ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

«Distributed Hash Table»

|  |  |
| --- | --- |
| Выполнили студенты | Щеголева Виктория Андреевна  Помелова Анастасия Викторовна |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИА-331 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | ст. преп. Кафедры ВС Д. М. Берлизов |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2024

**Оглавление**

[Введение 3](#_Toc184121850)

[Распределенные хеш-таблицы: определение и принципы хеширования данных. 4](#_Toc184121851)

[Основные особенности Distributed Hash Table 6](#_Toc184121852)

[Методы разрешения коллизий 6](#_Toc184121853)

[Существующие реализации распределенных хеш-таблиц 8](#_Toc184121854)

[Хранение и поиск данных в распределенных сетях 11](#_Toc184121855)

[Оценка производительности и надежности 13](#_Toc184121856)

[Исследование эффективности DHT. 14](#_Toc184121857)

[Заключения. 16](#_Toc184121858)

[Список использованных источников. 17](#_Toc184121859)

[Приложение. 18](#_Toc184121860)

# **Введение.**

За последние годы наблюдается быстрый рост систем массового распространения, которые привлекают миллионы пользователей. Приложения, такие как мессенджеры, платформы для обмена файлами и распространения контента, становятся всё более актуальными. Например, Skype за один год привлёк более 2,5 миллиона новых пользователей, а свыше 50% интернет-трафика проходит через BitTorrent. Это указывает на начало новой эры в разработке распределенных систем.

Разработчики таких систем сталкиваются с важными задачами, включая масштабируемость, гибкость и скорость развертывания. Парадигма P2P, или одноранговая модель, основывается на создании распределённых систем, где все участники обладают равными функциональными возможностями и ответственностью. Эти системы должны координировать свои действия без централизованного управления.

Масштабируемость считается критически важным аспектом для P2P-систем, однако не все из них это подтверждают. Примером может служить проблема поиска информации, которая остаётся актуальной для всех распределённых и P2P-систем. Мы изучим управление данными и методы их извлечения в распределённых системах, сравнив три основных подхода и выяснив, что некоторые из них могут сталкиваться с проблемами масштабируемости.

Чтобы решить эти вопросы, мы предлагаем концепцию распределенной хэш-таблицы (DHT) как эффективное средство для проектирования и развертывания высокомасштабируемых систем. В модели P2P все узлы сети имеют равные права и могут исполнять функции как клиента, так и сервера, в отличие от традиционных клиент-серверных архитектур, где один или несколько серверов обрабатывают множество клиентов. В P2P-сетях каждый узел может напрямую обмениваться данными с другими узлами.

# **Распределенные хеш-таблицы: определение и принципы хеширования данных.**

**1.Определение**

*Распределённая хэш-таблица (DHT)* представляет собой распределённую систему, организующую хранение и доступ к данным в виде пар "ключ-значение". В отличие от центральных хранилищ данных, DHT распределяет данные между множеством узлов в сети, что позволяет создать масштабируемую и устойчивая к сбоям инфраструктуру. Каждый узел в DHT хранит только часть данных, а взаимодействие между узлами осуществляется без необходимости в централизованном управляющем элементе.

**2. Принципы хеширования данных**

*Хеширование ключей:*

Ключи и значения: В DHT данные хранятся в виде пар "ключ-значение". Ключи преобразуются в хеш-значения с помощью хеш-функции.

Фиксированный размер: Хеш-функция генерирует фиксированный размер хеша, что позволяет равномерно распределять ключи по узлам.

*Распределение данных:*

Хеш-пространство: Все возможные хеш-значения формируют виртуальное пространство (например, от 0 до 2^m, где m — количество бит в хеше). Каждый узел отвечает за определенный диапазон хешей.

Распределение узлов: Узлы сети могут быть добавлены или удалены динамически. При этом данные перераспределяются по узлам, чтобы поддерживать баланс нагрузки.

*Хранение данных:*

Ответственность узлов: Каждый узел хранит данные, соответствующие определенному диапазону хеш-значений. Например, если узел имеет хеш 5, он может хранить все ключи, чьи хеши попадают в диапазон, который ему назначен.

*Репликация:*

Для повышения надежности и доступности данные могут быть реплицированы на нескольких узлах. Это означает, что один и тот же ключ может храниться на нескольких узлах для защиты от потери данных.

*Поиск данных:*

Логика поиска: чтобы найти значение по ключу, система сначала вычисляет хеш ключа, а затем определяет, какой узел отвечает за этот хеш. Запрос отправляется на соответствующий узел.

