

ОГЛАВЛЕНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ	4
Введение	6
Глава 1. Анализ исходных данных и проектирование	8
1.1 Постановка и анализ задачи.....	8
1.1.1 Назначение и предметная область	8
1.1.2 Анализ предполагаемых достоинств и недостатков тестирующей системы.....	9
1.1.3 Выбор средств разработки.....	10
1.2 Выбор исходного вида системы, ее структуры	11
1.3 Выбор архитектур серверной части для анализа.....	14
Глава 2. Разработка реализаций сервера и клиента	19
2.1 Установка и настройка необходимых компонентов	19
2.2 Конструирование внешнего вида клиентского приложения.....	20
2.3 Реализация многопоточного и многопоточного асинхронного серверов.....	25
2.3.1 Многопоточный сервер.....	26
2.3.2 Многопоточно-асинхронный сервер	46
Глава 3. Тестирование программного продукта.....	54
3.1 Тестирование функционирования системы	54
3.2 Анализ результатов.....	60
3.2.1 Тестирование многопоточного сервера.....	62
3.2.2 Тестирование многопоточно-асинхронного сервера	70
3.2.3 Общие итоги тестирования, сравнение результатов.....	78
Заключение	81
Список использованных источников	83
Приложение А. Файл main.cpp генератора нагрузки	84
Приложение Б. Файл server.cpp многопоточного сервера.....	100
Приложение В. Файл main.cpp многопоточно-асинхронного сервера	104

Приложение Г. Файл server.cpp многопоточно-асинхронного сервера	
107	
Приложение Д. Файл mainwindow.cpp клиентского приложения	132
Приложение Е. Файлы потока чтения клиентского приложения	139

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

В тексте выпускной квалификационной работы введены специальные термины на русском и английском языках:

Термин, сокращение	Полная форма
ПО	Программное обеспечение
Лог	Записи событий или действий, сделанные в программе или системе. Логи содержат информацию о произошедших событиях, таких как ошибки, предупреждения, запросы и другие активности в системе.
Логгирование	Запись логов
Поток	Поток в программировании — это выполнение одного или более процессов в одной программе параллельно или конкурентно.
Линкер	Компонент компиляции программного кода, который выполняет связывание объектных файлов, которые были созданы компилятором, в единую исполняемую программу или библиотеку.
Проблема гонок	Ситуация, когда несколько процессов или потоков имеют доступ к изменению участка памяти одновременно, что приводит к непредсказуемому поведению программы при сложно прогнозируемых ситуациях.
Труба данных	Механизм межпроцессного взаимодействия в операционных системах, который позволяет передавать данные между двумя процессами.
Докер	Программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений.
Образ	Статический файл, содержащий все необходимые компоненты для запуска приложения – его код, зависимости, библиотеки и операционную систему.
Контейнер	Легкий автономный исполняемый пакет программного обеспечения, который включает в себя все необходимые компоненты, необходимые для запуска приложения в изолированной среде.
Контейнеризация	Процесс создания контейнера приложения.

Термин, сокращение	Полная форма
Тепловая карта (Heatmap)	Графическое представление данных, в котором числовые значения таблицы отображаются при помощи цвета.
Корутина	Блок кода, работающий асинхронно с возможностью остановки работы с последующим возобновлением с сохранением состояния.
Конвейерная архитектура	Простая, но надежная и эффективная архитектура, состоящая из множества этапов обработки, которые последовательно взаимодействуют с данными.
Пул потоков	Набор из работающих потоков, каждый из которых используется для выполнения асинхронных вызовов приложения по необходимости.
Заморозка потока	Остановка эффективной работы потока по причине попадания в цикл ожидания выполнения какой-либо операции.

ВВЕДЕНИЕ

Бэкенд – важнейшее направление IT, на котором базируется все, с чем взаимодействует пользователь. Оно динамично развивается, что не является чем-то необычным, т.к. к нему предъявляются высокие требования по производительности, которые, к тому же, постоянно растут.

Одними из ключевых технологий, применяемых в бекэнде являются многопоточность и асинхронность. Современные языки программирования активно их используют и внедряют как в стандартные, так и в сторонние библиотеки их реализации для упрощения разработки и увеличения производительности программного продукта. Эти две технологии позволяют создавать крайне высокоэффективные программы, рассчитанные на огромное количество пользователей. При их комбинации можно достигнуть еще большей производительности. Любая компания использует их в разработке своего ПО и постоянно стремится найти оптимальное решение своих задач с их помощью. В частности, большинство серверов и сервисных структур основаны на них.

Многопоточность – особенность платформы или приложения, состоящая в разбиении процесса в ОС на некоторое количество потоков, выполняющихся параллельно, т.е. не последовательно друг за другом, что может помочь получить лучшее использование ресурсов компьютера.

Асинхронность – использование событий, которые возникают вне зависимости от основного потока программы и выполнение действий для их обработки в неблокирующем режиме, т.е. основной поток не останавливает свою работу на время их выполнения.

Целью разработки данного программного продукта является создание платформы для сравнения производительности и анализа преимуществ и недостатков относительно друг друга для многопоточной и многопоточно-асинхронной серверных архитектур.

Разработка описанного ПО является уникальной и как позволит разработчикам выбрать необходимую архитектуру для своего продукта, опираясь на визуализированные данные тестирования, так и может стать образцом для создания аналогичных систем. Данный проект будет актуален и крайне полезен начинающим разработчикам, которые смогут при его использовании изучить эти технологии, а также для опытных программистов, которые получают возможность использовать этот подход для своих крупных проектов.

В будущем разрабатываемое ПО может получить дополнительные функции и анализируемые параметры, благодаря легким и приспособляемым инструментам анализа, которые будут доступны пользователю для настройки под свои нужды.

Для создания программного продукта требуется разработать тестируемые реализации серверов – асинхронная и многопоточно-асинхронная, которые после будут встроены в тестирующую систему при помощи программы, имитирующей клиентскую нагрузку, логгирования и системы визуализации собранных данных.

ГЛАВА 1. АНАЛИЗ ИСХОДНЫХ ДАННЫХ И ПРОЕКТИРОВАНИЕ

1.1 Постановка и анализ задачи

1.1.1 Назначение и предметная область

Разрабатываемое программное обеспечение для сравнения производительности многопоточной и многопоточной асинхронной реализаций серверов с целью нахождения оптимальной технологии для выбранной задачи.

Предметная область, для которой разрабатывается ПО, связана с процессом нагрузочного тестирования и отображает ее с точки зрения профессионального разработчика.

Целью разработки данного программного продукта является создание платформы для сравнения производительности и анализа преимуществ и недостатков относительно друг друга для многопоточной и многопоточно-асинхронной серверных архитектур.

Также стоит принять во внимание необходимость наглядности и понятности измеряемых параметров для конечного пользователя для удобства и облегчения анализа.

Перед разрабатываемым программным продуктом ставится задача реализации следующего функционала:

- запуск, эффективная работа и логгирование тестируемых серверов;
- создание выбранной нагрузки при помощи клиента;
- функционал графического отображения результатов тестирования.

Также необходимо создать реальное клиентское приложение для иллюстрации правдивости работоспособности и применимости тестируемых систем. Однако вне демонстративной части это приложение не будет

использоваться, поэтому следует на данном этапе не следует делать его сложнее необходимого.

Для гибкости и универсальности использования ПО дополнительно необходимо использование средства визуализации, способного к изменению формул измеряемых параметров, за счет которой разработчики будут иметь возможность:

- создавать свои новые отслеживаемые параметры для анализа и сравнения тестируемых программ;
- изменять и удалять отслеживаемые изначально параметры;
- настраивать варианты иллюстрации: виды графиков, цвета иллюстрируемых данных и т.д.

1.1.2 Анализ предполагаемых достоинств и недостатков тестирующей системы

Для оценки актуальности темы и при разработке ПО необходимо предусмотреть его возможные достоинства и недостатки заранее перед началом работы над ним.

Разрабатываемое ПО позволит разработчикам:

- получить платформу для проведения нагрузочного тестирования продукта;
- наглядно увидеть недостатки и преимущества конкретной технологии;
- выбрать для себя идеально подходящий набор архитектурных подходов к созданию ПО;

Тестирующий программный продукт может создавать дополнительную вычислительную нагрузку на серверное оборудование при использовании, однако оно не будет принципиально влиять на его работу в силу не крупного требуемого объема выполняемых подзадач. Новые возможности безусловно перекрывают непринципиальные недостатки, возникающие от использования тестировочного подхода.

Благодаря уникальности и актуальности функционала программного продукта, предполагается, что достоинства разрабатываемого программного продукта более весомы по своей значимости. Таким образом, проект может в будущем стать отличной платформой для образования начинающих разработчиков.

1.1.3 Выбор средств разработки

Важно выбрать средства для реализации поставленной задачи, соответствующей функциональным требованиям.

Выбор целевой операционной системы

Операционная система — комплекс взаимосвязанных программ, предназначенных для управления устройством и организации взаимодействия с пользователем, который предоставляет базовый набор функций по управлению аппаратными средствами устройства.

Для разработки программного обеспечения было решено выбрать ОС Ubuntu Linux, т.к. она проста в использовании для разработчиков, т.к. позволяет быстро и без множества проблем устанавливать необходимые библиотеки, и гибка в настройке при сохранении простоты использования в данном дистрибутиве. Данные достоинства дадут возможность получить необходимый результат при выполнении задачи.

Выбор языка программирования

Для разработки любого программного обеспечения необходимо выбрать язык программирования. Так как изучаемое ПО требует максимальной эффективности, необходимо использование высокопроизводительного языка программирования для бекэнд разработки для создания серверов. Был выбран C++ в силу его широких возможностей.

Клиентские приложения, как правило, требуют обширного использования графики, пользовательских интерфейсов, мультимедиа и интерактивности. QT, благодаря обширной библиотеке компонентов и

удобной среде разработки, обеспечивает простоту и скорость разработки на разных операционных системах для таких приложений. QT также предоставляет возможности для создания взаимодействующих компонентов, а также инструменты для сборки и развертывания приложений.

В результате, использование QT для разработки клиентского приложения упрощает и ускоряет процесс разработки, обеспечивает высокий уровень функциональности и гибкость пользовательского интерфейса, что делает приложение более привлекательным для пользователей.

Для скриптов, отправляющих данные на сервисы иллюстрации, будет использоваться Python, по причине простоты разработки и наличия удобных библиотек.

1.2 Выбор исходного вида системы, ее структуры

Тестирующая система будет состоять из многих компонентов, связанных друг с другом посредством передаваемых данных.

Основой будет взят докерный контейнер с запущенным серверным приложением. Docker был выбран в силу его простоты и удобства:

- А) сервер будет запускаться в легко настраиваемой среде контейнера
- Б) проблемы несовместимости будут обходиться изолированностью контейнера от запускающей среды
- В) контейнер позволяет выбрать выводящий данные порт, что даст возможность продолжать как считывать, так и отправлять данные
- Г) логируемые данные также возможно считывать, поэтому проблем с их обработкой не должно быть
- Д) вносимые в настройки контейнера изменения будут требовать минимального времени на сохранение и использование благодаря “слоям” контейнера

Далее данные логов будут захвачены и обработаны запущенным python скриптом, который будет пересылать формируемые данные через отдельный порт на сервис Prometheus, измеряющий нужные метрики и запущенный в

своем контейнере, чьи данные будут передаваться еще одному контейнеру с запущенным сервисом Grafana.

Prometheus необходим для структуризации данных для последующего анализа в Grafana, который еще и может выводить наглядные и понятные иллюстрации обработанных данных.

Подробное описание серверных архитектур будет приведено в последующем разделе.

Получение данных

Сервер будет получать и обрабатывать данные после запуска сессии из двух источников:

- получение запросов от клиента;
- обработка данных из файлов чатов.

Первый способ будет реализован через считывание и обработку клиентских данных из сокета сессии.

Второй способ – будет применяться только к пользователям, находящимся в чате в нынешний момент:

1. при добавлении нового пользователя ему будут отправлены все сохраненные сообщения
2. при отправке нового сообщения сервер отправит всем клиентам все недостающие данные

Хранение чатов должно быть осуществлено при помощи стандартных файлов ОС Linux. Сообщения будут разделяться специальным символом, для каждого пользователя будет храниться позиция в файле, далее которой сообщения еще не отправлены. Для корректной обработки сообщений без опасности появления ошибок в силу многопоточности и асинхронности, необходима синхронизация соответствующего функционала. Она будет обеспечиваться интерфейсом соответствующего класса, что упростит разработку и понимание модели.

Получаемые сервером данные следует зафиксировать определенным набором для стабильности работы системы и простоты обработки. Серверу необходимы тип запроса и данных, отправленных вместе с ним.

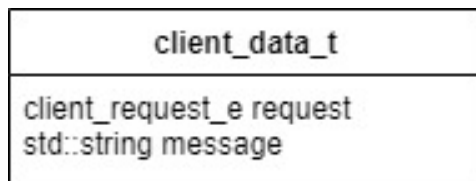


Рисунок 1 – Структура представления данных сервера

Передача данных

Для оптимального использования данных следует выбрать единую единицу информации, отправляемую сервером клиенту. Клиенту необходимо знать успешность выполнения операции на сервере, возможные возвращаемые им данные и конкретный тип клиентского запроса, на который этот ответ был сформирован. Альтернативная форма данных создаст проблемы при отладке продукта и усложнит его структуру.

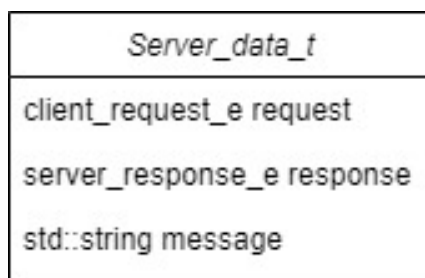


Рисунок 2 – Структура представления данных для клиента

Если обобщать пользователь может получать лишь результат своего запроса и новые сообщения, которые будут обработаны интерфейсом клиентского приложения. Созданная версия клиента будет базовой и упрощенной до необходимого минимума, но сервер должен иметь возможность простой доработки и расширения функционала.

Вся передача данных будет выполняться асинхронно для многопоточно-асинхронной архитектуры и с ожиданием отправки для многопоточного приложения. Такая разница в логике обмена сообщениями очевидно повлияет

на производительность, и разница между двумя реализациями будет заметна в итогах анализа.

Пользователь не сможет получить доступ к чатам, пока не займет свободное имя пользователя, после чего ему станет доступен список чатов для подключения, каждый из которых будет иметь свой список пользователей.

1.3 Выбор архитектур серверной части для анализа

Под архитектурой сервера следует понимать устройство компонентов, на которых он базируется, и взаимосвязей между ними. Для решения выбранной задачи следует разработать две архитектуры серверов.

Многопоточный сервер следует свести к максимально простой модели – конвейерная обработка запросов. Каждое из состояний клиента подразумевает различную логику обработки. При переходе из одного состояния в другое клиент будет переходить с одного «конвейера» к другому. Основными этапами будут: получение имени, получение списка чатов и выбор одного из них, нахождение в конкретном чате, отключение от сервера.

Необходимо отметить, что на каждый отдельный чат необходим отдельный поток обработки, который будет запускаться при подключении первого пользователя к нему и останавливаться при отключении от него последнего пользователя. В силу такой логики во избежание проблемы гонок необходимо произвести синхронизацию проверок для запуска и остановки потока.

Также следует выделить потенциальные преимущества и недостатки разрабатываемой архитектуры.

Достоинства:

- минимальное использование ООП результирует в минимальном использовании памяти

- использование проверенных и стабильных библиотек линукса дает уверенность в стабильности и прогнозируемости работы системы
- разделение этапов обработки данных в разные потоки облегчает отладку логики программы и позволяет использовать дополнительную производительность от многопоточности

Недостатки:

- передача данных от потока к потоку потенциально связана с проблемой гонок и ошибками в передаче данных
- запуск нового потока для каждого чата – ресурсозатратное действие
- остановка потока для ожидания нового сообщения уменьшает эффективность использования ресурсов потоков

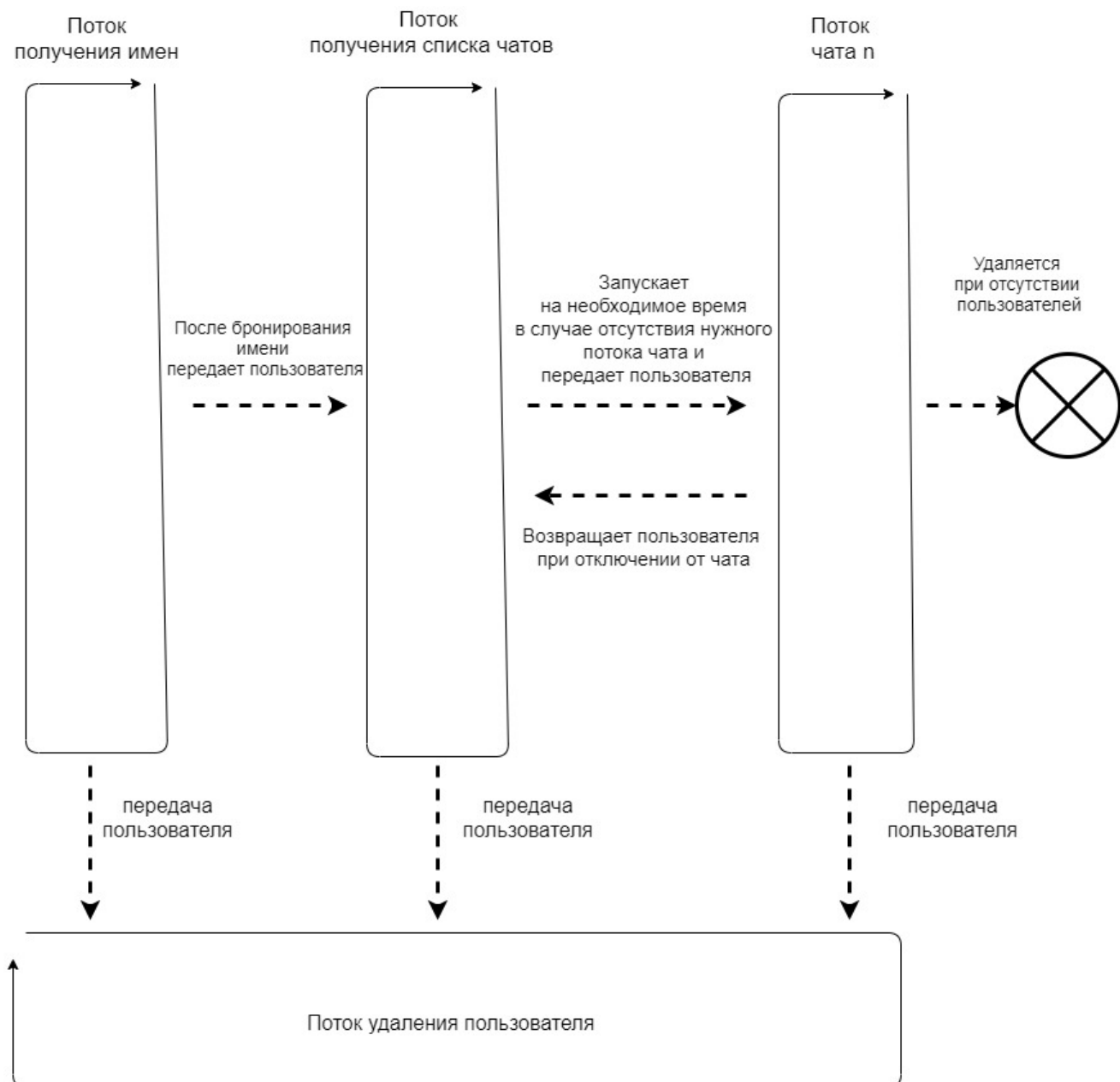


Рисунок 3 – Архитектура многопоточного сервера

Многопоточный асинхронный сервер имеет несколько более комплексную логику относительно предыдущего. Асинхронные задачи должны выполняться синхронизировано относительно задач своих групп, при том нужно различать что нужно синхронизировать относительно выбранной задачи, а что – нет. Поэтому для наиболее логичной организации был выбраны ООП в совместном использовании с корутинами.

Это позволяет создать необходимые интерфейсы классов, которые автоматически будут производить необходимые синхронизации в нужных

контекстах, а корутины дают возможность простой разработки асинхронной логики выполнения задач.

Модель устроена следующим образом: объекты класса чата хранятся и корректно управляются классом менеджера чатов, оба этих класса имеют свои уровни синхронизации. Класс сервера хранит в себе менеджер чатов, к которому будет обращаться создаваемый и запускаемый при подключении к серверу объект корутины сессии, который будет заниматься обработкой запросов пользователя асинхронно и параллельно работе сервера.

Такой архитектурный подход дает набор преимуществ:

- относительная простота разработки логики
- оптимизированность работы сервера за счет минимизации периодов ожидания какого-то действия благодаря использованию асинхронности
- безопасное использование ресурсов памяти – C++ известен возможными проблемами с ошибками памяти, максимальный упор на ООП дает возможность использовать преимущества высокопроизводительного языка с минимальным риском столкнуться с вышеописанным недостатком при правильной практике программирования – использованием умных указателей в качестве автоматизации защиты памяти
- использование пула потоков для асинхронной работы объединяет преимущества многопоточности и асинхронности в плане производительности
- инкапсуляция позволяет скрыть детали реализации от других объектов, обеспечивая простой и безопасный доступ к методам и свойствам объектов

Однако стоит отметить и потенциальные недостатки:

- большой упор на использование библиотек потенциально опасен сложностью в отладке ошибок
- ООП, дающий понятность и простоту модели, может увеличить расход памяти

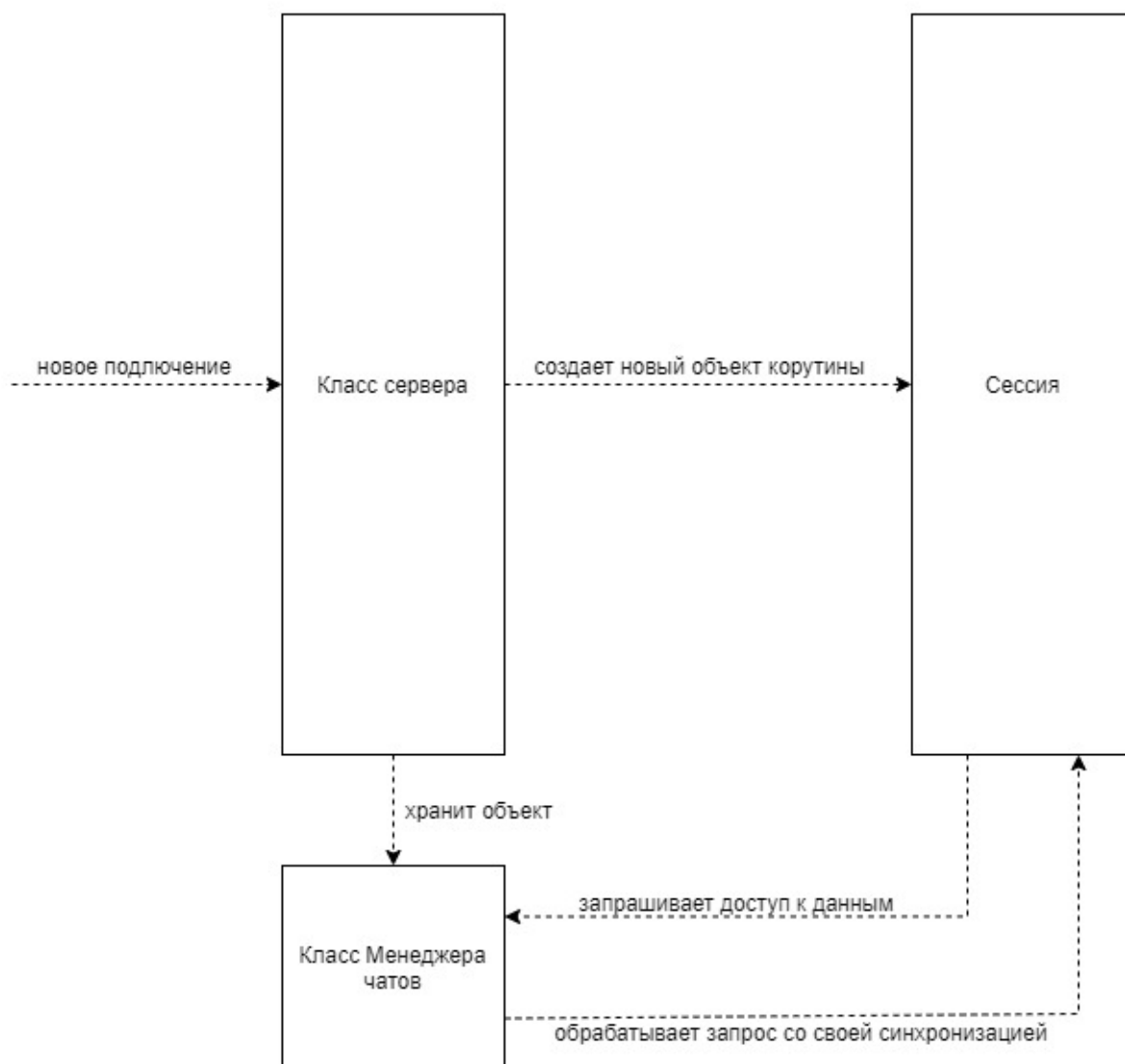


Рисунок 4 – Архитектура многопоточно-асинхронного сервера

ГЛАВА 2. РАЗРАБОТКА РЕАЛИЗАЦИЙ СЕРВЕРА И КЛИЕНТА

2.1 Установка и настройка необходимых компонентов

Перед непосредственной разработкой программного обеспечения требуется в первую очередь установить и настроить к работе необходимые компоненты – библиотеку BOOST и линкер CMake.

BOOST устанавливается при использовании менеджера зависимостей для C/C++ - Conan. Удобство данного инструмента главным образом заключается в полной автоматизации настройки среды разработки: достаточно прописать необходимые для разработки библиотеки и используемый линкер – в нашем случае это CMake. Также Conan приспособлен к кроссплатформенности, поддерживает большинство популярных библиотек, позволяет создавать свои.

Он устанавливается через скачивание необходимой версии с необходимого репозитория через терминал Linux.

Основные преимущества CMake:

- 1) Кроссплатформенность. CMake позволяет создавать кроссплатформенные проекты, а это многократно упрощает разработку, а также поддержку программного обеспечения относительно различных операционных систем, архитектур и компиляторов.
- 2) Простота и гибкость. Простой и гибкий синтаксис позволяет быстро и легко описывать собираемые проекты и их зависимости.
- 3) Автоматизация сборки.
- 4) Интеграция с другими инструментами. Многие IDE, менеджеры зависимости и другие инструменты разработки интегрируют CMake, а это дает возможность существенно ускорить и упростить процесс разработки.
- 5) Поддержка модульной структуры.

Преимущества библиотеки BOOST:

- 1) Широкий спектр функций, в том числе работа с сетью, вводом/выводом, строками, файлами и многим другим.
- 2) Обширная документация.
- 3) Проверенность и надежность – библиотека прошла множество тестов и признана качественным инструментом для разработчиков.
- 4) Поддерживает последние стандарты C++, а это очень пригодится мне в разработке.

2.2 Конструирование внешнего вида клиентского приложения

При разработке внешнего вида демонстрационной версии клиентского приложения было нужно отметить, что клиент в описанной системе находится в трех ключевых состояниях:

1. Без имени
2. Вне чата
3. В чате

Каждому из этих состояний должен соответствовать свой режим работы приложения.

Для первого состояния достаточно указать требования к формату вводимого имени, выделить поле для ввода и обозначить название приложения наверху.

