Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Операционные системы и системное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему

РАЗРАБОТКА МНОГОПОТОЧНОГО TCP СЕРВЕРА

БГУИР КП 6-05-0611-05 220 ПЗ

Студент Д.А. Манухо

Руководитель Д.А. Жалейко

МИНСК 2025

Учреждение образования

«Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой

\_\_\_\_\_\_\_\_ Б.В.Никульшин

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту Манухо Даниилу Алексеевичу

1. Тема проекта: Разработка многопоточного TCP сервера.

2. Срок сдачи студентом законченного проекта: *10 мая 2025 г.*

3. Исходные данные к проекту: *язык программирования C*

4. Содержание расчетно—пояснительной записки (перечень вопросов, которые

*Введение. 1. Обзор литературы. 2. Постановка задачи. 3. Системное проектирование. 4. Функциональное проектирование. 5. Разработка программных модулей. 6. Результат работы. Заключение. Список использованных источников. Приложения*

5. Перечень графического материала (с точным обозначением обязательных

чертежей и графиков):

1. *Схема структурная*
2. *Диаграмма классов*
3. *Ведомость документов*

6. Консультант по проекту *Жалейко Д. А.*

7. Календарный график работы над проектом на весь период проектирования

(с обозначением сроков выполнения и трудоемкости отдельных этапов):

*раздел 1, 2 к 06.03 — 5 %;*

*разделы 3, 4 к 07.04 — 30 %;*

*разделы 5, 6, 7, 8, 9 к 05.05 — 90 %;*

*оформление пояснительной записки и графического материала к 10.05 — 100%*

*Защита курсового проекта с 28.05 по 10.06.*

РУКОВОДИТЕЛЬ  *Д. А. Жалейко*

Задание принял к исполнению  *Д. А. Манухо*

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ ………………………………………………………………………4

1 ОБЗОР ЛИТЕРАТУРЫ ………………………………………………………..5

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ ………………………………………..7

3 ФУНКЦИОНАЛЬНОЕ ПРОКТИРОВАНИЕ ………………………………..9

4 РАЗРАБОТКА ПРОГРАМНЫХ МОДУЛЕЙ ……………………………….12

[4.1 Разработка схем алгоритмов 12](#_Toc199442820)

[4.2 Разработка алгоритмов 12](#_Toc199442821)

[5 РЕЗУЛЬТАТЫ РАБОТЫ 14](#_Toc199442822)

[6 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 18](#_Toc199442823)

[ПРИЛОЖЕНИЕ А 19](#_Toc199442824)

[ПРИЛОЖЕНИЕ Б 33](#_Toc199442825)

[(Структурная схема программы) 33](#_Toc199442826)

[ПРИЛОЖЕНИЕ В 34](#_Toc199442827)

[(Блок-схема алгоритма main()) 34](#_Toc199442828)

[ПРИЛОЖЕНИЕ Г 35](#_Toc199442829)

[(Блок-схема алгоритма handle\_client()) 35](#_Toc199442830)

[ПРИЛОЖЕНИЕ Д 36](#_Toc199442831)

[(Ведомость документов) 36](#_Toc199442832)

**ВВЕДЕНИЕ**

Сетевые технологии играют важную роль в современной цифровой инфраструктуре, обеспечивая взаимодействие между компьютерами, мобильными устройствами и серверами в локальных и глобальных сетях. Одним из основополагающих элементов таких взаимодействий являются серверные приложения, принимающие и обрабатывающие клиентские запросы по сети. Среди наиболее распространённых протоколов передачи данных — TCP (Transmission Control Protocol), обеспечивающий надёжную доставку сообщений и гарантированную последовательность передачи.

Серверные приложения на базе TCP широко используются в различных сферах: от мессенджеров и многопользовательских игр до систем удалённого администрирования и обмена файлами. Эффективная реализация TCP-сервера требует понимания системного программирования, работы с сокетами, многозадачности и управления ресурсами операционной системы.

Данный курсовой проект посвящён разработке многопоточного TCP-сервера на языке программирования C. Целью проекта является создание надёжного серверного приложения, способного обрабатывать множественные клиентские подключения одновременно, обеспечивая устойчивость, масштабируемость и минимальное время отклика. Особое внимание в работе уделено реализации многопоточности с использованием POSIX-потоков (pthreads), организации безопасной работы с памятью и корректному завершению потоков и соединений.

Проект может быть использован в качестве основы для создания различных клиент-серверных систем, а также служит учебным примером разработки низкоуровневых сетевых приложений с учётом особенностей взаимодействия с операционной системой.

**1 ОБЗОР ЛИТЕРАТУРЫ**

* 1. **Сервер как проект**

Любое клиент-серверное приложение, в том числе и сетевая игра, состоит из двух основных компонентов: серверной части и клиентской. Данный курсовой проект ориентирован на реализацию серверной части, обеспечивающей стабильное взаимодействие между клиентами.

TCP-сервер выполняет ключевую роль посредника и координатора в сетевом взаимодействии, обеспечивая:

* Установление и поддержание соединений с множеством клиентов одновременно.
* Получение и интерпретацию входящих сетевых пакетов.
* Передачу ответных пакетов, отражающих текущее состояние игровой или логической сессии.
* Поддержание целостности и синхронизации данных между подключёнными клиентами.

Для этого сервер использует механизмы многопоточности, позволяющие обрабатывать соединения параллельно, тем самым обеспечивая масштабируемость и отзывчивость.

* 1. **Анализ существующих аналогов**

В рамках сетевого программирования и игровых серверов существует множество реализаций, которые послужили источником вдохновения для построения архитектуры многопоточного сервера. Ниже рассмотрены два таких проекта, демонстрирующие подходы к сетевой логике и взаимодействию с клиентами.

**1.2.1 Minecraft Server**

Сервер Minecraft — один из самых известных игровых серверов, обеспечивающих мультиплеерную среду. Он управляет состоянием мира, обрабатывает перемещения игроков, действия с объектами и синхронизирует все изменения между участниками. Особенности:

* Поддержка множества клиентов одновременно.
* Асинхронная передача сообщений.
* Обработка коллизий и игровых событий.
* Простота масштабирования благодаря модульности.

