Воронежский государственный университет

На правах рукописи

Шилов Сергей Николаевич

МОДЕЛИ И АЛГОРИТМЫ БАЛАНСИРОВКИ НАГРУЗКИ В КЛАСТЕРНОЙ СИСТЕМЕ С ПОДДЕРЖКОЙ МЕХАНИЗМА РЕПЛИКАЦИИ

05.13.17 – Теоретические основы информатики

Диссертация на соискание ученой степени кандидата технических наук

Научный руководитель
доктор физ-мат. наук, профессор
Кургалин Сергей Дмитриевич

Воронеж – 2015 г.

Оглавление

ВВЕДЕНИЕ 5

ГЛАВА 1. КОМПЬЮТЕРНЫЕ КЛАСТЕРЫ. СИСТЕМА ДОМЕННЫХ ИМЕН. СОВРЕМЕННЫЕ ПОДХОДЫ К ПОСТРОЕНИЮ КЛАСТЕРНЫХ СИСТЕМ 15

1.1 Компьютерные кластеры 15
1.1.1 История компьютерных кластеров 15
1.1.2 Преимущества компьютерных кластеров 17
1.1.3 Классификация компьютерных кластеров 18

1.2 Система доменных имен 19
1.2.1 Характеристики системы доменных имен 19
1.2.2 Ресурсные записи системы доменных имен 21

1.3 Распределенные хеш-таблицы (DHT) 22
1.3.1 Основные сведения о распределенных хеш-таблицах: понятие, свойства, назначение 23
1.3.2 Консистентное хеширование 25
1.3.3 Chord DHT 29
1.3.4 Content Addressable Network (CAN) 31
1.3.5 Tapestry 34
1.3.6 Pastry 36
1.3.7 Amazon's Dynamo 38
1.3.8 DDNS 45

1.4 Выводы 47

ГЛАВА 2. МОДЕЛИ И АЛГОРИТМЫ БАЛАНСИРОВКИ НАГРУЗКИ В DNS-КЛАСТЕРЕ 49

2.1 Особенности DNS-сервиса: кэширование DNS-записей 49

2.2 Кластерная система с точки зрения теории массового обслуживания 51

2.3 Базовые подходы к балансировке нагрузки на основе распределенных хеш-таблиц 58

2.4 Одноуровневая модель организации таблиц вариантов распределения 61
2.4.1 Построение таблицы вариантов распределения 66
2.4.2 Алгоритм поиска ответственного узла для входящего домена 68
2.4.3 Проверка равномерности распределения нагрузки 70

2.5 Двухуровневая модель организации таблиц вариантов распределения 72

2.5.1 Недостатки одноуровневой модели организации таблиц вариантов распределения 72
2.5.2 Таблицы вариантов распределения 1-го и 2-го уровней 74
2.5.3 Алгоритм поиска ответственного узла с применением 76
таблиц 1-го и 2-го уровней 76

2.6 Разработанная модель балансировки нагрузки 80

2.7 Возникновение коллизий в процессе функционирования распределенной хеш-таблицы 81

2.8 Сложность алгоритмов построения таблиц вариантов распределения и поиска ответственного узла, оценка масштабируемости системы 82

2.9 «Zero-hop» маршрутизация 85

2.10 Взаимодействие DNS-клиента с узлами кластера 86

2.11 Выводы 93

ГЛАВА 3. АЛГОРИТМЫ РЕПЛИКАЦИИ DNS-ЗАПИСЕЙ В РАМКАХ КОМПЛЕКСА ПРОГРАММ БАЛАНСИРОВКИ НАГРУЗКИ 96

3.1 Репликация в вычислительной технике 96

3.2 Особенности задачи репликации DNS-записей 99

3.3 Алгоритм репликации ресурсных записей на основе взаимного перекрытия областей ответственности узлов 102

3.4 Алгоритм репликации ресурсных записей на основе использования ближайших областей ответственности узлов 105

3.5 Взаимодействие узлов кластера в процессе репликации DNS-записей 111

3.6 Временная сложность алгоритмов репликации 113

3.7 Структурная схема реализованного программного комплекса 114

3.8 Выводы 115

ГЛАВА 4. СТАТИСТИЧЕСКИЕ ИССЛЕДОВАНИЯ КОМПЛЕКСА ПРОГРАММ БАЛАНСИРОВКИ НАГРУЗКИ И ЕГО ВЕРИФИКАЦИЯ 117

4.1 Динамика статистических показателей с ростом числа уникальных запросов к системе 118

4.2 Проверка соответствия распределения входящих DNS-запросов среди узлов кластера равномерному закону 122

4.3 Выводы 125

ЗАКЛЮЧЕНИЕ 126

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 127

ПРИЛОЖЕНИЕ А 134

ПРИЛОЖЕНИЕ Б 138

ПРИЛОЖЕНИЕ В 140

ПРИЛОЖЕНИЕ Г 141

ВВЕДЕНИЕ

Актуальность темы. В настоящее время крайне актуальной задачей является разработка программных систем для новых информационных технологий, таких как программные комплексы балансировки нагрузки в компьютерных кластерных системах на основе распределенных хеш-таблиц. Разработки в области кластеризации ведутся крупнейшими университетами мира (Массачусетский технологический институт, Калифорнийский университет в Беркли, Принстонский университет и т.д.), а также ведущими мировыми производителями программного и аппаратного обеспечения, такими как Google, IBM, Microsoft, Sun Microsystems, Red Hat, Amazon, Apache и т.д. Со второй половины XX века кластеризация проявила себя как крайне перспективное направление развития компьютерных систем за счет возможности объединения вычислительных ресурсов и ресурсов хранения данных отдельных машин.

Применение кластеризации приводит к существенному повышению эффективности работы различных компьютеризированных сервисов и систем. Также использование кластеров позволяет повысить эффективность компьютерного моделирования за счет существенного снижения времени построения моделей [43].

Одной из важных областей применения кластеров являются сетевые сервисы, где кластеризация имеет ряд несомненных преимуществ по отношению к использованию отдельных высокопроизводительных машин. Прежде всего, это масштабируемость, высокая доступность сервиса (за счет использования множества узлов) и крайне выгодное соотношение стоимости оборудования и суммарной производительности.

При первоначальном построении сетевого сервиса в большинстве случаев априори неизвестна потенциальная нагрузка, которую должно будет обрабатывать программное и аппаратное обеспечение, так как заранее крайне сложно оценить число использующих сервис пользователей. Также особые трудности вносит тот факт, что количество пользователей сервиса может существенно варьироваться во

времени, причем как в большую, так и в меньшую сторону (например, волнообразное увеличение при наборе популярности сервиса или плавное уменьшение при планомерной потере интереса со стороны пользователей). Таким образом, выбор оптимальной мощности аппаратной составляющей здесь является крайне сложной задачей. Существенно упростить задачу помогает кластеризация сервиса. В этом случае такое свойство кластера, как масштабируемость, позволяет постепенно (или волнообразно) увеличивать или уменьшать аппаратные мощности сервиса, изменяя соответствующим образом количество узлов, составляющих кластер. Данный подход позволяет избежать недостатка или избыточности аппаратных мощностей, используя оптимальное соотношение аппаратного обеспечения и обрабатываемой в конкретный период времени нагрузки.

Одним из важнейших аспектов функционирования сетевых сервисов является их высокая доступность. Здесь кластер также имеет неоспоримые преимущества по сравнению с одиночными машинами за счет наличия множества равноправных узлов, которые могут принять на себя нагрузку узла, вышедшего из строя по причине сбоя программного или аппаратного обеспечения, проведения технического обслуживания и т.д.

В настоящее время сетевые сервисы являются одной из наиболее стремительно развивающихся областей информационных технологий, а, следовательно, все больше возрастает и актуальность исследования кластеризации систем.

Так как кластер состоит не менее чем из двух компьютеров (а в большинстве случаев их гораздо больше), то, следовательно, возникает необходимость в создании комплекса специального программного обеспечения, который позволит кластеру работать как единая согласованная система. Одно из центральных мест в подобном программном комплексе занимает система балансировки нагрузки на узлы кластера, необходимая для наиболее эффективного использования его объединенных вычислительных ресурсов. В зависимости от варианта реализации кластера, его назначения и обрабатываемых данных, разрабатываются различные

методы и алгоритмы балансировки нагрузки. Также система балансировки нагрузки должна учитывать особенности работы сервиса и его отличительные черты.

При наличии соответствующих механизмов программный комплекс, обеспечивающий функционирование кластера, также позволяет организовать резервное копирование каких-либо данных, полностью сохраняя работоспособность системы при временных (по причине сетевого сбоя, технического обслуживания, обновления программного обеспечения, появления различного рода проблем в работе сервера и т.д.) или постоянных выходах узлов из состава кластера. Применение подобных технологий существенно повышает надежность и стабильность кластерных систем.

Таким образом, актуальность темы диссертационных исследований обусловлена бурным ростом популярности кластерных систем, все больше показывающих свою эффективность, что позволяет говорить о кластеризации как о фундаментальной основе развития современных информационных технологий.

Представленная диссертационная работа посвящена разработке, реализации и исследованию моделей и алгоритмов, составляющих программных комплекс балансировки нагрузки и репликации в кластерных системах. Одной из основных областей применения комплекса являются кластеры, составленные из рекурсивных кэширующих DNS-серверов. В основе данной системы лежит инновационная технология распределенной хеш-таблицы (Distributed Hash Table, DHT).

Один из наиболее успешных подходов к созданию распределенных хеш-таблиц, называемый консистентным хешированием, был предложен в работах D. Karger, E. Lehman, T. Leighton et al. (Кембридж, США), а в работе I. Stoica, R. Morris, D. Karger et al. (Кембридж, США) представлена соответствующая реализация распределенной хеш-таблицы.

В рамках данной диссертации предложен оригинальный подход к созданию распределенной хеш-таблицы, который позволил реализовать систему балансировки нагрузки, соответствующую предъявляемым к ней требованиями. Разработанная схема разбиения пространства области значений используемой хеш-функции на

области ответственности узлов кластера в сочетании с принципами консистентного хеширования и концепцией виртуального представления узла позволили нивелировать неидеальность свойств хеш-функции в части равномерности распределения получаемых с ее помощью значений, а также особенности конкретного входящего DNS-трафика. Таким образом, удалось добиться требуемой равномерности распределения входящих запросов (а, следовательно, и нагрузки) среди узлов, составляющих кластер, что было подтверждено проведенными в диссертации исследованиями с использованием трафика, полученного от реальных пользователей. Применение оригинальной двухуровневой модели организации таблиц распределения позволило избежать эффекта смещения пространств ответственности узлов при временном выходе узла из состава кластера. Высокая скорость и эффективность работы системы достигнута за счет специальным образом подготовленной инфраструктуры, которая позволяет определить узел, ответственный за обработку элемента данных, с использованием нескольких арифметических операций. Все накладные расходы по перестроению внутренних структур, составляющих инфраструктуру системы, перенесены на событие изменения состава участников кластера, которое является достаточно редким по отношению к частоте поступления запросов. В ходе своего функционирования система работает только с оперативной памятью, полностью отсутствует работа с постоянным запоминающим устройством, что многократно уменьшает время поиска ответственного узла и делает это время более предсказуемым.

Также в рамках данной работы были разработаны и реализованы оригинальные алгоритмы репликации ресурсных записей как часть системы балансировки нагрузки. Данные алгоритмы позволяют существенно повысить стабильность и надежность кластерных систем. Алгоритмы определения узлов для репликации также состоят из нескольких арифметических операций и не снижают общее быстродействие системы балансировки нагрузки.

Цель и задачи исследования. Цель работы – разработка и анализ моделей и алгоритмов балансировки нагрузки в кластерной системе с поддержкой механизма репликации.

Для достижения цели в работе решались следующие задачи:

1. Разработка модели балансировки нагрузки на основе распределенной хеш-таблицы, соответствующей требованиям масштабируемости, быстродействия и равномерности распределения нагрузки.
2. Создание алгоритма поиска узла, ответственного за обработку элемента данных.
3. Создание алгоритмов репликации, повышающих стабильность, надежность и отказоустойчивость распределенной системы.
4. Разработка и анализ алгоритмов работы предложенных моделей, их реализация в форме комплекса компьютерных программ, проведение статистических исследований для верификации и тестирования программной реализации.

Объект исследования – кластерные системы.

Предмет исследования – модели и алгоритмы балансировки нагрузки и репликации в кластерной системе.

Методы исследования. При выполнении работы использовались: математические и статистические методы обработки данных, методы теории алгоритмов, теории вероятностей.

Новизна работы:

1. Предложена модель балансировки нагрузки в кластерной системе, отличительными чертами которой являются оригинальная схема разбиения области значений базовой хеш-функции, отсутствие необходимости хранения метаданных на постоянном запоминающем устройстве для обеспечения функционирования алгоритмов и гарантированное определение ответственного узла за один цикл поиска.

2. Разработан алгоритм поиска узла, ответственного за обработку элемента данных, имеющий константную временную сложность и не ограничивающий масштабируемость системы.

3. Созданы алгоритмы репликации, обеспечивающие наличие заданного числа резервных копий в зависимости от конкретных требований к надежности и отказоустойчивости системы, имеющие константную временную сложность и не ограничивающие масштабируемость системы.

4. На основе разработанной модели распределения нагрузки созданы и исследованы новые алгоритмы, отличающиеся высоким быстродействием, проведено тестирование и верификация их программной реализации.

Практическая значимость результатов работы заключается в том, что предложенные модели, алгоритмы и программы можно использовать в различных кластерных системах, где требуется обеспечить равномерное распределение нагрузки на узлы, масштабируемость в режиме реального времени без остановки функционирования системы, а также производить резервное копирование данных. Также возможно их широкое использование в области распределенных файловых систем для равномерного размещения метаданных на выделенных для данной цели узлах. За счет высокого быстродействия предложенные алгоритмы позволяют существенно повысить эффективность работы кластерных систем, а наличие алгоритмов репликации обеспечивает их надежность и отказоустойчивость.

Область исследования – содержание диссертации соответствует паспорту специальности 05.13.17 – «Теоретические основы информатики» (технические науки), область исследований соответствует п. 1 «Исследование, в том числе с помощью средств вычислительной техники, информационных процессов, информационных потребностей коллективных и индивидуальных пользователей»; п. 2 «Исследование информационных структур, разработка и анализ моделей информационных процессов и структур»; п. 14 «Разработка теоретических основ создания программных систем для новых информационных технологий».

Реализация результатов исследования. Результаты диссертации в форме комплексной системы распределения нагрузки с поддержкой механизма репликации ресурсных записей используются в проекте DNS-сервиса, реализуемом компанией «Релэкс» (г. Воронеж). В рамках данного проекта система успешно функционирует в составе ряда кластеров рекурсивных кэширующих DNS-серверов, расположенных в Европе и США.

Основные результаты, выносимые на защиту:

1. Модель балансировки нагрузки в кластерной системе, позволяющая избежать необходимости хранения метаданных на постоянном запоминающем устройстве для обеспечения функционирования алгоритмов, а также соответствующая требованиям масштабируемости, быстродействия и равномерности распределения нагрузки.
2. Алгоритм поиска узла, ответственного за обработку элемента данных, имеющий константную временную сложность и не ограничивающий масштабируемость системы.
3. Алгоритмы репликации, обеспечивающие наличие заданного числа резервных копий в зависимости от конкретных требований к надежности и отказоустойчивости системы, имеющие константную временную сложность и не ограничивающие масштабируемость системы.
4. Алгоритмы, построенные на основе разработанной модели распределения нагрузки, а также результаты их программной реализации.

Апробация работы. Основные положения диссертационной работы были представлены на XXI всероссийской научно-методической конференции «Телематика’2014», г. Санкт-Петербург, 2014 г.; XI, XII и XIV международных научно-методических конференциях «Информатика: проблемы, методология, технологии», г. Воронеж, 2011-2012, 2014 гг.; X международном семинаре «Физико-математическое моделирование систем», г. Воронеж, 2013 г. Результаты диссертационного исследования прикладного и теоретического характера нашли применение в проекте, реализуемом группой компаний «Релэкс» (г. Воронеж), внедрение результатов подтверждено соответствующим актом.

Публикации. Основные результаты диссертации опубликованы в 11 печатных изданиях, в том числе в четырех – из списка изданий, рекомендованных ВАК РФ. Получены два свидетельства о государственной регистрации программ для ЭВМ.

Личный вклад автора. Основные результаты по теме диссертации получены лично автором. Постановка задач диссертации предложена научным руководителем. Выбор базисных принципов модели балансировки нагрузки, разработка схемы разбиения области значений хеш-функции на области ответственности узлов кластерной системы, алгоритмов репликации проводились лично автором. Реализация в виде комплекса алгоритмов, информационных структур и программ, исследование, проверка достоверности результатов, получение выводов и их интерпретация выполнены автором.

Структура и объем работы. Диссертация состоит из введения, четырех глав, заключения, списка использованных источников, включающего 60 наименований научных трудов на русском и иностранных языках, и 4 приложений. Объем диссертации составляет 143 страницы, включая 126 страниц основного текста, содержащего 37 рисунков и 5 таблиц.

Содержание работы. Во введении обоснована актуальность темы диссертационной работы, сформулированы цели и задачи исследования, определены научная новизна и практическая значимость. Обсуждается проблема построения кластерных систем. Ставится задача разработки модели и алгоритмов балансировки нагрузки с учетом особенностей сервиса DNS, алгоритмов репликации ресурсных записей, а также их реализации в виде комплекса программ.

Первая глава посвящена рассмотрению подходов к построению кластерных систем балансировки нагрузки, обзору и анализу существующих реализаций распределенных хеш-таблиц. В главе раскрыто понятие компьютерного кластера, рассмотрены основные виды кластеров, описаны их свойства и особенности. Также представлены понятие и структура распределенных хеш-таблиц, их назначение и области применения. Основной целевой областью применения разработанных

моделей и алгоритмов является система доменных имен, имеющая свои свойства, особенности и ограничения, описанные в главе.

Вторая глава посвящена разработке модели балансировки нагрузки в кластерной системе с учетом особенностей системы доменных имен. Разработаны элементы модели балансировки, включающие в себя схему разбиения области значений базовой хеш-функции на области ответственности узлов кластера с учетом требований масштабируемости системы и равномерности распределения нагрузки на узлы, а также аспект хранения схемы разбиения, определяющий способ ее хранения в виде таблиц вариантов распределения. Создан алгоритм поиска ответственного за домен узла, проанализированы ситуации входа и выхода узлов из состава кластера.

Третья глава посвящена разработке и анализу алгоритмов репликации ресурсных записей на основе схемы разбиения пространства ключей на области ответственности узлов кластера, описанной во второй главе. Представлены и проанализированы два оригинальных алгоритма, первый из которых основан на взаимном перекрытии областей ответственности узлов, второй базируется на использовании некоторого количества ближайших к ответственному узлу областей ответственности.

Четвертая глава посвящена исследованию алгоритмов балансировки нагрузки. В данной главе приведены различные статистические показатели, найденные при использовании тестового набора доменов, проанализирована их динамика с ростом числа принятых системой запросов. Также для тестового набора доменов и доменов, взятых из DNS-трафика от реальных пользователей, произведена проверка качества алгоритмов в сфере равномерности распределения нагрузки.

В заключении сформулированы основные результаты работы:

1. Разработана модель балансировки нагрузки в кластерной системе, позволяющая избежать необходимости хранения метаданных на постоянном запоминающем устройстве для обеспечения функционирования алгоритмов, а

также соответствующая требованиям масштабируемости, быстродействия и равномерности распределения нагрузки.

2. Создан алгоритм поиска узла, ответственного за обработку элемента данных, имеющий константную временную сложность и не ограничивающий масштабируемость системы.

3. Созданы алгоритмы репликации, обеспечивающие наличие заданного числа резервных копий в зависимости от конкретных требований к надежности и отказоустойчивости системы, имеющие константную временную сложность и не ограничивающие масштабируемость системы.

4. На основе разработанной модели распределения нагрузки созданы и исследованы новые алгоритмы, отличающиеся высоким быстродействием, проведено тестирование и верификация их программной реализации.

Глава 1. Компьютерные кластеры. Система доменных имен. Современные подходы к построению кластерных систем

1.1 Компьютерные кластеры

Кластеризация компьютерных систем является фундаментальной основой развития современных информационных технологий. В рамках данной работы под кластером будет пониматься именно компьютерный кластер.

1.1.1 История компьютерных кластеров

История создания кластеров неразрывно связана с ранними разработками в области компьютерных сетей. Одной из причин для появления скоростной связи между компьютерами стала перспектива объединения вычислительных ресурсов отдельно стоящих машин. В начале 1970-х гг. группой разработчиков протокола TCP/IP и лабораторией Xerox PARC были закреплены стандарты сетевого взаимодействия. В университете Карнеги-Меллон появилась и операционная система Hydra ("Гидра") для компьютеров PDP-11 производства DEC, созданный на этой основе кластер был назван C.mpp (Питтсбург, штат Пенсильвания, США, 1971). Тем не менее, только около 1983 г. были созданы механизмы, позволяющие с лёгкостью пользоваться распределением задач и файлов через сеть.

Один из первых архитекторов кластерной технологии Грегори Пфистер (Gregory F. Pfister) в своей книге «In Search of Clusters» так сказал об истории создания компьютерных кластеров: «Практически каждый пресс-релиз от DEC, упоминая кластеры, говорит: «DEC, кто изобрел кластеры…». IBM также их не изобретал. Кластеры изобрели сами пользователи, так как они не могли вместить всю свою работу на одном компьютере, или же нуждались в резервной копии. Дата первого кластера неизвестна, однако было бы удивительно, если это было не в 60-х, или даже конце 50-х» [1].

Первым коммерческим проектом кластера стал ARCNet, созданный компанией Datapoint в 1977 г. ARCnet не имел коммерческого успеха, поэтому направление компьютерных кластеров фактически не развивалось до 1984 г., когда DEC построила свой VAXcluster на основе операционной системы VAX/VMS. ARCNet и VAXcluster были рассчитаны не только на совместные вычисления, но и сместное использование файловой системы и периферии с учётом сохранения целостности и однозначности данных. VAXCluster (называемый теперь VMSCluster) и сейчас можно приобрести для систем HP OpenVMS, использующих процессоры Alpha и Itanium.

История создания кластеров из обыкновенных персональных компьютеров во многом обязана проекту Parallel Virtual Machine. В 1989 г. это ПО для объединения компьютеров в виртуальный суперкомпьютер открыло возможность мгновенного создания кластеров. В результате суммарная производительность всех созданных тогда дешёвых кластеров обогнала по производительности сумму мощностей "серьёзных" коммерческих систем.

Создание кластеров на основе недорогих персональных компьютеров, объединённых сетью передачи данных, продолжилось в 1993 г. силами Американского аэрокосмического агентства (NASA), затем в 1995 г. получили развитие кластеры Beowulf, специально разработанные на основе этого принципа. Успехи таких систем подтолкнули развитие grid-сетей, которые существовали ещё с момента создания UNIX.

Таким образом, кластер – это локальная (расположенная территориально в одном месте) вычислительная система, состоящая из множества независимых компьютеров и сети, связывающей их. Кроме того, кластер является локальной системой потому, что он управляется в рамках отдельного административного домена как единая компьютерная система.

Кластер состоит из трех основных компонентов: собственно вычислителей (обработчиков) – компьютеров, образующих узлы кластера, сети, объединяющей эти узлы, и программного обеспечения, заставляющего всю конструкцию работать в стиле единого компьютера. В роли вычислительных узлов могут выступать

компьютеры различного класса: от самого обычного персонального компьютера до современного многопроцессорного сервера. Причем количество узлов в кластере практически ничем не ограничено.

1.1.2 Преимущества компьютерных кластеров

Применение кластеризации приводит к существенному повышению эффективности работы различных сервисов и систем. Одной из наиболее широких областей применения кластеров являются сетевые сервисы, где кластеризация имеет ряд несомненных преимуществ по отношению к использованию отдельных высокопроизводительных машин. Прежде всего, это масштабируемость, высокая доступность сервиса (за счет использования множества узлов) и крайне выгодное соотношение стоимости оборудования и суммарной производительности [2].

Опишем подробнее каждое из преимуществ.

Масштабируемость: кластеры хорошо подходят для нагрузок, свойственных Интернет-сервисам. Отличительной чертой подобных нагрузок является высокая параллельность (множество одновременных независимых пользователей, выполняющих аналогичные задачи). Для такого типа нагрузок производительность крупных кластеров может затмить мощность самых производительных одиночных машин . Более того, возможность постепенного расширения кластеров является огромным преимуществом в такой сфере, как Интернет-сервисы, где планирование аппаратных мощностей зависит от большого количества заранее неизвестных факторов [2]. Масштабируемость позволяет заменить планирование вычислительной мощности на ее постепенное увеличение или уменьшение в зависимости от актуальной нагрузки.

Высокая доступность: доступность означает возможность группе пользователей использовать систему. Если у них нет такой возможности, система считается недоступной. Кластеры имеют высокие показатели доступности за счет естественной избыточности числа независимых узлов: каждый из узлов имеет свой собственный процессор, диски, энергопитание и т.д. Таким образом, сокрытие отказов узлов кластера от пользователя является вопросом

программного обеспечения. Также данную особенность кластера можно использовать при обновлении программного обеспечения, где некоторое подмножество узлов может быть временно остановлено для процесса обновления. Подобная возможность является фундаментальной для сетевых сервисов, пользователи которых ожидают их постоянную и бесперебойную работу.

Соотношение стоимости оборудования и суммарной производительности: крайне выгодное соотношение стоимости и производительности достигается за счет возможности объединения в кластер достаточно бюджетных машин, суммарная вычислительная мощность которых может не уступать отдельным высокопроизводительным машинам.

Однако, несмотря на фундаментальность описанных выше преимуществ, их реализация на практике является достаточно сложной задачей.

1.1.3 Классификация компьютерных кластеров

При построении кластерных систем в зависимости от решаемой системой задачи выбирается определенный тип кластера, в соответствии с которым проектируется программное обеспечение, позволяющее кластеру работать единой, согласованной системой.

Традиционно выделяют следующие типы кластеров:

- кластеры высокой доступности (*High Availability Clusters*): создаются для обеспечения высокой доступности сервиса, предоставляемого кластером. Избыточное число узлов, входящих в кластер, гарантирует предоставление сервиса в случае отказа одного или нескольких серверов;
- кластеры распределения нагрузки (*Load Balancing Clusters*): принцип их действия строится на распределении запросов через один или несколько входных узлов, которые перенаправляют их на обработку в остальные, вычислительные узлы. Первоначальная цель такого кластера — производительность, однако, в них часто используются также и методы, повышающие надёжность;

- вычислительные кластеры (*Computing Clusters*): используются в вычислительных целях, в частности в научных исследованиях.

Конкретная технология может сочетать данные принципы в любой комбинации. Например, *Linux-HA* поддерживает режим обоюдной поглощающей конфигурации (англ. *takeover*), в котором критические запросы выполняются всеми узлами вместе, прочие же равномерно распределяются между ними.

1.2 Система доменных имен

Следующий объект обзора – система доменных имен.

Система доменных имён (Domain Name System, DNS) – компьютерная распределённая система для получения информации о доменах. Чаще всего используется для получения IP-адреса по имени хоста (компьютера или устройства), получения информации о маршрутизации почты, обслуживающих узлах для протоколов в домене (SRV-запись).

Распределённая база данных DNS поддерживается с помощью иерархии DNS-серверов, взаимодействующих по определённому протоколу.

Основой DNS является представление об иерархической структуре доменного имени и зонах [3]. Каждый сервер, отвечающий за имя, может делегировать ответственность за дальнейшую часть домена другому серверу (с административной точки зрения – другой организации или человеку), что позволяет возложить ответственность за актуальность информации на серверы различных организаций (людей), отвечающих только за «свою» часть доменного имени.

Начиная с 2010 года, в систему DNS внедряются средства проверки целостности передаваемых данных, называемые DNS Security Extensions (*DNSSEC*). Передаваемые данные не шифруются, но их достоверность проверяется криптографическими способами.

1.2.1 Характеристики системы доменных имен

DNS обладает следующими характеристиками:

- распределённость администрирования: ответственность за разные части иерархической структуры несут разные люди или организации;
- распределённость хранения информации: каждый узел сети в обязательном порядке должен хранить только те данные, которые входят в его зону ответственности и (возможно) адреса корневых DNS-серверов;
- кэширование информации: узел может хранить некоторое количество данных не из своей зоны ответственности для уменьшения нагрузки на сеть;
- иерархическая структура, в которой все узлы объединены в дерево, и каждый узел может или самостоятельно определять работу нижестоящих узлов, или делегировать (передавать) их другим узлам;
- резервирование: за хранение и обслуживание своих узлов (зон) отвечают (обычно) несколько серверов, разделённые как физически, так и логически, что обеспечивает сохранность данных и продолжение работы даже в случае сбоя одного из узлов.

DNS важна для работы Интернета, так как для соединения с узлом необходима информация о его IP-адресе, а для человека проще запоминать буквенные (обычно осмысленные) адреса, чем последовательность цифр IP-адреса. В некоторых случаях это позволяет использовать виртуальные серверы, например HTTP-серверы, различая их по имени запроса [3]. Первоначально преобразование между доменными и IP-адресами производилось с использованием специального текстового файла hosts, который составлялся централизованно и автоматически рассылался на каждую из машин в своей локальной сети. С ростом сети возникла необходимость в эффективном, автоматизированном механизме, которым и стала DNS [3, 14].

DNS была разработана Полом Мокапетрисом в 1983 году. Оригинальное описание механизмов работы содержится в RFC 882 и RFC 883. В 1987 публикация RFC 1034 и RFC 1035 изменила спецификацию DNS и отменила RFC 882 и RFC 883 как устаревшие.

Основным понятием DNS является домен. Домен (англ. domain – область) – узел в дереве имён, вместе со всеми подчинёнными ему узлами (если таковые имеются), то есть именованная ветвь или поддерево в дереве имен. Структура доменного имени отражает порядок следования узлов в иерархии; доменное имя читается слева направо от младших доменов к доменам высшего уровня (в порядке повышения значимости), корневым доменом всей системы является точка ('.'), ниже идут домены первого уровня (географические или тематические), затем – домены второго уровня, третьего и т. д. [3]. На практике точку в конце имени часто опускают, но она бывает важна в случаях разделения между относительными доменами и FQDN (англ. Fully Qualifed Domain Name, полностью определённое имя домена).

Основная функциональная единица DNS – DNS-сервер. DNS-сервер – специализированное программное обеспечение для обслуживания DNS, а также компьютер, на котором это ПО выполняется. DNS-сервер может быть ответственным за некоторые зоны и/или может перенаправлять запросы вышестоящим серверам.

1.2.2 Ресурсные записи системы доменных имен

Также очень важным понятием системы доменных имен является записи DNS. Записи DNS, или Ресурсные записи (англ. *Resource Records*, *RR*) – единицы хранения и передачи информации в DNS. Каждая ресурсная запись состоит из следующих полей [3]:

- имя (*NAME*) – доменное имя, к которому привязана или которому «принадлежит» данная ресурсная запись;
- TTL (*Time To Live*) – допустимое время хранения данной ресурсной записи в кэше неответственного DNS-сервера;
- тип (*TYPE*) ресурсной записи – определяет формат и назначение данной ресурсной записи;

- класс (*CLASS*) ресурсной записи; теоретически считается, что DNS может использоваться не только с TCP/IP, но и с другими типами сетей, код в поле класс определяет тип сети;
- длина поля данных (*RDLEN*);
- поле данных (*RDATA*), формат и содержание которого зависит от типа записи.

Критически важным аспектом для DNS-сервиса является время ответа сервера. Ведущие мировые DNS-провайдеры (Google, OpenDNS и т.д.) пытаются снизить это время за счет усовершенствования программного обеспечения DNS-серверов и применения различных техник, уменьшающих сетевые задержки, таких как anycast – метода рассылки пакетов (реализованный, в частности, в протоколе IPv6), позволяющий устройству посылать данные ближайшему из группы получателей.

1.3 Распределенные хеш-таблицы (DHT)

Изыскания в области распределенных хеш-таблиц изначально были мотивированы в частности пиринговыми системами, такими, как I2P, Napster, Gnutella, Freenet, которые использовали распределенные в Интернете ресурсы для создания одного единственного приложения. В частности, они использовали широкополосный Интернет и пространство на жестких дисках для предоставления сервиса распространения файлов.

Первые четыре реализации распределенных хеш-таблиц – CAN, Chord, Pastry и Tapestry – были выполнены в 2001-2004 гг. С тех пор изыскания в этой области велись достаточно активно с ростом популярности распределенных систем, все больше показывавших свою эффективность. Вне научных кругов технологию распределенной хеш-таблицы приняли как компонент в проектах BitTorrent и Coral Content Distribution Network.

1.3.1 Основные сведения о распределенных хеш-таблицах: понятие, свойства, назначение

Распределённая хеш-таблица (Distributed Hash Table, DHT) – это класс децентрализованных распределённых систем, которые обеспечивают поисковый сервис, похожий по принципу работы на таблицу хешей, и имеют структуру «ключ, значение», хранящиеся в DHT, а каждый участвующий узел может рациональным образом осуществлять поиск значения, ассоциированного с данным ключом. Ответственность за поддержку связи между ключом и значением распределяется между узлами, таким образом, изменение набора участников вызывает перераспределение ключей в соответствии с текущим составом системы. Это позволяет легко масштабировать DHT и постоянно отслеживать добавление/удаление узлов и отказы в их работе.

DHT характеризуется следующими свойствами:

- децентрализация: форма системы коллективных узлов без координации;
- масштабируемость: система будет одинаково эффективно функционировать при тысячах или миллионах узлов;
- отказоустойчивость: система будет одинаково надежна (в некотором смысле) с узлами постоянно подключающимися, отключающимися и выдающими ошибки.

DHT широко применяется в традиционных кластерных системах, где требуется обеспечить решение таких задач, как распределение нагрузки, целостность данных и высокая производительность (в частности гарантируя, что операции, такие как маршрутизация и хранение данных или поиск завершаются быстро).