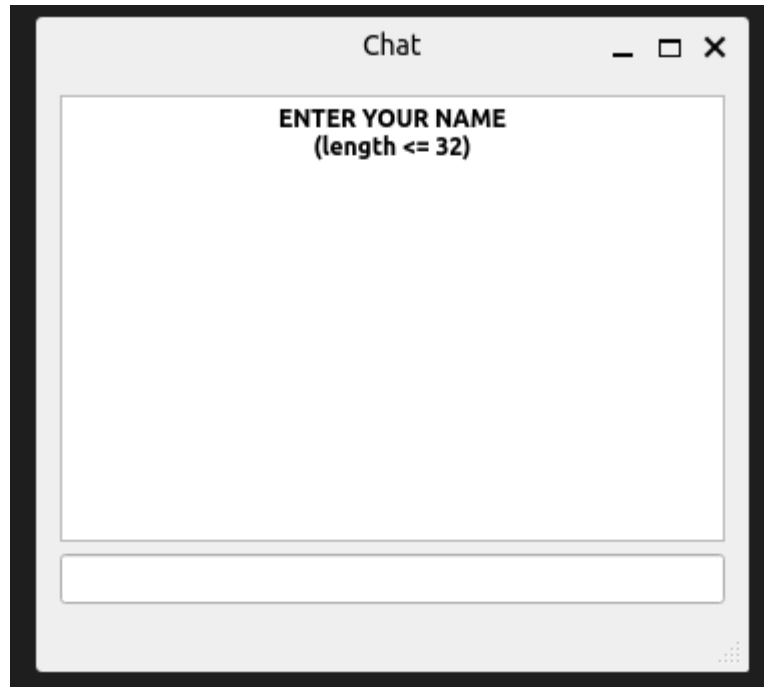


Рисунок 5 – Иллюстрация GUI регистрации

Если пользователь пытается занять уже используемое имя, он будет уведомлен об этом всплывающим окном. После его закрытия поле ввода имени будет очищено для нового ввода нового варианта имени пользователя.

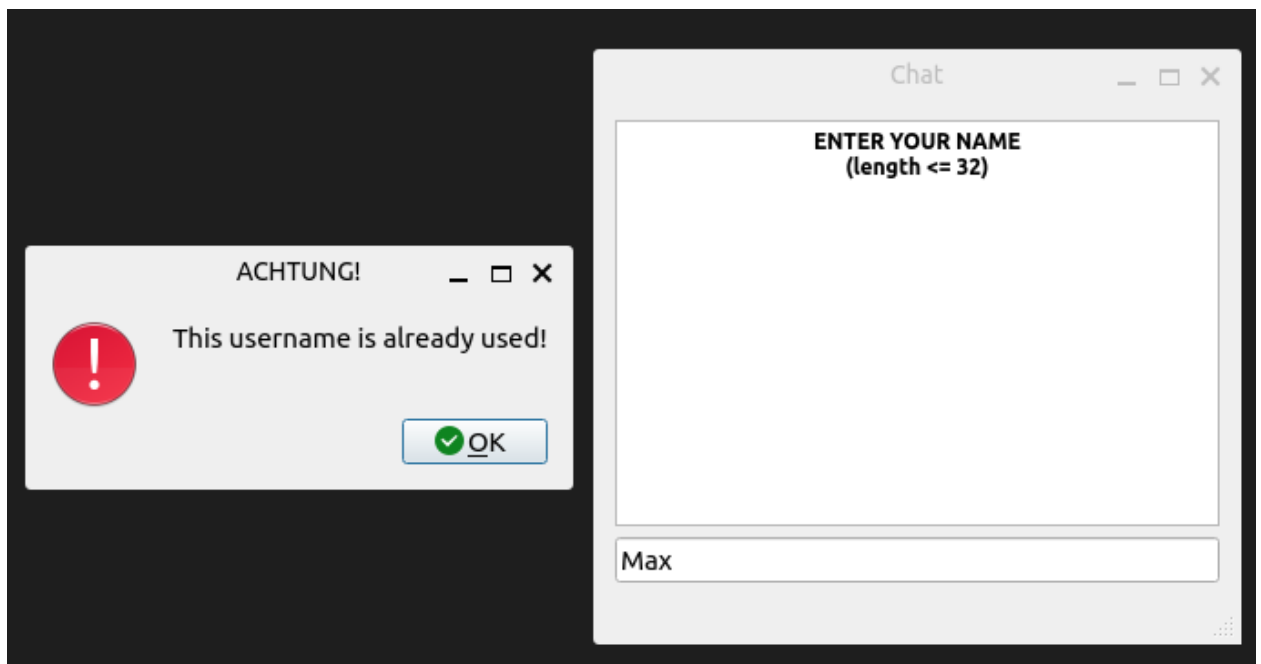


Рисунок 6 – Всплывающее окно при отказе в регистрации имени

В обратном случае приложение зарегистрирует пользователя и перейдет ко второму из режимов работы.

Для второго состояния необходимо рассмотреть чуть более сложную концепцию: постоянно обновляющийся список доступных чатов с полем для создания своего нового.

Каждый чат в списке должен быть представлен динамически созданной кнопкой подключения. После нажатия на любую из них на сервер будет передан запрос на подключение к выбранному чату, а приложение сменит свой режим работы на третье.

Стоит отдельно отметить, что при создании нового чата список чатов должен быть обновлен для каждого из пользователей, который зарегистрирован, но все еще находится вне какого-то либо чата.

Таким образом, пока пользователь не войдет в чат, список доступных, т.е. таким образом список кнопок подключения должен иметь возможность постоянного обновления.

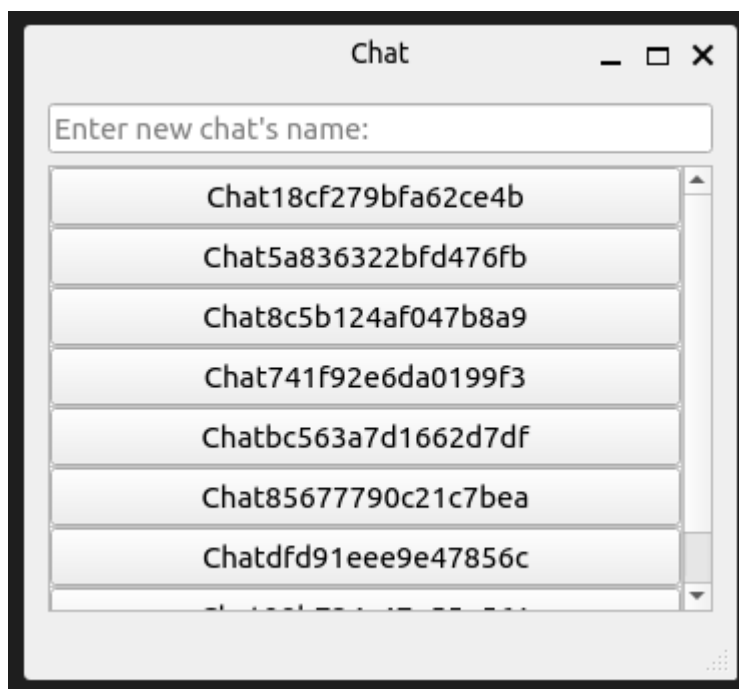


Рисунок 7 – GUI выбора чата из списка доступных

Если же пользователь попытается создать чат с неправильным (содержащим пробелы или излишне длинным) названием – он будет уведомлен о невозможности выполнения такого действия всплывающим окном. Поле ввода названия нового чата будет очищено после закрытия всплывающего окна.

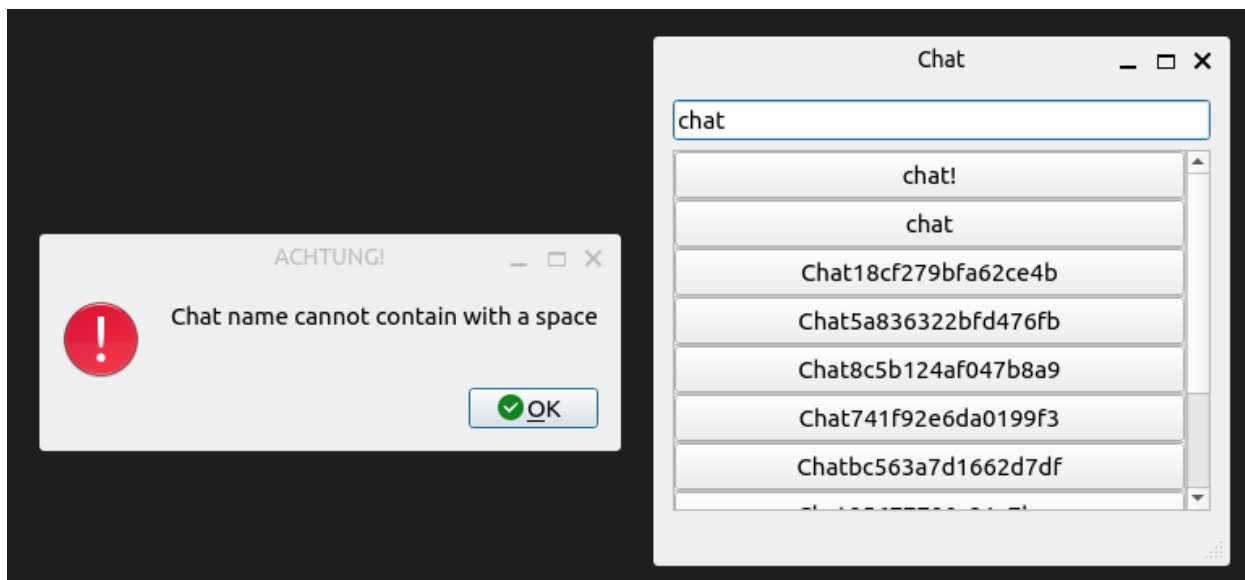


Рисунок 8 – Всплывающее окно при отказе в создании чата с неверным именем

Если же будет предпринята попытка создания чата с уже занятым именем, то запрос будет проигнорирован, а пользователь все еще сможет подключиться к чату с желаемым им именем.

При нажатии на кнопку выбранного чата приложение перейдет в последний, третий режим работы.

В чате пользователю необходимо предоставить следующий функционал:

1. Возможность покинуть чат
2. Возможность прочитать сообщения и пролистать их до более старых или новых
3. Возможность отправки своего сообщения

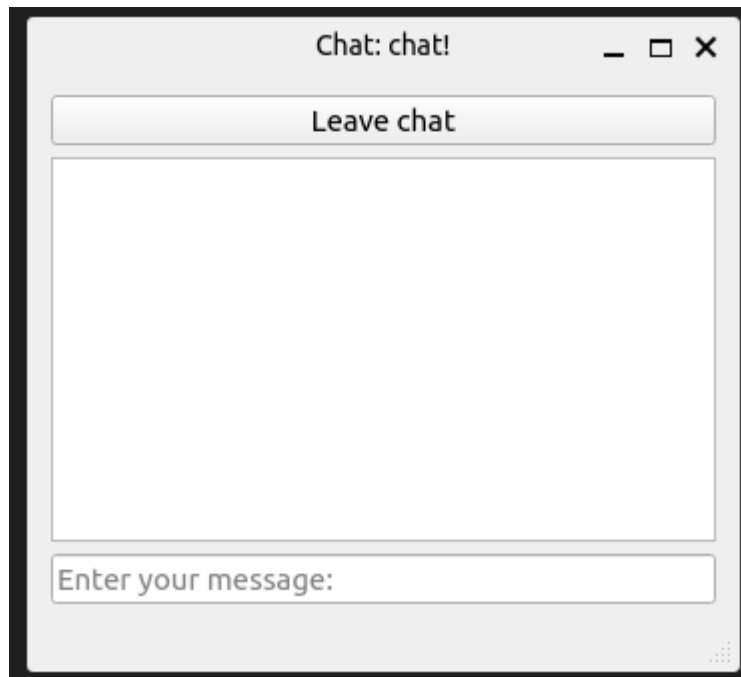


Рисунок 9 – GUI чата

Сверху будет указано название чата, к которому подключен пользователь

Сообщение вводится в нижнее поле и отправляется нажатием Enter. После отправки сообщение рассылается всем пользователям, подключенным к чату, и отражается в списке сообщений следующим образом:

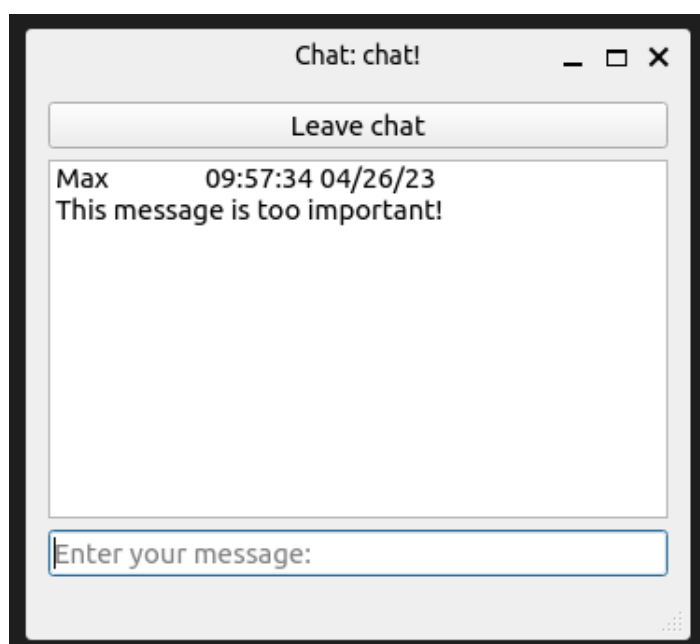


Рисунок 10 – Отображение сообщения в GUI чата

У каждого сообщения отмечаются автор, время и дата отправки помимо самого содержания.

Нажатие на кнопку выхода из чата пользователь переходит обратно во второй режим работы приложения и получает актуальный список чатов, откуда может как снова вернуться в изначальный чат, так и подключиться к другому.

При закрытии приложения пользователь отключается и выходит из системы. Его пользовательское имя освобождается для остальных и может быть уже занято при последующем подключении.

2.3 Реализация многопоточного и многопоточного асинхронного серверов

```
class Logger {
    Logger() = default;
    Logger(const Logger&) = delete;
public:
    static Logger& GetInstance() {
        static Logger obj;
        return obj;
    }
    // Выведите в поток все аргументы.
    void Log(const std::string_view, boost::json::value);
    void InitBoostLog();
}
```

В обеих реализациях серверов используется класс Logger, отвечающий за создание и вывод логов:

Данный класс является ярким примером принципа Singleton: невозможно получить в распоряжение более одного объекта класса, таким

образом все настройки, заданные при помощи метода **InitBoostLog**, будут легко применимы по всему проекту:

```
void Logger::InitBoostLog(){
    //setting up logging
    logging::add_common_attributes();
    logging::add_console_log(
        std::cout,
        keywords::format = &logger::MyFormatter,
        keywords::auto_flush = true
    );
}
```

В данном случае настраивается использование атрибутов системой логгирования, печать логов в стандартный поток вывода без задержки по настроенному формату.

2.3.1 Многопоточный сервер

Основные рутины потоков сервера

Рутинa **user_name_enter** отвечает за регистрацию пользователей в системе.

На вход ей подаются такие параметры, как входной и выходной конец трубы данных (pipe). Рутинa получает новых пользователей из входного конца при их подключении к серверу, после чего добавляет их в список отслеживаемых и переходит к их обработке запросов на регистрацию.

```
while((bytes = read(end_to_read_from, &new_socket, sizeof(int))) > 0)
{
    ev.data.fd = new_socket;
    epoll_ctl(no_name_epoll_fd, EPOLL_CTL_ADD, ev.data.fd, &ev);
}
```

```
}
```

При удачной регистрации пользователь удаляется из отслеживаемых и по выходному концу передается в следующую рутину – list_of_chats.

```
user_data[client_sockfd] = received.message_text;
used_usernames.insert(received.message_text);
resp.responce = s_success;
epoll_ctl(no_name_epoll_fd, EPOLL_CTL_DEL, client_sockfd, &ev);
if(!send_resp_to_client(&resp, client_sockfd)) write(disco_pipe[1],
&client_sockfd, sizeof(int));
std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);
write(end_to_write_to, &client_sockfd, sizeof(int));
```

Если пользователь отключается на моменте регистрации – он передается рутине disconnect для проведения процедур отключения от системы.

```
if(received.request == c_disconnect || err == 0){
    epoll_ctl(no_name_epoll_fd, EPOLL_CTL_DEL, client_sockfd,
&ev);

std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);
    write(disco_pipe[1], &client_sockfd, sizeof(int));
    boost::json::value data = {
        {"socket", client_sockfd}
    };
    logger::Logger::GetInstance().Log("User disconnected"sv, data);
    continue;
}
```

Обе этих рутины будут также рассмотрены чуть позже.

Бесконечный цикл обеспечивает непрерывную обработку запросов, получение новых сообщений опирается на использование `epoll`.

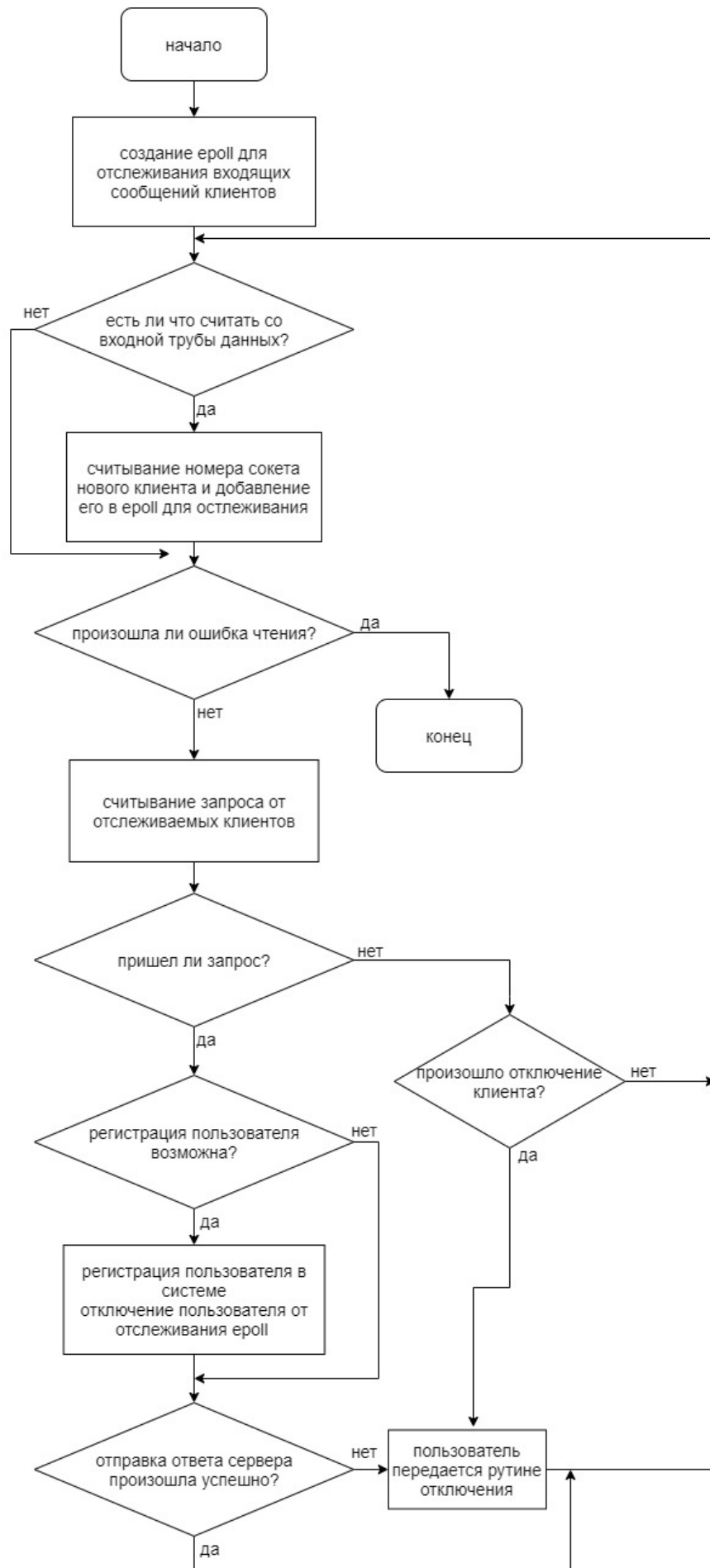


Рисунок 11 - Структурная схема алгоритма user_name_enter

Рутина list_of_chats отвечает за предоставление зарегистрированным пользователям вне чатов актуального списка доступных.

В этой рутине предусмотрены обработка подключения к чату, создания нового чата, отключения от системы.

Подключение к чату:

```
case c_connect_chat:
{
resp.request = c_connect_chat;
bool start_ok = true;
if (chats.count(received.message_text) == 0){
resp.responce = s_failure;
resp.message_text = "Can't connect to a non-existant chat";
break;
}
//if chat was unused -> setup
chats[received.message_text].mtx.lock();
if(chats[received.message_text].pipe.in == -1){
int tmp[2];
for(auto socket : closed_sockets){ // handle closed sockets
epoll_ctl(no_chat_epoll_fd, EPOLL_CTL_DEL, socket, &ev);
clients_without_chat.erase(socket);
std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);
write(disco_pipe[1], &socket, sizeof(int));
}
}
break;
```

```
}
```

Создание нового чата:

```
case c_create_chat:
{
    resp.request = c_create_chat;
    int check = creat((received.message_text + ".chat").c_str(), S_IRWXG |
S_IRWXO | S_IRWXU);
    if (errno == EEXIST) {
        resp.responce = s_failure;
        resp.message_text = "Chat with such name already exists";
        if(!send_resp_to_client(&resp, client_sockfd)) goto
c_disconnect_list_chats;
    }
    else if (check < 0){
        resp.responce = s_failure;
        resp.message_text = "Server error: could not create chat, try again later";
        boost::json::value data = {
            {"chat", received.message_text},
            {"error", std::error_code{errno,
std::generic_category()}.message()},
        };
        logger::Logger::GetInstance().Log("Chat creation error"sv, data);
        if(!send_resp_to_client(&resp, client_sockfd)) goto
c_disconnect_list_chats;
    }
    else{
        chats[received.message_text].subs = {};
        chats[received.message_text].pipe.in = -1;
        chats[received.message_text].pipe.out = -1;
```

```

boost::json::value data = {
    {"chat", received.message_text}
};
logger::Logger::GetInstance().Log("Chat created"sv, data);
std::unordered_set<int> closed_sockets;
for (int socket : clients_without_chat){
    if(!send_available_chats(socket)){ // if client disconnects while being
written into
        closed_sockets.insert(socket);
    }
}
for(auto socket : closed_sockets){ // handle closed sockets
    epoll_ctl(no_chat_epoll_fd, EPOLL_CTL_DEL, socket, &ev);
    clients_without_chat.erase(socket);
std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);
    write(disco_pipe[1], &socket, sizeof(int));
}
}
break;
}

```

Отключение от системы:

```

case c_disconnect:
c_disconnect_list_chats:
    epoll_ctl(no_chat_epoll_fd, EPOLL_CTL_DEL, client_sockfd, &ev);
    clients_without_chat.erase(client_sockfd);

    std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);
    write(disco_pipe[1], &client_sockfd, sizeof(int));

```

```
break;
```

Новые пользователи считываются из рутины `username_enter`, если же пользователь отключается от системы, он все так же передается в рутину `disconnect`.

При подключении к чату возникает необходимость в синхронизированной проверке наличия в нем пользователей. Если чат пуст, то его отдельный поток не запущен, тогда нужно провести процедуры подготовки к запуску рутины чата в отдельном потоке и только после передать ей пользователя.

В этой рутине предусмотрены обработка подключения к чату, создания нового чата, отключения от системы.

Новые пользователи считываются из рутины `username_enter`, если же пользователь отключается от системы, он все так же передается в рутину `disconnect`.

При подключении к чату возникает необходимость в синхронизированной проверке наличия в нем пользователей. Если чат пуст, то его отдельный поток не запущен, тогда нужно провести процедуры подготовки к запуску рутины чата в отдельном потоке и только после передать ей пользователя.

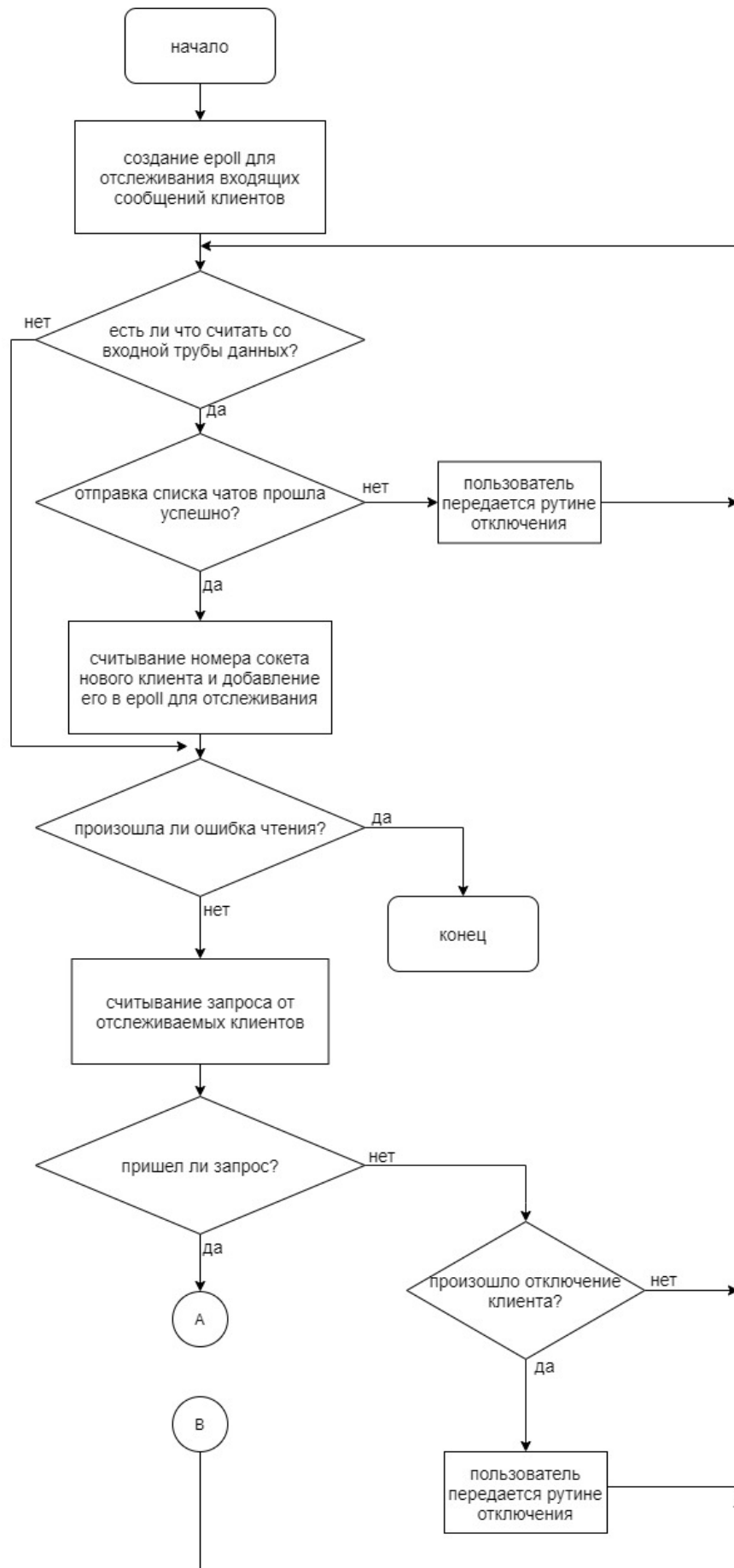


Рисунок 12 - Структурная схема алгоритма list_of_chats без обработки данных

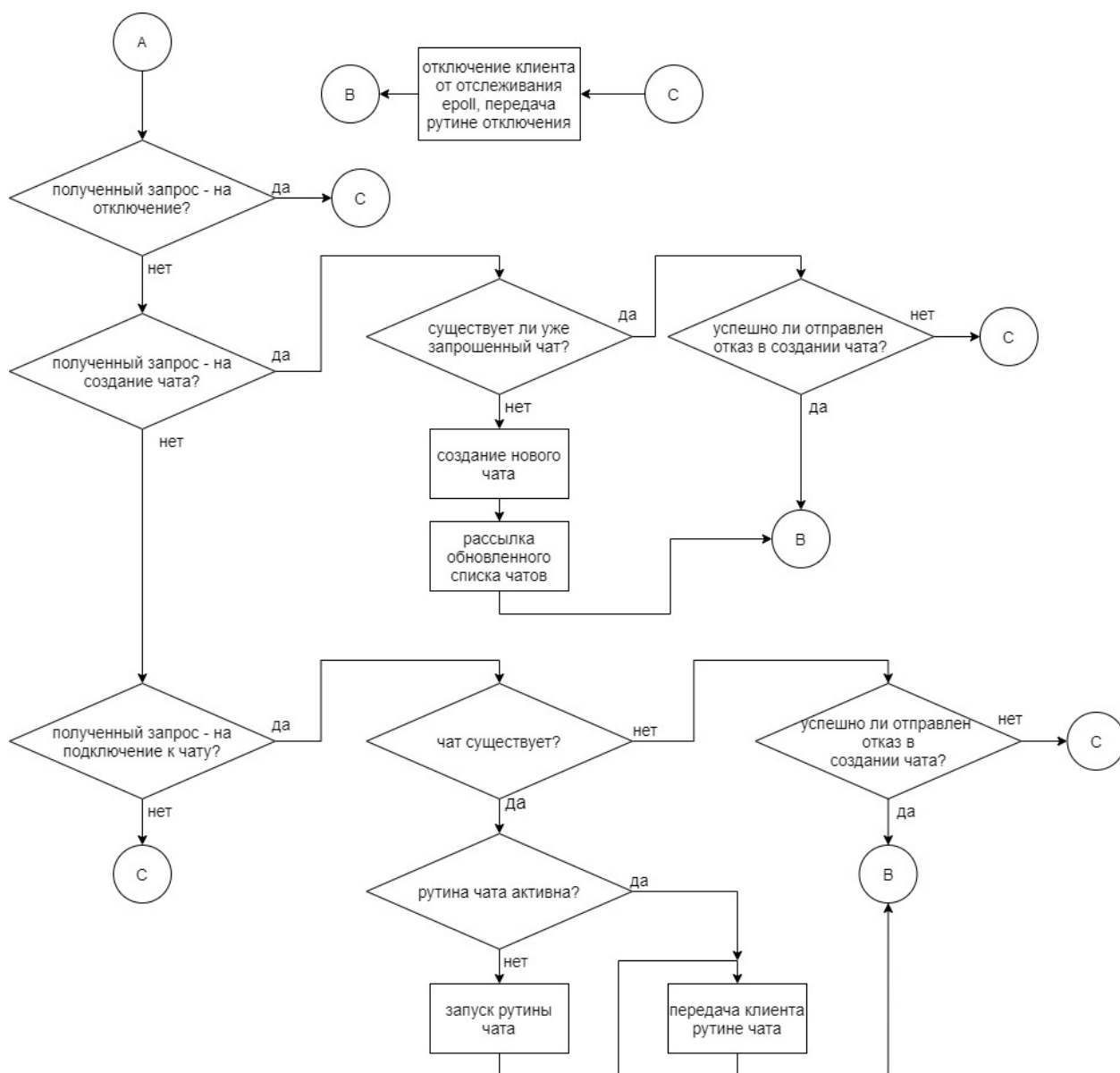


Рисунок 13 - Структурная схема алгоритма обработки запросов list_of_chats

Рутинa chat_message отвечает за обработку логики каждого конкретного чата в отдельном потоке.