**1.2.2 Netcat и учебные TCP-серверы**

Простые реализации TCP-серверов, такие как netcat и учебные многопоточные серверы, иллюстрируют базовую суть TCP-соединений. Они дают понимание системных вызовов, работы с сокетами, многопоточности через pthreads и синхронизации ресурсов.

* 1. **Анализ использованных средств разработки**

В рамках проекта в качестве основного языка программирования выбран язык C, благодаря своей высокой производительности и низкоуровневому доступу к системным ресурсам.

Для реализации многопоточности применяется POSIX-библиотека pthread, позволяющая:

* Создавать и управлять потоками.
* Использовать мьютексы и условные переменные для синхронизации.
* Минимизировать конфликты при доступе к разделяемым ресурсам.

Работа с сетевыми соединениями осуществляется с использованием стандартных UNIX-сокетов: socket(), bind(), listen(), accept(), recv(), send() и др. Такой подход даёт полное понимание принципов сетевого взаимодействия на низком уровне и делает проект кроссплатформенным.

* 1. **Постановка задачи**

Анализ аналогов показывает, что несмотря на различие по сложности и функциональности, все TCP-серверы следуют базовым принципам приёма, обработки и ответа на запросы клиентов. Цель данной работы — разработка многопоточного TCP-сервера, способного обрабатывать множественные подключения одновременно.

Основные задачи проекта:

* Инициализация TCP-соединений и прослушивание заданного порта.
* Обработка клиентских запросов в отдельных потоках.
* Синхронизация доступа к общим ресурсам (например, состоянию сессии).
* Безопасное завершение потоков и закрытие соединений.
* Обработка сетевых ошибок и исключительных ситуаций.
* Логирование активности клиентов и основных событий сервера.

Этот проект служит основой для построения устойчивой серверной архитектуры и может быть расширен в сторону поддержки более сложных протоколов, авторизации клиентов и взаимодействия с базами данных.

**2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ**

Проект реализован на основе модульной архитектуры, при которой серверное приложение разделено на независимые компоненты, каждый из которых выполняет строго определённую функцию. Такой подход обеспечивает:

* Повышенную читаемость и удобство сопровождения кода;
* Изоляцию функциональных блоков, что снижает количество потенциальных ошибок;
* Возможность масштабирования и модификации отдельных компонентов без затрагивания всей системы.
* Модульная структура китически важна для серверных решений, так как позволяет эффективно отлаживать, расширять и оптимизировать систему при росте нагрузки или функционала. Ниже описаны основные модули TCP-сервера и их взаимодействие.

**2.1 Модуль инициализации и запуска сервера**

Этот компонент отвечает за начальную настройку сервера и запуск его основных подсистем. Он создаёт TCP-сокет, настраивает его и привязывает к указанному порту; запускает цикл прослушивания входящих соединений; Инициализирует основные структуры и системные ресурсы (например, мьютексы, логирование); Обрабатывает системные сигналы для корректного завершения работы. Фактически, это точка входа в приложение, управляющая запуском и завершением всех процессов сервера.

**2.2 Модуль обработки клиентских подключений**

Основная задача модуля — приём входящих соединений с использованием системного вызова accept() и передача управления каждому клиенту в отдельном потоке. Он создаёт новый поток (pthread\_create) для каждого подключившегося клиента; поддерживает пул активных соединений;

Контролирует корректное завершение и очистку ресурсов после завершения соединения. Модуль обеспечивает масштабируемость сервера и его способность обрабатывать множество клиентов параллельно.

**2.3 Модуль обработки сообщений**

Каждый поток, связанный с клиентом, использует данный модуль для интерпретации и обработки входящих сообщений. Этот компонент:

* Читает данные от клиента (recv);
* Проводит разбор протокола (в простейшем виде — строки команд);
* Вызывает соответствующие функции (например, эхо-ответ, передача файла, работа с базой данных, если применимо);
* Отправляет ответ клиенту (send).

Данный модуль логически изолирован от сетевого взаимодействия и может быть адаптирован под любой прикладной протокол.

**2.4 Модуль синхронизации и безопасности данных**

Поскольку сервер работает в многопоточном режиме, важную роль играет контроль доступа к разделяемым ресурсам. Этот модуль реализует:

Мьютексы (pthread\_mutex\_t) для защиты разделяемых структур (например, логов или общего хранилища состояния);

Условные переменные (pthread\_cond\_t) при необходимости синхронизации между потоками;

Минимизацию гонок и предотвращение взаимных блокировок.

Он обеспечивает корректную и безопасную работу всех параллельных компонентов сервера.

**2.5 Модуль обработки команд**

Для работы с сервером реализован простой командный режим. С помощью него клиенты могут взаимодействовать с сервером с помощью простых команд. Чтобы перейти в режим работы с командами, клиент должен аутентифицироваться, прежде чем перейти к работе с сервером.

**2.6 Модуль обработки файлов**

С помощью этого модуля можно загружать и выгружать различные данные(файлы) на сервер и с сервера. Это делает удобным использование его вместо стандартного ftp сервера. Чтобы перейти в режим работы с файлами, клиент должен аутентифицироваться, прежде чем перейти к работе с сервером.

**3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ**

В данном разделе описываются основные структуры данных, используемые при разработке многопоточного TCP-сервера с поддержкой чата, выполнения команд и передачи файлов. Эти структуры определяют способ хранения состояния клиентов, управления соединениями и обработки данных.

