МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет	*Кафедра*
вычислительной техники	«Вычислительная техника»

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»

Профиль подготовки: «Программное обеспечение средств вычислительной техники и автоматизированных систем»

БАКАЛАВРСКАЯ РАБОТА

на тему

ПРОКСИРУЮЩИЙ DNS-СЕРВЕР

Студент 09.06.2026 (подпись, дата) Изосин Михаил Алексеевич (ФИО полностью)

Руководитель 09.06.2026 (подпись, дата) Егоров В.Ю. (фамилия, инициалы)

Нормоконтролёр 09.06.2026 (подпись, дата) Бычков А.С. (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 22.05.2026 г. № 14)

Заведующий кафедрой (подпись) Митрохин М.А. (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 19.06.26 № 42)

Секретарь ГЭК (подпись) Гурин Е.И. (фамилия, инициалы)

Пенза 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____М.А. Митрохин
«07» 04 2026 г.

ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ

1. Студент Изосин Михаил Алексеевич гр. 22ВВП1 факультета ВТ

2. Тема работы Проксирующий DNS-сервер

Тема утверждена приказом ПГУ №347/о от "7" апреля 2026 г.

3. Руководитель бакалаврской работы Егоров В.Ю

4. Задание на работу (назначение разработки, исходные данные и т.п.)
Разработать систему программ, включающую в себя проксирующий DNS-сервер с поддержкой кэширования записей, пересылки на внешний DNS-сервер и систему мониторинга, отображающую статистику сервера каждую секунду.

5. Перечень подлежащих разработке вопросов

Необходимо проанализировать предметную область: изучить и сравнить существующие аналоги. Выбрать язык программирования и среду разработки. Изучить структуру DNS-протокола и DNS-пакетов. Разработать систему программ: проксирующий DNS-сервер и систему мониторинга статистики в режиме реального времени. Протестировать реализованный DNS-proxy в различных состояниях на одинаковых наборах данных и сравнить с системным сервисом ОС Linux.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области	56	07.04.2026 – 13.04.2026	
2	Изучение и сравнение аналогов	16	14.04.2026 – 15.04.2026	
3	Выбор языка программирования, сторонних библиотек и среды разработки	16	16.04.2026 – 17.04.2026	
4	Разработка проксирующего DNS-сервера с возможностью кэширования записей и пересылки сообщений	96	18.04.2026 – 29.04.2026	
5	Разработка системы мониторинга статистики в режиме реального времени с использованием фреймворка Qt	96	30.04.2026 – 11.05.2026	
6	Тестирование работоспособности программы в различных состояниях кэша	16	12.05.2026 – 13.05.2026	
7	Сравнение тестовых проверок с system-resolved	16	14.05.2026 – 15.05.2026	
10	Получение и анализ результатов	16	16.05.2026 – 17.05.2026	
11	Оформление пояснительной записки	32	18.05.2026 – 21.05.2026	

Дата выдачи задания "__7__"_____апреля_____2026_г.__

Руководитель бакалаврской работы _____ Егоров В.Ю.
(подпись, дата)

Задание к исполнению принял студент _____ Изосин М.А.
(подпись, дата)

Работу к защите допустить
Декан факультета 17.06.2026 Фионова Л.Р.
(подпись, дата)

Реферат

Бакалаврская работа содержит 100 листов, 23 рисунка, 2 таблицы, 15 источников, 3 приложения.

ПРОКСИРУЮЩИЙ DNS СЕРВЕР, SYSTEMD-RESOLVED, DNS-ЗАПРОСЫ, КЭШИРОВАНИЕ, RESOLVER, FORWARDING, СИСТЕМА ПРОГРАММ, МОНИТОРИНГ, TTL

Цель работы – разработать систему программ, включающую в себя проксирующий DNS-сервер и систему мониторинга статистики в режиме реального времени, а также проанализировать существующие аналоги и структуру DNS и DNS-пакетов.

Разрабатываемый сервер обеспечивает возможность пересылки запросов на внешний DNS-сервер и последующее кэширование записей. Многопоточная архитектура позволяет ускорить обработку множества запросов и уменьшить время ожидания ответа.

Система мониторинга представляет собой графический интерфейс с поддержкой отображения актуальных данных о состоянии сервера за последнюю секунду.

В результате была разработана, протестирована и сопоставлена система программ с DNS-сервисом ОС Linux. Реализация ПО была выполнена на языке C++ с использованием фреймворка Qt и внешних библиотек для парсинга DNS-запросов и сериализацию в JSON. Тестирование проводилось с использованием написанного на языке Bash скрипта и утилиты dig для определения времени обработки запроса.

Содержание

Перечень сокращений и обозначений 5
Введение 7
1 Теоретические основы работы DNS proxy-серверов 8
1.1 Анализ темы 8
1.1.1 История и назначение системы доменных имен 8
1.1.2 Архитектура DNS 9
1.1.3 Структура DNS-пакета 13
1.1.4 Транспортный уровень DNS 17
1.2 Анализ существующих аналогов 17
1.3 Постановка задачи 21
2 Выбор и настройка инструментов разработки 23
2.1 Выбор языка программирования 23
2.2 Выбор внешних библиотек 23
2.3 Выбор инструментов разработки 25
2.3.1 Выбор системы сборки и системы контроля версий 25
2.3.2 Выбор операционной системы 25
2.3.3 Выбор средства виртуализации и установка ОС 25
2.3.4 Выбор сред разработки 26
2.3.5 Настройка сред разработки 27
3 Разработка программного модуля 30
3.1 Архитектура приложения 30
3.2 Разработка приложения 32
3.2.1 Реализация модуля DNSServer 32
3.2.2 Реализация модуля ThreadPool 32
3.2.3 Реализация модуля DNSParser 33
3.2.4 Реализация модуля DNSDispatcher 33
3.2.5 Реализация модуля DNSCache 34

3.2.6 Реализация модуля DNSResolver 35
3.2.7 Реализация модуля OutQueue 35
3.2.8 Реализация модуля Utils 35
3.2.9 Реализация модуля MetricsManager 36
3.2.10 Реализация модуля Client 37
3.2.11 Реализация модуля Keys 37
3.2.12 Реализация модуля Client 38
3.2.13 Реализация модуля DashboardController 38
3.2.14 Реализация модуля ToolBar 38
3.2.15 Реализация модуля CentralWidget 39
3.2.16 Реализация модуля BaseCustomPlot 39
3.2.17 Реализация модуля StatusBar 39
3.2.18 Реализация модуля DarkTheme 39
3.2.19 Реализация модуля Utils 40
3.2.20 Реализация модуля Keys 40
3.3 Описание приложения 40
3.3.1 Сборка серверной части 40
3.3.2 Подготовка к запуску и запуск DNS-сервера 41
3.3.3 Запуск DNS-клиента 43
3.3.4 Тестирование 43
4. Безопасность жизнедеятельности 47
Заключение 48
Список использованных источников 49
Приложение А – UML-диаграммы 51
Приложение Б – Листинг программы 57
Приложение В – Презентация 96

Перечень сокращений и обозначений

В настоящей работе применяют следующие сокращения и обозначения

ОС – операционная система

ПО – программное обеспечение

ЭВМ – электронно-вычислительная машина

CMAKE (cross-platform make) – система автоматизации сборки программного обеспечения

CPU (Central Process Unit) – центральный процессор

DHCP (Dynamic Host Configuration Protocol) – протокол динамической настройки узла

DNS (Domain Name System) – система доменных имен

DoH (DNS over HTTPS) – DNS через HTTPS

DoT (DNS over TLS) – DNS через TLS

FTP (File Transfer Protocol) – протокол передачи файлов

HTTP (HyperText Transfer Protocol) – протокол передачи гипертекста

HTTPS (HyperText Transfer Protocol Secure) – защищенный HTTP

IP (Internet Protocol) – межсетевой протокол

JSON (JavaScript Object Notation) – формат обмена данными

LDNS (Lightweight DNS library) – библиотека для работы с DNS

MVC (Model-View-Controller) – архитектурный шаблон

RAM (Random Access Memory) – оперативная память

RFC (Request for Comments) – серия технических документов Internet

TCP (Transmission Control Protocol) – протокол управления передачей

TLD (Top-Level Domain) – домен верхнего уровня

TLS (Transport Layer Security) – протокол защиты транспортного уровня

TTL (Time to live) – время жизни записи

UDP (User Datagram Protocol) – протокол пользовательских дейтаграмм

UI (User Interface) – пользовательский интерфейс

UML (Unified Modeling Language) – язык графического описания моделей программных систем

Введение

В настоящее время повсеместно используется Интернет. Социальные сети, сервисы доставки, банковские системы, облачные платформы, образовательные программы и платформы – все это стало неотъемлемой частью нашей жизни.

При использовании большинства интернет-ресурсов мы взаимодействуем с DNS, даже не догадываясь об этом. Система доменных имен (DNS, Domain Name System) критически важна для конечного пользователя. Она преобразует числовые IP-адреса в названия сайтов и ресурсов и обратно. Без DNS нам бы пришлось запоминать тысячи адресов, состоящих из нескольких чисел, разделенных точками.

Система доменных имен используется в большинстве сетевых запросов, поэтому производительность и надежность данной системы напрямую влияет на скорость доступа к интернет-ресурсам. В связи с постоянным ростом объема сетевого трафика возрастает и нагрузка на DNS-инфраструктуру, что приводит к задержкам, ошибкам и проблемам с доступом к ресурсам.

Одним из способов повышения производительности системы является использование DNS proxy-серверов, с механизмом локального кеширования запросов. Это позволяет сократить количество обращений к удаленным DNS-серверам и уменьшить время обработки запроса.

Целью данной работы является разработка многопоточного DNS proxy-сервера на языке C++ с возможностью кеширования запросов и системой мониторинга статистики в реальном времени.

1 Теоретические основы работы DNS proxy-серверов

1.1 Анализ темы

1.1.1 История и назначение системы доменных имен

Чтобы понять DNS, необходимо обратиться к истории сети ARPAnet [1]. Система DNS была создана с целью решения конкретных проблем этой сети, а сеть Интернет, выросшая из ARPAnet, сейчас является главным потребителем этих решений.

В ранних версиях сети ARPAnet всю информацию, необходимую для взаимных преобразований имен и адресов узлов сети, содержал централизованный текстовый файл HOSTS.TXT. За него отвечал Сетевой информационный центр (Network Information Center) Стэнфордского исследовательского института. Распространение обновлений осуществлялось централизованно: администраторы посылали изменения электронной почтой в NIC и периодически синхронизировали свои файлы HOSTS.TXT с копией на главном узле с помощью протокола FTP.

По мере роста сети такой подход стал создавать ряд проблем. Размер файла постоянно увеличивался. Еще быстрее рос информационный поток, связанный с необходимостью обновления файла на узлах. Потоки и нагрузка на главный узел в смысле сетевого трафика и работы процессора, связанных с раздачей файла, приближалась к предельной. Никакие два узла, описанные в файле HOSTS.TXT, не могли носить одинаковые имена. Не было способа предотвратить добавление узла с уже существующим именем, при том, что такое действие нарушало работу всей схемы. Синхронизация файлов в масштабах быстро растущей сети становилась все более сложной задачей.

Основная проблема заключалась в том, что схема с файлом HOSTS.TXT не поддавалась масштабированию. Это стало основной причиной разработки распределенной системы доменных имен DNS.

Для решения данных проблем была разработана и описана в RFC-

документе, Полом Мокапетрисом система доменных имен. Новая система позволила локально управлять данными, но делать эти данные доступными всем. Децентрализация администрирования решила проблемы с трафиком и нагрузкой, непосильными для единственного узла. Локальное управление данными упростило задачу обновления и синхронизации
информации. В новой системе используется иерархическое пространство имен для присваивания идентификаторов узлам, что позволяет гарантировать уникальность каждого отдельного имени.

Пол Мокапетрис издал документы RFC-882 и RFC-883, в которых описывалась система доменных имен. Эти RFC-документы были обновлены документами RFC-1034 и RFC-1035, которые и являются в настоящее время действующей спецификацией DNS [2, 3].

1.1.2 Архитектура DNS

В стандарте RFC-1034 (Request for Comments) описываются основные принципы работы системы доменных имен и ее структура, а именно:

– пространство доменных имен (Domain Name Space);

– серверы имен (Name Servers);

– резолверы (Resolvers).

Пространство доменных имен представляет собой иерархическую структуру, похожую на структуру файловой системы UNIX. Вся база данных представлена в виде перевернутого дерева, корень (корневой узел) которого расположен на самом верху. Каждый узел дерева имеет прикрепленную текстовую метку, которая идентифицирует его относительно родительского узла по аналогии с «относительным путевым именем» в файловой системе. Одна из меток, пустая, (она обозначается как “ ”) закреплена за корневым узлом дерева. В тексте корневой узел обозначается точкой (.).

Каждый узел является корнем новой ветви дерева. Такие ветви представляют собой домены – логические части пространства доменных имен. Домены могут быть разделены на более мелкие подразделы – поддомены,

которые образуют дочерние элементы относительно родительского домена.

Имя домена в DNS – это последовательность меток от узла, корневого для данного домена, до корня всего дерева; метки в имени домена разделяются точками. Подобная структура обеспечивает уникальность доменных имен и позволяет распределять ответственность за различные части пространства имен между независимыми организациями. Она представлена на рисунке 1 и 2.

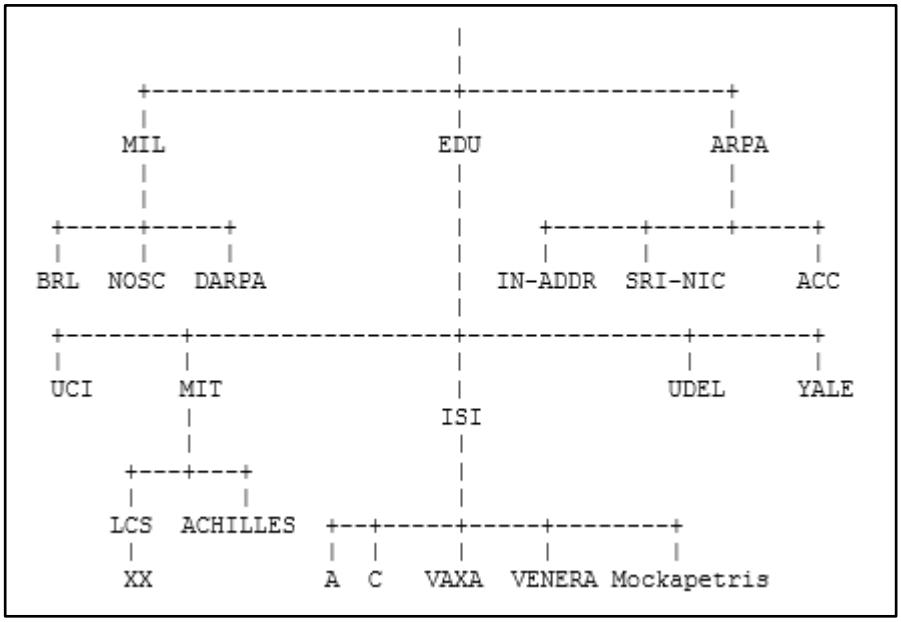


Рисунок 1 – Пример пространства доменных имен RFC-1034

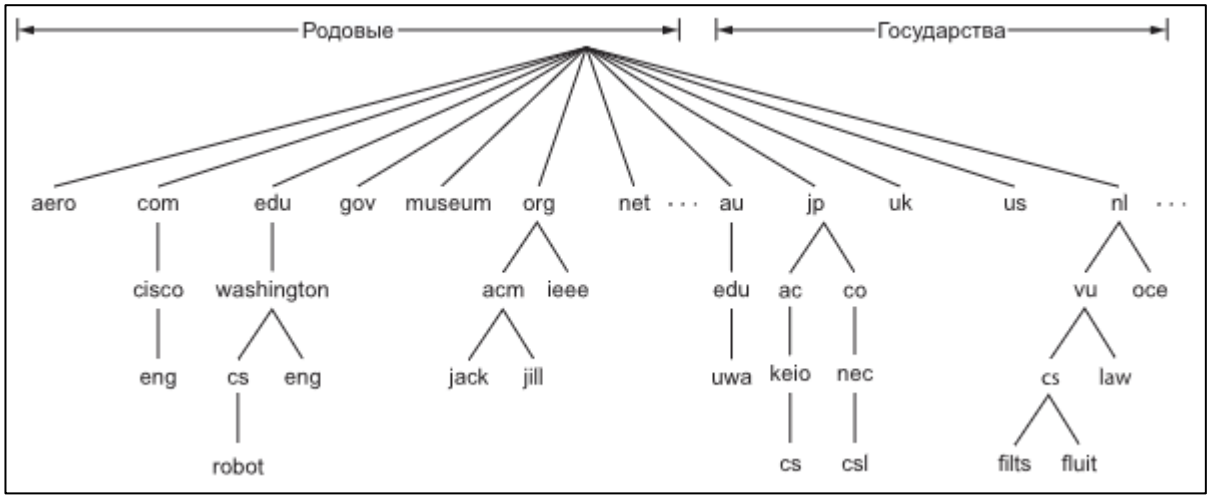


Рисунок 2 – Часть доменного пространства имен Интернета

Программы, владеющие информацией о пространстве доменных имен и

обрабатывающие запросы, называются DNS-серверами. Пространство имен разделено на непересекающиеся зоны (zones), представляющие собой отдельные административные сегменты DNS [4].

DNS-серверы обычно обладают полной информацией по определенным сегментам пространства доменных имен, которая загружается из файла либо может быть получена от других серверов имен. В таких случаях говорят, что сервер имен является авторитетным для конкретной зоны. DNS-серверы могут быть авторитетными также и для нескольких зон.

В спецификации DNS выделяются первичные (primary master) и вторичные (secondary master) DNS-серверы. Первичный сервер загружает данные зоны из локального файла зоны, тогда как вторичный сервер получает копию зональных данных от другого авторитетного DNS-сервера посредством механизма передачи зоны (zone transfer).

Использование вторичных DNS-серверов обеспечивает отказоустойчивость, распределение нагрузки и повышение доступности DNS-службы. Как первичные, так и вторичные серверы являются авторитетными для обслуживаемой зоны. Пример разделения на зоны – рисунок 3.

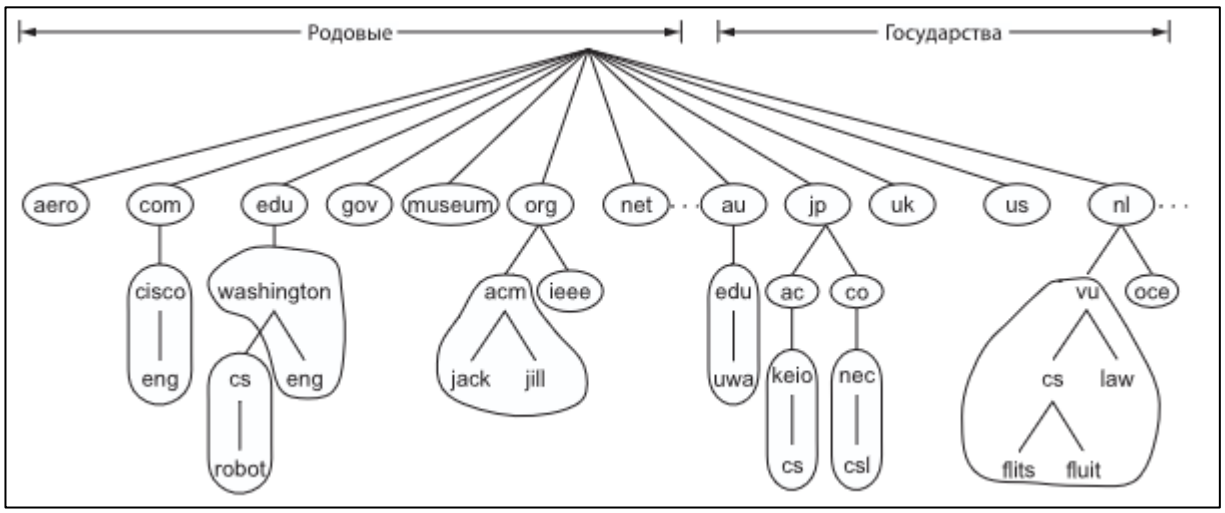


Рисунок 3 – Часть пространства имен DNS, разделенная на очерченные зоны

Процесс поиска IP-адресов по доменному имени называется разрешением имен (name resolution). В Domain Name System используются два основных типа запроса: рекурсивные и итеративные.

При рекурсивном запросе клиент посылает распознавателю (resolver) пакет, для того чтобы тот самостоятельно выполнил последовательность действий и вернул готовый ответ конечному пользователю. DNS-сервер действует как посредник, обращаясь к различным уровням иерархии для получения ответа. Он проверяет локальный кэш. Если не знает адрес, то обращается к корневым серверам. Корневой сервер – это сервер, находящийся на вершине иерархии и содержащий информацию о доменах верхнего уровня (Top Level Domain). Получив информацию о том, какой TLD ему необходим, он обращается к нему. Домен верхнего уровня знает, какой авторитетный сервер владеет данным доменом, поэтому запрашивает его и возвращает распознавателю. Resolver выполняет всю тяжелую работу по поиску IP-адресов, возвращая готовый ответ клиенту.

Для оптимизации работы распознавателя и сокращении запросов на удаленные сервера DNS-сервер использует метод кеширования записей на время, определяемое параметром TTL (Time to Live), содержащимся в ответе авторитетного сервера. Пример изображен на рисунке 4.