В этой рутине предусмотрен функционал выхода из чата, отключения от системы и отправки нового сообщения.

Выход из чата:

```

case c_leave_chat:
{
if(epoll_ctl(chat_epoll_fd, EPOLL_CTL_DEL, fd, &ev) == -1){
boost::json::value data = {
{"error", std::error_code{errno, std::generic_category()}.message()},
{"socket", fd},
{"chat", chat_name}
};
logger::Logger::GetInstance().Log("Failed to remove socket from epoll"sv,
data);
}

messages_offset.erase(fd);
chats[chat_name].subs.erase(fd); //unsubscribing
std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);

if(write(list_chats_pipe[1], &fd, sizeof(int)) < 0){
boost::json::value data = {
{"error", std::error_code{errno, std::generic_category()}.message()},
{"socket", fd},
{"chat", chat_name}
};
logger::Logger::GetInstance().Log("Failed to send client's fd to no_chat
routine"sv, data);
}
boost::json::value data = {
{"chat", chat_name},
{"socket", fd}
};

```

```
logger::Logger::GetInstance().Log("User left chat"sv, data);  
}  
goto OUT_OF_CYCLE;
```

Отключение от системы:

```
case c_disconnect:  
{  
boost::json::value data = {  
{"chat", chat_name},  
{"socket", fd}  
};  
logger::Logger::GetInstance().Log("Disconnecting user"sv, data);  
if(epoll_ctl(chat_epoll_fd, EPOLL_CTL_DEL, fd, &ev) == -1){  
boost::json::value data = {  
{"error", std::error_code{errno, std::generic_category()}.message()},  
{"socket", fd},  
{"chat", chat_name}  
};  
logger::Logger::GetInstance().Log("Failed to remove socket from epoll"sv,  
data);  
}  
messages_offset.erase(fd);  
  
chats[chat_name].subs.erase(fd); //unsubscribing  
std::atomic_thread_fence(std::memory_order::memory_order_seq_cst);  
write(disco_pipe[1], &fd, sizeof(int));  
}  
goto OUT_OF_CYCLE;
```

Отправка сообщения:

```
case c_send_message:
{
boost::json::value data = {
{"chat", chat_name},
{"socket", fd},
{"message", client_data.message_text}
};
logger::Logger::GetInstance().Log("Sending message"sv, data);
chat_file << user_data[fd] << '\t';
chat_file.write(timedate_string, strlen(timedate_string));
chat_file << std::endl << client_data.message_text << "\r";
}
break;
```

После каждой итерации цикла рутины происходит отправка недостающих сообщений для каждого пользователя чата, а также отключение пользователей, закрывших соединение.

Стоит отметить, что во избежание ошибок синхронизации отправки сообщений при каждой итерации цикла алгоритма проходит отправка потенциально неотправленных сообщений всем участникам чата.

В этой рутине, как и во всех остальных, можно встретить использование `std::atomic_thread_fence`. Этот класс используется для того, чтобы предотвратить перестановку операций компилятором во время оптимизаций.

Для понятной иллюстрации алгоритма работы этой рутины достаточно показать часть с обработкой запросов, т.к. основная его часть практически неотличима от предыдущей рутины, за исключением того, что для успешного

запуска рутины чата необходимо успешное открытие файла, и того, что при отсутствии подключенных пользователей рутина остановится.

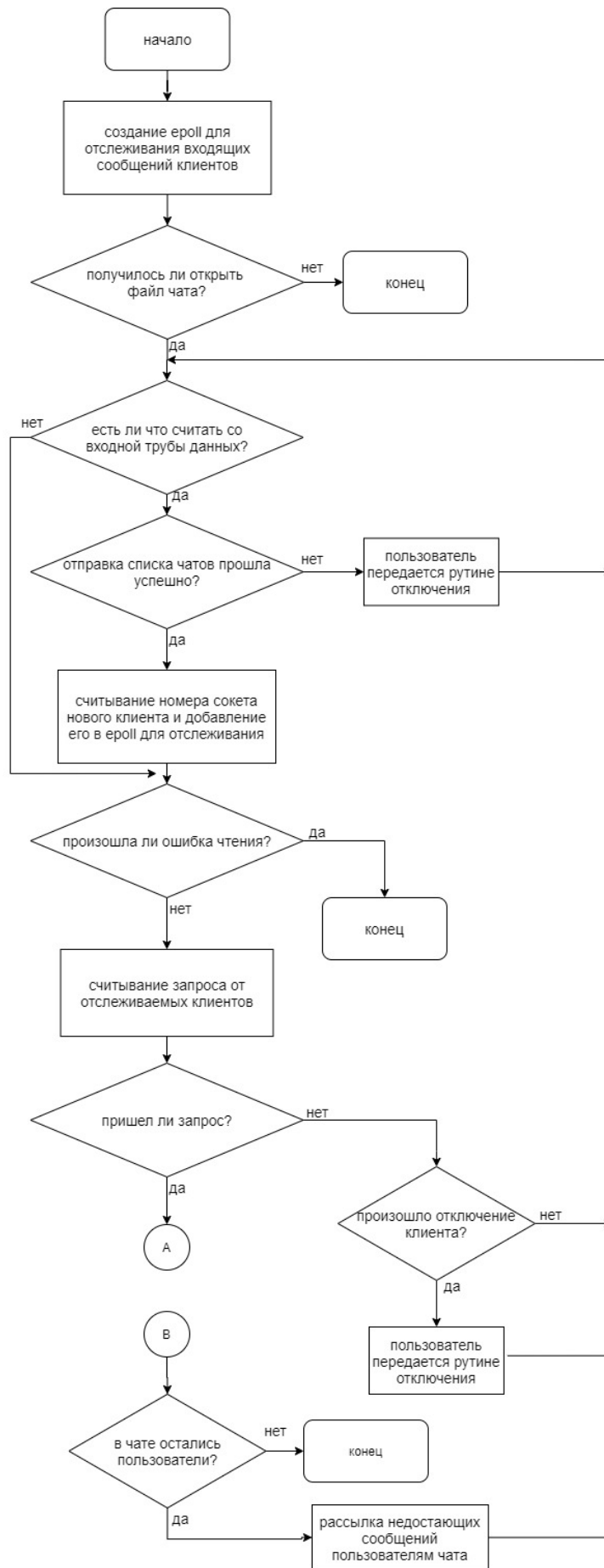


Рисунок 14 - Структурная схема алгоритма chat_message без обработки данных

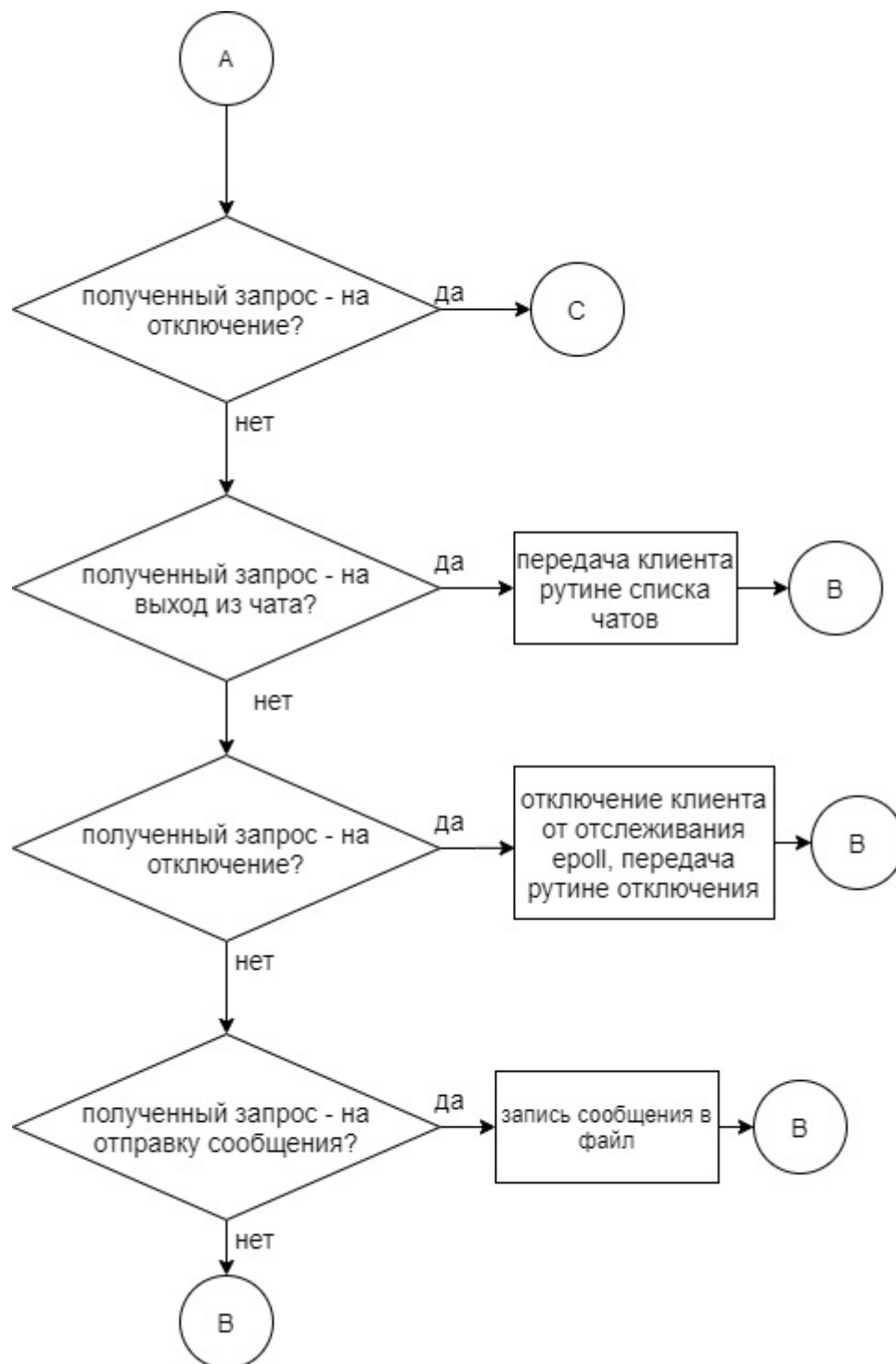


Рисунок 15 - Структурная схема алгоритма обработки запросов рутины chat_message

Последняя рутина, disconnect, отвечает за отключение пользователя:


```

void disconnect(){
    int read_bytes, fd;
    while (true) {
        while ((read_bytes = read(disco_pipe[0], &fd, sizeof(int))) > 0) {
            boost::json::value data = {
                {"socket", fd},
                {"name", user_data[fd]}
            };
            logger::Logger::GetInstance().Log("User was disconnected"sv,
data);

            end_resp_to_client(fd);
            used_usernames.erase(user_data[fd]);
            user_data.erase(fd);
        }
        if (read_bytes < 0 && errno != EAGAIN && errno != EINTR) {
            boost::json::value data = {
                {"error",
std::error_code{errno,
std::generic_category()}.message()}
            };
            logger::Logger::GetInstance().Log("Pipe      read      error
disconnected"sv, data);
        }
    }
}

```

В этой рутине происходит освобождение занятого имени пользователя и удаление промежуточных данных о нем из системы.

Во всех рутинах, описанных выше, применяются трубы (pipes). Они имеют максимальный размер данных, защищенных от проблем гонок. Нам не требуется синхронизация относительно них т.к. передаваемые данные пользователей меньше заданного размера.

Для данной рутины структурная схема алгоритма не является необходимой в силу ее относительной простоты.

Функционал отправки и получения данных

Функция `read_request_from_client` используется для простоты и единообразия считывания данных во всей реализации сервера.

```
int read_request_from_client(client_data_t* received, int sockfd){
    size_t length;
    int err;

    err = recv(sockfd, (void*)&(received->request),
sizeof(client_request_e), MSG_WAITALL);
    if (err != sizeof(client_request_e))
        return err;

    err = recv(sockfd, (void*)&length, sizeof(size_t),
MSG_WAITALL);
    if (err != sizeof(size_t))
        return err;

    received->message_text.resize(length);
    err = recv(sockfd, (void*)(received->message_text.data()), length,
MSG_WAITALL);
    if (err != length)
        return err;

    boost::json::value data = {
        {"request",
        {
```

```

        {"client_request_e",    StringifyRequest(received-
>request)},

        {"message", received->message_text}

    }

}

};

logger::Logger::GetInstance().Log("Read request from client"sv,
data);

return length + 1;

}

```

Функция `send_resp_to_client` выполняет аналогичные задачи относительно отправки данных пользователю.

```

bool send_resp_to_client(const server_data_t* resp, int sockfd){
    #if DEBUG_TRACE
        printf("%d : - send_resp_to_client()\n", getpid());
    #endif
    int written_bytes;
    written_bytes = send(sockfd, (void*)&resp->request,
sizeof(client_request_e), MSG_WAITALL | MSG_NOSIGNAL);
    if(written_bytes < 0) return false;
    written_bytes = send(sockfd, (void*)&resp->response,
sizeof(server_response_e), MSG_WAITALL | MSG_NOSIGNAL);
    if(written_bytes < 0) return false;
    size_t length = resp->message_text.size();
    written_bytes = send(sockfd, (void*)&length, sizeof(size_t),
MSG_WAITALL | MSG_NOSIGNAL);
    if(written_bytes < 0) return false;
}

```

```

        written_bytes = send(sockfd, (void*)(resp->message_text.c_str()),
resp->message_text.size(), MSG_WAITALL | MSG_NOSIGNAL);
        if(written_bytes < 0) return false;

        boost::json::value data = {
            {"response",
                {
                    {"client_request_e", StringifyRequest(resp->request)},
                    {"server_response_e", StringifyResponse(resp-
>responce)},
                    {"message", resp->message_text}
                }
            }
        };
        logger::Logger::GetInstance().Log("Sent response to client"sv,
data);

        return true;
    }

```

Инициализация сервера

После настройки параметров вроде адреса и порта сервера нужно начать отслеживать уже созданные ранее чаты, за это отвечает отдельная функция – `server_starting`.

```

int server_starting(){
    namespace fs = std::filesystem;

    boost::json::value data = { {"pid", getpid()} };
    logger::Logger::GetInstance().Log("Server started"sv, data);
}

```

```

fs::path path_of_chats(".");
for (const auto& entry : fs::directory_iterator(path_of_chats)) {
    if (entry.is_regular_file() && fs::path(entry).extension() ==
".chat") {
        chats[fs::path(entry).stem()].subs = { };
    }
}
return 1;
}

```

Если файл, обнаруженный в выбранной папке чатов, имеет расширение .chat, то файл чата учитывается системой, его название записывается, тем самым делая его доступным при дальнейшей работе.

2.3.2 Многопоточно-асинхронный сервер

Эта версия сервера в куда большей степени опирается на ООП, поэтому необходимо в первую очередь описать ключевые классы

Класс Chat – помощь в упрощении синхронизации работы с чатами и удобный интерфейс.

Конструктор класса выполняет не только стандартную задачу создания объекта класса, но и создает файл чата при его отсутствии на момент запуска.

Метод AddUser отвечает за добавление нового пользователя в чат. После подключения ему будут все сообщения, отправленные до запуска метода.

Метод SendMessage отправляет сообщение, если указанный пользователь находится в чате. Все пользователи чата получают отправленное сообщение.

Метод ContainsUser отвечает за проверку наличия пользователя в чате. Используется по большей части описанным позже классом ChatManager для корректной работы всей системы.

Метод DeleteUser отвечает за отключение пользователя из чата, если он вообще там находится.

Каждый из этих методов синхронизируется через инкапсулированный объект strand[1] таким образом, чтобы все асинхронные запросы к методам класса выполнялись отдельно и последовательно.

```
class Chat{
public:
    Chat(net::io_context& io, const std::filesystem::path& chat_path) :
strand(net::make_strand(io)), path(chat_path){

        // file.exceptions(std::ifstream::failbit | std::ifstream::badbit );
        try{
            std::ofstream {path, std::fstream::app};
            // If no file is created, then show the error message.
        } catch(std::ifstream::failure e){
            throw e;
        }
    }

    boost::asio::awaitable<void>
AddUser(std::shared_ptr<tcp::socket> socket, std::string username);
    boost::asio::awaitable<void>
SendMessage(std::shared_ptr<tcp::socket> socket, std::string message);
    boost::asio::awaitable<bool>
ContainsUser(std::shared_ptr<tcp::socket> socket);

    boost::asio::awaitable<void>
DeleteUser(std::shared_ptr<tcp::socket> socket);
private:
```

```

        net::strand<net::io_context::executor_type> strand;
        std::unordered_map<std::shared_ptr<tcp::socket>, std::streamoff>
messages_offset; // Session to offset in chat file
        std::unordered_map<std::shared_ptr<tcp::socket>, std::string>
usernames; // Session to username
        std::filesystem::path path; // Path to the file
    };

```

Также класс хранит имена пользователей и соответствующие каждому пользователю сдвиг внутри файла чата, а также путь до самого файла.

Класс ChatManager – основа логики сервера, он содержит занятые имена пользователя, управляет созданием новых чатов, обновлением их списка и подключением к любому из них.

```

class ChatManager{
public:
    ChatManager(net::io_context& io, std::string path) : io_(io),
chats_strand_(net::make_strand(io)),
usernames_strand_(net::make_strand(io)), path_of_chats(path.data()){
        namespace fs = std::filesystem;
        if(!fs::exists(path_of_chats) || !fs::is_directory(path_of_chats))
fs::create_directory(path_of_chats);
        for (const auto& entry : fs::directory_iterator(path_of_chats)) {
            if (entry.is_regular_file() && fs::path(entry).extension() ==
".chat") {
                chats_.insert({fs::path(entry).stem().string(), Chat(io,
entry.path())});
            }
        }
    }
}

```

```

        boost::asio::awaitable<bool>                SetName(std::string,
std::shared_ptr<tcp::socket>);

        boost::asio::awaitable<std::string> ChatList();

        boost::asio::awaitable<void>
UpdateChatList(std::shared_ptr<tcp::socket>);

        boost::asio::awaitable<void>                ConnectChat(std::string,
std::shared_ptr<tcp::socket>);

        boost::asio::awaitable<bool> CreateChat(std::string);

        boost::asio::awaitable<bool>
LeaveChat(std::shared_ptr<tcp::socket>);

        boost::asio::awaitable<void>                SendMessage(std::string,
std::shared_ptr<tcp::socket>);

        boost::asio::awaitable<void>
Disconnect(std::shared_ptr<tcp::socket>);

    private:
        std::unordered_map<std::string,  Chat>  chats_; // чаты с
юзернеймами

        std::unordered_map<std::shared_ptr<tcp::socket>, User> users_; //
юзеры

        net::io_context& io_;

        net::strand<net::io_context::executor_type>        chats_strand_,
usernames_strand_;

        std::filesystem::path path_of_chats;

};

```

Конструктор помимо основной своей функции также выполняет задачу учета всех файлов чатов, созданных ранее.

Метод `SetName` отвечает за установку имени пользователя при регистрации. До прохождения этого этапа пользователь не может взаимодействовать с чатами.

Метод `ChatList` выдает актуальный список чатов при использовании. Применяется в `UpdateChatList` и не только.

Метод `UpdateChatList` рассылает актуальный список чатов всем зарегистрированным пользователям, находящимся вне чатов.

Метод `ConnectChat` подключает пользователя к запрошенному чату, если он существует.

Метод `CreateChat` создает новый чат, если он еще не существует.

Метод `LeaveChat` удаляет пользователя из выбранного чата, если он находится в нем.

Метод `SendMessage` позволяет пользователю отправить сообщение, если он на момент запроса находится в чате.

Метод `Disconnect` отвечает за отключение отсоединившихся пользователей от системы. Их данные будут удалены, а имя пользователя – освобождено.

Все методы используют синхронизацию через два `strand` – относительно имен пользователей и относительно списка чатов.

Оба предыдущих класса во всех методах используют корутины, т.к. нужно асинхронно выполнять запросы к ним из корутины сессии.

<pre>boost::asio::awaitable<void> Session(tcp::socket&& socket_, ChatManager* chatManager);</pre>

Использование корутин дает возможность применить всю мощь асинхронности: в нужный момент выполнение блока кода корутины приостанавливается, сохраняя последнее состояние. После того, как придет очередь вернуть управление остановленной корутине, она продолжит свою работу с сохраненными свойствами.

Класс Server – класс, на который опирается вся логика подключения к системе новых пользователей.

Класс сервера постоянно находится в динамической памяти, продолжая принудительно поддерживать свой срок жизни, захватывая при каждом вызове методов `shared_ptr` на самого себя. Как только сервер остановится, его объект автоматически удалится.

```
class Server : public std::enable_shared_from_this<Server> {
public:
    explicit Server(net::io_context& ioc, const tcp::endpoint& endpoint,
std::string chats_path)
        : ioc_(ioc)
        , acceptor_(net::make_strand(ioc))
        , chat_manager_(ioc, chats_path)
        {
            acceptor_.open(endpoint.protocol());
            acceptor_.set_option(net::socket_base::reuse_address(true));
            // Привязываем acceptor к адресу и порту endpoint
            acceptor_.bind(endpoint);
            acceptor_.listen(net::socket_base::max_listen_connections);
        }
    void Run();
private:
    net::io_context& ioc_;
    tcp::acceptor acceptor_;
    ChatManager chat_manager_;
    std::shared_ptr<Server> GetSharedThis();
    void DoAccept();
    void OnAccept(sys::error_code ec, tcp::socket socket);
    void AsyncRunSession(tcp::socket&& socket);
```

```
};
```

Метод Run запускает работу сервера, он начинает принимать соединения, впервые запуская метод DoAccept.

Метод DoAccept вызывается для приема нового соединения. Происходит его асинхронное ожидание, к которому привязывается обработчиком OnAccept.

Метод OnAccept вызывается предыдущим методом, когда соединение произошло, чтобы запустить для него отдельную сессию и переключить сервер к ожиданию нового соединения вызовом метода DoAccept..

Метод AsyncRunSession вызывается методом OnAccept чтобы запустить отдельную корутину сессии от нового соединения. Она стартует в режиме detached, т.е. не имеет обработчика при остановке работы и идет отдельным потоком действий в контексте асинхронности.

Полученная система позволяет динамически настраивать расходуемые на работу ресурсы при сохранении производительности. Классовая структура многократно упрощает читаемость и тестируемость кода.

Для еще более глубокого понимания принципов работы системы следует проиллюстрировать их диаграммой классов. Использование структурной схемы алгоритма нецелесообразно, т.к. процессы происходят асинхронно, вызываясь по отдельности объектами классов.

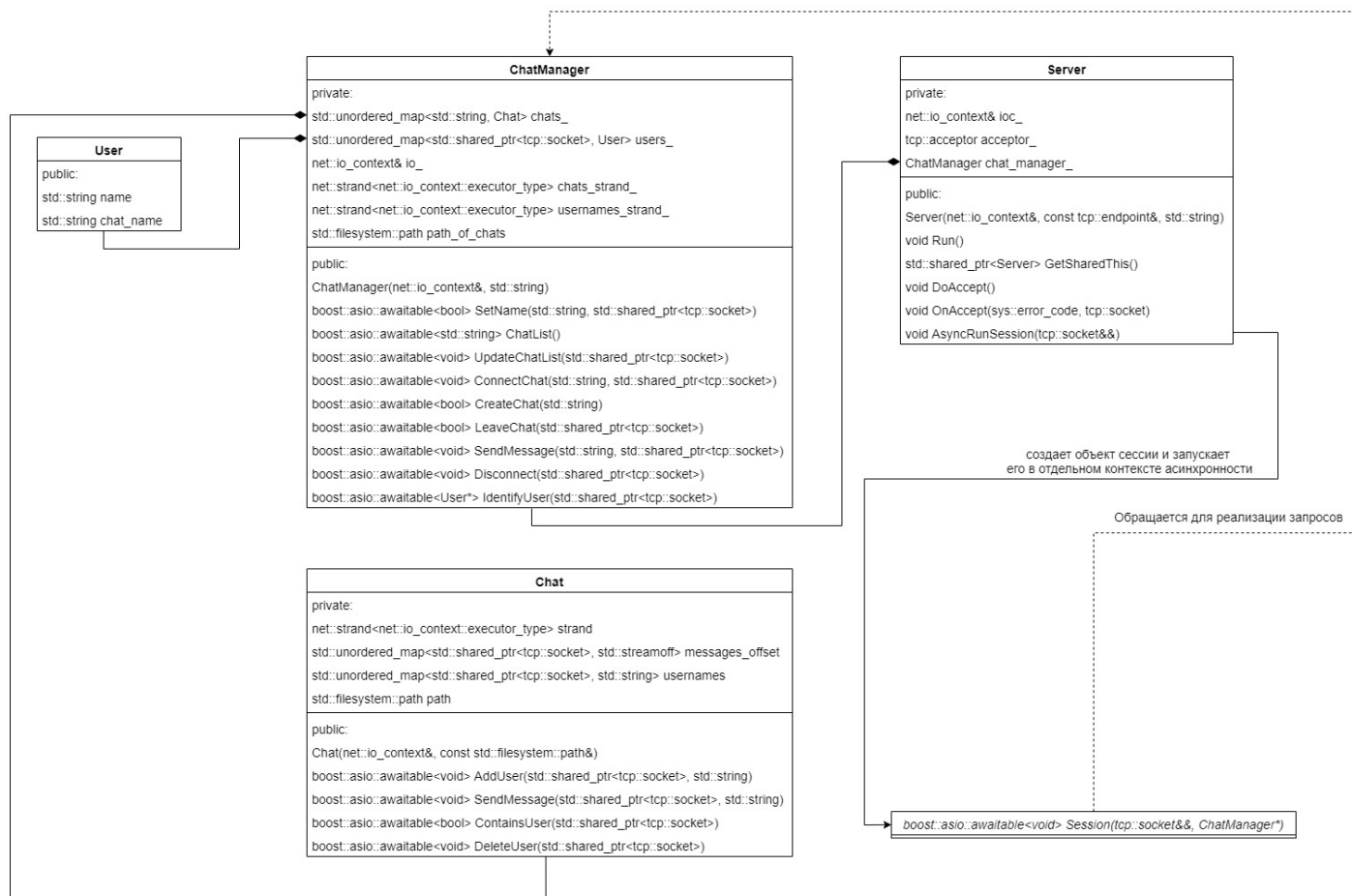


Рисунок 16 - Диаграмма классов многопоточно-асинхронного сервера

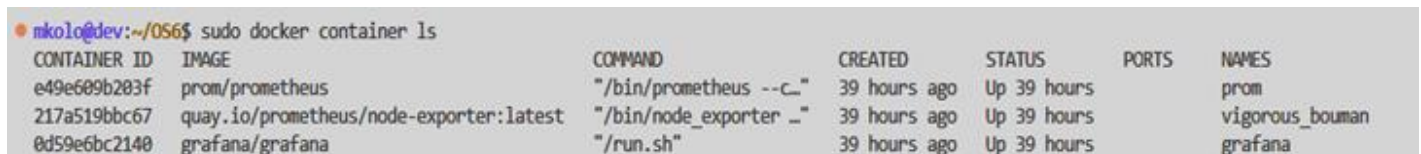
ГЛАВА 3. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

В данном разделе будут проанализированы работоспособность тестирующей системы и производительность разбираемых архитектур серверов.

