STORE и FIND_VALUE строятся на основе алгоритма поиска узлов, но добавляют логику работы с данными.

```
public class KademliaStorage {
    private final Map<KademliaId, StoredValue> localStorage;
    private final IterativeFinder finder;
    private final MessageSender messageSender;
    private final RoutingTable routingTable;

    public CompletableFuture<Void> store(KademliaId key, byte[]
value) {
        StoredValue storedValue = new StoredValue(value,
System.currentTimeMillis());
        localStorage.put(key, storedValue);

        // Find K closest nodes and store on them
        return finder.findNode(key)
            .thenCompose(nodes -> {
                List<CompletableFuture<Void>> storeFutures =
nodes.stream()
                    .limit(K)
                    .map(node -> messageSender.sendStore(node,
key, value))
                    .collect(Collectors.toList());

                return CompletableFuture.allOf(
                    storeFutures.toArray(new
CompletableFuture[0])
                );
            });
    }

    public CompletableFuture<Optional<byte[]>>
findValue(KademliaId key) {
        // Check local storage first
        StoredValue local = localStorage.get(key);
        if (local != null && !local.isExpired()) {
            return
CompletableFuture.completedFuture(Optional.of(local.getValue()));
        }

        // Search the network
        return findValueIterative(key, new HashSet<>(),
            new HashSet<>(routingTable.findClosestNodes(key,
K)));
    }

    private CompletableFuture<Optional<byte[]>>
```

```
findValueIterative(
            KademliaId key, Set<NodeInfo> contacted,
Set<NodeInfo> toContact) {

        if (toContact.isEmpty()) {
            return
CompletableFuture.completedFuture(Optional.empty());
        }

        List<NodeInfo> batch = toContact.stream()
            .limit(ALPHA)
            .collect(Collectors.toList());

        toContact.removeAll(batch);
        contacted.addAll(batch);

        List<CompletableFuture<FindValueResponse>> futures =
batch.stream()
            .map(node -> messageSender.sendFindValue(node, key))
            .collect(Collectors.toList());

        return CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0]))
            .thenCompose(v -> {
                for (CompletableFuture<FindValueResponse>
future : futures) {
                    try {
                        FindValueResponse response =
future.get();
                        if (response.hasValue()) {
                            return
CompletableFuture.completedFuture(
                                Optional.of(response.getValue())
                            );
                        } else {
                            // Add new nodes to search
                            for (NodeInfo node :
response.getClosestNodes()) {
                                if (!contacted.contains(node)) {
                                    toContact.add(node);
                                }
                            }
                        }
                    } catch (Exception e) {
                        // Log and continue
                    }
                }

                return findValueIterative(key, contacted,
toContact);
            });
```

```
    }
}
```

Рекомендации по тестированию

Модульное тестирование

Каждый компонент системы должен иметь комплексные модульные тесты. Особое внимание следует уделить тестированию XOR метрики, k-buckets и алгоритмов поиска.

```
@Test
public void testXorDistance() {
    byte[] id1 = new byte[20];
    byte[] id2 = new byte[20];
    id1[0] = (byte) 0xFF;
    id2[0] = (byte) 0x00;

    KademliaId kId1 = new KademliaId(id1);
    KademliaId kId2 = new KademliaId(id2);

    BigInteger distance = kId1.computeDistance(kId2);
    BigInteger expected = new
BigInteger("FF0000000000000000000000000000000000000000", 16);

    assertEquals(expected, distance);
}

@Test
public void testKBucketLRU() {
    KBucket bucket = new KBucket(2);
    NodeInfo node1 = new NodeInfo(new
KademliaId(generateRandomId()),
                                  new
InetSocketAddress("127.0.0.1", 8001));
    NodeInfo node2 = new NodeInfo(new
KademliaId(generateRandomId()),
                                  new
InetSocketAddress("127.0.0.1", 8002));
    NodeInfo node3 = new NodeInfo(new
KademliaId(generateRandomId()),
                                  new
InetSocketAddress("127.0.0.1", 8003));

    assertTrue(bucket.addNode(node1));
    assertTrue(bucket.addNode(node2));

    // Bucket is full, node3 should not be added if node1 is
```

```
alive
    when(mockPinger.ping(node1)).thenReturn(true);
    assertFalse(bucket.addNode(node3));

    // If node1 is dead, it should be replaced by node3
    when(mockPinger.ping(node1)).thenReturn(false);
    assertTrue(bucket.addNode(node3));

    List<NodeInfo> nodes = bucket.getNodes();
    assertEquals(2, nodes.size());
    assertFalse(nodes.contains(node1));
    assertTrue(nodes.contains(node3));
}
```

