Министерство науки и высшего образования РФ

Рязанский государственный университет имени В.Ф. Уткина

Кафедра ВПМ

**Пояснительная записка к курсовой работе**  
«Разработка клиент-серверного приложения на стеке TCP/IP»  
по дисциплине  
«Протоколы вычислительных сетей»

Выполнили:  
Студенты группы 343М  
Анастасьев А. А  
Асташкин М. С.  
Проверил:  
Коротаев А. Т.

Содержание

[Введение 3](#_Toc185018364)

[Цели и задачи 4](#_Toc185018365)

[Теоретическая часть 5](#_Toc185018366)

[Описание протоколов TCP/IP 5](#_Toc185018367)

[Описание модели OSI 6](#_Toc185018368)

[Практическая часть 8](#_Toc185018369)

[Выбор среды и языка программирования 8](#_Toc185018370)

[Создание простейшего сервера для обмена сообщений по протоколу TCP 9](#_Toc185018371)

[Создание чата 11](#_Toc185018372)

[Руководство программиста 13](#_Toc185018373)

[Компиляция программных модулей сервера и клиента 13](#_Toc185018374)

[Руководство пользователя 14](#_Toc185018375)

[Заключение 15](#_Toc185018376)

[Литература 16](#_Toc185018377)

[Приложение 17](#_Toc185018378)

[Код простейшего клиента 17](#_Toc185018379)

[Код простейшего сервера 17](#_Toc185018380)

[Код клиента для чата 19](#_Toc185018381)

[Код сервера для чата 21](#_Toc185018382)

Введение

В современных информационных технологиях компьютерные сети играют ключевую роль, обеспечивая взаимодействие между устройствами, обмен данными и предоставление сервисов. Одной из важнейших составляющих сетевого взаимодействия являются протоколы передачи данных, которые определяют правила и форматы обмена информацией между участниками сети.

В данной курсовой работе рассматривается практическая реализация взаимодействия клиент-сервер с использованием TCP-сокетов. Протокол TCP (Transmission Control Protocol) представляет собой один из основных протоколов транспортного уровня модели OSI [1]. Его ключевые характеристики включают обеспечение надежной передачи данных, управление потоком и контроль целостности сообщений, что делает его востребованным для создания приложений, требующих гарантированной доставки информации.

В рамках исследования будут рассмотрены основные принципы работы протокола TCP, его преимущества и ограничения, а также особенности реализации приложений на его основе. Курсовая работа акцентирует внимание на практическом применении теоретических знаний, полученных в рамках дисциплины «Протоколы вычислительных сетей».

Разработанный программный продукт и результаты проведённого анализа могут быть использованы для дальнейшего изучения сетевых технологий и создания более сложных распределённых систем.

Цели и задачи

Целью данной курсовой работы является исследование особенностей функционирования и применения протокола TCP в рамках взаимодействия клиент-сервер.

Для достижения поставленной цели необходимо решить следующие задачи.

Изучить теоретические основы работы протокола TCP, включая его основные характеристики, принципы функционирования и области применения.

Рассмотреть особенности работы с TCP-сокетами в современных программных средах.

Спроектировать архитектуру клиент-серверного приложения, демонстрирующего взаимодействие через TCP.

Реализовать прототип клиентской и серверной частей приложения с использованием TCP-сокетов.

Таким образом, работа направлена на исследование протокола TCP как инструмента для реализации сетевых приложений, а также на выявление его преимуществ и ограничений в условиях конкретного применения.

Теоретическая часть

Описание протоколов TCP/IP

Стек протоколов TCP/IP (Transmission Control Protocol/Internet Protocol) представляет собой набор сетевых протоколов, обеспечивающих передачу данных в интернете и других сетях. Он был разработан в 1970-х годах как универсальная система для соединения разнородных сетей, что обеспечило его широкое распространение. [2, 3]

Стек TCP/IP состоит из четырёх уровней.

**Уровень доступа к сети** (Network Access Layer): отвечает за физическую передачу данных между устройствами. Этот уровень включает технологии канального уровня, такие как Ethernet и Wi-Fi, а также методы доступа к среде передачи.

**Интернет-уровень** (Internet Layer): включает протокол IP (Internet Protocol), который обеспечивает маршрутизацию пакетов данных между устройствами. Основные функции: адресация, разбиение сообщений на пакеты и их доставка к получателю.

**Транспортный уровень** (Transport Layer): включает протоколы TCP и UDP (User Datagram Protocol). TCP обеспечивает надёжную передачу данных с установлением соединения, управление потоком и контроль ошибок. UDP, напротив, используется для более простой и быстрой передачи данных, не требующей надёжности.