Основным сервисом, предоставляемым распределенной хеш-таблицей, является операция поиска, которая возвращает значение, ассоциированное с некоторым ключом [4]. Типичный сценарий использования DHT следующий: клиент имеет некоторый ключ, для которого он желает получить ассоциированные данные. Этот ключ клиент предоставляет какому-либо узлу, входящему в объединенную распределенной хеш-таблицей систему, после чего узел

производит функцию поиска и возвращает данные, ассоциированные с предоставленным ключом [4].

Представление пары ключ/значение может быть произвольным. Например, ключ может быть строкой или объектом. Аналогично, значение может быть представлено строкой, числом или каким-либо бинарным представлением произвольного объекта. Фактическое представление зависит от конкретного приложения.

Важное свойство распределенных хеш-таблиц состоит в том, что они способны эффективно обрабатывать огромное количество элементов данных. Более того, количество скооперированных под управлением DHT узлов может быть очень большим и варьироваться от нескольких узлов до нескольких тысяч или (теоретически) миллионов [4]. Из-за того, что размер хранилища/памяти каждого из узлов является лимитированным, невозможно на каждом узле хранить все элементы данных локально. Поэтому, в DHT каждый узел является ответственным за определенную часть элементов данных, которые хранятся локально в памяти узла. При этом каждый узел должен иметь возможность найти элемент данных, ассоциированных с любым ключом (если такой элемент существует в данный момент в системе). Таким образом, узел должен запросить данные у ответственного узла в случае, если он их не содержит локально. Данная операция совершается с помощью DHT, которая позволяет с помощью своих внутренних структур найти ответственный за конкретный ключ узел.

Один из ключевых моментов DHT – разбиение пространства ключей. Схема разбиения пространства ключей на области ответственности распределяет принадлежность ключей среди участвующих узлов. Большинство DHT используют некоторые варианты так называемого консистентного хеширования для отображения ключей в узлы. Этот способ включает в себя функция $\delta(k_1, k_2)$, которая определяет абстрактное понятие расстояния между ключами k_1 и k_2, которое не имеет отношения к географическому расстоянию или сетевым задержкам. Каждый узел представляет собой единичный ключ, названный идентификатором (ID). Узел с ID i_x владеет всеми ключами k_m, для которых i_x

самый ближайший ID, измеряемый с помощью функции $\delta(k_m, i_x)$. Одной из самых распространенных схем является Chord DHT, которая рассматривает ключи как точки на окружности и $\delta(k_1, k_2)$ есть расстояние, которое проходится по часовой стрелке окружности от ключа k_1 к k_2. Таким образом, круг пространства ключей разделён на смежные сегменты, чьи концы являются идентификаторами узлов. Если i_1 и i_2 смежные ID, то узел с ID i_2 содержит все ключи, которые находятся между i_1 и i_2.

1.3.2 Консистентное хеширование

Ключевым моментом при реализации DHT является вопрос принципов разбиения пространства ключей хеш-функции [49, 50], т.е. определения, за какую область пространства должен отвечать каждый узел, а также концепция того, каким образом должна реагировать система DHT на изменение состава участников кластера.

Исторически первой функцией распределения нагрузки и определения принадлежности ключа конкретному узлу системы была функция модуля:

$$Id = (hash(O)) \bmod N, \tag{1.1}$$

где Id – номер узла, O – объект, для которого считается хеш-значение, N – количество функционирующих узлов кластера. Такая функция обеспечивает равномерное распределение ключей по узлам, однако проблемы возникают при переконфигурировании кластера: изменение количества узлов (т.е. изменение числа N) приводит к практически полной потере соотношения между ключами и узлами системы, что является абсолютно неприемлемым для большинства задач.

Альтернативой данному подходу является механизм консистентного хеширования (consistent hashing). Инновационная идея принципа консистентного хеширования, позволяющего избежать полного перераспределения ключей при изменении состава участвующих узлов, впервые было изложена в работе [6], которая имела одно из ключевых значений в истории развития распределенных хеш-таблиц.

Изначально данная идея была разработана сотрудниками MIT (Каргером и др.) для использования в распределенном кэшировании [6]. В дальнейшем область ее использования была существенно расширена. В работе [6] термин «консистентное хеширование» был представлен как способ распределения запросов среди изменяющегося набора веб-серверов. При этом операции добавления и удаления узлов требовали перемещения только K/N элементов данных (где K – общее количество элементов данных, N – число узлов до изменения состава участников системы) [6].

Консистентное хеширование также используется для уменьшения влияния отказов элементов системы (частичных отказов системы) в крупных веб-приложениях для обеспечения надежности кэширования [7].

Обширной областью применения консистентного хеширования являются и распределенные хеш-таблицы, которые используют данную технику для разбиения пространства ключей на области ответственности узлов, а также для осуществления эффективного поиска узла, ответственного за какой-либо ключ.

Основная идея консистентного хеширования состоит в следующем: все пространство ключей (область значений используемой хеш-функции) представляется в виде кольца, в котором пространство «соединяется» в своем максимальном и минимальном значениях. С каждым узлом ставится в соответствие идентификатор – число в пределах области значений хеш-функции. В оригинальном варианте консистентного хеширования идентификаторы получаются в результате вычисления хеш-значений, причем хеш-функция, использующаяся для этих целей, не обязательно должна быть полностью идентичной хеш-функции, с помощью которой вычисляются ключи для объектов, должны лишь совпадать размеры их областей значений [6, 7]. Таким образом, каждый узел получает некоторое положение на кольце. Ключевым моментом здесь является то, что между идентификатором узла и ключом, получающемся в результате вычисления хеш-функции для элемента данных, нет принципиальной разницы. Два соседних на кольце узла образуют область, за которую отвечает один из них (с большим или меньшим идентификатором). Каждый элемент

данных идентифицируется ключом (путем вычисления хеш-функции) и получает некоторую позицию на кольце. Далее по кольцу ищется первый встретившийся по часовой (или против часовой) стрелке идентификатор узла. Так элемент данных связывается с определенным узлом (рисунок 1.1).

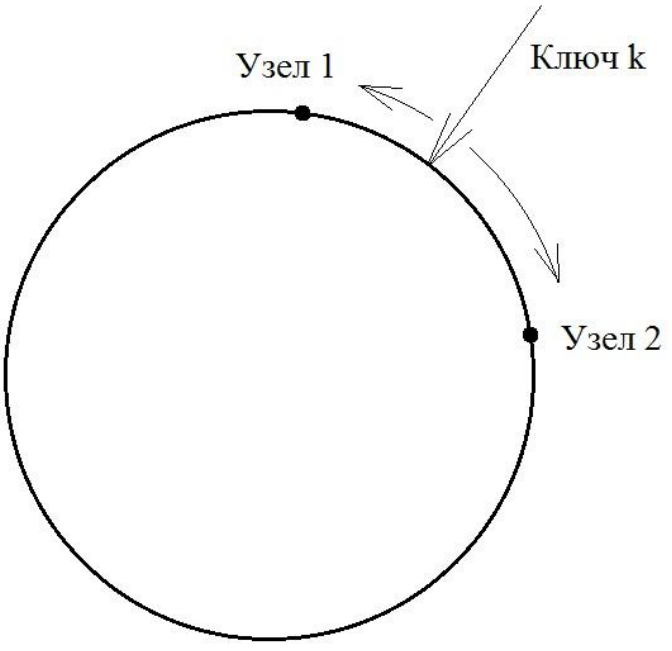

Рисунок 1.1 – Отображение элемента данных в узел системы посредством консистентного хеширования

Базовый алгоритм консистентного хеширования представляет некоторые недостатки [8]. Во-первых, случайное расположение узлов на кольце ведет к неравномерности распределения данных и нагрузки на узлы. Во-вторых, базовый алгоритм не учитывает неоднородность в производительности узлов. Для решения данных проблем используется следующий вариант консистентного хеширования: вместо отображения узла в одну единственную точку на окружности, каждый узел ассоциируется с множеством точек на кольце. Таким образом, используется концепция «виртуального узла» [8]. Виртуальный узел выглядит как один узел системы, но каждый реальный узел может быть ответственным за более чем один виртуальный. Соответственно, когда новый узел добавляется в систему, он

связывается с несколькими позициями на кольце. Подобный принцип использован в [6, 8, 9].

Наглядно данная концепция представлена на рисунке 1.2, где *узел i* является виртуальным узлом.

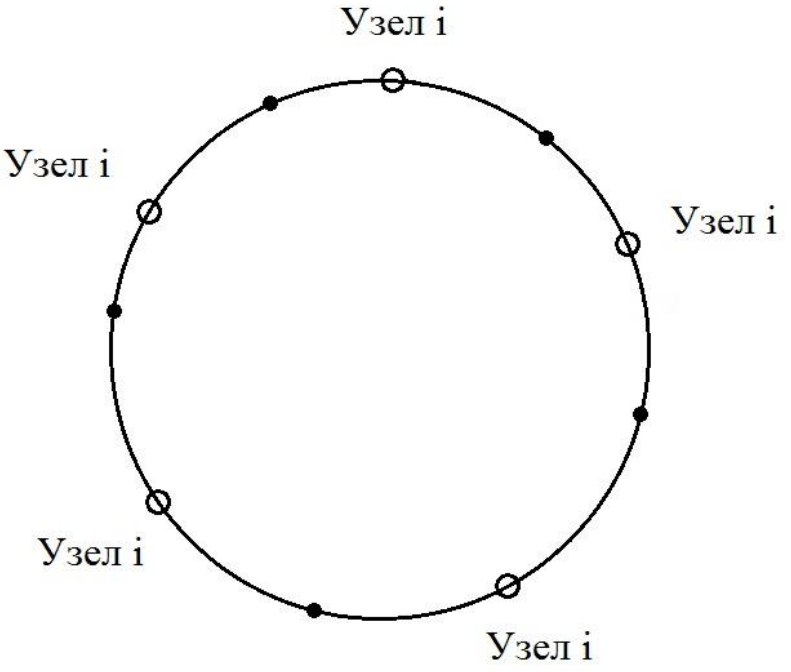

Рисунок 1.2 – Концепция виртуального представления узла

В случае, когда какой-либо узел становится недоступным, принадлежащие ему позиции на кольце перераспределяются среди оставшихся участников системы.

Аналогичный процесс происходит при добавлении узла в систему. В этом случае вошедший узел забирает себе часть позиций на кольце, получая таким образом собственные области ответственности.

Использование принципа виртуальных узлов имеет следующие преимущества:

- если узел становится недоступным (в связи с падением или плановым техническим обслуживанием), нагрузка, обрабатываемая на данном узле, равномерно перераспределяется среди оставшихся доступных узлов;

- если узел снова становится доступным, или новый узел был добавлен в систему, он принимает примерно равное количество нагрузки от каждого из других доступных узлов;

Количество виртуальных узлов, за которые узел несет ответственность, может быть определено на основе вычислительного потенциала узла, учитывая неоднородность в физической инфраструктуре.

Теперь произведем обзор существующих на данное время решений DHT. Как было сказано выше, первые четыре DHT системы были предложены приблизительно в 2001-2004 гг. Это системы CAN, Chord, Pastry и Tapestry. Остановимся на каждой из них более детально.

1.3.3 Chord DHT

Chord DHT – один из четырех оригинальных протоколов распределенных хеш-таблиц, наряду с CAN, Tapestry and Pastry, который был представлен в 2001 году. Chord DHT описан в работе [9].

Согласно протоколу Chord пространство ключей хеш-таблицы представляется в виде окружности. С каждым узлом, входящим в систему, ассоциирован идентификатор (ID), который также располагается на пространстве ключей. В таком случае максимальное количество узлов не может превышать 2^m, а окружность может содержать ключи и идентификаторы от 0 до $2^m - 1$, при этом m-битные идентификаторы и ключи получаются в результате описанного выше консистентного хеширования.

Основная функция хеширования – алгоритм SHA-1 [9]. Консистентное хеширование является неотъемлемой частью надежности и производительности Chord, так как ключи и идентификаторы узлов (соответствующие IP адресам) равномерно распределяются на пространстве ключей хеш-таблицы. Также, консистентное хеширование позволяет узлам присоединяться и покидать систему без прерывания работы системы.

В Chord каждый узел имеет потомка и предшественника. Потомком узла (или ключа) является следующий по часовой стрелке узел (ключ) на окружности

значений. Предшественник определяется в обратную сторону, т.е. против часовой стрелки. Если существует узел для каждого возможного ID, то потомком узла с ID равным 2 является узел с ID 3, а предшественником узла 1 является узел 0; однако, как правило, существуют промежутки в последовательности идентификаторов узлов. Например, потомком узла 145 может быть узел 177 (узлы с 146 по 176 не существуют в системе), в таком случае предшественником узла 177 является узел 145 [9].

Так как потомок (или предшественник) может покинуть сеть (по причине отказа или корректного выхода), каждый узел записывает весь сегмент круга, примыкающего к нему, а именно r узлов-предшественников и r узлов-потомков. Результатом является высокая вероятность того, что узел способен корректно определить потомка или предшественника, даже если сеть характеризуется высокой интенсивностью отказов.

Протокол Chord является одним из решений для соединения узлов в P2P сеть. И ключи, и узлы связаны с m-битным идентификатором. Для узла идентификатором является хеш-значение от его IP адреса, для ключа идентификатор – хэш-значение от какого-либо ключевого слова, например от имени файла. Использование слов «узлы» и «ключи» не является редкостью для обозначения этих идентификаторов, а не фактических узлов и ключей. Есть много других алгоритмов использования P2P, но это простой и довольно общий подход.

Окружность, содержащая пронумерованные позиции от 0 до $2^m - 1$, распределяется между узлами, присутствующими в сети. Ключ k связывается с узлом *потомок*(k), т.е. с тем узлом, идентификатор которого эквивалентен или следует за ключом k. Если имеется N узлов и K ключей, то каждый узел отвечает примерно за K/N ключей (при приемлемой равномерности распределения идентификаторов узлов на окружности).

В случае присоединения или отсоединения узла, изменяется ответственность за $O(K/N)$ ключей [9].

Если каждый узел знает только позицию своего потомка, то линейный поиск по сети может найти конкретный ключ, но это самый примитивный метод поиска,

так как любое сообщение потенциально может быть передано через большую часть сети. Система Chord реализует более быстрый метод поиска. Она требует от каждого узла держать таблицу, содержащую все до m записей, в этом случае i-я запись узла n будет содержать адрес узла $потомок\left(\left(n+2^{i-1}\right)\bmod 2^{m}\right)$. С такой таблицей маршрутизации количество узлов, с которыми необходимо будет связываться, чтобы найти нужный узел, для сети из N узлов является $O(\log(N))$ [9].

Наиболее распространенные варианты использования Chord:

- механизм распределения нагрузки на узлы в локальной сети при хранении или обработке на них информации, доступной извне. Данная схема позволяет разработчикам балансировать нагрузку между узлами без участия центрального сервера;
- поиск файлов по сети в поисковой базе данных, например P2P клиенты передачи файлов;
- крупномасштабный комбинаторный поиск: ключ выбирается как возможное решение, после чего происходит отображение ключа в узел сети, который в свою очередь определяет, является или нет данный ключ решением;
- хранилище с разделением временем. В сети, с включением компьютера в сеть, его доступные данные распространяются по сети с возможностью их дальнейшего извлечения по отключении компьютера из сети. Таким же образом, информация других компьютеров отправляется к рассматриваемому компьютеру для восстановления, когда они отключены от сети. В основном, описанный механизм применяется для узлов, которые не способны поддерживать постоянное подключение.

1.3.4 Content Addressable Network (CAN)

Content Addressable Network (CAN) – распределенная, децентрализованная инфраструктура P2P, которая обеспечивает функциональность хеш-таблицы в

масштабах Интернета. CAN является одним из четырех первоначальных решений распределенной хеш-таблицы, предложенной одновременно с Chord, Pastry и Tapestry. Описание CAN приведено в работе [10].

Как и другие распределенные хеш-таблицы, CAN проектировался масштабируемой, отказоустойчивой самоорганизующейся системой. Архитектурно CAN представляет собой виртуальное многомерное декартово пространство координат, которым представлена оверлейная сеть. Данное пространство координат является виртуальным логическим адресом, полностью независимым от физического местоположения и физической связанности узлов. Точки в пространстве ассоциируются с координатами. Все пространство координат динамически разделяется на все узлы системы таким образом, чтобы каждый узел обладал, по меньшей мере, одной определенной зоной в рамках общего пространства.

CAN поддерживает следующий принцип маршрутизации. Узел CAN поддерживает таблицу маршрутизации, которая содержит IP-адрес и виртуальные координаты зоны каждого из своих соседей. Все сообщения узел передает в направлении точки назначения на пространстве координат. При этом сначала узел определяет, какая соседняя зона является наиболее близкой к точке назначения, а затем через таблицу маршрутизации ищет IP адрес узла, отвечающего за эту зону [10].

Чтобы присоединиться к CAN, узел должен:

1. Найти узлы, уже присутствующие в оверлейной сети.
2. Определить зону, которая может быть разделена.
3. Обновить таблицы маршрутизации соседей в соответствии с новой зоной.

Чтобы найти уже присутствующие в оверлейной сети узлы, могут быть использованы так называемые *bootstrapping nodes* (узлы для старта) [10], которые проинформируют присоединяющийся узел об IP адресах уже включенных в сеть узлов.

После того, как новый узел был проинформирован об участвующих в сети узлах, он может сделать попытку определения зоны для себя. Присоединяющийся узел случайным образом выбирает точку на пространстве координат и посылает запрос на присоединение к сети, направленный к данной случайной точке, к одному из полученных IP адресов. Уже присутствующие в сети узлы направляют запрос на присоединение к нужному узлу через собственные *zone-to-IP* таблицы маршрутизации. Как только узел, обслуживающий зону, которая содержит точку назначения, получает запрос на присоединение, он может удовлетворить запрос, разделяя зону напополам, выделяя себе первую половину и отдавая вторую половину зоны присоединяющемуся узлу. В случае отклонения запроса, новый узел снова случайным образом выбирает точку на пространстве координат и посылает новый запрос на присоединение. Так продолжается до тех пор, пока его запрос не будет удовлетворен [10].

После того, как зона была разделена и распределена, соседние узлы должны обновить координаты двух новых зон и соответствующие IP адреса. Таблицы маршрутизации обновляются и обновления распространяются по сети.

Для корректной обработки ситуации выхода узла из состава сети, система CAN должна:

1. Определить выходящий узел.
2. Отдать высвобождающуюся зону соседнему с выходящим узлом узлу.
3. Обновить по сети таблицы маршрутизации.

Обнаружение выхода узла может быть произведено, например, через сообщения, которые периодически рассылают информацию о таблицах маршрутизации среди соседних узлов. После того, как предопределенный период «молчания» соседа истек, данный узел определяется как «отказавший» и считается вышедшим. В другом случае, при добровольном выходе узла из состава сети, он может уведомить об этом соседей путем рассылки соответствующих сообщений.

После того, как выходящий узел определен, его зона должна быть отдана или объединена. Сначала зона вышедшего узла анализируется на предмет того, с

какой из соседних зон она может слиться, чтобы сформировать корректную зону. Например, зона в двумерных координатах должна быть квадратной или прямоугольной, но не L-образной формы [10]. Тест проверки может циклично проходить по всем соседним зонам, чтобы определить, с какими зонами возможно слияние. Если ни один из потенциальных вариантов слияния не считается корректным, то соседний узел с наименьшей зоной захватывает зону. После этого, данный узел может делать периодические попытки на слияние его дополнительно контролируемых зон с соответствующими соседними зонами [10].

Если слияние прошло успешно, таблицы маршрутизации узлов соответствующих соседних зон обновляются, отражая происшедшее слияние. Сеть «увидит» данный подраздел как одну единственную зону после слияния и соответствующим образом скорректирует весь процесс маршрутизации.

При осуществлении захвата зоны, захватывающий узел обновляет таблицы маршрутизации узлов соседних зон таким образом, чтобы запросы к данной зоне разрешались в захвативший ее узел. В результате, сеть продолжает «видеть» подраздел как две отдельные зоны и, в соответствии с этим, корректирует процесс маршрутизации.

1.3.5 Tapestry

Tapestry – распределенная хеш-таблица, которая обеспечивает децентрализованное расположение объектов, маршрутизацию и инфраструктуру для распределенных приложений. Она состоит из P2P оверлейной сети, предлагающей эффективность, масштабируемость, самовосстанавливаемость и маршрутизацию к близлежащим ресурсам. Tapestry описана в работе [11].

Как было сказано выше, первое поколение P2P приложений, такие, как Napster и Gnutella, содержало недостатки, которые в первую очередь сказывались на масштабируемости. Для решения этой проблемы было создано второе поколение P2P приложений, включая Tapestry, Pastry, Chord, и CAN. В основе этих систем лежит механизм маршрутизации, основанный на ключах. Это позволяет гибко производить маршрутизацию сообщений и быструю адаптацию сети при

выходе и входе узлов. Tapestry представляет собой расширяемую инфраструктуру, которая обеспечивает децентрализованное расположение объектов, а также маршрутизацию сообщений, направленную на эффективность сетевого обмена и минимизацию задержек. Это достигается за счет того, что система Tapestry при инициализации создает локальные оптимальные таблицы маршрутизации и постоянно поддерживает их в целях уменьшения «растяжения» маршрутов [11].

Алгоритм работы Tapestry следующий. Каждый узел ассоциирован с уникальным идентификатором (*nodeID*), идентификаторы равномерно распределены в большом пространстве ключей. Tapestry использует алгоритм SHA-1 для реализации 160-битного пространства, представленного 40-разрядными шестнадцатеричными ключами [11]. *GUID* конкретных приложений (конечных точек) также ассоциированы с идентификаторами. Все *nodeID* и *GUID* примерно равномерно распределены в оверлейной сети, где каждый узел хранит несколько различных идентификаторов. Экспериментов было показано, что эффективность Tapestry увеличивается с размерами сети, т.о. совместное использование оверлейной сети несколькими приложениями повышает эффективность [11]. Для различия между приложениями также используется идентификатор.

Каждый идентификатор отображен в «живой» узел сети, который называется *root*. Пусть идентификатор *nodeID* узла равен *G*. Для того, чтобы его достичь, используются таблицы маршрутизации, содержащие идентификаторы и IP-адреса узлов. На каждом «прыжке» (*hop*) сообщение постепенно направляется все ближе к *G* с помощью так называемого incremental suffix-based routing до тех пор, пока он не будет достигнут [11].

Участники сети могут публиковать объекты с помощью периодической передачи сообщений о публикации в направлении корневого узла (*root*). Каждый узел на пути к корневому узлу хранит указатель на объект. Несколько серверов могут публиковать указатели на один и тот же объект. Между избыточными ссылками выставляется приоритет в зависимости от сетевых задержек и местоположения.

Местоположение объектов определяется через маршрутизацию сообщения к корневому узлу объекта. Каждый узел на пути к корневому проверяет его отображение и перенаправляет запрос надлежащим образом. Эффект такой маршрутизации заключается в сходимости близких путей, ведущих к одному узлу назначения [11].

Как и все остальные системы DHT, Tapestry содержит механизм обработки входа/выхода узлов. При входе в сеть новый узел становится корневым для своего *nodeID*. Далее корневой узел находит длину самого длинного префикса публикуемого им идентификатора, после чего посылает сообщение, которое достигает все существующие узлы, публикующие такой же префикс. Эти узлы добавляют новый узел в свои таблицы маршрутизации. Вошедший узел может стать корневым для некоторых из объектов, которые уже имели свои корневые узлы. Узлы сети связываются с новым узлом для передачи временного списка соседей, после чего он производит итеративный поиск ближайших соседей, чтобы заполнить все уровни своей таблицы маршрутизации.

Чтобы покинуть сеть, узел рассылает сообщения о своем выходе, после чего происходит корректировка таблиц маршрутизации. Объекты вышедшего узла перераспределяются или восстанавливаются из резервных копий.

Неожиданный отказ узла обрабатывается с помощью избыточности сети и восстановления указателей в целях исправления поврежденных ссылок.

1.3.6 Pastry

Pastry – оверлейная маршрутизирующая сетевая система, представляющая собой реализацию распределенной хеш-таблицы, аналогичную Chord. Представлена в работе [12].

В данной системе пары ключ-значение хранятся в избыточной P2P сети связанных узлов. Изначально, протокол Pastry начинает работу на узле после предоставления ему IP-адреса уже присутствующего в сети узла, после чего начинается динамическое построение или восстановление таблицы маршрутизации. Благодаря своей избыточной и децентрализованной природе,

система не имеет единой точки отказа, и любой узел в любое время может покидать сеть без предупреждения с малой вероятностью потери данных.

Данный протокол может пользоваться метриками маршрутизации, поставляемыми сторонними программами, такими как *ping* или *traceroute*, для определения наилучших маршрутов и сохранения их в таблице маршрутизации [12].

Несмотря на то, что функциональность Pastry в отношении распределенной хеш-таблицы практически идентична остальным системам DHT, отличительной чертой данной системы является маршрутизация оверлейной сети, построенная на концепции DHT. Это позволяет Pastry реализовать масштабируемость и отказоустойчивость сети при одновременном снижении общей стоимости маршрутизации пакетов с одного узла на другой, избегая необходимости «наводнения» пакетами объединяющей сети [12]. Поскольку метрики маршрутизации, поставляемые внешней программой, основаны на IP-адресе узла назначения, в качестве метрики может быть выбрано наименьшее количество «прыжков» (*hop count*), минимальная задержка, максимальная пропускная способность, или же сочетание показателей [12].

Пространство ключей хеш-таблицы представляется в виде окружности, аналогично с представлением Chord DHT, и позиция узла на окружности представлена 128-битным идентификатором (*ID*). Идентификаторы узлов выбираются на пространстве случайно и равномерно. Маршрутизация в оверлейной сети основана на хеш-таблице, где узлы обмениваются информацией, включающей список узлов-листьев, список соседей и таблицу маршрутизации. Список узлов-листьев состоит из $L/2$ ближайших по идентификатору узлов в каждом направлении по кругу [12].

Как было сказано выше, в дополнение к списку узлов-листьев также присутствует список соседей. Он представляет M наиболее близких в плане метрики маршрутизации узлов. И хотя данный список не участвует непосредственно в алгоритме маршрутизации, он используется для поддержания актуальности информации о местоположении вышеуказанных узлов.

1.3.7 Amazon's Dynamo

Особое влияние на идеи, изложенные в данной диссертации, оказала работа [8], в которой приведено подробное описание распределенного хранилища высокой доступности Dynamo от Amazon.

Dynamo – распределенная система хранения информации. Dynamo является быстрым, высоконадежным, распределенным хранилищем информации, представленной в виде пар ключ-значение. Это хранилище используется такими требовательными к быстродействию и надежности сервисами Amazon, как списки бестселлеров, корзины покупок, предпочтения пользователей, каталог продуктов и т.д. Для этих сервисов не нужны сложные реляционные модели данных. Им вполне хватает всего двух операций, предоставляемых Dynamo: *put*() для записи данных и *get*() для чтения [8].

Инфраструктура Dynamo состоит из сотен тысяч серверов, распределенных по дата-центрам с различным географическим положением. Это полностью децентрализованная система, узлы которой используют основанные на *gossip-протоколах* для установления взаимосвязей и обнаружения сбойных узлов [8].

Одно из ключевых архитектурных требований к системе Dynamo – требование масштабируемости. Это приводит к необходимости наличия механизма динамического разделения данных на множество узлов системы. Схема размещения данных Dynamo базируется на консистентном хешировании для распределения нагрузки на общее количество узлов [8]. Как было сказано выше, при консистентном хешировании область значений хеш-функции представляется в виде фиксированного кругового пространства, или кольца, в котором максимальное значение хеш-функции соединено с наименьшим. С каждым узлом системы связано случайное значение в пределах пространства, которое представляет позицию узла на кольце. Каждый элемент данных, идентифицируемый ключом, связывается с узлом, и происходит это следующим образом: для элемента данных вычисляется хеш-значение в целях определения его позиции на кольце, после чего находится первый по ходу часовой стрелки узел с позицией, большей чем позиция элемента данных. Исходя из этого, каждый узел

становится ответственным за область кольца, которая заключена между ним и его предшественником. Принципиальное преимущество консистентного хеширования состоит в том, что выход или вход узла в систему затрагивает только его ближайших соседей, остальные же узлы остаются незатронутыми [8]. В целом данный подход схож с Chord DHT.

Консистентное хеширование Dynamo использует в сочетании с описанной выше концепцией виртуального представления узла, где с каждым узлом связывается множество идентификаторов на кольце.

Dynamo рассматривает ключи и значения сохраняемых объектов как абстрактные блоки данных, структура которых для Dynamo не важна. Получив ключ объекта, Dynamo строит его MD5 хеш. Посчитанные хеш-значения используются для распределения объектов среди узлов системы Dynamo [8].

Также стоит заметить, что в целях достижения высокой доступности ресурсов и работоспособности системы, Dynamo реализует механизм репликации данных на несколько узлов. Каждый элемент данных реплицируется на N узлов, где N является конфигурируемым параметром. Каждый ключ k связан с узлом-координатором (узлом, который встречается первым по ходу часовой стрелки от ключа k). Координатор отвечает за репликацию элементов данных, попадающих в его диапазон. В добавление к локальному сохранению данных по каждому ключу в его диапазоне, координатор реплицирует данные на $N - 1$ узел-потомок, встречающийся на окружности по ходу часовой стрелки. Таким образом получается система, в которой каждый узел отвечает за область кольца между ним и его N-ым предшественником [8]. На рисунке 1.3 узел B реплицирует ключ k на узлы C и D в дополнение к локальному хранению. Узел D будет хранить ключи, попадающие в диапазоны (A, B], (B, C] и (C, D].

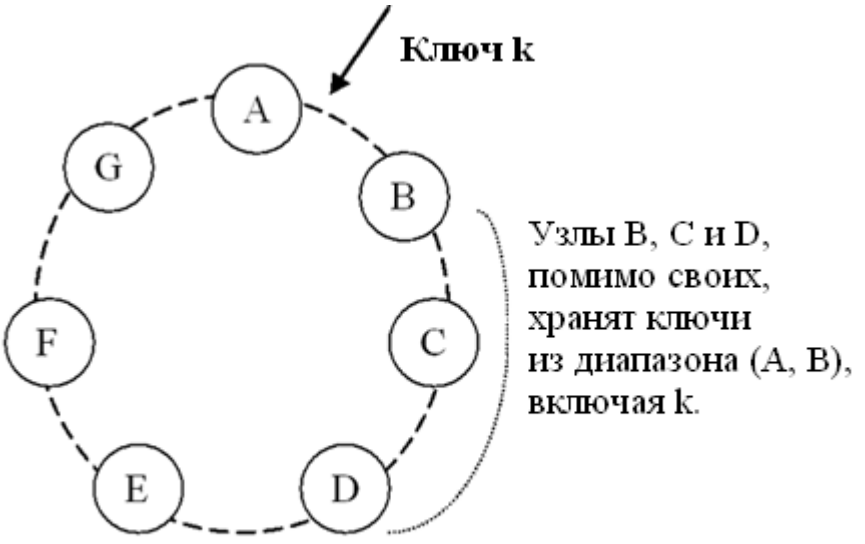

Рисунок 1.3 – Репликация ключей в системе Dynamo

Список узлов, которые являются ответственными за хранение определенного ключа, называется списком предпочтений. Система устроена таким образом, что каждый узел системы может определить, какие узлы должны быть в данном списке для определенного ключа. Для учета выхода узлов из системы, список предпочтений содержит более чем N узлов. Стоит заметить, что с использованием виртуальных узлов становится возможной ситуация, при которой первые N позиций потомков для определенного ключа могут принадлежать менее чем N физическим узлам (так как узел может содержать более чем одну из первых N позиций). Для решения этой проблемы, список предпочтений для ключей строится с пропуском определенных позиций в кольце, чтобы гарантировать то, что список содержит только различные физические узлы [8].

Схема разбиения пространства ключей в Dynamo развивалась поэтапно. В результате было реализовано три стратегии, каждая из которых имела свои показатели распределения нагрузки.

1-я стратегия: T случайных токенов на узел и разбиение по значению токенов. Это была изначальная стратегия разбиения пространства. В данной схеме каждый узел имеет T токенов (точек на пространстве значений), выбранных случайным образом. Токены всех узлов упорядочены на пространстве ключей в

соответствии со своим значением. Каждые два соседних токена определяют область (диапазон), за которую и отвечает определенный узел – владелец одного из этих токенов. Самый последний и самый первый токены формируют область, которая «оборачивает» наибольшее и наименьшее значения пространства ключей. Из-за того, что токены выбираются случайным образом, области варьируются в своих размерах. Когда узлы входят или покидают систему, набор токенов изменяется, и, соответственно, изменяются диапазоны. Стоит обратить внимание на то, что пространство, необходимое для поддержания метаданных членства в системе на каждом узле увеличивается линейно с ростом количества участвующих узлов [8].

При использовании данной стратегии возникают следующие проблемы. Во-первых, когда новый узел всходит в систему, он должен «забрать» свои диапазоны ключей у других узлов. Однако, узлы, передающие диапазоны ключей новому узлу, должны просканировать свое локальное постоянное хранилище, чтобы получить соответствующий набор элементов данных. Стоит заметить, что выполнение подобных операций сканирования является крайне ресурсоемким, и они должны производиться в фоновом режиме, не влияя на производительность. Это требует запускать задачу начальной загрузки с низким приоритетом. Однако это значительно снижает процесс загрузки, и в периоды покупательской активности, когда узлы получают миллионы запросов в день, загрузка может занять практически целый день [8].

Во-вторых, когда узлы входят или покидают систему, диапазоны ключей, обрабатываемые узлами, изменяются, и хеш-деревья (Merkle trees) для новых диапазонов должны быть пересчитаны, что является далеко не самой тривиальной операцией для выполнения на работающей системе [8]. Хеш-деревья являются частью системы репликации в Dynamo. Целью данной системы является снижение расходов на синхронизацию реплик после ввода отказавшего узла в строй. Поскольку каждый узел в Dynamo хранит большое количество объектов, то нужно быстро выбрать те из них, значение которых изменилось с момента последней репликации. Для этого в Dynamo используется механизм на основе хеш-деревьев.

