# Введение

**Актуальность работы.** Сейчас сложно представить современный мир без сети интернет. В нем мы делаем покупки, получаем образование, развлекаемся, общаемся. Свободный обмен информацией является одним из основных преимуществ глобальной паутины. Большую часть времени в интернете люди проводят в различных социальных сетях, мессенджерах.

Основой таких приложений и всего интернета в целом является стек протоколов TCP/IP. Стандартная архитектура таких приложений является клиент – серверной. Архитектура клиент-сервер - это вычислительная модель, в которой сервер хранит и обрабатывает большую часть ресурсов и служб, которые потребляются клиентом. В архитектуре этого типа один или несколько клиентских компьютеров подключены к центральному серверу через сеть или подключение к Интернету. Поверх транспортного протокола TCP для приложения часто пишут свой более высокоуровневый протокол обмена данными, который адаптирован под конкретную бизнес логику.

Современные тенденции требуют быстрой разработки программного обеспечения, чтобы в короткие сроки его можно было доставить пользователю, заказчику и получать прибыль. В связи с этим очень выгодно разрабатывать кроссплатформенное программное обеспечение. Выпуская свое приложение на разные платформы и операционные системы, можно увеличить количество пользователей, а также сократить количество затраченных ресурсов на разработку, потому что приложение нужно разработать всего раз. Для таких целей используют интерпретируемые языки которые не завязаны на конкретной платформе и исполняют код на виртуальной машине. Но их производительность не такая хорошая как на компилируемых языках. Наш выбор при написании приложения пал на язык C++ с использованием фреймворка Qt, на языке C++ при грамотном использовании можно добиться потрясающей производительности при этом используя высокоуровневые конструкции и ООП, но его стандартная библиотека довольно небольшая и простая, поэтому мы прибегли к использованию кроссплатформенного фреймворка в котором есть все, что нужно для разработки сетевого приложения с графическим интерфейсом.

**Цель** **работы –** реализация прикладного программного обеспечения клиент – серверного кроссплатформенного мессенджера.

Задачи, решаемые для достижения цели:

* Проектирование архитектуры клиент – серверного приложения
* Определение модели данных, протокола на которых будет основан клиент сервер
* Непосредственно реализация приложения в программном коде
* Тестирование

# Глава I. Проектирование приложения

## Основные понятия и концепции, используемые при разработке приложения

• Клиент - программа, запущенная на машине пользователя. Клиентские программы предоставляют интерфейс, который позволяет пользователю компьютера запрашивать службы сервера и отображать результаты, возвращаемые сервером.

• Сервер – программа, которая чаще всего находится на удаленной машине, но может быть и на локальной, ждет запросов от клиентов, а затем отвечает на них. В идеале сервер предоставляет клиентам стандартизированный прозрачный интерфейс, чтобы клиенты не знали о специфике системы (то есть аппаратного и программного обеспечения), предоставляющей услугу.

• Параллелизм - возможность разбить программу на части, которые могут выполняться независимо друг от друга. Это означает, что задачи могут выполняться не по порядку, а результат будет таким, как если бы они выполнялись по порядку.

• Кросс-платформенная разработка приложений - это создание одного приложения, которое может работать в разных операционных системах, вместо разработки разных версий приложения для каждой платформы. Кросс-платформенность может быть достигнута с помощью интерпретаторов, которые переводят байтовый код в машинный код, а также фреймворков, которые в рамках высокоуровневых конструкций могут собирать код, подходящий для целевой платформы.

• Фреймворк - это абстракция, реализующая общие функции, каркас для будущего приложения, обычно набор функций классов, которые разработчик может расширять и изменять для достижения желаемого поведения

## Язык С++

При выборе основного инструмента для разработки мы остановились на языке C++. В проекте используется стандарт C++17.

C++ - один из самых популярных языков программирования в мире, его можно найти в современных операционных системах, графических пользовательских интерфейсах и встроенных системах [5]. С++ мультипарадигменный, компилируемым язык со статической типизацией, чаще всего используют объектно-ориентированный подход, что дает четкую структуру программам и позволяет повторно использовать код, снижая затраты на разработку [5]. C++ является переносимым и может использоваться для разработки приложений, которые можно адаптировать к нескольким платформам [5].

## Фреймворк Qt

Qt - кроссплатформенный фреймворк для разработки настольных, встроенных и мобильных приложений. Поддерживаемые платформы включают Linux, Windows, Android, iOS, BlackBerry и другие.

Сам Qt не является языком программирования. Это среда разработки, написанная на C++. Препроцессор, MOC (компилятор метаобъектов), используется для расширения языка C++ такими функциями, как сигналы и слоты. Перед этапом компиляции MOC анализирует исходные файлы, написанные на расширенном Qt C++, и генерирует из них соответствующие стандарту исходные коды C++. Таким образом, сам фреймворк и приложения/библиотеки, которые его используют, могут быть скомпилированы с помощью любого стандартного компилятора C++, такого как Clang, GCC, MinGW и MSVC. Фреймоворк имеет обширную поддержку интернационализации. Qt также предоставляет Qt Quick, который включает декларативный язык сценариев под названием QML, который позволяет использовать JavaScript для обеспечения логики [4]. Благодаря Qt Quick стала возможной быстрая разработка приложений для мобильных устройств, в то время как логика по-прежнему может быть написана с использованием собственного кода для достижения максимально возможной производительности [4].

Другие функции включают доступ к базе данных SQL, синтаксический анализ XML, анализ JSON, управление потоками и поддержку сети [4].

Все классы в Qt наследуются от класса QObject (рисунок 1)

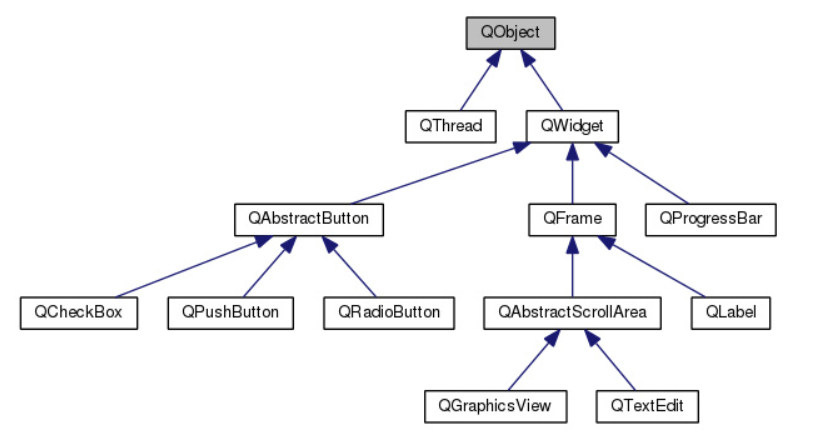


Рисунок 1 – Диаграмма наследования классов в Qt

Краткое описание основных модулей Qt приведено в таблице 1.

Таблица 1

Модули Qt

|  |  |
| --- | --- |
| **Модуль** | **Описание** |
| **Qt Core** | Единственный необходимый модуль Qt, содержащий классы, используемые другими модулями, включая мета-объектную систему, параллелизм и многопоточность, контейнеры, систему событий, плагины и средства ввода-вывода. |
| **Qt GUI** | Центральный модуль графического интерфейса. В Qt 5 этот модуль зависит от OpenGL, но больше не содержит классов виджетов. |
| **Qt Widgets** | Содержит классы для классических приложений с графическим интерфейсом на основе виджетов и классы QSceneGraph. |
| **Qt QML** | Модуль для языков QML и JavaScript. |

Продолжение таблицы 1

|  |  |
| --- | --- |
| **Qt Quick** | Модуль для приложения с графическим интерфейсом, написанный с использованием QML2. |
| **Qt Quick Controls** | Виджет, подобный элементам управления Qt Quick, предназначен в основном для настольных приложений. |
| **Qt Quick Layouts** | Макеты для размещения элементов в Qt Quick. |
| **Qt Network** | Слой сетевой абстракции. В комплекте с поддержкой TCP, UDP, HTTP, TLS, SSL (в Qt 4) и SPDY (начиная с Qt 5.3). |
| **Qt Multimedia** | Классы по работе с аудио, видео, радио и камерой. |
| **Qt Multimedia Widgets** | Виджеты от Qt Multimedia. |
| **Qt SQL** | Содержит классы для интеграции базы данных с использованием SQL. |
| **Qt WebEngine** | Набор Qt Widget и QML webview API на основе Chromium. |
| **Qt Test** | Классы для модульного тестирования приложений и библиотек Qt. |

Хотя приложения, использующие Qt, обычно пишутся на C++, существуют привязки QML к другим языкам. Они не являются частью Qt, но предоставляются различными третьими сторонами. Например, Riverbank Computing предоставляет коммерческие и бесплатные привязки Python для программного обеспечения (PyQt).

Основные модули Qt бесплатные для разработки open-source программного обеспечения. Для открытого исходного кода обычно используется лицензия GPL, которая предоставляют следующие преимущества:

* Свобода запускать программу для любых целей.
* Свобода изучать, как работает программа, и адаптировать ее к конкретным потребностям.
* Свобода распространять копии.
* Свобода улучшать программу и публиковать свои улучшения для всего сообщества.

## Сборка проекта с использованием Qt

Затронем тему сборки с использованием фреймворка Qt. Помимо исходных файлов в проекте помещается файл СMakeLists.txt. Он необходим для вызова утилиты cmake и последующего создания make файлов. Он хранит в себе заранее предусмотренные инструкции, в которых прописывается название проекта, иерархия модулей, утилиты необходимые для сборки проекта [2].