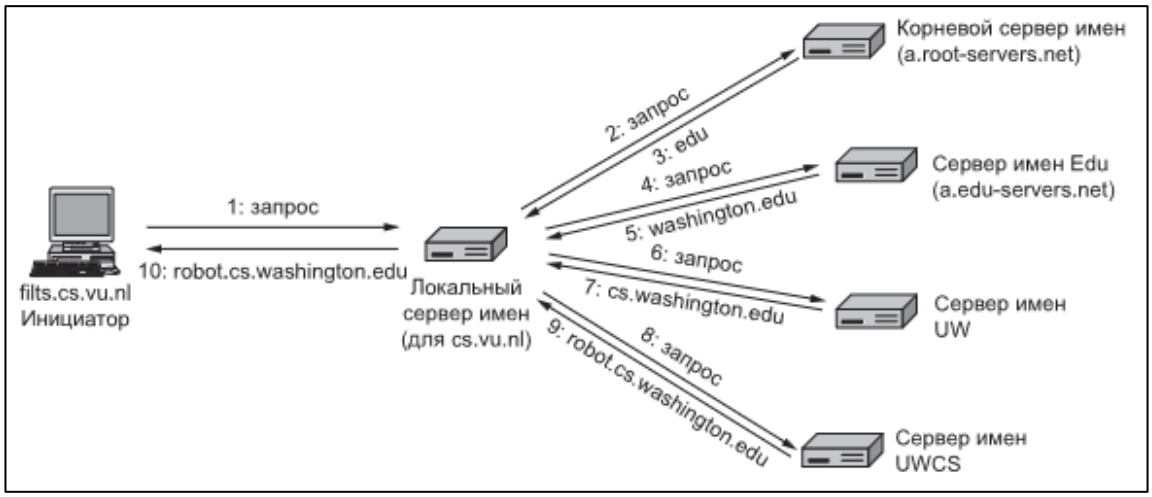


Рисунок 4 – Поиск распознавателем имени удаленного хоста в 10 шагах

В разрабатываемом DNS-прокси сервере используются механизмы пересылки DNS-запросов (forwarding), а также локальное кэширование DNS-записей. Прокси сервер принимает DNS-запросы клиентов, перенаправляет их на удаленные DNS-распознаватели и сохраняет полученные ответы в локальный кэш, сокращая время обработки DNS-запроса и уменьшая количество обращений к внешним серверам.

1.1.3 Структура DNS-пакета

Рассмотрим структуру DNS-пакета. Согласно RFC-1035 структура пакета состоит из пяти секций, представленных на рисунке 5.

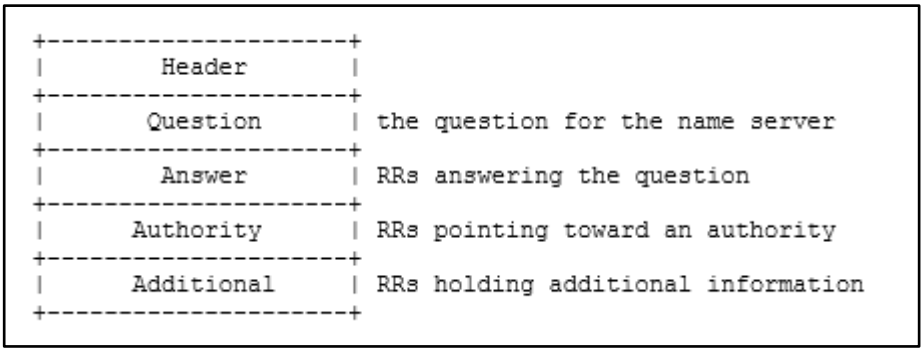


Рисунок 5 – Структура DNS-пакета из RFC-1035

– Header – содержит поля, которые определяют, какие из оставшихся разделов присутствуют, а также указывают, является ли сообщение запросом или ответом, стандартным запросом или каким-либо другим кодом операции и т.д.;

– Question – содержит поля, описывающие запрос к серверу имен. Эти поля относятся к типу запроса (QTYPE), типу класс запроса (QCLASS) и доменное имя запроса (QNAME);

– Answer;

– Authority;

– Additional.

Последние три раздела имеют одинаковый формат: раздел ответов содержит RR, которые отвечают на вопрос; раздел полномочий содержит RR, которые указывают на авторитетный сервер имен; раздел дополнительных

записей содержит RR, которые относятся к запросу, но не являются строго ответами на вопрос. На рисунке 6 представлена структура DNS-заголовка.

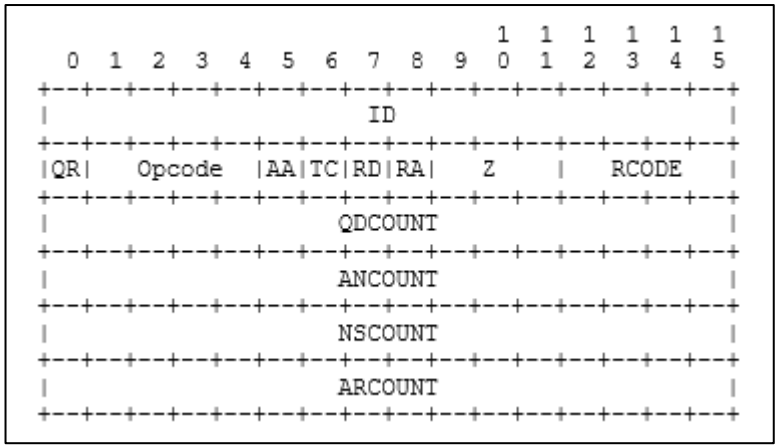


Рисунок 6 – Структура DNS-заголовка из RFC-1035

ID (16 бит) – идентификатор, присваиваемый программой, которая генерирует любой запрос. Указывает на то, что пакет принадлежит одной и той же сессии “запросов-ответов”.

QR (1 бит) – указывает, является ли это сообщение запросом (0) или ответом (1).

OPCODE (4 бита) – код операции. Определяет тип запроса. Задается автором запроса и копируется в ответ:

– 0 – стандартный запрос (QUERY);

– 1 – обратный запрос (IQUERY);

– 2 – запрос о состоянии сервера (STATUS);

– пункты 3-15 зарезервированы для использования в будущем.

AA (1 бит) – ответ от авторитетного сервера.

TC (1 бит) – сообщение было усечено из-за превышения длины.

RD (1 бит) – этот бит может быть задан в запросе и скопирован в ответ. Если задан RD, он направляет сервер имен для выполнения запроса рекурсивно;

RA (1 бит) – отправляется только в ответах, и сообщает о том, что сервер поддерживает рекурсию.

Z (3 бита) – зарезервировано для использования в будущем. Должно быть равно нулю во всех запросах.

RCODE (4 бит):

– 0 – значит запрос прошел без ошибок;

– 1 – ошибка связана с тем, что сервер не смог понять форму запроса;

– 2 – эта ошибка с некорректной работой сервера имен;

– 3 – имя, которое разрешает клиент не существует в данном домене;

– 4 – сервер не может выполнить запрос данного типа;

– 5 – этот код означает, что сервер не может удовлетворить запрос.

QDCOUNT (16 бит) – количество записей в секции запросов.

ANCOUNT (16 бит) – количество записей в секции ответы.

NSCOUNT (16 бит) – количество записей в Authority Section.

ARCOUNT (16 бит) – количество записей в Additional Record Section.

Рассмотрим структуру секции DNS-запроса, представленную на рисунке 7.

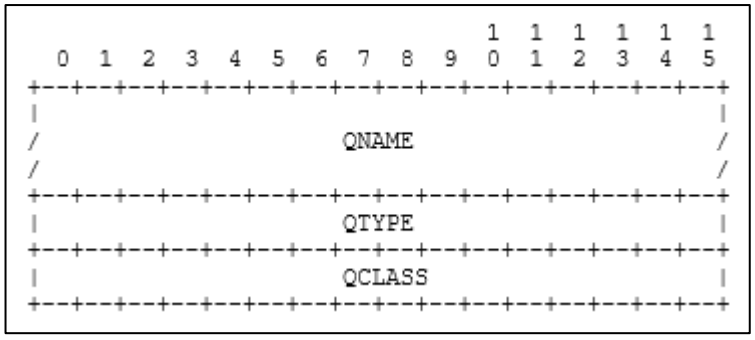


Рисунок 7 – Структура секции запроса из RFC-1035

QNAME – доменное имя, представленное в виде последовательности меток, где каждая метка состоит из октета длины, за которым следует соответствующее количество октетов. Доменное имя заканчивается октетом нулевой длины для нулевой метки корня. В этом поле может быть нечетное число октетов; заполнение не используется.

QTYPE – тип ресурсной записи DNS (A, AAAA, NS и др.).

QCLASS – определяющий класс запроса (Internet или другие).

Далее идет структура секции DNS-ответа – рисунок 8.

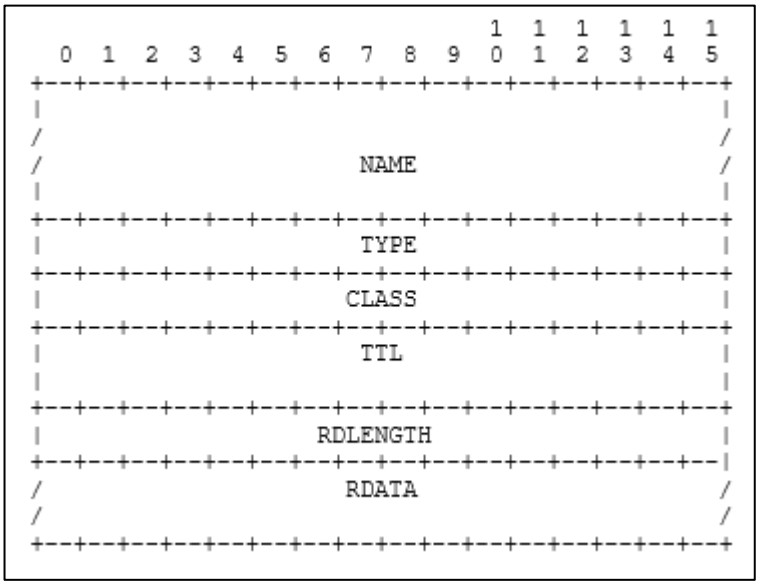


Рисунок 8 – Структура секции ответа из RFC-1035

NAME – доменное имя, к которому относится запись.

TYPE – тип записи DNS, которую мы ищем (A, AAAA, NS и др.).

CLASS – определяющий класс запроса (Internet или другие).

TTL (Time to Live) – целое число без знака, указывающее интервал (в секундах), в течение которого запись ресурса может быть кэширована, прежде чем ее следует удалить.

RDLENGTH – длина поля RDATA.

RDATA – строка данных, содержащая информацию о записи, в зависимости от ее типа.

Согласно RFC-1035, все ресурсные записи DNS имеют единый формат (рисунок 8). Конкретный тип записи определяет поле Type и содержимое RDATA – остальные поля идентичны для всех типов. Такой подход позволяет расширить DNS новыми типами записей без изменения базового формата пакета и логики парсера.

Наиболее распространенными типами ресурсных записей являются:

– A – содержит IPv4-адрес узла;

– AAAA – содержит IPv6-адрес узла;

– CNAME – задает псевдоним доменного имени;

– NS – указывает на авторитетный сервер для домена;

– TXT – хранит текстовую информацию для различных целей;

– MX – включает адрес почтового сервера и его приоритет.

Поддержка данных типов записей является достаточной для обработки большинства пользовательских DNS-запросов в рамках локальной сети и не только.

1.1.4 Транспортный уровень DNS

В классической реализации Domain Name System основным транспортным протоколом является UDP, использующий порт 53. Применение UDP позволяет минимизировать задержки и уменьшить сетевые накладные расходы за счет отсутствия необходимости устанавливать соединение между узлами.

Помимо DNS over UDP существуют и другие его модификации [5, 6, 7, 8]:

– DNS over TCP – передача DNS-сообщений по TCP;

– DNS over TLS (DoT) – передача зашифрованного сообщения с использованием TLS;

– DNS over HTTPS (DoH) – передача зашифрованного сообщения с использованием HTTPS.

В разработанном сервере используется базовый протокол DNS over UDP.

1.2 Анализ существующих аналогов

В современных операционных системах (ОС) уже существует ряд утилит, как встроенных системных, так и проприетарных, выполняющих пересылку, разделение, кэширование и обработку DNS-записей. Они необходимы для работы сети и настраиваются вместе с установкой ОС. Это могут быть как простые stub-клиенты, так и полноценные DNS-серверы с поддержкой настройки под свои нужды. Рассмотрим наиболее популярные альтернативы.

Systemd-resolved – это локальный резолвер в современных Linux-дистрибутивах, который получает DNS-настройки от сетевого стека (NetworkManager или systemd-networkd), кеширует ответы, поддерживает split DNS и умеет шифровать запросы к upstream через DNS-over-TLS (DoT) [5, 6, 7, 8]. Он выступает в роли stub-резолвера, слушая запросы на 127.0.0.53:53 и перенаправляя их в зависимости от стратегии пересылки и разделения по доменам/интерфейсам. Данный клиент поддерживает кэширование запросов, что позволяет добиться лучшей производительности при работе с сетью. При должной настройке клиента, можно добиться высокой скорости обработки запроса, корректной маршрутизации запросов, шифрации пакетов к upstream.

Недостатками являются:

– сложность внутренней архитектуры;

– избыточная функциональность для простых DNS-proxy сценариев;

– навыки настройки systemd-resolved;

– ограниченные возможности мониторинга внутренней статистики.

Разрабатываемый DNS-proxy сервер отличается от systemd-resolved сравнительно простой архитектурой и настройкой программы. Кроме того, разрабатываемый сервер поддерживает точный мониторинг статистики в режиме реального времени.

Dnsmasq представляет собой легковесный DNS-forwarder и DHCP-сервер, широко применяемый в Linux-системах, маршрутизаторах, встроенных устройствах и локальных сетях. Он поддерживает те же возможности что и systemd-resolved и добавляет возможность настройки DHCP сервера. Основной особенностью программы является ее низкое потребление ресурсов и простота настройки. Благодаря этому ее широко применяют в различных Linux-системах и маршрутизаторах.

В dnsmasq, объединяющем кэширующий DNS-резолвер и сервер DHCP, выявлено 7 уязвимостей, которым присвоено кодовое имя DNSpooq. Данные

проблемы устранены на данный момент. Кроме того, dnsmasq не является полноценным резолвером [9].

По архитектуре dnsmasq наиболее близок к разрабатываемому DNS proxy серверу. Он также использует DNS-forwarding + caching. Отличиями является развитая система метрик разрабатываемого proxy и низкая вероятность потенциальной угрозы, в связи со скрытым исходным кодом разрабатываемого сервера.

Рассмотрим альтернативы под ОС Windows. Как и в системе Linux, Windows имеет встроенный Windows клиент, который предназначен для разрешения доменных имен. Он работает как stub-resolver, проверяя кэш, затем файл hosts.txt и др. локальные источники. Затем перенаправляет запрос на внешний источник, указанный в конфигурации или полученный через DHCP. Преимуществами являются тесная интеграция с ОС Windows, автоматическая настройка и локальное кэширование записей. Недостатками же является неполноценный резолвер, отсутствие системы мониторинга. Возможности гибкой настройки DNS-forwarding ограничены средствами ОС.

В качестве иного решения для ОС Windows можно рассмотреть Acrylic DNS Proxy – локальный прокси, разработанный специально под ОС Windows. Его преимуществами является простая установка, настройка стратегий forwarding и фильтрации нежелательных имен, настройка собственных правил маршрутизации. К недостаткам можно отнести отсутствие рекурсивного режима и системы статистик.

Подводя итог в рассмотрении и сравнении наиболее популярных аналогов DNS-proxy сервера под разные операционные системы, можно составить таблицу (таблица 1), в которой будут собраны все особенности и недостатки данных решений.

Таблица 1 – Сравнение аналогов DNS-proxy

Критерий	systemd-resolved	dnsmasq	Acrylic	Windows DNS Client	DNS-proxy server
DNS forwarding	есть	есть	есть	есть	есть
DNS caching	есть	есть	есть	есть	есть
Мониторинг запросов	ограничен	ограничен	ограничен	отсутствует	realtime
Простота настройки	низкая	высокая	высокая	Очень высокая	средняя
Платформа	Linux	Linux	Windows	Windows	Linux
Прозрачность архитектуры	низкая	средняя	средняя	низкая	высокая

Анализ существующих DNS-решений показывает, что на данный момент существует большинство решений, разработанные для решения базовых задач разрешения доменных имен. Эти решения могут являться компонентами операционных систем или быть независимыми разработками под определенную платформу и нацеленные на повышение производительности при работе с DNS-запросами. Все вышеперечисленные альтернативы реализуют локальное кэширование запросов и forwarding на удаленный сервер. Некоторые серверы содержат сложную внутреннюю архитектуру и/или скрытый исходный код, что ограничивает возможность их расширения, модификации и интеграции в специализированные программные решения.

Разрабатываемый DNS-proxy сервер занимает нишу специализированного программного решения, ориентированного на обработку DNS-запросов с возможностью их пересылки и кэширования.

Ключевым отличием данной системы является независимость от зарубежных решений, расширенная система мониторинга статистик DNS-сервера. Предоставленная программа обладает сравнительно простой архитектурой, что облегчает ее анализ, модификацию и расширение функциональности по сравнению с системными резолверами.

Таким образом, разработанный проксирующий DNS-сервер обеспечивает баланс между функциональностью, наблюдаемостью системы и масштабируемостью, что делает ее актуальным решением для задач локального DNS-обслуживания.

1.3 Постановка задачи

В рамках данной работы требуется разработать многопоточный проксирующий DNS-сервер, обеспечивающий обработку DNS-запросов клиентов по протоколу UDP в соответствии с RFC-1035.

Разрабатываемая система должна принимать DNS-запросы, выполнять их пересылку (forwarding) на внешний DNS-резолвер, а также возвращать клиенту полученные ответы. Для повышения производительности необходимо реализовать механизм локального кэширования DNS-записей с использованием TTL (Time to Live), указанного в ответе авторитетного сервера.

Сервер должен поддерживать обработку 6 стандартных типов DNS-запросов, описанных выше. Размер обрабатываемого пакета не должен превышать 512 байт, согласно стандарту RFC-1035.

В качестве удаленного DNS-resolver используется публичный сервер «Яндекс.DNS» с IP-адресом – 77.88.8.8.

В рамках поставленной задачи требуется реализовать:

– прием и обработку DNS-запросов;

– пересылку DNS-запросов внешнему резолверу;

– прием и обработку DNS-ответов, возврат их клиенту;

– кэширование базовых 6 записей на время TTL;

– многопоточную обработку запросов для повышения производительности;

– систему мониторинга основных метрик работы сервера в реальном времени.

Ограничения разрабатываемой системы:

– отсутствие переключения на TCP при получении усеченных пакетов с флагом TC;

– отсутствие обработки DNS over TLS и других защищенных DNS протоколов;

– отсутствие полноценного рекурсивного резолвера (Recursing resolver).

2 Выбор и настройка инструментов разработки

2.1 Выбор языка программирования

Для разработки proxy DNS-сервера основным языком программирования стал C++ [10]. Данный язык отлично подходит для написания низкоуровневых, системных приложений, тесно связанных с сетевыми протоколами транспортного уровня. C++ отлично подходит для написания легковесного, производительного приложения с многопоточной архитектурой и эффективной работой с сетью, что критически важно при работе с протоколами, требующими низкие накладные расходы по памяти и высокой скорости обработки пакетов.

Основная логика приложения реализована на языке C++. Благодаря этому архитектура приложения имеет объектно-ориентированный вид.

Дополнительными особенностями выбора языка C++ является:

– поддержка сетевого программирования в Linux;

– автоматическое освобождение ресурсов;

– интеграция с библиотеками на C.

Для реализации графического интерфейса мониторинга статистики сервера был использован фреймворк Qt. Qt предоставляет средства разработки кроссплатформенных приложений и поддерживает взаимодействие с сервером через TCP сокет. Сигнально-слотовая архитектура фреймворка обеспечивает эффективное взаимодействие между компонентами приложения, что необходимо при разработке системы мониторинга в реальном времени.

2.2 Выбор внешних библиотек

Для реализации DNS-proxy сервера использовались сторонние библиотеки, обеспечивающие парсинг DNS-пакетов, сборку и обработку, а также библиотека для конвертирования метрик в JSON для передачи пакета клиенту.

В качестве инструмента для парсинга используется сторонняя библиотека ldns, написанная на языке C [11]. Данная библиотека предоставляет набор функций для работы с DNS-сообщениями, ресурсными записями, флагами заголовка и различными типами.

Использование библиотеки ldns позволило существенно упростить обработку DNS-пакетов и избежать ручного парсинга сложных DNS-запросов [11]. Библиотека используется для:

– сериализации и десериализации пакетов;

– формирования DNS-ответа;

– получения RDATA из каждой записи для последующего кэширования;

– работы с реализованными DNS-типами.

Для сериализации в JSON-файл и последующей передачи пакета с набором метрик клиенту используется библиотека nlohmann/json для языка C++ [12]. Данная библиотека предоставляет удобные средства работы с JSON-структурами позволяет выполнить преобразование объектов C++ в формат JSON и обратно.

Использование JSON необходимо для передачи статистики между серверной и клиентской частью приложения [12].

Для реализации графического интерфейса отображения динамических метрик была использована сторонняя библиотека QCustomPlot, предназначенная для построения интерактивных графиков в приложениях на базе Qt.

Использование QCustomPlot позволяет реализовать обновление графиков без полной перерисовки интерфейса, что обеспечивает высокую производительность и плавное отображение данных при работе с потоковыми значениями.

2.3 Выбор инструментов разработки

2.3.1 Выбор системы сборки и системы контроля версий

Для сборки клиентской части приложения использовалась система автоматизации сборки CMake.

Данный инструмент позволяет описывать структуру проекта в платформонезависимом виде. Использование Cmake также обеспечивает удобную поддержку многофайловой архитектуры проекта и дальнейшее расширение приложения.

Для управления исходным кодом и отслеживания изменений применялась система контроля версий Git. Использование Git позволило организовать хранение исходного кода, вести историю изменений проекта, а также упростить процесс разработки и тестирования отдельных компонентов DNS-proxy сервера.

2.3.2 Выбор операционной системы

Для разработки проксирующего DNS-сервера была выбрана платформа Ubuntu 24.04 LTS. Данный дистрибутив был выбран благодаря стабильной работы, широкой распространенности в области серверного программирования, а также наличию большого количества инструментов разработки и библиотек для языков C/C++.

2.3.3 Выбор средства виртуализации и установка ОС

Разработка и тестирование приложения выполнялись в виртуальной среде VMware Workstation (рисунок 9).

Использование виртуальной машины позволило изолировать сетевое окружение DNS-Сервера, упростить настройку сетевых интерфейсов и обеспечить безопасное тестирование работы приложения без влияния на основную ОС.