Смысл хеш-дерева состоит в том, что значение родителя является хешем от значений его потомков. Т.е., если изменяется значение самого нижнего узла дерева (объекта), то изменяются хеши во всех родителях этого узла и, в конце концов, в вершине дерева. Поскольку дерево состоит из нескольких независимых друг от друга поддеревьев, то это позволяет распараллеливать сравнение двух хеш-деревьев, так как их поддеревья можно сравнивать независимо друг от друга. Например, пусть есть два хеш-дерева, у вершин которых присутствует два потомка. Если значение в вершине двух деревьев совпадают – значит, деревья одинаковые (синхронизации реплик не требуется). Если не совпадает, значит можно отдельно сравнить левые поддеревья и правые поддеревья и т.д. Этот принцип и используется в Dynamo: каждый узел хранит хеш-дерево для всех своих ключей [8]. Когда приходит время синхронизации, узлы обмениваются сначала вершинами своих хеш-деревьев. Затем, при необходимости, вершинами поддеревьев и т.д. Такой механизм работает успешно, однако здесь как раз и возникает вопрос эффективности системы распределения ключей по узлам, поскольку в худших случаях при перераспределении диапазонов (например, при вводе в строй новых узлов) перестройка хеш-деревьев оказывалась слишком дорогостоящей операцией [8].

Наконец, не существует простого пути сделать «снимок» всего пространства ключей из-за случайности диапазонов, и это делает сложным процесс архивирования. В данной схеме архивирование всего пространства ключей требует получения ключей от каждого узла в отдельности, что является крайне неэффективным.

Главный вывод данной стратегии состоит в том, что схемы распределения данных и размещения данных взаимосвязаны. Например, в некоторых случаях более предпочтительным является добавление все большего количества узлов в систему для обработки увеличивающейся нагрузки запросов. Однако при этом сценарии невозможно добавлять в систему узлы без затрагивания распределения данных. В идеале, желательно использовать независимые схемы для распределения и размещения. Для этого были определены следующие стратегии.

2-я стратегия: T токенов, выданных каждому узлу случайным образом, и разбиение пространства ключей на равные диапазоны. Согласно данной схеме все пространство ключей разбивается на Q равных частей (диапазонов) и каждый узел связан с T случайными токенами [8]. Число диапазонов Q обычно выбирается таким образом, чтобы выполнялось условие $Q >> N$ и $Q >> S*T$, где S – количество узлов в системе. Согласно данной стратегии, токены используются только для того, чтобы построить функцию, которая занимается отображением значений из пространства ключей в упорядоченный список узлов, и не участвуют в разбиении пространства. Ключи, попадающие в определенный раздел, располагаются на первых N неповторяющихся узлах, токены которых встречаются по ходу часовой стрелки на кольце пространства, начиная с конца раздела. Ниже рисунок 1.4 иллюстрирует данную стратегию для количества узлов в системе $N = 3$. На приведенном примере узлы A, B, C встречаются по часовой стрелке от конца раздела, который содержит ключ K. Основными преимуществами данной стратегии являются отделение распределения от расположения ключей по узлам и возможность изменения схемы расположения непосредственно во время функционирования системы [8].

3-я стратегия: Q/S токенов на узел и разбиение пространства ключей на равные диапазоны [8]. По аналогии со 2-ой стратегией, все пространство ключей разбивается на Q равных диапазонов, и принцип расположения ключей отделен от схемы разбиения. Каждый узел имеет Q/S токенов, где S – количество узлов в системе. Когда узел покидает систему, его токены случайным образом распределяются на оставшиеся узлы, т.о. свойства системы полностью сохраняются. Аналогичным образом, когда узел входит в систему, он забирает токены от узлов системы, и в этом случае тоже полностью сохраняются свойства системы [8].

Рисунок 1.4 графически представляет стратегии разбиения пространства в системе Dynamo.

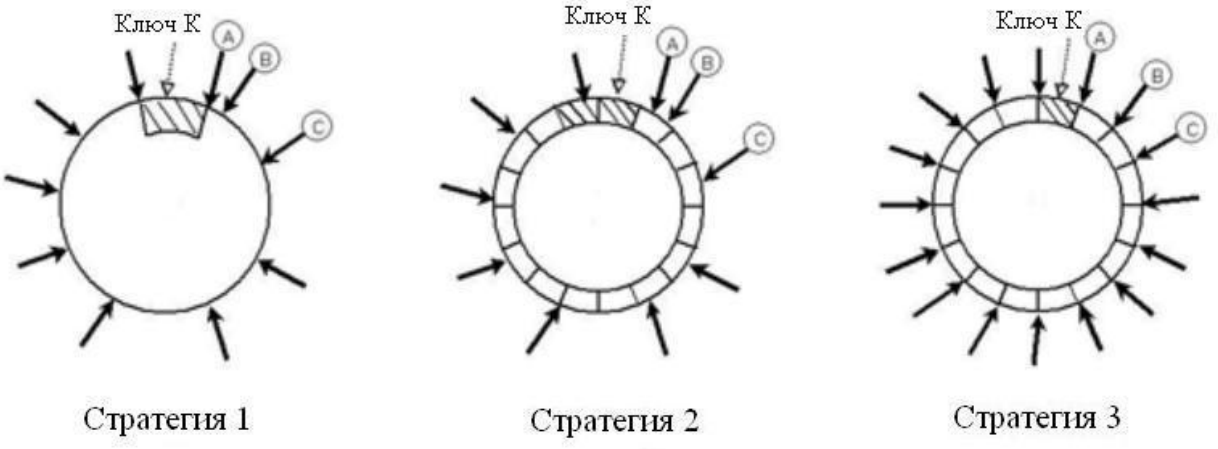

Рисунок 1.4 – Стратегии разбиения пространства ключей в системе Dynamo

Эффективность трех вышеперечисленных стратегий рассчитана для системы с количеством узлов в системе $S = 30$ и параметром репликации $N = 3$. Данные значения размера системы и количества реплик является типичным для большинства сервисов Amazon. Однако, сравнение этих различных схем справедливым образом довольно проблематично ввиду того, что различные стратегии имеют различные настройки конфигурации для своей наиболее эффективной работы. Например, свойство распределения для 1-ой стратегии зависит от количества токенов T, в то время как 3-я стратегия зависит от количества отрезков Q. Один из путей для сравнения состоит в том, чтобы оценить неравномерность в распределении нагрузки, в то время как во всех стратегиях используется одинаковый размер пространства для поддержания своего членства в системе. Например, в стратегии 1 каждый узел должен поддерживать позиции токенов для всех узлов на кольце, а в стратегии 3 каждый узел должен поддерживать информацию, относящуюся к разделам (диапазонам), связанным с каждым узлом [8].

В приведенном ниже эксперименте данные стратегии были оценены при варьировании соответствующих параметров T и Q. Эффективность балансировки нагрузки для каждой стратегии была определена для различных размеров информации о членстве в системе, которая требует поддержки на каждом узле, где

эффективность распределения нагрузки определяется как отношение среднего числа запросов, обслуживаемых каждым узлом, к максимальному количеству запросов, принятых самым нагруженным узлом.

Результаты теста приведены на рисунке 1.5. Как видно на рисунке, 3-я стратегия достигает наилучших результатов балансировки нагрузки, в то время как стратегия 2 имеет в этом смысле наихудшие показатели. В течение короткого времени, стратегия 2 служила как временная установка в процессе перехода системы Dynamo от использования стратегии 1 к использованию стратегии 3. По сравнению с 1-ой стратегией, стратегия 3 достигает лучшей эффективности при уменьшении размера информации о членстве, поддерживаемой каждым узлом, на три порядка в относительных величинах [8].

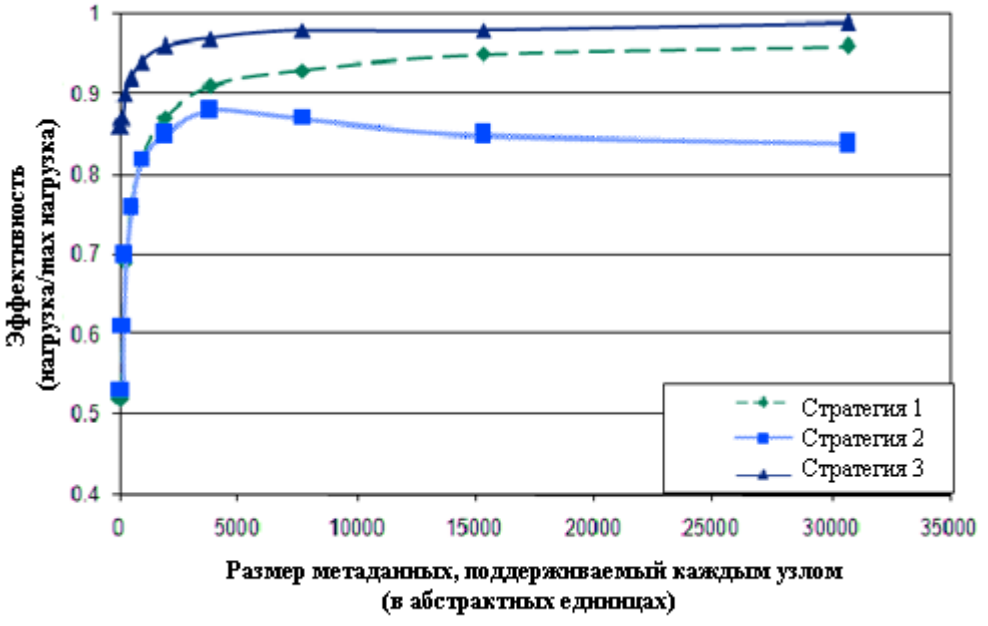

Рисунок 1.5 – Сравнение эффективности распределения нагрузки различных стратегий для системы из 30 узлов и параметра репликации $N = 3$
с одинаковым размером метаданных, поддерживаемым каждым узлом

1.3.8 DDNS

Что касается применения систем, основанных на технологии распределенной хеш-таблицы, в сфере DNS, то существует работа R. Cox, A. Muthitacharoen, R. T. Morris [13], описывающая реализацию сервиса поиска ресурсных записей на основе Chord DHT.

Данный исследовательский проект являлся попыткой реализации структуры DNS-сервиса, которая могла бы стать альтернативой текущей конвенциальной системе DNS.

Обслуживание данных DNS с помощью Chord позволяет существенно упростить процесс конфигурирования DNS-серверов [13]. Также Chord предоставляет механизм балансировки нагрузки на серверы и естественную устойчивость к DoS-атакам за счет множества участвующих узлов.

Реализованный прототип был назван DDNS.

Гранулярность операций с данными в DDNS равна набору ресурсных записей (*RRSet*) как и в классической системе DNS [13], где *RRSet* является списком ресурсных записей, соответствующих конкретному домену и типу запроса. DDNS хранит и получает наборы ресурсных записей, используя DHash [15] – основанную на Chord распределенную хеш-таблицу. DHash использует консистентное хеширование для связывания ключей с ответственными узлами, предоставляя таким образом механизм балансировки нагрузки. Также DHash обеспечивает надежность системы за счет реплицирования наборов ресурсных записей. Типичное число резервных копий – шесть.

Однако проект DDNS оказался нежизнеспособным. Данное решение имеет существенные недостатки по сравнению с текущей архитектурой DNS. Главный из них – большое время ответа системы. Эксперименты, проведенные авторами проекта DDNS, показали, что среднее время ответа для 1000 участвующих узлов порядка 350 мс, в то время как в текущей архитектуре DNS оно оценивается примерно в 43 мс [15]. Причем с ростом числа узлов DDNS среднее время ответа увеличивается.

Причина этому – количество удаленных вызовов и пересылок по сети. В процессе поиска нужного узла Chord пересылает пакеты от узла к узлу в соответствии со своими правилами маршрутизации, что крайне отрицательно влияет на время ответа системы. Соответственно, с ростом числа участников сети Chord это время увеличивается с ростом числа пересылок. Однако для сервиса

DNS время ответа является одним из наиболее важных параметров, что делает проект DDNS неприемлемым в текущем варианте.

1.4 Выводы

В настоящее время кластеризация является одним их основных направлений развития информационных технологий. Применение кластеризации приводит к существенному повышению эффективности работы различных сервисов и систем.

Одной из наиболее широких областей применения кластеров являются сетевые сервисы.

Компьютерный кластер имеет ряд неоспоримых преимуществ по сравнению с одиночными высокопроизводительными машинами, прежде всего это:

- масштабируемость;
- высокая доступность сервиса (за счет использования множества узлов);
- выгодное соотношение стоимости оборудования и суммарной производительности.

Кластер состоит из множества узлов, в связи с чем возникает проблема распределения нагрузки для обеспечения эффективности его функционирования.

Практика показывает, что наиболее эффективные решения при реализации систем балансировки нагрузки основаны на использовании технологии распределенной хеш-таблицы (*distributed hash table, DHT*), которая позволяет достичь вышеперечисленных преимуществ кластеризации.

Однако применение подобных технологий в сфере DNS имеет определенные сложности из-за особенностей DNS-сервиса. Одним из ключевых параметров DNS-провайдера является скорость ответа на запрос клиента. Ведущие мировые провайдеры DNS-сервиса, такие как Google, OpenDNS и др., стараются максимально снизить это время для привлечения как можно большей клиентской аудитории. Негативный опыт применения системы Chord [9] в исследовательском проекте DDNS [13] показывает необходимость применения иных подходов к построению распределенных и кластерных систем, предоставляющих DNS-сервис.

Таким образом, необходимы методы и алгоритмы балансировки нагрузки, имеющие минимальное негативное влияние на время ответа системы на входящий клиентский DNS-запрос.

Глава 2. Модели и алгоритмы балансировки нагрузки в DNS-кластере

2.1 Особенности DNS-сервиса: кэширование DNS-записей

В рамках данной работы будут рассматриваться исключительно рекурсивные кэширующие DNS-серверы. Обслуживание подобного типа серверов налагает дополнительные требования к системе балансировки нагрузки.

Кэширование является одним из жизненно важных элементов архитектуры DNS, так как оно позволяет значительно уменьшить число запросов, направляемых корневым серверам имен и серверам доменов верхнего уровня, которые, находясь на вершине дерева DNS, с большой вероятностью могут стать «узким местом» всей системы.

Как было сказано выше, абсолютно все DNS-провайдеры стремятся снизить время ожидания ответа клиентом. При этом помимо особенностей сети, серверного оборудования и внутренней логики на уровне программного обеспечения, большое значение здесь имеет содержание и размер кэша ресурсных записей сервера. Объединение нескольких DNS серверов в единый кластер позволяет получить распределенный кэш, при этом его общий размер является суммой размеров кэшей на каждом отдельном сервере. Распределенный кэш является одним из главных преимуществ, которое дает кластеризация в данном случае.

Кэширование является одним из основных способов сокращения времени ответа DNS-сервера, из-за чего кэшированию на стороне рекурсивного сервера уделяется особое внимание. В случае получения сервером рекурсивного запроса от клиента и при этом запрашиваемая запись не содержится в кэше, рекурсивный сервер опрашивает авторитативные серверы в порядке убывания уровня зон в имени, пока не найдёт ответ или не обнаружит, что домен не существует. Данная процедура является ресурсоемкой для сервера и может занимать существенное время (с учетом сетевых задержек).

Если запрашиваемая запись содержится в кэше и ее время жизни (TTL) не истекло, сервер незамедлительно отдает ее клиенту [3, 14], не производя дополнительных запросов к авторитативным серверам. В этом случае время ответа существенно сокращается по сравнению с опросом авторитативных серверов в поисках необходимой записи. Следовательно, если сервер будет получать только определенные домены, то он будет максимально эффективно использовать свой кэш, что положительно скажется на его производительности. Таким образом, система распределения нагрузки должна не просто равномерно распределять запросы среди узлов кластера, но, в то же время, ретранслировать запросы с определенными доменами определенным узлам кластера [16].

Распределенный кэш требует особого механизма обслуживания и поддержки для эффективного использования его содержимого. В особенности это касается масштабируемости кластера, ситуаций входа и выхода узлов. Кластер является динамической системой, в которой состав участников может со временем изменяться. При этом можно рассмотреть следующие основные ситуации и соответствующие требования к поведению системы распределения нагрузки:

1) ввод нового узла в состав кластера: данный узел должен в равных пропорциях забрать себе часть нагрузки с других узлов системы, при этом должна сохраниться общая равномерность распределения нагрузки на узлы;
2) временный выход узла из состава кластера: обслуживаемая им часть запросов должна равномерно перераспределиться между оставшимися участниками кластера. При этом запросы, обслуживаемые оставшимися узлами, не должны перераспределиться для сохранения актуальности кэшей;
3) вход узла после временного выхода: узел должен «вернуть» себе именно свою часть запросов, которую он обслуживал до выхода;
4) окончательный (безвозвратный) выход узла из состава кластера: так же, как и при временном выходе, обслуживаемая им часть запросов должна равномерно перераспределиться между оставшимися участниками

кластера, однако в этом случае возможно частичное перераспределение ключей.

В случае если алгоритм балансировки нагрузки обеспечивает корректную обработку перечисленных выше ситуаций, составляющие кластер DNS-серверы будут максимально эффективно использовать свои локальные кэши, что положительно повлияет на производительность DNS-кластера в целом.

2.2 Кластерная система с точки зрения теории массового обслуживания

Фактически, кластерная система, обрабатывающая входящие запросы (в частности, DNS-кластер), представляет собой систему массового обслуживания (СМО), так как подобная система предназначена для многоразового использования при решении однотипных задач. Возникающие при этом процессы получили название процессов обслуживания [42].

Поток событий характеризуется интенсивностью λ – частотой появления событий, поток обслуживания также характеризуется интенсивностью и имеет обозначение μ.

Каждая СМО состоит из определенного числа обслуживающих единиц, называемых каналами обслуживания. По числу каналов СМО подразделяют на одноканальные и многоканальные. Так как кластерная система включает в себя множество равноправных физических обрабатывающих узлов, то подобную систему можно охарактеризовать как многоканальная СМО.

В зависимости от своих особенностей конкретная кластерная система может быть представлена как многоканальная СМО с неограниченной очередью и многоканальная СМО с ограниченной очередью. В случае многоканальной СМО с неограниченной очередью и n каналами система может находиться в одном из состояний S_0, S_1, S_2, …, S_k, …, S_n, …, нумеруемых по числу заявок, находящихся в СМО: S_0 – в системе нет заявок (все каналы свободны); S_1 – занят один канал, остальные свободны; S_2 – заняты два канала, остальные свободны; …, S_k – занято k каналов, остальные свободны; …, S_n – заняты все n каналов, очереди нет; S_{n+1} –

заняты все n каналов, в очереди одна заявка; …, S_{n+r} – заняты все n каналов, в очереди r заявок.

Состояния системы можно представить в виде линейного графа (рисунок 2.1). В случае многоканальной СМО интенсивность потока обслуживаний не остается постоянной, а по мере увеличения числа заявок в СМО от 0 до n увеличивается от величины μ до величины $n\mu$, так как соответственно увеличивается число каналов обслуживания. При числе заявок в СМО большем, чем число каналов обслуживания n, интенсивность потока обслуживаний сохраняется равной $n\mu$ [42].

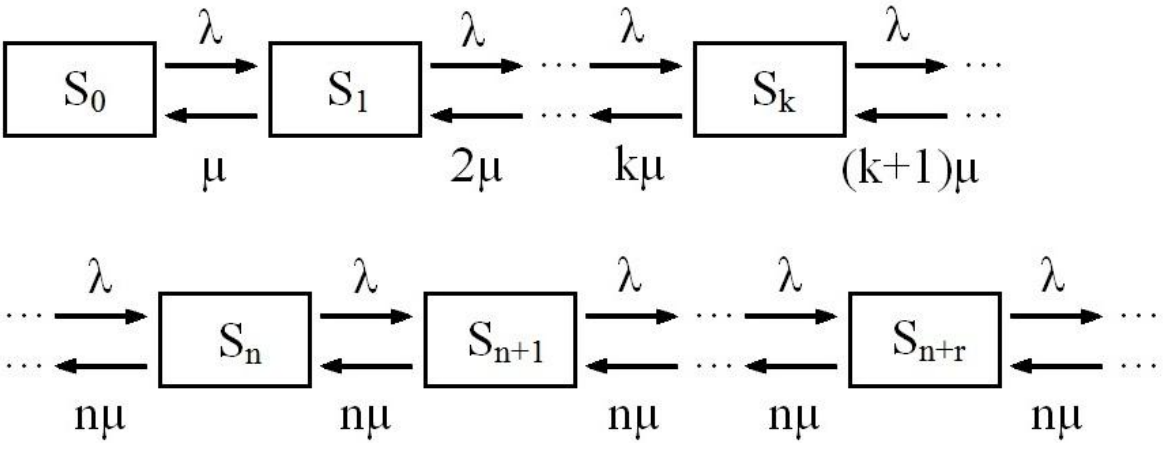

Рисунок 2.1 – Граф состояний многоканальной СМО с неограниченной очередью

Интенсивность потока заявок или интенсивность нагрузки канала выражается как:

$$\rho = \frac{\lambda}{\mu}. \tag{2.1}$$

Вероятностью i-го состояния называется вероятность $p_i(t)$ того, что в момент t система будет находиться в состоянии S_i. Для любого момента времени t сумма вероятностей всех состояний равна единице:

$$\sum_{i=0}^{n} p_i(t) = 1. \tag{2.2}$$

Особый интерес представляют вероятности системы $p_i(t)$ в предельном стационарном режиме, т.е. при $t \to \infty$, которые называются предельными вероятностями состояний.

Предельная вероятность S_i имеет четкий смысл: она показывает среднее относительное время пребывания системы в этом состоянии [42]. Например, если предельная вероятность состояния S_0 выражается как $p_0=0.5$, то это означает, что в среднем половину времени система находится в состоянии S_0.

При $\rho/n < 1$ предельные вероятности существуют, при $\rho/n \geq 1$ очередь растет до бесконечности. Формулы предельных вероятностей для n-канальной СМО с неограниченной очередью:

$$p_0 = \left(1 + \frac{\rho}{1!} + \frac{\rho^2}{2!} + \ldots + \frac{\rho^n}{n!} + \frac{\rho^{n+1}}{n!(n-\rho)}\right)^{-1},$$

$$p_1 = \frac{\rho}{1!} p_0, \ \ldots, \ p_k = \frac{\rho^k}{k!} p_0, \ \ldots, \ p_n = \frac{\rho^n}{n!} p_0,$$

$$p_{n+1} = \frac{\rho^{n+1}}{n \cdot n!} p_0, \ \ldots, \ p_{n+r} = \frac{\rho^{n+r}}{n^r \cdot n!} p_0.$$

Вероятность того, что заявка окажется в очереди:

$$p_{оч} = \frac{\rho^{n+1}}{n!(n-\rho)} p_0. \qquad (2.3)$$

Также можно выразить число занятых каналов:

$$\bar{k} = \frac{\lambda}{\mu} = \rho, \qquad (2.4)$$

среднее число заявок в очереди:

$$L_{оч} = \frac{\rho^{n+1} p_0}{n \cdot n! \left(1 - \frac{\rho}{n}\right)^2}, \qquad (2.5)$$

среднее число заявок в системе:

$$L_{сист} = L_{оч} + \rho, \qquad (2.6)$$

а также среднее время пребывания заявки в системе и в очереди:

$$T_{сист} = \frac{1}{\lambda} L_{сист}, \tag{2.7}$$

$$T_{оч} = \frac{1}{\lambda} L_{оч}. \tag{2.8}$$

Вышеперечисленные формулы справедливы для многоканальной СМО с неограниченной очередью. СМО с ограниченной очередью отличаются тем, что число заявок в очереди ограничено (не может превосходить некоторого заданного числа m). Если новая заявка поступает в момент, когда все места в очереди заняты, она покидает СМО необслуженной, т.е. получает отказ. В данном случае для вычисления предельных вероятностей состояний и показателей подобных СМО необходимо суммировать не бесконечную прогрессию, а конечную [42].

Таким образом, предельные вероятности в данном случае будут описываться следующими соотношениями:

$$p_0 = \left(1 + \frac{\rho}{1!} + \ldots + \frac{\rho^n}{n!} + \ldots + \frac{\rho^{n+1}\left(1 - \left(\frac{\rho}{n}\right)^m\right)}{n \cdot n!\left(1 - \frac{\rho}{n}\right)} \right)^{-1},$$

$$p_1 = \frac{\rho}{1!} p_0, \ldots, \; p_k = \frac{\rho^k}{k!} p_0, \ldots, \; p_n = \frac{\rho^n}{n!} p_0,$$

$$p_{n+1} = \frac{\rho^{n+1}}{n \cdot n!} p_0, \ldots, \; p_{n+r} = \frac{\rho^{n+r}}{n^r \cdot n!} p_0 \; (r = 1, \ldots, m).$$

Вероятность отказа (т.е. вероятность отклонения заявки в случае занятости всех очередей):

$$p_{отк} = p_{n+m} = \frac{\rho^{n+m}}{n^m \cdot n!} p_0. \tag{2.9}$$

В данном случае относительная и абсолютная пропускная способность соответственно:

$$Q = 1 - p_{отк} = 1 - \frac{\rho^{n+m}}{n^m \cdot n!} p_0, \tag{2.10}$$

$$A = \lambda Q = \lambda(1 - \frac{\rho^{n+m}}{n^m \cdot n!} p_0), \tag{2.11}$$

среднее число заявок в очереди:

$$L_{оч} = \frac{\rho^{n+1} p_0 \left[1 - \left(m + 1 - m\frac{\rho}{n} \right) \left(\frac{\rho}{n} \right)^m \right]}{n \cdot n! \left(1 - \frac{\rho}{n} \right)^2}, \tag{2.12}$$

среднее число занятых каналов:

$$\bar{k} = \rho \left(1 - \frac{\rho^{n+m}}{n^m \cdot n!} p_0 \right), \tag{2.13}$$

среднее число заявок в системе:

$$L_{сист} = L_{оч} + \bar{k}. \tag{2.14}$$

В зависимости от конкретной системы и от поставленной задачи выбирается та или иная модель многоканальной СМО.

Однако кластерная система с балансировкой нагрузки, в которой ключевым моментом является более или менее жесткая связь между определенным признаком заявки и конкретным обрабатывающим устройством (DNS-кластеры, распределенные файловые системы и т.д.), не является классической многоканальной СМО.

Рассмотрим менее жесткую связь между заявкой с определенным признаком и обрабатывающим устройством (например, DNS-кластер). Т.е. речь идет о многоканальной системе массового обслуживания, в которой при попадании заявки на невыделенное для нее устройство вместо ее отклонения происходит существенное увеличение времени ее обработки (DNS-запрос домена, ресурсные записи которого не содержатся в кэше сервера). Предположим, что каждая заявка с одинаковой вероятностью может попасть на любое из обрабатывающих устройств, т.е. в данном случае имеет место дискретное равномерное распределение случайной величины:

$$P(X = x_i) = \frac{1}{n},\ i = 1,...,n.$$

Математическое ожидание при дискретном равномерном распределении:

$$M[X] = \frac{1}{n}\sum_{i=1}^{n} x_i\ .$$

Типичным примером выбора обрабатывающих устройств с равномерным дискретным распределением без учета связи между заявками и обрабатывающими устройствами является алгоритм распределения нагрузки, имеющий название *Round-robin.* Распределение нагрузки здесь происходит методом перебора и упорядочения обрабатывающих устройств системы по круговому циклу. Алгоритм состоит в следующем: пусть имеется *N* обрабатывающих устройств и *M* задач с равным приоритетом. В этом случае 1-я задача $(m = 1)$ назначается для выполнения 1-му устройству, 2-я – 2-му и т.д., до достижения последнего устройства $(m = N)$. Тогда следующая задача $(m = N + 1)$ будет назначена 1-му устройству и так далее по круговому циклу.

При этом пусть для каждой заявки единовременно существует только одно выделенное устройство. Таким образом, среднее время обслуживания заявки (математическое ожидание времени обслуживания при равновероятном попадании заявки на любое из обрабатывающих устройств) будет выражаться следующим соотношением:

$$\bar{t} = \frac{n-1}{n}\frac{1}{\mu_2} + \frac{1}{n}\frac{1}{\mu_1} = \frac{n\mu_1 - \mu_1 + \mu_2}{n\mu_1\mu_2}, \qquad (2.15)$$

где n – количество обрабатывающих устройств, μ_1 – интенсивность потока обслуживания на выделенном для заявки устройстве, μ_2 – интенсивность потока обслуживания на устройстве, которое не является выделенным для данной заявки.

Время обслуживания заявки на выделенном для нее устройстве:

$$t_1 = \frac{1}{\mu_1}. \qquad (2.16)$$

Теперь имеется возможность найти отношение среднего времени обслуживания заявки к времени обслуживания заявки на выделенном устройстве:

$$\frac{\bar{t}}{t_1} = \frac{n\mu_1 - \mu_1 + \mu_2}{n\mu_1\mu_2} \div \frac{1}{\mu_1} = \frac{(n\mu_1 - \mu_1 + \mu_2)\mu_1}{n\mu_1\mu_2} = \frac{n\mu_1 - \mu_1 + \mu_2}{n\mu_2}. \qquad (2.17)$$

Предположим, что многоканальная СМО представляет собой DNS-кластер, состоящий из 5-ти узлов $(n=5)$. Для типичного DNS-сервера время обработки DNS-запроса с доменом, ресурсные записи которого находятся в локальном кэше, составляет порядка 1мс $(t_1=1)$. Анализ показывает, что в случае отсутствия записей в кэше обработка DNS-запроса в среднем составляет около 150мс $(t_2=150)$. Соответствующие значения интенсивности потоков обработки: $\mu_1 = \frac{1}{t_1} = 1$, $\mu_2 = \frac{1}{t_2} = \frac{1}{150}$. Подставляя полученные значения в формулу (2.17), находим выигрыш во времени обработки запросов на выделенном устройстве по отношению к случайному выбору обрабатывающего устройства:

$$\frac{\bar{t}}{t_1} = \frac{1\cdot 5 - 1 + \frac{1}{150}}{5\cdot\frac{1}{150}} = \frac{4 + \frac{1}{150}}{\frac{5}{150}} = \frac{600\cdot 150}{150\cdot 5} = 120.$$

Таким образом, при 5-ти узлах кластера и времени обработки запроса с участием и без участия кэша соответственно 1мс и 150мс, в среднем время обслуживания запросов системой сокращается в 120 раз при их обработке на выделенных устройствах по сравнению со случайым выбором обрабатывающих устройств. Данный факт иллюстрирует необходимость создания моделей и алгоритмов балансировки нагрузки в многоканальной СМО, которые позволят обеспечить связь между заявкой и обрабатывающим устройством по определенному признаку заявки с учетом масштабируемости системы.

Существуют также классы многоканальных СМО, где связь между конкретными заявками и соответствующими обрабатывающими устройствами является более жесткой, и ее потеря приводит к отказам в обслуживании заявок, а также к неработоспособности СМО. Примером подобных систем являются распределенные файловые системы.

2.3 Базовые подходы к балансировке нагрузки на основе распределенных хеш-таблиц

Наиболее эффективным решением для построения системы распределения нагрузки является система, основанная на принципе распределенной хеш-таблицы.

Одной из основных задач при реализации распределенной хеш-таблицы является выбор принципа разбиения пространства ключей на области ответственности узлов системы. Очевидно, что содержание кэшей DNS-серверов, объединенных в кластер, напрямую зависит от того, за какие области пространства ключей отвечают участвующие серверы.

В ходе разработки методов и алгоритмов балансировки нагрузки был проведен детальный анализ возможных вариантов разбиения области значений используемой хеш-функции.

Самый простой способ деления пространства ключей состоит в равномерном его разбиении на количество узлов и каждому узлу назначить собственный участок ответственности. Данный подход, реализующий концепцию реального представления узла, представлен на рисунке 2.2 (где 0, 1, 2 и т.д. являются идентификаторами узлов кластера).

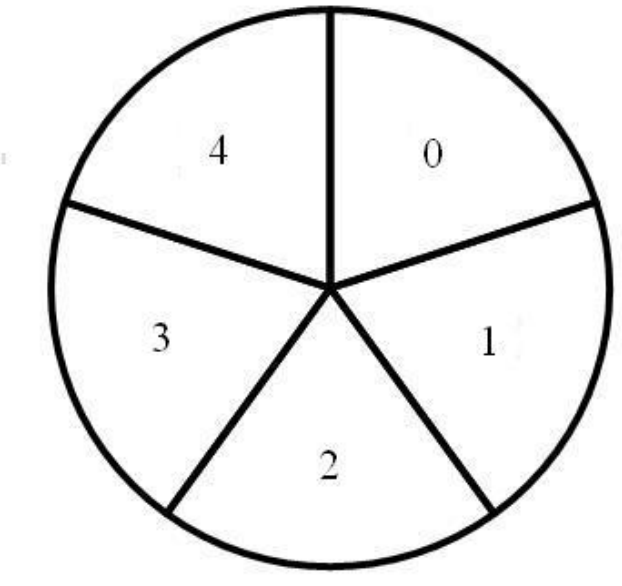

Рисунок 2.2 – Вариант разбиения пространства ключей при концепции реального представления узла

Данный вариант является самым простым для реализации. Но он имеет ряд серьезных недостатков. Например, при выходе одного из узлов (пусть это будет узел под номером 4) пространство снова делится между оставшимися узлами, как это показано на рисунок 2.3.

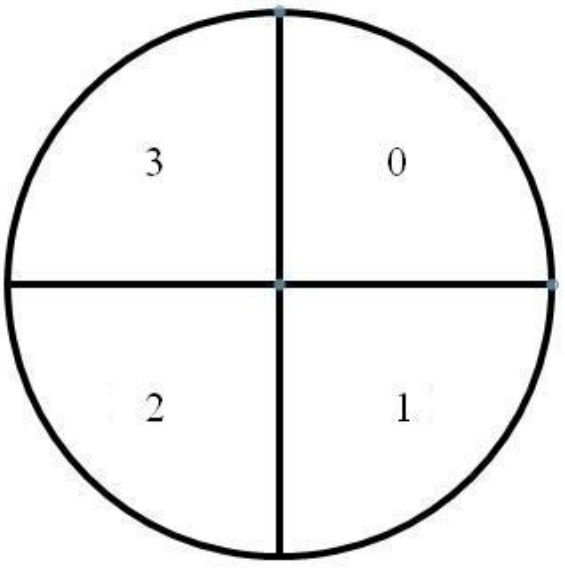

Рисунок 2.3 – Вариант разбиения пространства ключей при концепции реального представления узла после изменения состава участников кластера

Как говорилось выше, содержание кэша каждого узла полностью зависит от того, за какую часть пространства отвечает узел. Соответственно, если после изменения состава участников кластера пространство ответственности узла претерпит существенные изменения, то большая часть его кэша станет неактуальной. Таким образом, при подавляющем большинстве запросов узел уже не будет содержать запрашиваемых данных в локальном кэше, в результате для разрешения домена он будет обращаться к авторитативным серверам, производить дополнительные запросы, дожидаться ответов и т.д. Все это критическим образом скажется на производительности кластера в целом.