Маршрутизация: если узел не содержит запрашиваемое значение, он может перенаправить запрос к другому узлу, который, вероятно, имеет нужные данные. Это может происходить через несколько промежуточных узлов.

*Обработка изменений:*

Динамическое добавление и удаление узлов: Когда узлы добавляются или удаляются, DHT автоматически перераспределяет данные между оставшимися узлами. Это может быть выполнено с помощью алгоритмов, которые минимизируют количество перемещаемых данных.

Перераспределение: При удалении узла его данные должны быть переданы другим узлам. Это может быть сделано через процесс репликации или миграции данных.

*Устойчивость к сбоям:*

Обнаружение сбоев: Узлы могут периодически проверять состояние друг друга и обнаруживать сбои. Если узел выходит из строя, другие узлы могут взять на себя его функции.

*Резервное копирование данных:*

Репликация данных помогает избежать потери информации в случае сбоя одного или нескольких узлов.

# **Основные особенности Distributed Hash Table**

1*. Децентрализация:* нет центрального узла или сервера, который управляет всей таблицей. Каждый узел является равноправным участником и может как хранить, так и запрашивать данные.

2. *Масштабируемость:* DHT могут эффективно масштабироваться с увеличением числа узлов. При добавлении новых узлов система может перераспределять данные без значительных затрат.

3. *Эффективность поиска:* Поиск данных в DHT осуществляется с использованием алгоритмов маршрутизации, таких как Kademlia или Chord, которые обеспечивают логарифмическое время поиска в зависимости от количества узлов в сети.

4. *Устойчивость к сбоям:* DHT могут продолжать функционировать даже при выходе из строя нескольких узлов. Данные обычно дублируются на нескольких узлах для повышения надежности.

5. *Распределение нагрузки:* Данные равномерно распределяются между узлами, что помогает избежать перегрузки отдельных узлов и обеспечивает более равномерное использование ресурсов.

6. *Хеширование:* Ключи преобразуются в хеши, которые определяют местоположение данных в сети. Это позволяет быстро находить нужные записи. Иллюстрации оформляются по центру. Желательно использовать собственноручно сделанные иллюстрации в векторном формате (SVG). На каждый рисунок должна иметься ссылка в тексте отчёта.

# **Методы разрешения коллизий**

1. **Частичное хеширование (Consistent Hashing)**

*Принцип*: вместо того чтобы использовать фиксированное количество узлов для хранения данных, частичное хеширование распределяет данные по узлам на основе их хеш-значений. Если возникает коллизия, данные могут быть перенаправлены на следующий доступный узел в хеш-пространстве.

*Преимущества*: Этот метод позволяет равномерно распределять нагрузку и минимизировать перемещение данных при добавлении или удалении узлов.

1. **Репликация данных**

*Принцип*: Каждый ключ может храниться на нескольких узлах (репликах), что уменьшает вероятность потери данных из-за коллизий. Если один узел не может обработать запрос, система может обратиться к другим узлам, хранящим ту же информацию.

*Преимущества*: увеличивает доступность и надежность системы.

1. **Динамическое перераспределение данных**

*Принцип*: при обнаружении коллизии система может динамически перераспределить данные между узлами. Например, если два ключа имеют одинаковый хеш, один из них может быть перенаправлен на другой узел.

*Преимущества*: это позволяет избежать постоянного хранения нескольких значений на одном узле и поддерживать баланс нагрузки.

1. **Обработка конфликтов на уровне приложения**

*Принцип*: в некоторых системах разрешение коллизий может быть обработано на уровне приложения. Например, приложение может использовать дополнительные метаданные или временные метки для различения коллизирующих ключей.

*Преимущества*: позволяет более гибко управлять данными и принимать решения на основе бизнес-логики.

1. **Использование нескольких хеш-функций**

*Принцип*: при возникновении коллизии можно применить другую хеш-функцию для генерации нового индекса. Это может быть сделано итеративно до тех пор, пока не будет найден свободный индекс.

*Преимущества*: увеличивает вероятность нахождения уникального индекса для каждого ключа.

1. **Системы с версионностью**

*Принцип*: В некоторых системах можно хранить разные версии значения для одного и того же ключа. Каждая версия будет иметь свой уникальный идентификатор или временную метку.

*Преимущества*: Позволяет сохранять историю изменений и избегать потерь данных при конфликтах.

# **Существующие реализации распределенных хеш-таблиц**

1. **Chord**

**Chord** — это один из первых и наиболее известных протоколов для реализации распределённых хэш-таблиц (DHT). Он был разработан для обеспечения эффективного поиска и хранения данных в распределённых системах.