**Прикладной уровень** (Application Layer): предоставляет интерфейсы и протоколы для взаимодействия приложений, такие как HTTP, FTP, SMTP и другие.

Ключевым элементом стека является протокол TCP, который обеспечивает надежность передачи данных. Он использует механизмы трёхэтапного установления соединения (трёхстороннего рукопожатия), нумерацию пакетов, квитирование (ACK) и контроль ошибок для достижения гарантированной доставки данных. Протокол IP, в свою очередь, отвечает за маршрутизацию и передачу пакетов, но не гарантирует их сохранность, что компенсируется транспортными протоколами.

Стек TCP/IP стал основой для интернета благодаря своей гибкости, масштабируемости и способности работать в разнородных сетях.

Описание модели OSI

Модель OSI (Open Systems Interconnection) была разработана в 1984 году Международной организацией по стандартизации (ISO) как эталонная модель взаимодействия открытых систем. Она представляет собой концептуальную структуру, разделяющую функции сетевых систем на семь уровней. [4]

**Физический уровень** (Physical Layer): описывает электрические, механические и другие физические параметры передачи данных. Это уровень кабелей, разъёмов и сигналов.

**Канальный уровень** (Data Link Layer): отвечает за установление, поддержание и завершение соединений между узлами сети. Основные функции включают контроль ошибок и управление доступом к среде передачи.

**Сетевой уровень** (Network Layer): обеспечивает маршрутизацию данных между сетями. Основной протокол – IP, который осуществляет адресацию и разбиение сообщений на пакеты.

**Транспортный уровень** (Transport Layer): отвечает за надёжную передачу данных между конечными точками. Здесь работают протоколы TCP и UDP.

**Сеансовый уровень** (Session Layer): обеспечивает управление сеансами связи между приложениями. Его функции включают установление, поддержку и завершение сеансов.

**Уровень представления** (Presentation Layer): преобразует данные в форму, понятную для приложений, обеспечивая кодирование, шифрование и сжатие.

**Прикладной уровень** (Application Layer): предоставляет интерфейсы для конечных пользователей, включая такие протоколы, как HTTP, FTP, SMTP и другие.

Модель OSI носит теоретический характер и широко используется для описания сетевых систем и их взаимодействий.

Сравнение моделей OSI и TCP/IP показывает, что TCP/IP объединяет уровни OSI, что делает его более практичным. Например, уровни прикладной, представительский и сеансовый модели OSI соответствуют прикладному уровню TCP/IP, а сетевой и транспортный уровни аналогичны.

Таким образом, модель OSI представляет собой эталонный подход к описанию сетевого взаимодействия, тогда как TCP/IP – более прикладное решение для реальной передачи данных.

Практическая часть

Выбор среды и языка программирования

Для реализации клиент-серверного приложения с использованием протокола TCP в данной курсовой работе выбран язык программирования **Java 17** [5]. Этот выбор обусловлен рядом причин, связанных с особенностями языка, его экосистемы и актуальностью в контексте разработки сетевых приложений.

Java предоставляет возможность разработки приложений, работающих на различных операционных системах без необходимости внесения изменений в код. Это обеспечивается благодаря использованию виртуальной машины Java (JVM), которая интерпретирует байт-код.

Java имеет обширную стандартную библиотеку, включающую классы для работы с сетью, такие как java.net.Socket и java.net.ServerSocket, которые упрощают реализацию клиент-серверного взаимодействия. Эти инструменты позволяют эффективно работать с протоколом TCP, абстрагируясь от низкоуровневых деталей.

Java обеспечивает строгую типизацию, автоматическое управление памятью с помощью сборки мусора и встроенные механизмы обработки исключений, что минимизирует вероятность ошибок в коде. Это особенно важно при разработке сетевых приложений, где ошибки могут привести к потере данных или нарушению безопасности.

Для сетевых приложений, работающих в режиме реального времени, важно использовать многопоточность. Java предоставляет встроенные инструменты для работы с потоками, такие как классы Thread и ExecutorService, что позволяет создавать приложения, эффективно обрабатывающие несколько подключений.

Java остаётся одним из самых популярных языков программирования, что обеспечивает доступ к большому количеству обучающих материалов, библиотек и активной поддержке сообщества.

Java 17 является версией с долгосрочной поддержкой (LTS), что делает её стабильным и надёжным выбором для разработки приложений. Эта версия включает многочисленные улучшения по сравнению с предыдущими, такие как: обновления в API для повышения производительности и удобства использования; улучшения в системе обработки исключений; новые возможности языка, упрощающие разработку (например, текстовые блоки, улучшения switch).