Следовательно, если представлять проблему графически (для лучшего восприятия), для каждого узла необходимо добиться максимального пересечения его пространства ключей до изменения, и его пространства ключей после изменения состава участников кластера.

Для рассматриваемого варианта разбиения пространства при изменении состава участников, представленном на рисунке 2.2 и рисунке 2.3, наблюдается ситуация, представленная на рисунке 2.4.

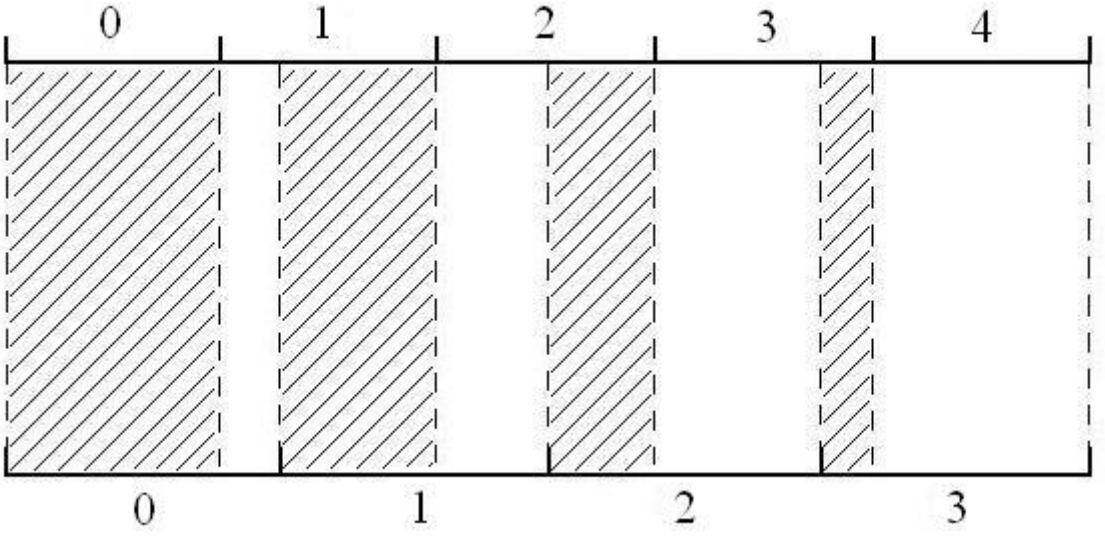

Рисунок 2.4 – Пересечение пространств узлов до и после изменения состава участников в линейном представлении (верхняя и нижняя горизонтальные линии – пространство ключей, разделенное между узлами 0, 1, 2 и т.д.)

Ситуация выхода узла под номером 4 из кластера, состоящего из узлов 0, 1, 2 и т.д., на рисунке 2.4 для удобства восприятия изображена в линейном представлении. Как видно, размеры пересечения пространств для каждого узла крайне неодинаковы. Наибольшее пересечение получилось для узла 0, большая часть его кэша останется актуальной, таким образом, изменение состава участников не сильно повлияет на скорость его работы. Но этого нельзя сказать про другие узлы. Высвободившийся участок вышедшего узла 4 полностью перешел только одному узлу под номером 3, причем этому узлу придется практически полностью обновить свой кэш, что существенно скажется на времени его ответов. Меньше всего обновлять свой кэш придется узлу 0, узлу 1 – уже больше и т.д. по нарастающей. В итоге получается большая неравномерность в перераспределении освободившегося пространства, что является неприемлемым.

Аналогичная ситуация наблюдается и при входе новых узлов в состав кластера.

Такое поведение при изменении состава участников кластера совершенно неприемлемо ввиду крайне большой неравномерности в перераспределении областей ответственности узлов, что критично скажется на скорости обработки запросов кластером.

Также отрицательным моментом данной концепции является то, что все пространство ключей разбивается на сравнительно небольшое количество отрезков, из-за чего их размеры будут достаточно велики. Так как в реальности одни домены запрашиваются клиентами гораздо чаще других, то неизбежно появление таких областей на пространстве ключей, для которых число попавших в них значений будет существенно превышать этот показатель в других областях. Данный факт приведет к неравномерности распределения нагрузки на узлы кластера.

2.4 Одноуровневая модель организации таблиц вариантов распределения

Таким образом, было принято решение остановиться на концепции виртуального узла (по аналогии с системой Dynamo). 3-я стратегия разбиения пространства ключей в системе Dynamo при тестировании равномерности распределения нагрузки показала наилучшие результаты, так как случайное распределение токенов по кольцу, как это делалось в 1-ой и 2-ой стратегии, приводило к неравномерным областям ответственности узлов, в отличие от третьей стратегии, где все пространство разбивалось на некоторое количество одинаковых отрезков.

В качестве основной функции хеширования было принято решение использовать 64-х битную хеш-функцию. Следовательно, мы имеем пространство ключей размером 2^{64}.

Далее по аналогии с третьей стратегией Dynamo необходимо разделить все пространство ключей на области достаточно небольшого размера, для того чтобы

нивелировать неоднородность распределения ключей на области значений хеш-функции.

Было решено разбить все пространство ключей на 2^{32} одинаковых частей, которые в дальнейшем будем называть блоками. Если «раздать» получившиеся блоки узлам (аналогично токенам), то каждому узлу придется хранить информацию о своих токенах, а при его выходе каким-либо образом перераспределять высвободившиеся блоки между оставшимися узлами. Также и при входе узел должен «забрать» у других узлов свою часть блоков. Причем, исходя из требования максимального пересечения пространств для каждого узла при изменении состава участников кластера, узел должен забирать и отдавать только определенные токены, а их количество должно зависеть от количества узлов в системе. Все эти действия в системе должны быть строго синхронизированы.

Также, такая концепция разбиения требует хранения довольно большого контекста, в результате чего задействуется жесткий диск. Однако работа с жестким диском вносит непредсказуемые задержки, которые могут критично сказаться на производительности системы. Так как обращение к системе балансировки нагрузки происходит при поступлении каждого клиентского запроса, то для обеспечения минимальной задержки, вносимой алгоритмом поиска ответственного узла, всю работу необходимо производить строго с использованием только оперативной памяти без обращения к жесткому диску. В этом случае время ответа сервера будет гораздо более предсказуемым.

Поэтому в данной системе было принято решение использовать «двойное» разбиение пространства. Т.е. все пространство сначала разбивается на 2^{32} одинаковых блоков, а в свою очередь каждый блок делится на количество присутствующих в данный момент узлов кластера. Получаем некоторое количество (равное числу узлов) одинаковых по размеру отрезков в пределах каждого блока, и уже за каждый такой отрезок отвечает определенный узел.

На данном этапе возникает задача того, каким образом распределить отрезки внутри каждого блока по узлам. В самом простом варианте, узлы можно

расположить в определенном порядке, например, в порядке возрастания идентификаторов, и повторить этот порядок в каждом блоке. В этом случае нет необходимости в хранении большого контекста, а вычислять узел математическим путем «на лету», при поступлении клиентского запроса.

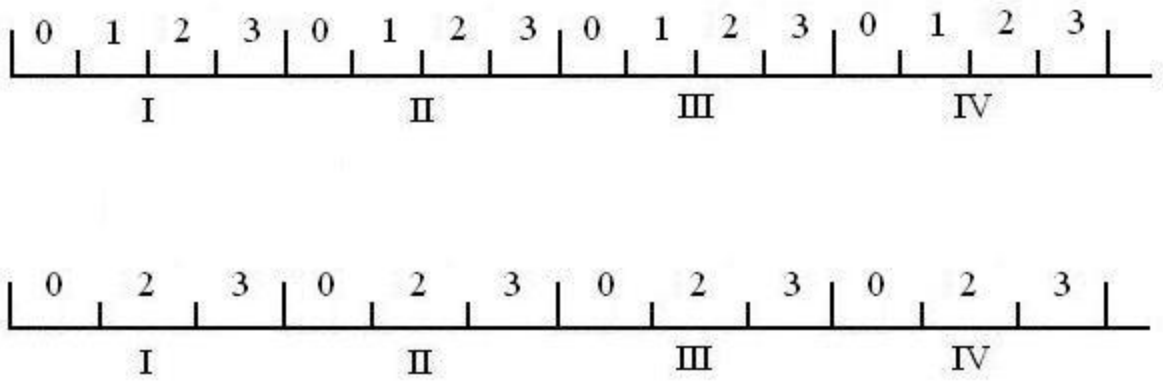

Рисунок 2.5 – Вариант разбиения пространства при двойном делении до и после выхода узла 1 из состава кластера

Данный подход представлен в линейном виде на рисунке 2.5, где I, II, III, IV – обозначение блоков; 0, 1, 2, 3 – идентификаторы узлов. Данная схема имеет высокие показатели по равномерности распределения нагрузки, но при каждом изменении состава участников в каждом блоке будет наблюдаться ситуация, изображенная на рисунке 2.4. На рисунке 2.5 из состава кластера вышел узел с идентификатором 1, при этом его пространство ответственности переходит только к его соседям – узлам 0 и 2, и такая ситуация будет повторяться в каждом блоке. В целом, данная ситуация эквивалентна первому простейшему варианту разбиения пространства с концепцией реального узла.

Следовательно, необходимо сделать так, чтобы при выходе узел отдавал свое пространство не только двум своим соседним узлам, а в равном количестве всем узлам кластера. Из этого следует, что порядок следования узлов в каждом блоке должен быть различным. В идеализированном случае система должна рассмотреть все возможные варианты перестановок узлов, тогда можно будет

добиться идеального перераспределения освободившегося пространства среди оставшихся узлов. Но при этом из комбинаторики известно, что для N узлов существует N! всевозможных перестановок. Таким образом, для относительно небольшого количества узлов равного 10, количество всевозможных вариантов 10! = 3628800. Данный подход очень сложен для вычисления, хранения и поиска искомого узла, что в конечном итоге критично сказывается на скорости работы системы. Также это решение практически не масштабируется и плохо ведет себя при изменении состава кластера, так как в каждом блоке порядок следования отрезков ответственности узлов не должен меняться при выходе или входе узла, что крайне сложно поддерживать при таком решении. А реализация функции пересчета вариантов расстановок с учетом всех ограничений представляется достаточно сложной задачей.

Поэтому решено было ввести 512 вариантов распределения узлов, где каждый вариант распределения соответствует одному блоку, и хранить их в виде таблицы. Т.к. вариантов распределения значительно меньше, чем блоков (512 и 2^{32}), то варианты распределения повторяются циклично: 1-му блоку соответствует 1-ый вариант, 2-му блоку – 2-ой вариант, 512-му – 512-ый, 513-му блоку – 1-ый вариант и т.д. Такое количество вариантов распределения было получено экспериментально исходя из равномерности распределения запросов: до 512 наблюдалось улучшение показателей равномерности, более 512 показатели практически не изменялись. Варианты распределения вычисляются с помощью собственно реализованной функции генерации псевдослучайных чисел (назовем ее *dht_rand()*), чтобы исключить различия в ее библиотечных реализациях и обеспечить одинаковое поведение на платформах с различной разрядностью.

Таким образом, получается таблица распределения, на основании которой происходит поиск узла, который является ответственным за определенный ключ.

Представим графически отображение таблицы распределения на область значений хеш-функции (рисунок 2.6).

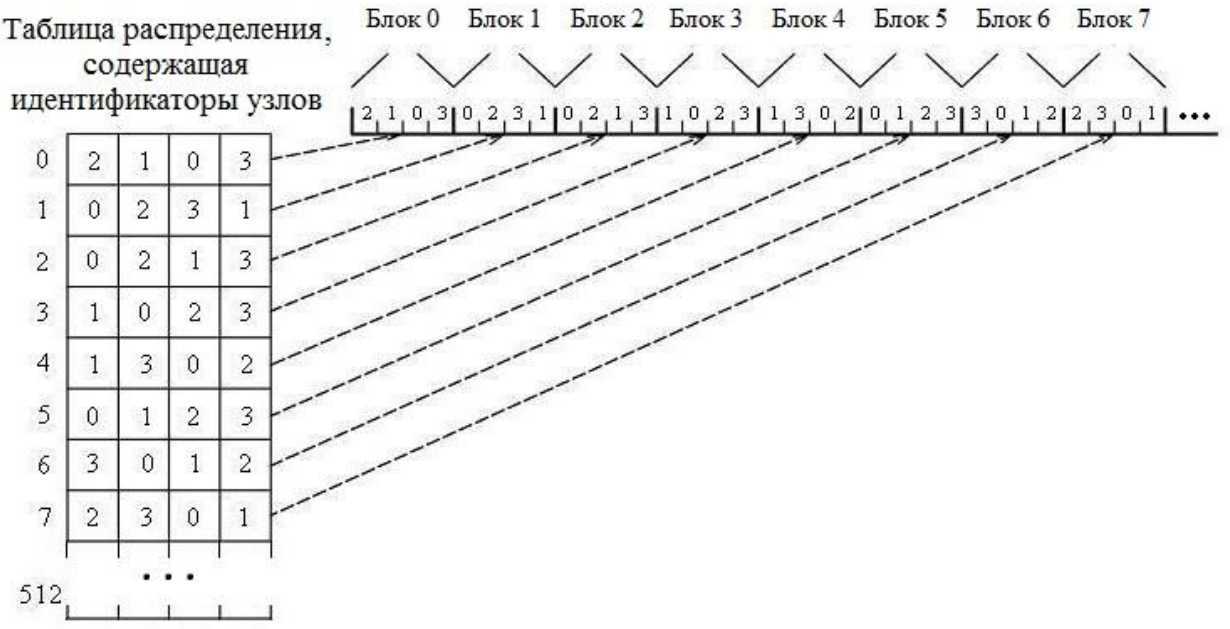

Рисунок 2.6 – Отображение таблицы распределения узлов
на область значений хеш-функции

Утверждение 1. Пусть L_1, L_2, …, L_n – суммарные размеры областей ответственности узлов с идентификаторами 1, 2, …, n соответственно, n – количество узлов в кластерной системе. Тогда при отсутствии неработоспособных узлов каждый из узлов системы суммарно отвечает за равную часть области значений хеш-функции:

$$L_1 = L_2 = \ldots = L_n. \tag{2.18}$$

Доказательство: возьмем i-ый узел системы, где i=1, 2, …, n. В каждом из блоков данный узел отвечает за области ответственности равного размера l_i:

$$l_i^1 = l_i^2 = \ldots = l_i^k = l_i, \tag{2.19}$$

где k – количество блоков. Таким образом, суммарный размер областей ответственности i-го узла будет определяться как:

$$L_i = \sum_{j=1}^{k} l_i^j = k l_i. \tag{2.20}$$

Так как в рамках каждого блока все узлы системы отвечают за области одинакового размера, то имеем:

$$l_1 = l_2 = \ldots = l_i = \ldots = l_n = l. \tag{2.21}$$

Таким образом, в конечном итоге получаем:

$$L_i = kl_i = kl,\ i = 1,...,n, \qquad (2.22)$$

где n – количество узлов в системе, т.е. $L_1 = L_2 = ... = L_n$.

Данное утверждение позволяет сделать предположение о равномерном размещении ключей среди узлов системы, т.е. о равномерности распределения нагрузки.

Информация об участвующих узлах предоставляется системой слежения за составом кластера. Данная децентрализованная система работает на каждом из узлов без присутствия какого-либо узла-координатора, таким образом достигается отсутствие единой точки отказа. Система слежения периодически передает от узла к узлу информацию о текущем составе кластера, при этом принимающий узел обязан отправить в ответ подтверждение приема данных. В случае если какой-либо из узлов не отвечает в течение определенного времени, он помечается как «мертвый», таблица узлов обновляется и рассылается участникам кластера.

Система слежения за составом кластера предоставляет информацию о текущем составе системе балансировки нагрузки. При этом система балансировки нагрузки имеет свою локальную копию таблицы узлов, представляющую собой массив, расположенный в оперативной памяти. Для экономии памяти элементы таблицы вариантов распределения узлов содержат позиции узлов в локальной таблице.

2.4.1 Построение таблицы вариантов распределения

Как было сказано выше, в каждой из строк таблицы должен быть свой порядок следования узлов, причем узлы не должны терять своего взаимного положения при вводе в систему новых узлов.

Предлагаемое решение состоит в получении некоторого случайного значения, исходя из параметра, который является уникальным для каждого узла – например, IP адреса. Получив на некотором пространстве случайные значения для узлов, можно «собрать» узлы в массив в порядке возрастания соответствующих им значений. Однако здесь присутствуют два ограничения:

1) для каждого варианта распределения (т.е. в каждой строке таблицы) узел должен получать различные положения;
2) узел должен получать с одной стороны случайное, а с другой стороны – постоянное значение для данного варианта распределения.

Для того чтобы узлы не теряли своего взаимного положения при изменении состава участников узлов, при вычислении случайных значений необходимо избегать нормирования на количество «живых» узлов кластера, иначе изменение состава кластера будет приводить к изменению относительных позиций узлов.

Исходя из вышеперечисленного, был разработан алгоритм построения варианта распределения, который состоит в следующем. Для начала создается временная таблица, полями которой являются IP узла и соответствующее хеш-значение, полученное с помощью функции *dht_rand()*. Далее для каждого узла генерируется хеш-значение на основании комбинации его IP адреса в численном представлении (при этом используется сетевой порядок байт, *network byte order*) и номера варианта распределения. Полученное значение и соответствующий IP адрес записываются во временную таблицу. После того, как хеш-значения получены для каждого IP, таблица сортируется по хеш-значениям в порядке возрастания (рисунок 2.7).

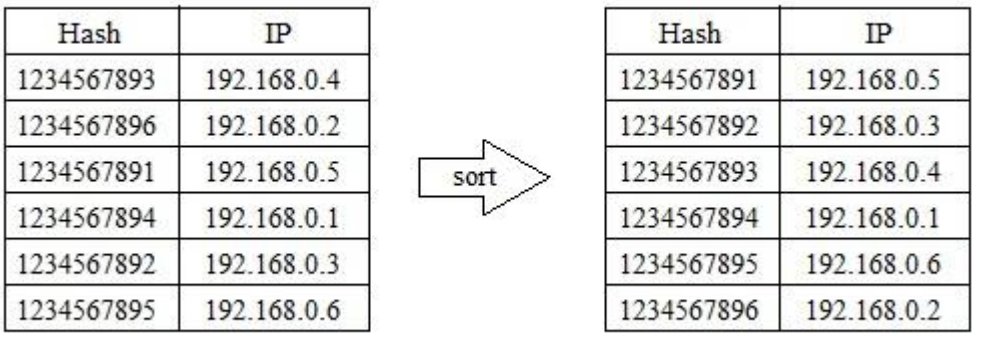

Рисунок 2.7 – Построение варианта распределения

Полученный порядок следования IP адресов и является вариантом распределения. Далее действия повторяются для следующего варианта.

Таким образом, мы получаем «детерминированную случайность»: с одной стороны, позиции узлов получаются случайным образом, но с другой эти позиции

не меняются для данного блока при изменении состава участников кластера.

В конечном итоге получается таблица размерностью 512×N (где N – количество узлов), в соответствии с которой строится функция отображения ключей в узлы. При изменении состава участников кластера все действия повторяются и таблица обновляется, причем обновление DHT на каждом из узлов происходит без остановки кластера.

Принципиально важным моментом в одноуровневой модели является то, что в процессе построения таблицы вариантов распределения учитываются только «живые» узлы кластера. При этом временно вышедшие узлы (по причине сбоя программного или аппаратного обеспечения, технического обслуживания и т.д.), помеченные как «мертвые» в таблице узлов, предоставляемой системой слежения за составом кластера, в процессе построения таблицы вариантов распределения игнорируются. Таким образом, таблица вариантов распределения всегда гарантированно содержит только позиции «живых» узлов.

2.4.2 Алгоритм поиска ответственного узла для входящего домена

Алгоритм поиска ответственного за входящий домен узла работает при поступлении каждого запроса.

Необходимо заметить, что алгоритм поиска ответственного узла должен быть максимально простым. Это обусловлено тем, что при работе системы в реальном времени крайне важна скорость обработки запросов. Следовательно, если на вычисление узла будет тратиться существенное количество времени, то смысл кластеризации системы теряется. Ответственный узел должен вычисляться математически «на лету», при поступлении запроса, без применения какого-либо сложного поиска по спискам, таблицам и т.д.

Также алгоритм поиска ответственного узла не должен ограничивать масштабируемость кластерной системы, т.е. скорость его работы не должна зависеть от количества узлов в системе.

В данной реализации искомый узел находится с помощью двух целочисленных делений и двух делений с остатком, что является приемлемым по

затратам времени: определяется номер блока, соответствующий ему вариант распределения, позиция отрезка ответственности узла и сам узел. Так как и таблица распределения, и таблица узлов представляют собой массивы, то вся процедура вычисления нужного узла происходит без какого-либо поиска по структурам данных, что обычно является весьма затратным по времени. На выходе каждого действия получается индекс массива, после чего происходит непосредственное обращение к элементу. Блок-схема алгоритма представлена на рисунке 2.8.

Рисунок 2.8 – Алгоритм поиска ответственного узла с применением одной таблицы вариантов распределения

Как было сказано выше, в одноуровневой модели присутствует одна таблица распределения, которая содержит только позиции «живых» узлов кластера. Следовательно, для каждого входящего домена алгоритм поиска ответственного

узла гарантированно с первого раза находит функционирующий в штатном режиме узел, который способен в данный момент обрабатывать входящие запросы.

Утверждение 2. Для любого элемента данных, идентифицируемого базовой хеш-функцией, существует ответственный узел в рамках текущего непустого состава кластерной системы:

$$\forall d \in D \; \exists \, s \in S, \; S \neq \varnothing, \tag{2.23}$$

где D – множество элементов данных, которые могут быть идентифицированы базовой хеш-функцией, S – непустое множество узлов кластерной системы.

Доказательство: каждый элемент данных d из множества D с помощью базовой хеш-функции может быть преобразован в определенное значение в пределах области значений хеш-функции:

$$0 \le h(d) < M, \tag{2.24}$$

где $h(d)$ – базовая хеш-функция, M – максимальное количество различных значений, которые могут быть получены с помощью базовой хеш-функции. В реализованном программном комплексе используется 64-х битная хеш-функция, таким образом:

$$0 \le h(d) < 2^{64}. \tag{2.25}$$

Согласно рассмотренной выше схеме разбиения вся область значений базовой хеш-функции [0, 2^{64}-1] делится на блоки, в свою очередь все пространство внутри каждого блока делится на области ответственности узлов. Таким образом, при наличии узлов в системе не существует такой части области значений базовой хеш-функции, за которую не отвечает ни один из узлов. Следовательно, для любого элемента данных, идентифицируемого базовой хеш-функцией, существует ответственный узел в рамках текущего непустого состава кластерной системы.

2.4.3 Проверка равномерности распределения нагрузки

В ходе тестирования системы распределения нагрузки за основу были взяты реальные данные – 5000000 уникальных доменов из запросов, принятых за две недели работы системой DNS. Тестовые запуски проводились для различного

количества узлов кластера: два, три, пять и десять узлов. В каждом из тестов к системе были обращены 5000000 запросов с собранными уникальными доменами**.**

На рисунке 2.9 в виде графика представлены результаты проведенных тестов для 5-ти узлов в кластере, где эффективность распределения нагрузки определяется как отношение среднего числа запросов, обслуживаемых каждым узлом, к максимальному количеству запросов, принятых самым нагруженным узлом. По горизонтали обозначено различное количество запросов к системе с шагом 1000000 (1000000, 2000000, 3000000 и т.д.), по вертикали – эффективность в виде отношения средней нагрузки к максимальной нагрузке.

Рисунок 2.9 – Показатели эффективности для 5-ти узлов кластера

Как видно на представленном графике, каждый узел системы получает практически одинаковое количество запросов. Расчеты показывают, что в среднем отклонение составляет около 0.07% (для запросов с уникальными доменами).

Таким образом, предложенный метод разбиения пространства ключей обладает высокими показателями равномерности распределения нагрузки.

2.5 Двухуровневая модель организации таблиц вариантов распределения

В данном пункте будут рассмотрены недостатки одноуровневой модели организации таблиц распределения и их устранение с помощью усовершенствованной модели применения DHT – двухуровневой модели организации таблиц вариантов распределения, а также ее детальное описание и анализ.

2.5.1 Недостатки одноуровневой модели организации таблиц вариантов распределения

В предыдущих пунктах данной главы была описана одноуровневая модель организации вариантов распределения. Данная модель содержит одну таблицу распределения, в которой находятся только «живые» узлы кластера. При этом было показано, что данная схема обладает высокими показателями равномерности распределения нагрузки и полностью удовлетворяет требованиям, предъявляемым к обработке входа узла в кластер.

Однако в ходе дальнейших исследований и выявились некоторые недостатки в процедуре обработки временного выхода узла из состава кластера.

Допустим, таблица содержит четыре «живых» узла, условно обозначенных как 0, 1, 2, 3. Стоит напомнить, что вся область значений хеш-функции (2^{64}) делится на 2^{32} блоков. В свою очередь каждый блок в равных частях делится на количество «живых» узлов кластера.

Теперь детально рассмотрим какой-либо из блоков, например, 1-ый, до и после выхода узла 2 из кластера (рисунок 2.10). Размеры блоков всегда остаются неизменными, изменяются только размеры пространств ответственности узлов внутри блоков.

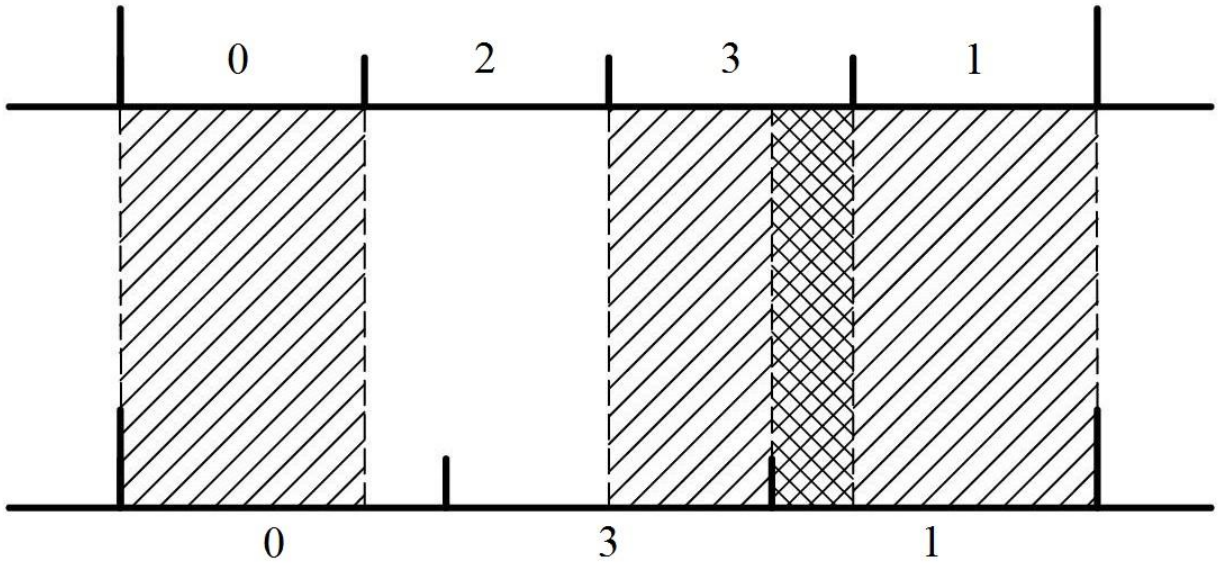

Рисунок 2.10 – Ситуация выхода узла 2 из кластера в пределах одного блока в линейном представлении (верхняя и нижняя горизонтальные линии – пространство ключей до и после выхода соответственно)

Как видно на рисунке 2.10, пространства ответственности узлов 0 и 1 только расширились (пересечение нового пространства с прежним для каждого узла показано обычной штриховкой). Для этих двух узлов прежнее пространство полностью включается в новое, а это значит, что домены, ключи которых попадают в эти пространства, не перешли в области ответственности других узлов. Следовательно, узлы 0 и 1 полностью сохранили актуальность своих кэшей.

Однако для узла 3 произошло смещение пространства ответственности. Область, помеченная перекрестной штриховкой, мигрировала из области ответственности узла 3 в область ответственности узла 1. Иначе говоря, кэш узла 3 стал частично неактуален из-за перехода к узлу 1 соответственной части доменов, обслуживаемых узлом 3 до выхода узла 2 из состава кластера. Т.е. требование отсутствия перераспределения запросов, обслуживаемых оставшимися узлами до и после изменения состава кластера, частично не выполняется.

На практике, так как область значений хеш-функции достаточно большая, такие смещения вызывают миграцию существенного количества ключей (порядка 358×10^6 ключей в пределах одного блока в данном конкретном случае, где кластер состоит из 4-х узлов), т.е. довольно большая часть доменов становится незакэшированной.

Для предотвращения описанной ситуации был реализован подход, основанный на применении двухуровневой модели организации таблиц вариантов распределения.

2.5.2 Таблицы вариантов распределения 1-го и 2-го уровней

Суть двухуровневой модели организации таблиц вариантов распределения состоит в применении двух таблиц распределения вместо одной. При этом одна из таблиц (2-ой уровень) фактически представляет собой ту же самую таблицу, которая применялась в одноуровневой модели, она содержит только «живые» узлы кластера.

Ключевым моментом здесь является введение еще одной таблицы вариантов распределения (1-ый уровень), содержащей как «живые» узлы кластера, так и те, которые помечены как «мертвые» [17].

Здесь стоит напомнить, что наряду с таблицами вариантов распределения в системе существует базовая таблица всех узлов кластера с пометкой «живой» или «мертвый», на ее основе строятся таблицы распределения. При этом существуют следующие сценарии изменения данной таблицы:

- появление нового узла в системе: узел добавляется в таблицу и помечается как «живой»;
- временный выход узла из кластера: соответствующий узел не удаляется из базовой таблицы, а лишь помечается как «мертвый»;
- вход узла обратно в кластер после временного выхода: соответствующий узел помечается в базовой таблице как «живой»;
- окончательный выход узла: соответствующий узел удаляется из базовой таблицы.

При любом из этих событий сначала изменяется базовая таблица, после чего на ее основе строятся таблицы вариантов распределения.

Процедура построения таблиц распределения теперь выглядит следующим образом. Как и в прошлой реализации, обе таблицы содержат 512 вариантов распределения. При построении варианта распределения для каждого узла

(независимо от того, является он в данный момент «живым» или «мертвым») рассчитывается хеш-значение на основе комбинации его IP-адреса и номера варианта, после чего полученные значения сортируются в порядке возрастания. В случае возникновения коллизий, т.е. совпадения хеш-значений (что маловероятно, однако такой вариант также необходимо предусмотреть), происходит сравнение значений IP-адресов узлов в численном представлении, при этом используется сетевой порядок байт (*network byte order*). Полученный после сортировки порядок следования хеш-значений определяет позиции соответствующих узлов в варианте распределения. При этом в таблицу распределения 1-го уровня записываются все узлы (включая «мертвые»), а в таблицу 2-го уровня – только «живые». Таким образом, обе таблицы строятся параллельно и имеют одинаковый порядок следования узлов в соответствующих вариантах распределения за исключением того, что в таблице 2-го уровня пропущены узлы, помеченные как «мертвые».

Графически отображение обеих таблиц распределения на одну и ту же область значений хеш-функции представлено на рисунке 2.11 (где все узлы являются «живыми») и рисунке 2.12 (где узел 2 становится «мертвым»). В обоих случаях отображение таблицы 1-го уровня располагается над отображением таблицы 2-го уровня.

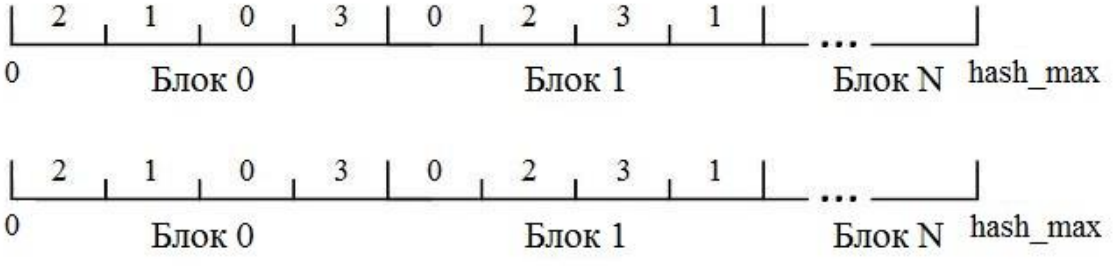

Рисунок 2.11 – Отображение таблиц распределения на область значений хеш-функции (отображение таблицы 1-го уровня располагается над отображением таблицы 2-го уровня) до временного выхода узла 2 из состава кластера

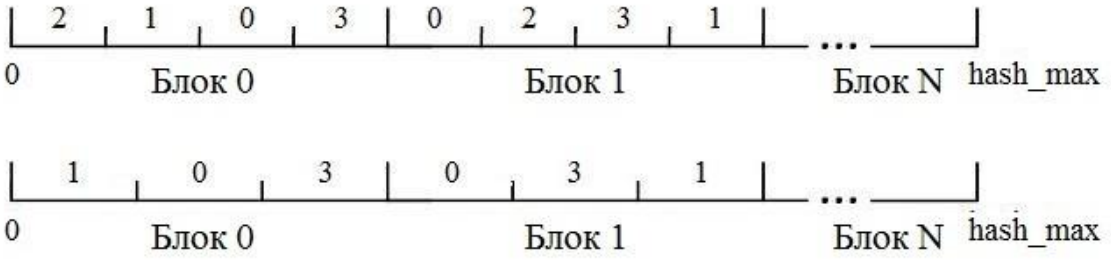

Рисунок 2.12 – Отображение таблиц распределения на область значений хеш-функции (отображение таблицы 1-го уровня располагается над отображением таблицы 2-го уровня), узел 2 временно вышел из состава кластера

Как видно на рисунке 2.11, обе таблицы имеют абсолютно одинаковое отображение. Однако на рисунке 2.12, где узел 2 стал «мертвым», произошло смещение пространств ответственности узлов для таблицы 2-го уровня (аналогичное тому, которое мы рассматривали при использовании одноуровневой схемы), но для таблицы 1-го уровня смещение не произошло за счет содержания «мертвого» узла.