* 1. **Основные характеристики Chord:**

*1) Узлы и идентификаторы:*

• Каждый узел в сети имеет уникальный идентификатор (ID), который обычно представляет собой хеш-значение, полученное от его IP-адреса или другого уникального значения.

• Идентификаторы узлов распределены по кругу, что позволяет использовать кольцевую структуру.

*2) Кольцевая структура:*

• Узлы организованы в виде логического кольца, где каждый узел знает о своем непосредственном соседе по часовой стрелке.

• Это позволяет эффективно находить узлы и данные, используя алгоритмы поиска.

*3) Хеширование ключей:*

• Ключи данных также хешируются для получения идентификаторов, которые затем размещаются на кольце.

• Каждый ключ хранится на узле, чей ID является первым узлом, который больше или равен ID ключа.

*4) Поиск данных:*

• Для поиска значения по ключу Chord использует алгоритм «поиска по маршруту». Узел инициирует поиск, отправляя запрос к своему ближайшему «пальцу» (finger) — узлу, который находится на определённом расстоянии от него.

• Каждый узел поддерживает таблицу пальцев, которая содержит ссылки на другие узлы, что позволяет быстро перемещаться по кольцу.

*5) Таблицы пальцев:*

• Каждому узлу в Chord соответствует таблица пальцев, которая содержит ссылки на узлы, находящиеся на различных расстояниях от текущего узла.

• Таблица пальцев позволяет ускорить процесс поиска, так как каждый узел может сразу же обратиться к узлам, которые находятся далеко от него.

*6) Обработка изменений:*

• Когда узел добавляется или удаляется из сети, Chord автоматически обновляет свои таблицы пальцев и связи с соседями.

• Это позволяет системе оставаться устойчивой к изменениям и поддерживать баланс нагрузки

* 1. **Преимущества Chord:**

1. *Масштабируемость:* Chord может эффективно работать в сетях с большим количеством узлов.
2. *Устойчивость к сбоям:* Система может продолжать функционировать даже при выходе из строя нескольких узлов.
3. *Простота реализации:* Алгоритмы Chord относительно просты в реализации и понимании.
   1. **Недостатки Chord:**

* Задержки при поиске: хотя Chord обеспечивает логарифмическое время поиска, в больших сетях задержки могут быть значительными.
* Неравномерность нагрузки: если ключи неравномерно распределены, некоторые узлы могут стать перегруженными.

1. **Kademlia**

**Kademlia** — это один из наиболее известных протоколов для реализации распределённых хэш-таблиц (DHT), который используется для организации хранения и поиска данных в распределённых системах. Kademlia была разработана для обеспечения высокой устойчивости к сбоям, эффективного поиска и минимизации задержек в сети.

* 1. **Основные характеристики Kademlia:**

*Идентификаторы узлов и ключей:*

• Каждый узел в сети имеет уникальный идентификатор (ID), который обычно представляет собой 160-битное значение, полученное с помощью хеш-функции (например, SHA-1).

• Ключи, по которым происходит поиск данных, также хешируются для получения идентификаторов.

*Кольцевая структура:*

• Узлы организованы в логическую структуру, подобную кольцу, где расстояние между идентификаторами определяется по метрике XOR.

• Это позволяет легко находить узлы, которые находятся ближе всего к искомому идентификатору.

*Бинды (k-buckets):*

• Каждый узел поддерживает структуру данных, называемую "k-bucket", которая хранит информацию о других узлах в сети.

• Каждый k-bucket содержит до k узлов, которые имеют схожие идентификаторы. Это позволяет эффективно управлять подключениями и обрабатывать запросы.

*Поиск данных:*

• Для поиска значения по ключу Kademlia использует алгоритм, основанный на "итеративном" или "рекурсивном" поиске.

• Узел отправляет запрос на ближайшие узлы к искомому идентификатору, которые затем возвращают информацию о других узлах, приближающихся к искомому ключу.

*Устойчивость к сбоям:*

• Kademlia обеспечивает высокую устойчивость к сбоям благодаря дублированию данных. Каждый ключ может храниться на нескольких узлах.

• При выходе из строя узла система автоматически обновляет свои данные, чтобы сохранить доступность информации.

*Распределение нагрузки:*

• Kademlia использует механизм, позволяющий равномерно распределять нагрузку среди узлов, что предотвращает перегрузку отдельных узлов.

* 1. **Преимущества Kademlia:**

*Эффективность поиска:* Поиск происходит за логарифмическое время относительно количества узлов в сети.

*Устойчивость к сбоям:* Система может продолжать работать даже при потере значительного количества узлов.