Сравнивая систему сборки C++ с системой сборки Qt, видно, что система сборки C++ (серые прямоугольники) осталась неизменной (рисунок 2). Мы все еще пишем код на C++. Однако мы добавляем больше источников и заголовков (зеленые прямоугольники) (рисунок 2). Здесь задействованы три генератора кода:

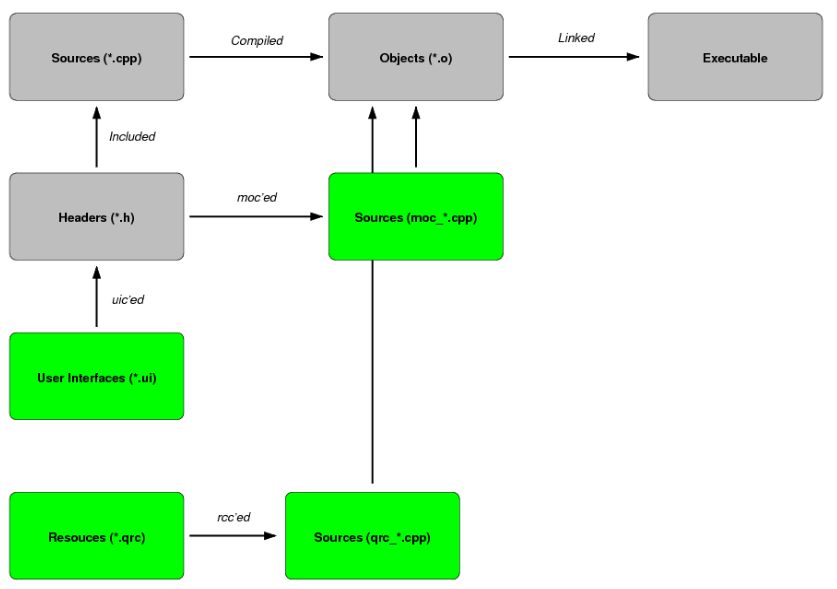


Рисунок 2 – Диаграмма этапов сборки в Qt

Метаобъектная компиляция

Компилятор метаобъектов (moc) принимает все классы, начиная с макроса Q\_OBJECT, и генерирует исходный файл C++ moc\_\*.сpp. Этот файл содержит информацию о моделируемом классе, такую, ​​как имя класса, дерево наследования и т. д., а также реализацию сигналов. Это означает, что, когда вы посылаете сигнал, вы фактически вызываете функцию, сгенерированную moc.

Компиляция пользовательского интерфейса

Компилятор пользовательского интерфейса (uic) берет проекты из Designer и создает файлы заголовков. Эти файлы заголовков затем, как обычно, включаются в исходные файлы, что позволяет вызвать setupUi для создания экземпляра дизайна пользовательского интерфейса.

Компиляция ресурсов

Компилятор ресурсов Qt (rcc) - позволяет встраивать изображения, текстовые файлы и т.д. в исполняемый файл, но при этом иметь доступ к ним как к файлам.

Далее, после компиляции файлов в файлы \*.h и \*.cpp, относящихся к фреймворку Qt, берутся все файлы и производятся стандартные действия по сборке, свойственные для языка C++.

Препроцессинг

Препроцессор работает с одним исходным файлом C++ за раз, заменяя директивы #include содержимым соответствующих файлов (обычно это просто объявления), выполняя замену макросов (#define) и выбирая разные части текста в зависимости от #if, Директивы #ifdef и #ifndef.

Компиляция

Шаг компиляции выполняется на каждом выходе препроцессора. Компилятор анализирует чистый исходный код C ++ (теперь без каких-либо директив препроцессора) и преобразует его в код сборки, создавая фактический двоичный файл в некотором формате (ELF, COFF, a.out, ...). Этот объектный файл содержит скомпилированный код (в двоичной форме) символов, определенных во входных данных. Символы в объектных файлах называются по имени.

Созданные объектные файлы могут быть помещены в специальные архивы, называемые статическими библиотеками, для облегчения повторного использования в дальнейшем.

Компоновка

Компоновщик - это то, что производит окончательный вывод компиляции из объектных файлов, созданных компилятором. Этот вывод может быть либо статической или динамической библиотекой либо исполняемым файлом. Он связывает все объектные файлы, заменяя ссылки на неопределенные символы правильными адресами. Каждый из этих символов может быть определен в других объектных файлах или в библиотеках.

## Механизм сигналов и слотов.

Сигналы и слоты используются для связи между объектами. Данный механизм - центральная черта Qt и именно та часть, которая больше всего отличается от функционала, предоставляемого другими фреймворками.

В программировании с графическим пользовательским интерфейсом, когда мы меняем состояние одного виджета, мы хотим, чтобы уведомлялся другой виджет. К примеру, если пользователь нажимает кнопку «Закрыть», мы хотим, чтобы вызывалась функция окна close(). В старых библиотеках такая связь достигается с помощью функций обратного вызова (callback). Обратный вызов - это указатель на функцию. Если вы хотите, чтобы функция обработки уведомляла вас о каком-либо событии, вы передаете указатель на другую функцию в функцию обработки. Затем функция обработки при необходимости вызывает обратный вызов. Функции обратного вызова имеют два главных недостатка: первое - они небезопасны по типу, второе - обратный вызов сильно связан с функцией обработки [4].

В Qt имеется альтернатива методу обратного вызова – механизм сигналов и слотов (рисунок 3). Сигнал излучается, когда происходит некоторое событие. Слот - это функция, которая вызывается в ответ на определенный сигнал.

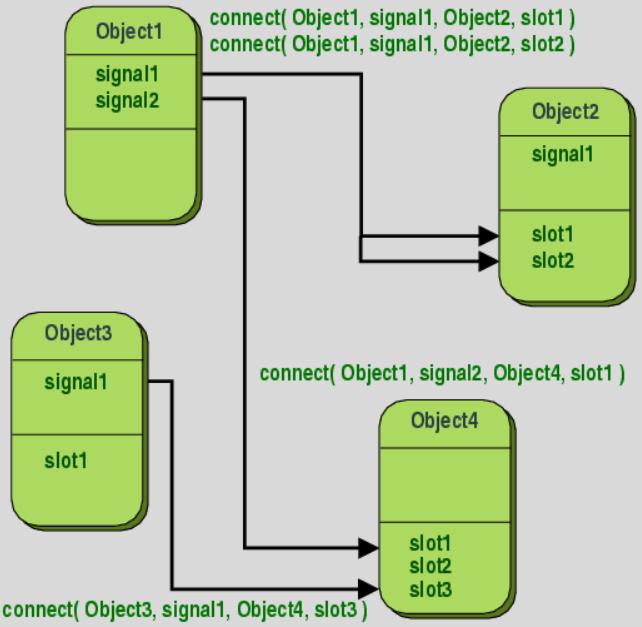


Рисунок 3 – Пример взаимодействия сигналов и слотов

Реализация данного механизма не тривиальна и требует метаобъектного компилятора, упомянутого выше. Если кратко, то метаобъектный компилятор генерирует для каждого специально помеченного класса мета таблицу, с помощью которой во время исполнения можно просматривать атрибуты объекта, то есть возможна рефлексия. У каждого объекта такого класса есть специальный список (рисунок 4), хранящий информацию о слотах, которые необходимо вызвать по излучению сигнала. С помощью такой реализации есть возможность соединять сигналы и слоты во время исполнения программы, что дает большую гибкость приложению.

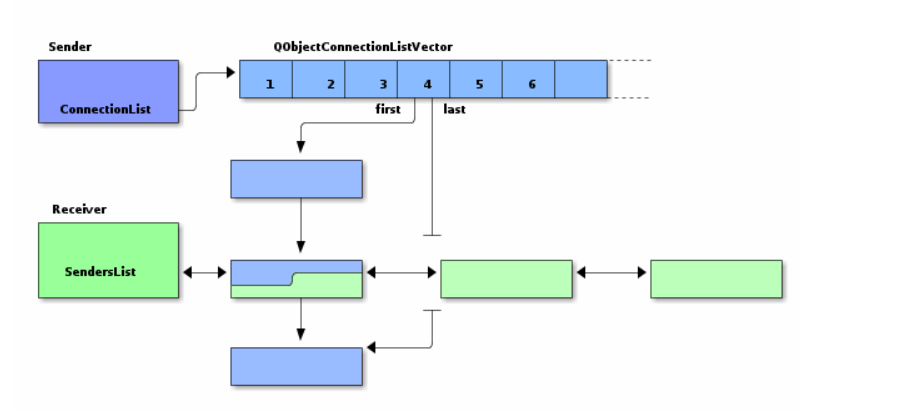


Рисунок 4 – Иллюстрация двусвязного списка хранящего сигналы и слоты

## Архитектура приложения

Архитектура для сетевого приложения заложена классическая, клиент – серверная.