При иллюстрации данных тестирования, графическое отображение результатов может быть крайне полезным инструментом для

3.1 Тестирование функционирования системы

Т.к. система в своей основе базируется на использовании докера, для иллюстрирования корректной работы в первую очередь следует пояснить скрипты запуска и показать список запущенных контейнеров.



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e49e609b203f	prom/prometheus	"/bin/prometheus --c..."	39 hours ago	Up 39 hours		prom
217a519bbc67	quay.io/prometheus/node-exporter:latest	"/bin/node_exporter -"	39 hours ago	Up 39 hours		vigorous_bouman
0d59e6bc2140	grafana/grafana	"/run.sh"	39 hours ago	Up 39 hours		grafana

Рисунок 17 – Скрин необходимых запущенных контейнеров

Т.к. мы тестируем сетевое приложение, генератор нагрузки будет запускаться на отдельном устройстве.

Эти три ключевых контейнера обеспечивают работоспособность нашей системы.

Контейнер prom/prometheus отвечает за:

- сбор метрик – сервис собирает из с различных источников – в нашем случае с приложений, запущенных в своих контейнерах. В дальнейшем эти метрики будет использовать уже Grafana
- хранение данных – Prometheus хранит собранные метрики в своей временной базе данных
- создание возможности обработки данных при помощи специального языка для запроса и анализа метрик – PromQL

Контейнер quay.io/prometheus/node-exporter отвечает за запуск важнейшего инструмента мониторинга – Node-exporter. Он предоставляет

метрики об использовании как системных ресурсов, так и производительности операционной системы Linux. Он устанавливается на каждом хосте, который мониторится Prometheus.

Последний из трех, но такой же важный, как и остальные, контейнер – grafana/grafana. Этот популярный в индустрии инструмент визуализации и анализа данных используется нами в целях визуального мониторинга производительности сложных систем. В нашей системе он подключается к Prometheus как к источнику данных, после чего необходимые данные передаются на гибкую систему дашбордов - наборов графиков и других средств визуализации данных.

Запуск Prometheus происходит по команде

```
sudo docker run -d --name prom --network="host" prom/prometheus
```

Докеру передаются параметры:

1. -d : “detached” режим – поток вывода сервера не направлен в консоль, он работает отдельно от нее
2. --network="host" : сети контейнера будут доступны все порты устройства

Далее для корректной работы сервиса ему нужно передать следующий файл настроек:

```
global:
  scrape_interval: 1m
  evaluation_interval: 5s
scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: [ 'localhost:9100' ]
  - job_name: 'server'
    static_configs:
      - targets: [ 'localhost:9200' ]
```

В нем указывается, что данные, анализируемые Prometheus, будут собираться относительно периода 1 минуты, максимальная задержка считывания параметра – 5 секунд.

Также настраиваются источники данных – данные от Node-exporter читаются с порта 9100, а работа сервера ожидается на порте 9200.

Для того, чтобы запущенный контейнер учел параметры из этого файла, необходимо скопировать его в контейнер и перезапустить его:

```
sudo docker cp prometheus.yml prom:/etc/prometheus  
sudo docker restart prom
```

Докеру передаются параметры для запуска Node-exporter:

```
sudo docker run -d --name prom --network="host" prom/node-exporter
```

Они аналогичны предыдущему запуску. После запуска все необходимые системные данные будут передаваться уже запущенному Prometheus.

Таким же образом запускаем Grafana:

```
sudo docker run -d --name prom --network="host" grafana/grafana
```

Результатом запуска всех трех сервисов получим возможность просматривать по ip сервера данные с порта 9090 для Prometheus и с порта 3000 для Grafana.

Далее следует собрать на двух различных устройствах образы серверов и тестировщика соответственно, чтобы потом на них запустить необходимые контейнеры.

Dockerfile для многопоточного и асинхронного серверов будет отличаться лишь набором файлов для сборки, поэтому проиллюстрируем лишь тот, что отвечает за многопоточный вариант.

```

# Создать образ на основе базового слоя gcc (там будет ОС и сам
компилятор).

# 11.3 — используемая версия gcc.

# Не просто создаём образ, но даём ему имя build
FROM gcc:11.3 as build

# Выполнить установку зависимостей внутри контейнера.
RUN apt update && \
    apt install -y \
        python3-pip \
        cmake \
    && \
    pip3 install conan==1.59

# Копируем conanfile.txt в контейнер и запускаем conan install
COPY conanfile.txt /app/

RUN mkdir /app/build && cd /app/build && \
    conan install .. -s compiler.libcxx=libstdc++11 -s build_type=Debug
--build=missing

# Только после этого копируем остальные исходники
COPY    server.cpp \
        logger.h logger.cpp \
        server_interface.h server_interface.cpp \
        server_routines.h server_routines.cpp \
        CMakeLists.txt \
    /app/

RUN cd /app/build && \
    cmake .. -DCMAKE_BUILD_TYPE=Debug && \
    cmake --build .

# Второй контейнер в том же докерфайле
FROM ubuntu:22.04 as run

```



```
# # Создадим пользователя usr
# RUN groupadd -r usr && useradd -r -g usr usr
# USER usr
# Скопируем приложение со сборочного контейнера в директорию
/app.
COPY --from=build /app/build/bin/server /app/
# Запускаем сервер
ENTRYPOINT ["/app/server"]
```

После сборки образа, необходимо запустить контейнер со следующими параметрами:

```
sudo docker build -t server .
sudo docker run --rm -p 5000:5000 server
```

Параметры запуска:

- --rm : контейнер удаляется после остановки работы
- -p 5000:5000 : 5000 порт контейнера соединяется с портом 5000 на устройстве

Dockerfile для сборки образа тестировщика приведен частично, т.к его build часть отличается от предыдущего примера лишь файлами сборки:

```
# Второй контейнер в том же докерфайле
FROM ubuntu:22.04 as run
# Создадим пользователя usr
RUN groupadd -r usr && useradd -r -g usr usr
USER usr

# Скопируем приложение со сборочного контейнера в директорию
/app.
```

```
COPY --from=build /app/build/bin/tester /app/  
# Запускаем тестировщик  
ENTRYPOINT ["/app/tester", "10"]
```

Аналогично собираем и запускаем тестировщика:

```
sudo docker build -t tester .  
sudo docker run ---rm --network=host tester | python3 script.py
```

Запуск тестировщика отличается тем, что он идет не в режиме detached и в том, что вывод программы будет передаваться в запущенный python скрипт для отправки логов на Prometheus.

В нем будут созданы 4 метрики, которые после будут обработаны:

количество правильных и неправильных переданных JSON, количество ошибок чтения (будут индикаторами окончания сессии по причине ошибки соединения) и задержка ответа на запрос.

Далее будут приведены части скрипта, отвечающие за запуск сервера, передающего данные логов, задание метрик и передачу метрики

```
import prometheus_client as prom  
...  
prom.start_http_server(9200)
```

```
# Создаём метрики  
good_lines = prom.Counter('webexporter_good_lines', 'Good JSON  
records')  
wrong_lines = prom.Counter('webexporter_wrong_lines', 'Wrong JSON  
records')  
read_errors = prom.Counter('client_read_errors', 'Client read errors')  
response_time = prom.Histogram('webserver_request_duration',  
'Response time', labelnames=['client_request_e', 'server_response_e'],
```

```
buckets=(.001, .002, .005, .010, .020, .050, .100, .200, .500,
float("inf"))))
```

```
# Регистрируем запрос
if data["message"] == "Read first response from server" or
data["message"] == "Wrote first response":
    total_time_seconds = data["data"]["response time"] / 1000
    response_time.labels(client_request_e =
data["data"]["response"]["client_request_e"],
server_response_e =
data["data"]["response"]["server_response_e"]).observe(total_time_sec
onds)
```

Остается лишь подключить к Grafana источник данных Prometheus и приступить к анализу получаемых данных при ее помощи.

3.2 Анализ результатов

В этом разделе мы проанализируем такие показатели, как использование CPU в процентах, использование RAM в процентах и задержку ответа.

Метрики использования CPU и RAM - ключевые при измерении производительности сервера в силу того, что они влияют на скорость работы и доступность ресурсов для обработки запросов, посылаемых клиентом на сторону сервера. Эти метрики помогают определить, насколько эффективно он использует свои ресурсы и готов ли обрабатывать большое количество запросов без ущерба для качества обслуживания пользователей. Поэтому, для поддержания высокой производительности сервера, важно следить за этими метриками и управлять ресурсами сервера соответственно.

Для тестирования производительности серверов будет использоваться виртуальная машина на ОС Linux Ubuntu со следующими параметрами:

- 4 процессорных ядра Intel Ice Lake
- 12 Гб RAM
- 60 Гб дискового пространства

Такой мощности вполне достаточно для стабильной работы сервера при достаточно высоких нагрузках. Дисковое пространство необходимо для хранения образов запускаемых контейнеров и дальнейшего хранения файлов чатов в уже работающих контейнерах.

С этого устройства при помощи node-exporter будем получать данные на Grafana. Данные использования CPU будут визуализированы классическими графиками по процентному соотношению от 0% до 100%.

Использование RAM лучше не иллюстрировать при помощи численного варианта – график будет показывать процентное значение используемой в данный момент памяти, исходя из ресурсов системы.

Стоит отдельно отметить, что задержка ответа будет измеряться на стороне приложения, генерирующего нагрузку, т.к. запросы в многопоточном сервере опираются на низкоуровневую логику Linux и невозможно будет корректно измерять задержку от запроса до ответа без существенного влияния на производительность: синхронизация, требующаяся для постоянного отслеживания приходящих запросов, вызовет захват ресурсов от их чтения, что конечно же будет отрицательно влиять на адекватность собираемых данных.

Для иллюстративности, мы рассмотрим три случая для каждой реализации сервера:

1. 2 клиента для иллюстрации минимальной нагрузки
2. 10 клиентов для того, чтобы проиллюстрировать как небольшой рост нагрузки на сервер будет влиять на изучаемые параметры

3. 100 клиентов на каждый сервер как максимальная измеряемая нагрузка
- Отдельно отметим, что 100 клиентов при стандартных условиях – весьма малое число, однако тестирующее приложение способно нагружать сервер многократно большим количеством запросов от каждого клиента: реальный пользователь никогда не достигнет такого количества отправляемых сообщений и переходов между чатами как минимум в силу того, что генератор нагрузки не пользуется графической частью, а обрабатывает данные напрямую.

3.2.1 Тестирование многопоточного сервера

2 клиента

Отобразим динамику расхода ресурсов:

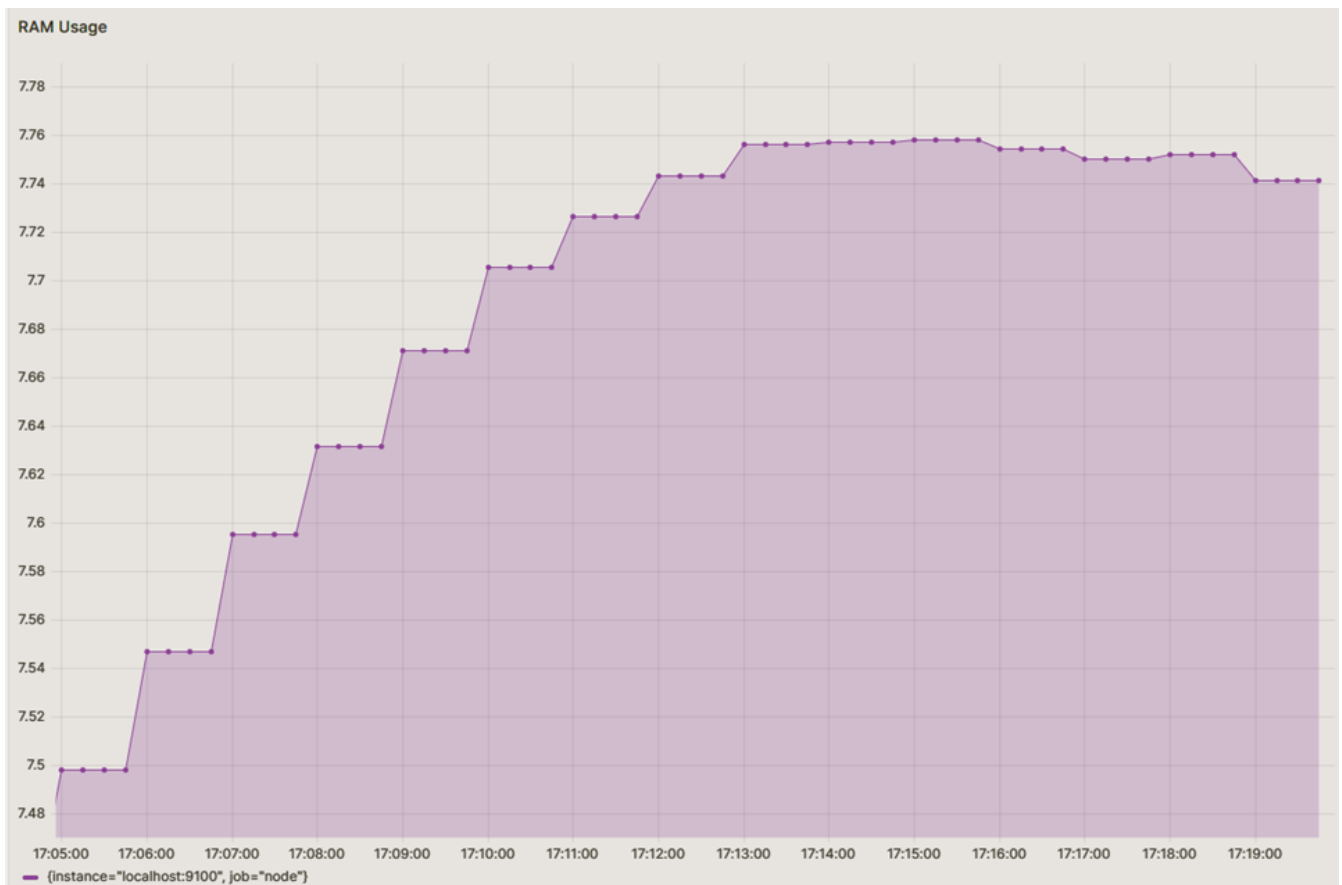


Рисунок 18 – График использования ОП на устройстве сервера при 2 клиентах для многопоточного сервера

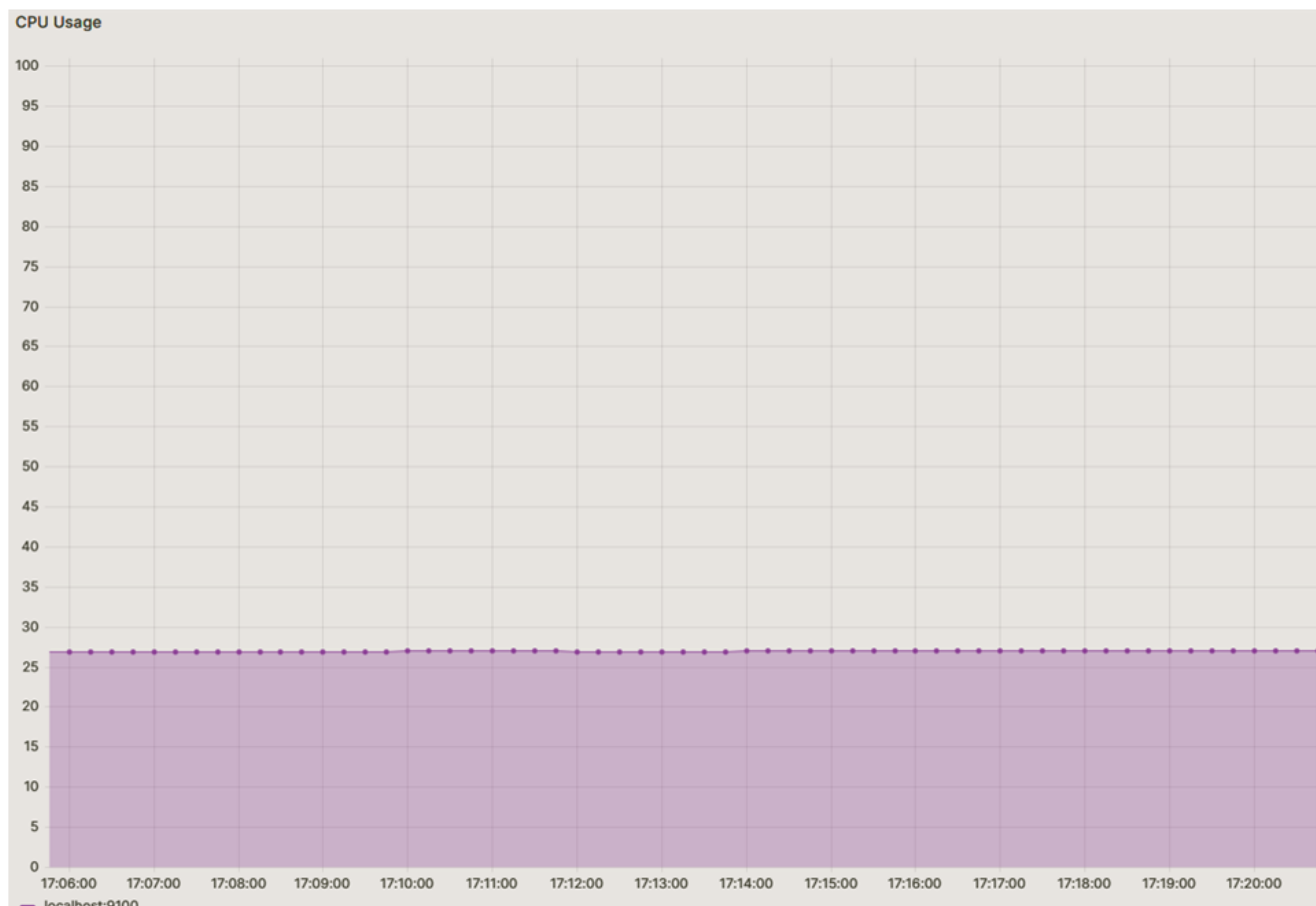


Рисунок 19 – График использования ресурсов процессора на устройстве сервера при 2 клиентах для многопоточного сервера

Как видно на графиках, захватив необходимое количество операционной памяти, приложение нашло баланс в ее использовании.

Колебания в использовании CPU минимальны, это объяснимо с учетом архитектуры сервера: выделяются четыре потока обработки, дополнительные формируются исключительно для использования чатов при необходимости. Т.к. количество клиентов минимально, то не стоит ожидать сильного использования ресурсов.

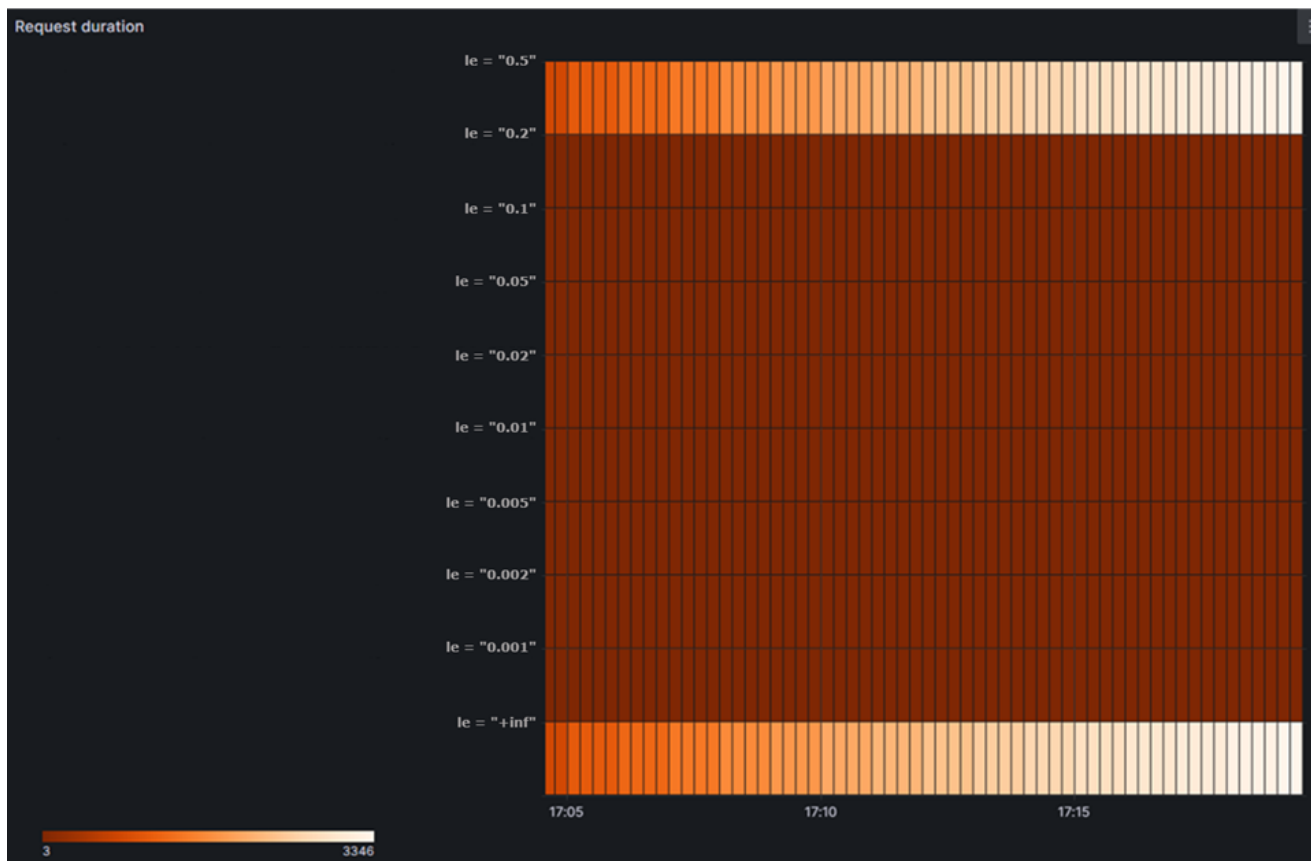


Рисунок 20 – Тепловая карта задержек ответов на запросы клиентов при 2 клиентах для многопоточного сервера

Задержка нарастает со временем использования сервера, т.к. ускоренные тестирующим приложением клиенты имеют возможность перескакивать из чата в чат, каждый раз получая все его содержимое. Отдельно стоит отметить, что распределение основных задержек между 0.5 и +inf объясняется, тем, что т.к. ожидание сообщений замораживает поток, выполняющий связанную задачу.

10 клиентов

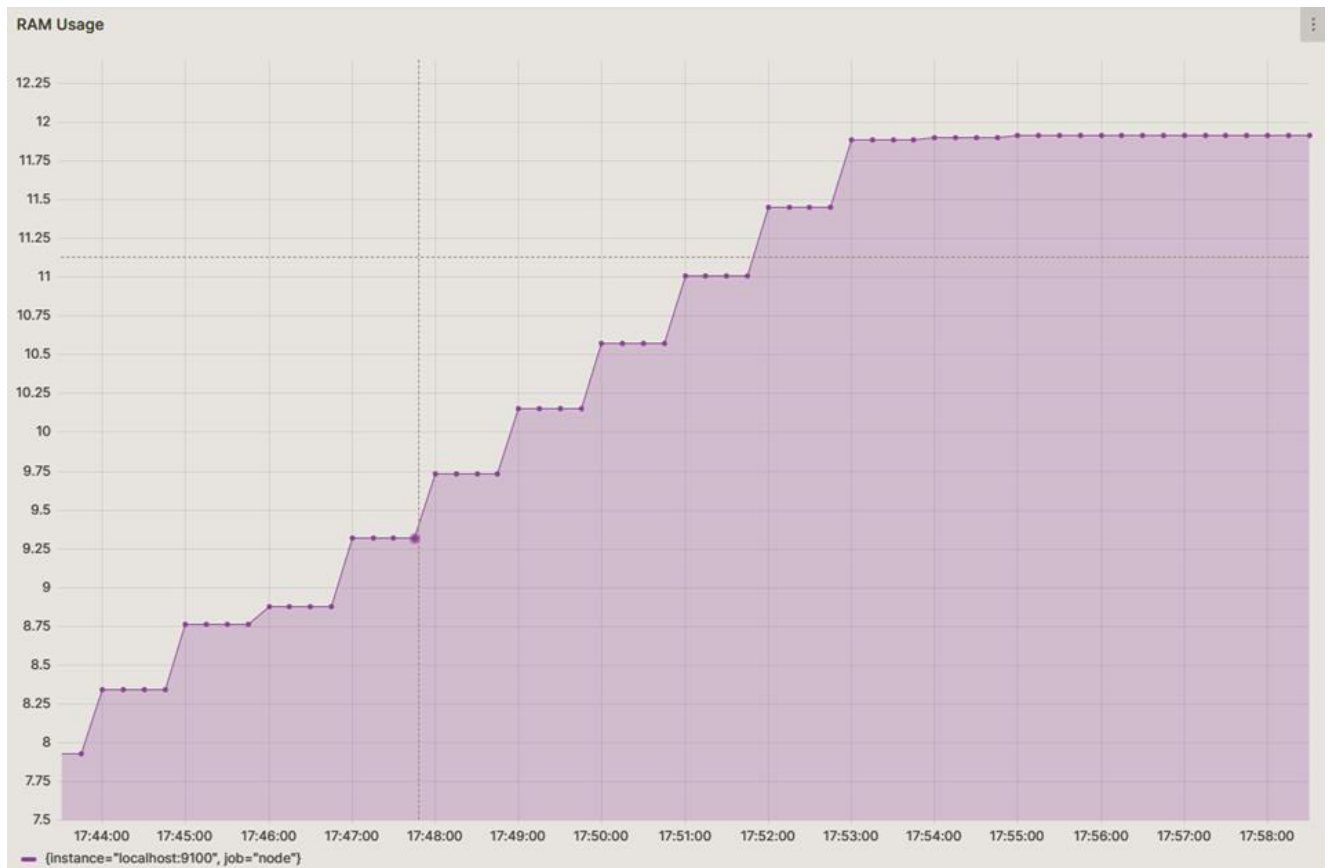


Рисунок 21 – График использования ОП на устройстве сервера при 10 клиентах для многопоточного сервера

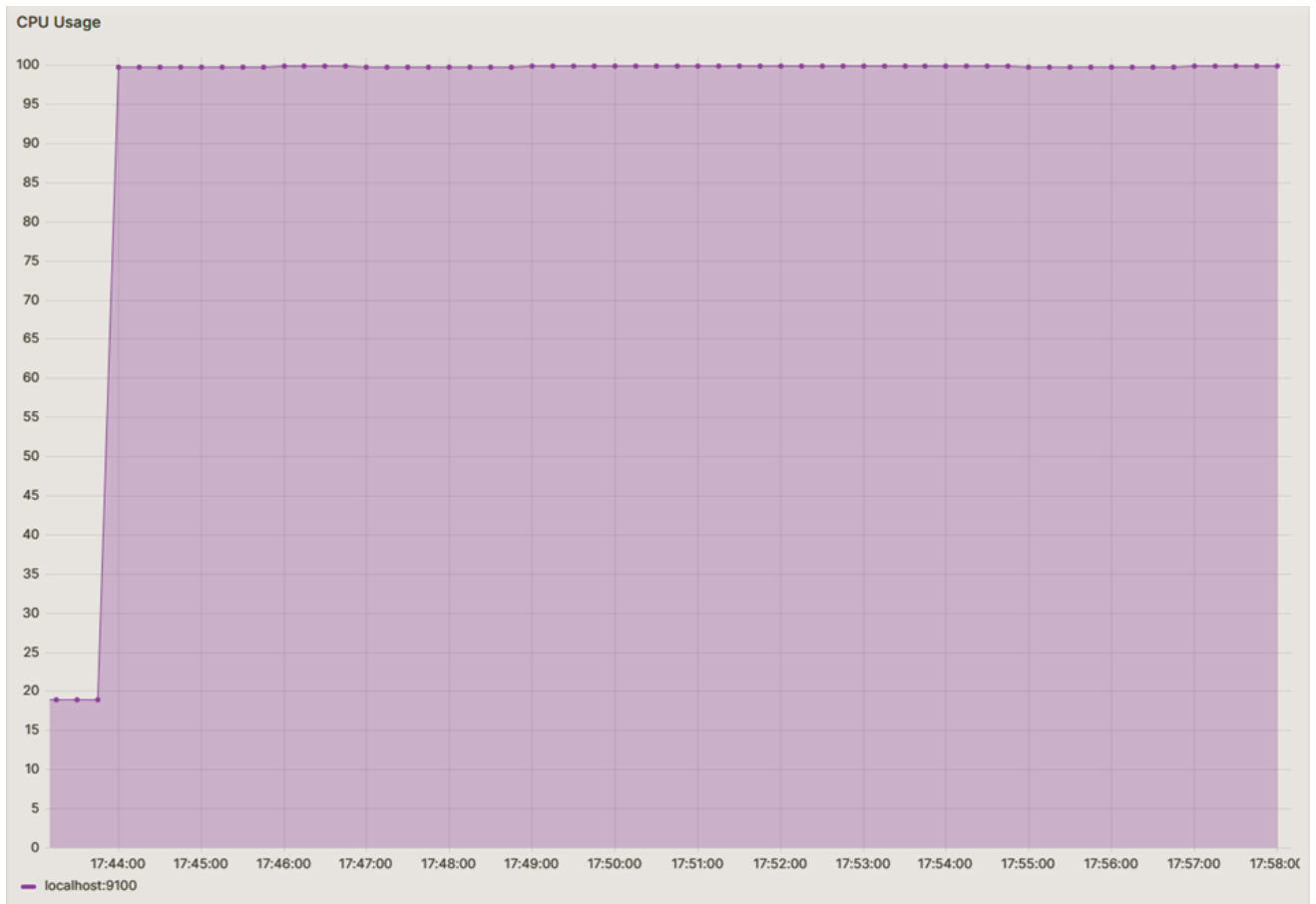


Рисунок 22 - График использования ресурсов процессора на устройстве сервера при 10 клиентах для многопоточного сервера

Новые чаты, опирающиеся на новые потоки, занимают все возможные ресурсы процессора, что, без всякого сомнения, крайне негативно отражается на работе сервера. Использование памяти остановилось в росте, набран необходимый для работы объем.