*Гибкость:* Kademlia может адаптироваться к изменениям в сети, поддерживая динамическое добавление и удаление узлов.

* 1. **Недостатки Kademlia:**

*Задержки при поиске:* хотя Kademlia обеспечивает быстрое время ответа, задержки могут увеличиваться при больших расстояниях между узлами.

*Сложность реализации:* реализация Kademlia может быть более сложной по сравнению с другими DHT-протоколами из-за необходимости управления к-бакетами и обработкой ошибок.

# **Хранение и поиск данных в распределенных сетях**

**Хранение и поиск данных в распределённых сетях —** это важные аспекты, которые обеспечивают эффективное управление данными в системах, состоящих из множества узлов (компьютеров или серверов). Такие сети могут включать облачные хранилища, пиринговые сети и другие распределённые системы. Рассмотрим ключевые концепции, технологии и методы, используемые для хранения и поиска данных в таких сетях.

1. **Основные концепции**

*Распределённые системы:* Системы, где данные и вычисления распределены между несколькими узлами, которые могут находиться в разных географических местах.

*Узлы:* Каждый компьютер или сервер в распределённой сети называется узлом. Узлы могут выполнять разные функции, включая хранение данных, обработку запросов и управление сетью.

*Доступность и отказоустойчивость:* важно, чтобы данные были доступны даже в случае сбоя одного или нескольких узлов. Это достигается за счёт репликации данных и использования алгоритмов для восстановления после сбоев.

1. **Методы хранения данных**

*Репликация:* Данные копируются на несколько узлов для повышения доступности и надёжности. Если один узел выходит из строя, данные всё ещё можно получить с другого узла.

*Шардинг:* Данные разбиваются на части (шарды), которые распределяются по различным узлам. Это позволяет эффективно масштабировать систему и улучшает производительность.

*Объектные хранилища:* используются для хранения неструктурированных данных (например, файлов, изображений). Примеры включают Amazon S3 и Google Cloud Storage.

*Базы данных NoSQL:* Такие базы данных (например, Cassandra, MongoDB) специально разработаны для работы в распределённых средах и обеспечивают гибкость в хранении и управлении данными.

1. **Методы поиска данных**

*Распределённые хэш-таблицы (DHT):* это структура данных, которая позволяет узлам быстро находить данные по ключу. Примеры включают Kademlia и Chord. Каждый узел отвечает за определённый диапазон ключей, что позволяет эффективно искать данные.

*Поиск по метаданным:* Данные могут быть индексированы с использованием метаданных, что облегчает их поиск. Например, можно использовать Elasticsearch для поиска по текстовым данным.

*Кэширование:* для ускорения поиска часто используемых данных можно использовать кэширование на уровне узлов или на уровне сети.

*Алгоритмы маршрутизации:* В DHT используются алгоритмы маршрутизации для определения пути к нужному узлу. Это позволяет минимизировать количество необходимых запросов и задержек при поиске.

1. **Проблемы и вызовы**

*Согласованность данных:* Обеспечение согласованности данных между узлами может быть сложной задачей, особенно при использовании репликации. Существуют разные модели согласованности (например, строгая, конечная, eventual consistency).

*Безопасность:* Распределённые сети подвержены различным угрозам, таким как атаки на узлы или перехват данных. Необходимы механизмы шифрования и аутентификации.

*Управление сетевыми ресурсами:* Эффективное использование ресурсов сети (пропускная способность, память) является важной задачей для обеспечения производительности.

# **Оценка производительности и надежности**

1. **Метрики производительности распределенных хеш-таблиц**

*Время отклика:* Измерение времени, которое требуется для выполнения операции на распределенной хеш-таблице.

*Трафик сети:* Количество данных, передаваемых между узлами, которое может влиять на производительность сети.

*Скорость записи и чтения данных:* Оценка производительности в зависимости от количества и размера записей, которые могут быть записаны и прочитаны за единицу времени.

*Ёмкость и масштабируемость:* Оценка максимального количества данных, которые возможно сохранить в распределенной хеш-таблице и её способность масштабирования при увеличении количества узлов.

1. **Проблемы надежности и их решения**

*Отказ узла:* Использование репликации для обеспечения доступности данных в случае сбоя одного из узлов.

*Синхронизация данных:* Использование механизмов синхронизации для обеспечения целостности данных и избежания конфликтов.