Создали виртуальную машину на Ubuntu 24.04. Произведем полную установку (рисунок 10).

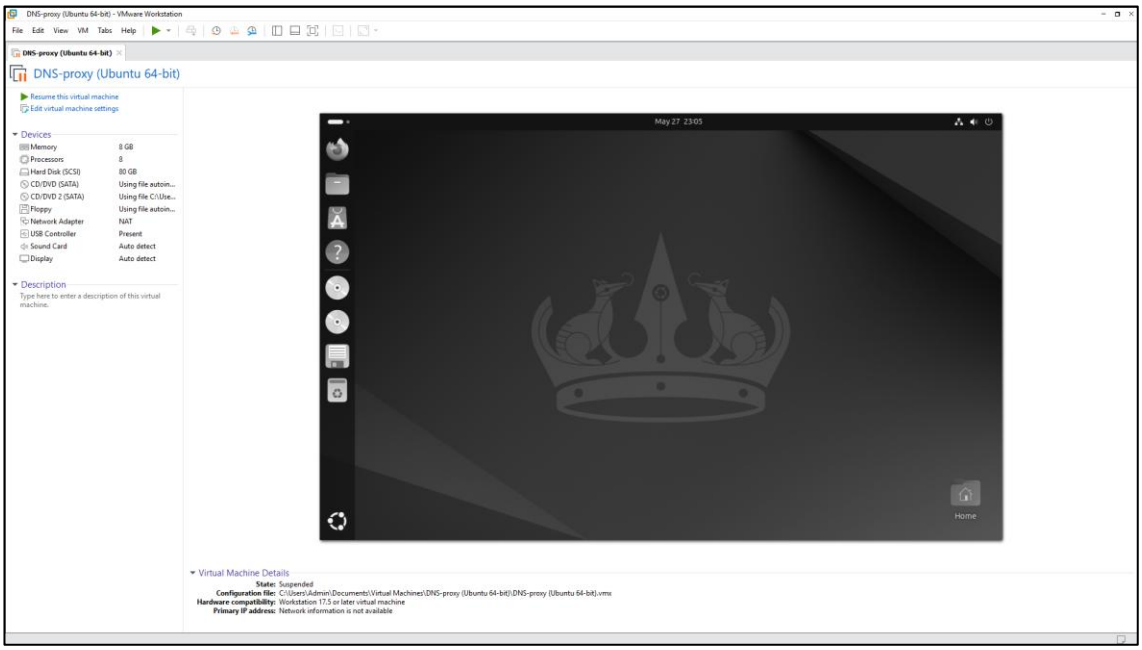


Рисунок 9 – Установка виртуальной машины в среде VMware

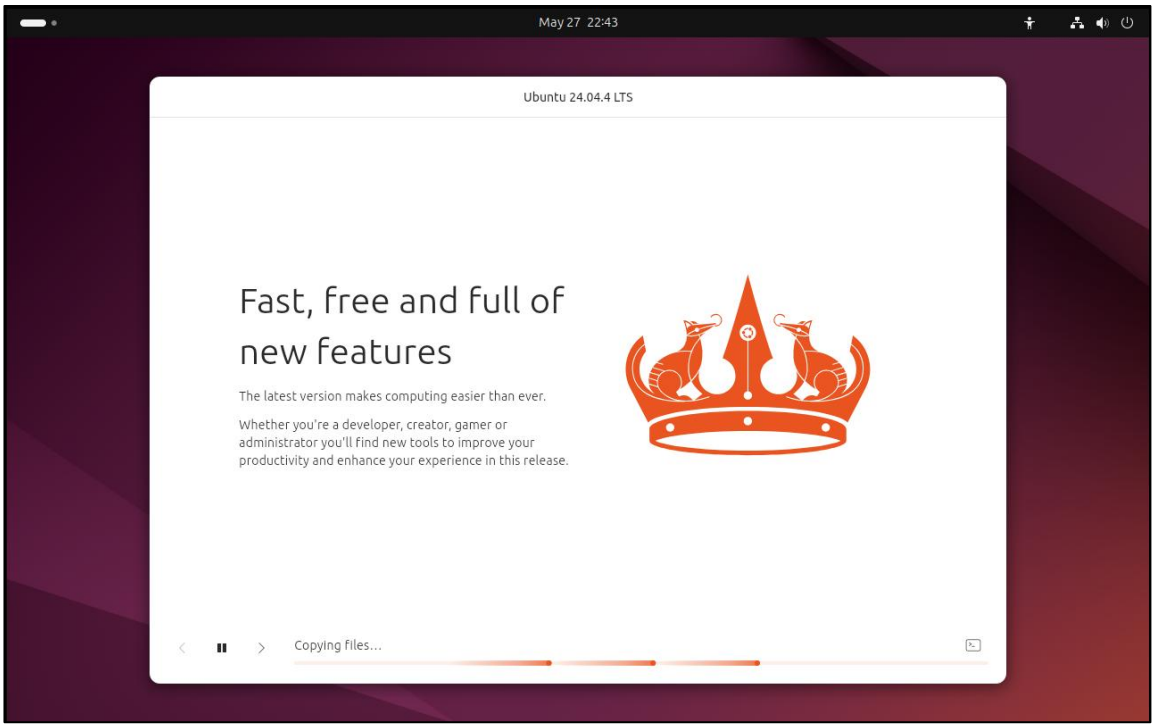


Рисунок 10 – Установка Ubuntu 24.04 LTS

2.3.4 Выбор сред разработки

После установки необходимо загрузить среды разработки и дополнительные зависимости. Для разработки серверной части проекта

используем VS Code. Данная среда идеально подходит для решения текущей задачи.

Она обладает сравнительно низким потреблением системных ресурсов, поддерживает большое количество расширений для языков C/C++ и не перегружает дополнительными расширениями.

В качестве среды для разработки клиентской части была выбрана среда Qt Creator. Она тесно связана с фреймворком Qt и является наиболее удобным решением для разработки кроссплатформенных приложений на языке C++ с Qt.

Данные среды были выбраны для наиболее оптимального решения задачи по разработке многопоточного сервера на C++ и клиентского мониторинга с использованием Qt благодаря сочетанию простоты настройки и поддержки только необходимых инструментов разработки.

2.3.5 Настройка сред разработки

Для установки VS Code выполним обновление пакетного менеджера и установку консольной программы для загрузки файлов по сети. Результат на рисунке 11.

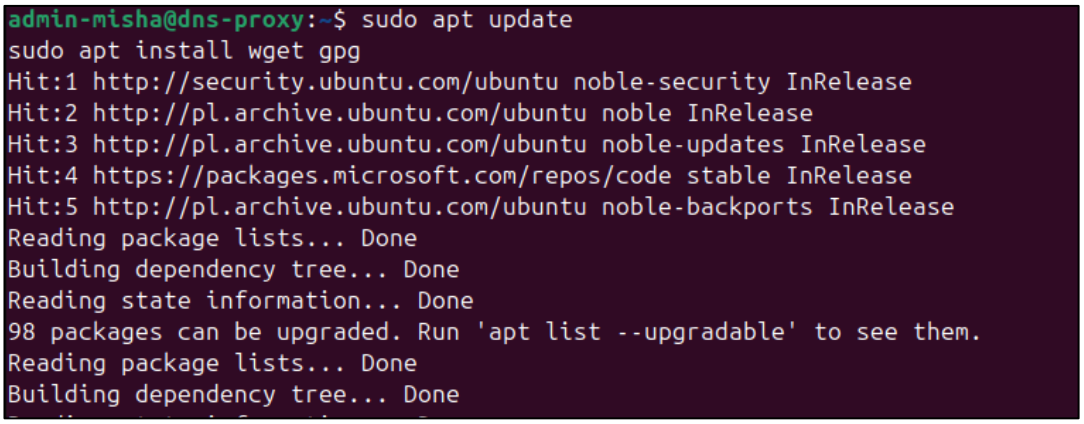


Рисунок 11 – Обновление пакетного менеджера и установка wget

Затем добавим ключ подписи репозитория Microsoft и подключим репозиторий Microsoft для установки code (рисунок 12).

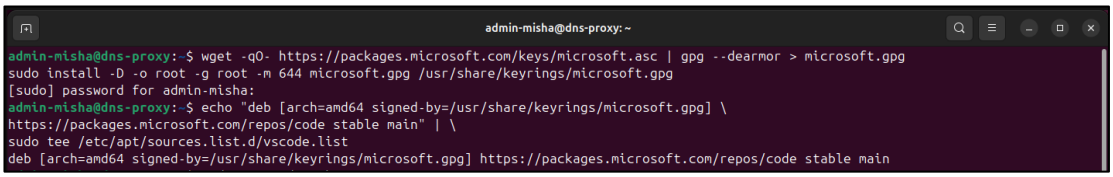


Рисунок 12 – Добавление зависимостей

Установим VS Code (рисунок 13).

```
admin-misha@dns-proxy:~$ sudo apt update
Hit:1 http://pl.archive.ubuntu.com/ubuntu noble InRelease
Hit:2 http://security.ubuntu.com/ubuntu noble-security InRelease
Hit:3 http://pl.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:4 http://pl.archive.ubuntu.com/ubuntu noble-backports InRelease
Get:5 https://packages.microsoft.com/repos/code stable InRelease [3,590 B]
Get:6 https://packages.microsoft.com/repos/code stable/main amd64 Packages [25.8 kB]
Fetched 29.4 kB in 1s (37.5 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
98 packages can be upgraded. Run 'apt list --upgradable' to see them.
admin-misha@dns-proxy:~$ sudo apt install code
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  socat
The following NEW packages will be installed:
  code socat
0 upgraded, 2 newly installed, 0 to remove and 98 not upgraded.
Need to get 159 MB of archives.
After this operation, 726 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Рисунок 13 – Установка VS Code

Перейдем к установке Qt Creator (рисунок 14).

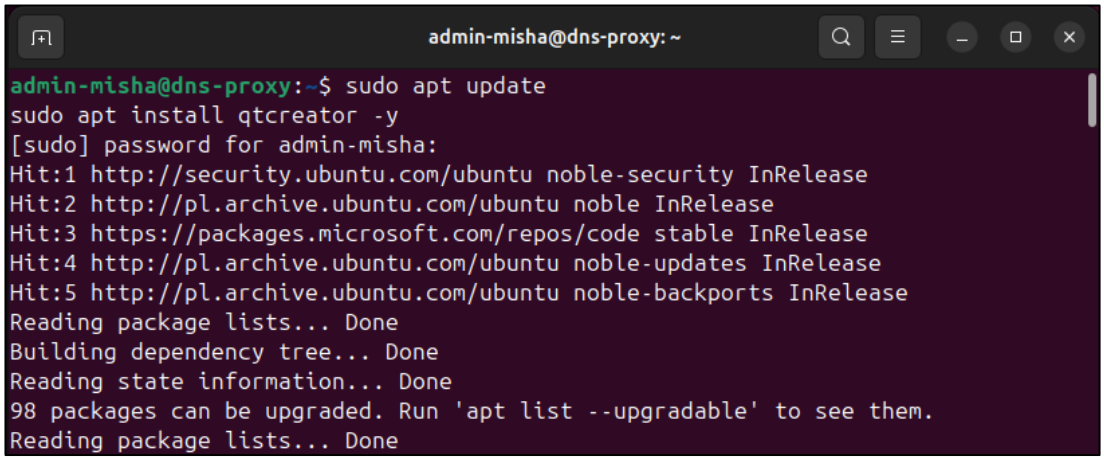


Рисунок 14 – Установка Qt Creator

Установим Qt и необходимые зависимости (рисунок 15).

```
admin-misha@dns-proxy:~$ sudo apt install qt6-base-dev qt6-tools-dev qt6-tools-d
ev-tools -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
qt6-tools-dev-tools is already the newest version (6.4.2-3build3).
qt6-tools-dev-tools set to manually installed.
The following additional packages will be installed:
  libgl-dev libglx-dev libopengl-dev libpthread-stubs0-dev libvulkan-dev
  libx11-dev libxau-dev libxcb1-dev libxdmcp-dev qmake6 qmake6-bin
  x11proto-dev xorg-sgml-doctools xtrans-dev
```

Рисунок 15 – Установка зависимостей Qt Creator

Также установим инструменты сборки (рисунок 16).

```
admin-misha@dns-proxy:~$ sudo apt install build-essential cmake g++ -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  bzip2 cmake-data dpkg-dev fakeroot g++-13 g++-13-x86-64-linux-gnu
  g++-x86-64-linux-gnu gcc gcc-13 gcc-13-x86-64-linux-gnu gcc-x86-64-linux-gnu
  libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl
  libcc1-0 libdpkg-perl libfakeroot libfile-fcntllock-perl libjsoncpp25
  librhash0 lto-disabled-list
```

Рисунок 16 – Установка инструментов сборки и компилятора

Все среды разработки и их зависимости установлены. Результаты приведены на рисунках 9-16.

3 Разработка программного модуля

3.1 Архитектура приложения

Разрабатываемое приложение представляет собой многопоточный DNS-proxy сервер, реализующий механизм пересылки DNS-запросов (forwarding) на внешний DNS-resolver с последующим локальным кэшированием полученных ответов. Архитектура приложения построена по модульному принципу, что позволяет разделить основные компоненты системы по их функциональному назначению и упростить дальнейшее масштабирование проекта.

Общая архитектура приложения включает клиентскую и серверную части. Серверная часть отвечает за прием и обработку запросов, кэширование и возврат результата.

Для более наглядного представления архитектуры разработанного DNS-Сервера и упрощения понимания структуры проекта была построена UML-диаграмма классов, расположенная в приложении (диаграмма 3).

Основные компоненты сервера:

– модуль приема/ответа DNS-запросов (DNSServer);

– модуль сериализации/десериализации DNS-пакета (DNSParser);

– модуль разбора пакета и ресурсных записей (DNSDispatcher);

– модуль кэширования DNS-записей (DNSCache);

– модуль пересылки запросов внешнему DNS-resolver (DNSResolver);

– модуль сбора метрик DNS-сервера (MetricsManager);

– модуль отправки статистики клиентам (Client);

– модуль многопоточной обработки запросов (ThreadPool);

– модуль очереди пакетов (OutQueue);

– модуль дополнительных инструментов (Utils);

– модуль протокола пересылки данных клиенту (Keys).

Реализован поведенческий паттерн «Команда», каждый новый пакет, преобразуется в класс «Task», который в последствие добавляется в очередь

задач [13]. Данное архитектурное решение обеспечивает высокую производительность, за счет возможности распараллеливания обработки пакетов, между несколькими потоками, а также хорошую масштабируемость вследствие зависимости DNS-задач (DNSTask) от интерфейса (Task).

Архитектура сервера построена таким образом, чтобы разделить конкретные задачи на отдельные модули, выстроить четкую последовательность вызовов и обеспечить масштабируемость системы. Классы выполняют единственную задачу, отсутствуют перегруженные компоненты.

Основные компоненты клиента:

– модуль подключения к серверу/получения метрик (Client);

– модуль обработки пакетов и обновления UI (DashboardController);

– модуль отображения верхней панели статистик (ToolBar);

– модуль отображения центральной части статистик (CentralWidget);

– модуль отображения нижней панели статистик (StatusBar)

– базовый модуль графика зависимостей (BaseCustomPlot);

– модуль стиля UI (DarkTheme);

– модуль дополнительных инструментов (Utils);

– модуль протокола пересылки данных клиенту (Keys).

Архитектура клиента реализована с использованием шаблона проектирования MVC (Model-View-Controller), который обеспечивает разделение логики приложения на три взаимодействующих компонента: модель данных, представление и контроллер.

Модель (Model) отвечает за хранение и обработку данных, поступающих от DNS-proxy сервера. Представление (View) реализует графический интерфейс пользователя на базе QtWidgets и обеспечивает визуализацию статистик в виде списков и таблиц. Контроллер (Controller) осуществляет обработку пользовательских событий и взаимодействие между моделью и представлением, а также отвечает за получение обновлений от серверной части по TCP-соединению.

Использование архитектуры MVC обеспечивает слабую связанность компонентов, упрощает сопровождение кода и позволяет независимо изменять логику обработки данных и пользовательский интерфейс без влияния на остальные части системы.

3.2 Разработка приложения

В данном разделе рассматривается реализация основных модулей серверной части proxy DNS-сервера. Каждый модуль отвечает за отдельную функциональную область системы: обработку DNS-пакетов, кэширование, пересылку запросов DNS-resolver, многопоточную обработку задач и сбор статистики работы сервера.

Рассмотрим реализацию сервера.

3.2.1 Реализация модуля DNSServer

При запуске, программа начинает слушать 53 UDP порт. Данный порт отвечает за прием DNS-запросов. Класс сервер, содержит основную нить, которая принимает пакеты и создает Task – DNSTask, которую затем кладет в входную очередь. Если в очереди содержатся пакеты, std::condition_variable будит потоки, которые забирают пакет и начинают с ним работать. Затем они возвращают результирующий пакет в выходную очередь. Когда выходная очередь не пустая, второй поток просыпается и отправляет пакет клиенту. Затем обратно засыпает.

Данный модуль отвечает только за прием/отправку сообщений клиенту и передачу пакета ThreadPool.

3.2.2 Реализация модуля ThreadPool

Данный класс представляет собой пул из 6 потоков, которые отвечают за парсинг и обработку Tasks (задач). Когда сервер получил пакет, сформировал задачу и добавил в очередь, он вызывает метод enqueue(newTask), в котором std::condition_variable будит 1 поток и вызывает у него метод workerLoop(), который вытаскивает следующий пакет из очереди и начинает обработку.

Сначала он сериализует пакет из сырых байт в DNS-пакет используя библиотеку ldns и класс DNSParser. Затем передает уже структурированный пакет в DNSDispatcher для полного его разбора, кэширования и получения результата. После выполнения, DNSDispatcher возвращает готовый пакет, который передается нити, а та кладет его в выходную очередь. На этом работа нити прекращается, и она снова засыпает.

3.2.3 Реализация модуля DNSParser

Данный класс является stateless и содержит 2 статичных метода, это сериализация и десериализация.

При десериализации DNS-пакета, lnds библиотека выделяет память под структуру ldns_pkt в куче (heap). После завершения работы с пакетом необходимо вызвать free(), иначе возникнет утечка памяти. Для автоматического освобождения памяти структура ldns_pkt была обернута в std::unique_ptr с модифицированным deleter. Такое решение позволяет реализовать RAII (Resource Acquisition Is Initialization) идиому и не беспокоиться об утечке.

```
//deleter form ldns_packet
struct PktDeleter {
void operator()(ldns_pkt* pkt) const {
if(pkt) {
ldns_pkt_free(pkt);
}
}
};
```

3.2.4 Реализация модуля DNSDispatcher

Перейдем к самому важному и обширному модулю программы – DNSDispatcher. Данный компонент отвечает за разбор пакета, проверку кэша, resolving и формирование ответного пакета. Когда workerLoop() десериализовал пакет и передал его в DNSDispatcher, dispatch проверяет пакет на флаг Qr, чтобы узнать это ответ или запрос. Если это запрос диспетчер формирует скелет пакета, который в последствие заполнит данными об успешном выполнении или ошибке. Затем вызываем метод получения ресурсных записей RRs и проверяем кэш на наличие абсолютно всех. Если хоть одна запись не найдена, диспетчер

передает пакет резолверу, который пересылает исходный пакет на внешний DNS-resolver (в данной программе используется публичный DNS-resolver Яндекса «77.88.8.8»). Если все записи найдены в кэше, то они получаются из кэша, вытаскивается RDATA соответствующая определенному типу записи и формируется результирующий пакет, который возвращается ThreadPool в workerLoop(). После того, как DNS-resolver выполнил forwarding, он возвращает результат – сам пакет, enum Statuc и строку ошибки. Если возникла ошибка, мы считаем метрики, формируем ошибочный пакет и возвращаем клиенту, иначе, снова получаем все записи RRs и добавляем их в кэш и только потом возвращаем пакет.

3.2.5 Реализация модуля DNSCache

Класс кэша представляет собой std::unordered_map (хэш таблица) с переопределенным hash() и operator==(). Выбор std::unordered_map обусловлен доступом к элементу за O(1) в среднем и за O(n) в худшем случае. Hash представляет собой базовый XOR структур ключа и значения с предварительным побитовым сдвигом значения на 1 влево.

```
struct RHash {
        std::size_t operator()(const Cache::Key& key) const {
            std::size_t h1 = std::hash<std::string>()(key.domain);
            std::size_t h2 =
std::hash<uint32_t>()(static_cast<uint32_t>(key.type));
            return h1 ^ (h2 << 1);
        }
    };
```

Кэш сохраняет записи на время TTL, полученное из ответа внешнего сервера, и выполняет lazy deletion, т.е. при вызове получения значения их кэша, происходит проверка валидности значения TTL, и в случае превышения времени, происходит удаление записи.