Таким образом, участвующие узлы полностью сохраняют актуальность локальных кэшей при временных выходах узлов из состава кластера.

2.5.3 Алгоритм поиска ответственного узла с применением таблиц 1-го и 2-го уровней

Теперь рассмотрим алгоритм поиска ответственного узла при использовании двухуровневой модели.

Как и в случае одноуровневой модели, при поступлении домена на обработку сначала для него рассчитывается хеш-значение. Далее на первом шаге алгоритм производит попытку определения ответственного узла, используя таблицу 1-го уровня, которая может содержать «мертвые» узлы. При попадании хеш-значения в область ответственности «живого» узла данный узел выбирается ответственным, и на этом алгоритм прекращает работу без использования таблицы

2-го уровня. Однако если хеш-значение попало в область ответственности «мертвого» узла, алгоритм на втором шаге принимает решение, используя таблицу 2-го уровня, где хеш-значение гарантированно попадает в область ответственности «живого» узла.

Блок-схема описанного алгоритма представлена на рисунке 2.13.

Рисунок 2.13 – Блок-схема алгоритма поиска ответственного узла с применением таблиц 1-го и 2-го уровней

Таким образом, алгоритм поиска ответственного узла состоит из одного или двух этапов:

1) попытка определения ответственного узла с использованием таблицы 1-го уровня;
2) определение ответственного узла с использованием таблицы 2-го уровня, если на первом этапе выбранный ответственный узел оказался «мертвым».

Представим рассмотренный алгоритм графически на примере блока 1 до и после временного выхода узла 2 из состава кластера (рисунок 2.14 и рисунок 2.15 соответственно).

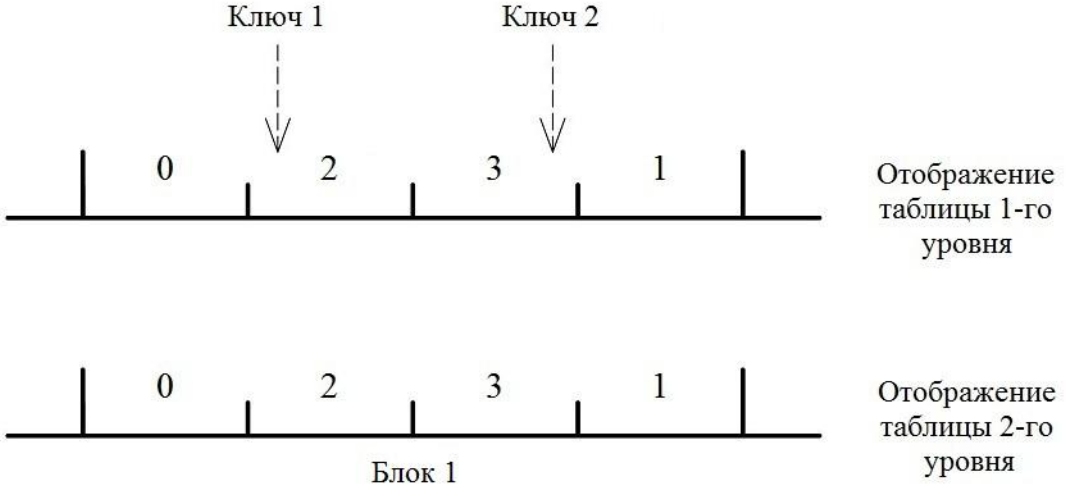

Рисунок 2.14 – Поиск ответственных узлов для двух различных доменов до временного выхода узла 2 из состава кластера

Предположим, что ключ 1 и ключ 2 получены от двух различных доменов. На рисунке 2.14 все узлы являются «живыми» и в обоих случаях решение принимается только на основании таблицы 1-го уровня: для ключа 1 ответственным является узел 2, для ключа 2 соответственно узел 3. Отметим, что при полном составе участников кластера решение всегда принимается с помощью таблицы 1-го уровня, т.к. таблицы 1-го и 2-го уровней в этом случае полностью идентичны.

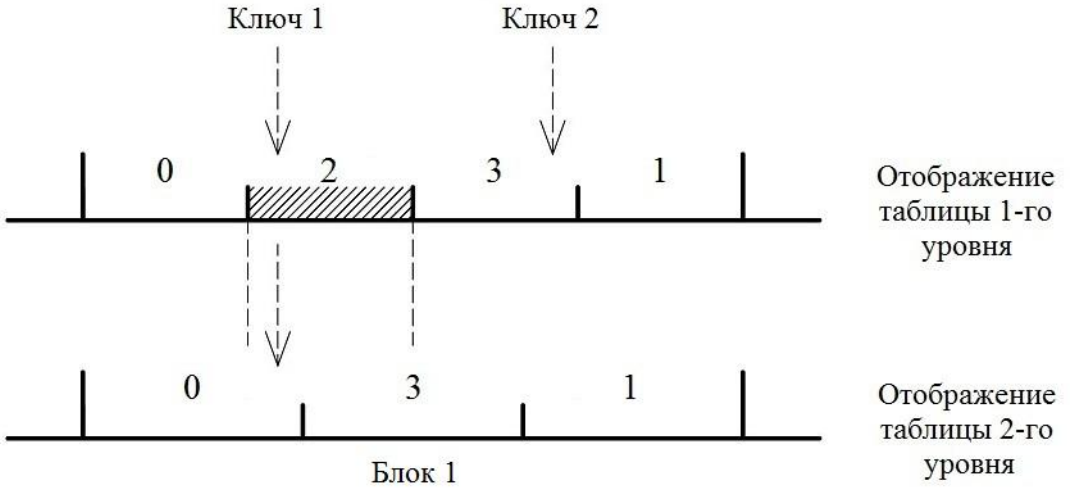

Рисунок 2.15 – Поиск ответственных узлов для двух различных доменов после временного выхода узла 2 из состава кластера

На рисунке 2.15 узел 2 временно вышел из состава кластера (его область ответственности помечена штриховкой). В этом случае ключ 1 на первом этапе попадает в область ответственности вышедшего узла 2, после чего «проваливается» через отображение таблицы 1-го уровня на отображение таблицы 2-го уровня, где попадает в область ответственности «живого» узла 0. Таким образом, теперь ответственным узлом для ключа 1 стал узел 0, для ключа 2 ответственный узел не изменился, что видно на рисунке 2.15.

Здесь именно ключ 2 достоин особого внимания. Рисунок 2.15 показывает, что для таблицы 2-го уровня произошло смещение пространств ответственности узлов, и при отсутствии таблицы 1-го уровня ответственным узлом для ключа 2 после выхода узла 2 стал бы узел 1, несмотря на то, что узел 3 не покинул состав кластера. В этом состоит недостаток одноуровневой модели, где таблица 1-го уровня отсутствует, и решение принимается только с помощью таблицы 2-го уровня. Применение таблицы 1-го уровня, содержащей и «мертвые», и «живые» узлы, позволяет полностью избежать миграцию ключей за счет отсутствия смещения пространств ответственности узлов. Именно в этом и есть суть применения двухуровневой модели.

На рисунке 2.15 видно, что пространство ответственности узла 2 после его выхода перераспределяется между узлами 0 и 3, причем в неравных пропорциях. Из этого можно предположить, что перераспределение неравномерное, узел 0 принял большую нагрузку, чем узел 3. Однако это происходит только внутри одного конкретного блока. Мы имеем 2^{32} блоков и 512 вариантов распределения с различным порядком следования узлов, что нивелирует локальную неравномерность перераспределения пространств ответственности. Данных факт подтвержден на практике.

Стоит заметить, что критическим здесь является предположение о том, что выход узла из кластера всегда является временным (по причине технического обслуживания, обновления программного обеспечения, различного рода сетевых проблем, сбоя в работе сервера и т.д.). Существует возможность и окончательного выхода узла из кластера с его удалением из базовой таблицы, однако после этого системе необходимо некоторое время на стабилизацию, т.к. происходит неизбежное смещение пространств ответственности узлов, чего лишен временный выход узла. Практика показывает, что безвозвратный выход узлов из состава DNS-кластера является крайне редким событием, в подавляющем большинстве случаев состав DNS-кластера является стабильным [17].

2.6 Разработанная модель балансировки нагрузки

Таким образом, в конечном итоге разработанная модель балансировки нагрузки представляет собой совокупность следующих компонентов:

- схема разбиения области значений базовой хеш-функции на области ответственности узлов кластерной системы;
- способ хранения схемы разбиения в виде таблиц вариантов распределения.

В свою очередь в рамках способа хранения схемы разбиения области значений базовой хеш-функции были разработаны одноуровневая и двухуровневая модели организации таблиц вариантов распределения, описанные выше в главе.

Рисунок 2.16 – Разработанная модель балансировки нагрузки

Графически разработанная в данной работе модель балансировки нагрузки представлена на рисунке 2.16.

2.7 Возникновение коллизий в процессе функционирования распределенной хеш-таблицы

В качестве базовой хеш-функции в реализованном программном комплексе используется MurmurHash2 (64-х битная версия) – быстрая некриптографическая хеш-функция общего назначения, разработанная Остином Эпплби (Austin Appleby) в 2008 г. Проведенные исследования Питера Канковского (Peter Kankowski) выявили ее высокие показатели быстродействия и распределения хеш-значений для различных входных данных [41].

Несмотря на высокие показатели хеш-функции MurmurHash2 в области уникальности генерируемых ключей, неизбежны ситуации, при которых возникают коллизии, т.е. получение одинаковых ключей для различных входных последовательностей. Однако в случае распределенных хеш-таблиц не требуется наличие специального механизма разрешения коллизий. Полученный с помощью базовой хеш-функции ключ участвует только в операции определения узла,

ответственного за обработку входящего запроса. Далее вся работа происходит непосредственно с элементом данных без участия ассоциированного с ним ключа. Таким образом, получение одинаковых ключей для различных элементов данных означает лишь их принадлежность одному и тому же узлу кластера.

2.8 Сложность алгоритмов построения таблиц вариантов распределения и поиска ответственного узла, оценка масштабируемости системы

В данном пункте под сложностью алгоритмов будет подразумеваться исключительно временная сложность.

Основными алгоритмами реализованного комплекса программ являются алгоритмы построения таблиц вариантов распределения и поиска ответственного за пришедший домен узла. Таким образом, сложность данных алгоритмов является одним из показателей, определяющих эффективность предлагаемых методов балансировки нагрузки и соответствующей программной реализации.

Наиболее важным алгоритмом во всем программном комплексе является алгоритм поиска ответственного за домен узла. Данный алгоритм работает для каждого входящего запроса, количество которых для публичного DNS-сервиса может достигать сотен и тысяч в секунду. В предыдущих пунктах было показано, что процедура вычисления ответственного узла включает в себя несколько арифметических операций по определению позиции необходимого элемента в таблице вариантов распределения, после чего следует непосредственное обращение по индексу к элементу в локальной таблице узлов. При этом в алгоритме не фигурирует количество участвующих узлов в системе. Таким образом, сложность алгоритма поиска ответственного узла не зависит от количества узлов кластера и является константным, т.е. временную сложность можно определить как $O(C)$. Следовательно, операция вычисления ответственного узла будет одинаково быстро работать для десятков и тысяч участвующих узлов. Данный факт позволяет заключить, что рассмотренный

алгоритм никаким образом не ограничивает масштабируемость распределенной системы.

В отличие от алгоритма поиска ответственного узла, алгоритм построения таблиц вариантов распределения зависит от количества узлов в системе. Наиболее сложным алгоритмом в процедуре построения таблиц вариантов распределения является быстрая сортировка (*quick sort*), среднее время работы которой составляет $O(N\log(N))$ [37, 38]. Таким образом, итоговая временная сложность алгоритма построения таблиц также определяется как $O(N\log(N))$, где N – количество узлов кластера.

Данные показатели временной сложности алгоритмов справедливы как для одноуровневой, так и для двухуровневой модели организации таблиц вариантов распределения.

Также была произведена оценка потенциальной масштабируемости системы, управляемой рассматриваемым программным комплексом. Алгоритм построения таблиц вариантов распределения, работа которого инициируется при изменении состава участников кластера, является единственным алгоритмом, где временная сложность зависит от количества узлов. Таким образом, показатели времени работы программной реализации данного алгоритма при варьируемом количестве участвующих узлов имеет определяющее значение при оценке потенциальной масштабируемости системы.

Время работы алгоритма, реализованного на языке программирования C [44, 45, 46, 47, 48] для соответствующего числа узлов кластера представлено в таблице 2.1. Данные показатели были получены экспериментально при использовании следующего аппаратного обеспечения:

- процессор: Intel(R) Core(TM) i7-3770K CPU, 8M Cache, 3.50GHz, 4 Cores, 8 Threads;
- объем оперативной памяти: 16 GB.

Таблица 2.1

Время работы алгоритма построения таблиц вариантов распределения для различного числа узлов кластера

Кол-во узлов	Время работы (мс)	Кол-во узлов	Время работы (мс)
100	3	10^4	565
200	7	2×10^4	1202
300	11	3×10^4	1869
400	16	4×10^4	2584
500	20	5×10^4	3312
10^3	43	10^5	7076
2×10^3	96	2×10^5	15333
3×10^3	149	3×10^5	23959
4×10^3	204	4×10^5	33431
5×10^3	262	5×10^5	42004
10^4	565	10^6	90811

В процессе эксперимента формировалась таблица узлов необходимого размера, содержащая корректные IP-адреса, после чего инициировался процесс построения таблиц вариантов распределения. Для данного процесса фиксировалось время начала и время окончания, на основании которых вычислялось общее время работы алгоритма построения таблиц.

В зависимости от конкретной решаемой задачи принимается время работы алгоритма, оцениваемое как критическое. Данные в таблице 2.1 показывают, что для подавляющего числа задач система может быть масштабируема до 10000 узлов, при этом в максимальном случае время построения таблиц оценивается как ~0.5 сек. Также для определенного круга задач возможно масштабирование до 100000 узлов, где максимальное время работы алгоритма занимает ~7 сек.

Маловероятным по удовлетворительности, однако потенциально возможным является масштабирование до 1000000 узлов, в этом случае время работы достигает значения ~1.5 мин. Применение более высокопроизводительного аппаратного обеспечения и усовершенствование алгоритмов позволит в перспективе снизить это время до более приемлемого.

Как было указано выше, алгоритм поиска ответственного узла работает для каждого входящего запроса, в то время как процесс перестроения таблиц

вариантов распределения инициируется при изменении состава участников кластера. Таким образом, соотношение показателей временной сложности алгоритмов построения таблиц вариантов распределения и поиска ответственного узла показывает, что наиболее эффективно описываемые методы и алгоритмы, а также соответствующая их программная реализация, могут быть применены в рамках кластерных систем, состав участников которых является относительно стабильным.

2.9 «Zero-hop» маршрутизация

На протяжении последних лет было предложено множество реализаций распределенных хеш-таблиц, таких как Kademlia [28], CAN [10], Chord [9], Pastry [12], Tapestry [11], Memcached [35], Dynamo [8], Cycloid [30], Ketama [29], RIAK [31], Maidsafe-DHT [32], Cassandra [33] и C-MPI [34]. В большинстве из этих систем количество операций по маршрутизации имеет логарифмическую зависимость от количества участвующих узлов [36], что ограничивает их возможность масштабируемости. Показательным примером здесь является проект DDNS [13], где в качестве распределенной хеш-таблицы использовалась Chord DHT, имеющая зависимость операций маршрутизации $\log(N)$ [9], где N – количество узлов системы. Следствием подобной зависимости стало существенное увеличение времени ответа системы с ростом числа участвующих узлов, достигая 350 мс при числе узлов равном 1000 [13].

В работе [8], описывающей распределенное хранилище Dynamo, заявляется, что Dynamo является «*zero-hop*» DHT, т.е. распределенной хеш-таблицей, в которой каждый из узлов содержит достаточное количество информации о маршрутизации для того, чтобы непосредственно переадресовывать данные необходимому узлу [8]. Однако существенным недостатком Dynamo является то, что данная система является исключительно внутренним проектом Amazon. Это делает невозможным использование Dynamo в отличной от Amazon инфраструктуре.

Также существует одна из новейших работ [36], представляющая ZHT – «*zero-hop*» распределенную хеш-таблицу, отвечающая требованиям высокопроизводительных компьютерных систем. ZHT имеет целью явиться «строительным блоком» для будущих распределенных систем, таких как параллельные и распределенные файловые системы, системы организации распределенного выполнения задач, а также системы параллельного программирования [36]. В работе [36] декларируется, что ZHT обеспечивает высокую доступность сервиса, отказоустойчивость системы, высокую производительность и минимальные задержки при возможности масштабируемости до миллионов узлов. Также имеется возможность динамического входа и выхода узлов и отказоустойчивой репликации.

Разработчики ZHT особенно акцентируют внимание на отсутствии многократных пересылок по сети при поиске необходимого узла («*zero-hop*» маршрутизация), выделяя данное свойство как одно из основных преимуществ перед существующими решениями.

Реализованный в рамках данной диссертации комплекс программ на основе разработанных автором методов и алгоритмов также в полной мере может называться «*zero-hop*» DHT, так как алгоритм вычисления ответственного узла способен однозначно определить ответственный за пришедший домен узел, и при этом каждый из узлов способен переадресовать входящий запрос непосредственно необходимому узлу системы. Таким образом, количество операций маршрутизации не зависит от количества узлов кластера, что позволяет минимизировать задержки на пересылку по сети, в то время как скорость ответа является одним из наиболее важных параметров DNS-сервиса.

2.10 Взаимодействие DNS-клиента с узлами кластера

Представленные в данном пункте изображения и диаграммы полностью соответствуют текущему применению рассматриваемого комплекса алгоритмов и программ как единой системы балансировки нагрузки в проекте DNS-сервиса, реализуемом компанией «Релэкс». Изображения и диаграммы справедливы как

для одноуровневой, так и для двухуровневой модели организации таблиц вариантов распределения.

Схема взаимодействия DNS-клиента с узлами DNS-кластера, а также взаимодействие непосредственно узлов кластера, представлены на рисунок 2.17 и рисунок 2.18.

Рисунок 2.17 демонстрирует ситуацию, когда все узлы кластера являются «живыми». На первом шаге клиент посылает кластеру DNS-запрос на разрешение домена. Здесь стоит напомнить, что точкой входа в кластер является любой из его узлов, т.е. у клиента есть возможность отправления запроса любому известному ему узлу кластера. При получения запроса от клиента узел кластера обращается к собственной системе балансировки нагрузки с целью определения ответственного за пришедший домен узла (рисунок 2.17). В случае если принявший запрос узел оказывается ответственным за пришедший домен, то данный узел производит поиск в своем локальном кэше, обращается к авторитативным серверам в случае отсутствия данного домена в кэше или по причине истекшего времени жизни соответствующей кэшированной записи (TTL), формирует результирующий ответ и отправляет его клиенту. При этом внутрикластерных пересылок не происходит.

Иначе, если принявший запрос узел не является ответственным за пришедший домен, то он незамедлительно производит переадресацию запроса ответственному узлу. Данный сценарий изображен на рисунке 2.17. Приняв переадресованный запрос, ответственный узел производит вышеперечисленные операции по поиску соответствующей записи в локальном кэше и обращению в случае необходимости к авторитативным серверам. Сформировав ответ, ответственный узел отправляет его узлу, первоначально принявшему запрос, который, в свою очередь, незамедлительно отправляет ответ клиенту (рисунок 2.17).

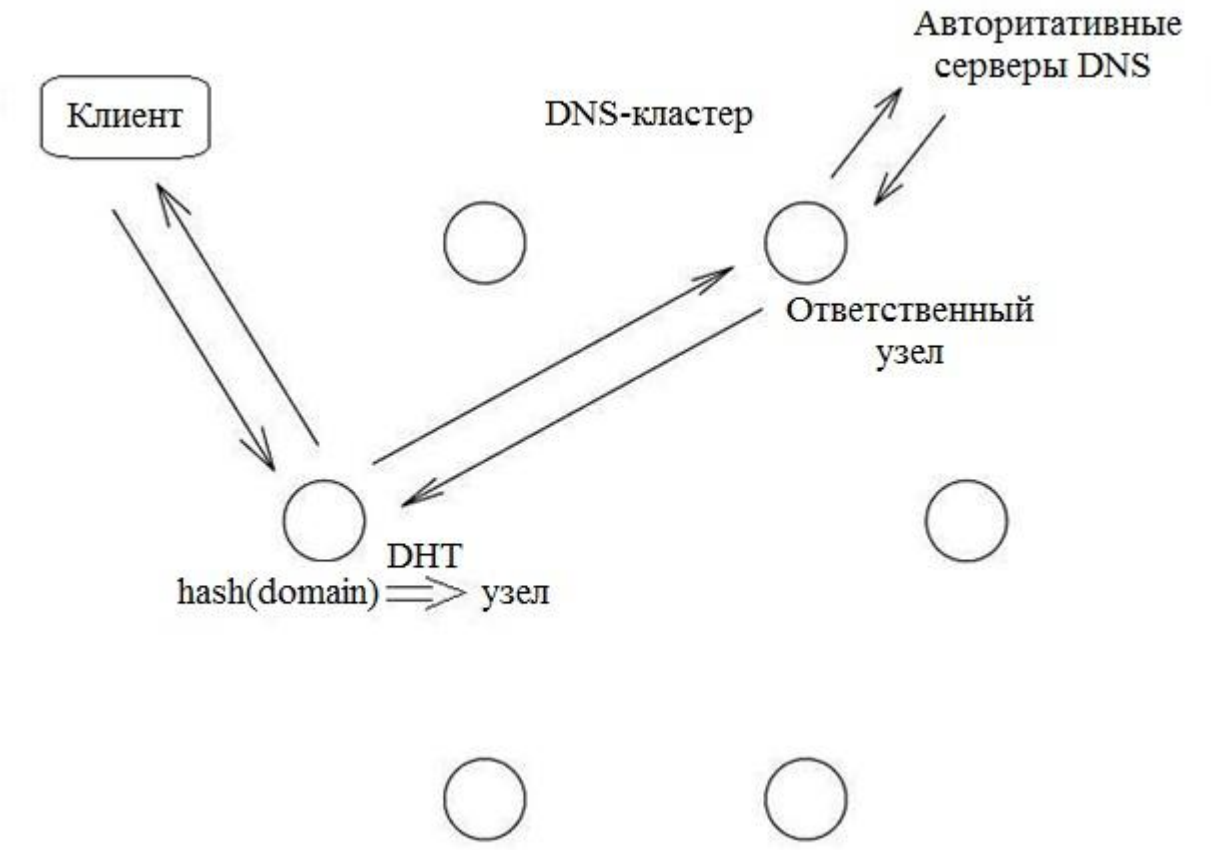

Рисунок 2.17 – Взаимодействие клиента с DNS-кластером, а также внутрикластерный обмен информацией при полном составе кластера

В случае выхода какого-либо узла из состава кластера запросы с доменами, за которые он был ответственным, начинают равномерно перераспределяться среди оставшихся участников кластера, что иллюстрирует рисунок 2.18.

Как было описано выше, при выходе узла области его ответственности равномерно перераспределяются среди «живых» узлов. В соответствии с этим начинают переадресовываться входящие запросы для поддержания равномерной загрузки узлов кластера.

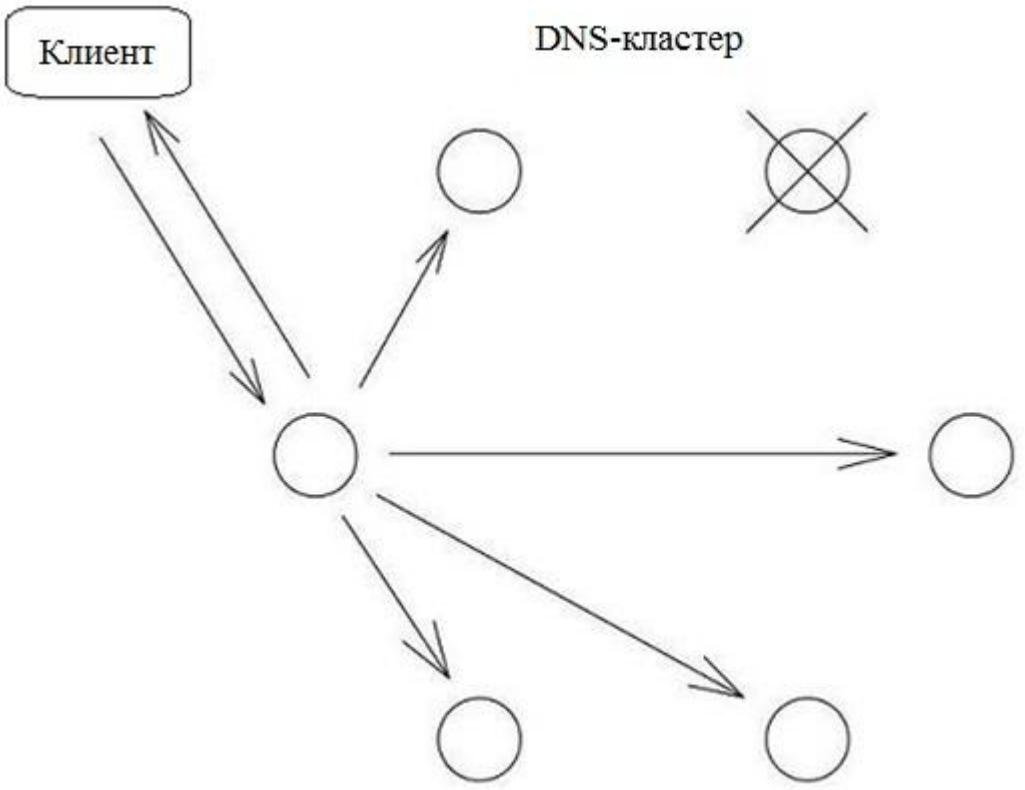

Рисунок 2.18 – Перераспределение входящих запросов с доменами, за которые был ответственен вышедший из состава кластера узел

На рисунке 2.19 в виде диаграммы последовательности изображен процесс обновления внутренних структур системы балансировки нагрузки (таблиц вариантов распределения, необходимого контекста и т.д.) при изменении состава кластера. Здесь представлены три объекта:

- одна из нитей сервера (*Thread*), обрабатывающая входящие запросы;
- некоторая система слежения за составом кластера (назовем ее *Cluster Membership Observer, CMO*);
- система балансировки нагрузки (*DHT*).

Перед обработкой каждого нового запроса *Thread* опрашивает систему слежения за составом кластера на предмет изменений в ее локальной таблице узлов (вызов *is_host_table_changed()* на рисунке 2.19). При получении положительного ответа *Thread* запрашивает от *CMO* обновленную таблицу узлов, соответствующую текущему составу кластера (вызов *get_host_table()*). На следующем шаге *Thread* производит конвертацию таблицы, полученной от *CMO*, в

формат таблицы узлов, использующийся в *DHT*. После этого *Thread* инициирует обновление внутренних структур *DHT*, предоставляя актуальную таблицу узлов в соответствующем формате. В процессе обновления происходит перестроение таблиц вариантов распределения, пересчет различных параметров, необходимых для корректной работы системы, и т.д. После завершения процесса обновления *DHT* готова к продолжению работы.

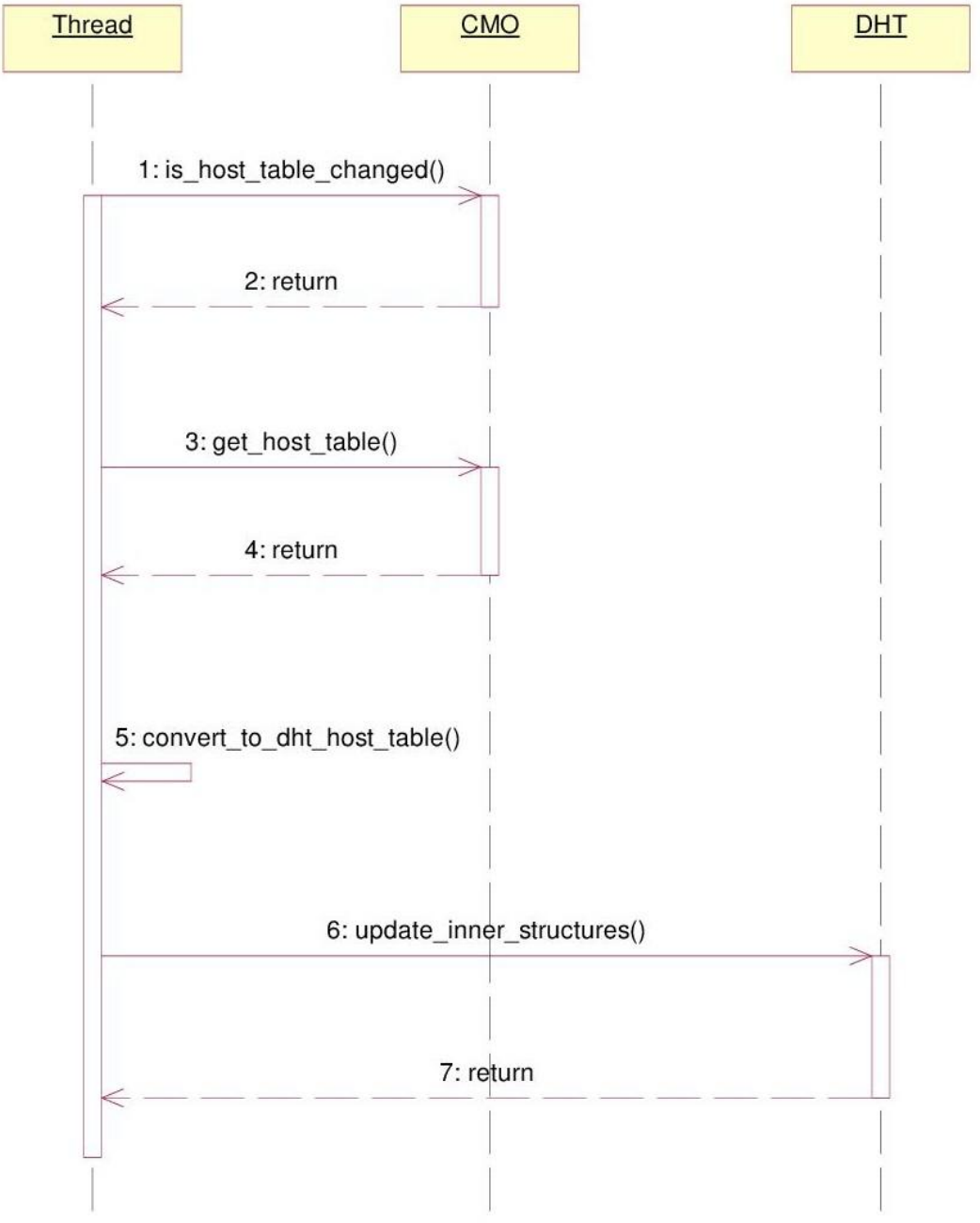

Рисунок 2.19 – Процедура обновления внутренних структур системы балансировки нагрузки при изменении состава кластера

То, что функционал по опросу системы слежения за составом кластера и конвертации таблиц должно реализовывать конкретное приложение, использующее рассматриваемый комплекс программ балансировки нагрузки, имеет принципиальное значение. Подобная архитектура позволяет «отвязать» систему балансировки нагрузки от системы слежения за составом кластера, т.е. сделать их полностью независимыми. Это приводит к возможности использования рассматриваемого комплекса программ балансировки нагрузки в сочетании с различными системами слежения за составом кластерной системы, используемыми в зависимости от конкретной задачи.

Общая схема взаимодействия клиента и узлов кластера представлена на рисунке 2.20 также в виде диаграммы последовательности, где представлена ситуация с переадресацией запроса.

Здесь на первом шаге клиент посылает рекурсивный запрос на разрешение домена одному из узлов кластера (*Node i*). После принятия запроса данный узел первым делом рассчитывает хеш-значение для входящего домена, после чего обращается к своей системе балансировки нагрузки с целью определения ответственного за пришедший домен узла. На рисунке 2.20 ответственным оказывается узел, обозначенный как *Node j*. Далее следует переадресация запроса ответственному узлу.

Приняв переадресованный запрос, *Node j* производит процедуру разрешения домена (поиск в кэше и обращение при необходимости к авторитативным серверам), после чего возвращает результат узлу *Node i*, который, в свою очередь, отправляет итоговый ответ клиенту.