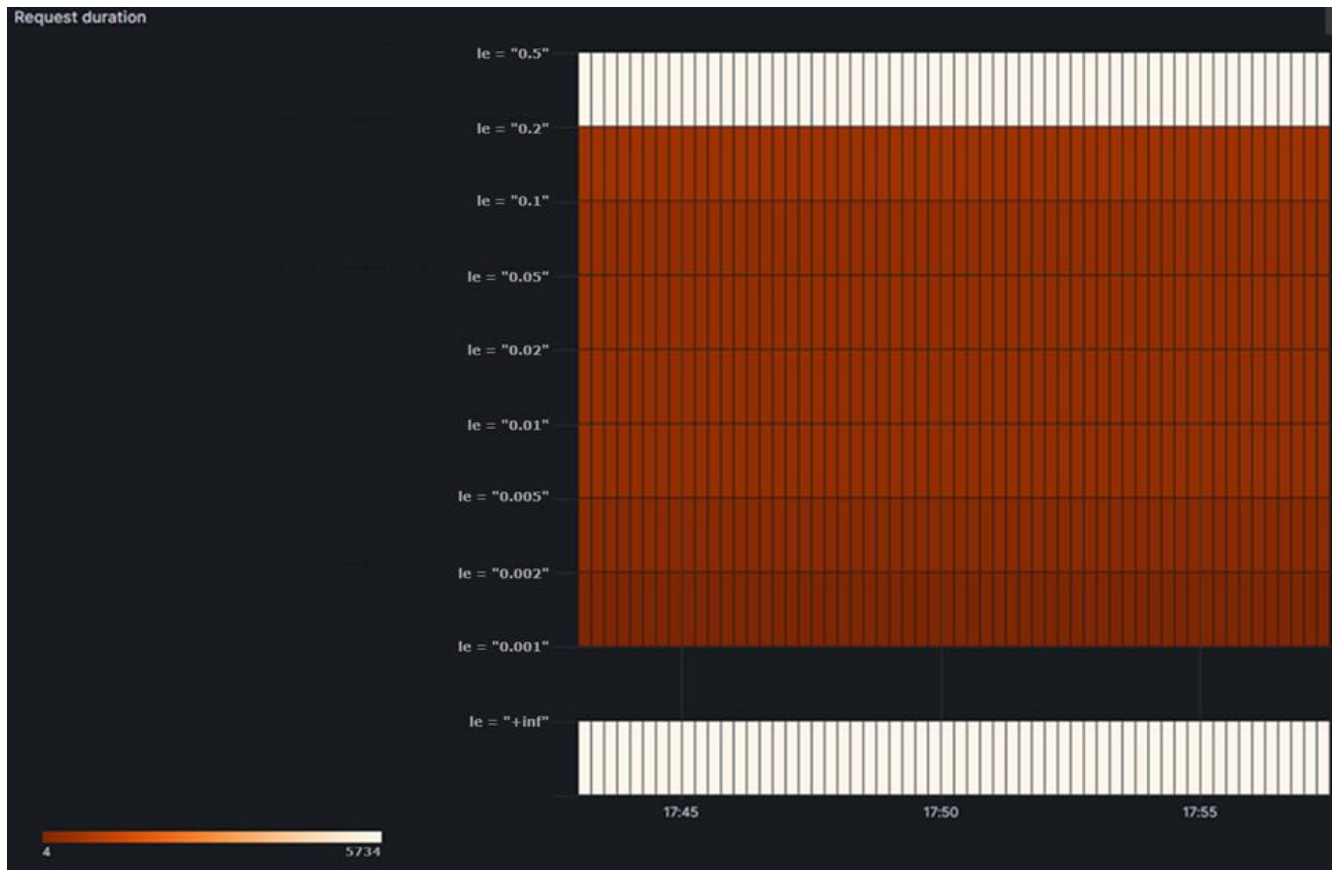


Рисунок 23 - Тепловая карта задержек ответов на запросы клиентов при 2 клиентах для многопоточного сервера

Здесь заметно, что при большем количестве клиентов резко растет задержка на ответ – снова дает знать о себе остановка потока в ожидании сообщения.

100 клиентов

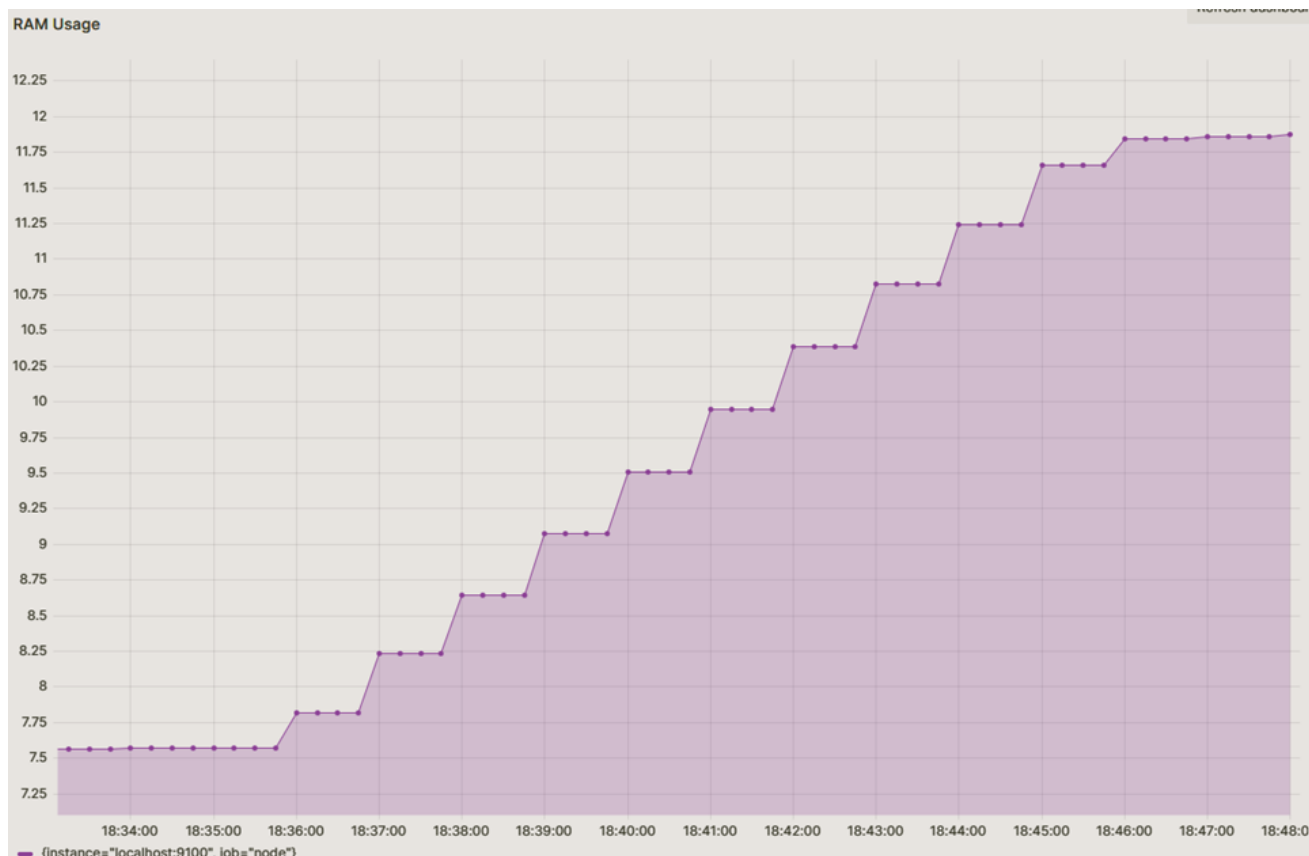


Рисунок 24 - График использования ОП на устройстве сервера при 100 клиентах для многопоточного сервера

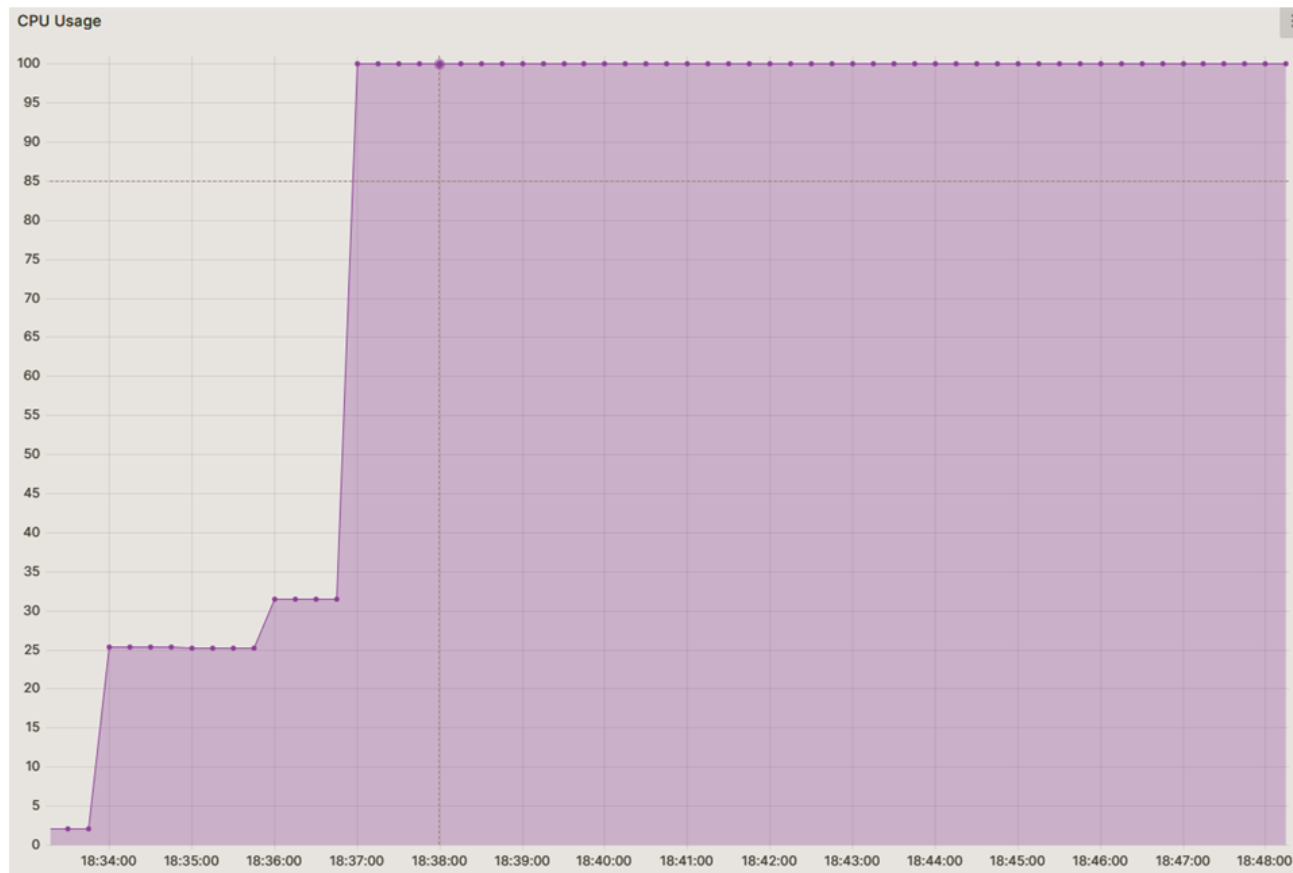


Рисунок 25 - График использования ресурсов процессора на устройстве сервера при 100 клиентах для многопоточного сервера

Как и в предыдущем тесте, захватив необходимое количество оперативной памяти, сервер уперся в потолок в ресурсах процессора и продолжил свою работу, снизив эффективность обработки.

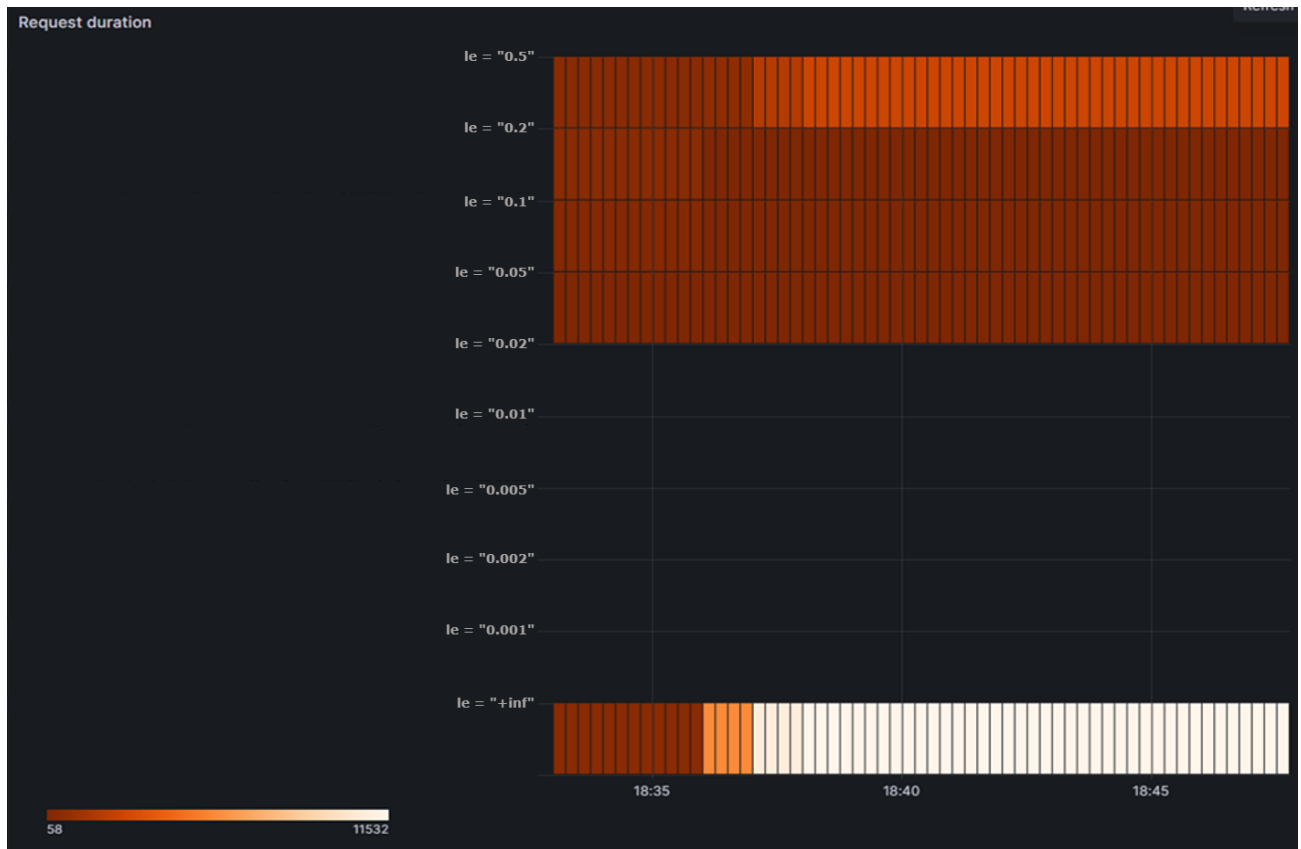


Рисунок 26 - Тепловая карта задержек ответов на запросы клиентов при 100 клиентах для многопоточного сервера

Приведенный график еще ярче показывает, что не асинхронный подход имеет яркие недостатки при работе с веб сервером. Доминирующее число ответов сервера занимали более полсекунды, что неприемлемо для приложения, рассчитанного на высокую нагрузку.

Итоги тестирования многопоточного сервера

Полученные данные о работе сервера, основанного на многопоточной конвейерной архитектуре, ярко иллюстрируют сложившуюся для него ситуацию:

- Сервер, базирующийся на выбранной в данном разделе логике, главным своим достоинством имеет малое использование оперативной памяти
- Опора исключительно на дополнительные потоки для каждой рутины сервера приводит к нерациональному расходу ресурсов процессора создаваемыми потоками. Ключевое влияние оказывают потоки чатов.
- Задержка, с которой сталкивается клиент, растет крайне быстро и в таком же темпе достигает неприемлемых значений

3.2.2 Тестирование многопоточно-асинхронного сервера

2 клиента

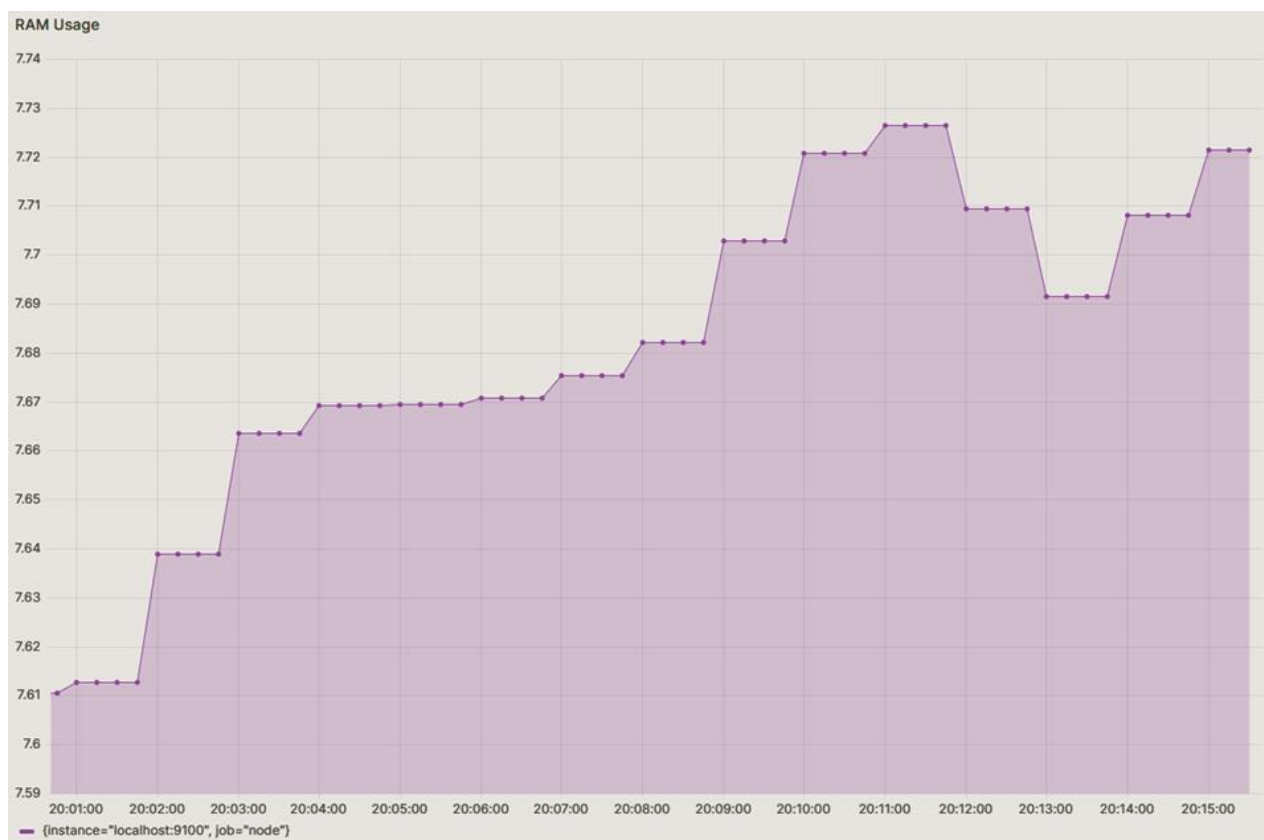


Рисунок 27 - График использования ОП на устройстве сервера при 2 клиентах для многопоточно-асинхронного сервера

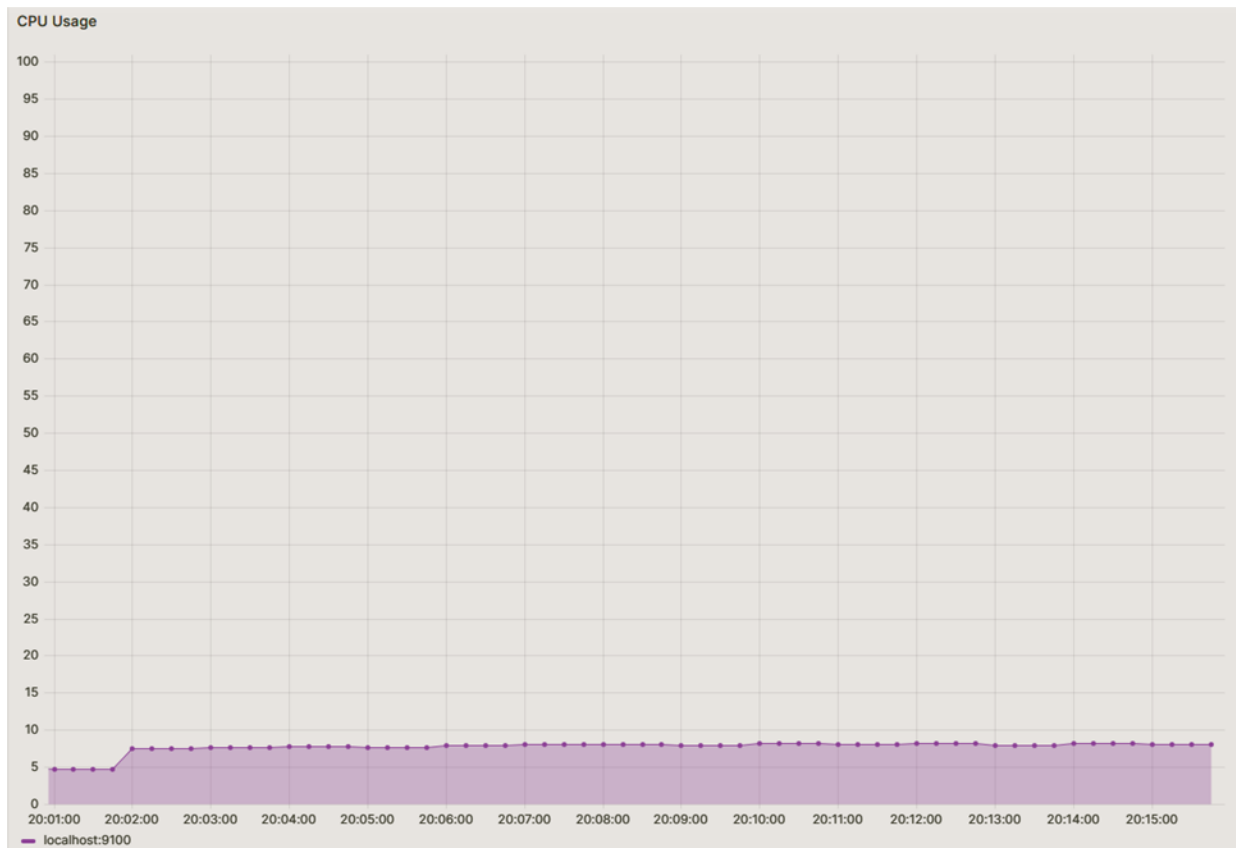


Рисунок 28 - График использования ресурсов процессора на устройстве сервера при 2 клиентах для многопоточно-асинхронного сервера

Скачки в объеме используемой памяти объясняются созданием и деструкцией промежуточных объектов корутин — приложение полностью опирается на ООП. В дальнейшем динамика сохранится.

Уже на этом этапе стоит отметить крайне рациональное использование ресурсов процессора. Эта особенность проявляется в силу использования пула потоков.