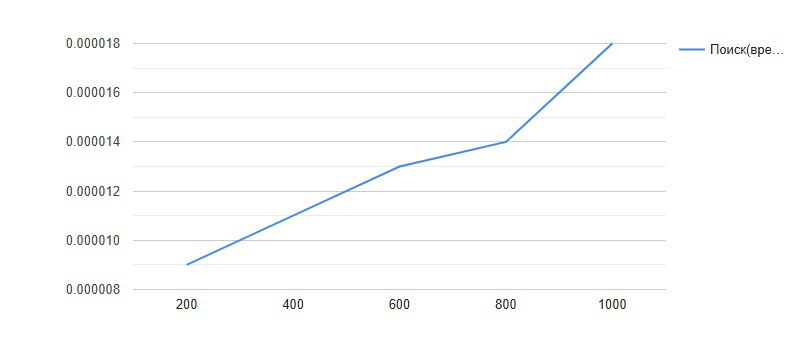
*Защита данных:* Использование механизмов для шифрования и безопасной передачи данных между узлами.

*Ненадежность сети:* Использование механизмов перезапуска и повторной отправки запросов при ненадежной передаче данных.

*Различные типы отказов:* Создание механизма обнаружения и лечения сбоев, включая отказ машинного оборудования, ошибки сети и программные сбои.

# **Исследование эффективности DHT.**

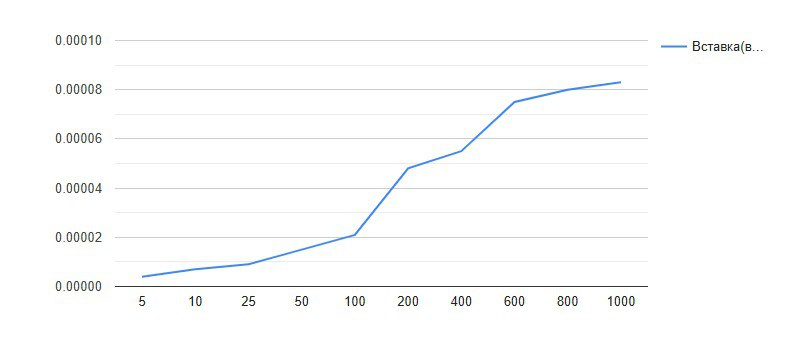
1. **Chord**



Количество элементов

Рис.1 - График работы функции вставки

Время, с



Количество элементов

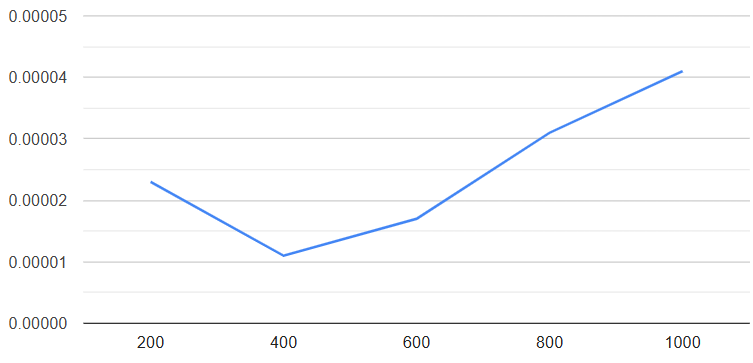
Рис.2 - График работы функции поиска

Время, с

Сложность вставки и поиска O(log N), где N - количество узлов.

Сама сложность также зависит от загруженности узлов и соответственно от количества узлов. Функции работают равномерно.

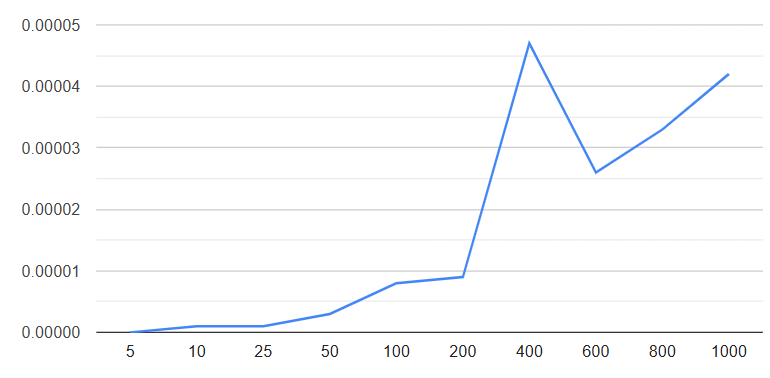
1. **Kademlia**

****

Количество элементов

Рис.3- График работы функции вставки

Время, с



Количество элементов

Рис.4- График работы функции поиска

Время, с

Сложность вставки и поиска O(log N), где N - количество узлов.