**Обоснование выбора Java для данной работы**

Для реализации клиент-серверного приложения на основе протокола TCP необходим язык, который поддерживает кроссплатформенную разработку, обладает мощными средствами работы с сетью и многопоточностью, а также обеспечивает надёжность и удобство разработки. Java 17 полностью соответствует этим требованиям.

Кроме того, использование Java способствует приобретению навыков работы с популярным инструментом разработки, который востребован в различных областях, включая разработку корпоративного ПО и распределённых систем.

Таким образом, выбор Java 17 для реализации клиент-серверного приложения в рамках данной курсовой работы является оптимальным решением, обеспечивающим удобство разработки, надёжность и высокую производительность приложения.

Создание простейшего сервера для обмена сообщений по протоколу TCP

Целью создания сервера является разработка программного компонента, способного принимать входящие TCP-соединения от клиентов, получать передаваемые сообщения и отображать их для дальнейшей обработки или анализа.

Задачами сервера являются:

1. Установить TCP-сокет для прослушивания входящих соединений на указанном порту.
2. Обрабатывать входящие подключения клиентов.
3. Получать сообщения от подключенных клиентов.
4. Отображать полученные сообщения в консоли.
5. Завершать соединения после обработки сообщений.

Рассмотрим функции сервера. Сервер будет представлен классом SimpleTCPServer, который реализует следующие функции:

* startServer(int port)
  + Назначение: запускает сервер на указанном порту, открывает сокет для входящих соединений.
  + Параметры:
    - port — номер порта, на котором сервер будет слушать соединения.
  + Логика: создаёт объект ServerSocket и переходит в режим ожидания клиентских подключений.
* handleClient(Socket clientSocket)
  + Назначение: обрабатывает подключение от клиента, получает сообщение и выводит его в консоль.
  + Параметры:
    - clientSocket — сокет, установленный для связи с клиентом.
  + Логика: считывает данные из входного потока сокета и отображает их в консоли.

Целью создания клиента является разработка программы, которая устанавливает TCP-соединение с сервером и передаёт ему сообщение.

Рассмотрим функции клиента. Клиент будет представлен классом SimpleTCPClient, реализующим следующие функции:

* sendMessage(String host, int port, String message)
  + Назначение: устанавливает соединение с сервером и отправляет сообщение.
  + Параметры:
    - host — IP-адрес или имя хоста сервера.
    - port — номер порта, на котором сервер принимает соединения.
    - message — сообщение, которое нужно отправить.
  + Логика: создаёт объект Socket, отправляет сообщение через поток вывода и закрывает соединение.

Создание чата

Целью разработки сервера является создание централизованного узла, который может одновременно обслуживать нескольких клиентов (минимум трёх), обеспечивать их взаимодействие, регистрировать события в логах и администрировать соединения.

Задачами сервера являются:

1. Устанавливать TCP-соединение с клиентами и обрабатывать их запросы.
2. Поддерживать одновременную работу нескольких клиентов, распределяя задачи по потокам.
3. Вести логирование всех ключевых событий, включая:
4. подключение/отключение клиента;
5. получение и отправку сообщений;
6. ошибки в процессе работы.
7. Передавать сервисные сообщения клиентам (например, уведомление о получении сообщения).
8. Обеспечивать возможность администрирования (например, отключение определённого клиента).

Рассмотрим функции сервера:

* start()
  + Назначение: запускает сервер и готовит его к приёму соединений.
  + Логика: создаёт объект ServerSocket, запускает обработчики клиентов в отдельных потоках.
* ClientHandler.run()
  + Назначение: обрабатывает взаимодействие с клиентом: принимает сообщения, отправляет подтверждения, записывает события в логи.
  + Логика: запускается в отдельном потоке для каждого клиента. Считывает сообщения, добавляет записи в логи, передаёт сообщения другим клиентам.
* disconnectClient(int clientId)
  + Назначение: позволяет администратору завершить соединение с клиентом.
  + Логика: закрывает сокет клиента, добавляет событие в лог, уведомляет остальных клиентов.

Цель клиента – предоставить пользователю удобный интерфейс для подключения к серверу, отправки и получения сообщений, а также выполнения управляющих команд.

Задачами клиента являются:

1. Установить соединение с сервером.
2. Передавать текстовые сообщения серверу и получать подтверждения о доставке.
3. Реализовать интерфейс для выполнения управляющих команд (подключение, отключение и т.д.).
4. Логировать события на клиентской стороне (например, отправленные и полученные сообщения).
5. Обрабатывать ошибки, такие как недоступность сервера.