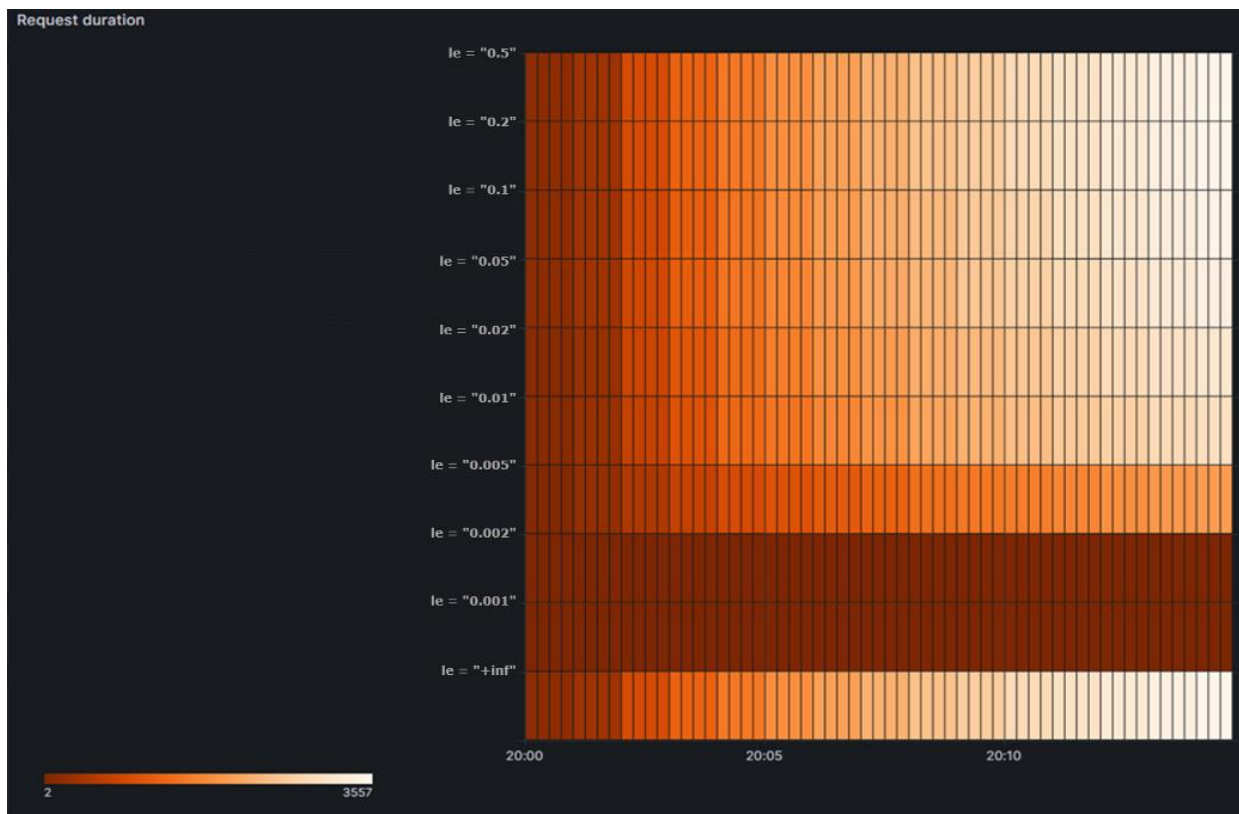


Рисунок 29 - Тепловая карта задержек ответов на запросы клиентов при 2 клиентах для многопоточно-асинхронного сервера

Задержки до получения ответа клиентами имеют куда более распределенный характер. Это вызвано тем, что клиентские запросы вызывают различные процедуры, которые происходят асинхронно, с минимизированным ожиданием друг друга, поэтому куда ярче заметно распределение задержек для запросов в разных контекстах работы сервера.

10 клиентов

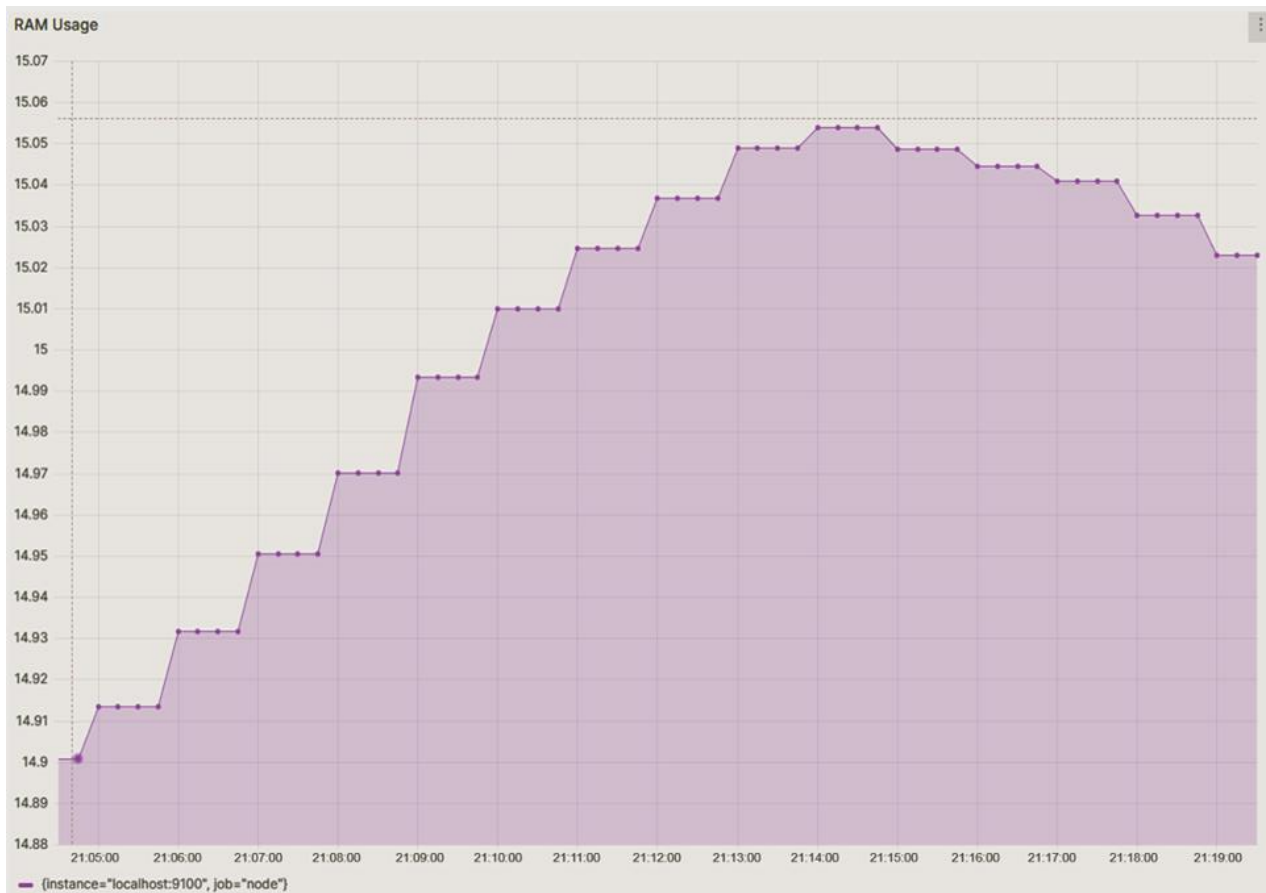


Рисунок 30 - График использования ОП на устройстве сервера при 10 клиентах для многопоточно-асинхронного сервера

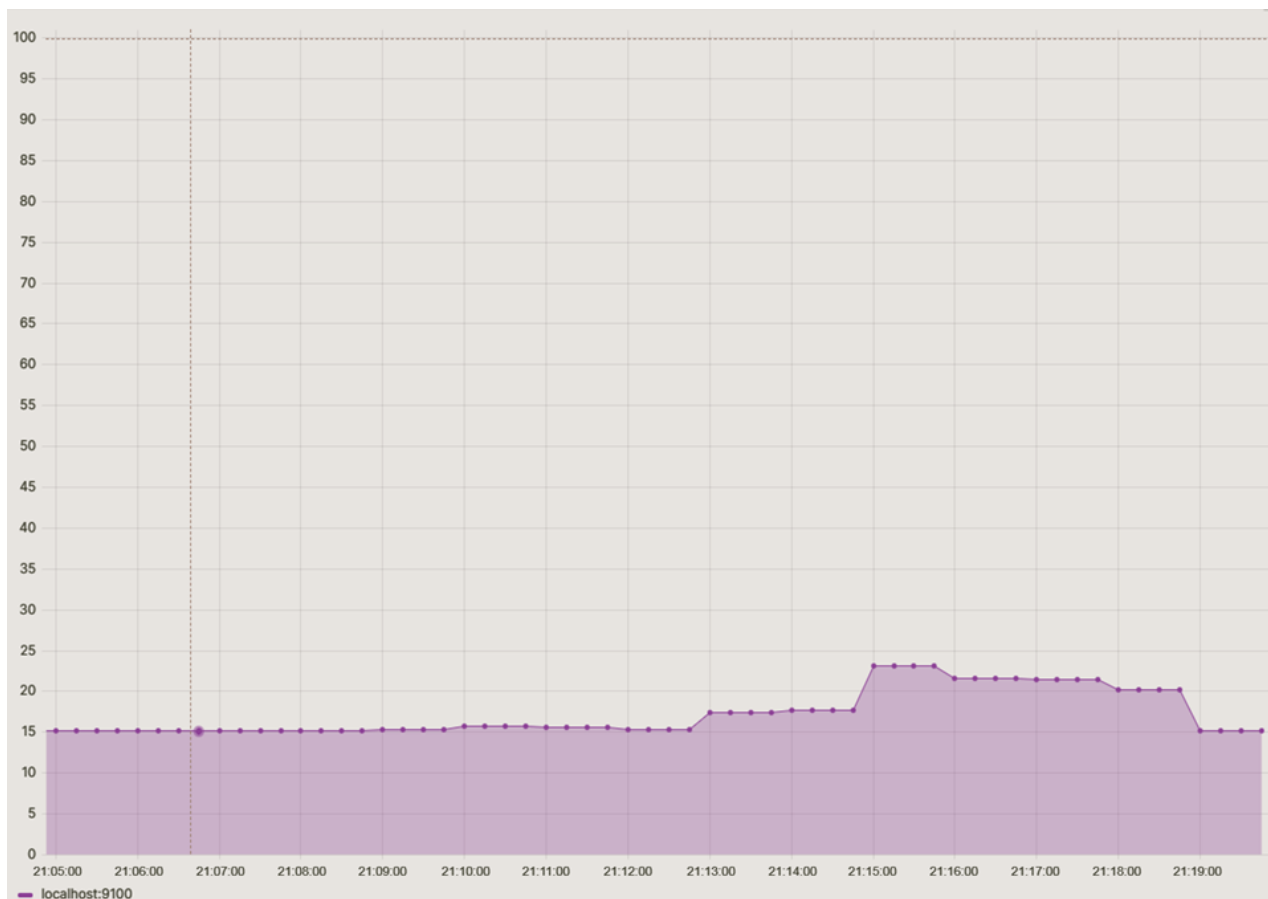


Рисунок 31 – График использования ресурсов процессора на устройстве сервера при 10 клиентах для многопоточно-асинхронного сервера

Динамика, отмеченная в предыдущем тесте, повторяется с учетом обновленных условий. Наблюдаются скачки в использовании ресурсов памяти, а возросшие затраты ресурсов процессора все еще можно назвать крайне эффективными в использовании.

Смена динамики возрастания используемой ОП на уменьшение могла возникнуть в силу постепенного избавления от временных объектов многочисленно вызванных корутин.

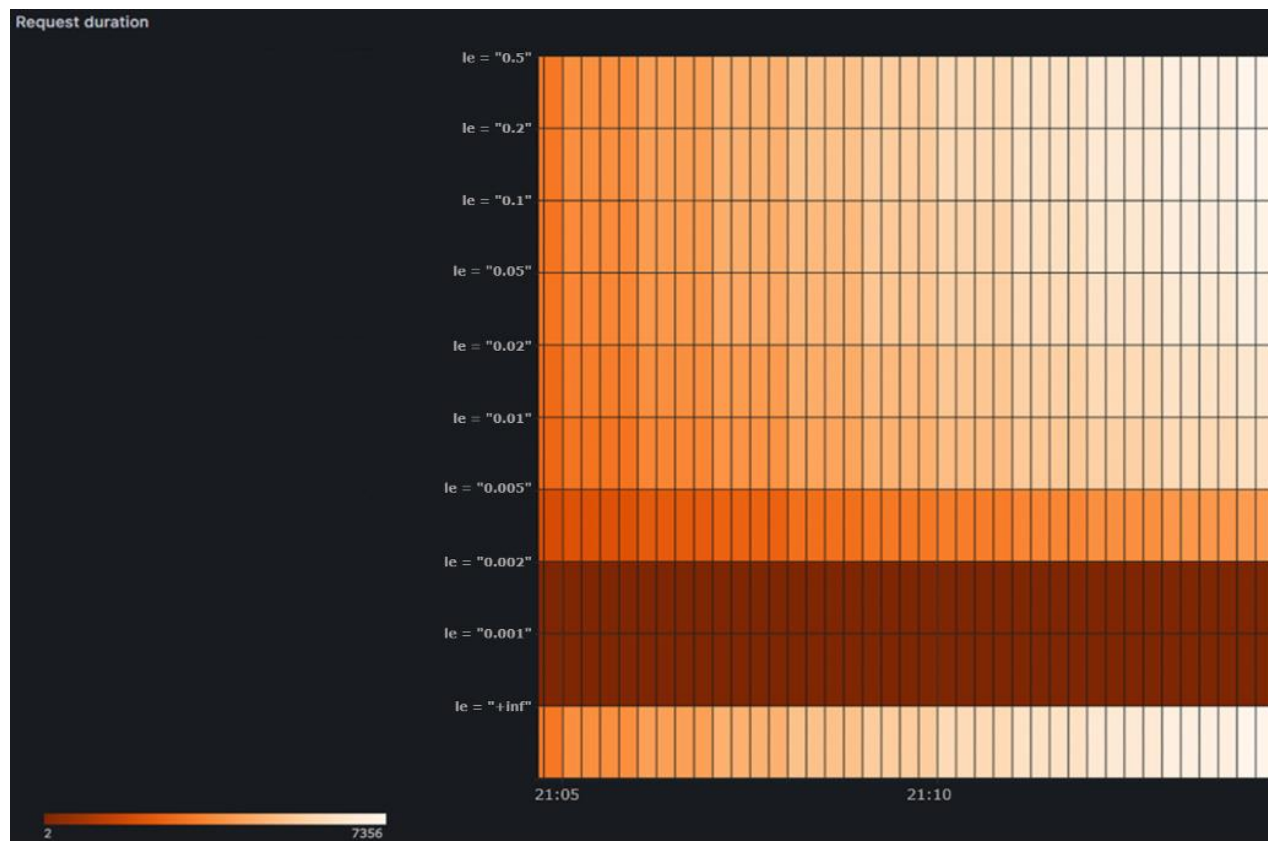


Рисунок 32 – Тепловая карта задержек ответов на запросы клиентов при 10 клиентах для многопоточно-асинхронного сервера

Тепловая карта показывает, что прирост задержек увеличил скорость, что объясняется возросшим количеством пользователей и возросшим вследствие количества сообщений в каждом чате.

Несколько ярче стало смещение в сторону более высоких задержек, однако они все еще остаются вполне адекватными.

100 клиентов

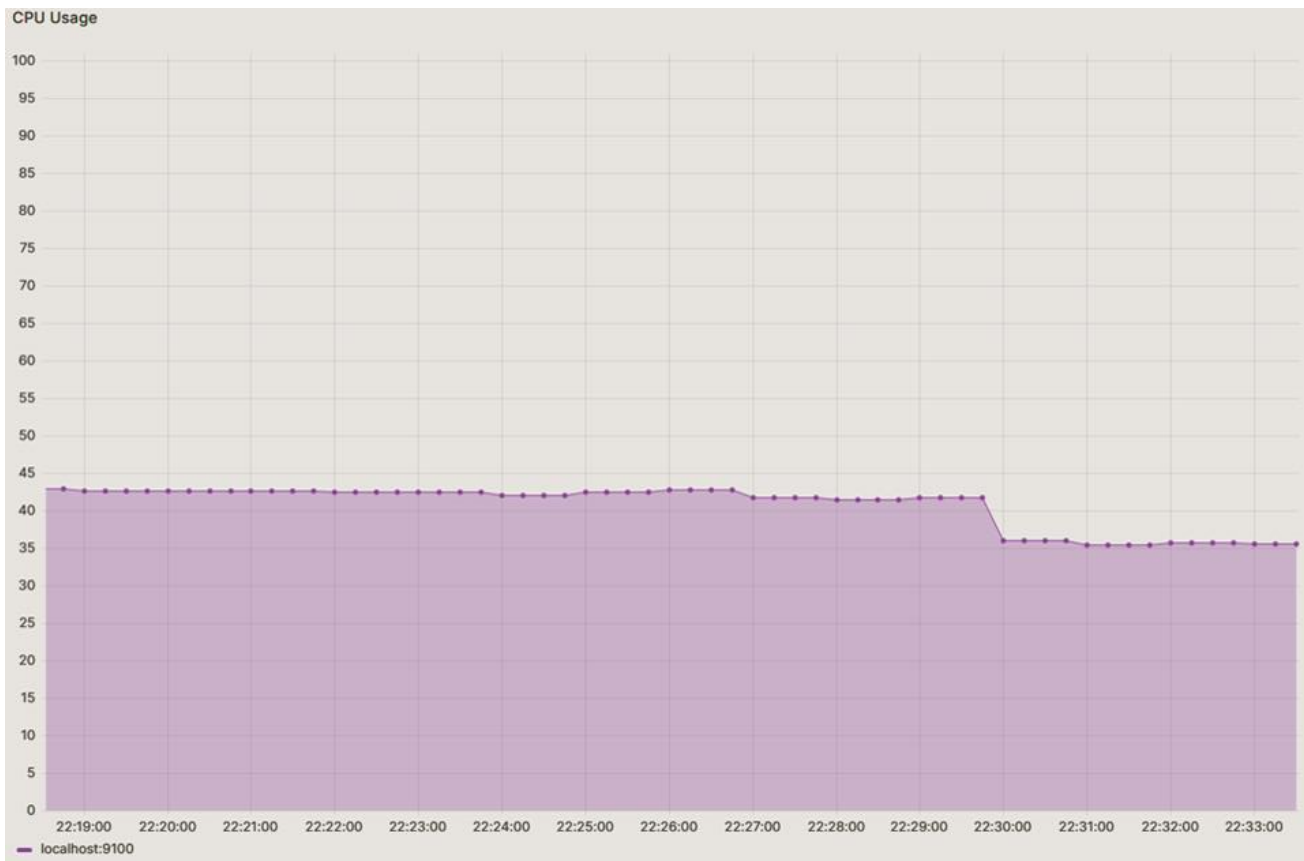


Рисунок 33 - График использования ОП на устройстве сервера при 100 клиентах для многопоточно-асинхронного сервера

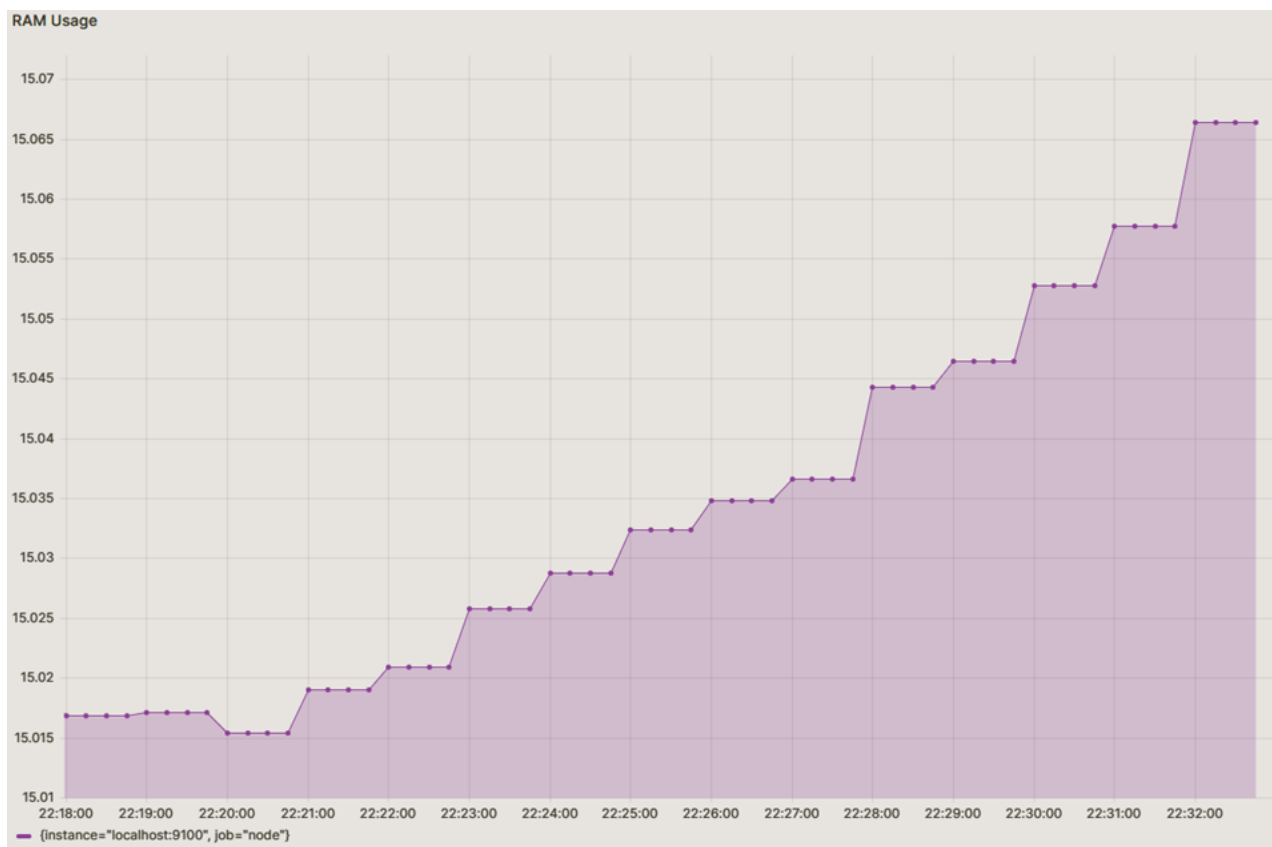


Рисунок 34 - График использования ресурсов процессора на устройстве сервера при 100 клиентах для многопоточно-асинхронного сервера

На самом сложном этапе тестирования подскочило использование памяти – множество запущенных асинхронно корутин скачками захватывали ресурсы памяти, найдя баланс лишь в самом конце периода измерения.

Объем памяти, используемый приложением, все еще можно считать весьма малым.

Ресурсы процессора, будучи безусловно более загруженными относительно предыдущих кейсов тестирования, все еще имеют запас для роста расходов.

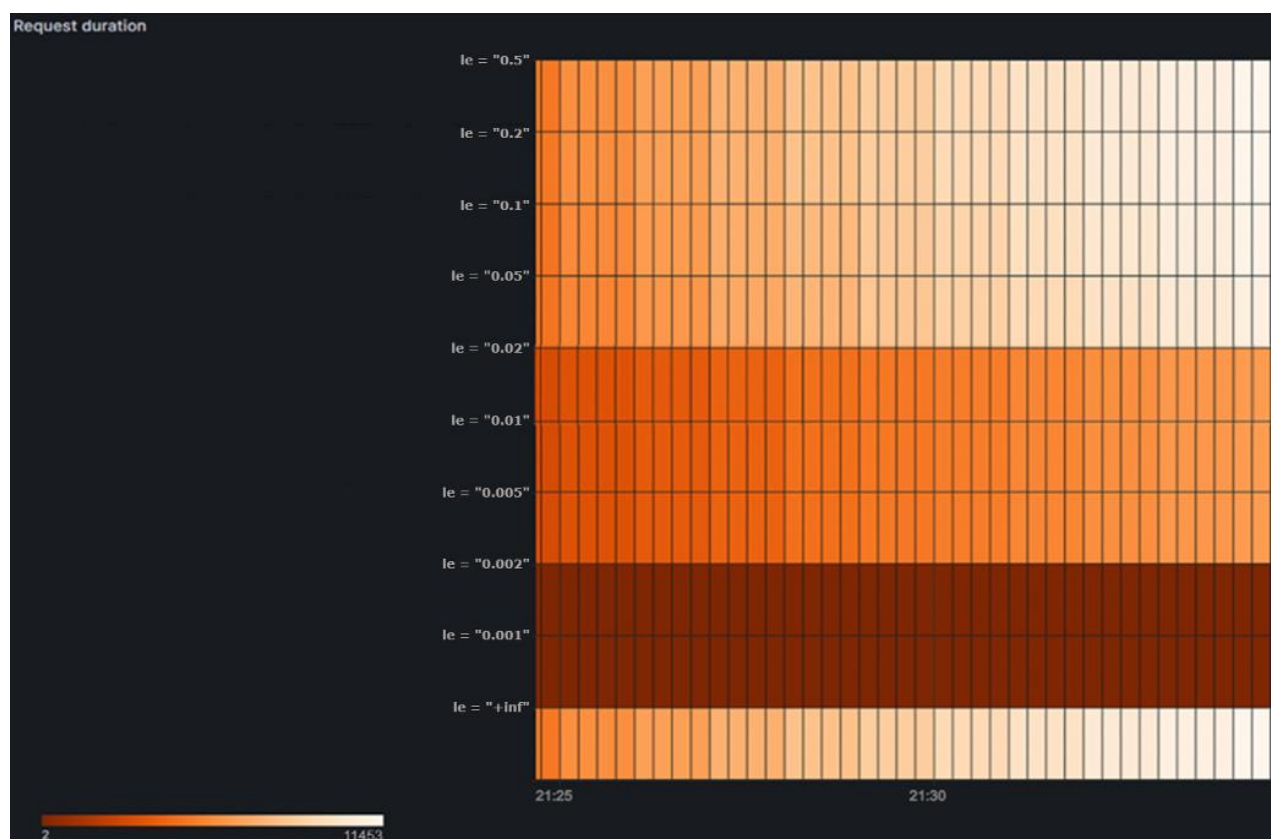


Рисунок 35 - Тепловая карта задержек ответов на запросы клиентов при 100 клиентах для многопоточно-асинхронного сервера

В данном случае тестирования максимальное выбранное количество пользователей хорошо проявило способности сервера выдерживать возрастающие нагрузки. Общая доля запросов, попадающих во временной

отрезок, необходимый для ответа, больший полсекунды, все еще не является даже близко доминирующей, хотя и есть проседание доли малых задержек.

Итоги тестирования многопоточно-асинхронного сервера

Анализ полученных в результате тестов данных показал следующие моменты:

- ООП, используемое в данной реализации сервера в куда более полной форме, имеет яркое влияние на его работу: хотя использование памяти и возросло, приспособление методов классов к асинхронности через превращение их в корутины дало большой скачок в производительности.
- Задержки, с которыми сталкивались клиенты, были вполне удовлетворительными
- Использование асинхронности предотвратило создание гигантских очередей, уничтожающих шансы на эффективную работу системы

3.2.3 Общие итоги тестирования, сравнение результатов

В сравнении столь отличных реализаций серверов стоит структурировать параметры, относительно которых оно будет происходить.

Использование ресурсов памяти:

Многопоточный сервер имеет более скромные потребности в оперативной памяти. К тому же характер ее использования не имеет склонности к резким перепадам, в отличие от многопоточно-асинхронной архитектуры, чья полная опора на ООП создает необходимость в создании временных объектов классов, которые являются основой логики работы приложения.

Использование ресурсов процессора:

Явное преимущество в эффективности использования вычислительных ресурсов процессора достигается многопоточно-асинхронной реализацией за счет нескольких ключевых факторов.

1. Использование пула потоков вместо динамического создания при необходимости ограничивает рост используемых ресурсов процессора.
2. Опора на асинхронный движок BOOST позволяет создавать отдельные очереди для синхронизации, при этом сохраняя возможность автоматического асинхронного многопоточного выполнения задач.

При готовности и отсутствии более раннего претендента на выделенные ресурсы задача захватит контроль над вычислительными ресурсами потока из зафиксированного пула. Это дает возможность реализации максимально эффективных процессов чтения и обработки, минимизирующих ожидание.

3. Более конкретные и упрощенные сценарии синхронизации в ООП дают возможность лучше оптимизировать процессы системы

Задержки:

В первую очередь, в силу ранее отмеченных преимуществ, многопоточно-асинхронная реализация серверной логики является очевидным победителем по этому параметру.

Однако стоит подметить, что задержки в своем многообразии имеют разные характеры распределения.

Т.к. многопоточный сервер сталкивается с формированием очередей на чтение с полной заморозкой потоков, то его периоды задержки ярко смещены в сторону длинных отрезков времени и чем выше нагрузка, тем все большая доля попадает к запросам, ожидавшимся более чем 0.5 секунды.

В свою очередь, многопоточно-асинхронный сервер имеет куда более красочный и интересный характер распределения периодов ожидания, который заметно проявляется на тепловой карте задержек в тестах. Т.к. фактор заморозки потоков минимизирован до того, что его не обязательно учитывать, куда большее влияние имеет уже состояние системы, на которой основан

сервер. За счет этого представлены куда более разнообразные варианты распределений, которые, к тому же, будут незначительно отличаться в зависимости от того, как будут поданы задачи на сервер.

Финализируя проведенный анализ, более современный подход многопоточного сервера обеспечил ему победу в сравнении итоговой производительности, однако внедрение асинхронности потенциально будет тем, что позволит многопоточному серверу, базирующемуся на конвейерной логике, догнать своего оппонента.

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе приведены и проиллюстрированы этапы анализа исходных данных и проектирования, разработки программного обеспечения, тестирования двух архитектур серверов и работоспособности тестирующей системы в целом и анализа результатов тестирования в форме графического отображения.