```
std::optional<Cache::Record> DNSCache::get(const Cache::Key& key)
{
    std::lock_guard lock(m_mtx);

    auto it = m_records.find(key);
    if(it == m_records.end()) {
        return std::nullopt;
```

```
    }

    if(isExpired(it->second)) {
        std::cerr << "DNSCache::get: Entry TTL finished. Delete
entry" << std::endl;
        m_records.erase(it);
        return std::nullopt;
    }

    return it->second;
}
```

Благодаря использованию механизма std::optional модуль кэширования может явно возвращать результат наличия или отсутствия записи. Это позволяет избежать дополнительных флагов проверки валидности значения.

3.2.6 Реализация модуля DNSResolver

Данный класс представляет собой forwarding переданного пакета на внешний DNS-resolver, с повторной отправкой до окончания времени доступности и проверкой пакета на подлинность. Он не реализует полный рекурсивный резолвер с доступом к root-серверам, а лишь делегирует эту работу публичному серверу.

В данном классе создается 6 thread_local сокетов, что позволяет безопасно работать с сокетами без мьютексов или других объектов синхронизации. Затем в цикле выполняется запрос на внешний сервер, и проверка, что пакет пришел действительно от нужного нам сервера и ID ответа совпадает с ID запроса. Если время истекло и запрос не пришел, возвращается ответный пакет с ошибкой – “TIMEOUT”.

Итогом является указатель на пакет и enum Status.

3.2.7 Реализация модуля OutQueue

Модуль выходной очереди представляет собой std::queue, в которую workerLoop() добавляет готовый ответ. Как только в очереди появляется запись, std::condition_variable будит поток sender и передает следующий пакет в очереди для возврата клиенту.

3.2.8 Реализация модуля Utils

Данный класс содержит различные функции и структуры в различных областях видимости. Этот модуль – набор инструментов, не относящихся к конкретному классу, а использующихся во многих местах. Тут содержатся методы преобразования DNS-типов во внутренние и обратно, структуры различных метрик и методы их преобразования в JSON.

3.2.9 Реализация модуля MetricsManager

Класс MetricsManager представляет собой сбор статистики о всем сервере. Для повышения производительности в этом классе также реализована отдельная нить, ответственная за сбор, сериализацию и передачу пакета для отправки клиентам. Каждую секунду поток просыпается и начинает собирать данные со всех частей.

Он собирает такие метрики, как:

– status – статус сервера;

– uptime – время работы сервера до 24ч;

– request / sec – число запросов за 1 секунду;

– cache hit rate – процент пакетов, которые не пришлось перенаправлять. Они были взяты из кэша;

– latency p95 – 95-ый перцентиль, нижний порог времени быстрее которого обработаются 95% всех пакетов. Оно точнее показывает среднее время обработки пакета и не чувствителен к всплескам, в отличие он avg;

– resolve avg – средняя время обработки запроса на этапе обращения к внешним DNS-resolver, включающее отправку DNS-запроса, ожидание и получение ответа;

– errors – процент ошибочных пакетов;

– top domains – список из структур, содержащих топ 10 самых популярных доменов и число обращений к ним;

– query types – список самый распространенных записей в кэше (макс.– 4);

– recent errors – список структур, содержащих точное время, домен и ошибку. Представляет таблицу последних ошибок;

– cache entries – количество записей в кэше;

– total requests – общее количество запросов к DNS-серверу;

– RAM – размер занятого пространства в RAM (в MB);

– CPU – процент использования процессорного времени процессом DNS-proxy, относительно общего времени его работы;

– общее количество существующих поток процесса на текущий момент.

Системные метрики (число потоков и размер занятого пространства) загружаются из файла процесса «/proc/self/status» [14].

```
uint32_t MetricsManager::getStatusMetric(const std::string&
metric) const {
    std::ifstream stat("/proc/self/status");
    std::string line;

    while(getline(stat, line)) {
        if(line.compare(0, metric.size(), metric) == 0) {
            std::string value = line.substr(metric.size());
            return std::stoul(value);
        }
    }

    return 0;
}
```

Метод createSnapshot создает структуру со всеми метриками. Затем с помощью библиотеки nlohmann/json производится сериализация всей структуры в JSON для последующей передачи пакета всем клиентам по TCP порту.

3.2.10 Реализация модуля Client

Модуль клиента сход с модулем сервера. В нем поток слушает соединения на зарезервированном порту 6060 и добавляет клиентский сокет в список. Затем MetricsManager начинает рассылку всем подключенным клиентам, получит их у Client и отправляет всем пакеты со статистикой.

3.2.11 Реализация модуля Keys

Данный модуль представляет собой список ключей, по которым будет осуществляться чтение/запись метрик. Этот модуль был продублирован для клиента, чтобы не допустить ошибки при десериализации пакета.

Рассмотрим реализацию клиента. Клиент написан с использованием классического модуля Qt – Qt Widgets.

3.2.12 Реализация модуля Client

Класс клиент представляет собой класс, осуществляющий подключение к серверу. Класс посылает сигнал, при корректном подключении, при отключении от сервера и при ошибке. При корректном подключении слот onReadyRead начинает получать пакеты от сервера, десериализовать из JSON и передавать их в контроллер для распределения метрик по категориям. При отключении от сервера, включается метод переподключения к серверу с интервалом в 5 секунд. Как только программа запускается, сразу же вызывается отображение основного окна с последними метриками (при старте программы метрики пустые, а статус сервера – stop). Если клиент отключился или сервер завершил работу по определенным причинам, dashboard-окно остается показываться с последними метриками. Такой подход дает возможность увидеть последнее состояние программы до падения.

3.2.13 Реализация модуля DashboardController

Данный класс содержит три главных компонента представления: ToolBar, CentralWidgets и StatusBar и обеспечивает взаимодействие между ними и данными, полученными из класса Client. Контроллер разбивает всю систему метрик на 3 категории, относя необходимые метрики в разные структуры и передавая их разным представлениям. Кроме того, он соединяет сигналы о соединении и разрыве сервера с UI, что позволяет своевременно обновлять статистику.

3.2.14 Реализация модуля ToolBar

Один из трех главных компонентов представления – верхняя панель метрик. Тут содержится самая главная информация, необходимая для просмотра.

Весь ToolBar представляет собой RowLayout с разбиением на отдельные метрики. Каждая метрика отображается в своей ячейке с дополнительным текстом, что позволяет понять смысл значения. Как и все 3 компонента UI

ToolBar содержит метод обновления статистики updateState(), вызываемый контроллером при получении данных.

3.2.15 Реализация модуля CentralWidget

Центральная часть представляет собой Grid (2×3). На верхней строке представлены три графика:

– график зависимости количества запросов в секунду от времени;

– график зависимости количества попаданий запросов из кэша от времени;

– график зависимости 95 перцентиля обработки пакета от времени.

Ниже идут списки. Список самых популярных доменов с их названием и процентом обращений от всех обращений.

Затем Grid (2×2) с информацией о самых популярных типах в кэше (до 4 записей). Показывает название типа и процент занятости в кэше.

Последним отображается список ошибок, полученных во время формирования пакета. Строка записи содержит точное время, доменное имя и ошибку.

3.2.16 Реализация модуля BaseCustomPlot

Данный компонент представляет собой базовый класс-график с установленным стилем и методами. Используется в качестве родителя при создании трех графиков зависимости метрики от времени с периодом обновления – 1 секунда. Использует внешнюю библиотеку QCustomPlot, для более гибкой настройки графика и его вида.

3.2.17 Реализация модуля StatusBar

StatusBar последний компонент UI, расположенный снизу окна. Тут содержится менее важная, но также полезная статистика, информирующая больше о количественной характеристике сервера (количество записей в кэше, количество запросов к серверу и т.д.). Также содержит updateState().

3.2.18 Реализация модуля DarkTheme

Данный модуль предназначен для установки глобального стиля приложения и его компонентов, шрифтов, общих цветов и типов.

3.2.19 Реализация модуля Utils

Данный модуль, как и в серверной части содержит вспомогательные инструменты разработки. Тут содержатся все структуры метрик и структура пакета Snapshot, а также методы парсинга цельного пакета на 3 части.

3.2.20 Реализация модуля Keys

Как уже было сказано ранее, модуль Keys, соответствует модулю Keys из серверной части для корректного чтения/записи пакетов.

Таким образом была разработана и рассмотрена программа многопоточного проксирующего DNS-сервера с возможностью локального кэширования и поддержкой пересылки DNS-запроса на внешний сервер, а также программа мониторинга состояния работы системы в режиме реального времени.

3.3 Описание приложения

3.3.1 Сборка серверной части

Перед началом сборки необходимо установить из репозитория proxy DNS-сервер – https://github.com/megawalnut/proxy_dns.git. Также необходимо установить дополнительные библиотеки.

Установка пакетного менеджера выполняется с помощью пакетного менеджера apt:

– sudo apt install build-essential gcc g++ cmake git pkg-config -y;

– sudo apt install nlohmann-json3-dev;

– sudo apt install libldns-dev ldnsutils -y;

– sudo apt install qt6-base-dev qt6-base-dev-tools;

– sudo apt install libqt6svg6-dev.

Сборка серверной части выполняется с использованием компилятора GNU C++ (g++). Для компиляции используется возможности стандарта C++17.

Компиляция осуществляется следующей командой:

```
g++ -std=c++17 $(find sources -name "*.cpp") -I headers -o build/dns -lldns
```

При этом используются следующие параметры компиляции:

– -std=c++17 – использование минимального стандарта языка;

– -I headers – подключение каталога заголовочных файлов;

– -o build/dns – формирование исполняемого файла dns в каталогbuild;

– -lldns – подключение библиотеки ldns для работы с DNS-пакетами.

В результате компиляции формируется исполняемый файл DNS-proxy сервер, готовый к запуску в ОС Linux.

3.3.2 Подготовка к запуску и запуск DNS-сервера

Перед запуском DNS-сервера необходимо отключить системный DNS resolver, который слушает порт 53. В ОС Ubuntu данную функцию выполняет system-resolved. Для этого выполним команды и сравним с рисунком 17:

– `systemctl stop systemd-resolved.service`;

– `systemctl disable systemd-resolved.service`.

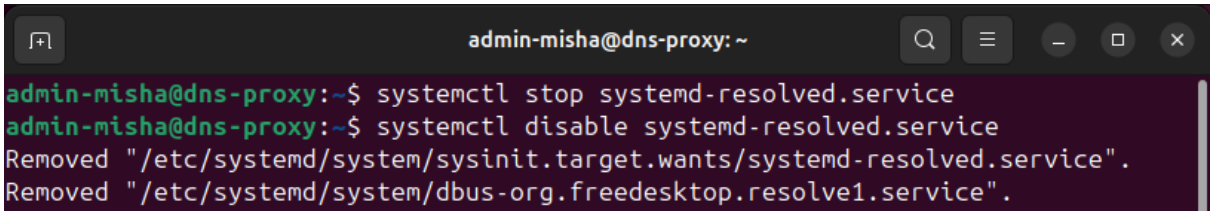


Рисунок 17 – Отключение системной службы system-resolved

Выключив системную службу, необходимо изменить адрес DNS-сервера на нашу программа (рисунок 18). Для этого:

– откроем файл /etc/resolve.conf с правами суперпользователя;

– в файле изменим адрес nameserver 127.0.0.53 на 127.0.0.1.

После изменений, ОС будет отправлять запросы на локальный DNS-сервер, запущенный на адресе 127.0.0.1 и порту 53.

Запустим исполняемый файл dns с правами суперпользователя:

```
sudo ./build/dns
```

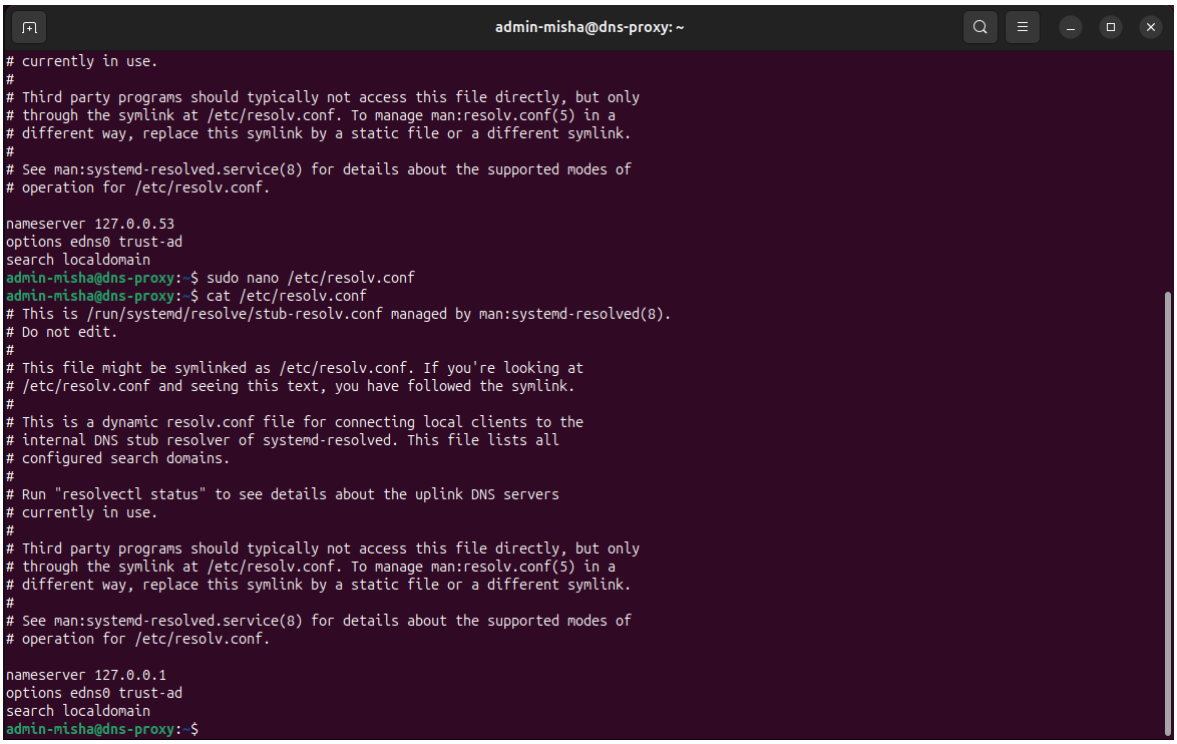


Рисунок 18 – Изменение адреса системного обработчика DNS-запросов

Произведем запуск DNS-сервера. Результат представлен на рисунке 19.

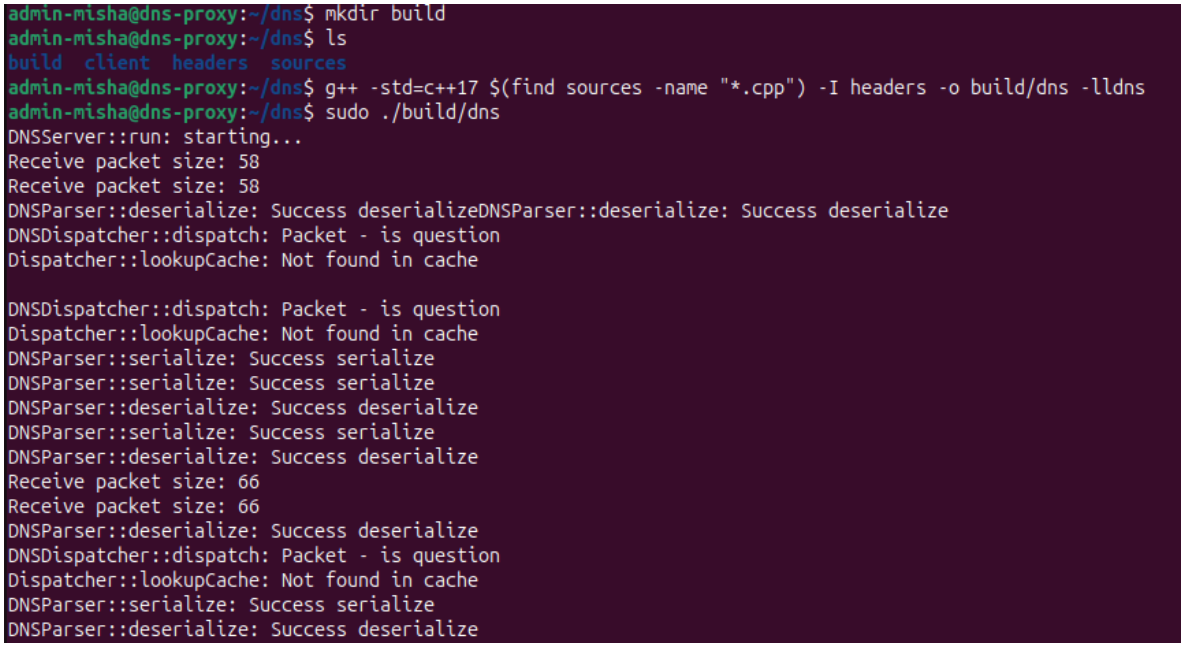


Рисунок 19 – Запуск DNS-proxy сервера

3.3.3 Запуск DNS-клиента

Для запуска DNS-клиента необходимо открыть qtcreator и собрать проект, нажав комбинацию клавиш «Ctrl + R». После сборки запустится клиент и выведется окно метрик (рисунок 20).

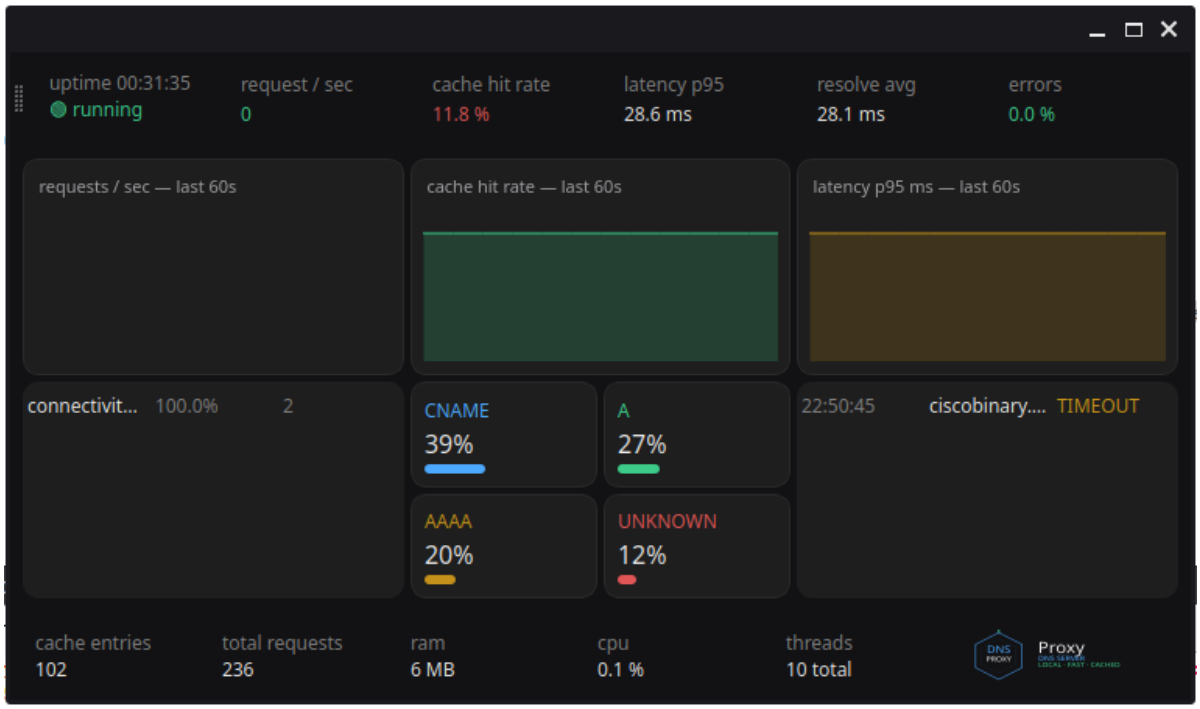


Рисунок 20 – Запуск DNS-клиента

Для большего понимания порядка взаимодействия объектов системы при выполнении DNS-обработки запроса, были разработаны диаграммы последовательности для общего DNS-запроса и для системы сбора метрик. Диаграммы расположены в приложении (диаграмма 1, 2). Кроме того, для описания физической архитектуры системы, включая вычислительные узлы и способы их размещения была создана диаграмма развертывания (диаграмма 4).

3.3.4 Тестирование

Для оценки производительности DNS-proxy сервера было проведено тестирование оценки скорости обработки DNS-пакетов в разных режимах программ: проксирующий DNS-сервер и systemd-resolved.

В качестве инструмента тестирования использовалась утилита dig, позволяющая формировать DNS-запросы и измерять время получения ответа от сервера.

Для удобства проверки был создан скрипт test.sh. В нем реализован цикл по заданному списку доменов. Внутри цикла вызывается команда получения информации конкретной записи о конкретном домене. Код скрипта представлен в приложении Б.

Проверим скорость обработки запросов на пустом кэше и на заполненном, а также добавим для сравнения время обработки пакета при обращении через systemd-resolved. Проверили скорость обработки DNS-запросов на пустом кэше проксирующего DNS-сервера. Результат на рисунке 21.

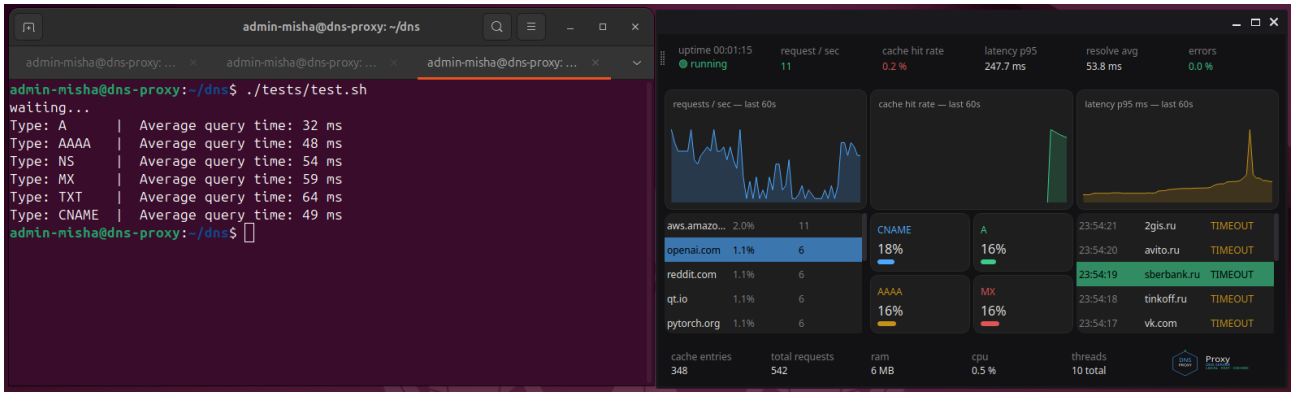


Рисунок 21 – Время обработки запросов с пустым кэшем

Проверили скорость обработки DNS-запросов на заполненном кэше проксирующего DNS-сервера. Процент запросов, обработанных из кэша превышает 50%. Результат на рисунке 22.

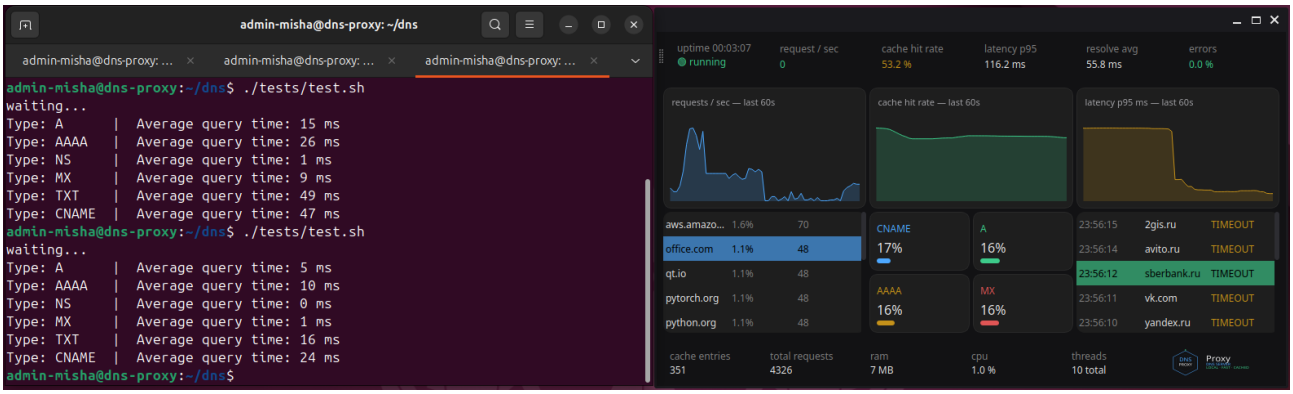


Рисунок 22 – Время обработки запросов с заполненным кэшем

Проверили скорость обработки DNS-запросов через system-resolved. Результат на рисунке 23.

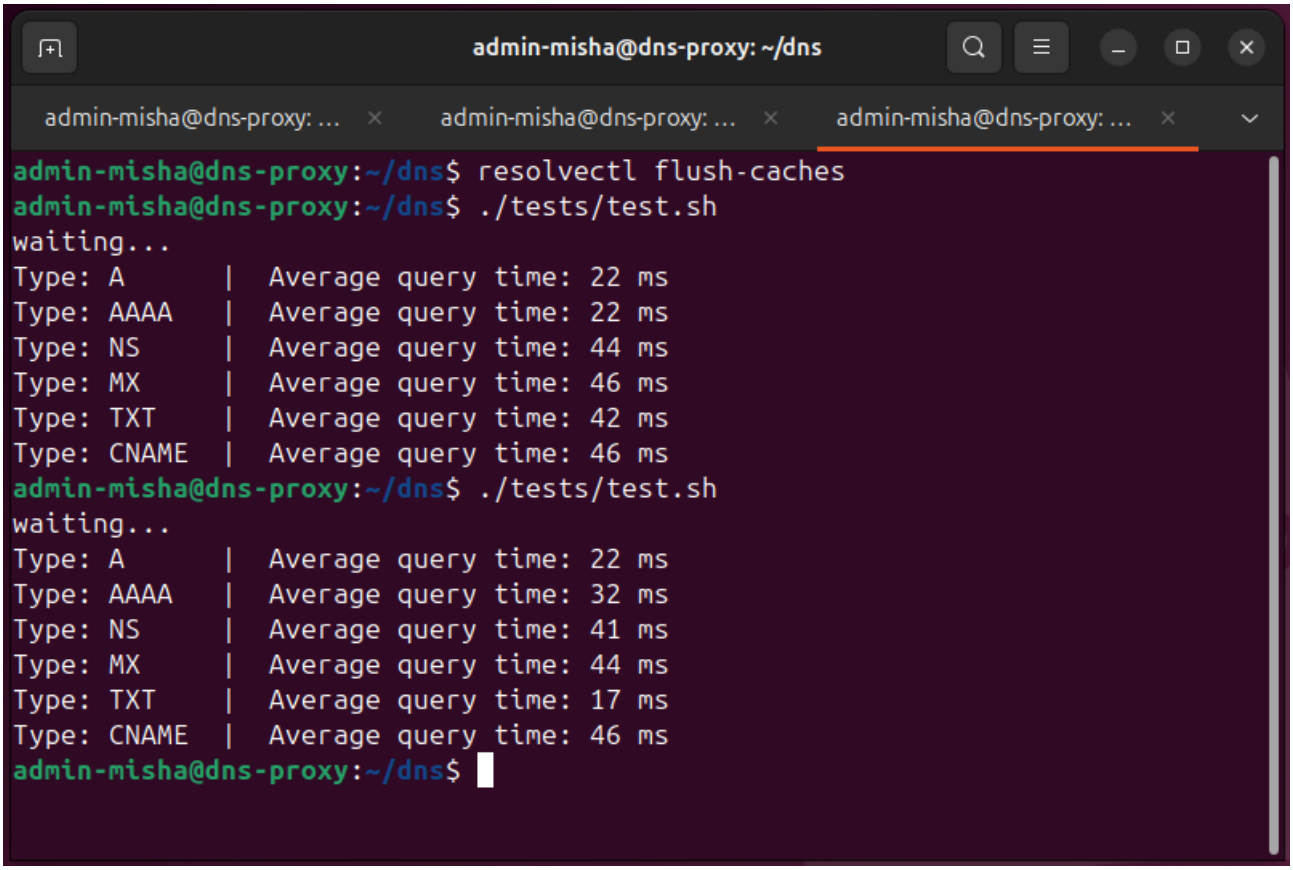


Рисунок 23 – Время обработки запросов через system-resolved

Результаты приведены в следующей таблице 2.

Таблица 2 – Результаты значений скорости обработки DNS-запросов, обрабатываемые программами в различных состояниях

Тип записи	DNS-proxy (пустой кэш), мс	DNS-proxy (полный кэш), мс	system-resolved, мс
A	32	5	22
AAAA	48	10	22
NS	54	0	41
MX	59	1	44
TXT	64	16	17
CNAME	49	24	46

Изучив таблицу, можно сделать выводы. При пустом кэше, собственный DNS-сервер обрабатывает запросы в пределах от 30 до 65 мс. Когда кэш программы заполнился и процент запросов, обработанных из кэша, достиг более 50%, время обработки пакетов снизилось до диапазона [0:25]. Из-за того, что нельзя более гибко управлять состоянием systemd-resolved, его наилучшее время соответствует времени обработки пакета из кэша, а наихудшее – время resolving.

Изучив результаты тестирования, можно сделать вывод, что полученные результаты подтверждают эффективность кэширования записей.

Сравнение с системным сервисом показало схожие результаты. Но из-за того, что не до конца известно, взяты ли запросы из кэша или получены от внешнего сервера, нельзя с точностью сравнить данные программы. Полученные данные представляют собой усредненное значение обработки DNS-запросов разными программами в различных состояниях.

4 Безопасность жизнедеятельности

В данном разделе рассматриваются ключевые требования к безопасности и рекомендации по организации работы за персональным компьютером.

При разработке проксирующего DNS сервера были использованы средства ЭВМ, поэтому необходимо обеспечить безопасность условий труда при работе с персональным компьютером.

Расположение ЭВМ и помещение должно соответствовать санитарно-гигиеническим требованиям [15].

Естественное и искусственное освещение должно соответствовать нормам. Рекомендуется располагать окна в помещениях с ЭВМ на север и северо-восток. Минимальная площадь рабочего места – 6м^2. Для внутренней отделки интерьера помещений должны использоваться диффузно-отражающие материалы с коэффициентом отражения для потолка -0,7 -0,8; для стен -0,5 -0,6; для пола -0,3 -0,5. Помещения должны быть оборудованы защитным заземлением в соответствии с техническими требованиями по эксплуатации.

Монитор располагается на расстоянии 50-70 см от пользователя, а уровень освещенности должен обеспечивать комфортную работу.

Не допускается превышение нормы содержания вредных химических веществ в воздухе. Не допускается установка шумящих устройств, уровни шума которого превышают норму.

В помещении должны соблюдаться требования пожарной безопасности. Компьютерное оборудование должно подключаться только к исправной электрической сети, а использование поврежденных кабелей не допускается.

Рабочее место должно быть оснащено средствами пожаротушения, а пользователь должен знать правила использования средств и порядок действий при возникновении пожара.

Соблюдение перечисленных требований необходимо для создания безопасных условий труда при работе с персональным компьютером.

Заключение

В ходе выполнения выпускной квалификационной работы были решены следующие задачи:

– проведен анализ предметной области и существующих аналогов проксирующих DNS серверов;

– разработана система программ, состоящая из проксирующего сервера и системы мониторинга в режиме реального времени;

– протестированы и сопоставлены: проксирующий DNS сервер и system-resolved в различных состояниях на одинаковых наборах данных.

В рамках дипломной работы был разработан многопоточный DNS-сервер с поддержкой пересылки запросов на внешний узел и кэширования записей на время TTL. Добавлен модуль сбора и отправки статистики о сервере программе, отвечающей за отображение метрик с графическим интерфейсом.

Разработанная система программ была сопоставлена с системным сервисом system-resolved ОС Linux. Результатом сравнения стала таблица, отражающая эффективность кэширования записей. Программа продемонстрировала схожие результаты с усредненными значениями скорости обработки DNS-запросов системным сервисом.

Дальнейшее развитие системы программ может включать различные улучшения:

– реализация других DNS протоколов;

– добавление поддержки флага TC;

– добавление новых DNS-типов записей;

– реализация полноценного resolving;

– оптимизация формирования DNS-ответов, посредством сложного составления ответа;

– добавление модуля логирования в файл;

– добавление персистентного кэша.

Список использованных источников

1. Ли К., Альбитц П. DNS и BIND, 5-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008. – 712 с.
2. Mockapetris P. RFC 1034. Domain names – concepts and facilities [Электронный ресурс]. – URL: https://www.rfc-editor.org/rfc/rfc1035 (дата обращения: 08.04.2026).
3. Mockapetris P. RFC 1035. Domain names – implementation and specification [Электронный ресурс]. – URL: https://www.rfc-editor.org/rfc/rfc1035 (дата обращения: 08.04.2026).
4. Таненбаум Э., Уэзеролл Д. Компьютерные сети. 5-е изд. – СПб.: Питер, 2012. – 960 с.
5. Hu Z., Zhu L., Heidemann J., Mankin A., Wessels D., Hoffman P. RFC 7858. Specification for DNS over Transport Layer Security (TLS) [Электронный ресурс]. – URL: https://www.rfc-editor.org/rfc/rfc7858 (дата обращения: 10.04.2026).
6. DNS over TLS vs. DNS over HTTPS | Secure DNS [Электронный ресурс]. – URL: https://www.cloudflare.com/learning/dns/dns-over-tls/ (дата обращения: 10.04.2026).
7. DNS over TLS [Электронный ресурс]. – URL: https://developers.cloudflare.com/1.1.1.1/encryption/dns-over-tls (дата обращения: 10.04.2026).
8. DNS и разрешение имен [Электронный ресурс] – URL: https://vasilisa-l.gitbook.io/blue-team-cookbook/other/dns (дата обращения: 10.04.2026).
9. Уязвимости в Dnsmasq, позволяющие подменить содержимое в кэше DNS [Электронный ресурс] – URL: https://www.opennet.ru/opennews/art.shtml?num=54436 (дата обращения: 11.04.2026).

10. C++ Reference [Электронный ресурс]. – URL: https://cppreference.com (дата обращения: 01.05.2026).

11. LDNS Documentation [Электронный ресурс]. – URL: https://nlnetlabs.nl/documentation/ldns (дата обращения: 01.05.2026).

12. JSON for Modern C++ [Электронный ресурс]. – URL: https://github.com/nlohmann/json (дата обращения: 01.05.2026).

13. Command [Электронный ресурс] – URL: https://refactoring.guru/ru/design-patterns/command (дата обращения: 17.05.2026).

14. getrusage(2) – Linux manual page [Электронный ресурс]. – URL: https://man7.org/linux/man-pages/man2/getrusage.2.html (дата обращения: 11.05.2026).

15. Гигиенические требования к персональным электронно-вычислительным машинам и организации работы. Санитарно-эпидемиологические правила и нормативы СанПиН 2.2.2/2.4.1340-03. [Электронный ресурс]. – URL: https://normativ.kontur.ru/document?moduleId=9&documentId=108885 (дата обращения: 18.05.2026).

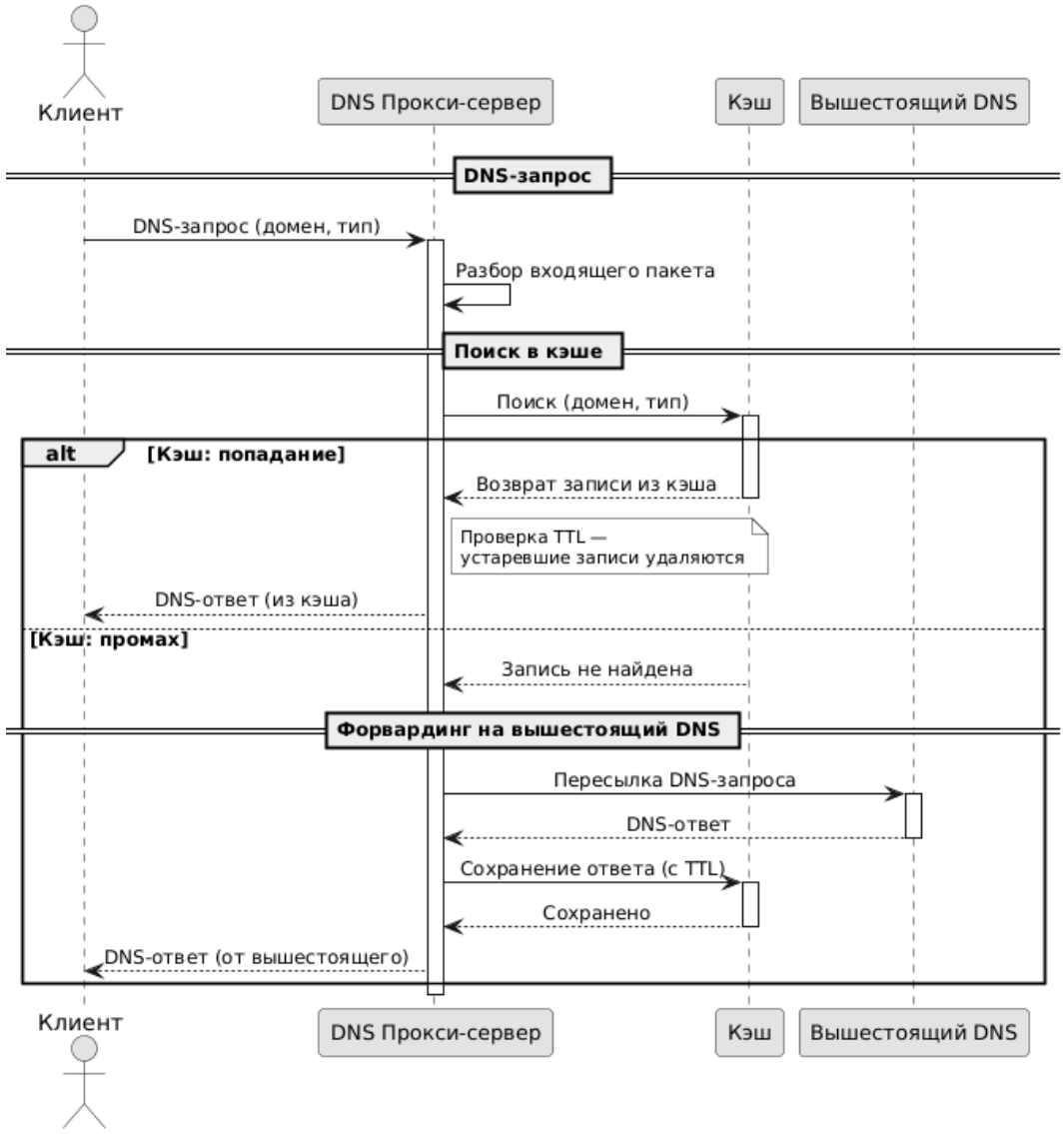
Приложение А

(обязательное)

Проксирующий DNS сервер

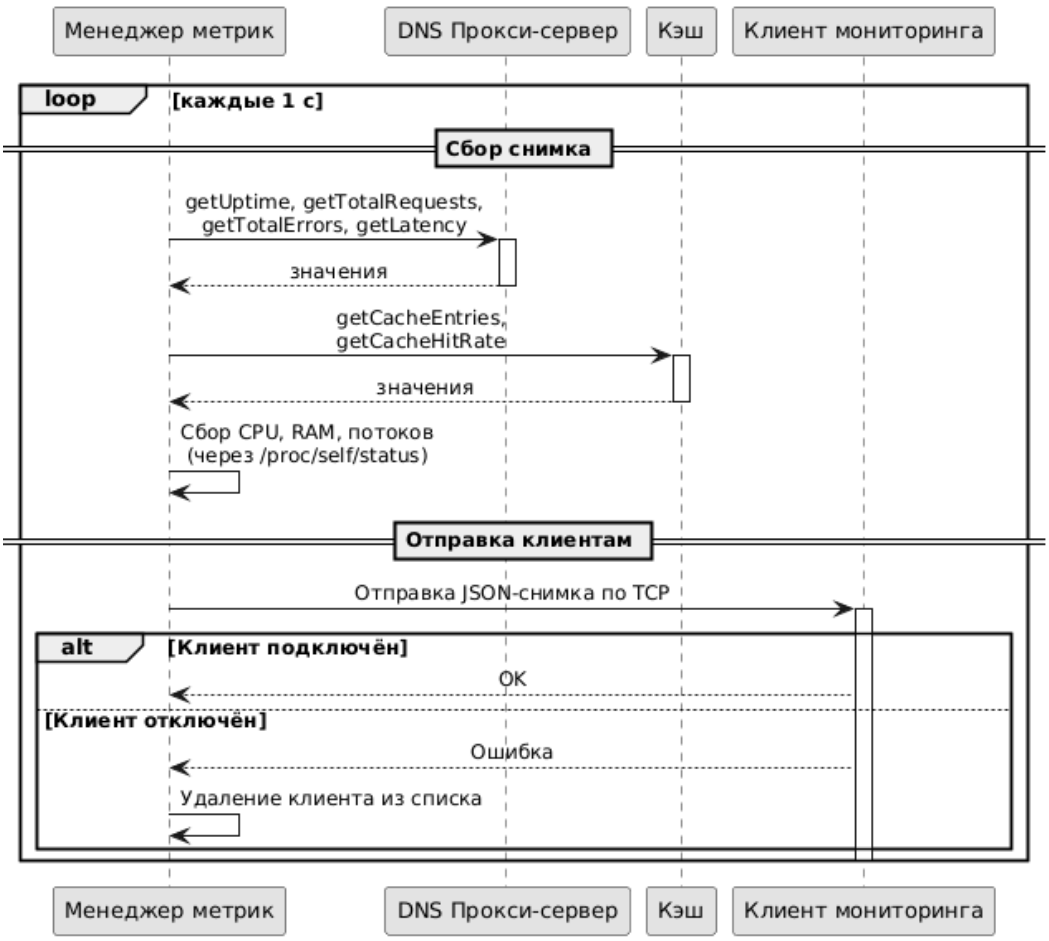
UML-диаграммы

UML-диаграмма последовательностей обработки DNS-запросов



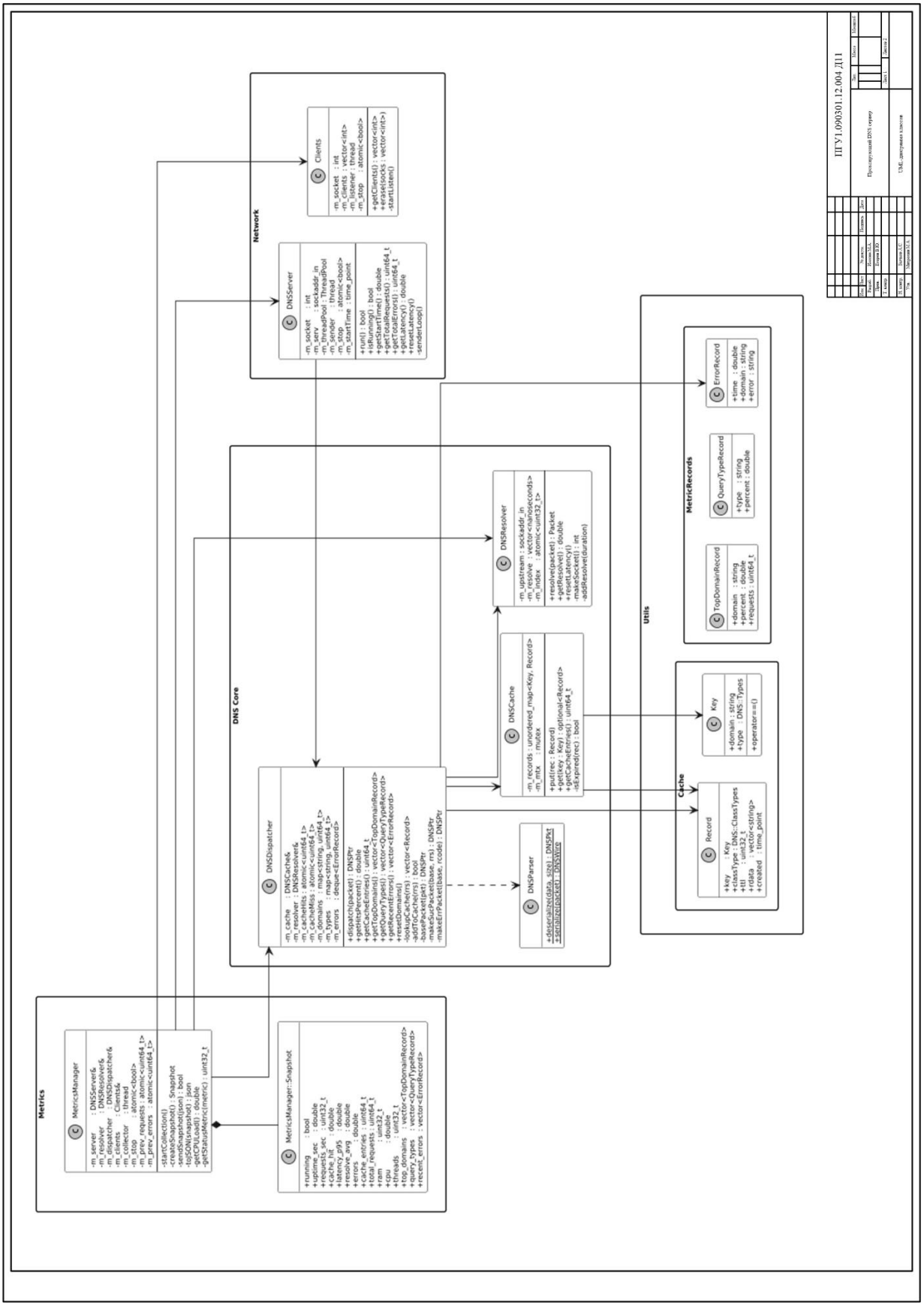
Изм.	Лист	№ докум.	Подпись	Дата	ПГУ1.090301.12.001 ПЗ	Лист 52

UML-диаграмма последовательностей сбора метрик

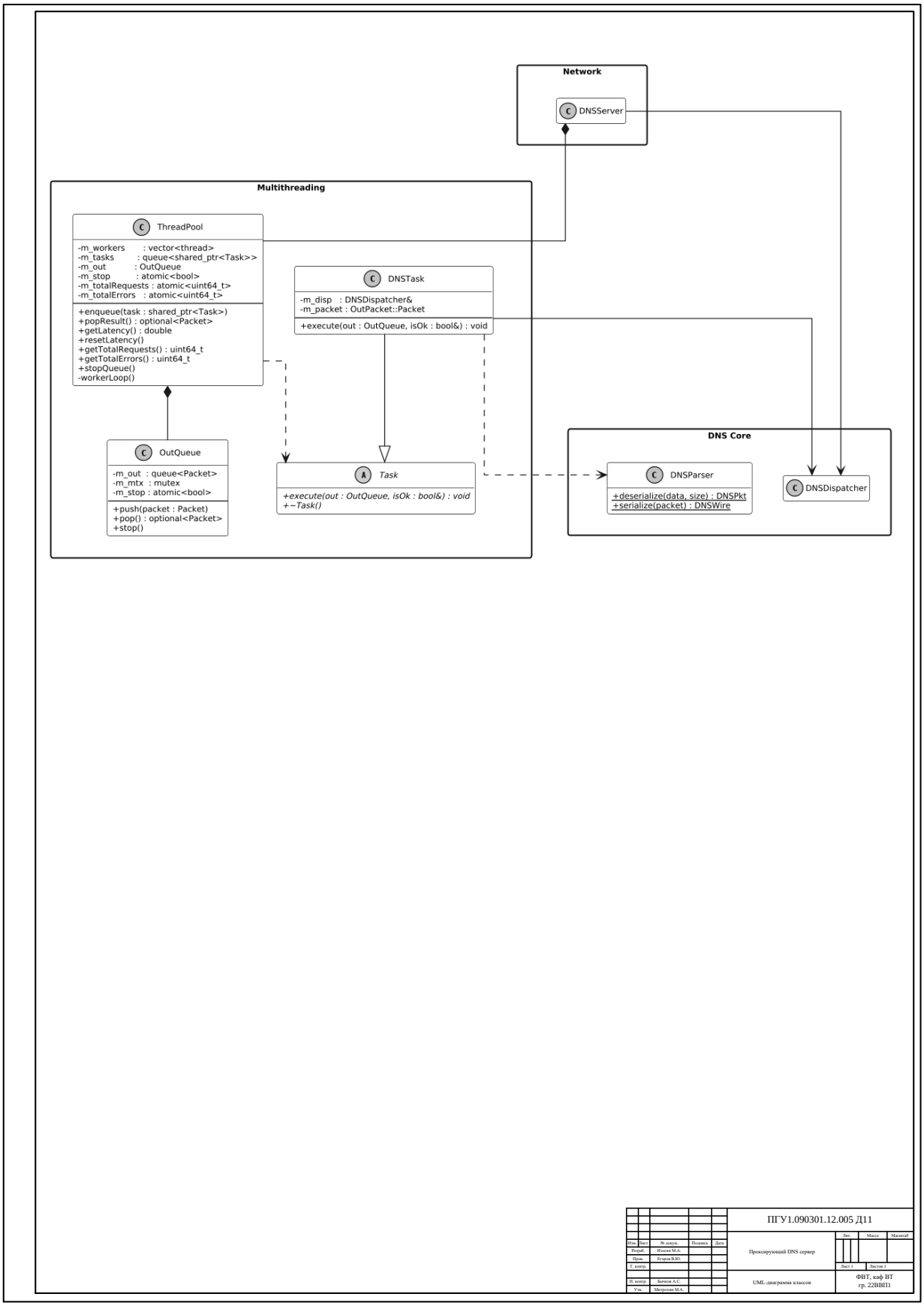


					ПГУ1.090301.12.003 Д14		
Изм.	Лист	№ докум.	Подпись	Дата			
Разраб.		Ионин М.А.			Проксирующий DNS сервер	Лит.	Масса	Масштаб
Пров.		Егоров В.Ю.						
Т. контр.						Лист 1	Листов 1	
Н. контр.		Бычков А.С.			UML-диаграмма последовательностей сбора метрик	ФВТ, каф ВТ гр. 22ВВП1		
Утв.		Метрошин М.А.						

UML-диаграмма классов

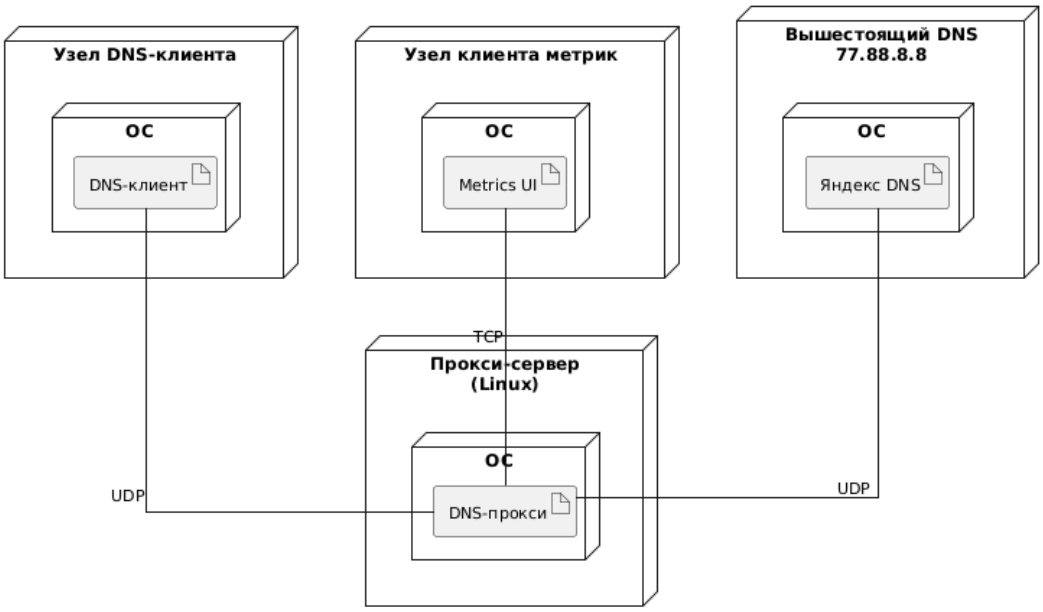


UML-диаграмма классов



					ПГУ1.090301.12.005 Д11	
Изм.	Лист	№ докум.	Подпись	Дата		
Разраб.		Исхаков М.А.			Проксирующий DNS сервер	Лит. / Масса / Масштаб
Пров.		Егоров В.Ю.				
Т. контр.						Лист 1 / Листов 1
Н. контр.		Бычков А.С.			UML-диаграмма классов	ФВТ, каф ВТ гр. 22ВВП1
Утв.		Мисрихан М.А.				

UML-диаграмма развертывания



					ПГУ1.090301.12.006 Д19	
Изм.	Лист	№ докум.	Подпись	Дата		
Разраб.		Исаков М.А.			Прокси́рующий DNS сервер	Лит. / Масса / Масштаб
Пров.		Егоров В.Ю.				
Т. контр.						Лист 1 / Листов 1
Н. контр.		Баянов А.С.			UML-диаграмма развертывания	ФВТ, каф ВТ гр. 22ВВП1
Утв.		Мигунова М.А.				

Приложение Б

(обязательное)

Проксирующий DNS сервер

Листинг программы

Файл test.sh