**3.1 Структуры клиентских сессий**

Для управления состоянием каждого подключенного клиента используется структура Client, которая содержит:

typedef struct {

int socket; // Сокет соединения с клиентом

char username[MAX\_USERNAME\_LEN]; // Имя пользователя (для чата)

bool authenticated; // Флаг аутентификации

} Client;

Для управления всеми подключенными клиентами используется структура ClientList:

typedef struct {

Client\* clients[MAX\_CLIENTS]; // Массив указателей на клиентов

int count; // Текущее количество клиентов

pthread\_mutex\_t mutex; // Мьютекс для потокобезопасности

} ClientList;

Ключевые функции для работы с клиентами:

* add\_client() - добавление нового клиента
* remove\_client() - удаление клиента
* broadcast\_message() - рассылка сообщений всем клиентам чата

**3.2 Структура сервера**

Сервер использует следующие глобальные переменные и константы:

#define PORT 8080 // Порт по умолчанию

#define MAX\_CLIENTS 100 // Максимальное число клиентов

#define BUFFER\_SIZE 1024 // Размер буфера сообщений

#define MAX\_USERNAME\_LEN 50 // Макс. длина имени пользователя

#define FILE\_DIR "files/" // Директория для файлов

Основные компоненты сервера:

* Главный сокет (server\_socket) для прослушивания подключений
* Пул потоков для обработки клиентов
* Глобальная структура client\_list для хранения подключений
* Мьютексы для синхронизации доступа к общим ресурсам

**3.3 Режимы работы сервера**

Сервер поддерживает три основных режима работы:

1. Чат:
   * Хранение истории сообщений в структуре:

typedef struct {

char message[BUFFER\_SIZE];

char username[MAX\_USERNAME\_LEN];

time\_t timestamp;

} ChatMessage;

* + Функции:
    - handle\_chat\_mode() - обработка чата
    - send\_chat\_history() - отправка истории новым клиентам

1. Выполнение команд:
   * Использование popen() для выполнения системных команд
   * Функция handle\_command\_mode() для обработки запросов
2. Передача файлов:
   * Работа с директорией FILE\_DIR
   * Функции:
     + handle\_file\_transfer\_mode() - основной обработчик
     + Загрузка/скачивание через системные вызовы read()/write()

**3.4 Сетевое взаимодействие**

Для работы с сетью используются:

* socket()/bind()/listen()/accept() - основные сокет-функции
* send()/recv() - передача данных
* select() (опционально) для мониторинга нескольких сокетов

**3.5 Потокобезопасность**

Для обеспечения потокобезопасности используются:

* pthread\_mutex\_t для защиты:
  + Списка клиентов
  + Доступа к базе данных SQLite
  + Истории чата
* Атомарные операции при работе с сокетами

**3.6 База данных**

Для аутентификации используется SQLite с таблицей:

CREATE TABLE users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL UNIQUE,

password TEXT NOT NULL

);

Функции работы с БД:

* authenticate\_user() - проверка логина/пароля
* register\_user() - регистрация новых пользователей
* init\_database() - инициализация БД

**3.7 Дополнительные структуры**

* Временные файлы: создаются в FILE\_DIR с уникальными именами
* Буферизация: используется для передачи больших файлов
* Форматирование сообщений: единый формат для клиент-серверного взаимодействия

Данная структура обеспечивает:

1. Многопоточную обработку клиентов
2. Безопасный доступ к общим ресурсам
3. Гибкое управление различными режимами работы
4. Масштабируемость для добавления новых функций

**4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ**

В данном разделе рассмотрены алгоритмы работы ключевых функций, реализующих основной цикл обработки клиентских соединений и функциональность TCP-сервера с поддержкой чата, выполнения команд и передачи файлов.

4.1 Разработка схем алгоритмов

Схема алгоритма основного цикла сервера, реализованного в функции main(), приведена в приложении Б.

Схема алгоритма обработки клиентской сессии, реализованного в функции handle\_client(), приведена в приложении В.

4.2 Разработка алгоритмов

В основном цикле сервера main() происходит следующая последовательность действий:

1. Ожидание нового подключения:
   * Сервер блокируется на вызове accept(), ожидая входящего соединения
   * При поступлении соединения создается новый клиентский сокет
2. Создание клиентской сессии:
   * Выделяется память под структуру Client
   * Инициализируются поля сокета (новый дескриптор) и статуса аутентификации
   * Имя пользователя пока не заполнено
3. Запуск обработчика клиента:
   * Создается новый поток через pthread\_create()
   * Потоку передается указатель на структуру клиента
   * Поток отсоединяется (pthread\_detach) для автономной работы
4. Возврат к ожиданию:
   * Сервер немедленно возвращается к вызову accept()
   * Готов принимать следующее подключение, не дожидаясь завершения обработки предыдущего

Функция handle\_client(void\* arg) является ядром обработки клиентских соединений и реализует следующий алгоритм:

1. Инициализация сессии:
   * Получение структуры Client из параметра
   * Выделение буферов для обмена сообщениями
   * Установка таймаутов на сокет (опционально)
2. Процесс аутентификации:
   * Запрос учетных данных
   * Проверка через SQLite базу
   * Отказ при неудачной аутентификации
3. Выбор режима работы:
   * Отправка клиенту меню доступных режимов
   * Ожидание выбора (1-чат, 2-команды, 3-файлы)
   * Ветвление на соответствующий обработчик
4. Обработка режима чата:
   * Получение и рассылка сообщений
   * Ведение истории чата
   * Обработка команд (напр. /exit)
5. Обработка режима команд:
   * Безопасное выполнение shell-команд
   * Возврат результата клиенту
   * Валидация ввода
6. Обработка режима передачи файлов:
   * Поддержка команд upload/download
   * Проверка прав доступа к файлам
   * Бинарная передача с контролем целостности
7. Завершение сессии:
   * Удаление клиента из списка активных
   * Закрытие сокета
   * Освобождение ресурсов
   * Логирование завершения сессии

**5 РЕЗУЛЬТАТЫ РАБОТЫ**

**5.1 Запуск сервера**

Сервер реализован как многопоточное TCP-приложение и запускается командой из каталога bin. Для запуска сервера используется исполняемый файл, предварительно собранный с помощью компилятора gcc с поддержкой POSIX API

gcc -std=c11 server.c -o server -lpthread

danil@MacBook-Air-danil bin % ./server

После запуска сервер начинает прослушивание порта 8080 и выводит следующее сообщение:

Server started on port 8080

Waiting for connections...