Рассмотрим функции клиента:

* connectToServer(int port)
  + Назначение: Устанавливает соединение с сервером.
  + Логика: Создаёт объект Socket, подключается к указанному серверу, записывает событие в лог.
* sendMessage(PrintWriter out)
  + Назначение: Отправляет текстовое сообщение на сервер.
  + Логика: Передаёт данные через выходной поток, записывает событие в лог.
* receiveMessage(BufferedReader in)
  + Назначение: Принимает сообщения от сервера (включая подтверждения и сообщения от других клиентов).
  + Логика: Считывает данные из входного потока, выводит их в интерфейс и логирует.

Руководство программиста

Компиляция программных модулей сервера и клиента

Компилируем файлы:

javac -d bin/client socket-sample/client/\*\*.java

javac -d bin/server socket-sample/server /\*\*.java

Создаём манифест (если нужен):

Manifest-Version: 1.0

Main-Class: chat.MultiClientServerRunner

--------

Manifest-Version: 1.0

Main-Class: chat.ChatClientRunner

Сохраняем два файла MANIFEST.MF в разные директории.

Упаковываем JAR:

jar cfm client.jar socket-sample/client/MANIFEST.MF -C bin/client .

jar cfm server.jar socket-sample/client/MANIFEST.MF -C bin/server .

Запускаем JAR:

java -jar server.jar

java -jar client.jar

Руководство пользователя

Для начала работы с клиентом чата, выполните команду start <port>, где <port> — это номер порта, на котором работает сервер чата. После успешного подключения к серверу, вы можете начать отправлять сообщения. Просто введите текст сообщения в консоли и нажмите **Enter**. Клиент автоматически отправит ваше сообщение на сервер, и вы получите подтверждение о его получении.

Если вы хотите завершить работу клиента, используйте команду exit. Это отключит ваше соединение с сервером и закроет клиентскую программу. Во время работы клиента вы можете вводить текстовые сообщения без предварительного указания команды — каждое введённое сообщение будет немедленно отправлено на сервер.

Пример взаимодействия:

> 2024-12-13T21:26:40: [INFO] Ожидание команд...

> start 12345

> 2024-12-13T21:26:51: [INFO]: Подключено к серверу чата!

> Привет, мир!

> 2024-12-13T21:26:57: [SERV]: Message 'Привет, мир!' received

> exit

> 2024-12-13T21:27:02: [INFO]: Socket closed

Заключение

В ходе выполнения курсовой работы была успешно разработана клиент-серверная система для обмена сообщениями по протоколу TCP. Цель работы, состоящая в создании простого чата, была достигнута. Разработанный сервер поддерживает многократные подключения клиентов, а клиентский интерфейс позволяет пользователю легко отправлять и получать сообщения.

Все поставленные задачи были выполнены: реализованы основные функции сервера и клиента, обеспечена обработка сообщений и ведение логов, а также учтены обработчики ошибок и сервисные сообщения. Система демонстрирует стабильную работу при подключении нескольких клиентов и эффективно выполняет все необходимые функции.

Таким образом, цель курсовой работы была достигнута, задачи выполнены, и полученные результаты подтверждают успешное освоение работы с протоколом TCP в контексте клиент-серверных приложений.

Литература

1. **Таненбаум, А. С.** Компьютерные сети: учебник / А. С. Таненбаум. — 5-е изд. — М.: Вильямс, 2011. — 1024 с.
2. **Купер, Д.** Сетевые технологии: практическое руководство / Д. Купер. — М.: ДМК Пресс, 2010. — 432 с.
3. **Хайнд, Дж.** Основы компьютерных сетей и протоколов TCP/IP / Дж. Хайнд. — СПб.: Питер, 2007. — 704 с.
4. **Мэйер, Л.** Сетевые программирования на языке Java / Л. Мэйер. — М.: Диалектика, 2009. — 456 с.
5. **Oracle**. Java 17 Documentation. — <https://docs.oracle.com> [Дата обращения: 13.12.2024].