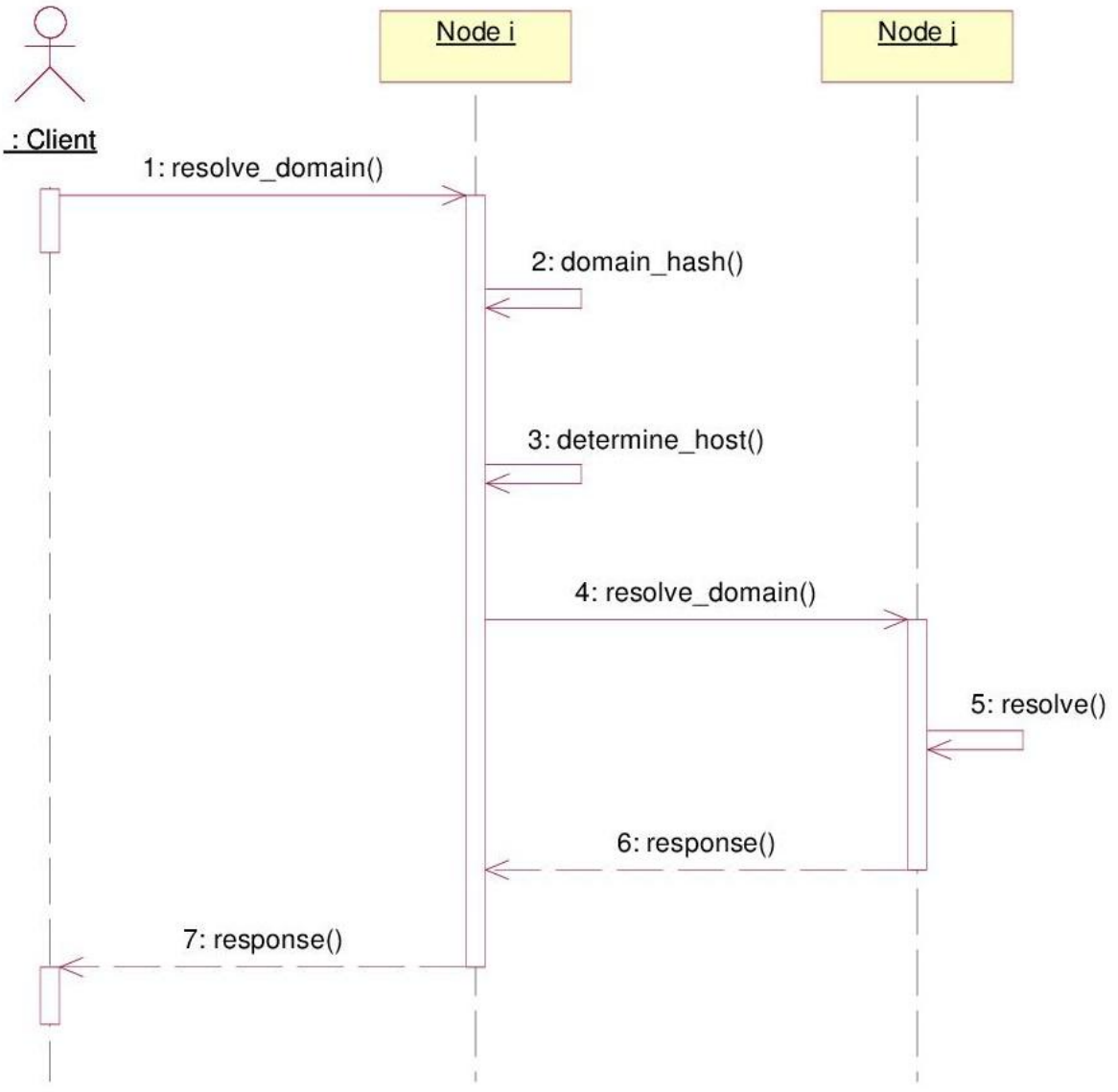

Рисунок 2.20 – Взаимодействие клиента и узлов кластера

При этом узлам необходимо отличать запрос, пришедший от клиента, от переадресованного запроса, чтобы избежать повторных обращений к системе балансировки нагрузки. Самыми распространенными здесь являются два варианта: различие по интерфейсу и различие по входящему порту. Различие по интерфейсу выглядит менее предпочтительным, так как, обычно, количество интерфейсов существенно уступает количеству свободных портов.

2.11 Выводы

Для рекурсивных кэширующих DNS-серверов размер и содержание локального кэша имеет ключевое значение для их скорости работы. Объединение подобных серверов в кластер позволяет получить «распределенный кэш», размер которого равен суммарному размеру всех локальных кэшей участвующих серверов. Распределенный кэш является одним из основных преимуществ (помимо масштабируемости, высокой доступности сервиса и экономической эффективности), которое дает кластеризация в данном случае.

Наличие множества узлов требует соответствующих механизмов балансировки нагрузки для обеспечения эффективности функционирования DNS-кластера. При этом должны учитываться особенности DNS-сервиса, такие как кэширование DNS-записей и скорость ответа, откуда вытекают высокие требования к быстродействию системы балансировки нагрузки.

Так как кластер является динамической системой, в которой состав участников может со временем изменяться, то особое внимание должно быть уделено масштабируемости кластера.

В рамках данной работы были разработаны методы и алгоритмы балансировки нагрузки в DNS-кластере, соответствующие вышеуказанным требованиям, реализованные в виде комплекса программ. Данные методы основаны на технологии распределенной хеш-таблицы (DHT) в сочетании с принципами консистентного хеширования и концепцией виртуального представления узла. Оригинальный метод разбиения пространства ключей на области ответственности позволил обеспечить масштабируемость системы и высокие показатели равномерности распределения нагрузки (детально данный аспект будет рассмотрен в последней главе).

В результате были разработаны одноуровневая и двухуровневая модели организации таблиц вариантов распределения.

Одноуровневая модель показала высокие результаты в области быстродействия, равномерности распределения нагрузки, а также в ситуации добавления новых узлов в систему.

Однако дальнейший анализ показал, что одноуровневая модель имеет определенные недостатки в обработке ситуации временного выхода узлов из состава кластера, что проявлялось в некотором смещении областей ответственности оставшихся узлов и появлении в связи с этим эффекта миграции ключей. Подобное поведение приводило к тому, что часть закэшированных записей становилось неактуальной.

Решением данной проблемы явилась двухуровневая модель организации таблиц распределения. Применение данной схемы позволило исключить эффект смещения областей ответственности при временных выходах узлов из состава кластера. Отсутствие миграции ключей при временных выходах узлов позволяет полностью сохранить актуальность кэшей остающихся участников системы.

Таким образом, двухуровневая модель позволяет обеспечить более корректное поведение системы при изменении состава ее участников. При этом двухуровневая модель обладает всеми достоинствами одноуровневой модели:

- высокие показатели равномерности распределения нагрузки;
- масштабируемость системы;
- быстродействие (непосредственно вычисление ответственного узла составляет несколько арифметических операций и условных переходов);
- высокая надежность;
- гибкость поведения при изменении состава участников кластера.

Высокие показатели быстродействия достигнуты за счет того, что все накладные расходы перенесены в процесс подготовки необходимой инфраструктуры, который инициируется в случае изменения состава участников кластера, что является достаточно редким событием по отношению к частоте прихода запросов. Также в процессе функционирования системы полностью исключена работа с жестким диском, имеющая свойство вносить неконтролируемые задержки. Данное свойство достигнуто за счет того, что все необходимые данные строятся и содержатся в оперативной памяти.

Необходимо отметить, что временная сложность алгоритма вычисления ответственного узла не зависит от количества узлов кластера и определяется как

$O(C)$. Таким образом, алгоритм будет одинаково быстро работать при десятках и тысячах участвующих узлов. Этот факт позволяет заключить, что данный алгоритм никаким образом не ограничивает масштабируемость кластерной системы.

В отличие от алгоритма вычисления ответственного узла, алгоритм построения таблиц вариантов распределения зависит от количества узлов в системе и определяется как $O(N\log(N))$, где N – количество узлов кластера.

Описанные в данной главе эксперименты показали, что для подавляющего числа задач система, управляемая рассматриваемым программным комплексом, может быть масштабируема до 10000 узлов, при этом в максимальном случае время построения таблиц оценивается как ~0.5 сек. Также для определенного круга задач возможно масштабирование до 100000 узлов, где максимальное время работы алгоритма занимает ~7 сек. Маловероятным по удовлетворительности, однако потенциально возможным является масштабирование до 1000000 узлов, в этом случае время работы достигает значения ~1.5 мин.

Алгоритм поиска ответственного узла работает для каждого входящего запроса, в то время как процесс перестроения таблиц вариантов распределения инициируется при изменении состава участников кластера. Таким образом, соотношение показателей временной сложности алгоритмов построения таблиц вариантов распределения и поиска ответственного узла показывает, что наиболее эффективно описываемые методы и алгоритмы, а также соответствующая их программная реализация, могут быть применены в рамках кластерных систем, состав участников которых является относительно стабильным.

Особого внимания достоин тот факт, что реализованный программный комплекс может быть характеризован как «*zero-hop*» DHT, т.е. распределенная хеш-таблица, в которой каждый из узлов локально содержит достаточное количество информации о маршрутизации, чтобы непосредственно переадресовывать запрос необходимому узлу системы. Данное свойство позволяет минимизировать задержки на переадресацию входящего запроса по сети.

Глава 3. Алгоритмы репликации DNS-записей в рамках комплекса программ балансировки нагрузки

3.1 Репликация в вычислительной технике

При реализации кластерных систем крайне актуальной проблемой является сохранение работоспособности системы при временных (по причине сетевого сбоя, технического обслуживания, обновления программного обеспечения, различного рода проблемах в работе сервера и т.д.) или постоянных выходах узлов из состава кластера.

Решением данной проблемы является репликация.

В вычислительной технике репликация включает в себя обмен информацией с целью обеспечения согласованности между избыточными ресурсами для повышения надежности системы, ее отказоустойчивости и высокой доступности.

Различают следующие виды репликации:

- репликация данных, когда идентичные данные хранятся на множестве различных устройств хранения;
- вычислительная репликация, когда одна и та же вычислительная задача выполняется множество раз.

Обычно, вычислительная задача является либо реплицированной «в пространстве» (иначе говоря, исполняемой на различных устройствах), либо реплицированной «во времени», т.е. исполняемой повторно на одном устройстве. Вычислительную репликацию «в пространстве» или «во времени» часто связывают с алгоритмами планирования выполнения задач [18].

Как правило, доступ к реплицированным данным полностью идентичен доступу к единичным нереплицированным данным. Сам же процесс репликации должен быть абсолютно прозрачным для внешнего пользователя. Также, в случае отказа каких-либо компонентов системы, обеспечение отказоустойчивости всей системы за счет наличия реплицированных данных должен быть скрытым

настолько, насколько это возможно, что относится к репликации данных в целях обеспечения качества предоставления сервиса (*Quality of Service*, *QoS*) [19].

Для повышения отказоустойчивости в кластерных системах применяется две стратегии репликации:

1) активная: производится за счет обработки одного и того же клиентского запроса на каждом из узлов системы;
2) пассивная: подразумевает обработку клиентского запроса на одном узле, после чего результат обработки распространяется на другие узлы системы.

Примеры активной и пассивной репликации при взаимодействии клиента с кластерной системой представлены на рисунке 3.1 и рисунке 3.2 соответственно.

Рисунок 3.1 – Активная репликация: клиентский запрос обрабатывается каждым узлом системы

Рисунок 3.2 – Пассивная репликация: клиентский запрос обрабатывается на одном узле, результат распространяется на другие узлы системы

Также существует разделение типов репликации по времени. Здесь различают следующие типы репликации:

- синхронная репликация: если какие-либо данные обновляются, то их реплицированные копии должны быть обновлены в ходе группы последовательных операций, которая представляет собой логическую единицу работы с данными (транзакции), что требует ожидания подтверждения записи всех реплицированных фрагментов данных;
- асинхронная репликация: процесс записи считается завершенным, как только локальное хранилище это подтверждает, при этом реплицированные данные обновляются с некоторой задержкой. В ходе такого обновления реплицированные данные могут некоторое время находиться в несогласованном состоянии.

Синхронная репликация является более ресурсоемкой операцией по сравнению с асинхронной репликацией. Однако возможность несогласованного состояния реплицированных данных в случае асинхронной репликации существенно ограничивает область применения данного ее типа. Проблема согласованности данных имеет особое значение в такой области, как распределенные файловые системы [20, 21, 22, 23]. В зависимости от конкретной задачи выбирается определенный тип репликации.

3.2 Особенности задачи репликации DNS-записей

В рамках данной работы под репликацией будет пониматься исключительно репликация данных, т.е. процесс создания резервных копий данных на физически различных DNS-серверах.

Таким образом, при наличии соответствующих механизмов, системы, обеспечивающие функционирование кластера как единого целого, позволяют организовать резервное копирование данных, которыми оперирует кластер. Применение подобных технологий существенно повышает надежность и стабильность кластерных систем.

Помимо механизмов распределения нагрузки и обработки входа и выхода узлов из кластера, в рамках описываемого комплекса программ реализована репликация ресурсных записей. Наличие этого процесса позволяет иметь несколько копий ресурсной записи (резервное копирование), что делает выход ответственного за эту запись узла абсолютно незаметным для пользователя.

DNS обладает следующими характеристиками, имеющими большое значение при решении поставленной задачи:

- кэширование информации: узел может хранить некоторое количество данных не из своей зоны ответственности для уменьшения нагрузки на сеть;
- резервирование: за хранение и обслуживание своих узлов (зон) отвечают (обычно) несколько серверов, разделённые как физически, так и

логически, что обеспечивает сохранность данных и продолжение работы даже в случае сбоя одного из узлов.

Классический вариант обмена информацией между DNS-клиентом и рекурсивным DNS-сервером подразумевает единовременное общение клиента с одним сервером (не будем здесь рассматривать отдельные специфичные реализации DNS-клиентов, позволяющих производить параллельный опрос множества серверов одновременно). Таким образом, алгоритм должен обеспечивать описанную выше пассивную репликацию, где клиентский запрос обрабатывается одним из серверов кластера, после чего данный сервер производит распространение полученных данных среди остальных узлов, формируя таким образом резервные копии на физически различных устройствах.

Основные особенности задачи репликации данных в DNS-кластере состоят в следующем:

- резервное копирование должно производиться на физически различные узлы кластера, в этом случае проблемы с оборудованием (дисковым, сетевым и т.д.) на конкретном узле не повлияют на общую работоспособность системы;
- резервное копирование должно осуществляться в соответствии с алгоритмами системы балансировки нагрузки, управляющей распределением входящих запросов на узлы кластера;
- резервное копирование не должно быть излишне избыточным для экономии ресурсов памяти и снижения нагрузки на локальную сеть, объединяющую узлы.

Также большое внимание должно уделяться сложности самого алгоритма репликации и связанным с ним накладным расходам, т.к. репликация не должна сильно влиять на производительность высоконагруженной системы, которой, в частности, является кластер DNS.

Как говорилось выше, объединение нескольких DNS серверов в единый кластер позволяет получить распределенный кэш, при этом его общий размер является суммой размеров кэшей на каждом отдельном сервере. Наличие

актуальной записи в кэше позволяет избежать обращения к вышестоящим и авторитативным серверам, которое может занимать достаточно большое количество времени. Распределенный кэш является одним из главных преимуществ, которое дает кластеризация в данном случае.

Однако распределенный кэш требует особого механизма обслуживания и поддержки для эффективного использования его содержимого. В особенности это касается масштабируемости кластера, ситуаций входа и выхода узлов.

Алгоритм репликации должен полностью соответствовать алгоритму распределения нагрузки для эффективного использования реплицированных записей в случае выхода узлов из состава кластера. При этом очевидно, что содержание кэша конкретного узла зависит от областей его ответственности.

Так как при выходе узла из состава кластера его области ответственности перераспределяются среди оставшихся активных узлов, то возникает ситуация, когда узлы начинают получать новые для себя домены, которые они не обслуживали ранее. Соответственно, эти домены отсутствуют в локальных кэшах оставшихся узлов, что заставляет их чаще обращаться к авторитативным серверам. Это приводит к временному снижению производительности кластера, ему требуется некоторое время на стабилизацию работы.

Чтобы избежать подобной ситуации, был реализован алгоритм репликации ресурсных записей, позволяющий хранить несколько копий записи на различных узлах кластера, причем выбор узлов для репликации соответствует алгоритму распределения запросов. Алгоритм репликации должен «предугадывать», какие узлы будут обслуживать данный домен при выходе ответственного узла, чтобы репликация была эффективной и не излишне избыточной.

В результате анализа проблемы, проектирования и разработки алгоритмов было реализовано две версии алгоритма репликации. Рассмотрим подробно оба решения.

3.3 Алгоритм репликации ресурсных записей на основе взаимного перекрытия областей ответственности узлов

Первый вариант алгоритма репликации основывается на принципе пересечения областей ответственности узлов. Основная идея состоит во взаимном перекрытии отрезков ответственности, в соответствии с которой определяется не только узел, отвечающий за данный участок пространства ключей, но и узел, чей отрезок имеет с ним зону перекрытия [24]. При этом первому из них будет послан запрос на разрешение домена, а второму – специальный запрос от первичного узла после разрешения домена, содержащий соответствующие ресурсные записи, получив который узел только выполнит кэширование данных.

Размер области перекрытия является конфигурируемым параметром, который задается в процентном соотношении перекрытия. Значение перекрытия задается при старте системы, а также может быть переопределено уже во время ее функционирования, что существенно добавляет гибкости управлению системой.

На рисунке 3.3 изображены три варианта перекрытия областей для узла с идентификатором 1 в пределах одного блока с определенным распределением узлов: 0% перекрытия (без репликации), 50% и 100% перекрытия. Здесь при попадании ключа в заштрихованные области запрос на разрешение домена будет послан узлу 1, а соответствующему перекрывающему узлу (с идентификатором 2 или 0) будет послан специальный запрос на кэширование данных. При попадании ключа в незаштрихованную область ответственности узла 1, будет послан только один запрос на разрешение домена, репликация в этом случае производится не будет. Как видно на рисунке 3.3, при 100% перекрытии репликация будет производиться при поступлении каждого запроса. В изображенном блоке ключи из области ответственности узла 1 будут реплицироваться узлам с идентификаторами 2 и 0, но так как в других блоках будут иметь место различные варианты распределения узлов, то и реплицироваться ключи из областей узла 1 будут различным узлам системы. В этом смысле также присутствует равномерность распределения нагрузки на узлы кластера.

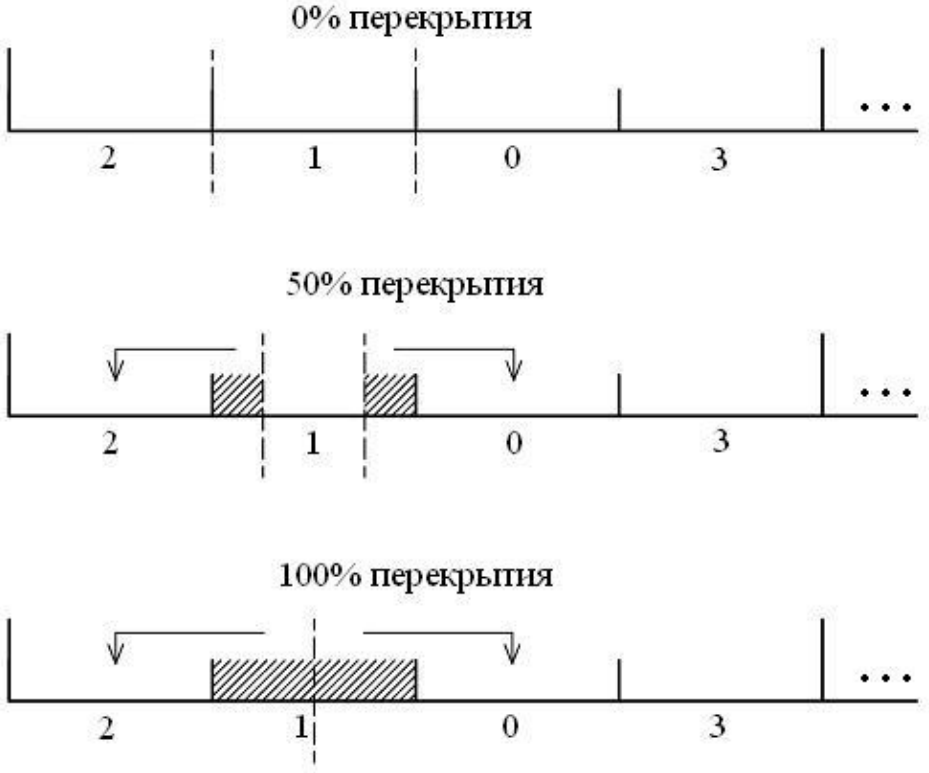

Рисунок 3.3 – Репликация записей на основе взаимного перекрытия областей ответственности узлов

Представим пошагово рассмотренный алгоритм (перед началом работы алгоритма репликации мы уже имеем соответствующее хеш-значение для домена, посчитанное системой распределения нагрузки):

1. На первом шаге по хеш-значению определяется номер блока и его идентификатор от 0 до 511 (по количеству вариантов распределения).
2. Определяется ответственный узел и положение ключа в данном блоке как остаток от деления хеш-значения на размер блока.
3. По положению ключа в блоке рассчитывается его положение уже относительно границ области ответственности узла как остаток от деления на размер области ответственности.
4. Далее производится проверка, попадает ли ключ, положение которого было найдено на предыдущем шаге, в заданные области перекрытия справа и слева.

5. В случае попадания в область перекрытия соответствующий соседний узел выбирается для репликации ресурсных записей запрашиваемого домена, иначе репликация производиться не будет.

Блок-схема описываемого алгоритма представлена на рисунке 3.4.

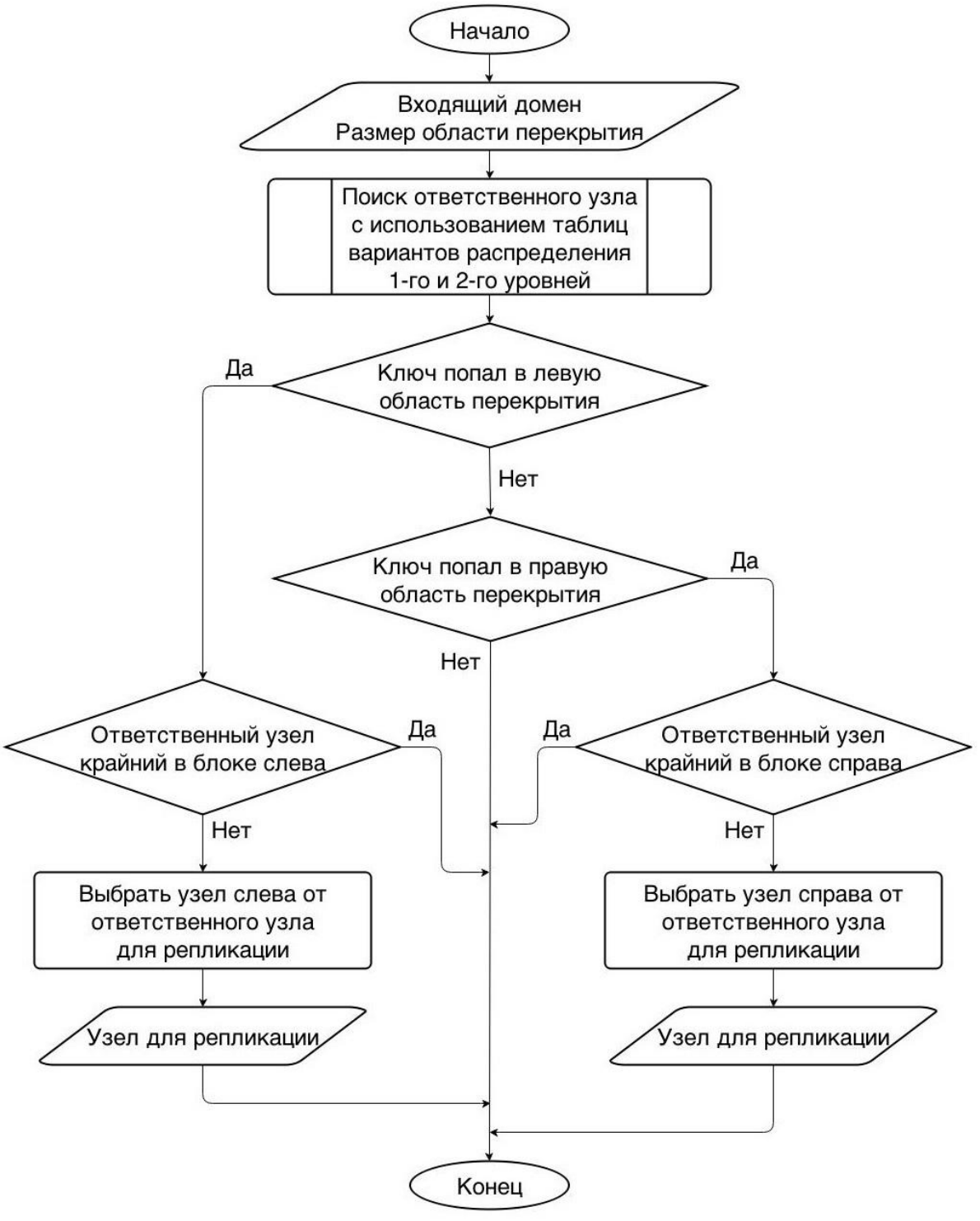

Рисунок 3.4 – Блок-схема алгоритма репликации ресурсных записей на основе взаимного перекрытия областей ответственности узлов

Таким образом, при выходе узла из кластера перекрывающие его область ответственности узлы уже будут частично содержать его кэш, что положительно скажется на производительности кластера.

Как и в исследовательском проекте DDNS, описанном в работе R. Cox, A. Muthitacharoen, R. T. Morris [13], единицей репликации здесь является набор ресурсных записей, соответствующих входящему домену и типу запроса.

Однако такое решение имеет определенный недостаток. В случае, когда процент перекрытия пространств меньше 100%, невозможно заранее предугадать, какие домены будут реплицироваться, а какие нет. Это вносит неясность в поведение репликационного механизма, оно становится несколько непредсказуемым.

Также при таком подходе мы не можем получить больше, чем одну копию данных. Несколько копий на различных узлах не поддерживаются в рамках данного алгоритма.

3.4 Алгоритм репликации ресурсных записей на основе использования ближайших областей ответственности узлов

В результате усовершенствования описанного в предыдущем пункте решения, был реализован второй вариант алгоритма репликации, в котором удалось избавиться от недостатков первого.

Основная идея состоит в том, чтобы всегда производить репликацию на четное количество узлов, которые потенциально могут стать ответственными за данный домен [24]. В самом простом случае репликация будет производиться на два соседних узла. Стоит заметить, что, в соответствии с алгоритмом распределения нагрузки, репликация должна производиться только в пределах блока. Т.е. если ответственный узел находится в блоке на крайней позиции (первым или последним), то для репликации будут выбраны последующие (или предыдущие) узлы.

Распределенная система хранения данных Dynamo от Amazon, где также используется принцип согласованного хеширования (consistent hashing), имеет несколько похожий алгоритм репликации [8]. Однако рассматриваемый репликационный механизм реализован с учетом принципов, лежащих в основе реализованного комплекса алгоритмов распределения нагрузки, вследствие чего имеют место существенные отличия от алгоритма репликации системы Dynamo.

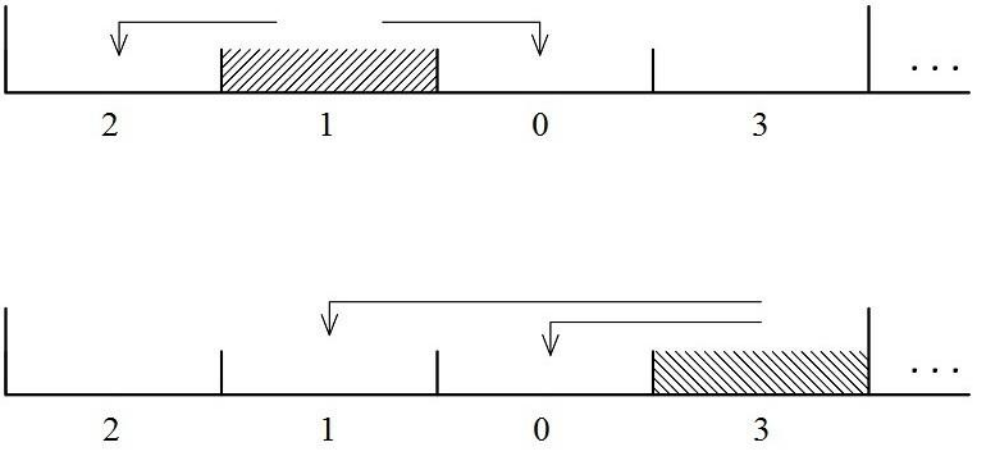

Рисунок 3.5 – Репликация записей с двумя резервными копиями для кластера из четырех узлов

На рисунке 3.5 изображены два случая репликации с двумя резервными копиями для четырех узлов кластера. Заштрихованная область – область ответственности узла, в которую попадает запрашиваемый домен. Стрелками указаны узлы, которые будут выбраны для репликации записей запрашиваемого домена. В первом случае (верхнее изображение) домен попал в область ответственности узла 1, репликация будет производиться на соседние для него узлы 2 и 0. Во втором случае (нижнее изображение) запрашиваемый домен попал в область узла 3, который является крайним в рассматриваемом блоке. Следовательно, репликация в данном случае будет производиться на два предыдущих узла 0 и 1.

Отметим принципиальную важность четности количества узлов для репликации. В случае, когда узлами для репликации выбираются соседние узлы по отношению к ответственному, достаточно непросто заранее предугадать, какой

именно из узлов будет обслуживать конкретный реплицированный домен после выхода ответственного узла.

Представим конкретную ситуацию на примере рассмотренного выше блока. Пусть ответственным для доменов example1.com и example2.com является узел 1. Соответственно, example1.com и example2.com будут реплицированы на узлы 2 и 0. Однако при выходе узла 1 из состава кластера домены example1.com и example2.com уже будут обслуживаться различными узлами в соответствии с их областями ответственности (рисунок 3.6).

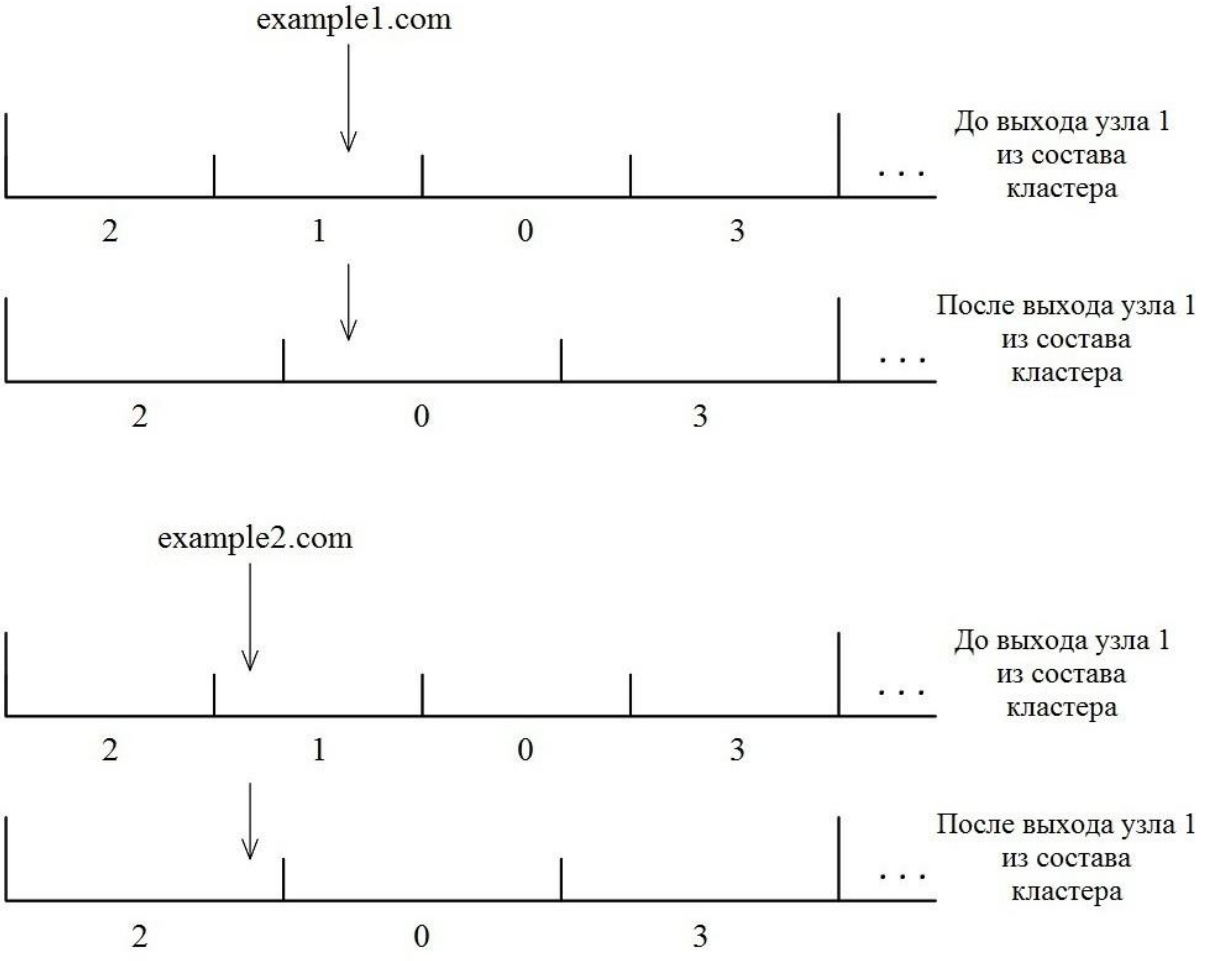

Рисунок 3.6 – Обслуживание доменов example1.com и example2.com до и после выхода узла 1 из состава кластера

Как видно на рисунке 3.6, после выхода узла 1 домен example1.com будет обслуживаться узлом 0, а домен example2.com узлом 2. Для того чтобы до выхода узла 1 определить конкретный узел, который будет обслуживать example1.com или example2.com, и произвести репликацию только на него, необходимо вычислить

будущую границу между областями ответственности узлов 2 и 0. Причем такое вычисление придется делать при каждом входящем запросе, что отрицательно скажется на производительности алгоритма балансировки нагрузки. Так как быстродействие является здесь важнейшим показателем, предлагается ввести некоторую избыточность репликации и производить ее на четное количество узлов.

Однако здесь нельзя говорить об избыточности в полной мере, так как при одновременном выходе из кластера узлов 0 и 1 оба домена (example1.com и example2.com) будут обслуживаться узлом 2, который уже будет содержать эти домены в кэше благодаря репликации (несмотря на то, что до выхода узлов 0 и 1 репликация домена example1.com на узел 2 могла показаться избыточной).

Также алгоритм позволяет задавать уровень покрытия репликации, т.е. число резервных копий: 2, 4, 6 и т.д. На рисунке 3.7 представлен пример с четырьмя резервными копиями в случае кластера из семи узлов.