```
domains=(
"cloudflare.com"
"dns.google"
"github.com"
"stackoverflow.com"
"stackexchange.com"
"superuser.com"
"serverfault.com"
"wikipedia.org"
"wikimedia.org"
"mozilla.org"
"ubuntu.com"
"debian.org"
"kernel.org"
"archlinux.org"
"qt.io"
"python.org"
"cplusplus.com"
"cppreference.com"
"openai.com"
"chatgpt.com"
"microsoft.com"
"azure.com"
"office.com"
"outlook.com"
"apple.com"
"icloud.com"
"amazon.com"
"aws.amazon.com"
"reddit.com"
"linkedin.com"
"netflix.com"
"spotify.com"
"discord.com"
"telegram.org"
"zoom.us"
"dropbox.com"
"adobe.com"
"oracle.com"
"intel.com"
"amd.com"
"nvidia.com"
"docker.com"
"kubernetes.io"
"gitlab.com"
"bitbucket.org"
"atlassian.com"
"digitalocean.com"
"linode.com"
```

```
"akamai.com"
"fastly.com"
"nginx.org"
"apache.org"
"mysql.com"
"postgresql.org"
"mongodb.com"
"elastic.co"
"grafana.com"
"prometheus.io"
"jetbrains.com"
"php.net"
"golang.org"
"rust-lang.org"
"nodejs.org"
"npmjs.com"
"pypi.org"
"tensorflow.org"
"pytorch.org"
"huggingface.co"
"cnn.com"
"bbc.com"
"reuters.com"
"nytimes.com"
"theguardian.com"
"forbes.com"
"techcrunch.com"
"arstechnica.com"
"ycombinator.com"
"yandex.ru"
"vk.com"
"habr.com"
"lenta.ru"
"tinkoff.ru"
"sberbank.ru"
"gosuslugi.ru"
"ozon.ru"
"wildberries.ru"
"avito.ru"
"2gis.ru"
"rutube.ru"
"kaspersky.com")

types=(
"A"
"AAAA"
"NS"
"MX"
"TXT"
"CNAME")
```

```
    echo "waiting..."
    for type in "${types[@]}"; do
      sum=0
      count=0
      for domain in "${domains[@]}"; do
        time=$(dig "$domain" "$type" +tries=1 +time=1 | grep "Query
time:" | awk '{print $4}')
        sum=$((time + sum))
        ((count++))
      done
      avg=$((sum / count))
      printf "%-11s %-3s %-10s\n" "Type: $type" " | " "Average query
time: $avg ms"
    done
```

Файл main.cpp

```
#include "../headers/Net/dnsServer.h"
#include "../headers/DNS/dnsResolver.h"
#include "../headers/DNS/dnsDispatcher.h"
#include "../headers/Metrics/metricsManager.h"
#include "../headers/Net/clients.h"

static constexpr inline const char* DNS_RESOLVER = "77.88.8.8";

int main(int argc, const char* argv[]) {
    Clients clients;
    DNSCache cache;
    DNSResolver resolver(DNS_RESOLVER);
    DNSDispatcher dispatcher(cache, resolver);
    DNSServer server(dispatcher);
    MetricsManager metrics(server, resolver, dispatcher, clients);

    if(!server.run()) {
        perror("Main::main: Failed to start server");
        return -1;
    }
    return 0;
}
```

Файл dnsServer.h

```
#ifndef DNSSERVER_H
#define DNSSERVER_H

#include <iostream>
#include <sys/socket.h>
#include <atomic>

#include "../DNS/dnsParser.h"
#include "../DNS/dnsDispatcher.h"
#include "../Multithreading/threadPool.h"
#include "../Multithreading/Tasks/task.h"
```

```
#include "../Multithreading/Tasks/dnsTask.h"
#include "../Common/utils.h"

using namespace Utils;

class DNSServer final {
    // we specifically limit the buffer, since the proxy does not
support TC
    static constexpr inline std::size_t BUFFER_SIZE = 512;
    static constexpr inline int UDP_DNS_PORT = 53;

public:
    DNSServer(DNSDispatcher& disp);
    ~DNSServer();

    bool run();

    bool isRunning() const;
    double getStartTime() const;
    uint64_t getTotalRequests() const;
    uint64_t getTotalErrors() const;
    double getLatency();
    void resetLatency();

private:
    void senderLoop();

private:
    int m_socket = -1;
    sockaddr_in m_serv;
    DNSDispatcher& m_dispatcher;

    ThreadPool m_threadPool;
    std::thread m_sender;
    std::atomic<bool> m_stop {};
    std::chrono::steady_clock::time_point m_startTime;
};

#endif // DNSSERVER_H
```

Файл dnsServer.cpp

```
#include "../../headers/Net/dnsServer.h"

DNSServer::DNSServer(DNSDispatcher& disp) : m_dispatcher(disp) {
    m_serv.sin_family = AF_INET;
    m_serv.sin_port = htons(UDP_DNS_PORT);
    m_serv.sin_addr.s_addr = htonl(INADDR_ANY);

    m_startTime = std::chrono::steady_clock::now();

    m_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    if(m_socket  < 0) {
        perror("DNSServer::DNSServer: Socket failed");
        throw std::runtime_error("Socket failed");
    }

    if(bind(m_socket, reinterpret_cast<sockaddr*>(&m_serv),
sizeof(m_serv)) < 0) {
        perror("DNSServer::DNSServer: Bind failed");
        close(m_socket);
        m_socket = -1;
        throw std::runtime_error("Bind failed");
    }

    m_sender = std::thread([this]() {
        senderLoop();
    });
}

bool DNSServer::isRunning() const {
    return !m_stop;
}

double DNSServer::getStartTime() const {
    return m_stop ? double{} : std::chrono::duration<double,
std::milli>(std::chrono::steady_clock::now() -
m_startTime).count();
}

uint64_t DNSServer::getTotalRequests() const {
    return m_threadPool.getTotalRequests();
}

uint64_t DNSServer::getTotalErrors() const {
    return m_threadPool.getTotalErrors();
}

double DNSServer::getLatency() {
    return m_threadPool.getLatency();
}

void DNSServer::resetLatency() {
    m_threadPool.resetLatency();
}

bool DNSServer::run() {
    if(m_socket < 0) {
        perror("DNSServer::run: Socket closed");
        return false;
    }

    std::cout << "DNSServer::run: starting..." << std::endl;
```

```
    while(!m_stop) {
        OutPacket::Packet packet;
        packet.data.resize(BUFFER_SIZE);

        socklen_t len = sizeof(packet.client);

        // get receive
        ssize_t sizePacket = recvfrom(m_socket,
packet.data.data(), packet.data.size(),
            0, reinterpret_cast<sockaddr*>(&packet.client), &len);

        if(sizePacket < 0) {
            perror("DNSServer::run: Recvfrom failed");
            continue;
        }

        packet.size = sizePacket;
        packet.data.resize(sizePacket);

        // create dnsTask
        std::shared_ptr<Task> newTask =
std::make_shared<DNSTask>(m_dispatcher, std::move(packet));

        m_threadPool.enqueue(newTask);

    }
    return true;
}

void DNSServer::senderLoop() {
    while(true) {
        std::optional<OutPacket::Packet> packet =
m_threadPool.popResult();
        if(!packet) {
            break;
        }

        auto& receive = packet.value();

        // send answer
        ssize_t senSize = sendto(m_socket, receive.data.data(),
receive.size, 0,
                    reinterpret_cast<sockaddr*>(&receive.client),
sizeof(receive.client));

        if(senSize <= 0) {
            perror("DNSServer::senderLoop: Failed sendto");
        }
    }
}
```

```
DNSServer::~DNSServer() {
    m_stop = true;
    m_threadPool.stopQueue();

    if(m_socket >= 0) {
        shutdown(m_socket, SHUT_RDWR);
        close(m_socket);
    }

    if(m_sender.joinable()) {
        m_sender.join();
    }
}
```

Файл client.h

```
#ifndef CLIENTS_H
#define CLIENTS_H

#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include <atomic>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <algorithm>

class Clients final {
    static constexpr inline int CLIENTS_PORT = 6060;
public:
    Clients();
    ~Clients();

    void erase(std::vector<int>  sock);
    std::vector<int> getClients();

private:
    void startListen();

private:
    int m_socket = -1;
    std::vector<int> m_clients;

    std::mutex m_mtx;
    std::thread m_listener;
    std::atomic<bool> m_stop{};
};

#endif // CLIENTS_H
```

Файл client.cpp

```
#include "../../headers/Net/clients.h"

Clients::Clients() {
    sockaddr_in server{};
    server.sin_family = AF_INET;
    server.sin_port = htons(CLIENTS_PORT);
    server.sin_addr.s_addr = INADDR_ANY;

    m_socket = socket(AF_INET, SOCK_STREAM, 0);
    if(m_socket  < 0) {
        perror("Clients::Clients: Socket failed");
        throw std::runtime_error("Socket failed");
    }

    int optVal = 1;
    if(setsockopt(m_socket, SOL_SOCKET, SO_REUSEADDR, &optVal,
sizeof(optVal)) < 0) {
        perror("Clients::Clients: Setsockopt failed");
        close(m_socket);
        throw std::runtime_error("Setsockopt failed");
    }

    if(bind(m_socket, reinterpret_cast<sockaddr*>(&server),
sizeof(server)) < 0) {
        perror("Clients::Clients: Bind failed");
        throw std::runtime_error("Bind failed");
    }

    m_listener = std::thread([this]() {
        startListen();
    });
}

Clients::~Clients() {
    m_stop = true;

    if(m_socket >= 0) {
        shutdown(m_socket, SHUT_RDWR);
        close(m_socket);
    }

    {
        std::lock_guard lock{ m_mtx };
        for(const auto& client : m_clients) {
            shutdown(client, SHUT_RDWR);
            close(client);
        }
        m_clients.clear();
    }

    if(m_listener.joinable()) {
```

```
        m_listener.join();
    }
}

void Clients::erase(std::vector<int> socks) {
    std::lock_guard lock{ m_mtx };

    for(const auto& sock : socks) {
        auto it = std::find(m_clients.cbegin(), m_clients.cend(),
sock);

        if(it != m_clients.cend()) {
            close(sock);
            m_clients.erase(it);
        }
    }
}

std::vector<int> Clients::getClients() {
    std::lock_guard lock{ m_mtx };
    return m_clients;
}

void Clients::startListen() {
    if(m_socket < 0) {
        perror("Clients::startListen: Socket closed");
        return;
    }

    if(listen(m_socket, SOMAXCONN) < 0) {
        perror("Clients::startListen: Listen failed");
        return;
    }

    while(!m_stop) {
        sockaddr_in newClient;
        socklen_t len = sizeof(newClient);

        // get receive
        int clientSocket = accept(m_socket,
reinterpret_cast<sockaddr*>(&newClient), &len);

        if(clientSocket < 0) {
            if(m_stop) {
                break;
            }
            perror("Clients::startListen: Accept failed");
            continue;
        }
        {
            std::lock_guard lock{ m_mtx };
```

```
            m_clients.push_back(clientSocket);
        }
    }
}
```

Файл dnsCache.h

```
#ifndef DNSCACHE_H
#define DNSCACHE_H

#include <iostream>
#include <unordered_map>
#include <optional>
#include <mutex>
#include <atomic>

#include "../Common/utils.h"

using namespace Utils;

class DNSCache final {
public:
    struct RHash {
        std::size_t operator()(const Cache::Key& key) const {
            std::size_t h1 = std::hash<std::string>()(key.domain);
            std::size_t h2 =
std::hash<uint32_t>()(static_cast<uint32_t>(key.type));
            return h1 ^ (h2 << 1);
        }
    };

public:
    void put(const Cache::Record& rec);
    std::optional<Cache::Record> get(const Cache::Key& key);

    uint64_t getCacheEntries();

private:
    bool isExpired(const Cache::Record& rec) const;

private:
    std::unordered_map<Cache::Key, Cache::Record, RHash>
m_records;
    std::mutex m_mtx;
};

#endif // DNSCACHE_H
```

Файл dnsCache.cpp

```
#include "../../headers/DNS/dnsCache.h"

void DNSCache::put(const Cache::Record& rec) {
```

```
    Cache::Record tmp = rec;
    tmp.created = std::chrono::steady_clock::now();

    {
        std::lock_guard lock(m_mtx);
        m_records[rec.key] = tmp;
    }
}

std::optional<Cache::Record> DNSCache::get(const Cache::Key& key)
{
    std::lock_guard lock(m_mtx);

    auto it = m_records.find(key);
    if(it == m_records.end()) {
        return std::nullopt;
    }

    if(isExpired(it->second)) {
        std::cerr << "DNSCache::get: Entry TTL finished. Delete
entry" << std::endl;
        m_records.erase(it);
        return std::nullopt;
    }

    return it->second;
}

bool DNSCache::isExpired(const Cache::Record& rec) const {
    return std::chrono::steady_clock::now() - rec.created >
           std::chrono::seconds(rec.ttl);   // if now > last valid
time
};

uint64_t DNSCache::getCacheEntries() {
    std::lock_guard lock(m_mtx);
    return m_records.size();
}
```

Файл dnsParser.h

```
#ifndef DNSPARSER_H
#define DNSPARSER_H

#include <iostream>
#include <memory>
#include <vector>
#include <ldns/ldns.h>

#include "../Common/utils.h"

using namespace Utils;
```

```
class DNSParser final {
public:
    //deleter form ldns_packet
    struct PktDeleter {
        void operator()(ldns_pkt* pkt) const {
            if(pkt) {
                ldns_pkt_free(pkt);
            }
        }
    };

    using DNSPtr = std::unique_ptr<ldns_pkt, PktDeleter>;
    using DNSPkt = std::pair<Parse::Status, DNSPtr>;
    using DNSWire = std::pair<Parse::Status,
std::vector<uint8_t>>;

    static DNSPkt deserialize(const std::vector<uint8_t>& packet,
const std::size_t size);
    static DNSWire serialize(const DNSPtr& packet);
};

#endif // DNSPARSER_H
```

Файл dnsParser.cpp

```
#include "../../headers/DNS/dnsParser.h"