Сама сложность также зависит от загруженности узлов и соответственно от количества узлов. Функция работает быстрее, чем Chord, возможны сбои при большом количестве узлов

# **Заключения.**

В результате выполнения работы разработан и исследован алгоритм распределенной хеш-таблицы (DHT), который обеспечивает эффективное распределение и хранение данных в сети. Осуществлено моделирование разработанного алгоритма, что позволило выявить его преимущества в контексте масштабируемости и устойчивости к сбоям. Показано, что алгоритм демонстрирует высокую производительность при увеличении числа узлов в системе, а также обеспечивает быструю и надежную маршрутизацию запросов. Результаты моделирования подтвердили его пригодность для использования в распределенных системах, таких как P2P-сети, что открывает возможности для дальнейших исследований и практического применения в различных областях, включая облачные вычисления и интернет вещей.

# **Список использованных источников.**

1. Ахо А. В., Хопкрофт Д., Ульман Д. Д. Структуры данных и алгоритмы. – М.: Вильямс, 2001. – 384 с.

2. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ.– 3-е изд. – М.: Вильямс, 2013. – 1328 с.

3. Кормен Т. Х. Алгоритмы: Вводный курс. - М.: Вильямс, 2014. - 208 с.

4. Левитин А. В. Алгоритмы: введение в разработку и анализ. – М.: Вильямс, 2006.–576 с.

5. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ. Структуры данных.

Сортировка. Поиск. – К.: ДиаСофт, 2001. – 688 с.

6. Скиена С. С. Алгоритмы. Руководство по разработке. – 2-е изд. – СПб: БХВ, 2011 –720 с.

7. Макконнелл Дж. Основы современных алгоритмов. – 2-е изд. – М.: Техносфера, 2004.– 368 с.

8. Миллер Р. Последовательные и параллельные алгоритмы: общий подход. – М.:

БИНОМ, 2006. – 406 с.

9. Сегаран Т. Программируем коллективный разум. – М.: Символ-Плюс, 2008. – 368 с.

10. Хараламбос Марманис X., Бабенко Д. Алгоритмы интеллектуального Интернета. – М.:

Символ-Плюс, 2011. – 480 с.

11. Маннинг К., Рагхаван П., Шютце Х. Введение в информационный поиск. – М.:

Вильямс, 2011. – 528 с.

12. Каллан Р. Основные концепции нейронных сетей. – М.: Вильямс, 2003. – 288 с.

# **Приложение.**

**Kademlia**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#include <sys/time.h>

#include <time.h>

#include <math.h>

#define MAX\_NODES 125

#define NODE\_ID\_LENGTH 20

#define BUCKET\_SIZE 8

long long current\_time\_in\_microseconds() {

    struct timeval tv;

    gettimeofday(&tv, NULL);

    return (long long)(tv.tv\_sec) \* 1000000 + tv.tv\_usec;

}

typedef struct Node {

    uint8\_t id[NODE\_ID\_LENGTH];

    struct Node\* next;

} Node;

typedef struct Bucket {

    Node\* nodes[BUCKET\_SIZE];

    int count;

} Bucket;

typedef struct Kademlia {

    Bucket buckets[MAX\_NODES];

} Kademlia;

Node\* create\_node(uint8\_t\* id) {

    Node\* new\_node = malloc(sizeof(Node));

    memcpy(new\_node->id, id, NODE\_ID\_LENGTH);

    new\_node->next = NULL;

    return new\_node;

}

void xor\_ids(uint8\_t\* id1, uint8\_t\* id2, uint8\_t\* result) {

    for (int i = 0; i < NODE\_ID\_LENGTH; i++) {

        result[i] = id1[i] ^ id2[i];

    }

}

void hash\_key(const char\* key, uint8\_t\* id) {

    for (int i = 0; i < NODE\_ID\_LENGTH; i++) {

        id[i] = (uint8\_t)(key[i % strlen(key)] + i);

    }

}

void add\_node(Kademlia\* kademlia, uint8\_t\* id) {

    int bucket\_index = 0;

    Bucket\* bucket = &kademlia->buckets[bucket\_index];

    if (bucket->count < BUCKET\_SIZE) {

        bucket->nodes[bucket->count++] = create\_node(id);

        // printf("Node added: ");

        // for (int i = 0; i < NODE\_ID\_LENGTH; i++) {

        //     printf("%02x", id[i]);

        // }

        // printf("\n");

    } else {

   //     printf("Bucket is full, cannot add node.\n");

    }

}

Node\* find\_node(Kademlia\* kademlia, uint8\_t\* id) {

    int bucket\_index = 0;