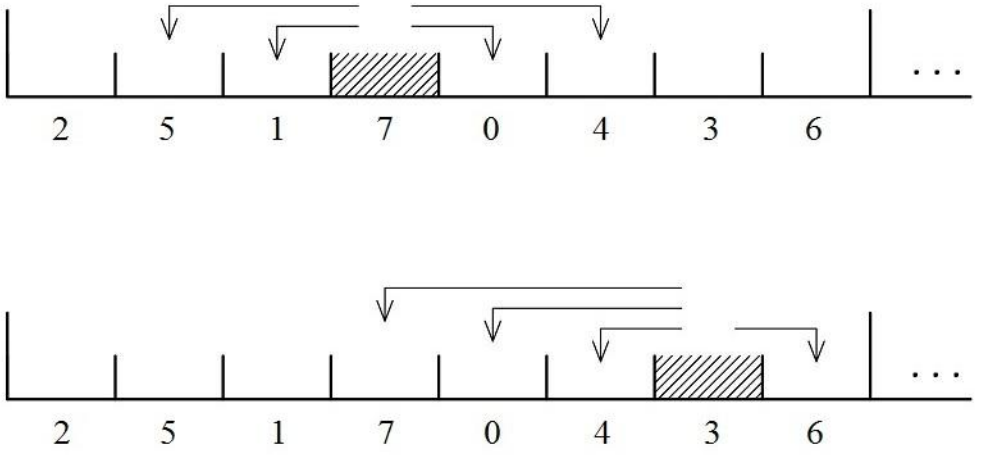

Рисунок 3.7 – Репликация записей с четырьмя резервными копиями для кластера из восьми узлов

Таким образом, предложенный алгоритм позволяет задавать максимальный уровень репликации с полным покрытием, при котором каждый узел будет содержать полную копию ресурсных записей других узлов кластера.

Рассмотрим алгоритм пошагово (как и в первом варианте перед началом работы алгоритма мы уже имеем хеш-значение для домена):

1. По рассчитанному хеш-значению определяется номер блока и его идентификатор от 0 до 511.
2. Определяется ответственный узел на основании идентификатора блока, хеш-значения и размера областей ответственности узлов.
3. По таблице распределения выбирается указанное количество соседних узлов внутри заданного блока, стоящих последовательно справа и слева от ответственного узла, с учетом возможных описанных выше краевых условий.
4. На выбранные узлы производится репликация.

Блок-схема описываемого алгоритма представлена на рисунке 3.8.

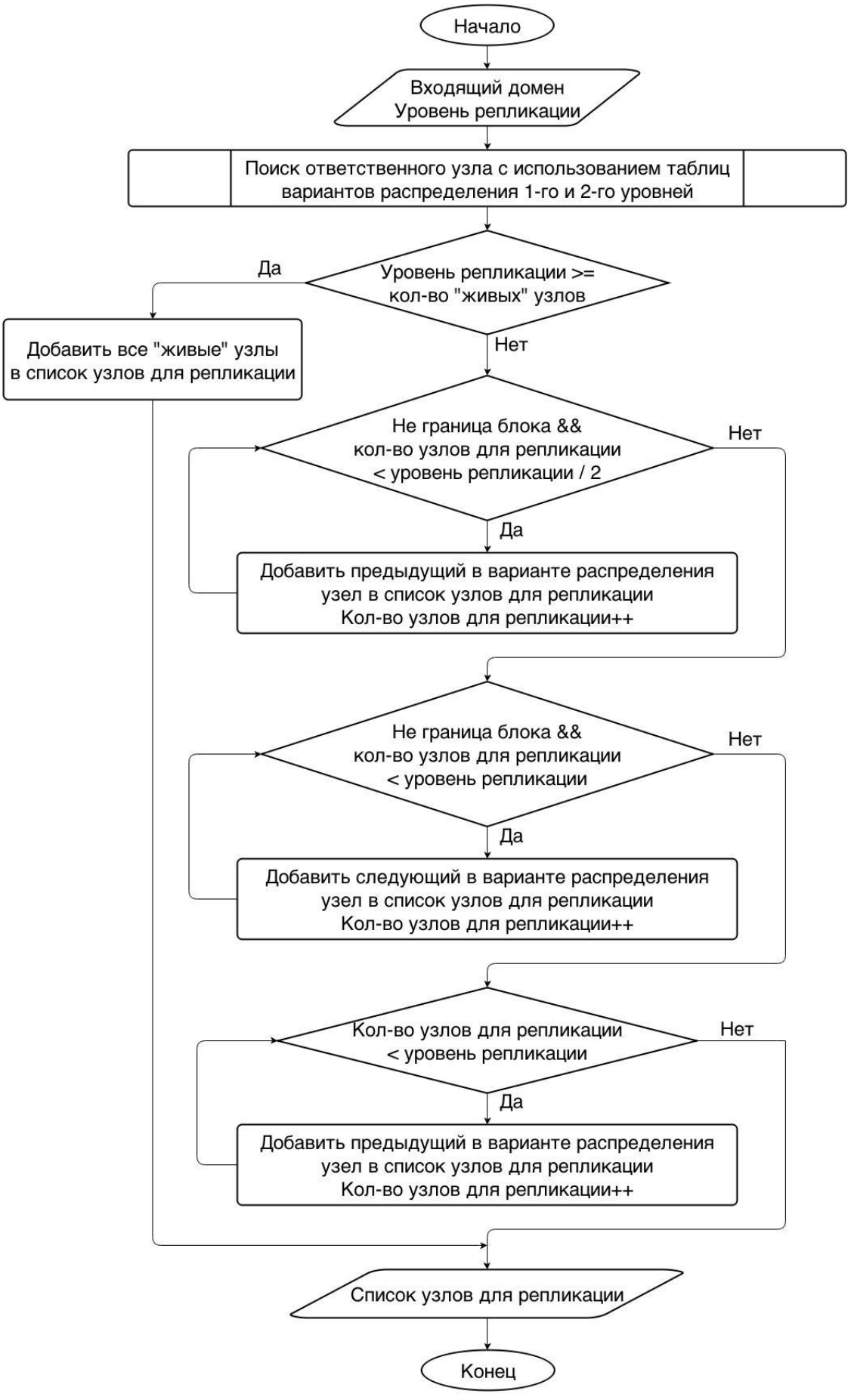

Рисунок 3.8 – Блок-схема алгоритма репликации на основе использования ближайших к ответственному узлу областей ответственности

Как и в первом варианте алгоритма репликации, единицей реплицируемых данных является набор ресурсных записей, соответствующих входящему домену и типу запроса.

Отдельно стоит отметить, что при использовании обоих вариантов алгоритма репликации исключительно от конкретного приложения, использующего данный комплекс алгоритмов распределения нагрузки, зависит то, будет ли непосредственно процесс репликации синхронным или асинхронным. Строго синхронным является алгоритм определения узлов, на которые следует производить резервное копирование.

3.5 Взаимодействие узлов кластера в процессе репликации DNS-записей

Как и в случае описания взаимодействия узлов кластера при переадресации водящего запроса, представленные в данном пункте изображения полностью соответствуют текущей схеме применения рассматриваемого комплекса алгоритмов и программ в проекте DNS-сервиса, реализуемом компанией «Релэкс».

Взаимодействие узлов кластера в процессе репликации DNS-записей представлено в виде диаграммы последовательности на рисунке 3.9, а также в виде схемы на рисунке 3.10. Данные изображения справедливы как для первого, так и для второго варианта алгоритма репликации.

Для упрощения восприятия и соответствия изображений обоим алгоритмам репликации в данном пункте рассматривается случай с двумя резервными копиями.

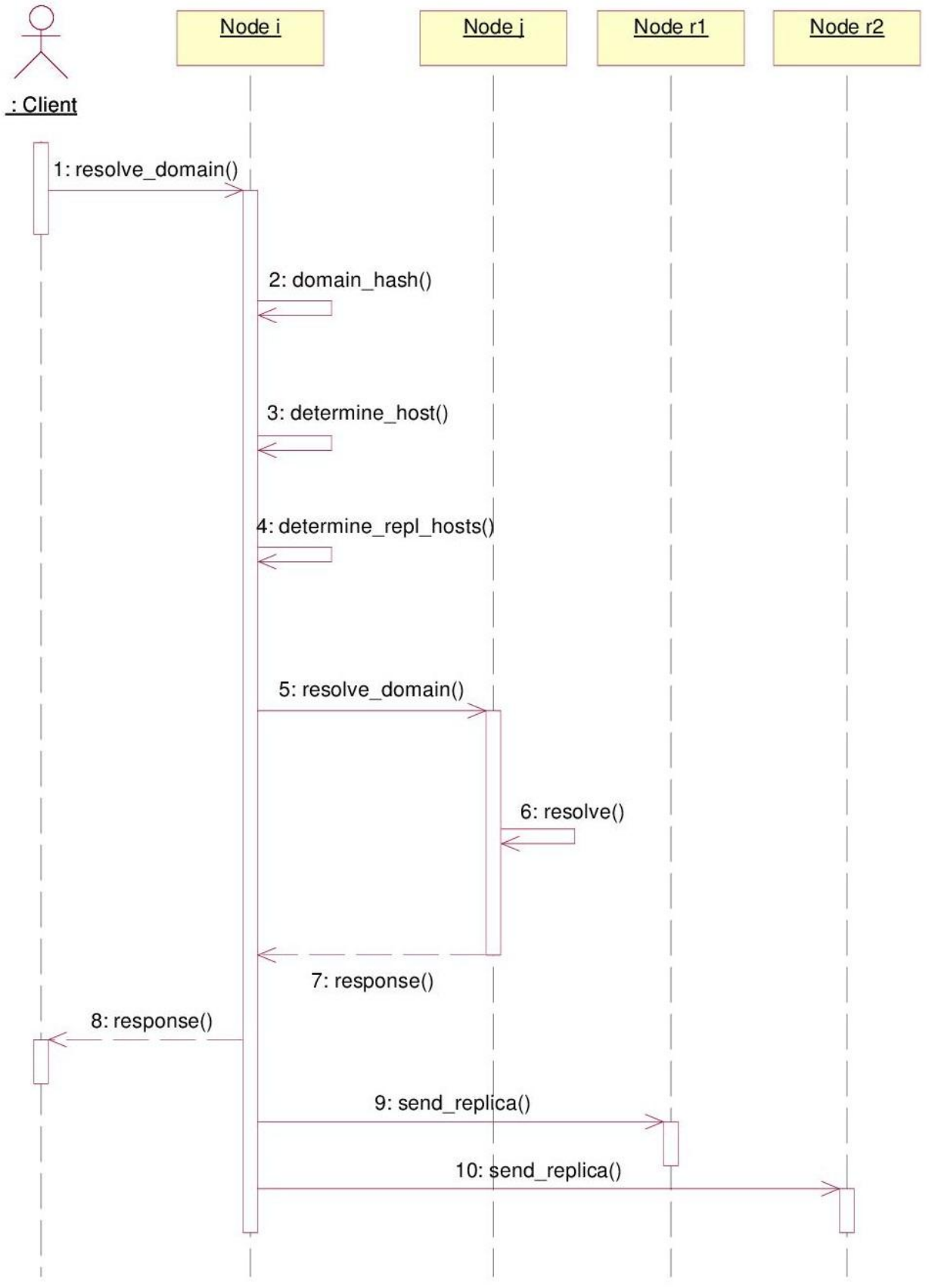

Рисунок 3.9 – Последовательность выполнения операций в процессе обработки входящего запроса с использованием механизма репликации

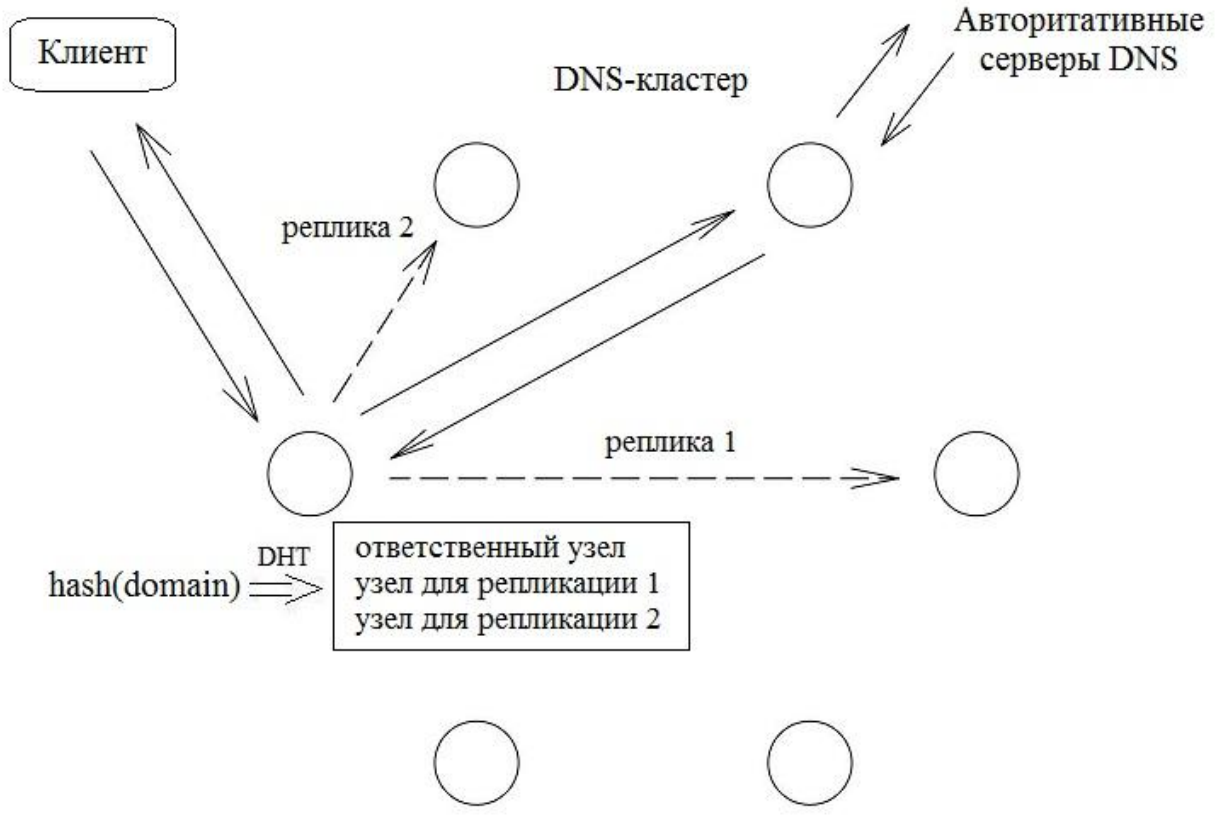

Рисунок 3.10 – Схема обработки входящего запроса с использованием механизма репликации

На рисунке 3.10 принявший запрос узел, используя систему балансировки нагрузки, определяет ответственный узел для домена и два узла для резервного копирования соответствующих ресурсных записей, после чего запрос переадресовывается ответственному узлу.

Получив ответ от ответственного узла, изначально принявший запрос узел формирует результирующий ответ и отправляет его клиенту, после чего данный узел производит построение специального пакета, содержащего домен с соответствующими ресурсными записями, и рассылает его узлам кластера, выбранным для репликации.

3.6 Временная сложность алгоритмов репликации

Первый вариант алгоритма репликации основан на взаимном перекрытии областей ответственности узлов, причем размеры отрезков перекрытия

определяются при изменении состава участников кластера. Время работы алгоритма не зависит как от размеров отрезков перекрытия, так и от количества узлов системы и является константным. Таким образом, временная сложность первого варианта алгоритма репликации определяется как $O(C)$.

Время работы второго варианта алгоритма репликации, основанного на использовании некоторого количества ближайших областей ответственности узлов, зависит от заданного уровня репликации и также не зависит от количества узлов в системе. Т.е. при фиксированном уровне репликации временная сложность является константной и может быть определена как $O(C)$.

Таким образом, оба варианта алгоритма репликации, как и алгоритм поиска ответственного узла в случае одноуровневой и двухуровневой моделей, не ограничивают масштабируемость кластерной системы.

3.7 Структурная схема реализованного программного комплекса

На основе разработанных в рамках данной работы моделей и алгоритмов был реализован программный комплекс. Структурная схема взаимодействия основных модулей комплекса выглядит так, как представлено на рисунке 3.11.

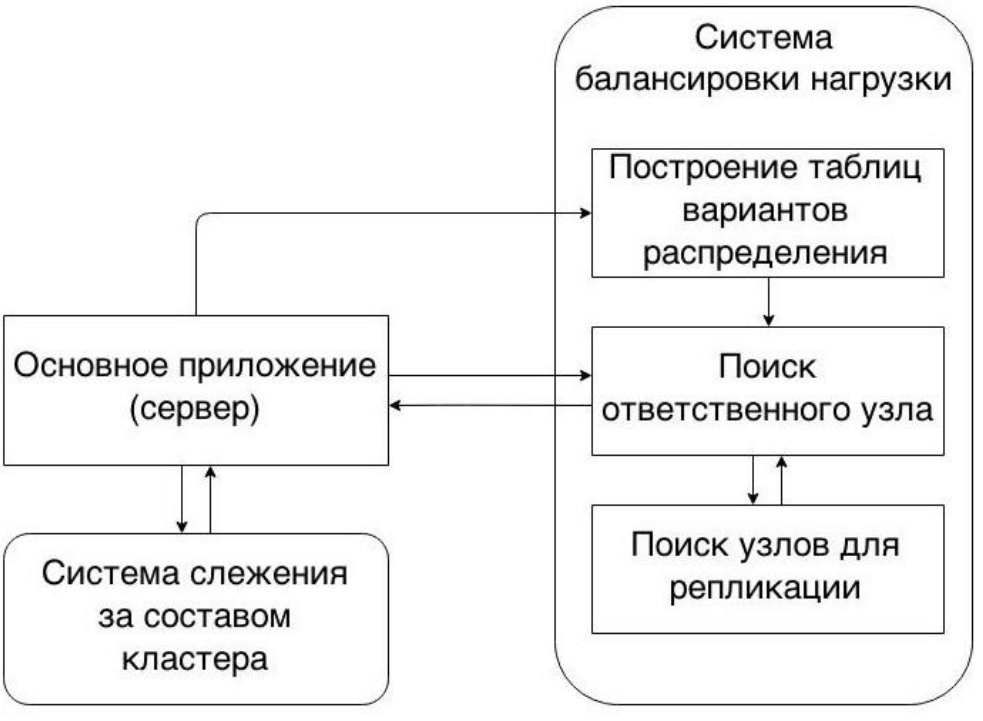

Рисунок 3.11 – Структурная схема взаимодействия основных модулей

В проекте DNS-сервиса, реализуемом компанией «Релэкс» (г. Воронеж), в рамках которого были внедрены результаты диссертации в качестве комплексной системы балансировки нагрузки и репликации, роль основного приложения выполняет рекурсивный кеширующий DNS-сервер.

Так как к системе балансировки нагрузки предъявлялись высокие требования по быстродействию и уровню производительности, то для ее реализации в качестве языка программирования был выбран язык C [44, 45, 46, 47, 48]. Программный комплекс балансировки нагрузки способен функционировать на семействах операционных систем Linux [51, 52, 53, 54, 55] и Windows [56, 57].

3.8 Выводы

Анализ показывает, что второй рассмотренный вариант с репликацией на четное количество узлов выглядит гораздо более предпочтительным, чем первый, где репликация производится на основе перекрытия участков ответственности. Второй вариант гораздо более предсказуем и управляем. Также он позволяет иметь множество копий ресурсных записей, вплоть до резервного копирования на все узлы кластера. К настоящему времени именно это вариант был успешно реализован автором данной диссертации для ряда DNS кластеров, расположенных в Европе и США. Применение данного принципа репликации позволило существенно повысить стабильность работы кластера при плановом обслуживании, обновлении программного обеспечения, сбоях в работе серверов и т.д. Благодаря репликации выход какого-либо узла из состава кластера стал абсолютно незаметным для пользователя.

Как было указано выше, второй вариант реализации имеет много общего с механизмом репликации системы Dynamo от Amazon [8]. Однако существуют свои особенности, связанные с конкретной реализацией комплекса алгоритмов распределения нагрузки на узлы кластера. Произвести полноценное сравнение этих систем на данный момент практически невозможно в связи с тем, что Dynamo является закрытым решением. Публично известны лишь общие

принципы, описанные в [8]. Исходя из этой информации, можно заключить, что обе системы опираются на взаимное расположение отрезков ответственности узлов на области значений используемой хеш-функции при выборе узлов для репликации. Разница состоит в том, что Dynamo выбирает N-1 узел по часовой стрелке (область значений хеш-функции в этом случае для удобства представляется в виде кольца, где нулевое значение «соединяется» с максимальным) [8]. Таким образом, обе системы способны покрыть репликацией произвольное количество узлов, используя при этом несколько разные алгоритмы, соответствующие сценариям обработки выхода узлов. Данные о производительности Dynamo в части репликации не опубликованы. Однако о рассмотренной реализации можно сказать, что при составлении списка узлов для резервного копирования алгоритм выполняет лишь несколько арифметических действий и смещений в массиве. Набор этих операций исполняется крайне быстро (порядка 1 мкс) [24].

Как для первого, так и для второго варианта алгоритма репликации (при фиксированном уровне репликации) временная сложность является константной и может быть определена как $O(C)$. Таким образом, оба варианта алгоритма репликации не ограничивают масштабируемость кластерной системы.

Тестирование и реальная эксплуатация второго варианта реализации репликационного механизма показали высокие результаты в области надежности и быстродействия.

Однако первый вариант репликационного механизма также может давать преимущества в определенных ситуациях. Например, при существенно ограниченных ресурсах памяти, вследствие чего размеры локальных кэшей могут быть сильно лимитированы, и репликация на произвольное четное количество узлов может стать слишком «дорогой». В таком случае подобная «частичная» репликация поможет повысить стабильность работы кластера при небольших дополнительных затратах памяти на хранение некоторого процента реплицированных ресурсных записей.

Глава 4. Статистические исследования комплекса программ балансировки нагрузки и его верификация

При построении кластерных систем в зависимости от решаемой системой задачи выбирается определенный тип кластера, в соответствии с которым проектируется программное обеспечение, позволяющее кластеру работать единой, согласованной системой.

Традиционно выделяют следующие типы кластеров:

- кластеры высокой доступности (*High Availability Clusters*): создаются для обеспечения высокой доступности сервиса, предоставляемого кластером;
- кластеры распределения нагрузки (*Load Balancing Clusters*): принцип их действия строится на распределении запросов через один или несколько входных узлов, которые перенаправляют их на обработку в остальные, вычислительные узлы;
- вычислительные кластеры (*Computing Clusters*): используются в вычислительных целях, в частности в научных исследованиях.

В зависимости от используемого типа кластера выбираются различные показатели и оценки качества программного комплекса, обеспечивающего его функционирование.

Реализованный комплекс алгоритмов распределения нагрузки одновременно позволяет обеспечить свойства двух типов кластеров: кластера высокой доступности сервиса и кластера распределения нагрузки.

В предыдущих главах было проверено, что система полностью удовлетворяет требованиям масштабируемости (что вытекает из принципов ее реализации) и быстродействию алгоритма вычисления узла для перенаправления входящего запроса (несколько арифметических операций и прямой доступ по индексу, суммарно выполняемых менее микросекунды). Также стоит отметить, что каждый из узлов DNS-кластера является входным для пользователя. Комплекс алгоритмов распределения нагрузки и репликации не предполагает наличие каких-

либо управляющих или координирующих узлов, каждый из узлов является независимым и полнофункциональным. Таким образом, отсутствует единая точка отказа системы, что повышает надежность кластера как кластера высокой доступности.

Однако основным требованием к системе является как можно более равномерное распределение нагрузки (входящих DNS-запросов). Утверждение 1 показывает, что при полном составе кластерной системы суммарно узлы отвечают за равные части области значений базовой хеш-функции, что позволяет сделать предположение о равномерности распределения запросов среди узлов системы. Однако характер реальных данных требует проведения вычислительного эксперимента с их участием для проверки данного предположения. В рамках данной главы будет произведена верификация комплекса в части равномерности распределения нагрузки на узлы кластера, составленного из рекурсивных кэширующих DNS-серверов.

4.1 Динамика статистических показателей с ростом числа уникальных запросов к системе

Т.к. для вычисления ответственного узла в комплексе алгоритмов распределения нагрузки используется хеш-функция, поведение системы во времени носит вероятностный характер [39, 40].

Для начала стоит произвести оценку статистических показателей [58, 59, 60] алгоритмов распределения нагрузки в зависимости от количества входящих запросов, таким образом можно проследить динамику качественных показателей комплекса.

Для этих целей было собрано 7×10^6 уникальных доменов. Для того чтобы статистические оценки были наиболее близкими к реальности, размер тестового кластера был выбран равным пяти узлам. Это количество соответствует среднему размеру DNS-кластеров, используемых в рамках упомянутого выше проекта компании «Релэкс», где были внедрены результаты диссертации, для обслуживания клиентов. При этом были использованы реальные IP-адреса (IP-

адрес узла учитывается при определении его областей ответственности как уникальный идентификатор), назначенные узлам на одном из DNS-кластеров, развернутом в США.

Данной тестовой системе посылалось различное количество уникальных запросов из собранного множества доменов: 50, 100, 500, 1000 и т.д. При этом фиксировалось число попавших на каждый узел запросов и на основе этих данных рассчитывались различные абсолютные и относительные статистические показатели.

Для чистоты эксперимента алгоритм репликации был отключен.

Полученные результаты представлены в таблице 4.1 и таблице 4.2.

Таблица 4.1

Динамика абсолютных статистических показателей
в зависимости от количества входящих DNS запросов

Кол-во запросов	Выбор. среднее	Размах вариа-ции	Среднее линейное отклоне-ние	Средне-квадрати-ческое отклонение
50	10	10	3.2	3.68782
100	20	10	3.6	3.84708
500	100	11	3.6	4
10^3	200	35	8	11.2606
5×10^3	10^3	54	16	18.4391
10^4	2×10^3	99	24.8	32.4654
5×10^4	10^4	144	40	49.8317
10^5	2×10^4	231	85.2	89.8844
5×10^5	10^5	1010	344.8	383.776
10^6	2×10^5	1315	393.2	459.135
2×10^6	4×10^5	1107	338.4	403.461
3×10^6	6×10^5	2011	617.6	722.985
4×10^6	8×10^5	2307	806.8	890.916
5×10^6	10^6	3063	817.2	1038.95
6×10^6	1.2×10^6	2135	862.8	891.708
7×10^6	1.4×10^6	2437	802.8	902.246

Таблица 4.2

Динамика относительных статистических показателей в зависимости от количества входящих DNS запросов

Кол-во запросов	Выбор. среднее	Коэф-т осцил-ляции	Линейный коэф-т вариации	Коэф-т вариации
50	10	1 (100%)	0.32 (32%)	0.3688 (36.88%)
100	20	0.5 (50%)	0.18 (18%)	0.1924 (19.24%)
500	100	0.11 (11%)	0.036 (3.6%)	0.04 (4%)
10^3	200	0.175 (17.5%)	0.04 (4%)	0.0563 (5.63%)
5×10^3	10^3	0.054 (5.4%)	0.016 (1.6%)	0.0184 (1.84%)
10^4	2×10^3	0.0495 (4.95%)	0.0124 (1.24%)	0.0162 (1.62%)
5×10^4	10^4	0.0144 (1.44%)	0.004 (0.4%)	0.0050 (0.50%)
10^5	2×10^4	0.0116 (1.16%)	0.0043 (0.43%)	0.0045 (0.45%)
5×10^5	10^5	0.0101 (1.01%)	0.0034 (0.34%)	0.0038 (0.38%)
10^6	2×10^5	0.0066 (0.66%)	0.0019 (0.19%)	0.0023 (0.23%)
2×10^6	4×10^5	0.0028 (0.28%)	0.0008 (0.08%)	0.0010 (0.10%)
3×10^6	6×10^5	0.0034 (0.34%)	0.0010 (0.10%)	0.0012 (0.12%)
4×10^6	8×10^5	0.0029 (0.29%)	0.0010 (0.10%)	0.0011 (0.11%)
5×10^6	10^6	0.0031 (0.31%)	0.0008 (0.08%)	0.0010 (0.10%)
6×10^6	1.2×10^6	0.0018 (0.18%)	0.0007 (0.07%)	0.0007 (0.07%)
7×10^6	1.4×10^6	0.0017 (0.17%)	0.0006 (0.06%)	0.0006 (0.06%)

Особый интерес в полученных оценках вызывают относительные статистические показатели, такие как коэффициент вариации.

Данные в таблице 4.2 показывают, что коэффициент вариации убывает с ростом числа входящих уникальных запросов. Это означает, что разброс количества получаемых каждым из узлов кластера запросов относительно общего их числа, получаемых кластером в целом, уменьшается с возрастанием числа запросов к системе. Т.е. показатели равномерности распределения улучшаются с течением времени работы системы при наличии входящего потока уникальных доменов. Наглядно данный факт продемонстрирован на рисунке 4.1, где показана зависимость значения коэффициента вариации от количества поступивших в систему запросов (данные взяты из таблицы 4.2).

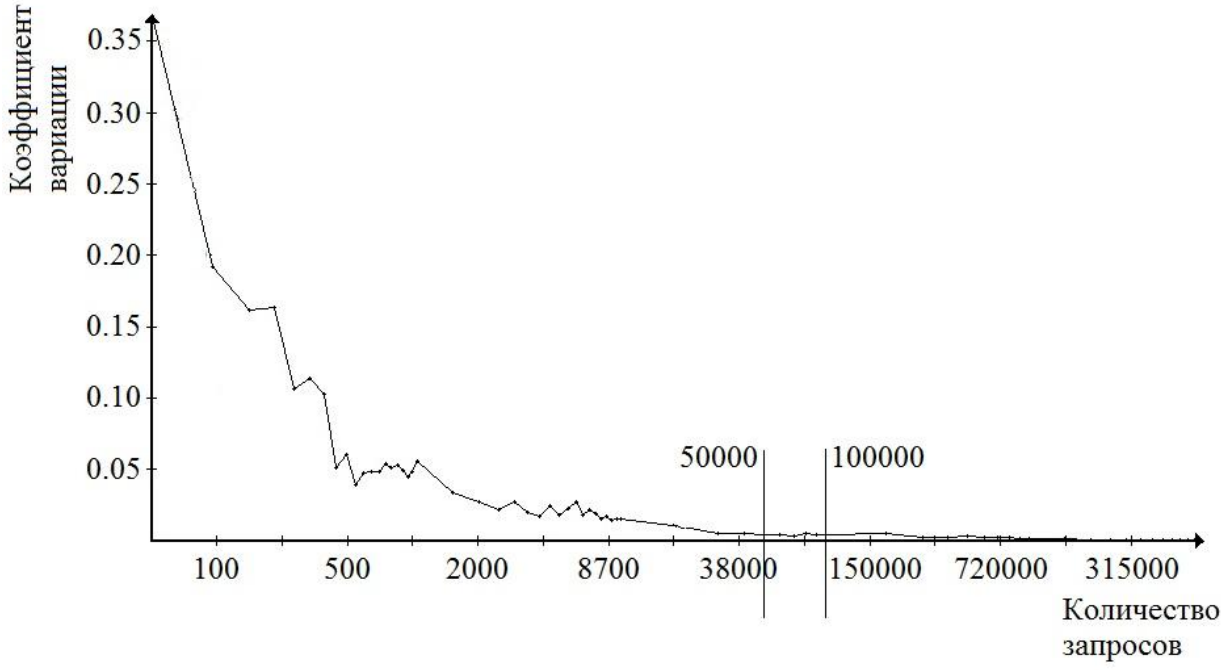

Рисунок 4.1 – Зависимость коэффициента вариации
от количества входящих запросов

На графике, изображенном на рисунке 4.1, присутствуют отсечки на 50000 и 100000 запросов. На данном интервале статистические показатели системы изменяются крайне незначительно (в частности, наблюдается изменение коэффициента вариации в пределах 0.05%). Данный факт помогает выявить достаточный объем выборки для применения критерия согласия Пирсона.

4.2 Проверка соответствия распределения входящих DNS-запросов среди узлов кластера равномерному закону

На следующем этапе необходимо проверить соответствие системы требованию равномерности распределения нагрузки, которое является важнейшим для данной системы. В качестве критерия соответствия данному требованию выберем соответствие числа запросов, получаемых каждым из узлов кластера, равномерному закону распределения. Будем использовать при этом критерий согласия Пирсона (критерий χ^2), который дает возможность оценить степень согласованности теоретического и статистического распределений [25].

Критерий χ^2 К. Пирсона основан на использовании в качестве меры отклонения экспериментальных данных от гипотетического распределения той же величины, которая служит для построения доверительной области для неизвестной плотности, с заменой неизвестных истинных значений вероятностей попадания в интервалы вероятностями, вычисленными по гипотетическому распределению [26]. Формула расчета χ^2:

$$\chi^2 = n\sum_{i=1}^{k}\frac{(p_i' - p_i)^2}{p_i}, \tag{4.1}$$

где k – количество интервалов (категорий), n – объем выборки, p_i' – наблюдаемая частота в i-м интервале, p_i – вероятность попадания изучаемой случайной величины в i-ый интервал, вычисляемая в соответствии с гипотетическим законом распределением.

Для удобства вычислений n можно ввести под знак суммы и, учитывая, что $p_i' = \frac{n_i}{n}$, где n_i – число значений в i-м интервале, привести формулу (30) к виду:

$$\chi^2 = \sum_{i=1}^{k}\frac{(n_i - np_i)^2}{np_i}. \tag{4.2}$$

При проверке соответствия какому-либо закону распределения возникает вопрос достаточности объема статистической выборки, на основе которой

выносится вердикт. В решении данного вопроса нам поможет приведенное выше исследование динамики поведения коэффициента вариации с ростом числа уникальных запросов к системе. Как было указано в предыдущем пункте, данные из таблицы 4.2 показывают (а также отсечки на рисунке 4.1), что при получении системой более 50000 запросов ее статистические показатели изменяются незначительно (наблюдается изменение коэффициента вариации в пределах 0.05%). Таким образом, 100000 входящих запросов может считаться достаточным объемом выборки для оценки соответствия равномерному закону распределения с помощью критерия согласия Пирсона.

Для получения статистических данных, необходимых для применения критерия, системе, включающей 5 узлов, было послано 100000 уникальных запросов. Представим в табличном виде промежуточные результаты алгоритма критерия и итоговое фактическое значение χ^2 (таблица 4.3).