// converting the byte stream into a basic dns structure, see
RFC1035
/*static*/
DNSParser::DNSPkt DNSParser::deserialize(const
std::vector<uint8_t>& packet, const std::size_t size) {
    ldns_pkt *pkt = nullptr;

    if(ldns_wire2pkt(&pkt, packet.data(), size) != LDNS_STATUS_OK)
{
        std::cerr << "DNSParser::deserialize: Failed deserialize
packet" << std::endl;
        return { Parse::Status::Err, DNSPtr(nullptr) };
    }

    std::cout << "DNSParser::deserialize: Success deserialize" <<
std::endl;
    return { Parse::Status::Ok, DNSPtr(pkt) };
}

/*static*/
DNSParser::DNSWire DNSParser::serialize(const DNSPtr& packet) {
    uint8_t* wire = nullptr;
    std::size_t wireSize = 0;
```

```
    ldns_status ok = ldns_pkt2wire(&wire, packet.get(),
&wireSize);
    if(ok != LDNS_STATUS_OK || !wire || wireSize == 0) {
        std::cerr << "DNSParser::serialize: Failed serialize
packet" << std::endl;
        return { Parse::Status::Err, {} };
    }

    std::cout << "DNSParser::serialize: Success serialize" <<
std::endl;
    std::vector<uint8_t> vec(wire, wire + wireSize);
    free(wire);

    return { Parse::Status::Ok, vec };
}
```

Файл dnsResolver.h

```
#ifndef DNSRESOLVER_H
#define DNSRESOLVER_H

#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <mutex>
#include <numeric>
#include <atomic>

#include "dnsParser.h"
#include "../Common/utils.h"

using namespace Utils;

class DNSResolver final {
    static constexpr inline std::size_t BUFFER_SIZE = 512;
    static constexpr inline int UDP_DNS_PORT = 53;
    static constexpr inline int RESOLVE_SIZE = 1000;

public:
    struct Packet {
        Parse::Status status = Parse::Status::Err;
        DNSParser::DNSPtr packet = DNSParser::DNSPtr(nullptr);
        std::string error = "";
    };

    DNSResolver(const std::string& addr);

    DNSResolver::Packet resolve(const DNSParser::DNSPtr& packet);
    double getResolve();
    void resetLatency();

private:
```

```
    static int makeSocket();
    void addResolve(std::chrono::nanoseconds duration);

private:
    sockaddr_in m_upstream;
    std::mutex m_mtxRes;

    std::atomic<uint32_t> m_index {};
    std::vector<std::chrono::nanoseconds> m_resolve;
};

#endif // DNSRESOLVER_H
```

Файл dnsResolver.cpp

```
#include "../../headers/DNS/dnsResolver.h"

DNSResolver::DNSResolver(const std::string& addr/*77.88.8.8*/) {
    m_resolve.reserve(RESOLVE_SIZE);
    m_upstream.sin_family = AF_INET;
    m_upstream.sin_port = htons(UDP_DNS_PORT);
    if(inet_pton(AF_INET, addr.c_str(), &m_upstream.sin_addr) !=
1) {
        throw std::invalid_argument("Failed resolving address");
    }
}

void DNSResolver::addResolve(std::chrono::nanoseconds duration) {
    std::lock_guard lock{ m_mtxRes };
    if(m_index < RESOLVE_SIZE)
        m_resolve.push_back(duration);
    else
        m_resolve[m_index % RESOLVE_SIZE] = duration;   //
push_front
    ++m_index;
}

double DNSResolver::getResolve() {
    std::lock_guard lock{ m_mtxRes };
    if(m_resolve.empty()) {
        return {};
    }

    auto acc = std::accumulate(m_resolve.cbegin(),
m_resolve.cend(), std::chrono::nanoseconds{});  // average

    double res = std::chrono::duration<double, std::milli>(acc /
m_resolve.size()).count();   // ms

    return res;
}
```

```
void DNSResolver::resetLatency() {
    std::lock_guard lock{ m_mtxRes };
    m_resolve.clear();
    m_index = 0;
}

DNSResolver::Packet DNSResolver::resolve(const DNSParser::DNSPtr&
packet) {
    auto start = std::chrono::steady_clock::now();
    Packet res;

    thread_local int sock = makeSocket();
    if(sock < 0) {
        perror("DNSResolver::resolve: Socket closed");
        res.error = "FORMERR";
        return res;
    }

    // remember for get exactly my packet
    const uint16_t id = ldns_pkt_id(packet.get());

    // sserialize ldns packet to wire bytes
    const auto& [ok, question] = DNSParser::serialize(packet);
    if(ok != Parse::Status::Ok) {
        res.error = "FORMERR";
        return res;
    }

    // send to remote host
    ssize_t senSize = sendto(sock, question.data(),
question.size(), 0,
            reinterpret_cast<sockaddr*>(&m_upstream),
sizeof(m_upstream));

    if(senSize <= 0) {
        perror("DNSResolver::resolve: Failed sendto");
        res.error = "SERVFAIL";
        return res;
    }

    sockaddr_in from{};
    socklen_t len = sizeof(from);

    // buffer for aPacket
    std::vector<uint8_t> answer(BUFFER_SIZE);

    const auto timeUpstream = std::chrono::steady_clock::now() +
std::chrono::seconds(1);
    while(std::chrono::steady_clock::now() < timeUpstream) {
        answer.assign(BUFFER_SIZE, 0);
```

```
        // get receive
        ssize_t recSize = recvfrom(sock, answer.data(),
answer.size(),
                0, reinterpret_cast<sockaddr*>(&from), &len);

        if(recSize < 0) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                continue;
            }

            perror("DNSResolver::resolve: Failed sendto");
            res.error = "SERVFAIL";
            return res;
        }

        // header size
        if(recSize < 12) {
            continue;
        }

        // validate host
        if(from.sin_addr.s_addr != m_upstream.sin_addr.s_addr ||
            from.sin_port != m_upstream.sin_port) {
            continue;
        }

        // first 2 bytes of header - ID
        const uint16_t receivedID =
(static_cast<uint16_t>(answer[0]) << 8) |

static_cast<uint16_t>(answer[1]);

        if(receivedID != id) {
            continue;
        }

        auto [ok, pkt] = DNSParser::deserialize(answer, recSize);
        if(ok != Parse::Status::Ok) {
            continue;
        }

        addResolve(std::chrono::steady_clock::now() - start);
        return {Parse::Status::Ok, std::move(pkt), ""};
    }
    res.error = "TIMEOUT";
    return res;
}

/*static*/
int DNSResolver::makeSocket() {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    if(sock < 0) {
        perror("DNSResolver::DNSResolver: Socket failed");
        throw std::runtime_error("Socket failed");
    }

    // for block receive if server is unreacheble
    timeval tv{};
    tv.tv_sec = 0;
    tv.tv_usec = 100000;

    if(setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv))
< 0) {
        perror("DNSResolver::DNSResolver: Setsockopt failed");
        close(sock);
        throw std::runtime_error("Setsockopt failed");
    }
    return sock;
}
```

Файл dnsDispatcher.h

```
#ifndef DNSDISPATCHER_H
#define DNSDISPATCHER_H

#include <iostream>
#include <ldns/ldns.h>
#include <map>
#include <algorithm>
#include <deque>

#include "dnsParser.h"
#include "dnsResolver.h"
#include "dnsCache.h"
#include "../Common/utils.h"

using namespace Utils;

class DNSDispatcher final {
    static constexpr inline uint32_t MAX_ERRORS = 100;
    static constexpr inline uint32_t TOP_DOMAINS_SIZE = 10;
    static constexpr inline uint32_t QUERY_TYPES_SIZE = 6;
    static constexpr inline uint32_t ERRORS_SIZE = 10;
public:
    DNSDispatcher(DNSCache& cache, DNSResolver& resolver);

    // main method
    DNSParser::DNSPtr dispatch(const  DNSParser::DNSPtr& packet);

    // metrics
    double getHitsPercent() const;
    uint64_t getCacheEntries() const;
    std::vector<MetricRecords::TopDomainRecord> getTopDomains();
```

```
    std::vector<MetricRecords::QueryTypeRecord> getQueryTypes();
    std::vector<MetricRecords::ErrorRecord> getRecentErrors();
    void resetDomains();

private:
    // interacting with cache
    std::vector<Cache::Record> lookupCache(const
std::vector<ldns_rr*>& rr);
    bool addToCache(const std::vector<ldns_rr*>& rrs);

    // forming packet
    DNSParser::DNSPtr basePacket(const DNSParser::DNSPtr&
clientPacket) const;
    DNSParser::DNSPtr makeErrPacket(const DNSParser::DNSPtr&
basePacket,
                                    ldns_pkt_rcode  rcode) const;
    DNSParser::DNSPtr makeSucPacket(const DNSParser::DNSPtr&
basePacket,
                                    const
std::vector<Cache::Record>& rrs) const;

    // additional for forming
    static ldns_rr* buildRr(const Cache::Record& rr);
    static bool addRData(ldns_rr* newRr, const Cache::Record& rr);
    std::vector<ldns_rr*> getQuestionRRs(const DNSParser::DNSPtr&
pkt) const;
    std::vector<ldns_rr*> getAnswerRRs(const DNSParser::DNSPtr&
pkt) const;

    // metrics
    void addMetrics(const std::string& domain, DNS::Types type);
    void addError(const std::string& domain, const std::string&
error);

    // other
    static bool isAnswer(const DNSParser::DNSPtr& pkt);
    static std::string getFirstDomain(ldns_rr* rr);

private:
    std::mutex m_mtx;
    std::atomic<uint64_t> m_cacheHits {};
    std::atomic<uint64_t> m_cacheMiss {};
    std::map<std::string, uint64_t> m_domains;
    std::map<std::string, uint64_t> m_types;
    std::deque<MetricRecords::ErrorRecord> m_errors;

    DNSCache& m_cache;
    DNSResolver& m_resolver;
};

#endif // DNSDISPATCHER_H
```

Файл dnsDispatcher.cpp

```
#include "../../headers/DNS/dnsDispatcher.h"

#include <sstream>

using namespace Utils;

DNSDispatcher::DNSDispatcher(DNSCache& cache, DNSResolver&
resolver)
    :
    m_cache(cache),
    m_resolver(resolver)
{}


// ---------------------------------- MAIN METHOD ---------------
--------------------
DNSParser::DNSPtr DNSDispatcher::dispatch(const DNSParser::DNSPtr&
clientPacket) {
    // checking qr
    if(isAnswer(clientPacket)) {
        return
DNSParser::DNSPtr(ldns_pkt_clone(clientPacket.get()));
    }
    std::cout << "DNSDispatcher::dispatch: Packet - is question"
<< std::endl;

    // forming skeleton
    DNSParser::DNSPtr aPacket = basePacket(clientPacket);
    if(!aPacket) {
        std::cerr << "DNSDispatcher::dispatch: Failed create base
packet" << std::endl;
        addError("unknown", "FORMERR");
        return makeErrPacket(clientPacket, LDNS_RCODE_FORMERR);
    }

    // get questions from client
    std::vector<ldns_rr*> resources =
getQuestionRRs(clientPacket);
    if(resources.empty()) {
        std::cerr << "DNSDispatcher::dispatch: Question empty" <<
std::endl;
        addError("unknown", "FORMERR");
        return makeErrPacket(aPacket, LDNS_RCODE_FORMERR);
    }

    // add metrics
    std::string errDomain = getFirstDomain(resources[0]);
    const ldns_rr_type type = ldns_rr_get_type(resources[0]);
    auto t =  DNS::fromDNSType(type);
```

```
    addMetrics(errDomain, t);

    // check cache
    if(t != DNS::Types::Unknown) {
        std::vector<Cache::Record> result =
lookupCache(resources);
        if(!result.empty()) {
            ++m_cacheHits;
            return makeSucPacket(aPacket, result);
        }
        ++m_cacheMiss;
    }

    // resolving + creating answer
    DNSResolver::Packet res;
    res = m_resolver.resolve(clientPacket);
    if(res.status != Parse::Status::Ok) {
        std::cerr << "DNSDispatcher::dispatch: Failed resolving"
<< std::endl;
        addError(errDomain, res.error);

        if (res.error == "NXDOMAIN")
            return makeErrPacket(aPacket, LDNS_RCODE_NXDOMAIN);
        return makeErrPacket(aPacket, LDNS_RCODE_SERVFAIL);
    }

    //  get answer from resolving packet
    resources.clear();
    resources = getAnswerRRs(res.packet);
    if(resources.empty()) {
        return makeErrPacket(aPacket, LDNS_RCODE_NOERROR);
    }

    for(const auto& ansRr : resources) {
        auto ansType = DNS::fromDNSType(ldns_rr_get_type(ansRr));
        if(ansType != t) {
            addMetrics(getFirstDomain(ansRr), ansType);
        }
    }

    // add to cache
    if(!addToCache(resources)) {
        std::cerr << "DNSDispatcher::dispatch: Failed to adding in
cache" << std::endl;
    }

    return DNSParser::DNSPtr(ldns_pkt_clone(res.packet.get()));
}
```

```
// ------------------------------------ CACHE ------------------
--------------------
std::vector<Cache::Record> DNSDispatcher::lookupCache(const
std::vector<ldns_rr*>& rrs) {
    std::vector<Cache::Record> result;
    result.reserve(rrs.size());

    for(const auto& rr : rrs) {
        // get type
        const ldns_rr_type type = ldns_rr_get_type(rr);
        auto t = DNS::fromDNSType(type);
        if(t == DNS::Types::Unknown) {
            std::cerr << "Dispatcher::lookupCache: Type don't
support or unknown" << std::endl;
            return {};
        }

        // get domain
        const ldns_rdf* domain = ldns_rr_owner(rr);
        if(!domain) {
            std::cerr << "Dispatcher::lookupCache: Invalid domain"
<< std::endl;
            return {};
        }

        char* cName = ldns_rdf2str(domain);
        if(!cName) {
            return {};
        }

        std::string name(cName);
        free(cName);

        // trim
        if(!name.empty() && name.back() == '.') {
            name.pop_back();
        }

        std::string current = name;
        int step = 0;
        bool resolved = false;

        while (++step < 9) {

            // create key & find cache record
            Cache::Key key({ current, t });

            auto rec = m_cache.get(key);
            if(rec) {
                result.push_back(rec.value());
                resolved = true;
```

```
                break;
            }

            // create key & find cache record CNAME
            Cache::Key cnameKey({ current, DNS::Types::CNAME });

            auto cnameRec = m_cache.get(cnameKey);
            if(!cnameRec) {
                std::cout << "Dispatcher::lookupCache: Not found
in cache" << std::endl;
                return {};
            }

            result.push_back(cnameRec.value());

            if(cnameRec->rdata.empty()) {
                return {};
            }

            current = cnameRec->rdata[0];
            // trim
            if(!current.empty() && current.back() == '.') {
                current.pop_back();
            }
        }

        if(!resolved) {
            std::cerr << "Dispatcher::lookupCache: CNAME chain too
deep" << std::endl;
            return {};
        }
    }

    return result;
}
bool DNSDispatcher::addToCache(const std::vector<ldns_rr*>& rrs) {
    if(rrs.empty()) {
        std::cerr << "DNSDispatcher::addToCache: Answers empty" <<
std::endl;
        return false;
    }

    bool allOk = true;
    for(const auto& rr : rrs) {
        // get type
        const ldns_rr_type type = ldns_rr_get_type(rr);
        auto t =  DNS::fromDNSType(type);
        if(t == DNS::Types::Unknown) {
            std::cerr << "Dispatcher::addToCache: Type don't
support or unknown" << std::endl;
            continue;
```

```
        }

        // get domain
        const ldns_rdf* domain = ldns_rr_owner(rr);
        if(!domain) {
            std::cerr << "Dispatcher::addToCache: Invalid domain"
<< std::endl;
            allOk = false;
            continue;
        }

        char* cName = ldns_rdf2str(domain);
        if(!cName) {
            allOk = false;
            continue;
        }

        std::string name(cName);
        free(cName);

        // trim
        if(!name.empty() && name.back() == '.') {
            name.pop_back();
        }

        // create key & find cache record
        Cache::Key key;
        key.domain = name;
        key.type = t;

        // create record
        Cache::Record rec;

        // set key
        rec.key = key;

        // set ttl
        rec.created = std::chrono::steady_clock::now();
        rec.ttl = ldns_rr_ttl(rr);

        // set class
        rec.classType =
DNS::fromDNSClassType(ldns_rr_get_class(rr));

        // adding rdata
        std::vector<std::string> rdata;

        const std::size_t count = ldns_rr_rd_count(rr);
        for(std::size_t i = 0; i < count; ++i) {
            const ldns_rdf* rdf = ldns_rr_rdf(rr, i);
            if(!rdf) {
```

```
                std::cerr << "DNSDispatcher::addToCache: Invalid
rdata" << std::endl;
                allOk = false;
                continue;
            }

            char* cName2 = ldns_rdf2str(rdf);
            if(!cName2) {
                allOk = false;
                continue;
            }
            rdata.emplace_back(cName2);
            free(cName2);
        }

        rec.rdata = rdata;

        m_cache.put(rec);
    }
    return allOk;
}


// ---------------------------------- FORMING PACKET --------------
--------------------
DNSParser::DNSPtr DNSDispatcher::basePacket(const
DNSParser::DNSPtr& pkt) const {
    if(!pkt) {
        std::cerr << "DNSDispatcher::basePacket: Invalid packet"
<< std::endl;
        return DNSParser::DNSPtr(nullptr);
    }

    // ---------------------- PACKET ------------------------
    // base receive packet
    ldns_pkt* base = ldns_pkt_new();
    if(!base) {
        std::cerr << "DNSDispatcher::basePacket: Failed to
allocate packet" << std::endl;
        return DNSParser::DNSPtr(nullptr);
    }

    // ---------------------- HEADER ------------------------
    // set flags
    ldns_pkt_set_id(base, ldns_pkt_id(pkt.get()));
    ldns_pkt_set_qr(base, true);    // 1 - receive
    ldns_pkt_set_opcode(base, ldns_pkt_get_opcode(pkt.get()));
    ldns_pkt_set_aa(base, false);   // were are not authority
    ldns_pkt_set_tc(base, false);   // packet is divided
    ldns_pkt_set_rd(base, ldns_pkt_rd(pkt.get()));
    ldns_pkt_set_ra(base, true);    // we supporting recursing
```

```
    ldns_pkt_set_ad(base, false);
    ldns_pkt_set_cd(base, ldns_pkt_cd(pkt.get()));
    ldns_pkt_set_rcode(base, LDNS_RCODE_NOERROR);

    // --------------------- QUESTION ----------------------
    // copy question
    ldns_rr_list* questions = ldns_pkt_question(pkt.get());
    if(!questions) {
        std::cerr << "DNSDispatcher::basePacket:Questions empty"
<< std::endl;
        ldns_pkt_free(base);
        return DNSParser::DNSPtr(nullptr);
    }

    ldns_rr_list* copy = ldns_rr_list_clone(questions);
    if(!copy) {
        std::cerr << "DNSDispatcher::basePacket: Failed to
allocate question" << std::endl;
        ldns_pkt_free(base);
        return DNSParser::DNSPtr(nullptr);
    }
    ldns_pkt_set_question(base, copy);
    ldns_pkt_set_qdcount(base, ldns_rr_list_rr_count(copy));

    // --------------------- ANSWER ----------------------
    // FILL ANSWER LATER!
    // -------------------- AUTHORITY --------------------
    // FILL AUTHORITY LATER!
    // ------------------- ADDITIONAL --------------------
    // FILL ADDITIONAL LATER!