    Bucket\* bucket = &kademlia->buckets[bucket\_index];

    for (int i = 0; i < bucket->count; i++) {

        if (memcmp(bucket->nodes[i]->id, id, NODE\_ID\_LENGTH) == 0) {

            return bucket->nodes[i];

        }

    }

    return NULL;

}

void remove\_node(Kademlia\* kademlia, uint8\_t\* id) {

    int bucket\_index = 0;

    Bucket\* bucket = &kademlia->buckets[bucket\_index];

    for (int i = 0; i < bucket->count; i++) {

        if (memcmp(bucket->nodes[i]->id, id, NODE\_ID\_LENGTH) == 0) {

            free(bucket->nodes[i]);

            bucket->nodes[i] = bucket->nodes[--bucket->count];

            printf("Node removed: ");

            for (int j = 0; j < NODE\_ID\_LENGTH; j++) {

                printf("%02x", id[j]);

            }

            printf("\n");

            return;

        }

    }

    printf("Node not found.\n");

}

void free\_kademlia(Kademlia\* kademlia) {

    for (int i = 0; i < MAX\_NODES; i++) {

        for (int j = 0; j < kademlia->buckets[i].count; j++) {

            free(kademlia->buckets[i].nodes[j]);

        }

    }

    free(kademlia);

}

int main() {

    Kademlia\* kademlia = malloc(sizeof(Kademlia));

    Kademlia\* kademlia1 = malloc(sizeof(Kademlia));

    memset(kademlia, 0, sizeof(Kademlia));

    memset(kademlia1, 0, sizeof(Kademlia));

    long long start\_time\_add, end\_time\_add;

    printf("Add nodes\n");

   for (int i = 0; i < 1000; i+=5) {

    start\_time\_add = current\_time\_in\_microseconds();

    for (int j = 0; j < i; j++) {

            uint8\_t id[NODE\_ID\_LENGTH] = {0};

            for (int k = 0; k < NODE\_ID\_LENGTH; k++) {

                id[k] = (uint8\_t)((k + i + j) % 256);

            }

            add\_node(kademlia, id);

    }

    end\_time\_add = current\_time\_in\_microseconds();

    double add\_time = (double)(end\_time\_add - start\_time\_add) / 1000000.0;

    printf("%f\n", add\_time);

   }

    long long start\_time\_find, end\_time\_find;

    printf("Find nodes\n");

    for (int i = 0; i < 1000; i+=5) {

        start\_time\_add = current\_time\_in\_microseconds();

        for (int j = 0; j < i; j++) {

        uint8\_t id[NODE\_ID\_LENGTH] = {0};

        for (int j = 0; j < NODE\_ID\_LENGTH; j++) {

            id[j] = (uint8\_t)((j + i) % 256);

        }

        find\_node(kademlia, id);

        }

        end\_time\_add = current\_time\_in\_microseconds();

        double add\_time = (double)(end\_time\_add - start\_time\_add) / 1000000.0;

        printf("%f\n", add\_time);

    }

    uint8\_t id1[NODE\_ID\_LENGTH] = {0};

    for (int j = 0; j < NODE\_ID\_LENGTH; j++) {

            id1[j] = (uint8\_t)((j + 1) % 256);

        }

    add\_node(kademlia1, id1);

    remove\_node(kademlia1, id1);

    free\_kademlia(kademlia);

    return 0;

}

**Chord**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_NODES 10

#define HASH\_SPACE 512

typedef struct KeyValue {

    char\* key;

    char\* value;

    struct KeyValue\* next;

} KeyValue;

typedef struct Node {

    int id;

    struct Node\* next;

    KeyValue\* data; // Указатель на список пар ключ-значение

} Node;

typedef struct Chord {

    Node\* nodes[MAX\_NODES];

    int node\_count;

} Chord;

// Функция для создания нового узла

Node\* create\_node(int id) {

    Node\* new\_node = malloc(sizeof(Node));

    new\_node->id = id;

    new\_node->next = NULL;

    new\_node->data = NULL; // Инициализация данных

    return new\_node;

}

// Хеш-функция для получения идентификатора

int hash(const char\* key) {

    unsigned long hash = 5381;

    int c;

    while ((c = \*key++)) {

        hash = ((hash << 5) + hash) + c; // hash \* 33 + c

    }

    return hash % HASH\_SPACE; // Возвращаем хеш в пределах HASH\_SPACE

}

// Создание Chord

Chord\* create\_chord() {

    Chord\* chord = malloc(sizeof(Chord));

    chord->node\_count = 0;

    memset(chord->nodes, 0, sizeof(chord->nodes)); // Инициализация узлов

    return chord;