Таблица 4.3

Расчет фактического значения χ^2
для 100000 уникальных запросов

№ узла (i)	Факт. кол-во запросов (n_i)	Теор. кол-во запросов (np_i)	$n_i - np_i$	$\frac{(n_i - np_i)^2}{np_i}$
1	19937	20000	-63	0.19845
2	20128	20000	128	0.8192
3	19953	20000	-47	0.11045
4	19897	20000	-103	0.53045
5	20085	20000	85	0.36125
				χ^2 = 2.0198

Выберем уровень значимости равным 0.05, число степеней свободы 3. Критическое значение χ^2 при уровне значимости 0.05 и числе степеней свободы 3 составляет 7.81. Фактическое значение χ^2 равно 2.0198, что значительно ниже табличного критического значения, из чего можно сделать вывод, что теоретическое равномерное распределение удовлетворительно описывает эмпирические данные.

Данная оценка была получена при использовании уникальных запросов. Однако в реальности поток DNS запросов далеко не уникален, определенные домены запрашиваются пользователями гораздо чаще других (домены поисковых сервисов, новостей, погоды и т.д.).

Для получения более близкой к реальности оценки за двое суток был собран общий DNS трафик компании «Релэкс» в том же объеме 100000 доменов, что использовался в оценке при участии уникального трафика. Далее собранный трафик был распределен среди тех же 5-ти узлов тестового кластера, описанного выше. Полученные значения количества запросов, принятых каждым из узлов, были использованы для расчета значения χ^2 (таблица 4.4).

Таблица 4.4

Расчет фактического значения χ^2
для 100000 запросов из DNS-трафика компании «Релэкс»

№ узла (i)	Факт. кол-во запросов (n_i)	Теор. кол-во запросов (np_i)	$n_i - np_i$	$\frac{(n_i - np_i)^2}{np_i}$
1	19932	20000	-68	0.2312
2	20321	20000	321	5.15205
3	19883	20000	-117	0.68445
4	19990	20000	-10	0.005
5	19874	20000	-126	0.7938
				$\chi^2 = 6.8665$

Для данной выборки фактическое значение χ^2 равно 6.8665. Это значение превышает величину χ^2, полученную при исследовании статистических свойств системы с использованием уникальных доменов, где она была равна 2.0198. Однако это значение также ниже критического 7.81 (как и в первом случае, уровень значимости 0.05). Таким образом, опытные данные, полученные при участии DNS трафика реальных пользователей, также не противоречат равномерному закону распределения.

На основании проведенных статистических исследований мы можем сделать вывод о том, что рассматриваемый комплекс алгоритмов распределения нагрузки

полностью соответствует предъявляемым к ней требованиям в области равномерности распределения входящих DNS запросов (а, следовательно, и суммарной нагрузки) среди узлов кластера.

4.3 Выводы

Проведенное статистическое исследование и верификация, описанное в рамках данной работы, позволило с помощью математического аппарата показать, что, помимо требований к масштабируемости и быстродействию, реализованный комплекс алгоритмов распределения нагрузки действительно удовлетворяет наиболее важному из требований – равномерному распределению входящих запросов между узлами, составляющих DNS кластер.

Таким образом, в процессе функционирования кластера каждому из его серверов (узлов) приходит на обработку практически одинаковое количество DNS запросов, что приводит к равномерному заполнению внутренних очередей серверов. Это позволяет избежать существенной разницы во времени обработки запросов каждым из серверов по отношению к другим участникам кластера, т.е. с точки зрения пользователя кластер выглядит как одиночная машина независимо от того, с каким из узлов он работает в данный момент и как переключается между ними.

Таким образом, кластер, находящийся под управлением данного комплекса, можно в полной мере назвать кластером распределения нагрузки.

ЗАКЛЮЧЕНИЕ

В заключении сформулируем основные результаты работы.

1. Разработана модель балансировки нагрузки в кластерной системе, позволяющая избежать необходимости хранения метаданных на постоянном запоминающем устройстве для обеспечения функционирования алгоритмов, а также соответствующая требованиям масштабируемости, быстродействия и равномерности распределения нагрузки.

2. Создан алгоритм поиска узла, ответственного за обработку элемента данных, имеющий константную временную сложность и не ограничивающий масштабируемость системы.

3. Созданы алгоритмы репликации, обеспечивающие наличие заданного числа резервных копий в зависимости от конкретных требований к надежности и отказоустойчивости системы, имеющие константную временную сложность и не ограничивающие масштабируемость системы.

4. На основе разработанной модели распределения нагрузки созданы и исследованы новые алгоритмы, отличающиеся высоким быстродействием, проведено тестирование и верификация их программной реализации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Pfister G. In Search of Clusters / G. Pfister. 2nd Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1998. – p. 36.
2. Fox A. Cluster-based scalable network services / A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, P. Gauthier // In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (Saint Malo, France, October 05 - 08, 1997). – New York: ACM Press, 1997. – pp. 78-91.
3. Mockapetris P. Domain Names - Concepts and Facilities / P. Mockapetris // RFC 1034, The Internet Society, 1987.
4. Ghodsi A. Distributed k-ary System: Algorithms for Distributed Hash Tables. / A. Ghodsi. – KTH-Royal Institute of Technology, 2006. – pp. 1-40.
5. Aberery K. The essence of P2P: A reference architecture for overlay networks / K. Aberery, L. O. Alimaz, A. Ghodsi, S. Girdzijauskasy, S. Haridix, M. Hauswirth // In Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (Konstanz, Germany, 31 Aug.-2 Sept. 2005). IEEE, 2005. – pp. 11-20.
6. Karger D. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web / D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, R. Panigrahy // In Proceeding of the Twenty-Ninth Annual ACM symposium on Theory of computing (El Paso, TX, USA, May 04 - 06, 1997). – New York: ACM Press, 1997. – pp. 654-663.
7. Karger D. Web caching with consistent hashing / D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, Y. Yerushalmi // In Proceedings of the Eighth International Conference on World Wide Web (Toronto, Canada, May 11-14, 1999). – New York: Elsevier North-Holland, 1999. – pp. 1203-1213.
8. DeCandia G. Dynamo: Amazon’s Highly Available Key-value Store / G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman,

A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels // In Proceedings of the 21st ACM Symposium on Operating Systems Principles (Skamania Lodge Stevenson, WA, USA, October 14-17, 2007). – New York: ACM Press, 2007. – pp. 205-218.

9. Stoica I. Chord: A scalable peer-to-peer lookup service for internet applications / I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan // In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (San Diego, CA, USA, August 27-31, 2001). – New York: ACM Press, 2001. – pp.149-160.
10. Ratnasamy S. A Scalable Content-Addressable Network / S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker // In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (San Diego, CA, USA, August 27-31, 2001). – New York: ACM Press, 2001. – pp. 1-6.
11. Zhao B. Tapestry: A Resilient Global-Scale Overlay for Service Deployment / B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, J. Kubiatowicz // IEEE Journal On Selected Areas In Communication. – Vol. 22. – No. 1. – 2004. – pp. 41-48.
12. Rowstron A. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems / A. Rowstron, P. Druschel // In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms. – London: Springer-Verlag, 2001. – pp. 329-350.
13. Cox R. Serving DNS using a Peer-to-Peer Lookup Service / R. Cox, A. Muthitacharoen, R. T. Morris // In Proceedings of the First International Workshop on Peer-to-Peer Systems (Cambridge, MA, USA, March 7-8, 2002). – London: Springer-Verlag, 2002. – pp. 155-165.
14. Mockapetris P. Development of the Domain Name System / P. Mockapetris, K. Dunlap // In Proceedings of SIGCOMM ’88 Symposium on Communications architectures and protocols (Stanford, CA, USA, August 16-18, 1988). – New York: ACM Press, 1988. – pp. 123-133.

15. Dabek F. Wide-area cooperative storage with CFS / F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica // In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (Chateau Lake Louise, Banff, Canada, October 21-24, 2001). – New York: ACM Press, 2001. – pp. 202-215.
16. Шилов С.Н. Реализация инфраструктуры распределенной хеш-таблицы в рамках кластерной системы DNS / С.Н. Шилов, С.Д. Кургалин, А.А. Крыловецкий // Вестник ВГУ. Серия: Системный анализ и информационные технологии. – № 2. – Воронеж: изд-во Воронежского гос. ун-та, 2012. – С. 74-79.
17. Шилов С.Н. Двухуровневая схема организации таблиц распределения запросов в кластерной системе DNS / С.Н. Шилов, С.Д. Кургалин, А.А. Крыловецкий // Вестник ВГУ. Серия: Системный анализ и информационные технологии. – № 1. – Воронеж: изд-во Воронежского гос. ун-та, 2014. – С. 90-96.
18. Mansouri N. Combination of data replication and scheduling algorithm for improving data availability in Data Grids / N. Mansouri, G. H. Dastghaibyfard, E. Mansouri // Journal of Network and Computer Applications. – Volume 36. – Issue 2. – London: Academic Press Ltd., 2013. – pp. 711-722.
19. Andronikou V. Dynamic QoS-aware data replication in grid environments based on data «importance» / V. Andronikou, K. Mamouras, K. Tserpes, D. Kyriazis, T. Varvarigou // Future Generation Computer Systems. – Volume 28. – Issue 3. – Amsterdam: Elsevier Science Publishers B. V., 2012. – pp. 544-553.
20. Levy E. Distributed file systems: concepts and examples / E. Levy, A. Silberschatz // ACM Computing Surveys (CSUR). – Volume 22. – Issue 4. – New York: ACM Press, 1990. – pp. 321-374.
21. Sandberg R. Design and implementation of the Sun network filesystem / R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, B. Lyon // Innovations in Internetworking. – Norwood: Artech House Inc., 1988. – pp. 379-390.
22. Nagle D. The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage / D. Nagle, D. Serenyi, A. Matthews // In Proceedings of the

2004 ACM/IEEE conference on Supercomputing (Pittsburgh, Pennsylvania, USA, November 06-12, 2004). – Washington: IEEE Computer Society. 2004. – p. 53.

23. Ghemawat S. The Google file system / S. Ghemawat, H. Gobioff, S.-T. Leung // In Proceedings of the nineteenth ACM symposium on Operating systems principles (The Sagamore, Bolton Landing (Lake George), New York, USA, October 19-22, 2003). – New York: ACM Press, 2003. – pp. 29-43.
24. Шилов С.Н. Анализ и реализация механизма репликации ресурсных записей в DNS кластере / С.Н. Шилов, С.Д. Кургалин, А.А. Крыловецкий // Информационные технологии. – №6. – М.: Новые технологии, 2014. – С. 38-43.
25. Вентцель Е.С. Теория вероятностей / Е.С. Вентцель. – 4-е изд. – М: Наука, 1969. – С. 149-158.
26. Пугачев В.С. Теория вероятностей и математическая статистика / В.С. Пугачев. Учеб. пособие. – 2-е изд., исправл. и дополн. – М.: ФИЗМАТЛИТ, 2002. – С. 335-344.
27. Coulouris G. Distributed Systems: Concepts and Design / G. Coulouris, J. Dollimore, T. Kindberg, G. Blair // 5th Edition. – Addison-Wesley Publishing Company, 2011. – 1008 p.
28. Maymounkov P. Kademlia: A Peer-to-peer Information System Based on the XOR Metric / P. Maymounkov, D. Mazieres // In Proceedings of the First International Workshop on Peer-to-Peer Systems (Cambridge, MA, USA, March 7-8, 2002). – London: Springer-Verlag, 2002. – pp. 53-65.
29. Ketama // Ketama: Consistent Hashing. URL: http://www.audioscrobbler.net/development/ketama/ (Дата обращения: 27.08.2014).
30. Shen H. Cycloid: a constant-degree and lookup-efficient P2P overlay network / H. Shen, C. Xu, G. Chen // Performance Evaluation - P2P computing systems. – Volume 63. – Issue 3. – Amsterdam: Elsevier Science Publishers B. V., 2006. – pp. 195-216.

31. Riak // Riak: distributed, scalable, open source key/value store. URL: http://docs.basho.com/riak/latest/theory/concepts/ (Дата обращения: 27.08.2014).
32. Maidsafe-DHT // Maidsafe-DHT: Kademlia DHT with NAT traversal. URL: http://code.google.com/p/maidsafe-dht/ (Дата обращения: 27.08.2014).
33. Lakshman A. Cassandra: a decentralized structured storage system / A. Lakshman, P. Malik // ACM SIGOPS Operating Systems Review. – Volume 44. – Issue 2. – New York: ACM Press, 2010. – pp. 35-40.
34. Wozniak J.M. C-MPI: A DHT implementation for grid and HPC environments / J.M. Wozniak, B. Jacobs, R. Latham, S. Lang, S.W. Son, R. Ross // Tech. Rep. ANL/MCS-P1746-0410, Argonne National Laboratory, 2010.
35. Fitzpatrick B. Distributed caching with Memcached / B. Fitzpatrick. Linux Journal. – Volume 2004. – Issue 124. – Houston: Belltown Media, 2004. – p. 5.
36. Li T. ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table / T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu // In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (Boston, Massachusetts, USA, May 20-24, 2013). – Washington: IEEE Computer Society, 2013. – pp. 775-787.
37. Левитин А. В. Алгоритмы: введение в разработку и анализ / А. В. Левитин. – М.: Вильямс, 2005. – С. 174-179.
38. Кормен Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн // Под ред. И. В. Красикова. – 2-е изд. – М.: Вильямс, 2005. – 1296 с.
39. Шилов С.Н. Анализ статистических показателей системы распределения нагрузки в DNS кластере / С.Н. Шилов. Информационные технологии моделирования и управления. – № 4(88). – Воронеж, 2014. – С. 341-347.
40. Шилов С.Н. Статистическое исследование системы распределения нагрузки в DNS кластере / С.Н. Шилов. Системы управления и информационные технологии. – №3(57). – Воронеж, 2014. – С. 96-99.

41. Kankowski P. Hash functions: An empirical comparison. URL: http://www.strchr.com/hash_functions (Дата обращения: 05.09.2014).
42. Кремер Н.Ш. Исследование операций в экономике / Н.Ш. Кремер, Б.А. Путко, И.М. Тришин, М.Н. Фридман // Под ред. проф. Н.Ш. Кремера. – М.: ЮНИТИ, 2002. – 407 с.
43. Арзамасцев А.А. Нейросетевое моделирование социального объекта с использованием кластерных систем / А.А. Арзамасцев, О.В. Крючин, Н.А. Зенкова, Д.В. Слетков // Вестник Тамбовского Университета. Серия: Естественные и технические науки. – Т.15. – №5. – Тамбов, 2010. – С. 1460-1464.
44. Керниган Б. Язык программирования Си / Б. Керниган, Д. Ритчи // – 2-е изд. – М.: Вильямс, 2007. – 304 с.
45. Шилдт Г. C: полное руководство / Г. Шилдт. – 4-е изд. – М.: Вильямс, 2010. – 704 с.
46. Прата С. Язык программирования С: Лекции и упражнения / С. Прата. – М.: Вильямс, 2006. – 960 с.
47. Кочан С. Программирование на языке Си / С. Кочан. – 3-е изд. – М.: Вильямс, 2006. – 496 с.
48. Павловская Т.А. C/C++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб: Питер, 2003. – 461 с.
49. Кнут Д. Искусство программирования. Сортировка и поиск / Д. Кнут. – Т. 3. – 2-е изд. – М.: Вильямс, 2007. – 824 с.
50. Вирт Н. Алгоритмы и структуры данных / Н. Вирт. – СПб.: Невский Диалект, 2008. – 352 с.
51. Керниган Б. UNIX – универсальная среда программирования / Б. Керниган, Р. Пайк // – М.: Финансы и статистика, 1992. – 304 с.
52. Стахнов А.А. Linux в подлиннике / А.А. Стахнов. – СПб.: БХВ-Петербург, 2011. – 782 с.
53. Колисниченко Д.Н. Linux: Полное руководство / Д.Н. Колисниченко, П.В. Аллен // – СПб.: Наука и Техника, 2006. – 784 с.

54. Стивенс У. UNIX. разработка сетевых приложений / У. Стивенс. – СПб.: Питер, 2003. – 1088 с.
55. Роббинс А. Unix. Справочник / А. Роббинс. – СПб.: КУДИЦ-Пресс, 2007. – 864 с.
56. Руссинович М. Внутреннее устройство Microsoft Windows / М. Руссинович, Д. Соломон // – 6-е изд. – СПб.: Питер, 2013. – 800 с.
57. Харт Дж. Системное программирование в среде Microsoft Windows / Дж. Харт. – 3-е изд. – М.: Вильямс, 2005. – 592 с.
58. Гмурман В.Е. Теория вероятностей и математическая статистика / В.Е. Гмурман. – М.: Высшая школа, 2003. – 479 с.
59. Козлов М.В. Элементы теории вероятности в примерах и задачах / М.В. Козлов. – М.: Изд. МГУ, 1990. – 344 с.
60. Гнеденко Б.В. Курс теории вероятностей / Б.В. Гнеденко. – 8-е изд., испр. и доп. – М.: Едиториал УРСС, 2005. – 448 с.

Приложение А

Исходный код алгоритма построения таблиц вариантов распределения

```
DHT_ERR
dht_update_host_table(DHT dh_table, const DHT_HOST dht_host_table, unsigned
int dht_host_table_size)
{
    unsigned int i, j;
    DHT_HOST host;
    unsigned int all_hosts_pos;
    unsigned int alive_hosts_pos;
    dht_hashed_id_item *hashed_ids_arr;

    assert(dht_host_table && "dht_host_table_updated(): NULL host table
passed");
    assert(dht_host_table_size && "dht_host_table_updated(): zero size of
host table passed");

    if (!dht_host_table || !dht_host_table_size)
        return DHT_E_BAD_PARAM;

    dh_table->dht_host_table_size = 0;

    if (dh_table->alive_hosts)
    {
        free(dh_table->alive_hosts);
        dh_table->alive_hosts = NULL;
    }
    if (dh_table->all_hosts)
    {
        free(dh_table->all_hosts);
        dh_table->all_hosts = NULL;
    }

    dh_table->all_hosts_array_len = 0;
    dh_table->alive_hosts_array_len = 0;

    if (dh_table->dht_host_table_capacity < dht_host_table_size)
    {
        /* increase capacity of dht_host_table incrementally */
        dh_table->dht_host_table_capacity = dht_host_table_size +
DHT_HOST_TABLE_INCR_SIZE;

        dh_table->dht_host_table = (DHT_HOST)realloc(dh_table-
>dht_host_table, dh_table->dht_host_table_capacity * sizeof(struct
dht_host_t));

        if (!dh_table->dht_host_table)
        {
            dh_table->dht_host_table_size = 0;
            dh_table->dht_host_table_capacity = 0;
            return DHT_E_NO_MEMORY;
        }
    }
```

```
    memcpy(dh_table->dht_host_table, dht_host_table, dht_host_table_size *
sizeof(struct dht_host_t));
    dh_table->dht_host_table_size = dht_host_table_size;

    dh_table->all_hosts_array_len = dht_host_table_size;

    for (i = 0; i < dh_table->dht_host_table_size; i++)
    {
        if (!dh_table->dht_host_table[i].dead)
            dh_table->alive_hosts_array_len++;
    }

    if (!dh_table->alive_hosts_array_len)
        return DHT_E_NO_ALIVE_HOSTS;

    switch (dh_table->alive_hosts_array_len)
    {
        /* there is one alive node */
        case 1:
        {
            dh_table->variants_count = 1;

            if (!(dh_table->alive_hosts = (unsigned int
**)malloc(sizeof(unsigned int *) + sizeof(unsigned int))))
            {
                dh_table->variants_count = 0;
                return DHT_E_NO_MEMORY;
            }

            dh_table->alive_hosts[0] = (unsigned int *)(dh_table-
>alive_hosts + dh_table->variants_count);

            for (i = 0; i < dh_table->dht_host_table_size; i++)
            {
                if (!dh_table->dht_host_table[i].dead)
                {
                    dh_table->alive_hosts[0][0] = i;
                    break;
                }
            }

            /* we do not need to use all_hosts table if there is only one
alive node */
            if (dh_table->all_hosts)
            {
                free(dh_table->all_hosts);
                dh_table->all_hosts = NULL;
                dh_table->all_hosts_array_len = 0;
            }

            break;
        }

        /* there are several alive nodes */
        default:
        {
            dh_table->variants_count = DHT_MAX_VARIANTS_COUNT;
```

```
            if (!dh_table->all_hosts)
            {
                if (!(dh_table->all_hosts = (unsigned int
**)malloc(dh_table->variants_count * sizeof(unsigned int *)
                                                                    +
dh_table->variants_count * dh_table->all_hosts_array_len * sizeof(unsigned
int))))
                {
                    dh_table->variants_count = 0;
                    return DHT_E_NO_MEMORY;
                }

                for (i = 0; i < dh_table->variants_count; i++)
                    dh_table->all_hosts[i] = (unsigned int *)(dh_table-
>all_hosts + dh_table->variants_count) + i * dh_table->all_hosts_array_len;
            }

            if (!(dh_table->alive_hosts = (unsigned int **)malloc(dh_table-
>variants_count * sizeof(unsigned int *)
                                                                  +
dh_table->variants_count * dh_table->alive_hosts_array_len *
sizeof(unsigned int))))
            {
                dh_table->variants_count = 0;
                return DHT_E_NO_MEMORY;
            }

            for (i = 0; i < dh_table->variants_count; i++)
                dh_table->alive_hosts[i] = (unsigned int *)(dh_table-
>alive_hosts + dh_table->variants_count) + i * dh_table-
>alive_hosts_array_len;

            hashed_ids_arr = (dht_hashed_id_item *)malloc(dh_table-
>dht_host_table_size * sizeof(dht_hashed_id_item));

            for (i = 0; i < dh_table->variants_count; i++)
            {
                /* start building dispersion variant, calculate hash values
by host address
                   considering current dispersion variant, write hash
value, host position
                   in dht_host_table and host address to corresponding
array item */
                for (j = 0; j < dh_table->dht_host_table_size; j++)
                {
                    host = &dh_table->dht_host_table[j];
                    hashed_ids_arr[j].id_hash = dht_hash64_key((char
*)&host->id, sizeof(host->id), 17 * (i + 1));
                    hashed_ids_arr[j].pos = j;
                    hashed_ids_arr[j].id = dh_table->dht_host_table[j].id;
                }

                /* sort array by hashed host address to get dispersion
variant */
                qsort(hashed_ids_arr, dh_table->dht_host_table_size,
sizeof(hashed_ids_arr[0]), &dht_compare_hash_values);

                all_hosts_pos = 0;
```

```
                alive_hosts_pos = 0;

                for (j = 0; j < dh_table->dht_host_table_size; j++)
                {
                    if (all_hosts_pos < dh_table->all_hosts_array_len)
                    {
                        /* add dead or alive host to all_hosts array */
                        dh_table->all_hosts[i][all_hosts_pos] =
hashed_ids_arr[j].pos;
                        all_hosts_pos++;

                        /* add only alive host to alive_hosts array */
                        if (!(dh_table-
>dht_host_table[hashed_ids_arr[j].pos].dead)
                            && alive_hosts_pos < dh_table-
>alive_hosts_array_len)
                        {
                            dh_table->alive_hosts[i][alive_hosts_pos] =
hashed_ids_arr[j].pos;
                            alive_hosts_pos++;
                        }
                    }
                }
            }

            free(hashed_ids_arr);

            /* calculate responsibility areas for hosts in all_hosts and
alive_hosts tables */
            dh_table->all_hosts_area_size = DHT_BLOCK_SIZE / dh_table-
>all_hosts_array_len;
            /* during integer division we may lose the fractional part as
area size is integer,
               in this case whole block is not covered by calculated
responsibility areas.
               So we should "round up" the size of responsibility areas if
it is needed */
            if (DHT_BLOCK_SIZE % dh_table->all_hosts_array_len)
                dh_table->all_hosts_area_size += 1;

            dh_table->alive_hosts_area_size = DHT_BLOCK_SIZE / dh_table-
>alive_hosts_array_len;
            if (DHT_BLOCK_SIZE % dh_table->alive_hosts_array_len)
                dh_table->alive_hosts_area_size += 1;

            break;
        }
    }

    return DHT_E_OK;
}
```

Приложение Б

Исходный код алгоритма поиска ответственного узла

```
DHT_ERR
dht_determine_host(DHT dh_table, DHT_UINT64 domain_hash, DHT_HOST *host,
DHT_HOST *replic_hosts, unsigned int *replic_hosts_len)
{
    unsigned int n, block_id, repl_hosts_num;
    unsigned int hash_mod_BLOCK_SIZE;
    unsigned int pos, rpos, rpos_tmp_left, rpos_tmp_right;
    DHT_UINT64 _pos, _rpos;
    DHT_HOST host_tmp;

    assert(dh_table && "dht_determine_host(): NULL DHT context passed");
    assert(host && "dht_determine_host(): NULL target host passed");

    if (host)
        *host = NULL;
    else
        return DHT_E_BAD_PARAM;

    if (!dh_table)
        return DHT_E_BAD_PARAM;

    if (!dh_table->dht_host_table)
        return DHT_E_NULL_HOST_TABLE;

    if (!dh_table->dht_host_table_size)
        return DHT_E_ZERO_HOST_TABLE_SIZE;

    if (dh_table->replication_factor)
    {
        assert(replic_hosts && "dht_determine_host(): NULL pointer to
replication hosts passed");
        assert(replic_hosts_len && "dht_determine_host(): NULL pointer to
replication hosts length passed");

        if (!replic_hosts || !replic_hosts_len)
            return DHT_E_BAD_PARAM;

        memset(replic_hosts, 0, (*replic_hosts_len) * sizeof(DHT_HOST));
        *replic_hosts_len = 0;
    }

    if (!dh_table->alive_hosts_array_len)
        return DHT_E_NO_ALIVE_HOSTS;

    if (dh_table->alive_hosts_array_len > 1)
    {
        /* determine host that will be sent a request */
        block_id = (unsigned int)((domain_hash / DHT_BLOCK_SIZE) %
(dh_table->variants_count));
        hash_mod_BLOCK_SIZE =  domain_hash % DHT_BLOCK_SIZE;

        _pos = hash_mod_BLOCK_SIZE / (dh_table->all_hosts_area_size);
```

```
        assert(UINT_MAX >= _pos && "dht_determine_host(): value of host
position in all_hosts array is overflowed");
        if (_pos > UINT_MAX)
            return DHT_E_VAL_OVERFLOWED;
        pos = (unsigned int)_pos;

        /* we can check this earlier together with UINT_MAX, but these
checks are divided to distinguish different errors */
        assert(dh_table->all_hosts_array_len > pos &&
"dht_determine_host(): host position is out of all_hosts array, check host
position calculation");
        if (pos >= dh_table->all_hosts_array_len)
            return DHT_E_FAIL;

        /* get host from table, which contains all (alive and dead) hosts
*/
        *host = &dh_table->dht_host_table[dh_table-
>all_hosts[block_id][pos]];
        /* if chosen host is dead, then get host from table, which contains
only alive hosts */
        if ((*host)->dead)
        {
            _pos = hash_mod_BLOCK_SIZE / (dh_table->alive_hosts_area_size);

            assert(UINT_MAX >= _pos && "dht_determine_host(): value of host
position _pos in alive_hosts array is overflowed");
            if (_pos > UINT_MAX)
                return DHT_E_VAL_OVERFLOWED;
            pos = (unsigned int)_pos;

            assert(dh_table->alive_hosts_array_len > pos &&
"dht_determine_host(): host position is out of alive_hosts array, check
host position calculation");
            if (pos >= dh_table->alive_hosts_array_len)
                return DHT_E_FAIL;

            *host = &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][pos]];
        }

        if (dh_table->replication_factor)
        {
            /* replication mechanism starts working from this point,
               It is not called as a function to avoid unnecessary
               repeating of calculation of some needed values */
        }
    }
    else if (dh_table->alive_hosts_array_len == 1)
    {
        /* there is just one alive host in a cluster */
        *host = &dh_table->dht_host_table[dh_table->alive_hosts[0][0]];
    }
    else  /* table is not yet initialized  */
    {
        return DHT_E_NO_ALIVE_HOSTS;
    }
    return DHT_E_OK;
}
```

Приложение В

Исходный код алгоритма репликации на основе взаимного перекрытия областей ответственности узлов

```
/* determine neighbor host that will be sent a hash value if host area
intersection is used */
if (dh_table->intersect_area_size && neighbor)
{
    intersect_mod = hash_mod_BLOCK_SIZE  % (dh_table->area_size);

    if (intersect_mod <= (dh_table->intersect_area_size))
    {
        if (pos - 1 >= 0)
        {
             neighbor_block_id = block_id;
             *neighbor = dh_table->hosts[block_id][pos - 1];
        }
        else
        {
             neighbor_pos = dh_table->hosts_array_len - 1;
             neighbor_block_id = (block_id - 1 + dh_table->variants_count)
% (dh_table->variants_count);

             *neighbor = (dh_table->hosts[block_id][pos] != dh_table-
>hosts[neighbor_block_id][neighbor_pos])
                         ? dh_table->hosts[neighbor_block_id][neighbor_pos]
                         : dh_table->hosts[neighbor_block_id][--
neighbor_pos];
        }

    }
    else if (intersect_mod >= ((dh_table->area_size) - (dh_table-
>intersect_area_size)))
    {
        if (pos + 1 < dh_table->hosts_array_len)
        {
            neighbor_block_id = block_id;
            *neighbor = dh_table->hosts[block_id][pos + 1];
        }
        else
        {
            neighbor_pos = 0;
            neighbor_block_id = (block_id + 1) % (dh_table-
>variants_count);

             *neighbor = (dh_table->hosts[block_id][pos] != dh_table-
>hosts[neighbor_block_id][neighbor_pos])
                        ? dh_table->hosts[neighbor_block_id][neighbor_pos]
                        : dh_table-
>hosts[neighbor_block_id][++neighbor_pos];
        }
    }
}
```

Приложение Г

Исходный код алгоритма репликации на основе использования ближайших к ответственному узлу областей ответственности

```
/* calculate position of host for key handling in alive_hosts table
 * NOTE: only table which contains only alive nodes is used for
replication. */

_rpos = hash_mod_BLOCK_SIZE / (dh_table->alive_hosts_area_size);
assert(UINT_MAX >= _rpos && "dht_determine_host(): value of host position
_rpos in alive_hosts array is overflowed");
if (_rpos > UINT_MAX)
    return DHT_E_VAL_OVERFLOWED;
rpos = (unsigned int)_rpos;

assert(dh_table->alive_hosts_array_len > rpos && "dht_determine_host():
host position for replication mechanism is out of alive_hosts array, check
host position calculation");
if (rpos >= dh_table->alive_hosts_array_len)
    return DHT_E_REPL_FAIL;

repl_hosts_num = 0;

if (dh_table->replication_factor >= dh_table->alive_hosts_array_len - 1 /*
responsible host */)
{
    /* replication level is greater than count of alive hosts exclude
responsible host,
       add all hosts but responsible to replication list */
    for (n = 0; n < dh_table->alive_hosts_array_len; n++)
    {
        host_tmp            =            &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][n]];
        if (*host != host_tmp)
        {
            replic_hosts[repl_hosts_num] = host_tmp;
            repl_hosts_num++;
        }
    }

    *replic_hosts_len = repl_hosts_num;
}
else
{
    /* check boundary conditions */
    if (0 == rpos)
    {
        /* host is the leftmost in the distribution variant,
           go to the right to gather hosts for replication */
        rpos_tmp_right = rpos;
        while (repl_hosts_num != dh_table->replication_factor)
        {
            host_tmp          =          &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][rpos_tmp_right]];
            if (*host != host_tmp) /* skip responsible host */
```

```
            {
                replic_hosts[repl_hosts_num] = host_tmp;
                repl_hosts_num++;
            }

            rpos_tmp_right++;
        }

        *replic_hosts_len = repl_hosts_num;
    }
    else if (dh_table->alive_hosts_array_len - 1 == rpos)
    {
        /* host is the rightmost in the distribution variant,
           go to the left to gather hosts for replication */
        rpos_tmp_left = rpos;
        while (repl_hosts_num != dh_table->replication_factor)
        {
            host_tmp          =          &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][rpos_tmp_left]];
            if (*host != host_tmp) /* skip responsible host */
            {
                replic_hosts[repl_hosts_num] = host_tmp;
                repl_hosts_num++;
            }

            rpos_tmp_left--;
        }

        *replic_hosts_len = repl_hosts_num;
        }
        else
        {
            /* firstly go to the left from the host position (including
host position)
               and gather hosts for replication */
            rpos_tmp_left = rpos + 1;
            while ((0 != rpos_tmp_left) && (repl_hosts_num < dh_table-
>replication_factor / 2))
            {
                rpos_tmp_left--;

                host_tmp        =        &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][rpos_tmp_left]];
                if (*host != host_tmp) /* skip responsible host */
                {
                    replic_hosts[repl_hosts_num] = host_tmp;
                    repl_hosts_num++;
                }
            }

            /* now go to the right from the host position (excluding host
position as we
               consider it at the previous step) and gather hosts for
replication */
            rpos_tmp_right = rpos;
            while ((dh_table->alive_hosts_array_len - 1 != rpos_tmp_right)
&& (repl_hosts_num < dh_table->replication_factor))
            {
```

```
                rpos_tmp_right++;

                host_tmp        =        &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][rpos_tmp_right]];
                if (*host != host_tmp) /* skip responsible host */
                {
                    replic_hosts[repl_hosts_num] = host_tmp;
                    repl_hosts_num++;
                }
            }

            /* check whether current number of hosts for replication is
lower than
               replication factor (i.e. right boundary of the block was
reached at the previous step),
               gather hosts from the left to reach replication factor.
               Here we do not need to check boundary condition */
            while (repl_hosts_num < dh_table->replication_factor)
            {
                rpos_tmp_left--;

                host_tmp        =        &dh_table->dht_host_table[dh_table-
>alive_hosts[block_id][rpos_tmp_left]];
                if (*host != host_tmp) /* skip responsible host */
                {
                    replic_hosts[repl_hosts_num] = host_tmp;
                    repl_hosts_num++;
                }
            }

            *replic_hosts_len = repl_hosts_num;
        }
    }
}
```