    // ---------------------- RETURN ----------------------
    return DNSParser::DNSPtr(base);
}
DNSParser::DNSPtr DNSDispatcher::makeSucPacket(const
DNSParser::DNSPtr& basePacket,
                                               const
std::vector<Cache::Record>& rrs) const {
    // ----------------------- PACKET -----------------------
    if(!basePacket) {
        std::cerr << "DNSDispatcher::makeSucPacket: Invalid
packet" << std::endl;
        return DNSParser::DNSPtr(nullptr);
    }

    auto base = ldns_pkt_clone(basePacket.get());
    if(!base) {
        std::cerr << "DNSDispatcher::makeSucPacket: Failed to
clone packet" << std::endl;
        return makeErrPacket(basePacket, LDNS_RCODE_FORMERR);
    }
```

```
    // --------------------- HEADER ------------------------
    // all flags set in basePacket()

    // --------------------- ANSWER ------------------------
    ldns_rr_list* answer = ldns_rr_list_new();
    if(!answer) {
        std::cerr << "DNSDispatcher::makeSucPacket: Failed to
allocate answer" << std::endl;
        ldns_pkt_free(base);
        return makeErrPacket(basePacket, LDNS_RCODE_FORMERR);
    }

    for(const auto& rr : rrs) {
        ldns_rr* built = buildRr(rr);
        if(built) {
            ldns_rr_list_push_rr(answer, built);
        }
    }

    ldns_pkt_set_answer(base, answer);
    ldns_pkt_set_ancount(base, ldns_rr_list_rr_count(answer));

    // --------------------- AUTHORITY ---------------------
    ldns_pkt_set_authority(base, ldns_rr_list_new());

    // --------------------- ADDITIONAL --------------------
    ldns_pkt_set_additional(base, ldns_rr_list_new());

    return DNSParser::DNSPtr(base);
}
DNSParser::DNSPtr DNSDispatcher::makeErrPacket(const
DNSParser::DNSPtr& basePacket,
                                ldns_pkt_rcode  rcode) const {
    // ---------------------- PACKET -----------------------
    if(!basePacket) {
        std::cerr << "DNSDispatcher::makeErrPacket: Invalid
packet" << std::endl;
        return DNSParser::DNSPtr(nullptr);
    }

    auto base = ldns_pkt_clone(basePacket.get());
    if(!base) {
        base = ldns_pkt_new();
        if(!base) {
            std::cerr << "DNSDispatcher::makeErrPacket: Critical
alloc failure" << std::endl;
            return DNSParser::DNSPtr(nullptr);
        }
        ldns_pkt_set_qr(base, true);
    }
```

```
    // ---------------------- HEADER -----------------------
    // set flags
    ldns_pkt_set_rcode(base, rcode);    // set rcode
    ldns_pkt_set_qr(base, true);
    ldns_pkt_set_aa(base, false);

    // --------------------- ANSWER ------------------------
    ldns_pkt_set_answer(base, ldns_rr_list_new());

    // -------------------- AUTHORITY ----------------------
    ldns_pkt_set_authority(base, ldns_rr_list_new());

    // -------------------- ADDITIONAL ---------------------
    ldns_pkt_set_additional(base, ldns_rr_list_new());

    return DNSParser::DNSPtr(base);
}

    // -------------------------- ADDITIONAL FOR FORM ------------
---------------------
std::vector<ldns_rr*> DNSDispatcher::getQuestionRRs(const
DNSParser::DNSPtr& pkt) const {
    // get question chapter
    ldns_rr_list* questions = ldns_pkt_question(pkt.get());
    if(!questions) {
        std::cerr << "Dispatcher::getQuestionRRs: Questions empty"
<< std::endl;
        return {};
    }

    // get questions count
    const std::size_t questionsCount =
ldns_rr_list_rr_count(questions);

    std::vector<ldns_rr*> resources;
    resources.reserve(questionsCount);

    for(std::size_t i = 0; i < questionsCount; ++i) {
        // struct resource
        //-+-+-+-+-+-+-+-+-+-+
        //|      NAME       |
        //|      TYPE       |
        //|      CLASS      |
        //|       TTL       |
        //|     RDLENGTH    |
        //|      RDATA      |
        //-+-+-+-+-+-+-+-+-+-+
        resources.emplace_back(ldns_rr_list_rr(questions, i));
    }
    return resources;
```

```
}
std::vector<ldns_rr*> DNSDispatcher::getAnswerRRs(const
DNSParser::DNSPtr& pkt) const {
    // get answers chapter
    ldns_rr_list* answers = ldns_pkt_answer(pkt.get());
    if(!answers) {
        std::cerr << "Dispatcher::getAnswerRRs: Answers empty" <<
std::endl;
        return {};
    }

    // get answers count
    const std::size_t answersCount =
ldns_rr_list_rr_count(answers);

    std::vector<ldns_rr*> resources;
    resources.reserve(answersCount);

    for(std::size_t i = 0; i < answersCount; ++i) {
        // struct resource
        //-+-+-+-+-+-+-+-+-+-+
        //|      NAME       |
        //|      TYPE       |
        //|      CLASS      |
        //|       TTL       |
        //|     RDLENGTH    |
        //|      RDATA      |
        //-+-+-+-+-+-+-+-+-+-+
        resources.emplace_back(ldns_rr_list_rr(answers, i));
    }
    return resources;
}
/*static*/
ldns_rr* DNSDispatcher::buildRr(const Cache::Record& rr) {
    ldns_rr* newRr = ldns_rr_new();
    if(!newRr) {
        std::cerr << "DNSDispatcher::buildRr: Failed to allocate
resource" << std::endl;
        return {};
    }

    // get owner
    std::string domain = rr.key.domain;
    if(domain.empty() || domain.back() != '.') {
        domain += '.';
    }

    ldns_rdf* owner = ldns_dname_new_frm_str(domain.c_str());
    if(!owner) {
        ldns_rr_free(newRr);
```

```
        std::cerr << "DNSDispatcher::buildRr: Failed to allocate
owner" << std::endl;
        return {};
    }

    // set owner
    ldns_rr_set_owner(newRr, owner);

    // set type
    ldns_rr_set_type(newRr, toDNSType(rr.key.type));

    // set class
    ldns_rr_set_class(newRr, toDNSClassType(rr.classType));

    // set ttl
    ldns_rr_set_ttl(newRr, rr.ttl);

    // set rdata
    if(!addRData(newRr, rr)) {
        ldns_rr_free(newRr);
        std::cerr << "DNSDispatcher::createAnswer: Failed add
rdata" << std::endl;
        return {};
    }

    return newRr;
}
/*static*/
bool DNSDispatcher::addRData(ldns_rr* newRr, const Cache::Record&
rr) {
    // set rdata
    switch (rr.key.type) {
        case DNS::Types::A: {
            for(const auto& ip : rr.rdata) {
                struct in_addr addr{};
                if(inet_pton(AF_INET, ip.c_str(), &addr) != 1) {
                    std::cerr << "DNSDispatcher::buildRr: Failed
adding ipv4 address" << std::endl;
                    return false;
                }

                void* data = malloc(4);
                if(!data) return false;
                memcpy(data, &addr, 4);

                ldns_rdf* rdf = ldns_rdf_new(LDNS_RDF_TYPE_A, 4,
data);
                if(!rdf) {
                    std::cerr << "DNSDispatcher::buildRr: Failed
to allocate A rdata" << std::endl;
                    return false;
```

```
                }

                ldns_rr_push_rdf(newRr, rdf);
            }
            break;
        }
        case DNS::Types::AAAA: {
            for(const auto& ip : rr.rdata) {
                struct in6_addr addr6{};
                if(inet_pton(AF_INET6, ip.c_str(), &addr6) != 1) {
                    std::cerr << "DNSDispatcher::buildRr: Failed
adding ipv6 address" << std::endl;
                    return false;
                }

                void* data = malloc(16);
                if(!data) return false;
                memcpy(data, &addr6, 16);

                ldns_rdf* rdf = ldns_rdf_new(LDNS_RDF_TYPE_AAAA,
16, data);
                if(!rdf) {
                    std::cerr << "DNSDispatcher::buildRr: Failed
to allocate AAAA data" << std::endl;
                    return false;
                }

                ldns_rr_push_rdf(newRr, rdf);
            }
            break;
        }
        case DNS::Types::CNAME:
        case DNS::Types::NS: {
            for(const auto& name : rr.rdata) {
                ldns_rdf* rdf =
ldns_dname_new_frm_str(name.c_str());
                if(!rdf) {
                    std::cerr << "DNSDispatcher::buildRr: Failed
to allocate CNAME/NS data" << std::endl;
                    return false;
                }

                ldns_rr_push_rdf(newRr, rdf);
            }
            break;
        }
        case DNS::Types::MX: {
            for(const auto& mx : rr.rdata) {
                std::istringstream iss(mx);

                int priority = 0;
```

```
                std::string host;

                iss >> priority >> host;

                ldns_rdf* pref =
ldns_native2rdf_int16(LDNS_RDF_TYPE_INT16, priority);
                ldns_rdf* exch =
ldns_dname_new_frm_str(host.c_str());
                if(!pref || !exch) {
                    if(pref) {
                        ldns_rdf_free(pref);
                    }

                    if(exch) {
                        ldns_rdf_free(exch);
                    }

                    return false;
                }

                ldns_rr_push_rdf(newRr, pref);
                ldns_rr_push_rdf(newRr, exch);
            }
            break;
        }
        case DNS::Types::TXT: {
            for(const auto& txt : rr.rdata) {
                ldns_rdf* rdf =
ldns_rdf_new_frm_str(LDNS_RDF_TYPE_STR, txt.c_str());
                if(!rdf) {
                    std::cerr << "DNSDispatcher::buildRr: Failed
to allocate TXT data" << std::endl;
                    return false;
                }

                ldns_rr_push_rdf(newRr, rdf);
            }
            break;
        }
        default: {
            return false;
        }
    }
    return true;
}


// ------------------------------------ METRICS -----------------
----------------------
double DNSDispatcher::getHitsPercent() const {
    if(m_cacheHits + m_cacheMiss == 0) {
```

```
        return {};
    }
    return  static_cast<double>(m_cacheHits) /
            static_cast<double>(m_cacheHits + m_cacheMiss) *
100.0;
}
uint64_t DNSDispatcher::getCacheEntries() const {
    return m_cache.getCacheEntries();
}
std::vector<MetricRecords::TopDomainRecord>
DNSDispatcher::getTopDomains() {
    std::lock_guard lock{ m_mtx };

    // copy for sort
    std::vector<std::pair<std::string, uint64_t>>
sorted(m_domains.begin(), m_domains.end());
    std::sort(sorted.begin(), sorted.end(),
              [](const auto& a, const auto& b) { return a.second >
b.second; });

    std::vector<MetricRecords::TopDomainRecord> result;

    std::size_t min = std::min(sorted.size(),
std::size_t(TOP_DOMAINS_SIZE));

    uint64_t total = 0;
    for (const auto& [k, v] : sorted) {
        total += v;
    }

    for (std::size_t i = 0; i < min; ++i) {
        double percent = total > 0
                         ? static_cast<double>(sorted[i].second) /
static_cast<double>(total) * 100.0
                         : 0.0;


result.emplace_back(MetricRecords::TopDomainRecord{sorted[i].first
, percent, sorted[i].second});
    }

    return result;
}
std::vector<MetricRecords::QueryTypeRecord>
DNSDispatcher::getQueryTypes() {
    std::lock_guard lock{m_mtx};

    // copy for sort
    std::vector<std::pair<std::string, uint64_t>>
sorted(m_types.begin(), m_types.end());
    std::sort(sorted.begin(), sorted.end(),
```

```
              [](const auto& a, const auto& b) { return a.second >
b.second; });

    std::vector<MetricRecords::QueryTypeRecord> result;

    std::size_t min = std::min(sorted.size(),
std::size_t(QUERY_TYPES_SIZE));

    uint64_t total = 0;
    for (const auto& [k, v] : sorted) {
        total += v;
    }

    for (std::size_t i = 0; i < min; ++i) {
        double percent = total > 0
                         ? static_cast<double>(sorted[i].second) /
static_cast<double>(total) * 100.0
                         : 0.0;


result.emplace_back(MetricRecords::QueryTypeRecord{sorted[i].first
, percent});
    }

    return result;
}
std::vector<MetricRecords::ErrorRecord>
DNSDispatcher::getRecentErrors() {
    std::lock_guard lock{ m_mtx };

    std::vector<MetricRecords::ErrorRecord>
sorted(m_errors.cbegin(), m_errors.cend());

    std::sort(sorted.begin(), sorted.end(),
                [](const auto& a, const auto& b) { return a.time >
b.time; });

    return {sorted.cbegin(), sorted.cend()};
}
void DNSDispatcher::addMetrics(const std::string& domain,
DNS::Types type) {
    std::lock_guard lock{ m_mtx };

    ++m_domains[domain];
    ++m_types[DNS::typeToStr(type)];
}
void DNSDispatcher::addError(const std::string& domain, const
std::string& error) {
    std::lock_guard lock{ m_mtx };
    if(m_errors.size() >= ERRORS_SIZE) {
        m_errors.pop_front();
```

```
    }
    double time = std::chrono::duration<double,
std::milli>(std::chrono::system_clock::now().time_since_epoch()).c
ount();
    m_errors.emplace_back(MetricRecords::ErrorRecord{time, domain,
error});
}
void DNSDispatcher::resetDomains() {
    std::lock_guard lock{ m_mtx };
    m_domains.clear();
}

// ------------------------------------ OTHER -----------------
----------------------
/*static*/
bool DNSDispatcher::isAnswer(const DNSParser::DNSPtr& pkt) {
    return ldns_pkt_qr(pkt.get());
}
/*static*/
std::string DNSDispatcher::getFirstDomain(ldns_rr* rr) {
    // get domain
    const ldns_rdf* domain = ldns_rr_owner(rr);
    if(!domain) {
        std::cerr << "Dispatcher::lookupCache: Invalid domain" <<
std::endl;
        return "unknown";
    }

    char* cName = ldns_rdf2str(domain);
    if(!cName) {
        return "unknown";
    }

    std::string name(cName);
    free(cName);

    // trim
    if(!name.empty() && name.back() == '.') {
        name.pop_back();
    }

    return name;
}
```

Файл threadPool.h

```
#ifndef THREADPOOL_H
#define THREADPOOL_H

#include <iostream>
#include <memory>
#include <queue>
```

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <algorithm>

#include "../../headers/Common/utils.h"
#include "outQueue.h"
#include "Tasks/task.h"


using namespace Utils;

class ThreadPool final {
    static constexpr inline uint32_t WORKERS_SIZE = 6;
    static constexpr inline uint32_t LATENCY_SIZE = 1000;
public:
    ThreadPool(std::size_t workers = WORKERS_SIZE);
    ~ThreadPool();

    void enqueue(std::shared_ptr<Task> newTask);
    std::optional<OutPacket::Packet> popResult();
    double getLatency();
    void resetLatency();
    uint64_t getTotalRequests() const;
    uint64_t getTotalErrors() const;
    void stopQueue();

private:
    void workerLoop();
    void addLatency(std::chrono::nanoseconds duration);

private:
    std::vector<std::thread> m_workers;
    std::queue<std::shared_ptr<Task>> m_tasks;

    std::mutex m_mtx;
    std::mutex m_mtxLat;
    std::condition_variable m_cv;

    OutQueue m_out;
    std::atomic<bool> m_stop {};

    std::vector<std::chrono::nanoseconds> m_latency;
    std::atomic<uint32_t> m_index{};
    std::atomic<uint64_t> m_totalRequests{};
    std::atomic<uint64_t> m_totalErrors{};
};

#endif // THREADPOOL_H
```

Файл threadPool.cpp

```
#include "../../headers/Multithreading/threadPool.h"

ThreadPool::ThreadPool(std::size_t workers) {
    m_workers.reserve(workers);
    m_latency.reserve(LATENCY_SIZE);

    for(std::size_t i = 0; i < workers; ++i) {
        m_workers.emplace_back([this]() {
            workerLoop();
        });
    }
}

void ThreadPool::addLatency(std::chrono::nanoseconds duration) {
    std::lock_guard lock{ m_mtxLat };
    if(m_index < LATENCY_SIZE)
        m_latency.push_back(duration);
    else
        m_latency[m_index % LATENCY_SIZE] = duration;   //
push_front
    ++m_index;
}

double ThreadPool::getLatency() {
    std::vector<std::chrono::nanoseconds> copy;
    {
        std::lock_guard lock{m_mtxLat};
        copy = m_latency;
    }
    if(copy.empty()) {
        return {};
    }

    std::sort(copy.begin(), copy.end());

    std::size_t count = copy.size() * 0.95;
    if(count >= copy.size()) {
        --count;
    }

    return std::chrono::duration<double,
std::milli>(copy[count]).count();  // 95 percentile
}

void ThreadPool::resetLatency() {
    std::lock_guard lock{ m_mtxLat };
    m_latency.clear();
    m_index = 0;
}

uint64_t ThreadPool::getTotalRequests() const {
```

```
    return m_totalRequests;
}

uint64_t ThreadPool::getTotalErrors() const {
    return m_totalErrors;
}

void ThreadPool::workerLoop() {
    while(true) {
        std::shared_ptr<Task> current;

        {
            std::unique_lock lock{ m_mtx };

            m_cv.wait(lock, [this](){
                return m_stop || !m_tasks.empty();
            });

            if(m_stop && m_tasks.empty()) {
                return;
            }

            current = std::move(m_tasks.front());
            m_tasks.pop();
        }

        ++m_totalRequests;
        bool isOk;

        auto start = std::chrono::steady_clock::now();
        current->execute(m_out, isOk);
        addLatency(std::chrono::steady_clock::now() - start);
        if(!isOk) {
            ++m_totalErrors;
        }
    }
}

void ThreadPool::enqueue(std::shared_ptr<Task> task) {
    {
        std::lock_guard lock{ m_mtx };
        if(m_stop) {
            return;
        }

        m_tasks.push(std::move(task));
    }

    m_cv.notify_one();
}
```

```
std::optional<OutPacket::Packet> ThreadPool::popResult() {
    return m_out.pop();
}

void ThreadPool::stopQueue() {
    m_out.stop();
}

ThreadPool::~ThreadPool() {
    {
        std::lock_guard lock{ m_mtx };
        m_stop = true;
    }

    m_cv.notify_all();

    for(auto& w : m_workers) {
        if(w.joinable()) {
            w.join();
        }
    }
}
```

Файл task.h

```
#ifndef TASK_H
#define TASK_H

#include "../outQueue.h"

class Task {
public:
    virtual void execute(OutQueue& out, bool& isOk) = 0;
    virtual ~Task() = default;
};

#endif // TASK_H
```

Остальной код можно найти на https://github.com/megawalnut/proxy_dns.

Приложение В

(обязательное)

Проксирующий DNS сервер

Презентация

Выпускная квалификационная работа бакалавра

ПРОКСИРУЮЩИЙ DNS СЕРВЕР

Автор работы: студент гр. 22ВВП1 Изосин М.А.

Научный руководитель: доцент, к.т.н. Егоров В.Ю.

Почему тема актуальна?

- повышение производительности DNS
- снижение нагрузки на внешние DNS-серверы
- мониторинг состояния DNS-сервера в реальном времени
- применение в корпоративных и локальных сетях

2

Цели и задачи проекта

- разработка многопоточного DNS proxy-сервера
- обработка DNS-запросов в соответствии с RFC 1035
- пересылка запросов к upstream
- добавление TTL-кэширования 6 основных типов DNS-записей
- создание системы мониторинга статистики в реальном времени

3

Сравнение аналогов

Имя / Критерий	systemd-resolved	dnsmasq	Acrylic	Windows DNS Client	DNS-proxy server
DNS forwarding	есть	есть	есть	есть	есть
DNS caching	есть	есть	есть	есть	есть
Мониторинг запросов	ограничен	ограничен	ограничен	отсутствует	realtime
Простота настройки	низкая	высокая	высокая	Очень высокая	средняя
Платформа	Linux	Linux	Windows	Windows	Linux
Прозрачность архитектуры	низкая	средняя	средняя	низкая	высокая

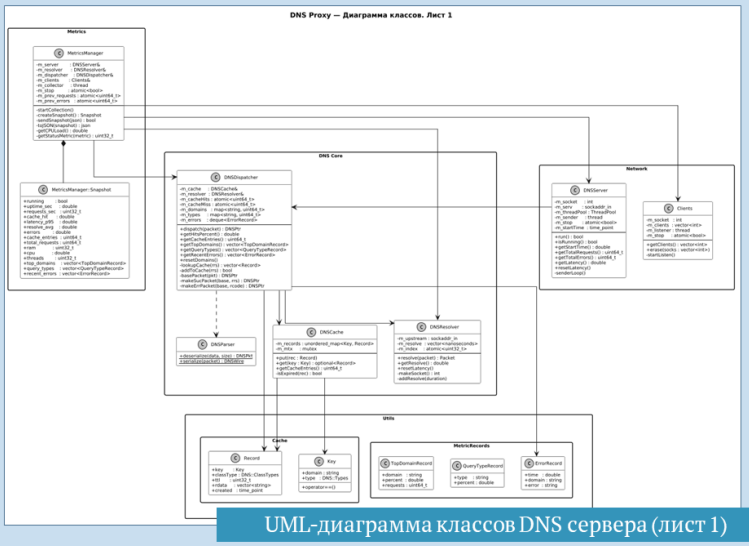
4

Архитектура разработанной системы

UML-диаграмма последовательности DNS запросов

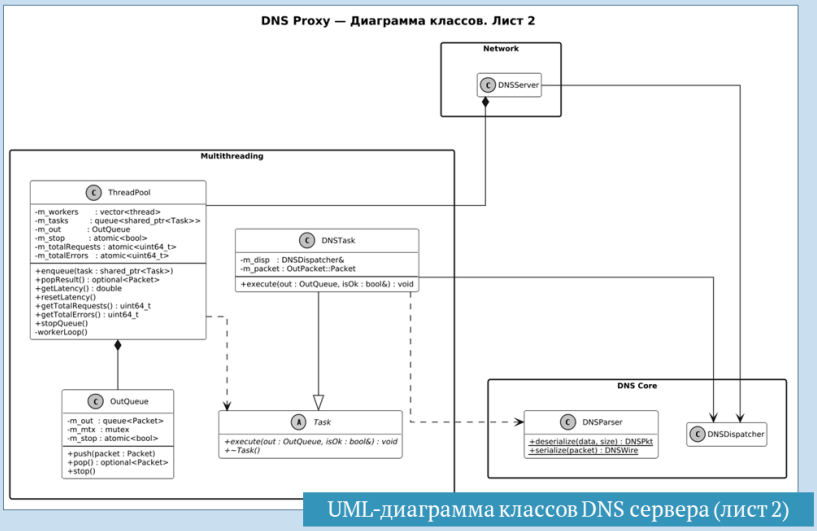
UML-диаграмма последовательности сбора метрик

5



UML-диаграмма классов DNS сервера (лист 1)

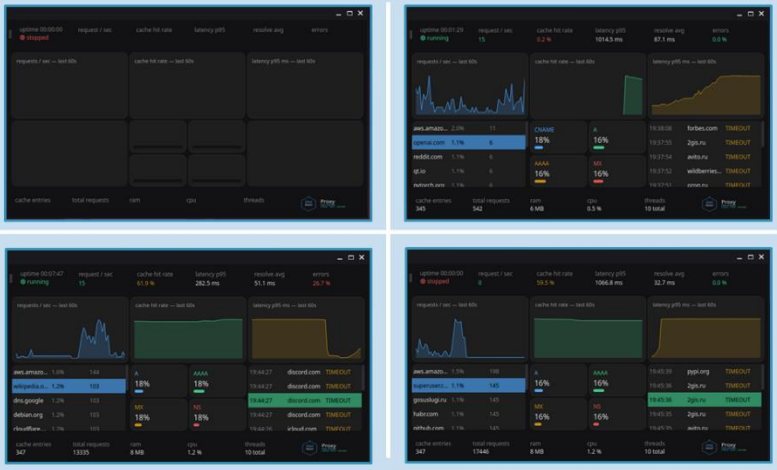
6



UML-диаграмма классов DNS сервера (лист 2)

7

Демонстрация работы кэширования DNS-прокси



8

Сравнение работы DNS-прокси с systemd-resolved при пустом и заполненном кэше

Имя / Тип записи	DNS-proxy (пустой кэш), мс	DNS-proxy (полный кэш), мс	system-resolved, мс
A	32	5	22
AAAA	48	10	22
NS	54	0	41
MX	59	1	44
TXT	64	16	17
CNAME	49	24	46

9

Возможности дальнейшего развития

- поддержка TCP fallback при фрагментации ответов (флаг TC)
- реализация DNS over TLS (DoT) и DNS over HTTPS (DoH)
- добавление кэширования других DNS-записей
- реализация полноценного рекурсивного DNS-резолвера
- внедрение персистентного кэша
- добавление логирования в файл

10

Спасибо за внимание!
Готов ответить на ваши вопросы

11