Основные преимущества такого подхода:

1. Централизация. В Peer to Peer архитектуре нет централизованного администрирования, но в архитектуре клиент-сервер есть возможность централизованного управления. Серверы помогают в администрировании всей системы, а также права доступа и распределение ресурсов осуществляется серверами [1].
2. Правильное управление. Поскольку все файлы хранятся в одном месте, управление файлами упрощается.
3. Возможность резервного копирования и восстановления. Сделать резервную копию всех данных легко, поскольку данные хранятся на сервере. Предположим, что произошла какая-то поломка и данные утеряны, их можно легко и эффективно восстановить. В одноранговых вычислениях мы должны делать резервные копии на каждой рабочей станции.
4. Обновление и масштабируемость. Если мы хотим внести изменения, нам нужно будет просто обновить сервер. Кроме того, мы можем добавлять новые ресурсы и системы, внося необходимые изменения на сервере.
5. Доступность. С различных платформ в сети к серверу можно получить удаленный доступ.
6. По мере того, как новая информация загружается в базу данных, у каждой рабочей станции нет необходимости увеличивать собственную емкость хранения (как это может иметь место в одноранговых системах). Все изменения производятся только на центральном компьютере, на котором существует база данных сервера.
7. Безопасность. Правила, определяющие безопасность и права доступа, могут быть определены во время настройки сервера.
8. Серверы могут играть разные роли для разных клиентов.

Общий план архитектуры описываемого Мессенджера представлен на рисунке 5.

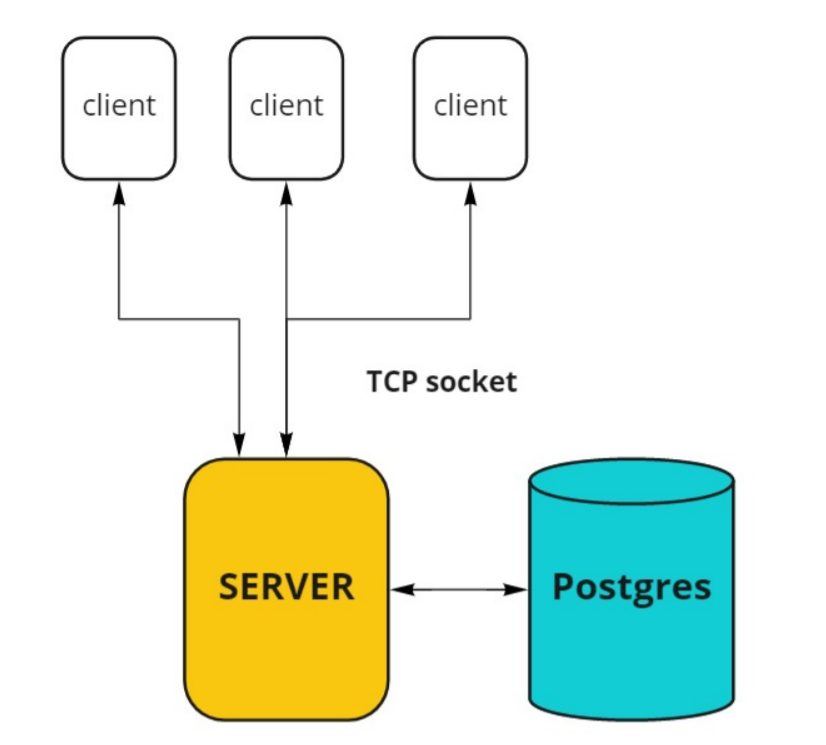


Рисунок 5 – Диаграмма компонентов системы

Сервер работает на удаленной машине и взаимодействует с СУБД PostgreSQL для работы с данными. Клиенты могут исполняться на разных машинах так и на одной. Клиенты не общаются с друг другом на прямую, всё взаимодействие происходит только с сервером. Сервер обрабатывает все события: соединения пользователей, отсоединения, появление сообщений от пользователей и т.д.

# Глава II. Реализация приложения

## План проекта

Создание приложения осуществлялось по следующему плану:

1. Исследование в области реализации подобных сетевых приложений.
2. Проектирование архитектуры.
3. Подготовка инфраструктуры для разработки приложения (настройка системы сборки, системы контроля версий, настройка IDE).
4. Сборка библиотеки для возможности подключения к СУБД Postgres на языке C++.
5. Разработка схемы базы данных для приложения.
6. Разработка протокола на базе JSON для передачи сообщений между клиентом и сервером.
7. Разработка базовой части сервера.
8. Разработка пула потоков для сервера.
9. Разработка пула соединений для сервера.
10. Разработка базовой части клиента.
11. Разработка графического интерфейса клиента.
12. Переход на SSL сокеты для безопасной передачи пакетов в сети.
13. Тестирование системы.

Приложение должно удовлетворять следующим требованиям:

Клиентская часть должна использовать минимальное количество вычислительных ресурсов и занимать минимум памяти на диске, иметь понятный и минималистичный интерфейс. Должна иметься возможность присоединения к каналу с помощью пароля и отправка сообщений в канал. Передача данных должна быть зашифрована.

Серверная частьдолжна состоять из основного консольного приложения и СУБД PostgreSQL, в которой хранятся данные о канале, пользователях. Сервер должен поддерживать одновременное подключение множества клиентов и работу с ними без конфликтов. Сервер должен писать информативные логи, с помощью которых можно узнать по какой причине произошел сбой, смоделировать причину и устранить её. Передача данных должна быть зашифрована.

## Структура проекта

Файловая структура клиентской части представлена на рисунке 6.

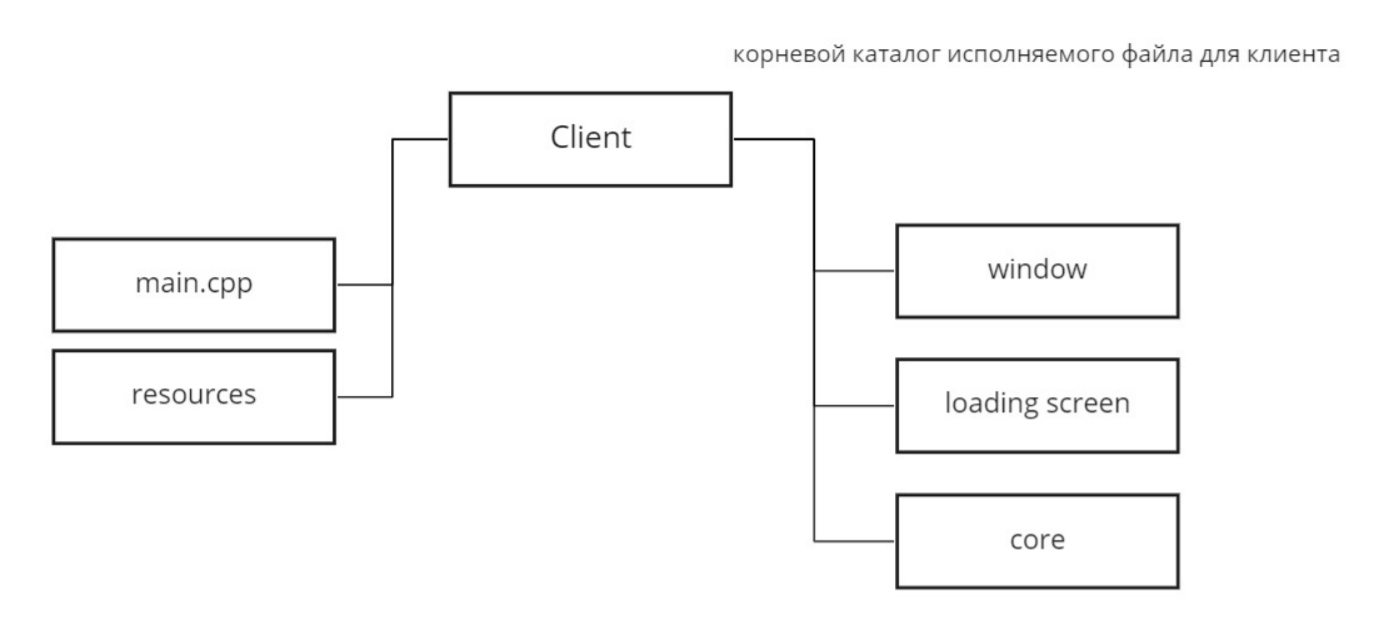


Рисунок 6 - Файловая структура клиентской части

Файловая структура серверной части представлена на рисунке 7.

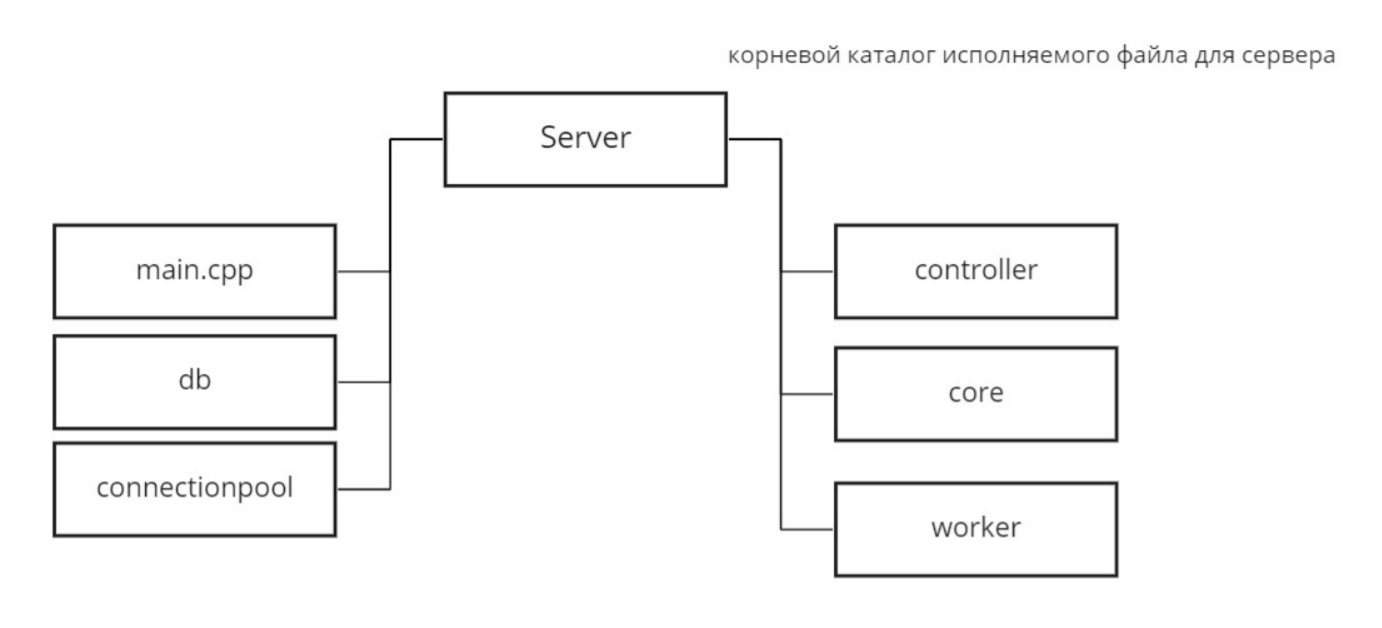


Рисунок 7 - Файловая структура серверной части

Взаимодействие компонентов изображено на рисунке 8.