Глава анализа исходных данных и проектирования структурировал и описал этапы конструирования архитектур серверов, особенностей получаемых и отправляемых серверами данных.

Глава разработки реализаций сервера и клиента показала основную часть работы – непосредственное создание программного продукта в форме двух серверов и клиентского приложения. Также было разработано приложение для имитации нагрузки от выбранного числа клиентов, которое будет использовано для тестирования в дальнейшем.

Глава тестирования разделилась на логические три части – тестирование работоспособности системы, нагрузочное тестирование серверов и анализ результатов. Первая часть заключалась в настройке и запуске докером основных контейнеров, отвечающих за обработку передаваемых данных, которые будут использованы для анализа. Вторая отвечает за формирование первичных выводов на основе графической визуализации полученных во время тестирования данных. Последняя – за сравнительный анализ данных, полученных на всех этапах тестирования двух реализаций серверов.

В результате выполненной работы была создана программная платформа для сравнения производительности и анализа положительных и отрицательных характеристик многопоточной и многопоточно-асинхронной серверных архитектур относительно друг друга.

Используемые программные платформы и средства разработки подходят выбранным операционной системе и используемым в разработке языкам программирования.

Разработка серверов на двух различных архитектурах показала недостатки и преимущества подходов со стороны разработки:

Для многопоточного сервера:

Достоинства: простота проектирования, малое использование ОП, относительно простая отладка

Недостатки: низкая читаемость низкоуровневого кода, высокий расход ресурсов процессора, заморозка потоков при чтении запроса

Возможности для улучшения: внедрение асинхронности для каждого из подсервисов, использование корутин, внедрение кеширования сообщений на стороне клиента

Для многопоточно-асинхронного сервера:

Достоинства: хорошая читаемость опирающегося на ООП кода, низкий расход ресурсов процессора, простота настройки синхронизации за счет использования современной библиотеки BOOST

Недостатки: сложность отладки основанного в большой мере на библиотеке приложения, высокое использование ОП

Возможности для улучшения: внедрение кеширования на стороне клиента, оптимизация исключения клиентов из списка рассылки для чатов

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация библиотеки BOOST: <https://www.boost.org/doc/>
2. Документация QT: <https://doc.qt.io/>
3. John Torjo; Wisnu Anggoro Boost.Asio C++ Network Programming - Second Edition - Packt Publishing, 2015 – 200 с.
4. Разработка приложений на C++ с использованием Boost. Рецепты, упрощающие разработку вашего приложения / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 346 с.: ил.
5. Расширенное описание библиотек, входящих в BOOST: <https://theboostcpplibraries.com/>
6. Шлее М. Qt 5.10. Профессиональное программирование на C++. -СПб.: БХВ-Петербург, 2018. -1072 с.: ил.
7. Моуэт Э. Использование Docker / пер. с англ. А.В. Снастина; науч. Ред. А.А. Маркелов. – М.: ДМК Пресс, 2017. – 354 с.: ил.

ПРИЛОЖЕНИЕ А. ФАЙЛ MAIN.CPP ГЕНЕРАТОРА НАГРУЗКИ

```
#include <iostream>
#include <coroutine>
#include <memory>
#include <thread>
#include <sstream>
#include <iomanip>
#include <random>
#include <vector>
#include <chrono>

#include <boost/array.hpp>
#include <boost/scoped_ptr.hpp>

#include <boost/asio/read.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/use_awaitable.hpp>
#include <boost/asio/awaitable.hpp>
#include <boost/asio/experimental/as_tuple.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <boost/asio/signal_set.hpp>
#include <boost/asio/io_context.hpp>
#include <boost/asio/detached.hpp>
#include <boost/asio/co_spawn.hpp>
#include <boost/asio/connect.hpp>

// #include <boost/json.hpp>

#include "logger.h"

namespace net = boost::asio;
using tcp = net::ip::tcp;
```

```

namespace sys = boost::system;

using namespace std::literals;

enum client_request_e {
    c_set_name = 0,
    c_create_chat,
    c_connect_chat,
    c_send_message,
    c_leave_chat,
    c_disconnect,
    c_receive_message,
    c_wrong_request,
    c_get_chats
};

enum server_response_e{
    s_success = 0,
    s_failure,
};

struct client_data_t{
    client_request_e request;
    std::string message_text;
};

struct server_data_t{
    client_request_e request;
    server_response_e response;
    std::string message_text;
};

```

```

//get random element from container
template<typename Iter, typename RandomGenerator>
Iter select_randomly(Iter start, Iter end, RandomGenerator& g) {
    std::uniform_int_distribution<> dis(0, std::distance(start,
end) - 1);
    std::advance(start, dis(g));
    return start;
}

template<typename Iter>
Iter select_randomly(Iter start, Iter end) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    return select_randomly(start, end, gen);
}

// Запускает функцию fn на n потоках, включая текущий
template <typename Fn>
void RunThreads(unsigned n, const Fn& fn) {
    n = std::max(1u, n);
    std::vector<std::jthread> workers;
    workers.reserve(n - 1);
    // Запускаем n-1 рабочих потоков, выполняющих функцию fn
    while (--n) {
        workers.emplace_back(fn);
    }
    fn();
}

std::string StringifyRequest(client_request_e r){
    switch(r){
        case 0: return "c_set_name"s;
        case 1: return "c_create_chat"s;
    }
}

```

```

        case 2: return "c_connect_chat"s;
        case 3: return "c_send_message"s;
        case 4: return "c_leave_chat"s;
        case 5: return "c_disconnect"s;
        case 6: return "c_receive_message"s;
        case 7: return "c_wrong_request"s;
        case 8: return "c_get_chats"s;
        default: return "wrong input"s;
    }
}

std::string StringifyResponse(server_response_e r){
    switch(r){
        case 0: return "s_success"s;
        case 1: return "s_failure"s;
        default: return "wrong input"s;
    }
}

net::awaitable<void> Write(std::shared_ptr<tcp::socket> socket,
client_data_t&& request){
    size_t message_length = request.message_text.size();
    boost::scoped_ptr<client_data_t> safe_request(new
client_data_t(request));

    // std::cout << "Sending message: {request: " <<
response.request
    // << ", response: " << response.response << ", message: \"\"
<< response.message_text << "\"}\"" << std::endl;

    auto [ec, bytes] = co_await net::async_write(
        *socket,
        boost::array<boost::asio::mutable_buffer, 4>({

```

```

        net::buffer((void*)&safe_request->request,
sizeof(client_request_e)),
        net::buffer((void*)&message_length,
sizeof(size_t)),
        net::buffer(safe_request->message_text)
    )),
    // boost::asio::transfer_all(), // maybe needed
    net::experimental::as_tuple(net::use_awaitable)
);

if(ec) {
    //log here
    boost::json::value data = {
        {"error", ec.what()}
    };
    logger::Logger::GetInstance().Log("Client      write
error"sv, data);
    throw ec;
}

    boost::json::value data = {
        {"request",
            {
                {"client_request_e",
StringifyRequest(request.request)},
                {"message", request.message_text}
            }
        }
    };
    logger::Logger::GetInstance().Log("Wrote      request      to
server"sv, data);
}

```

```

    net::awaitable<server_data_t> Read(std::shared_ptr<tcp::socket>
socket){
    server_data_t response;
    size_t message_length;

    auto [ec, bytes] = co_await
        boost::asio::async_read(
            *socket,
            boost::array<boost::asio::mutable_buffer, 3>({
                net::buffer((void*)&response.request,
sizeof(client_request_e)),
                net::buffer((void*)&response.response,
sizeof(server_response_e)),
                net::buffer((void*)&message_length,
sizeof(size_t)),
            }),
            // boost::asio::transfer_all(), // maybe needed

boost::asio::experimental::as_tuple(boost::asio::use_awaitable)
        );
    if (!ec) {
        response.message_text.resize(message_length);
        auto [ec2, bytes2] = co_await
boost::asio::async_read(
            *socket,
            boost::asio::buffer(response.message_text),
            // boost::asio::transfer_all(), // maybe needed

boost::asio::experimental::as_tuple(boost::asio::use_awaitable)
        );
        if(ec2) {
            //log here
            boost::json::value data = {

```



```

        {"error", ec.what()}
    };
    logger::Logger::GetInstance().Log("Client read
error"sv, data);
    throw ec2;
}
}
else {
    //log here
    boost::json::value data = {
        {"error", ec.what()}
    };
    logger::Logger::GetInstance().Log("Client read
error"sv, data);
    throw ec;
}
boost::json::value data = {
{"response",
    {
        {"client_request_e",
StringifyRequest(response.request)},
        {"server_response_e",
StringifyResponse(response.response)},
        {"message", response.message_text}
    }
}
};
    logger::Logger::GetInstance().Log("Read response from
server"sv, data);
    co_return response;
}

```

```

        net::awaitable<server_data_t> Read(std::shared_ptr<tcp::socket>
socket, std::chrono::system_clock::time_point start_ts){
    server_data_t response;
    size_t message_length;

    auto [ec, bytes] = co_await
        boost::asio::async_read(
            *socket,
            boost::array<boost::asio::mutable_buffer, 3>({
                net::buffer((void*)&response.request,
sizeof(client_request_e)),
                net::buffer((void*)&response.response,
sizeof(server_response_e)),
                net::buffer((void*)&message_length,
sizeof(size_t)),
            }),
            // boost::asio::transfer_all(), // maybe needed

boost::asio::experimental::as_tuple(boost::asio::use_awaitable)
        );
    if (!ec) {
        response.message_text.resize(message_length);
        auto [ec2, bytes2] = co_await
boost::asio::async_read(
            *socket,
            boost::asio::buffer(response.message_text),
            // boost::asio::transfer_all(), // maybe needed

boost::asio::experimental::as_tuple(boost::asio::use_awaitable)
        );
        if(ec2) {
            //log here
            boost::json::value data = {

```

```

        {"error", ec.what()}
    };
    logger::Logger::GetInstance().Log("Client read
error"sv, data);
    throw ec2;
}
}
else {
    //log here
    boost::json::value data = {
        {"error", ec.what()}
    };
    logger::Logger::GetInstance().Log("Client read
error"sv, data);
    throw ec;
}
    std::chrono::system_clock::time_point end_ts =
std::chrono::system_clock::now();
    boost::json::value data = {
        {"response",
            {
                {"client_request_e",
StringifyRequest(response.request)},
                {"server_response_e",
StringifyResponse(response.response)},
                {"message", response.message_text}
            }
        },
        {"response time", (end_ts - start_ts).count()}
    };
    logger::Logger::GetInstance().Log("Read first response
from server"sv, data);
    co_return response;

```

```

    }

    std::vector<std::string> SplitString(std::string str, const
std::string& delimiter){

        size_t pos;
        std::vector<std::string> split;
        while ((pos = str.find(delimiter)) != std::string::npos) {
            split.push_back(str.substr(0, pos));
            str.erase(0, pos + delimiter.length());
        }
        return split;
    }

    boost::asio::awaitable<void> User(tcp::socket socket_){
        try{
            std::shared_ptr<tcp::socket> socket =
std::make_shared<tcp::socket>(std::move(socket_));

            client_data_t request;
            server_data_t response;
            //////////////////////////////////////////
            //////////////////////////////////////////
            ////

            //set name
            std::random_device random_device;
            std::mt19937_64 generator{[&] {

std::uniform_int_distribution<std::mt19937_64::result_type> dist;
                return dist(random_device);
            }}();

```

```

do{ // try until the name will be set
    //generating random name
    std::stringstream ss;
    ss << "Bot" << std::setw(8) << std::setfill('0') <<
std::hex << generator();
    request.message_text = ss.str();
    request.request = c_set_name;

    co_await Write(socket, std::move(request));
    response = co_await Read(socket);
}while(response.response == s_failure);

while(true){ //loop to switch between being in and out
of chat
    //////////////////////////////////////
    //////////////////////////////////////
    ////

    //list of chats
    //getting rid of extra messages
    do{
        response = co_await Read(socket); // receiving
list of chats
    }while(response.request != c_get_chats); // other
messages are just ignored
    //split here
    auto  chats  =  SplitString(response.message_text,
"\n"s);

    while(true){ // list of chat loop
        if(chats.size() < 4 || generator() % 1000 < 2){
            // 0.2% possibility to create chat & if no
chats -> it will be created by default

```

```

        request.request = c_create_chat;
        // generating random chat name
        std::stringstream ss;
        ss      <<  "Chat"      <<  std::setw(8)      <<
std::setfill('0') << std::hex << generator();
        request.message_text = ss.str();
        co_await Write(socket, std::move(request));
        while(true){
            response = co_await Read(socket);
            if(response.request == c_get_chats) {
                chats
SplitString(response.message_text, "\n"s);
                break;
            }
        }
        else{ // connect to chat
            request.request = c_connect_chat;
            request.message_text
*select_randomly(chats.begin(), chats.end());
            co_await Write(socket, std::move(request));
            while(true){
                response = co_await Read(socket);
                if(response.request == c_get_chats)
chats = SplitString(response.message_text, "\n"s);
                else break;
            }
            if(response.response == s_success) break; //
after connecting
        }
    }
}

```

```

////////////////////////////////////
////////////////////////////////////
////

        //in_chat

        while(true){ // chat loop
        //getting rid of extra messages
            while(socket->available() > 0){
                response = co_await Read(socket);
                if(response.request != c_get_chats &&
response.request != c_receive_message) break;
            }
            auto i = generator();
            if(i % 100 < 5){ // 5% probability that user
leaves chat

                request.message_text.clear();
                request.request = c_leave_chat;
                co_await Write(socket, std::move(request));
                break;
            }
            else{
                // random message
                std::stringstream ss;
                ss << std::setw(16) << std::setfill('0') <<
std::hex << generator() << generator();
                request.message_text = ss.str();
                request.request = c_send_message;
                co_await Write(socket, std::move(request));
            }
        }

        //////////////////////////////////////
        //////////////////////////////////////
        ////

```

```

        //after exiting the chat loop user returns to out-of-chat
loop
    }
}
catch(sys::error_code& ec){}
catch(std::exception& ex){
    boost::json::value data = {
        {"exception", ex.what()}
    };
    logger::Logger::GetInstance().Log("Client      caught
exception"sv, data);
}
catch(...){
    boost::json::value data = {};
    logger::Logger::GetInstance().Log("Client      unknown
error"sv, data);
}

    boost::json::value data = {};
    logger::Logger::GetInstance().Log("Client      finished      it's
work"sv, data);

    co_return;
}

int main(int argc, const char* argv[]) {
    if(argc != 2){
        std::cout << "The only one correct parameter - number of
user (currently " << argc - 1 << "arguments used)" << std::endl;
        return EXIT_FAILURE;
    }
    int N = std::stoi(argv[1]);
    try {

```



```

LOG_INIT()

// 1. Инициализируем io_context
const unsigned num_threads =
std::thread::hardware_concurrency();
net::io_context ioc(num_threads);

// 2. Добавляем асинхронный обработчик сигналов SIGINT и
SIGTERM
net::signal_set signals(ioc, SIGINT, SIGTERM);
signals.async_wait([&ioc](const sys::error_code& ec,
[[maybe_unused]] int signal_number) {
    if (!ec) {
        ioc.stop();
    }
});

// 3. Настройка сетевых параметров
constexpr boost::asio::ip::port_type port = 5000;
// пока локальный адрес, но дальше надо вписать другой
net::ip::tcp::endpoint endpoint =
net::ip::tcp::endpoint(net::ip::address_v4::from_string("127.0.0.1"),
port);

//4. Запускаем N клиентов внутри контекста
for(auto i = 0; i < N; ++i){
    tcp::socket socket(ioc);
    tcp::resolver resolver(ioc);
    net::connect(socket, resolver.resolve(endpoint));
    boost::asio::co_spawn(ioc, User(std::move(socket)),
net::detached);
}

```

```

        // Эта надпись сообщает тестам о том, что тестер запущен
и готов генерировать нагрузку
        boost::json::value data = {
            {"port", port},
            {"endpoint address", endpoint.address().to_string()}
        };
        logger::Logger::GetInstance().Log("Tester    started"sv,
data);

        // 5. Запускаем обработку асинхронных операций
        RunThreads(std::max(1u, num_threads), [&ioc] {
            ioc.run();
        });
    }
    catch (const std::exception& ex) {
        boost::json::value data = {
            {"exception", ex.what()}
        };
        logger::Logger::GetInstance().Log("Tester    caught
exception"sv, data);
        return EXIT_FAILURE;
    }
}

```

ПРИЛОЖЕНИЕ Б. ФАЙЛ SERVER.CPP

МНОГОПОТОЧНОГО СЕРВЕРА

```
#include "server_interface.h"
#include "server_routines.h"
#include "logger.h"

extern std::unordered_map<std::string, chat_info> chats;
extern std::unordered_set<std::string> used_usernames;
extern std::unordered_map<int, std::string> user_data;

extern int disco_pipe[2];
extern int list_chats_pipe[2];
extern int username_enter_pipe[2];

int main(int argc, char *argv[])
{
    LOG_INIT()

    using namespace std::chrono_literals;
    if (!server_starting())
        exit(EXIT_FAILURE);
    int listen_socket;
    struct sockaddr_in address;
    // fd_set readfds;

    /* creates an UN-named socket inside the kernel and returns
     * an integer known as socket descriptor
     * This function takes domain/family as its first argument.
     * For Internet family of IPv4 addresses we use AF_INET
     */
    listen_socket = socket(AF_INET, SOCK_STREAM /*|
SOCK_NONBLOCK*/, IPPROTO_TCP);
    bzero(&address, sizeof(address));
```

```

        address.sin_port = htons(5000);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr = inet_addr("0.0.0.0");
        /*
         * The call to the function "bind()" assigns the details
specified
         * in the structure address to the socket created in the step
above
         */
        if (bind(listen_socket, (struct sockaddr *)&address,
sizeof(address)) == -1)
        {
            boost::json::value data = {"error code",
std::error_code{errno, std::generic_category()}.message()};
            logger::Logger::GetInstance().Log("Bind error"sv, data);
            close(listen_socket);
            return EXIT_FAILURE;
        }

        if (listen(listen_socket, 1000) == -1)
        {
            boost::json::value data = {"error code",
std::error_code{errno, std::generic_category()}.message()};
            logger::Logger::GetInstance().Log("Listen error"sv,
data);

            shutdown(listen_socket, SHUT_RDWR);
            return EXIT_FAILURE;
        }
        if(pipe(list_chats_pipe) < 0){
            boost::json::value data = {"error code",
std::error_code{errno, std::generic_category()}.message()};
            logger::Logger::GetInstance().Log("List of chats pipe
opening error"sv, data);

```

```

        shutdown(listen_socket, SHUT_RDWR);
        return EXIT_FAILURE;
    }
    if(pipe(disco_pipe) < 0){
        boost::json::value data = {"error code",
std::error_code{errno, std::generic_category()}.message()}};
        logger::Logger::GetInstance().Log("Disconnection pipe
opening error"sv, data);
        shutdown(listen_socket, SHUT_RDWR);
        close(list_chats_pipe[0]);
        close(list_chats_pipe[1]);
        return EXIT_FAILURE;
    }
    if(pipe(username_enter_pipe) < 0){
        boost::json::value data = {"error code",
std::error_code{errno, std::generic_category()}.message()}};
        logger::Logger::GetInstance().Log("Username enter pipe
opening error"sv, data);
        shutdown(listen_socket, SHUT_RDWR);
        close(list_chats_pipe[0]);
        close(list_chats_pipe[1]);
        close(disco_pipe[0]);
        close(disco_pipe[1]);
        return EXIT_FAILURE;
    }
    fcntl(list_chats_pipe[0], F_SETFL, fcntl(list_chats_pipe[0],
F_GETFL) | O_NONBLOCK);
    fcntl(disco_pipe[0], F_SETFL, fcntl(disco_pipe[0], F_GETFL)
| O_NONBLOCK);
    fcntl(username_enter_pipe[0], F_SETFL,
fcntl(username_enter_pipe[0], F_GETFL) | O_NONBLOCK);

```

```

        std::thread(user_name_enter, username_enter_pipe[0],
list_chats_pipe[1]).detach();
        std::thread(list_of_chats, list_chats_pipe[0]).detach();
        std::thread(disconnect).detach();

        struct sockaddr_in client_address;
        socklen_t address_length = sizeof(sockaddr_in);
        int connection_socket;
        while ((connection_socket = accept(listen_socket, (struct
sockaddr *)&client_address, &address_length)) != -1)
        {
            struct timeval tv;
            tv.tv_sec = 0;
            tv.tv_usec = TIMEOUT_MICRO;
            setsockopt(connection_socket, SOL_SOCKET, SO_RCVTIMEO,
(const char*)&tv, sizeof tv);
            write(username_enter_pipe[1], &connection_socket,
sizeof(int));
        }
        if (!(errno & (EAGAIN | EWOULDBLOCK)))
        {
            boost::json::value data = {"error code",
std::error_code{errno, std::generic_category()}.message()};
            logger::Logger::GetInstance().Log("Accept failure"sv,
data);

            return EXIT_FAILURE;
        }
        // perror("Server died with status");
        server_ending(); // because why not
        exit(EXIT_SUCCESS);
    }

```

ПРИЛОЖЕНИЕ В. ФАЙЛ MAIN.CPP МНОГОПОТОЧНО-АСИНХРОННОГО СЕРВЕРА

```
#include <boost/asio/io_context.hpp>
#include <boost/asio/signal_set.hpp>

#include <iostream>
#include <thread>

#include "server.h"
#include "logger.h"

using namespace std::literals;
namespace net = boost::asio;
namespace sys = boost::system;

namespace {

// Запускает функцию fn на n потоках, включая текущий
template <typename Fn>
void RunThreads(unsigned n, const Fn& fn) {
    n = std::max(1u, n);
    std::vector<std::jthread> workers;
    workers.reserve(n - 1);
    // Запускаем n-1 рабочих потоков, выполняющих функцию fn
    while (--n) {
        workers.emplace_back(fn);
    }
    fn();
}

} // namespace

int main(int argc, const char* argv[]) {
```

```

try {

    LOG_INIT()

    // 1. Инициализируем io_context
    const unsigned num_threads =
std::thread::hardware_concurrency();
    net::io_context ioc(num_threads);

    // 2. Добавляем асинхронный обработчик сигналов SIGINT и
SIGTERM
    net::signal_set signals(ioc, SIGINT, SIGTERM);
    signals.async_wait([&ioc](const sys::error_code& ec,
[[maybe_unused]] int signal_number) {
        if (!ec) {
            boost::json::value data = {
                {"signal", signal_number}
            };
            logger::Logger::GetInstance().Log("Signal
received"sv, data);
            ioc.stop();
        }
    });

    // 3. Настройка сетевых параметров
    constexpr boost::asio::ip::port_type port = 5000;
    const auto address =
net::ip::tcp::endpoint(net::ip::tcp::v4(), port).address();
    net::ip::tcp::endpoint endpoint = {address, port};

    //4. Запускаем серверную обработку запросов
    std::make_shared<server::Server>(ioc, endpoint,
"./chats"s)->Run();
}

```



```

        // Эта надпись сообщает тестам о том, что сервер запущен
и готов обрабатывать запросы
        boost::json::value data = {
            {"address", address.to_string()},
            {"port", port}
        };
        logger::Logger::GetInstance().Log("Server    started"sv,
data);

        // 5. Запускаем обработку асинхронных операций
        RunThreads(std::max(1u, num_threads), [&ioc] {
            ioc.run();
        });
    }
    catch (const std::exception& ex) {
        boost::json::value data = {
            {"exception", ex.what()}
        };
        logger::Logger::GetInstance().Log("Exception
occured"sv, data);
        return EXIT_FAILURE;
    }
}

```

ПРИЛОЖЕНИЕ Г. ФАЙЛ SERVER.CPP МНОГОПОТОЧНО-АСИНХРОННОГО СЕРВЕРА

```
#include "server.h"
#include "logger.h"

#include <iostream>
#include <filesystem>
#include <fstream>
#include <ctime>

#include <boost/asio/detached.hpp>
#include <boost/asio/co_spawn.hpp>

#include <boost/scoped_ptr.hpp>

namespace server {
std::string StringifyRequest(client_request_e r){
    switch(r){
        case 0: return "c_set_name"s;
        case 1: return "c_create_chat"s;
        case 2: return "c_connect_chat"s;
        case 3: return "c_send_message"s;
        case 4: return "c_leave_chat"s;
        case 5: return "c_disconnect"s;
        case 6: return "c_receive_message"s;
        case 7: return "c_wrong_request"s;
        case 8: return "c_get_chats"s;
        default: return "wrong input"s;
    }
}

std::string StringifyResponse(server_response_e r){
    switch(r){
```

```

        case 0: return "s_success"s;
        case 1: return "s_failure"s;
        default: return "wrong input"s;
    }
}

net::awaitable<void> Write(std::shared_ptr<tcp::socket> socket,
server_data_t&& response){
    size_t message_length = response.message_text.size();
    boost::scoped_ptr<server_data_t> safe_response(new
server_data_t(response));

    boost::json::value data = {
        {"response",
        {
            {"client_request_e",
StringifyRequest(safe_response->request)},
            {"server_response_e",
StringifyResponse(safe_response->response)},
            {"message", safe_response->message_text}
        }
        },
        {"socket", int(socket->native_handle())}
    };
    logger::Logger::GetInstance().Log("Wrote response"sv, data);

    auto [ec, bytes] = co_await net::async_write(
        *socket,
        boost::array<boost::asio::mutable_buffer, 4>({
            net::buffer((void*)&safe_response->request,
sizeof(client_request_e)),
            net::buffer((void*)&safe_response->response,
sizeof(server_response_e)),

```

```

        net::buffer((void*)&message_length,
sizeof(size_t)),
        net::buffer(safe_response->message_text)
    }},
    // boost::asio::transfer_all(), // maybe needed
    net::experimental::as_tuple(net::use_awaitable)
);

if(ec) {
    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"error", ec.what()}
    };
    logger::Logger::GetInstance().Log("Write      error"sv,
data);

    throw ec;
}

}

// Chat methods

boost::asio::awaitable<void>
Chat::AddUser(std::shared_ptr<tcp::socket>      socket,      std::string
username){
    co_await net::dispatch(strand, boost::asio::use_awaitable);

    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", username},
        {"chat_name", path.stem().c_str()}
    };
};

```

```

        logger::Logger::GetInstance().Log("User added to chat"sv,
data);

        messages_offset.insert({socket, 0});

        std::fstream chat_file(path);
        chat_file.seekp(0, chat_file.end);
        auto file_size = chat_file.tellp();

        server_data_t resp;

        std::streamoff off = 0;
        chat_file.seekg(off, chat_file.beg);

        while (off != file_size) {
            resp.response = server_response_e::s_success;
            resp.request = client_request_e::c_receive_message; //
now we change the request that we are serving because we did connect
to chat

            std::getline(chat_file, resp.message_text, '\r');
            off = chat_file.tellp();
            co_await Write(socket, std::move(resp));
            // invoking saved method to send all unread chat messages
to connected user

            // std::cout << "[off = " << off << ", file_size = " <<
file_size << "]\n";
        }

        messages_offset[socket] = off;
        usernames[socket] = username;
    }

```

```

        boost::asio::awaitable<void>
Chat::DeleteUser(std::shared_ptr<tcp::socket> socket){
    co_await net::dispatch(strand, boost::asio::use_awaitable);

    std::string username;
    if(usernames.contains(socket))            username            =
usernames.at(socket);

    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", username},
        {"chat_name", path.stem().c_str()}
    };
    logger::Logger::GetInstance().Log("User        deleted        from
chat"sv, data);

    messages_offset.erase(socket);
    usernames.erase(socket);

}

        boost::asio::awaitable<void>
Chat::SendMessage(std::shared_ptr<tcp::socket> socket, std::string
message){
    co_await net::dispatch(strand, boost::asio::use_awaitable);

    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", usernames.at(socket)},
        {"chat_name", path.stem().c_str()},
        {"message", message}
    };
};