Client test added to slot 0

Client test authenticated successfully

Client test disconnected

Removing client test (socket 4)

При подключении клиента сервер выполняет процедуру аутентификации. В случае успешной аутентификации клиент добавляется в пул подключений:

Client test added to slot 0

Client test authenticated successfully

При отключении клиента отображается соответствующее сообщение:

Client test disconnected

Removing client test (socket 4)

**5.2 Аутентификация и выбор режима работы**

Клиент подключается к серверу с помощью любой TCP-утилиты, например, nc:

danil@MacBook-Air-danil bin % nc localhost 8080

Сначала сервер запрашивает имя пользователя и пароль:

Enter your username: test

Enter your password: test

После успешной авторизации клиенту предлагается выбрать режим работы:

Enter mode:

1 - Chat

2 - SSH-like console

3 - File transfer

0 - Exit

Input:

Каждый режим представляет собой отдельную ветку обработки, запущенную в отдельном потоке.

**5.3 Режим чата**

При выборе режима 1 клиент переходит в текстовый чат, где может обмениваться сообщениями с другими пользователями в реальном времени. Пример вывода сессии двух клиентов:

=== CHAT MODE ===

Logged in as: test

Type /exit to quit

------------------

test2: hi everyone

hi test2

test2: im good

**5.4 Командный режим**

При выборе режима 2 активируется терминалоподобная оболочка, реализованная через fork/exec, которая позволяет выполнять команды на стороне сервера в ограниченном окружении. Пример вывода:

=== Command Mode (fork/exec implementation) ===

Type 'exit' to quit

user@server:/Users/danil/osisp/course/bin$ ls

client.cpp server.cpp file.txt

**5.5 Режим работы с файлами**

В режиме передачи файлов клиент может загружать и скачивать файлы с сервера, а также просматривать содержимое директории files для обмена файлами. Список доступных команд отображается при входе:

=== File Transfer Mode ===

Available commands:

download <filename> - Download a file

upload <filename> - Upload a file

list - List available files

exit - Exit this mode

ftp> list

Available files:

file.txt 89 bytes

**5.3 Тестирование функциональности**

Функциональность сервера тестировалась во всех трёх режимах:

Чат (режим 1): клиенты могут отправлять сообщения, которые транслируются другим подключённым пользователям. Сервер корректно обрабатывает множественные подключения и сообщения без задержек.

SSH-подобная консоль (режим 2): клиент отправляет команды, которые выполняются на стороне сервера с возвратом stdout/ stderr. Проверялись команды ls, pwd, echo и другие. Результаты корректно возвращались клиенту.

Передача файлов (режим 3): реализована возможность отправки и приёма файлов через TCP-соединение. Передача как текстовых, так и бинарных файлов выполняется без искажения данных.

Дополнительно производилось тестирование с использованием valgrind для выявления утечек памяти:

valgrind --leak-check=full --track-origins=yes ./server

Пример отчёта:

==123456== HEAP SUMMARY:

==123456== in use at exit: 0 bytes in 0 blocks

==123456== total heap usage: 500 allocs, 500 frees, 256,000 bytes allocated

==123456== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Результаты показывают отсутствие утечек памяти и корректную работу с памятью даже при множественных параллельных подключениях.

Тестирование проводилось на macOS и Linux-системах. Сервер показал стабильную работу при параллельной нагрузке, корректную обработку отключений клиентов, а также устойчивость к ошибочным командам и повторным подключениям.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были углублены знания языка программирования C++ и получены практические навыки в области системного и сетевого программирования в POSIX-совместимых операционных системах. Изучен и применён интерфейс сокетов реализации многопоточного TCP-сервера, обеспечивающего приём и обработку клиентских соединений по протоколу TCP/IP.

Получен опыт построения сетевых приложений с использованием потоков POSIX (pthreads) для параллельной обработки клиентов, включая создание и завершение потоков, а также синхронизацию доступа к общим ресурсам. Вместо мультиплексирования с помощью poll() или select(), в рамках данной реализации была выбрана модель с выделением отдельного потока на каждого клиента, что обеспечило простоту логики и достаточную производительность при небольшом числе одновременных подключений.

Разработанный сервер поддерживает три основных режима работы: чат, терминальную оболочку (с запуском команд через fork/exec) и передачу файлов. В чат-режиме реализован обмен сообщениями между подключёнными клиентами в реальном времени. Терминальный режим позволяет клиенту удалённо выполнять команды на сервере в ограничённой среде. Режим передачи файлов предоставляет команды для загрузки, скачивания и просмотра содержимого серверного каталога, обеспечивая ограниченный, но функциональный аналог FTP.

Также реализована базовая система аутентификации пользователей с хранением пар логин/пароль, а сама архитектура приложения организована с учётом возможного расширения (например, внедрения баз данных, системы прав или шифрования трафика).

Сервер протестирован на устойчивость при множественных параллельных соединениях, в том числе при переключении режимов и отключении клиентов. Корректное завершение сессий и отсутствие утечек памяти подтверждены с помощью анализа средствами valgrind.

Тем не менее, проект остаётся открытым для дальнейших улучшений. Возможна реализация разграничения прав доступа, добавление полноценного шифрования (например, TLS), расширение командной оболочки до полноценного удалённого управления.

Данный проект предоставил ценный практический опыт проектирования и реализации сетевых серверов на C с акцентом на многопоточность, безопасность работы с файлами, межпоточное взаимодействие и реализацию пользовательских интерфейсов через текстовые протоколы.