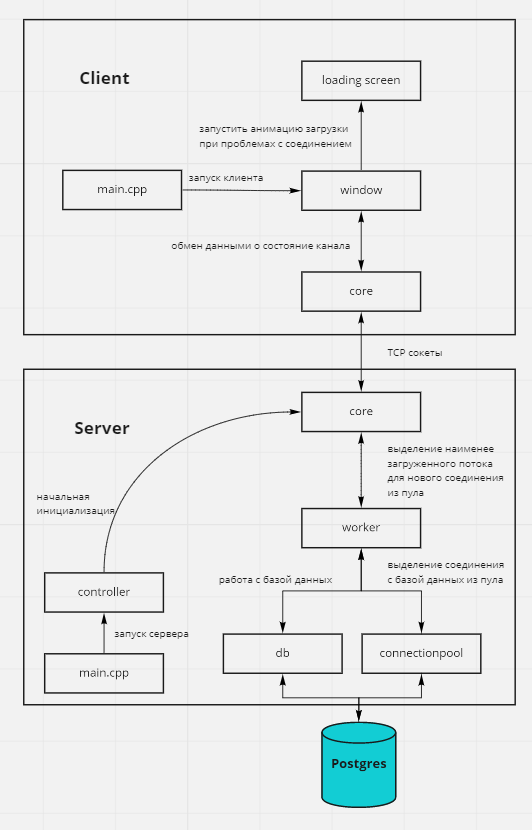


Рисунок 8 - Диаграмма потоков данных

## База данных и взаимодействие с ней

В качестве СУБД выбрана PostgreSQL так как она является самой передовой реляционной базой данных с открытым исходным кодом [3]. Она также доступна почти для любой операционной системы. Для работы с PostgreSQL в Qt используются классы QSqlDatabase и QSqlQuery. Также к серверу присоединяется статическая библиотека libpq.lib которая необходима для работы с Postgres.

Схема базы данных, описанная с помощью ERD диаграммы изображена на рисунке 9.

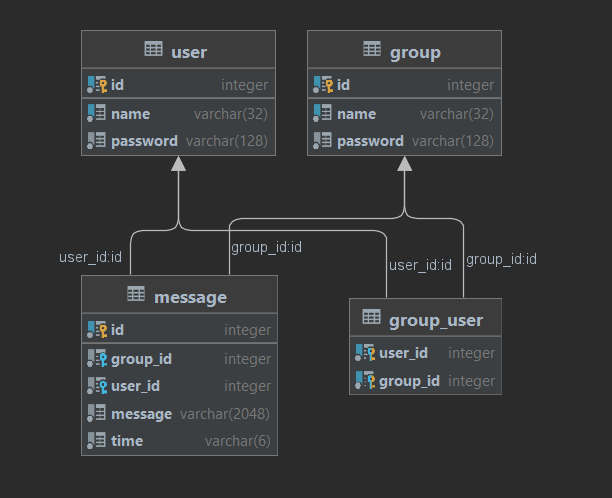


Рисунок 9 - Схема БД

Для эффективной работы с базой данных на стороне сервера был создан класс пула соединений (рисунок 10).

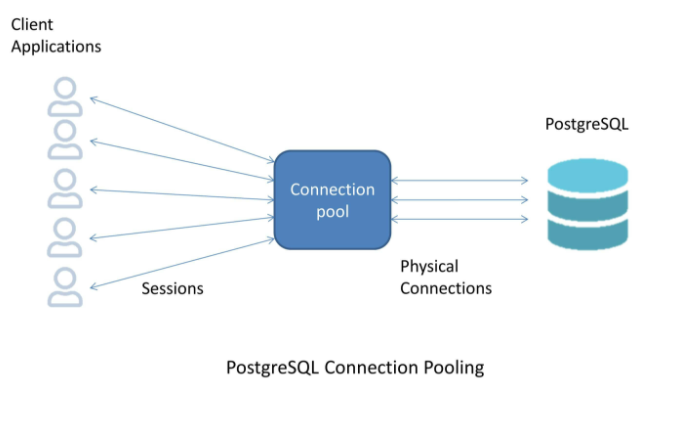


Рисунок 10 - Схема работы с пулом соединений

Характеристики пула соединений с базой данных, которые необходимо достичь при реализации:

* Не нужно знать имя соединения при установлении соединения
* Поддержка многопоточности, чтобы гарантировать, что полученное соединение не используется другими потоками
* Создание соединений по запросу
* Возможность создавать несколько подключений
* Возможность контролировать количество подключений
* Возможность повторно использовать соединения, вместо того, чтобы каждый раз воссоздавать новое соединение
* Автоматическое переподключение после отключения
* Когда соединение недоступно, поток, получающий соединение, должен будет ждать определенное количество времени, чтобы попытаться продолжить получение, и не будет возвращать недопустимое соединение, пока не истечет время ожидания
* Легкое закрытие соединений.

Интерфейс для работы с пулом соединений представлен на таблице 2.

Таблица 2

Открытый интерфейс пула соединений

| метод | код |
| --- | --- |
| Получить соединение | auto conn = ConnectionPool::getConnection(); |
| Освободить соединение | ConnectionPool::releaseConnection(conn); |
| Освободить пул соединений | ConnectionPool::release(); |

Пример использования интерфейса пула соединений в функции добавления сообщения в базу данных (рисунок 11).

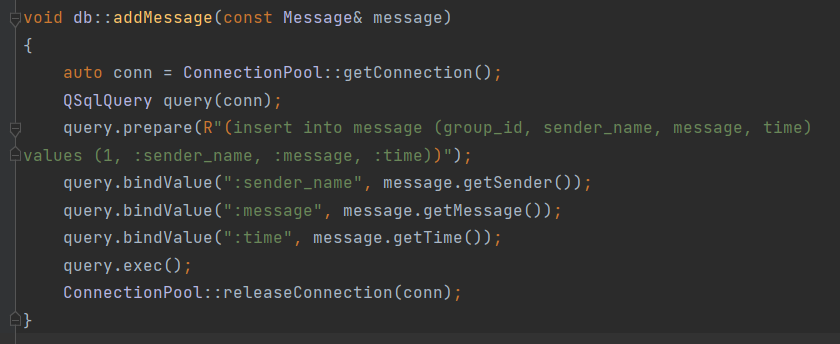


Рисунок 11 - Метод addMessage

## Протокол взаимодействия клиента сервера

Для передачи информации по сети используются TCP сокеты, для которых нужно указать лишь адрес и порт назначения. Но просто сокеты могут передавать лишь байты, что не очень удобно при реализации приложения. Поэтому в качестве формата передачи сообщений был выбран формат JSON. JSON - облегченный формат обмена данными. Людям легко читать и писать c помощью него. Машины также легко анализируют и генерируют его.

JSON состоит из двух структур:

* Коллекция пар имя/значение. На разных языках это реализовано как объект, запись, структура, словарь, хеш-таблица, список с ключами или ассоциативный массив.
* Упорядоченный список значений. В большинстве языков это реализовано как массив, вектор, список или последовательность.

Для передачи сообщений между клиентом и сервером формируется JSON структура, у которой могут быть следующие типы и данные внутри сообщения (рисунок 12).



Рисунок 12 - Типы сообщений

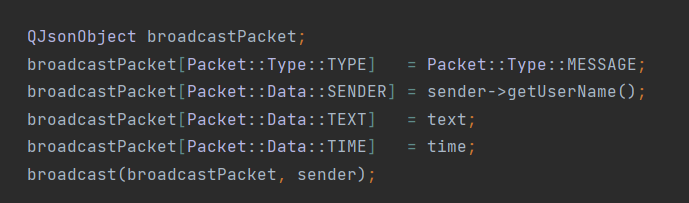
Пример формирования JSON сообщения и отправки его всем пользователям в группе (рисунок 13).

Рисунок 13 - Код формирования сообщения

После формирования сообщения оно сериализуется и отправляется по сети через TCP сокет (рисунок 14).