Интеграционное тестирование

Интеграционные тесты должны проверять взаимодействие между компонентами и корректность работы протокола в целом.

```
@Test
public void testNetworkBootstrap() {
    // Create bootstrap node
    KademliaNode bootstrap = new
KademliaNode(generateRandomId(), 8000);
    bootstrap.start();

    // Create and join additional nodes
    List<KademliaNode> nodes = new ArrayList<>();
    for (int i = 1; i <= 10; i++) {
        KademliaNode node = new
KademliaNode(generateRandomId(), 8000 + i);
        node.join(bootstrap.getAddress());
        nodes.add(node);
    }

    // Wait for network stabilization
    Thread.sleep(5000);

    // Verify that all nodes know about each other
    for (KademliaNode node : nodes) {
        List<NodeInfo> knownNodes = node.getAllKnownNodes();
        assertTrue(knownNodes.size() >= Math.min(10, K));
    }

    // Test data storage and retrieval
    KademliaId key = new KademliaId(generateRandomId());
    byte[] value = "test data".getBytes();

    nodes.get(0).store(key, value).get();
```

```
    Optional<byte[]> retrieved =
nodes.get(5).findValue(key).get();
    assertTrue(retrieved.isPresent());
    assertArrayEquals(value, retrieved.get());
}
```

Тестирование производительности

Тесты производительности должны измерять время отклика системы при различных нагрузках и размерах сети.

```
@Test
public void testSearchPerformance() {
    // Create network of 1000 nodes
    List<KademliaNode> nodes = createNetwork(1000);

    // Store 10000 random key-value pairs
    Map<KademliaId, byte[]> testData = generateTestData(10000);
    storeDataRandomly(nodes, testData);

    // Measure search performance
    long startTime = System.currentTimeMillis();
    int successfulSearches = 0;

    for (KademliaId key : testData.keySet()) {
        KademliaNode randomNode =
nodes.get(random.nextInt(nodes.size()));
        Optional<byte[]> result =
randomNode.findValue(key).get(5, TimeUnit.SECONDS);
        if (result.isPresent()) {
            successfulSearches++;
        }
    }

    long endTime = System.currentTimeMillis();
    double averageSearchTime = (endTime - startTime) / (double)
testData.size();
    double successRate = successfulSearches / (double)
testData.size();

    assertTrue("Success rate should be > 95%", successRate >
0.95);
    assertTrue("Average search time should be < 100ms",
averageSearchTime < 100);
}
```

Метрики и мониторинг

Ключевые метрики производительности

Система должна собирать и предоставлять следующие метрики для мониторинга производительности и здоровья сети:

1. **Метрики сети**:
2. Количество активных соединений
3. Средняя задержка сети до соседних узлов
4. Количество успешных/неуспешных сообщений
5. Пропускная способность сети (байт/сек)
6. **Метрики таблицы маршрутизации**:
7. Количество узлов в каждом k-bucket
8. Средний возраст записей в k-buckets
9. Частота обновления таблицы маршрутизации
10. **Метрики хранилища**:
11. Количество хранимых ключей
12. Размер локального хранилища
13. Частота операций чтения/записи
14. Время отклика операций хранилища
15. **Метрики поиска**:
16. Среднее время поиска узлов
17. Среднее время поиска значений
18. Количество шагов в итеративном поиске
19. Процент успешных поисков