6 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] POSIX Sockets [Электронный ресурс]. — The Open Group. — Режим доступа: <https://pubs.opengroup.org/onlinepubs/9699919799/functions/socket.html> — Дата доступа: 04.05.2025  
 [2] Stevens W. R. UNIX Network Programming. Volume 1: The Sockets Networking API. — 3rd ed. — Addison-Wesley, 2004.  
 [3] TCP/IP Illustrated, Volume 1: The Protocols / W. Richard Stevens — Addison-Wesley, 1994.  
 [4] Beej’s Guide to Network Programming [Электронный ресурс] / Brian “Beej Jorgensen” Hall. — Режим доступа: https://beej.us/guide/bgnet/ — Дата доступа: 04.05.2025  
 [5] RFC 793 — Transmission Control Protocol [Электронный ресурс]. — Internet Engineering Task Force (IETF). — Режим доступа: https://datatracker.ietf.org/doc/html/rfc793 — Дата доступа: 04.05.2025  
 [6] Linux Man Pages [Электронный ресурс]. — Режим доступа: https://man7.org/linux/man-pages/ — Дата доступа: 04.05.2025

**ПРИЛОЖЕНИЕ А**

(Листинг кода)

Файл auth.h

#ifndef AUTH\_H

#define AUTH\_H

#include <stdbool.h>

#include <pthread.h>

extern pthread\_mutex\_t db\_mutex;

bool authenticate\_user(const char\* username, const char\* password);

void init\_database();

#endif

Файл auth.с

#include "../include/auth.h"

#include <sqlite3.h>

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdbool.h> // Добавлено для bool

extern pthread\_mutex\_t db\_mutex;

void init\_database() {

sqlite3\* db;

char\* err\_msg = 0;

int rc = sqlite3\_open("../db/auth.db", &db);

if (rc != SQLITE\_OK) {

fprintf(stderr, "Cannot open the database: %s\n", sqlite3\_errmsg(db));

exit(1);

}

const char\* sql = "CREATE TABLE IF NOT EXISTS users("

"id INTEGER PRIMARY KEY AUTOINCREMENT,"

"username TEXT NOT NULL UNIQUE,"

"password TEXT NOT NULL);";

pthread\_mutex\_lock(&db\_mutex);

rc = sqlite3\_exec(db, sql, 0, 0, &err\_msg);

pthread\_mutex\_unlock(&db\_mutex);

if (rc != SQLITE\_OK) {

fprintf(stderr, "SQL error: %s\n", err\_msg);

sqlite3\_free(err\_msg);

}

sqlite3\_close(db);

}

bool authenticate\_user(const char\* username, const char\* password) {

sqlite3\* db;

sqlite3\_stmt\* stmt;

int rc = sqlite3\_open("../db/auth.db", &db);

const char\* sql = "SELECT 1 FROM users WHERE username = ? AND password = ?;";

rc = sqlite3\_prepare\_v2(db, sql, -1, &stmt, NULL);

sqlite3\_bind\_text(stmt, 1, username, -1, SQLITE\_STATIC);

sqlite3\_bind\_text(stmt, 2, password, -1, SQLITE\_STATIC);

rc = sqlite3\_step(stmt);

bool result = (rc == SQLITE\_ROW);

sqlite3\_finalize(stmt);

sqlite3\_close(db);

return result;

}

Файл chat.h

#ifndef CHAT\_H

#define CHAT\_H

#include <pthread.h>

#include <stdbool.h>

#include <sys/socket.h>

#define BUFFER\_SIZE 1024

void handle\_chat\_mode(int client\_socket, const char\* username);

void send\_chat\_history(int client\_socket);

void add\_to\_chat\_history(const char\* username, const char\* message);

#endif

Файл chat.с

#include "../include/chat.h"

#include "../include/server.h"

#include <time.h>

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#define MAX\_CHAT\_HISTORY 100

typedef struct {

char message[BUFFER\_SIZE];

char username[MAX\_USERNAME\_LEN];

time\_t timestamp;

} ChatMessage;

ChatMessage chat\_history[MAX\_CHAT\_HISTORY];

int chat\_history\_count = 0;

pthread\_mutex\_t chat\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void add\_to\_chat\_history(const char\* username, const char\* message) {

pthread\_mutex\_lock(&chat\_mutex);

if (chat\_history\_count >= MAX\_CHAT\_HISTORY) {

for (int i = 1; i < MAX\_CHAT\_HISTORY; i++) {

chat\_history[i-1] = chat\_history[i];

}

chat\_history\_count--;

}

strncpy(chat\_history[chat\_history\_count].username, username, MAX\_USERNAME\_LEN);

strncpy(chat\_history[chat\_history\_count].message, message, BUFFER\_SIZE);

chat\_history[chat\_history\_count].timestamp = time(NULL);

chat\_history\_count++;

pthread\_mutex\_unlock(&chat\_mutex);

}

void handle\_chat\_mode(int client\_socket, const char\* username) {

char buffer[BUFFER\_SIZE];

send\_message(client\_socket, "\n=== CHAT MODE ===\n");

send\_message(client\_socket, "Logged in as: ");

send\_message(client\_socket, username);

send\_message(client\_socket, "\nType /exit to quit\n");

send\_message(client\_socket, "------------------\n");

while (1) {

memset(buffer, 0, BUFFER\_SIZE);

ssize\_t bytes\_received = recv(client\_socket, buffer, BUFFER\_SIZE - 1, 0);

if (bytes\_received <= 0) {

printf("Client %s disconnected\n", username);

break;

}

buffer[strcspn(buffer, "\n")] = '\0';

if (strcmp(buffer, "/exit") == 0) {

break;

}

if (strlen(buffer) > 0) {

add\_to\_chat\_history(username, buffer);

char broadcast\_msg[BUFFER\_SIZE + MAX\_USERNAME\_LEN + 10];

snprintf(broadcast\_msg, sizeof(broadcast\_msg), "%s: %s\n", username, buffer);

broadcast\_message(broadcast\_msg, client\_socket);

}

}

send\_message(client\_socket, "\nGoodbye!\n\n");

close(client\_socket);

}

Файл commands.h

#ifndef COMMANDS\_H

#define COMMANDS\_H

#include <stdbool.h>

void handle\_command\_mode(int client\_socket);

#endif

Файл commands.c

#include "../include/commands.h"

#include "../include/server.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <fcntl.h>

#include <limits.h>

#include <errno.h>

#include <signal.h>

typedef struct {

char cwd[PATH\_MAX];