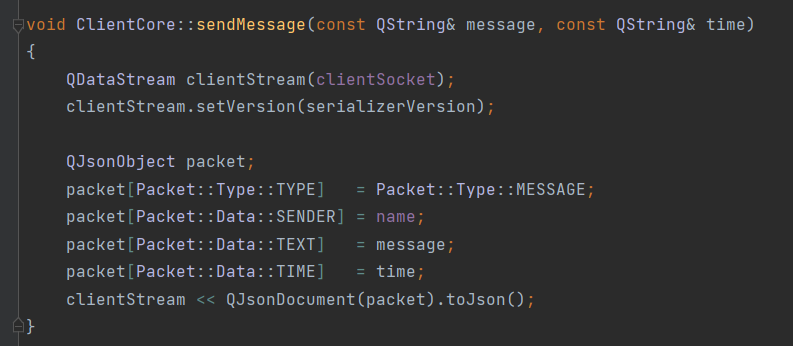


Рисунок 14 - Метод sendMessage

Парсинг структуры рассмотренной выше (рисунок 15).

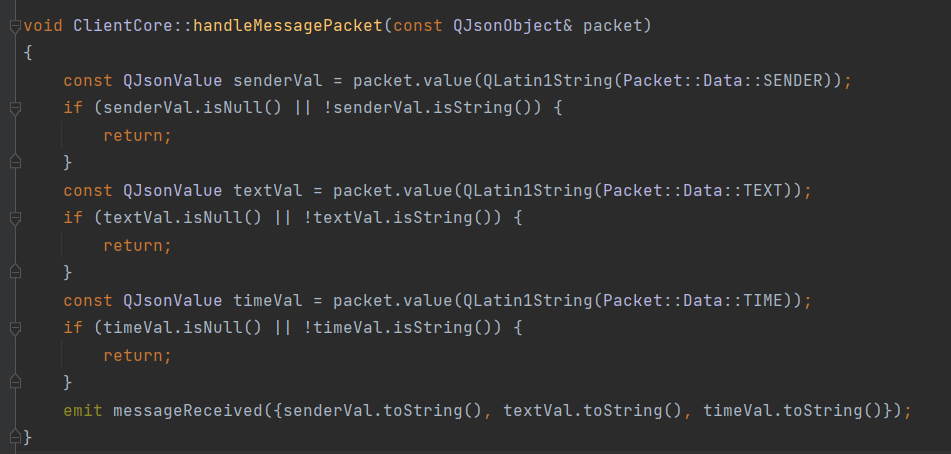


Рисунок 15 - Метод handleMessagePacket

Структура выше отображается в класс Message (рисунок 16), который содержит имя отправителя, непосредственно сообщение и время отправки сообщения.

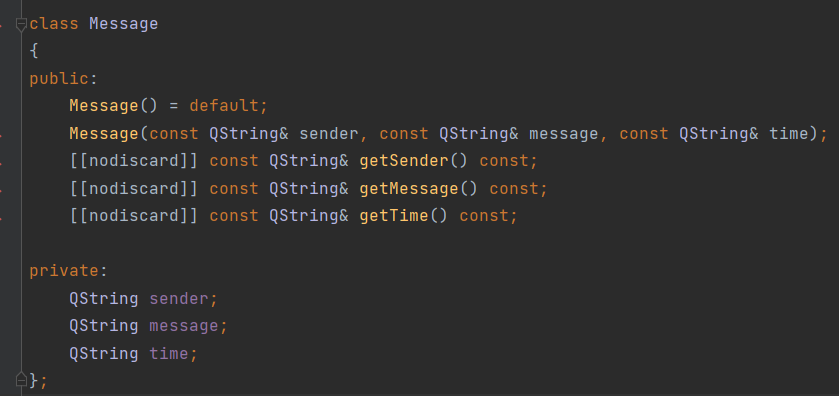


Рисунок 16 - Класс Message

## Реализация Сервера

Основным классом является класс ServerCore (рисунок 17), который наследуется от класса QTcpServer. Используется перегрузка виртуального метода *incomingConnection* для того, чтобы удобно обрабатывать входящие подключения.

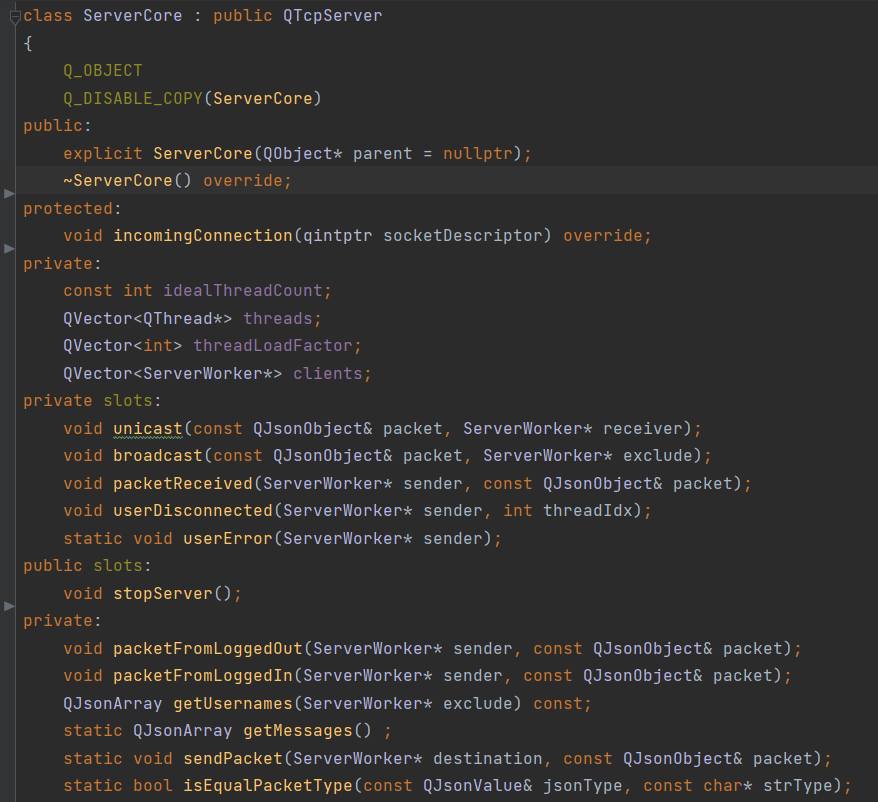


Рисунок 17 - Класс ServerCore

Запускается основной модуль из объекта класса ServerController (рисунок 18).

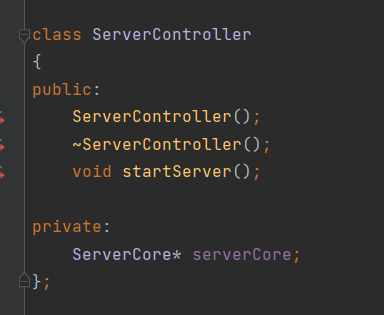


Рисунок 18 - Класс ServerCore

Основная работа ServerCore это обработка событий от клиентов (соединение, прием сообщения).

В классе содержится пул потоков *threads* (рисунок 19) для присоединяющихся клиентов. Как только присоединяется новый клиент под него выделяется наименее загруженный поток из пула. При старте пул пустой - потоки не выделены, потом, при присоединении новых клиентов, создаются новые потоки до ограничения *threadLoadFactor,* который задается с помощью функции *QThread::idealThreadCount.*

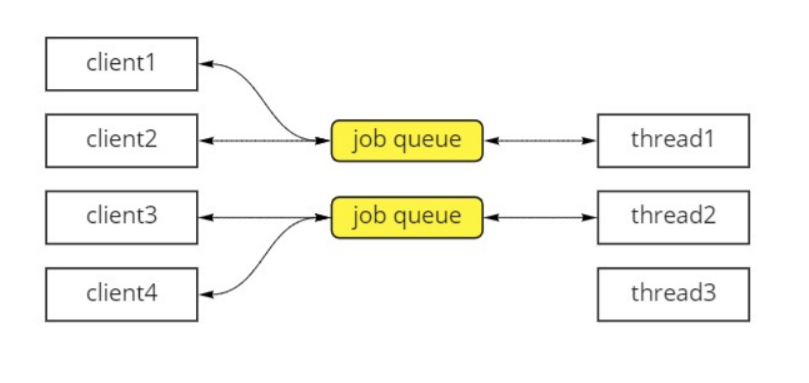


Рисунок 19 - Схема работы пула потоков

Под каждого клиента в сервере выделяется сокет, который хранится в объекте класса ServerWorker, в свою очередь worker работает в потоке, который выделился из пула потоков.

Объявление класса ServerWorker (рисунок 20).

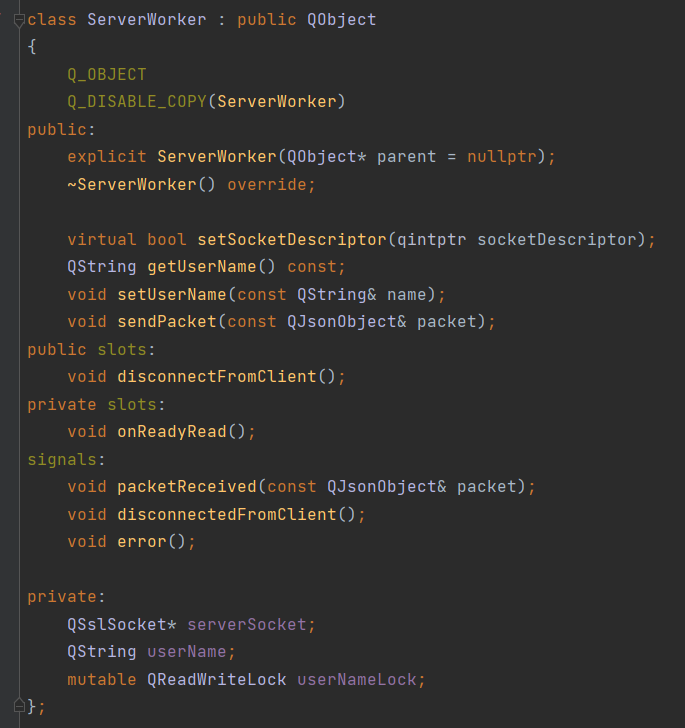


Рисунок 20 - Класс ServerWorker

Прослушивание входящих пакетов в виде JSON структуры происходит в методе onReadyRead (рисунок 21).