}

// Добавление узла в Chord

void add\_node(Chord\* chord) {

    if (chord->node\_count >= MAX\_NODES) {

        printf("Maximum number of nodes reached.\n");

        return;

    }

    Node\* new\_node = create\_node(chord->node\_count);

    chord->nodes[chord->node\_count++] = new\_node;

    printf("Node %d added.\n", new\_node->id);

}

// Получение узла для ключа

Node\* get\_node(Chord\* chord, const char\* key) {

    int key\_hash = hash(key);

    for (int i = 0; i < chord->node\_count; i++) {

        if (chord->nodes[i]->id >= key\_hash) {

            return chord->nodes[i];

        }

    }

    return chord->nodes[0]; // Возвращаем первый узел, если не нашли

}

// Добавление данных в DHT

void add\_data(Chord\* chord, const char\* key, const char\* value) {

    Node\* node = get\_node(chord, key);

    if (node) {

        KeyValue\* new\_kv = malloc(sizeof(KeyValue));

        new\_kv->key = strdup(key);

        new\_kv->value = strdup(value);

        new\_kv->next = node->data; // Вставляем в начало списка

        node->data = new\_kv; // Обновляем указатель на данные

    } else {

        printf("No nodes available.\n");

    }

}

// Поиск данных в DHT

const char\* search\_data(Chord\* chord, const char\* key) {

    Node\* node = get\_node(chord, key);

    if (node && node->data) {

        KeyValue\* current = node->data;

        while (current != NULL) {

            if (strcmp(current->key, key) == 0) {

                return current->value; // Возвращаем значение, если нашли ключ

            }

            current = current->next;

        }

    }

    return NULL; // Если не нашли значение

}

// Удаление данных из DHT

void remove\_data(Chord\* chord, const char\* key) {

    Node\* node = get\_node(chord, key);

    if (node && node->data) {

        KeyValue\* current = node->data;

        KeyValue\* prev = NULL;

        while (current != NULL) {

            if (strcmp(current->key, key) == 0) {

                if (prev) {

                    prev->next = current->next; // Удаляем элемент из списка

                } else {

                    node->data = current->next; // Удаляем первый элемент

                }

                free(current->key);

                free(current->value);

                free(current);

                printf("Data '%s' removed from Node %d.\n", key, node->id);

                return;

            }

            prev = current;

            current = current->next;

        }

        printf("Data '%s' not found in Node %d.\n", key, node->id);

    } else {

        printf("No nodes available or no data to remove.\n");

    }

}

void free\_chord(Chord\* chord) {

    for (int i = 0; i < chord->node\_count; i++) {

        KeyValue\* current = chord->nodes[i]->data;

        while (current) {

            KeyValue\* temp = current;

            current = current->next;

            free(temp->key);

            free(temp->value);

            free(temp);

        }

        free(chord->nodes[i]);

    }

    free(chord);

}

int main() {

    Chord\* chord = create\_chord();

    // Создание узлов

    add\_node(chord);

    add\_node(chord);

    add\_node(chord);

    //clock\_t start\_time = clock();

    // Добавление данных

    for (int i = 0; i < 800; i++) {

        // Создаем строку для хранения символа

        char ch[2]; // Массив из 2-х элементов (1 символ + 1 для '0')

        ch[0] = i + '0'; // Преобразуем число в символ

        ch[1] = '0'; // Завершаем строку нулевым символом

        add\_data(chord, ch, ch); // Передаем строку в функцию

    }

    //  clock\_t end\_time = clock();

    // double time\_taken = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

    // printf("Время выполнения: %f секунд\n", time\_taken);

    char ch[2]; // Массив из 2-х элементов (1 символ + 1 для '0')

    ch[0] = 20 + '0'; // Преобразуем число в символ

    ch[1] = '0'; // Завершаем строку нулевым символом

    // Поиск данных

    clock\_t start\_time = clock();

    const char\* value = search\_data(chord, ch);

    if (value) {

        printf("Found: key1 -> %s\n", value);

    } else {

        printf("Data for 'key1' not found.\n");

    }

    clock\_t end\_time = clock();

    double time\_taken = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

    printf("Время выполнения: %f секунд\n", time\_taken);

    // Удаление данных

    remove\_data(chord, "key1");

    // Попробуем снова найти удаленные данные

    value = search\_data(chord, "key1");

    if (value) {

        printf("Found: key1 -> %s\n", value);

    } else {

        printf("Data for 'key1' not found.\n");

    }

    // Освобождение памяти

    free\_chord(chord);

    return 0;

}