} SessionState;

static SessionState session;

static void sigchld\_handler(int sig) {

while (waitpid(-1, NULL, WNOHANG) > 0);

}

void execute\_command(int client\_socket, const char\* command) {

if (strncmp(command, "cd ", 3) == 0 || strcmp(command, "cd") == 0) {

char path[PATH\_MAX];

if (strcmp(command, "cd") == 0) {

strcpy(path, getenv("HOME") ? getenv("HOME") : "/");

} else {

strcpy(path, command + 3);

}

if (chdir(path) == -1) {

char error\_msg[256];

snprintf(error\_msg, sizeof(error\_msg), "cd: %s: %s\n", path, strerror(errno));

send(client\_socket, error\_msg, strlen(error\_msg), 0);

}

getcwd(session.cwd, sizeof(session.cwd));

return;

}

int pipe\_stdout[2], pipe\_stderr[2];

pid\_t pid;

if (pipe(pipe\_stdout) < 0 || pipe(pipe\_stderr) < 0) {

send\_message(client\_socket, "Error creating pipe\n");

return;

}

struct sigaction sa;

sa.sa\_handler = sigchld\_handler;

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = SA\_RESTART | SA\_NOCLDSTOP;

if (sigaction(SIGCHLD, &sa, NULL) == -1) {

perror("sigaction");

return;

}

pid = fork();

if (pid < 0) {

send\_message(client\_socket, "Error fork\n");

return;

}

if (pid == 0) {

dup2(pipe\_stdout[1], STDOUT\_FILENO);

dup2(pipe\_stderr[1], STDERR\_FILENO);

close(pipe\_stdout[0]);

close(pipe\_stderr[0]);

close(pipe\_stdout[1]);

close(pipe\_stderr[1]);

chdir(session.cwd);

char\* args[64];

int i = 0;

char\* token = strtok((char\*)command, " ");

while (token != NULL && i < 63) {

args[i++] = token;

token = strtok(NULL, " ");

}

args[i] = NULL;

execvp(args[0], args);

char error\_msg[256];

snprintf(error\_msg, sizeof(error\_msg), "Error command: %s\n", strerror(errno));

write(STDERR\_FILENO, error\_msg, strlen(error\_msg));

exit(EXIT\_FAILURE);

} else {

close(pipe\_stdout[1]);

close(pipe\_stderr[1]);

char buffer[BUFFER\_SIZE];

ssize\_t n;

while ((n = read(pipe\_stdout[0], buffer, sizeof(buffer))) > 0) {

send(client\_socket, buffer, n, 0);

}

while ((n = read(pipe\_stderr[0], buffer, sizeof(buffer))) > 0) {

send(client\_socket, buffer, n, 0);

}

close(pipe\_stdout[0]);

close(pipe\_stderr[0]);

int status;

waitpid(pid, &status, 0);

if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {

char exit\_msg[64];

snprintf(exit\_msg, sizeof(exit\_msg), "Command ended with code %d\n", WEXITSTATUS(status));

send\_message(client\_socket, exit\_msg);

}

}

}

void handle\_command\_mode(int client\_socket) {

char buffer[BUFFER\_SIZE];

getcwd(session.cwd, sizeof(session.cwd));

send\_message(client\_socket, "=== Command Mode (fork/exec implementation) ===\n");

send\_message(client\_socket, "Type 'exit' to quit\n");

while (1) {

char prompt[PATH\_MAX + 64];

snprintf(prompt, sizeof(prompt), "user@server:%s$ ", session.cwd);

send\_message(client\_socket, prompt);

ssize\_t bytes\_received = recv(client\_socket, buffer, BUFFER\_SIZE - 1, 0);

if (bytes\_received <= 0) {

break;

}

buffer[bytes\_received] = '\0';

buffer[strcspn(buffer, "\n\r")] = '\0';

if (strcmp(buffer, "exit") == 0) {

send\_message(client\_socket, "Exiting command mode...\n");

break;

}

if (strlen(buffer) == 0) {

continue;

}

execute\_command(client\_socket, buffer);

}

}

Файл file\_transfer.h

#ifndef FILE\_TRANSFER\_H

#define FILE\_TRANSFER\_H

#include <sys/types.h>

#include <stdbool.h>

void handle\_file\_transfer\_mode(int client\_socket);

#endif

Файл file\_transfer.c

#include "../include/file\_transfer.h"

#include "../include/server.h"

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <dirent.h>

#include <sys/stat.h>

#include <sys/socket.h>

#include <fcntl.h>

#include <errno.h>

#include <stdlib.h>

#include <limits.h>

#define FILE\_DIR "files/"

#define MAX\_FILENAME\_LEN 256

#define MIN(a, b) ((a) < (b) ? (a) : (b))

static void ensure\_files\_dir() {

struct stat st = {0};

if (stat(FILE\_DIR, &st) == -1) {

mkdir(FILE\_DIR, 0755);

}

}

void handle\_file\_transfer\_mode(int client\_socket) {

ensure\_files\_dir();

char buffer[BUFFER\_SIZE];

send\_message(client\_socket, "\n=== File Transfer Mode ===\n");

send\_message(client\_socket, "Available commands:\n"

"download <filename> - Download a file\n"

"upload <filename> - Upload a file\n"

"list - List available files\n"

"exit - Exit this mode\n");

while (1) {

send\_message(client\_socket, "ftp> ");

ssize\_t bytes\_received = recv(client\_socket, buffer, BUFFER\_SIZE - 1, 0);

if (bytes\_received <= 0) {

break;

}

buffer[bytes\_received] = '\0';

buffer[strcspn(buffer, "\r\n")] = '\0';

if (strcmp(buffer, "exit") == 0) {

break;