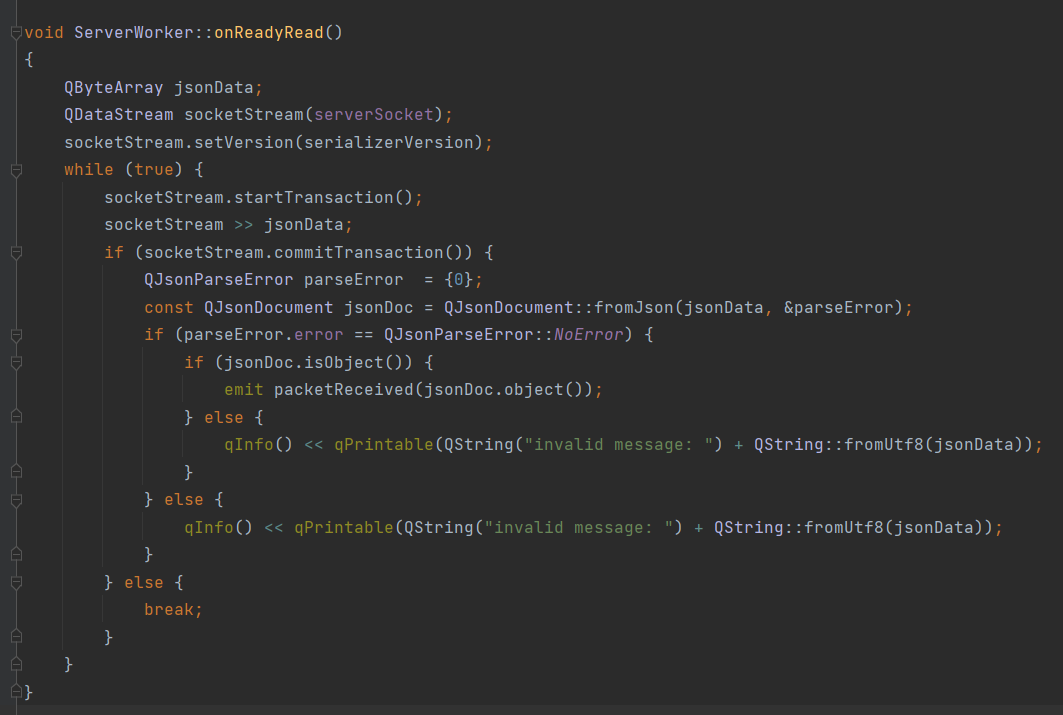


Рисунок 21 - Метод onReadyRead

Далее после получения полноценной структуры она отправляется дальше необходимому обработчику на основе типа сообщения.

## Реализация Клиента

Клиентская часть разделена на два основных класса. За графический интерфейс и взаимодействие с пользователем отвечает класс ClientWindow, за общение с сервером отвечает класс ClientCore.

Класс ClientWindow (рисунок 22) наследуется от класса QWidget и имеет графический интерфейс, который реализован в ui формате, который компилируется в h файл с помощью компилятора uic.

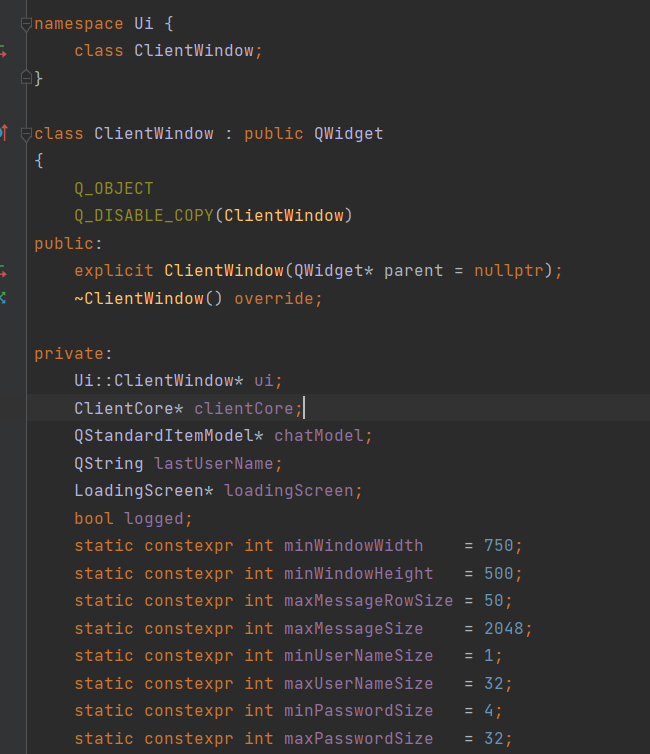


Рисунок 22 - Класс ClientWindow

Ui файл (рисунок 23) форматом похож на XML.

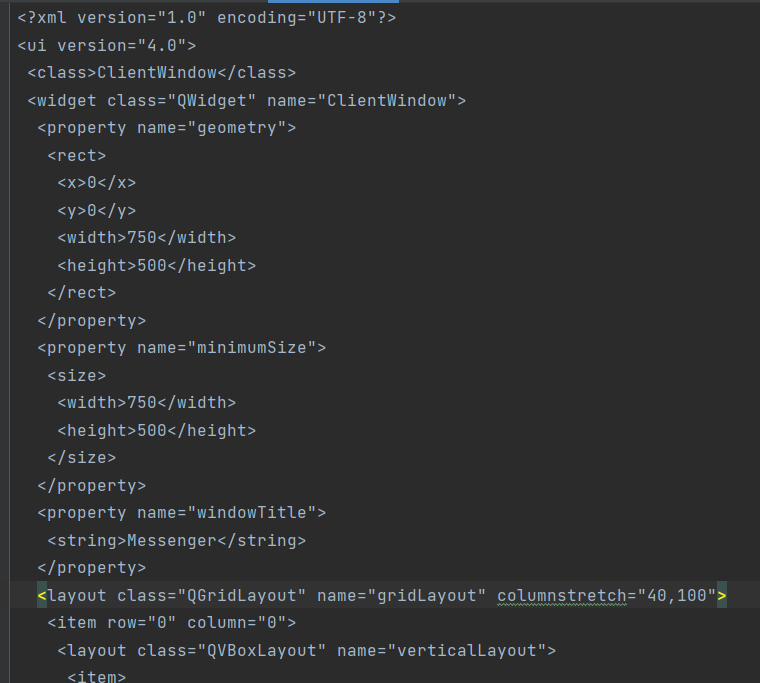


Рисунок 23 - Ui код

Графический интерфейс, который получается с помощью такого файла представлен на рисунке 24.

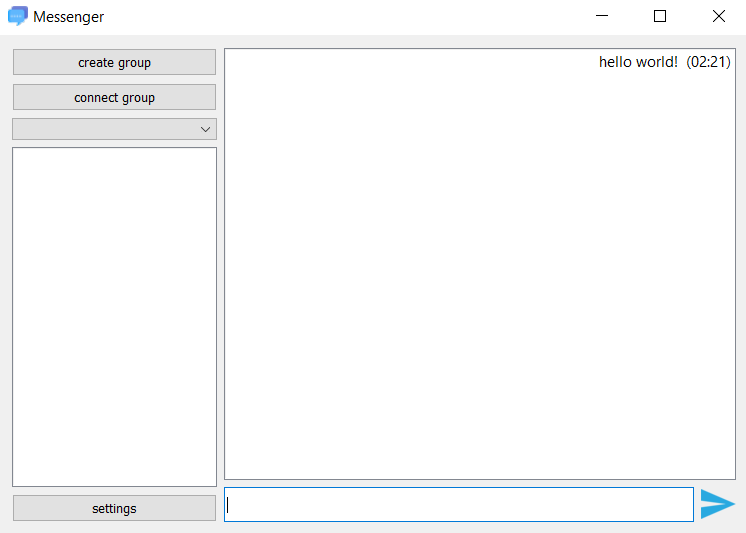


Рисунок 24 - Главное окно приложения

Класс ClientCore (рисунок 25) взаимодействует с сервером для обмена сообщениями, а также с классом ClientWindow, чтобы отображать состояние в графическом интерфейсе.