Реализация системы мониторинга

```
public class KademliaMetrics {
    private final MeterRegistry meterRegistry;
    private final Timer searchTimer;
    private final Counter successfulSearches;
    private final Counter failedSearches;
    private final Gauge routingTableSize;
    private final Gauge localStorageSize;
```

```
    public KademliaMetrics(MeterRegistry meterRegistry,
                           RoutingTable routingTable,
                           LocalStorage localStorage) {
        this.meterRegistry = meterRegistry;

        this.searchTimer =
Timer.builder("kademlia.search.duration")
            .description("Time taken for search operations")
            .register(meterRegistry);

        this.successfulSearches =
Counter.builder("kademlia.search.success")
            .description("Number of successful searches")
            .register(meterRegistry);

        this.failedSearches =
Counter.builder("kademlia.search.failure")
            .description("Number of failed searches")
            .register(meterRegistry);

        this.routingTableSize =
Gauge.builder("kademlia.routing.table.size")
            .description("Number of nodes in routing table")
            .register(meterRegistry, routingTable, rt ->
rt.getTotalNodes());

        this.localStorageSize =
Gauge.builder("kademlia.storage.local.size")
            .description("Number of items in local storage")
            .register(meterRegistry, localStorage, ls ->
ls.getSize());
    }

    public Timer.Sample startSearchTimer() {
        return Timer.start(meterRegistry);
    }

    public void recordSuccessfulSearch(Timer.Sample sample) {
        sample.stop(searchTimer);
        successfulSearches.increment();
    }

    public void recordFailedSearch(Timer.Sample sample) {
        sample.stop(searchTimer);
        failedSearches.increment();
    }
}
```

Веб-интерфейс для мониторинга

Рекомендуется создать простой веб-интерфейс для визуализации состояния узла и сети. Интерфейс должен отображать:

- Текущее состояние таблицы маршрутизации
- Графики производительности в реальном времени
- Список активных соединений
- Статистику операций хранения и поиска
- Логи системных событий

```
 @RestController
@RequestMapping("/api/kademlia")
public class KademliaMonitoringController {

    private final KademliaNode node;
    private final KademliaMetrics metrics;

    @GetMapping("/status")
    public ResponseEntity<NodeStatus> getNodeStatus() {
        NodeStatus status = NodeStatus.builder()
            .nodeId(node.getId().toString())
            .address(node.getAddress().toString())
            .uptime(node.getUptime())
            .routingTableSize(node.getRoutingTable().getTotalNodes())
            .localStorageSize(node.getLocalStorage().getSize())
            .build();

        return ResponseEntity.ok(status);
    }

    @GetMapping("/routing-table")
    public ResponseEntity<List<BucketInfo>> getRoutingTable() {
        List<BucketInfo> buckets =
node.getRoutingTable().getBuckets()
            .stream()
            .map(bucket -> BucketInfo.builder()
                .index(bucket.getIndex())
                .nodeCount(bucket.getNodeCount())
                .nodes(bucket.getNodes())
                .build())
            .collect(Collectors.toList());

        return ResponseEntity.ok(buckets);
    }

    @GetMapping("/metrics")
    public ResponseEntity<Map<String, Object>> getMetrics() {
        Map<String, Object> metricsData = new HashMap<>();
```

```
        meterRegistry.getMeters().forEach(meter -> {
            if (meter.getId().getName().startsWith("kademlia"))
{
                metricsData.put(meter.getId().getName(),

meter.measure().iterator().next().getValue());
            }
        });

        return ResponseEntity.ok(metricsData);
    }
}
```

Этот технический план предоставляет конкретные примеры реализации и детальные рекомендации для завершения системы DHT. Следование этому плану позволит создать полнофункциональную и производительную реализацию алгоритма Kademlia, которая будет соответствовать современным стандартам качества программного обеспечения.