}

else if (strcmp(buffer, "list") == 0) {

DIR \*dir;

struct dirent \*ent;

if ((dir = opendir(FILE\_DIR)) != NULL) {

send\_message(client\_socket, "Available files:\n");

int file\_count = 0;

while ((ent = readdir(dir)) != NULL) {

if (ent->d\_type == DT\_REG) {

char file\_info[512];

struct stat st;

char filepath[512];

snprintf(filepath, sizeof(filepath), "%s%s", FILE\_DIR, ent->d\_name);

if (stat(filepath, &st) == 0) {

snprintf(file\_info, sizeof(file\_info), "%-30s %8lld bytes\n",

ent->d\_name, (long long)st.st\_size);

send\_message(client\_socket, file\_info);

file\_count++;

}

}

}

if (file\_count == 0) {

send\_message(client\_socket, "No files available\n");

}

closedir(dir);

} else {

send\_message(client\_socket, "Failed to open directory");

}

}

else if (strncmp(buffer, "download ", 9) == 0) {

char filename[MAX\_FILENAME\_LEN];

strncpy(filename, buffer + 9, MAX\_FILENAME\_LEN - 1);

filename[MAX\_FILENAME\_LEN - 1] = '\0';

char filepath[PATH\_MAX];

snprintf(filepath, sizeof(filepath), "%s%s", FILE\_DIR, filename);

struct stat st;

if (stat(filepath, &st) != 0 || !S\_ISREG(st.st\_mode)) {

send\_message(client\_socket, "Error: File not found");

continue;

}

int fd = open(filepath, O\_RDONLY);

if (fd < 0) {

send\_message(client\_socket, "Error: Cannot open file");

continue;

}

char header[128];

snprintf(header, sizeof(header), "FILE\_START:%s:%lld\n", filename, (long long)st.st\_size);

send\_message(client\_socket, header);

char file\_buffer[BUFFER\_SIZE];

ssize\_t bytes\_read, bytes\_sent;

off\_t total\_sent = 0;

while ((bytes\_read = read(fd, file\_buffer, sizeof(file\_buffer))) > 0) {

bytes\_sent = send(client\_socket, file\_buffer, bytes\_read, 0);

if (bytes\_sent <= 0) {

break;

}

total\_sent += bytes\_sent;

}

close(fd);

if (total\_sent == st.st\_size) {

send\_message(client\_socket, "FILE\_END:Transfer complete\n");

} else {

send\_message(client\_socket, "FILE\_ERR:Transfer interrupted\n");

}

}

else if (strncmp(buffer, "upload ", 7) == 0) {

char filename[256];

sscanf(buffer + 7, "%255s", filename);

printf("Upload request for: %s\n", filename);

char filepath[PATH\_MAX];

snprintf(filepath, sizeof(filepath), "%s/%s", FILE\_DIR, filename);

printf("Full destination path: %s\n", filepath);

int fd = open(filepath, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd == -1) {

perror("open() failed");

send\_message(client\_socket, "Upload failed: cannot create file\n");

continue;

}

printf("File opened for writing, fd=%d\n", fd);

char buf[BUFFER\_SIZE];

ssize\_t bytes\_received;

off\_t total = 0;

while ((bytes\_received = recv(client\_socket, buf, sizeof(buf), 0)) > 0) {

ssize\_t written = write(fd, buf, bytes\_received);

if (written != bytes\_received) {

perror("write() failed");

break;

}

total += written;

printf("Received %zd bytes (total: %ld)\n", bytes\_received, total);

}

close(fd);

printf("Upload completed. Total bytes: %ld\n", total);

send\_message(client\_socket, "Upload complete\n");

if (access(filepath, F\_OK) == 0) {

printf("File verification: EXISTS\n");

} else {

printf("File verification: MISSING (errno=%d)\n", errno);

}

}

else {

send\_message(client\_socket, "Error: Unknown command\n");

}

}

send\_message(client\_socket, "Exiting file transfer mode\n");

}

Файл server.h

#ifndef SERVER\_H

#define SERVER\_H

#include <stdbool.h>

#include <pthread.h>

#include <sys/socket.h>

#include <netinet/in.h>

#define PORT 8080

#define MAX\_CLIENTS 100

#define BUFFER\_SIZE 1024

#define MAX\_USERNAME\_LEN 50

#define MAX\_PASSWORD\_LEN 50

#if defined(\_\_APPLE\_\_) || defined(\_\_MACH\_\_)

#define TCP\_KEEPALIVE\_OPTIONS 1

#define TCP\_KEEPIDLE TCP\_KEEPALIVE

#elif defined(\_\_linux\_\_)

#define TCP\_KEEPALIVE\_OPTIONS 1

#endif

typedef struct {

int socket;

char username[MAX\_USERNAME\_LEN];

bool authenticated;

pthread\_t thread\_id;

bool should\_free;

} Client;

typedef struct {

Client\* clients[MAX\_CLIENTS];

int count;

pthread\_mutex\_t mutex;

volatile bool running;

} ClientList;

extern ClientList client\_list;

extern pthread\_mutex\_t db\_mutex;

void init\_server(void);

void shutdown\_server(void);

void\* handle\_client(void\* arg);

void add\_client(Client\* client);

void remove\_client(int socket);

void broadcast\_message(const char\* message, int sender\_socket);

void send\_message(int socket, const char\* message);

ssize\_t receive\_message(int socket, char\* buffer, size\_t size);

int create\_server\_socket(int port);

#endif

Файл server.c

#include "../include/server.h"

#include "../include/chat.h"

#include "../include/auth.h"

#include "../include/commands.h"

#include "../include/file\_transfer.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <signal.h>

#include <pthread.h>

#include <netinet/tcp.h>

#include <netinet/in.h>

ClientList client\_list = {0};

pthread\_mutex\_t db\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void init\_server(void) {

client\_list.count = 0;

client\_list.running = true;

pthread\_mutex\_init(&client\_list.mutex, NULL);

signal(SIGPIPE, SIG\_IGN);

}

void shutdown\_server() {

pthread\_mutex\_lock(&client\_list.mutex);

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

if (client\_list.clients[i] != NULL) {

printf("Shutting down client %s\n", client\_list.clients[i]->username);

if (client\_list.clients[i]->socket > 0) {

shutdown(client\_list.clients[i]->socket, SHUT\_RDWR);

close(client\_list.clients[i]->socket);