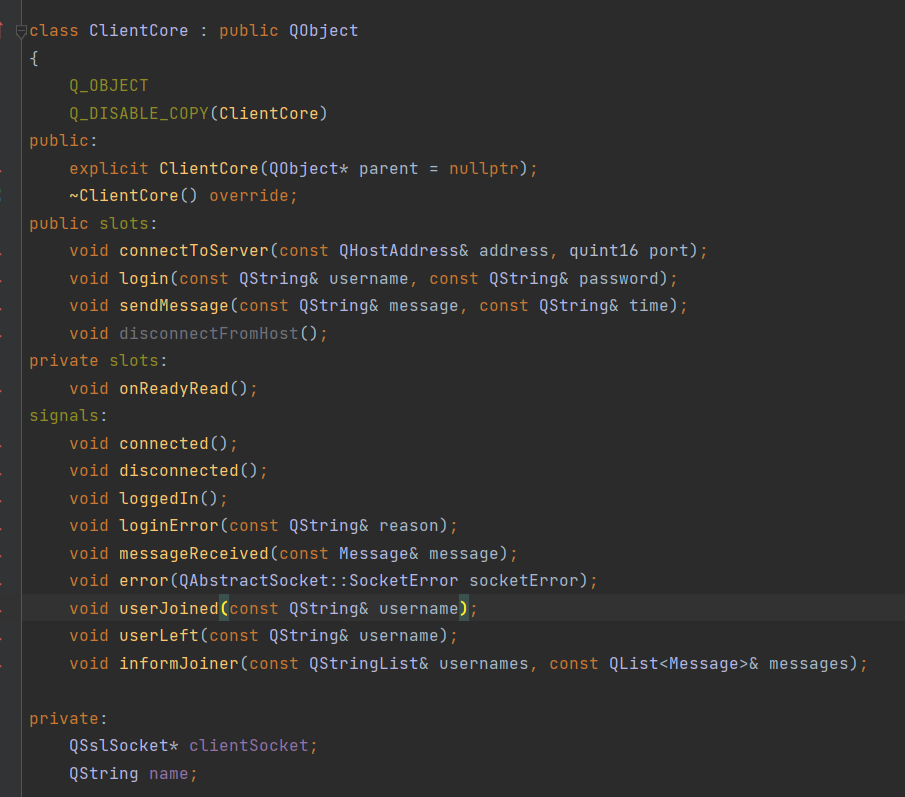


Рисунок 25 - Класс ClientCore

Продемонстрируем работу приложения:

Если нет соединения с сервером, клиент будет пытаться присоединиться к серверу, и будет гореть окно с анимацией загрузки (рисунок 26).

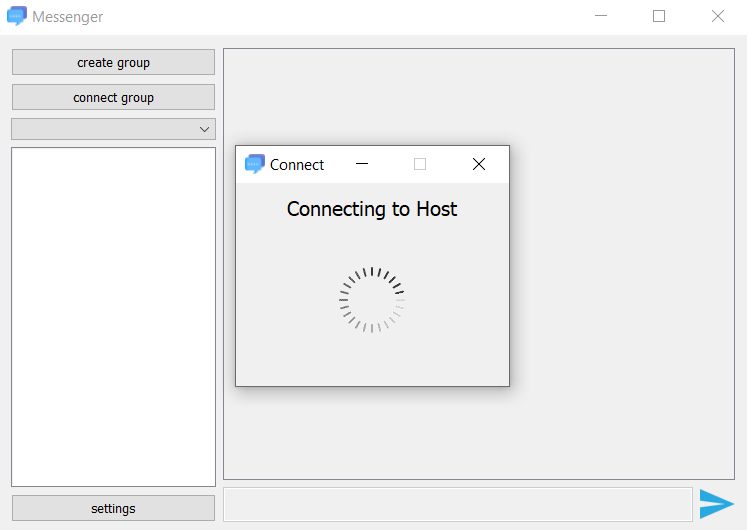


Рисунок 26 - Окно загрузки

После того как соединение прошло, нужно указать свое имя и пароль группы (рисунок 27).

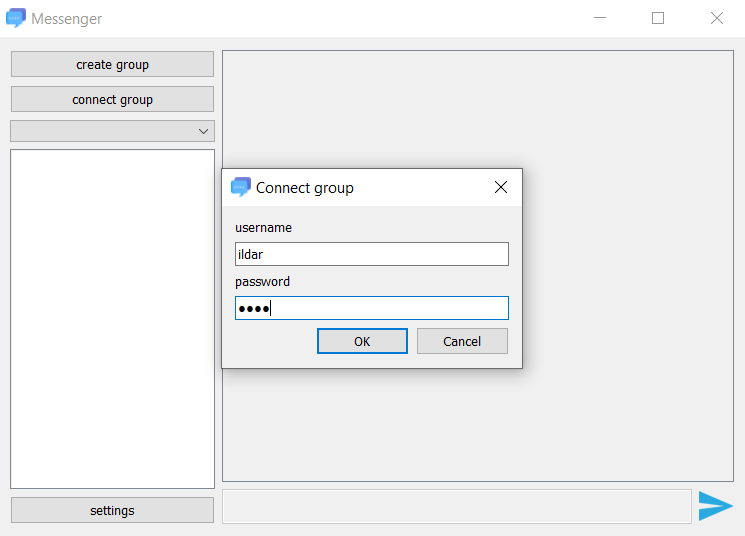


Рисунок 27 - Окно авторизации

Если пароль неверный, приложение предупредит об этом пользователя и предложит ввести реквизиты повторно (рисунок 28).

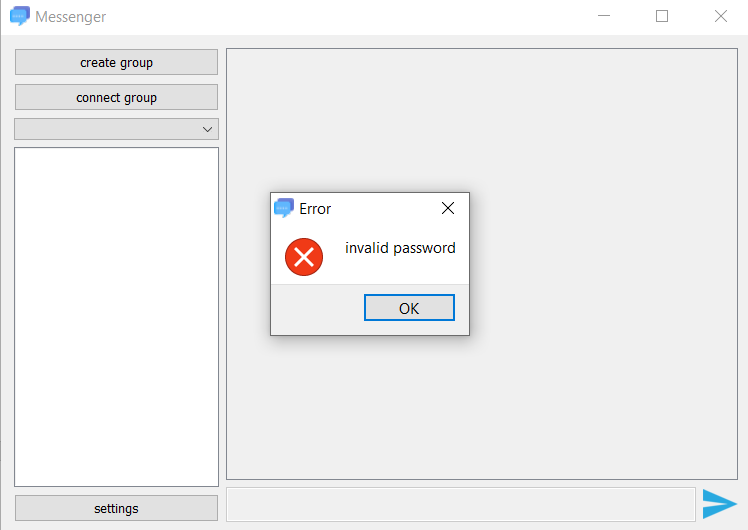


Рисунок 28 - Окно ошибки

Если пароль верный, у нас открывается доступ к чату (рисунок 29).

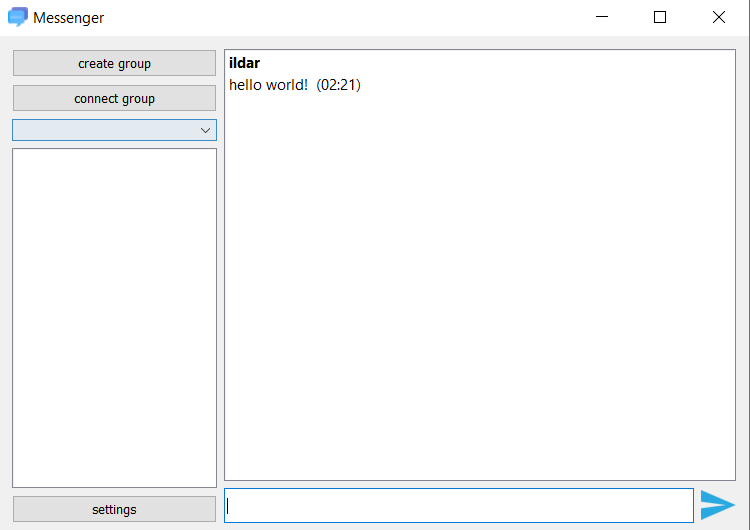


Рисунок 29 - Окно с чатом

Откроем еще несколько клиентов для симуляции активного общения.

При присоединении нового пользователя, имя пользователя, который онлайн, появляется в боковом виджете. Также в чате видно, что присоединился новый пользователь (рисунок 30).

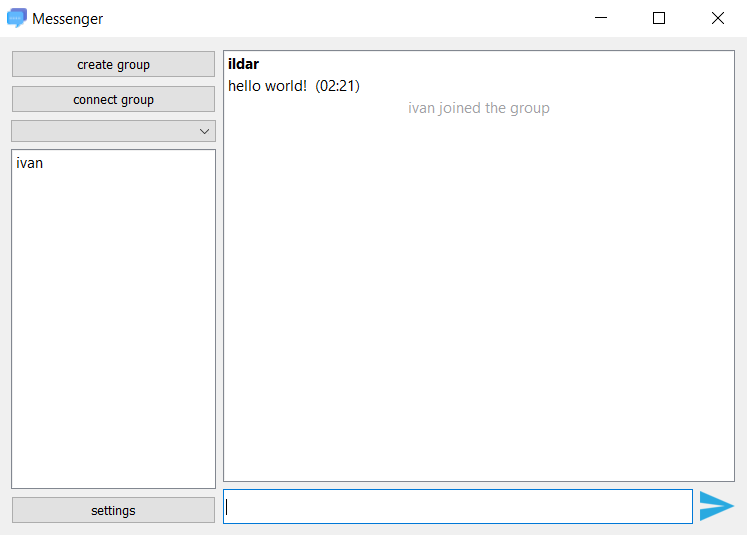


Рисунок 30 - Окно с чатом

Общение нескольких клиентов продемонстрировано на рисунке 31.