```

```

        logger::Logger::GetInstance().Log("User sent message to
chat"sv, data);

        if(messages_offset.contains(socket)){
            std::fstream chat_file;

            chat_file.open(path,          std::fstream::out          |
std::fstream::app);

            chat_file << usernames[socket] << "\t";

            time_t curr_time;
            tm * curr_tm;
            std::string timedate;
            timedate.reserve(100);
            time(&curr_time);
            curr_tm = localtime(&curr_time);
            strftime(timedate.data(), 99, "%T %D", curr_tm);

            chat_file << timedate.c_str() << std::endl << message <<
"\r";

            auto file_size = chat_file.tellp();

            chat_file.close();

            chat_file.open(path,          std::fstream::in          |
std::fstream::app);

            for(auto& [usr_socket, off] : messages_offset){
                chat_file.seekg(off, chat_file.beg);
                while(off != file_size){

```

```

        server_data_t resp;
        resp.request =
client_request_e::c_receive_message;
        resp.response = server_response_e::s_success;
        std::getline(chat_file, resp.message_text,
'\r');

        off = chat_file.tellp();
        try{
            co_await Write(usr_socket, std::move(resp));
        } catch(sys::error_code& ec){
            if(usr_socket == socket) throw ec;
            // if our socket is broken - that's a problem
            // else - let it be destroyed afterwards by
it's session
        }
    }
}

        chat_file.close();
    }
}

    boost::asio::awaitable<bool>
Chat::ContainsUser(std::shared_ptr<tcp::socket> socket){
    co_await net::dispatch(strand, boost::asio::use_awaitable);

    std::string username;
    if(usernames.contains(socket)) username =
usernames.at(socket);
    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", username},
        {"chat_name", path.stem().c_str()}
    }
}

```



```

    };
    logger::Logger::GetInstance().Log("Checked if user is in the
chat"sv, data);

    co_return messages_offset.contains(socket);
}

// // ChatManager methods

boost::asio::awaitable<bool> ChatManager::SetName(std::string
name, std::shared_ptr<tcp::socket> socket){
    bool result = true;
    co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);
    for(auto [socket_ptr, user] : users_){
        if(user.name == name){
            result = false;
            break;
        }
    }

    if(result) {
        users_.insert({socket, {name, ""}});
    }

    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", name},
        {"result", int(result)} // maybe change to another type
    };
    logger::Logger::GetInstance().Log("Name requested by a
user"sv, data);

```

```

        co_return result;
    }

    boost::asio::awaitable<std::string> ChatManager::ChatList(){
        co_await net::dispatch(chats_strand_,
boost::asio::use_awaitable);

        std::string list;
        for(auto chat : chats_) list += chat.first + "\n";

        boost::json::value data = {
            {"chat list", list}
        };
        logger::Logger::GetInstance().Log("Formed a chatlist"sv,
data);

        co_return list;
    }

    boost::asio::awaitable<void>
ChatManager::UpdateChatList(std::shared_ptr<tcp::socket> socket){
        co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);

        server_data_t resp;
        std::string list = co_await ChatList();

        boost::json::value data = {
            {"chat list", list}
        };
        logger::Logger::GetInstance().Log("Chat list updated"sv,
data);

```

```

        for(auto [user_socket, user] : users_){
            if(user.chat_name.empty()){
                resp.message_text = list;
                resp.request = c_get_chats;
                resp.response = s_success;
                try{
                    co_await Write(user_socket, std::move(resp));
                } catch(sys::error_code& ec){
                    if(user_socket == socket) throw ec;
                    // our socket - our problem, else - nah
                }
            }
        }
    }

    boost::asio::awaitable<void>
    ChatManager::Disconnect(std::shared_ptr<tcp::socket> socket){
        co_await net::dispatch(usernames_strand_,
        boost::asio::use_awaitable);

        std::string name;
        if(users_.contains(socket)) name = users_.at(socket).name;

        boost::json::value data = {
            {"socket", int(socket->native_handle())},
            {"username", name}
        };
        logger::Logger::GetInstance().Log("User      disconnected"sv,
data);

        if(users_.contains(socket)) {
            User usr = users_.at(socket);

```

```

        co_await net::dispatch(chats_strand_,
boost::asio::use_awaitable);

        //synchronized deletion of the user listing in chat
through chat's method
        if(chats_.contains(usr.chat_name)) {
            co_await
chats_.at(usr.chat_name).DeleteUser(socket); // throws exception
because ???
        }
        //synchronized deletion of the user from the list of
users in the chat manager
        co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);
        users_.erase(socket);
    }
}

boost::asio::awaitable<void>
ChatManager::ConnectChat(std::string chat_name,
std::shared_ptr<tcp::socket> socket){ //Sends everything the client
needs, including success or failure notification
    using namespace std::literals;
    co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);

    std::string name;
    if(users_.contains(socket)) name = users_.at(socket).name;

    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", name},
        {"chat name", chat_name}
    }
}

```

```

        };
        logger::Logger::GetInstance().Log("User connected to
chat"sv, data);

        if(users_.contains(socket)) {
            User usr = users_.at(socket);
            co_await net::dispatch(chats_strand_,
boost::asio::use_awaitable);

            server_data_t resp;
            resp.message_text = chat_name;
            resp.request = client_request_e::c_connect_chat;

            if(!chats_.contains(chat_name)) {
                resp.response = server_response_e::s_failure;
                co_await Write(socket, std::move(resp)); // invoking
saved method to inform client that the chat failed to connect
                co_return;
            }

            resp.response = server_response_e::s_success;
            co_await Write(socket, std::move(resp)); // invoking
saved method to inform client that the chat is connected RN
            //keep the state because AddUser method will be the last
one to send the message

            co_await chats_.at(chat_name).AddUser(socket, usr.name);

            //synchronized update of user status
            co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);

```

```

        if(users_.contains(socket)) users_.at(socket).chat_name
= chat_name;

    }
}

boost::asio::awaitable<bool>
ChatManager::CreateChat(std::string chat_name){
    co_await net::dispatch(chats_strand_,
boost::asio::use_awaitable);

    bool result = true;
    if(chats_.contains(chat_name) || chat_name == ""){
        result = false;
    }
    if(result) chats_.insert({chat_name, {io_, path_of_chats /
(chat_name + ".chat")}}});

    boost::json::value data = {
        {"chat name", chat_name},
        {"result", int(result)}
    };
    logger::Logger::GetInstance().Log("Chat creation request"sv,
data);

    co_return result;
}

boost::asio::awaitable<bool>
ChatManager::LeaveChat(std::shared_ptr<tcp::socket> socket){
    co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);

```

```

        bool result;

        if(users_.contains(socket)){
            User usr = users_.at(socket);
            co_await net::dispatch(chats_strand_,
boost::asio::use_awaitable);
            if(!chats_.contains(usr.chat_name) ||
                !co_await
chats_.at(usr.chat_name).ContainsUser(socket)){
                result = false;
            }
            else{
                co_await
chats_.at(usr.chat_name).DeleteUser(socket);
                co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);
                if(users_.contains(socket))
users_.at(socket).chat_name.clear();
                result = true;
            }
        }

        std::string name;
        if(users_.contains(socket)) name = users_.at(socket).name;

        boost::json::value data = {
            {"socket", int(socket->native_handle())},
            {"username", name},
            {"result", int(result)}
        };
        logger::Logger::GetInstance().Log("Leave chat request"sv,
data);

```

```

        co_return result;
    }

    boost::asio::awaitable<void>
ChatManager::SendMessage(std::string message,
std::shared_ptr<tcp::socket> socket){ // this one actually sends
messages by itself
    co_await net::dispatch(usernames_strand_,
boost::asio::use_awaitable);

    std::string name;
    if(users_.contains(socket)) name = users_.at(socket).name;

    boost::json::value data = {
        {"socket", int(socket->native_handle())},
        {"username", name},
        {"message", message}
    };
    logger::Logger::GetInstance().Log("Send message request"sv,
data);

    if(users_.contains(socket)){
        User usr = users_.at(socket);

        co_await net::dispatch(chats_strand_,
boost::asio::use_awaitable);
        if(chats_.contains(usr.chat_name)) co_await
chats_.at(usr.chat_name).SendMessage(socket, message);
    }
}

// boost::asio::awaitable<User*>
ChatManager::IdentifyUser(tcp::socket* socket){

```



```

        //          co_await    net::dispatch(usernames_strand_,
boost::asio::use_awaitable);

        //      boost::json::value data = {
        //          {"socket", int(socket->native_handle())},
        //      };
        //          logger::Logger::GetInstance().Log("Identify    user
request"sv, data);

        //      User* usr;
        //      if(!users_.contains(socket)) usr = nullptr;
        //      else usr = &users_.at(socket);
        //      co_return usr;
        //  }

// Server methods

void Server::Run(){
    DoAccept();
}

std::shared_ptr<Server> Server::GetSharedThis(){
    return this->shared_from_this();
}

void Server::DoAccept() {
    // Метод socket::async_accept создаст сокет и передаст его
передан в OnAccept
    acceptor_.async_accept(
        // Передаём последовательный исполнитель, в котором будут
вызываться обработчики
        // асинхронных операций сокета

```

```

        net::make_strand(ioc_),
        // При помощи bind_front_handler создаём обработчик,
        привязанный к методу OnAccept
        // текущего объекта.
        beast::bind_front_handler(&Server::OnAccept,
        GetSharedThis())));
    }

    void Server::OnAccept(sys::error_code ec, tcp::socket socket) {
        using namespace std::literals;

        if (ec) {
            boost::json::value data = {
                {"error", ec.what()}
            };
            logger::Logger::GetInstance().Log("Accept      error"sv,
data);
        }

        // Для того, чтобы подключение произошло
        // нужно прочитать сначала данные для подтверждения
        подключения

        // Асинхронно обрабатываем сессию
        AsyncRunSession(std::move(socket));

        // Принимаем новое соединение
        DoAccept();
    }

    void Server::AsyncRunSession(tcp::socket&& socket) {

```

```

        boost::asio::co_spawn(ioc_, Session(std::move(socket),
&chat_manager_), boost::asio::detached);
    }

    // Session
    boost::asio::awaitable<void> Session(tcp::socket&& socket,
ChatManager* chatManager){

        bool running = true;
        session_state sessionState = session_state::reading;
        client_state clientState = client_state::no_name;
        client_data_t request_;
        server_data_t response_;
        size_t message_length;

        std::shared_ptr<tcp::socket> socket_ =
std::make_shared<tcp::socket>(std::move(socket));

        boost::json::value data = {
            {"socket", int(socket_>native_handle())}
        };
        logger::Logger::GetInstance().Log("Session started"sv,
data);

        try{
            while(true){
                auto [ec, bytes] = co_await
                boost::asio::async_read(
                    *socket_,
                    boost::array<boost::asio::mutable_buffer, 2>({

```

```

boost::asio::buffer((void*)&request_.request,
sizeof(request_.request)),
                boost::asio::buffer((void*)&message_length,
sizeof(message_length))
                }),
                // boost::asio::transfer_all(), // maybe needed

boost::asio::experimental::as_tuple(boost::asio::use_awaitable)
);
if (!ec) {
    request_.message_text.resize(message_length);
    auto [ec2, bytes2] = co_await
    boost::asio::async_read(
        *socket_,
        boost::asio::buffer(request_.message_text),
        // boost::asio::transfer_all(), // maybe
needed

boost::asio::experimental::as_tuple(boost::asio::use_awaitable)
);
if(!ec2) sessionState = session_state::handling;
else {
    //log here
    boost::json::value data = {
        {"socket", int(socket_
>native_handle())},
        {"error", ec2.what()}
    };
    logger::Logger::GetInstance().Log("Read
error"sv, data);

    throw ec2;
}

```

```

    }
    else {
        //log here
        boost::json::value data = {
            {"socket", int(socket_>native_handle())},
            {"error", ec.what()}
        };
        logger::Logger::GetInstance().Log("Read
error"sv, data);

        throw ec;
    }
    // std::cout << "Received message: {request: " <<
request_.request << ", message: \"" << request_.message_text << "\"}"
<< std::endl;

    response_.request = request_.request;
    switch(request_.request){
        case client_request_e::c_set_name:
            if(clientState != client_state::no_name){
                response_.message_text = "Cannot set
name twice!";

                response_.response =
server_response_e::s_failure;
                co_await Write(socket_,
std::move(response_));
            } else {
                if(co_await chatManager-
>SetName(request_.message_text, socket_)){
                    clientState =
client_state::list_chats;

                    response_.message_text = "";
                    response_.response =
server_response_e::s_success;

```

```

                                co_await          Write(socket_,
std::move(response_));

                                response_.request          =
client_request_e::c_get_chats;

                                response_.response          =
server_response_e::s_success;

                                response_.message_text  =  co_await
chatManager->ChatList();

                                co_await          Write(socket_,
std::move(response_));

                                } else {
                                    response_.message_text  =  "Username
is already in use";

                                    response_.response          =
server_response_e::s_failure;

                                    co_await          Write(socket_,
std::move(response_));
                                }
                            }
                        break;
                    case client_request_e::c_create_chat:
                        if(clientState != client_state::list_chats){
                            response_.message_text = "Cannot create
chats, wrong state";

                            response_.response          =
server_response_e::s_failure;

                            co_await          Write(socket_,
std::move(response_));
                        }

```

```

else{
    if(!co_await chatManager->CreateChat(request_.message_text)){
        response_.message_text = "Chat with
such name already exists";
        response_.response =
server_response_e::s_failure;
        co_await Write(socket_,
std::move(response_));
    }
    else{
        // send everyone out of chat new list
        co_await chatManager->UpdateChatList(socket_);
    }
}
break;
case client_request_e::c_connect_chat:
    if(clientState != client_state::list_chats){
        response_.message_text = "Cannot connect
chat, wrong state!";
        response_.response =
server_response_e::s_failure;
        co_await Write(socket_,
std::move(response_));
    }
    else {
        co_await chatManager->ConnectChat(request_.message_text, socket_);
        clientState = client_state::in_chat;
    }
    break;
case client_request_e::c_send_message:

```

```

        if(clientState != client_state::in_chat){
            response_.message_text = "";
            response_.response =
server_response_e::s_failure;
            co_await Write(socket_,
std::move(response_));
        }
        else
            co_await chatManager-
>SendMessage(request_.message_text, socket_);
            break;
        case client_request_e::c_leave_chat:
            if(clientState == client_state::in_chat &&
co_await chatManager->LeaveChat(socket_)){
                // Can leave chat, ok state && could to
leave the chat
                clientState = client_state::list_chats;
                response_.request =
client_request_e::c_get_chats;
                response_.response =
server_response_e::s_success;
                response_.message_text = co_await
chatManager->ChatList();
                co_await Write(socket_,
std::move(response_));
            }
            else{
                response_.message_text = "";
                response_.response =
server_response_e::s_failure;
                co_await Write(socket_,
std::move(response_));
            }

```



```

        //no messages about leaving
        break;
    default:
        response_.message_text = "Wrong request";
        response_.request =
client_request_e::c_wrong_request;
        response_.response =
server_response_e::s_failure;
        co_await Write(socket_,
std::move(response_));
    }
}
}
catch(std::exception ex){
    boost::json::value data = {
        {"exception", ex.what()}
    };
    logger::Logger::GetInstance().Log("Exception
occured"sv, data);
}
catch(...){}
// catch(beast::error_code ec){ //read\write errors
//     std::cout << "Boost error occured: " << ec.what() <<
std::endl;
// }
//disconnection
// std::cout << "Session ended!" << std::endl;
co_await chatManager->Disconnect(socket_);

data = {
    {"socket", int(socket_->native_handle())}
};
logger::Logger::GetInstance().Log("Session ended"sv, data);

```

```
        socket_->close();  
    }  
  
} // namespace server
```

ПРИЛОЖЕНИЕ Д. ФАЙЛ MAINWINDOW.CPP

КЛИЕНТСКОГО ПРИЛОЖЕНИЯ

```
#include "mainwindow.h"
#include "../ui_mainwindow.h"
#include <string.h>
#include <QTableView>
#include <QListWidget>
#include <QPushButton>
#include <QDebug>
#include <QHeaderView>
#include <QStandardItem>
#include "client_interface.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    MainWindow::setWindowTitle("Chat");
    socket = new QTcpSocket(this);
    bool start_ok = StartConnection();
    connect(ui->NameLine, &QLineEdit::returnPressed, this,
&MainWindow::on_nameLine_returnPressed1st);
    CRH = new ReadRoutineHandler(this, socket);
    if(start_ok){
        CRH->startThread();
    }
    else {
        QMessageBox::warning(this, "ACHTUNG!", "Server error:
could not connect.\nTry reopening app once we reactivate the server!");
        this->close();
    }
}
```

```

void MainWindow::on_serverDown(){
    QMessageBox::warning(this, "ACHTUNG!", "Server is down, try
reconnecting later");
    this->close();
}

bool MainWindow::StartConnection(){
    socket->connectToHost("127.0.0.1", 5000);
    socket->setReadBufferSize(10000);
    connect(socket,      &QTcpSocket::disconnected,      socket,
&QTcpSocket::deleteLater);
    if(socket->waitForConnected(3000)){
        qDebug() << "Connected!";
        connect(socket,      &QTcpSocket::errorOccurred,      this,
&MainWindow::on_serverDown);
        return true;
    }
    else{
        qDebug() << "Not connected!";
        return false;
    }
}

void MainWindow::show_chats(const std::string& chat_list){
    //deleting start ui
    MainWindow::setWindowTitle("Chat");
    SafeCleaning(ui->verticalLayout->layout());
    //creating mid ui
    QLineEdit* NewChatLine = new QLineEdit();
    NewChatLine->setPlaceholderText("Enter new chat's name:");
    connect(NewChatLine,      &QLineEdit::returnPressed,      this,
&MainWindow::on_createChat1st);
}

```

```

        ui->verticalLayout->insertWidget(0, NewChatLine);
        QTableView* ChatsTable = new QTableView;
        QStringList                chats                                =
QString::fromStdString(chat_list).split('\n');
        model = new QStandardItemModel(chats.length() - 1, 1);
        ChatsTable->setModel(model);
        for(size_t row = 0; row < chats.length() - 1; ++row){
            auto chat = new QPushButton();
            chat->setText(chats.at(row));
            ChatsTable->setIndexWidget(model->index(row, 0), chat);
            connect(chat,          &QPushButton::clicked,          this,
&MainWindow::on_chatButton_Clicked1st);
            connect(ChatsTable,    &QTableView::destroyed,    chat,
&QPushButton::deleteLater);
        }
        ChatsTable->horizontalHeader()->hide();
        ChatsTable->verticalHeader()->hide();
        ChatsTable->horizontalHeader()-
>setStretchLastSection(true); //try to fill the whole widget
        ui->verticalLayout->insertWidget(1, ChatsTable);
        ChatsTable->show();
    }
    MainWindow::~MainWindow()
    {
        delete ui;
        CRH->stopThread();
        delete CRH;
    }

    void MainWindow::on_nameLine_returnPressed1st()
    {
        if(ui->NameLine->text().size() > 32){

```

```

        QMessageBox::warning(this, "ACHTUNG!", "This name is too
big!");
        ui->NameLine->clear();
    }
    else{
        c_message.request = c_set_name;
        c_message.message_text = ui->NameLine-
>text().toStdString();
        SendToServer(socket, &c_message);
    }
}

void MainWindow::on_nameLine_returnPressed2nd(server_data_t
s_message){
    if(s_message.responce == s_failure){
        QMessageBox::warning(this, "ACHTUNG!", "This username is
already used!");
    }
}

void MainWindow::list_of_chats(server_data_t s_message){
    if(s_message.responce == s_failure){
        QMessageBox::warning(this, "ACHTUNG!", "Could not get
list of chats");
    }
    else{
        show_chats(s_message.message_text);
    }
}

void MainWindow::on_chatButton_Clicked1st(){
    QPushButton* chat = qobject_cast<QPushButton*>(sender()); //
get QObject that emitted the signal

```

```

        c_message.request = c_connect_chat;
        c_message.message_text = chat->text().toStdString();
        SendToServer(socket, &c_message);
    }

    void MainWindow::on_chatButton_Clicked2nd(server_data_t
s_message){
        if(s_message.responce == s_success){
            SafeCleaning(ui->verticalLayout->layout());
            MainWindow::setWindowTitle("Chat: " +
QString::fromStdString(s_message.message_text));
            QPushButton* LeaveButton = new QPushButton("Leave
chat");

            connect(LeaveButton, &QPushButton::clicked, this,
&MainWindow::on_leaveChat1st);

            QListWidget* QLW = new QListWidget();
            QLineEdit* MessageLine = new QLineEdit();
            MessageLine->setPlaceholderText("Enter your
message:");

            connect(MessageLine, &QLineEdit::returnPressed,
this, &MainWindow::on_sendMessage);

            ui->verticalLayout->insertWidget(0, LeaveButton);
            ui->verticalLayout->insertWidget(1, QLW);
            ui->verticalLayout->insertWidget(2, MessageLine);
        }
        else{
            QMessageBox::warning(this, "ACHTUNG!", "Could not
connect to this chat!");
        }
    }

    void MainWindow::on_newMessage(std::string& message_text){
        static size_t row = 0;

```

```

        auto QLW = qobject_cast<QListWidget*>(ui->verticalLayout-
>itemAt(1)->widget());
        QLW->addItem(QString::fromStdString(message_text));
    }

    void MainWindow::on_createChat1st(){
        auto line = qobject_cast<QLineEdit*>(sender());
        if(line->text().contains(' ')){
            QMessageBox::warning(this, "ACHTUNG!", "Chat name cannot
contain with a space");
            line->clear();
        }
        else{
            c_message.request = c_create_chat;
            c_message.message_text = line->text().toStdString();
            SendToServer(socket, &c_message);
        }
        line->clear();
    }

    void MainWindow::on_createChat2nd(server_data_t s_message){
        if(s_message.responce == s_failure){
            QMessageBox::warning(this, "ACHTUNG!", "Could not create
this chat");
        }
    }

    void MainWindow::on_leaveChat1st(){
        client_data_t c_message;
        c_message.request = c_leave_chat;
        c_message.message_text = "";
        SendToServer(socket, &c_message);
    }

```



```
void MainWindow::on_sendMessage(){
    client_data_t c_message;
    c_message.message_text = qobject_cast<QLineEdit*>(sender())-
>text().toString();
    c_message.request = c_send_message;
    SendToServer(socket, &c_message);
    qobject_cast<QLineEdit*>(sender())->clear();
}
```

ПРИЛОЖЕНИЕ Е. ФАЙЛЫ ПОТОКА ЧТЕНИЯ КЛИЕНТСКОГО ПРИЛОЖЕНИЯ

readroutine.h — определение класса для неблокирующего чтения в клиентском приложении:

```
#pragma once

#include <QThread>
#include <QTcpSocket>
#include <QDebug>
#include <QObject>
#include <memory>
#include "client_interface.h"

enum client_state_t{
    no_name = 0,
    no_chat,
    in_chat
};

class ReadRoutine : public QObject {
    Q_OBJECT

    enum unblocked_read_state {
        REQ_TYPE,
        RESP_TYPE,
        LENGTH,
        MESSAGE
    };

    server_data_t s_d;
    QTcpSocket* tcp_socket;
```

```

        size_t off = 0;
        unblocked_read_state state = REQ_TYPE;
        size_t length;
        bool readed = false;

public:
        std::atomic_bool go_on;
        std::atomic_int client_state;
        ReadRoutine(QTcpSocket*);

public slots:
        bool UnblockedReadFromServer(); /* создает и запускает */
        void stop(); /* останавливает */

signals:
        void finished(); /* сигнал о завершении работы */
        void new_message_come(std::string&);
        void get_chats(server_data_t);
        void connect_chat(server_data_t);
        void create_chat(server_data_t);
        void send_name(server_data_t);
        void leave_chat(server_data_t);
};

class ReadRoutineHandler : public QObject {
        Q_OBJECT

        QTcpSocket* tcp_socket;
public:
        ReadRoutineHandler(QObject *parent, QTcpSocket*);
        ~ReadRoutineHandler();
        void startThread();
        ReadRoutine* chat_proc;

```

```

public slots:
    void stopThread();

signals:
    void stop(); //остановка потока
};

```

readroutine.cpp – определение методов класса из readroutine.h в клиентском приложении:

```

#include "readroutine.h"
#include "client_interface.h"
#include "mainwindow.h"
#include <QDebug>

ReadRoutine::ReadRoutine(QTcpSocket* socket) {
    tcp_socket = socket;
    go_on = true;
    length = sizeof(server_responce_e);
}

bool ReadRoutine::UnblockedReadFromServer(){
    int received = 1;
    static size_t help_length;

    while (received > 0) {
        switch (state) {
            case REQ_TYPE:
                received = tcp_socket->read((char*)&s_d.request,
length);

                if (received < 0) return false;
                off += received;
                if (off == length) {

```

```

        state = RESP_TYPE;
        length = sizeof(server_responce_e);
        off = 0;
    }
    break;
case RESP_TYPE:
    received = tcp_socket-
>read((char*)&s_d.responce, length);
    if (received < 0) return false;
    off += received;
    if (off == length) {
        state = LENGTH;
        length = sizeof(size_t);
        off = 0;
    }
    break;
case LENGTH:
    received = tcp_socket->read((char*)&help_length,
length);

    if (received < 0) return false;
    off += received;
    if (off == length) {
        length = help_length;
        state = MESSAGE;
        s_d.message_text.resize(length);
        off = 0;
    }
    break;
case MESSAGE:
    received = tcp_socket-
>read((char*)s_d.message_text.data(), length);
    if (received < 0) return false;
    off += received;

```

```

        if (off == length) {
            state = REQ_TYPE;
            length = sizeof(server_responce_e);
            off = 0;
            readed = true;
            qDebug() << "Client state: " << client_state;
            switch(client_state){
                case no_name:
                    switch(s_d.request){
                        case c_set_name:
                            client_state = no_chat;
                            emit send_name(s_d);
                            break;
                        default:
                            qDebug() << "Chat routine
received wrong data";
                    }
                    break;
                case no_chat:
                    switch(s_d.request){
                        case c_get_chats:
                            emit get_chats(s_d);
                            break;
                        case c_create_chat:
                            emit create_chat(s_d);
                            break;
                        case c_connect_chat:
                            if(s_d.responce ==
s_success) client_state = in_chat;
                            emit connect_chat(s_d);
                            break;
                        default:

```

```

qDebug() << "Chat routine
received wrong data";

    }
    break;
case in_chat:
    switch(s_d.request){
        case c_receive_message:
            emit
new_message_come(s_d.message_text);

            break;
        case c_get_chats:
            client_state = no_chat;
            emit get_chats(s_d);
            break;
        default:
            qDebug() << "Chat routine
received wrong data";

    }
}
}
}
}
}
return true;
}

void ReadRoutine::stop() {
    emit finished();
}

ReadRoutineHandler::ReadRoutineHandler(QObject*parent,
QTcpSocket* tcp_socket) {
    setParent(parent);
    this->tcp_socket = tcp_socket;

```

```

    }

    void ReadRoutineHandler::startThread(){
        chat_proc = new ReadRoutine(tcp_socket);
        chat_proc->client_state = no_name;
        QThread* thread = new QThread;
        chat_proc->moveToThread(thread);
        //signals to connect two threads
        connect(chat_proc,                &ReadRoutine::new_message_come,
qobject_cast<MainWindow*>(parent()), &MainWindow::on_newMessage);
        connect(chat_proc,                &ReadRoutine::get_chats,
qobject_cast<MainWindow*>(parent()), &MainWindow::list_of_chats);
        connect(chat_proc,                &ReadRoutine::create_chat,
qobject_cast<MainWindow*>(parent()), &MainWindow::on_createChat2nd);
        connect(chat_proc,                &ReadRoutine::connect_chat,
qobject_cast<MainWindow*>(parent()),
&MainWindow::on_chatButton_Clicked2nd);
        connect(chat_proc,                &ReadRoutine::send_name,
qobject_cast<MainWindow*>(parent()),
&MainWindow::on_nameLine_returnPressed2nd);
        //correct memory management if thread is stopped and so on
        connect(chat_proc,                &ReadRoutine::finished,        thread,
&QThread::quit);
        connect(qobject_cast<QIODevice*>(tcp_socket),
&QIODevice::readyRead,                chat_proc,
&ReadRoutine::UnblockedReadFromServer);
        connect(this,                &ReadRoutineHandler::stop,        chat_proc,
&ReadRoutine::stop);
        connect(chat_proc,                &ReadRoutine::finished,        chat_proc,
&ReadRoutine::deleteLater);
        connect(thread,                &QThread::finished,        thread,
&QThread::deleteLater);
        thread->start();
    }

```



```
}

void ReadRoutineHandler::stopThread() {
    emit stop();
}

ReadRoutineHandler::~~ReadRoutineHandler(){}
```