}

free(client\_list.clients[i]);

client\_list.clients[i] = NULL;

}

}

client\_list.count = 0;

pthread\_mutex\_unlock(&client\_list.mutex);

}

void add\_client(Client\* client) {

pthread\_mutex\_lock(&client\_list.mutex);

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

if (client\_list.clients[i] && client\_list.clients[i]->socket == client->socket) {

pthread\_mutex\_unlock(&client\_list.mutex);

return;

}

}

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

if (client\_list.clients[i] == NULL) {

client\_list.clients[i] = client;

client\_list.count++;

printf("Client %s added to slot %d\n", client->username, i);

break;

}

}

pthread\_mutex\_unlock(&client\_list.mutex);

}

void remove\_client(int socket) {

pthread\_mutex\_lock(&client\_list.mutex);

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

if (client\_list.clients[i] != NULL &&

client\_list.clients[i]->socket == socket) {

printf("Removing client %s (socket %d)\n",

client\_list.clients[i]->username, socket);

client\_list.clients[i]->should\_free = false;

free(client\_list.clients[i]);

client\_list.clients[i] = NULL;

client\_list.count--;

break;

}

}

pthread\_mutex\_unlock(&client\_list.mutex);

}

void broadcast\_message(const char\* message, int sender\_socket) {

pthread\_mutex\_lock(&client\_list.mutex);

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

if (client\_list.clients[i] != NULL &&

client\_list.clients[i]->socket != sender\_socket &&

client\_list.clients[i]->authenticated) {

if (send(client\_list.clients[i]->socket, message, strlen(message), MSG\_NOSIGNAL) < 0) {

printf("Failed to send to client %s, marking for removal\n",

client\_list.clients[i]->username);

client\_list.clients[i]->socket = -1;

}

}

}

pthread\_mutex\_unlock(&client\_list.mutex);

}

void send\_message(int socket, const char\* message) {

send(socket, message, strlen(message), 0);

}

ssize\_t receive\_message(int socket, char\* buffer, size\_t size) {

memset(buffer, 0, size);

ssize\_t bytes\_received = recv(socket, buffer, size - 1, 0);

if (bytes\_received <= 0) {

return bytes\_received;

}

buffer[strcspn(buffer, "\r\n")] = '\0';

return bytes\_received;

}

int create\_server\_socket(int port) {

int server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket < 0) {

perror("Error creating socket");

exit(EXIT\_FAILURE);

}

int opt = 1;

setsockopt(server\_socket, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

struct sockaddr\_in server\_addr;

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(port);

if (bind(server\_socket, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("Error bind");

exit(EXIT\_FAILURE);

}

if (listen(server\_socket, 10) < 0) {

perror("Error listen");

exit(EXIT\_FAILURE);

}

return server\_socket;

}

bool authenticate\_client(Client\* client) {

char buffer[BUFFER\_SIZE];

char username[MAX\_USERNAME\_LEN];

char password[MAX\_PASSWORD\_LEN];

send\_message(client->socket, "Enter your username: ");

if (receive\_message(client->socket, buffer, BUFFER\_SIZE) <= 0) {

return false;

}

strncpy(username, buffer, MAX\_USERNAME\_LEN);

send\_message(client->socket, "Enter your password: ");

if (receive\_message(client->socket, buffer, BUFFER\_SIZE) <= 0) {

return false;

}

strncpy(password, buffer, MAX\_PASSWORD\_LEN);

pthread\_mutex\_lock(&db\_mutex);

bool auth\_result = authenticate\_user(username, password);

pthread\_mutex\_unlock(&db\_mutex);

if (!auth\_result) {

send\_message(client->socket, "Authentication failed. Disconnecting...\n");

return false;

}

strncpy(client->username, username, MAX\_USERNAME\_LEN);

client->authenticated = true;

add\_client(client);

printf("Client %s authenticated successfully\n", username);

return true;

}

void\* handle\_client(void\* arg) {

Client\* client = (Client\*)arg;

char buffer[BUFFER\_SIZE];

client->should\_free = true;

struct timeval tv;

tv.tv\_sec = 30;

tv.tv\_usec = 0;

setsockopt(client->socket, SOL\_SOCKET, SO\_RCVTIMEO, &tv, sizeof(tv));

if (!authenticate\_client(client)) {

goto cleanup;

}

while (client->authenticated) {

send\_message(client->socket, "\nEnter mode:\n");

send\_message(client->socket, "1 - Chat\n");

send\_message(client->socket, "2 - SSH-like console\n");

send\_message(client->socket, "3 - File transfer\n");

send\_message(client->socket, "0 - Exit\n");

send\_message(client->socket, "Input: ");

ssize\_t bytes\_received = recv(client->socket, buffer, BUFFER\_SIZE - 1, 0);

if (bytes\_received <= 0) {

printf("Client %s disconnected\n", client->username);

break;

}

buffer[bytes\_received] = '\0';

buffer[strcspn(buffer, "\r\n")] = '\0';

int mode = atoi(buffer);

switch (mode) {

case 1:

handle\_chat\_mode(client->socket, client->username);

break;

case 2:

handle\_command\_mode(client->socket);

break;

case 3:

handle\_file\_transfer\_mode(client->socket);

break;

case 0:

send\_message(client->socket, "Goodbye!\n");

client->authenticated = false;

break;

default:

send\_message(client->socket, "Incorrect mode\n");

break;

}

if (!client->authenticated) {

break;

}

}

cleanup:

if (client->authenticated) {

remove\_client(client->socket);

}

if (client->socket > 0) {

close(client->socket);

client->socket = -1;

}

if (client->should\_free) {

free(client);

}

return NULL;

}

**ПРИЛОЖЕНИЕ Б**

(Структурная схема программы)

**ПРИЛОЖЕНИЕ В**

(Блок-схема алгоритма main())

**ПРИЛОЖЕНИЕ Г**

(Блок-схема алгоритма handle\_client())

**ПРИЛОЖЕНИЕ Д**

(Ведомость документов)