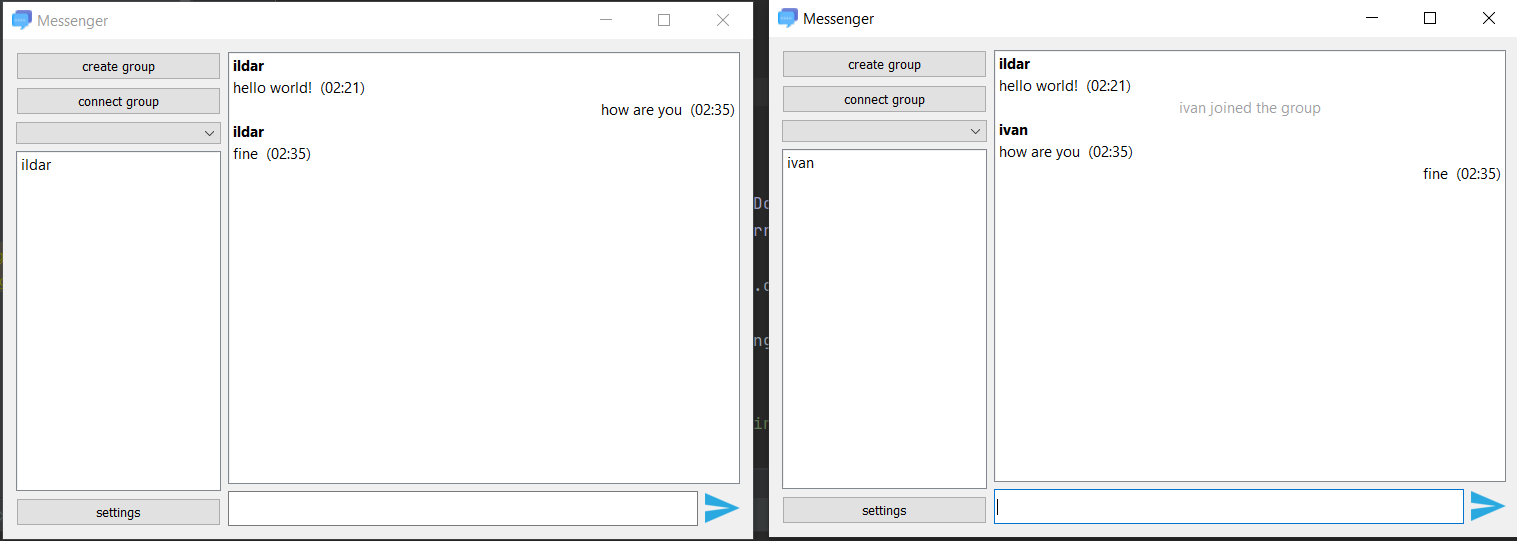


Рисунок 31 - Общение нескольких клиентов

Общение большого количества клиентов продемонстрировано на рисунке 32.

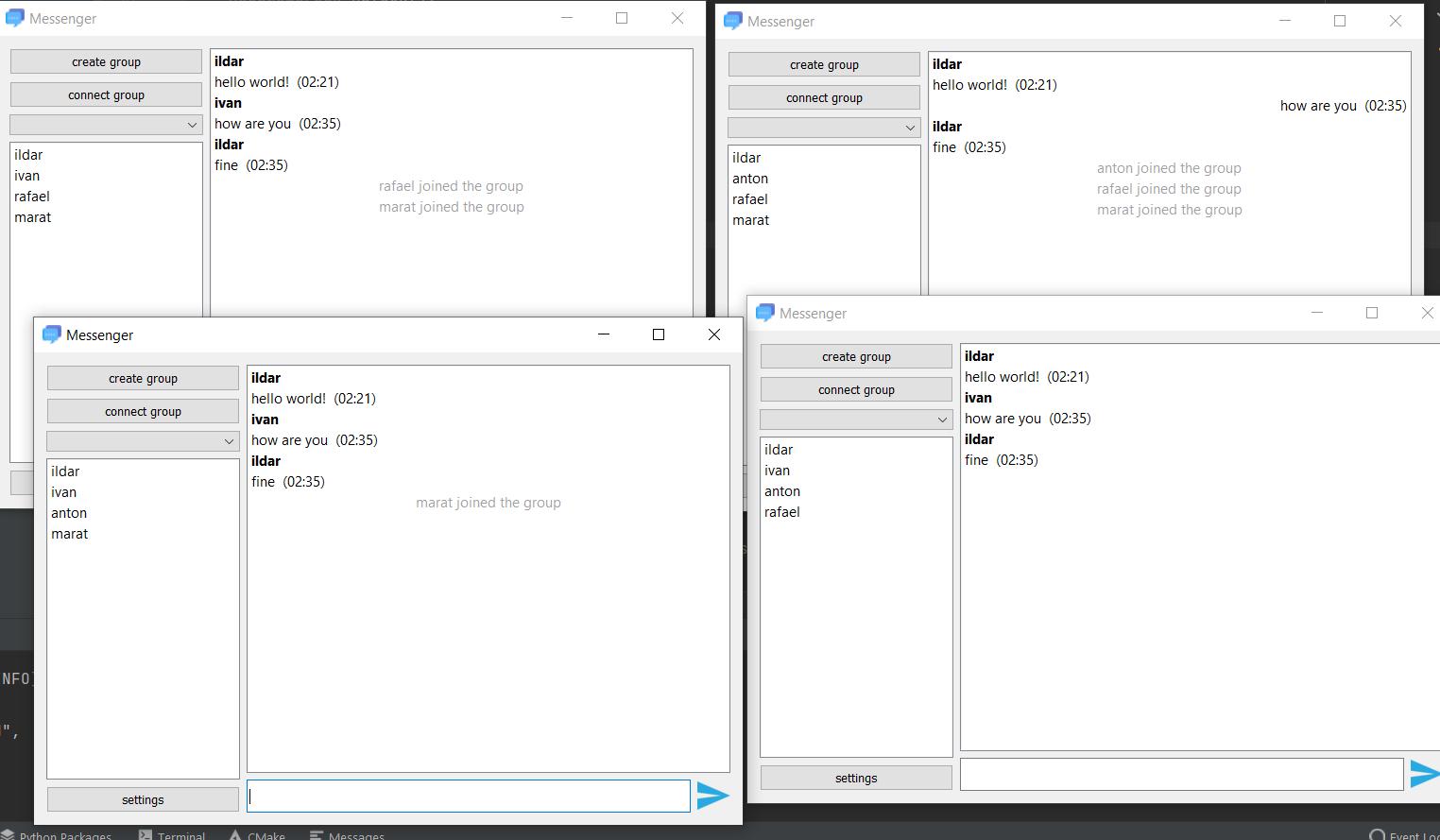


Рисунок 32 - Общение множества клиентов

При отсоединении пользователя его имя исчезает с боковой панели других клиентов. Также в чате появляется сообщение о данном событии (рисунок 33).

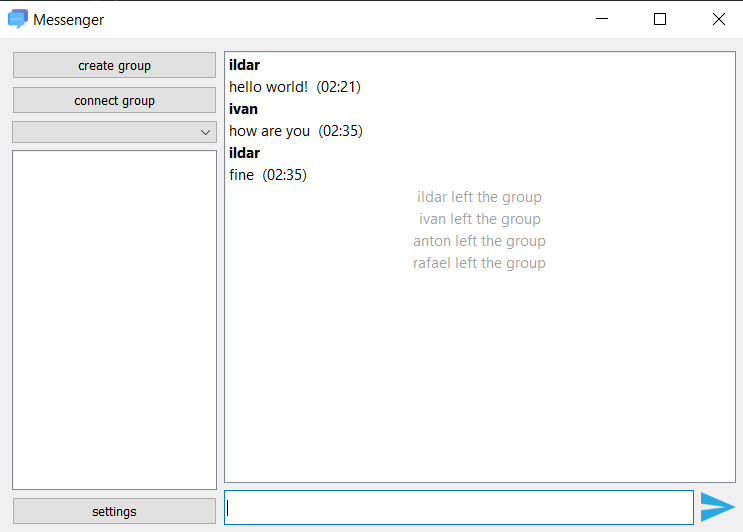


Рисунок 33 - Окно чата

При заходе клиента в чат у него появляется вся история прошлых сообщений (рисунок 34).

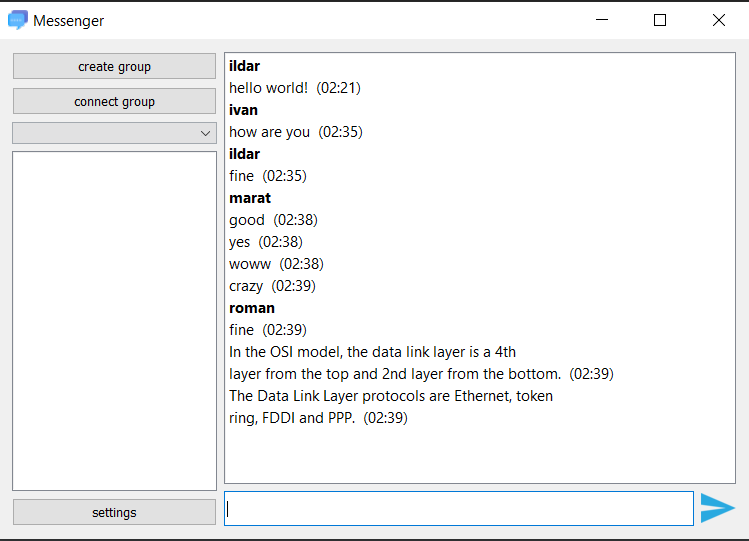


Рисунок 34 - Окно чата

# Заключение

В курсовой работе было сделано: исследование в области реализации сетевых приложений, разработка схемы базы данных для приложения, разработка протокола на базе JSON для передачи сообщений между клиентом и сервером, разработка сервера и клиента, разработка пула потоков и пула соединений. В процессе проектирования и разработки использовались проверенные временем концепции и методологии. Данную программу можно развивать и дальше, добавив новые функциональные возможности.

Таким образом, все поставленные задачи были решены, цель курсовой работы достигнута.