****Приложение****

Код простейшего клиента

import java.io.OutputStreamWriter;

import java.io.PrintWriter;

import java.net.Socket;

public class SimpleTCPClient {

public void sendMessage(String host, int port, String message) {

try (Socket socket = new Socket(host, port);

PrintWriter writer = new PrintWriter(

new OutputStreamWriter(socket.getOutputStream()), true)) {

writer.println(message);

System.out.println("Сообщение отправлено: " + message);

} catch (Exception e) {

System.err.println("Ошибка отправки сообщения: " + e.getMessage());

}

}

public static void main(String[] args) {

SimpleTCPClient client = new SimpleTCPClient();

client.sendMessage("127.0.0.1", 12345, "Привет, сервер!");

}

}

Код простейшего сервера

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.net.ServerSocket;

import java.net.Socket;

public class SimpleTCPServer {

private int port;

public SimpleTCPServer(int port) {

this.port = port;

}

public void startServer() {

try (ServerSocket serverSocket = new ServerSocket(port)) {

System.out.println("Сервер запущен на порту: " + port);

while (true) {

Socket clientSocket = serverSocket.accept();

System.out.println("Подключен клиент: " + clientSocket.getInetAddress());

handleClient(clientSocket);

}

} catch (Exception e) {

System.err.println("Ошибка при запуске сервера: " + e.getMessage());

}

}

private void handleClient(Socket clientSocket) {

try (BufferedReader reader = new BufferedReader(

new InputStreamReader(clientSocket.getInputStream()))) {

String message = reader.readLine();

System.out.println("Сообщение от клиента: " + message);

} catch (Exception e) {

System.err.println("Ошибка обработки клиента: " + e.getMessage());

}

}

public static void main(String[] args) {

SimpleTCPServer server = new SimpleTCPServer(12345);

server.startServer();

}

}

Код клиента для чата

package chat;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.util.concurrent.atomic.AtomicBoolean;

import static chat.Logger.logError;

import static chat.Logger.logInfo;

public class SocketClient {

private static final String HOST = "localhost";

private final AtomicBoolean closed = new AtomicBoolean(false);

private final BufferedReader consoleInput = new BufferedReader(new InputStreamReader(System.in));

public void start() throws IOException {

int port = getPortFromInput();

connectToServer(port);

}

private void connectToServer(int port) throws IOException {

try (Socket socket = new Socket(HOST, port)) {

logInfo("[INFO]: Подключено к серверу чата!");

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

startServerListening(in);

startConsoleReading(out);

}

}

private void startConsoleReading(PrintWriter out) {

Thread readingThread = new Thread(() -> sendMessage(out));

readingThread.start();

try {

readingThread.join();

} catch (InterruptedException e) {

logInfo("Close connection...");

}

}

private void sendMessage(PrintWriter out) {

String message;

try {

while ((message = consoleInput.readLine()) != null && !closed.get()) {

if ("exit".equalsIgnoreCase(message)) {

break;

}

out.println(message);

}

} catch (IOException e) {

logError("[ERROR]: Ошибка клиента: " + e.getMessage());

}

}

private void startServerListening(BufferedReader in) {

new Thread(() -> receiveMessage(in)).start();

}

private void receiveMessage(BufferedReader in) {

try {

String serverMessage;

while ((serverMessage = in.readLine()) != null) {

receiveMessage("[SERV]: %s%n".formatted(serverMessage));

}

} catch (Exception e) {

logError("[INFO]: Ошибка чтения сообщений: " + e.getMessage());

} finally {

closed.set(true);

try {

System.in.close();

} catch (IOException e) {

logError("SYSTEM ERROR");

}

}

}

private static void receiveMessage(String serverMessage) {

logInfo(serverMessage);

}

private int getPortFromInput() throws IOException {

logInfo("[INFO] Ожидание команд...");

int port;

while (true) {

String message = consoleInput.readLine();

if (message.startsWith("start")) {

port = Integer.parseInt(message.split(" ")[1]);

break;

} else if (message.startsWith("exit")) {

throw new IOException();

}

}

return port;

}

}

Код сервера для чата

package chat;

import java.io.BufferedReader;

import java.io.Closeable;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.atomic.AtomicInteger;

import static chat.Logger.logError;

import static chat.Logger.logInfo;

public class SocketServer {

private static final int PORT = 12345;

private final AtomicInteger currentClientId = new AtomicInteger();

private final Map<Integer, Closeable> activeClients = new ConcurrentHashMap<>();

public void start() throws IOException {

logInfo("[INFO]: Сервер запущен, ожидаем подключения...");

new Thread(this::startServer).start();

BufferedReader consoleInput = new BufferedReader(new InputStreamReader(System.in));

String message;

while ((message = consoleInput.readLine()) != null) {

if (message.equals("exit")) {

return;

}

checkForKillClient(message);

}

}

private void checkForKillClient(String message) throws IOException {

if (message.startsWith("kill")) {

int clientId = Integer.parseInt(message.split(" ")[1]);

disconnectClient(clientId);

}

}

private void disconnectClient(int clientId) throws IOException {

activeClients.remove(clientId).close();

}

private void startServer() {

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

Socket clientSocket = serverSocket.accept();

int clientId = currentClientId.getAndIncrement();

Thread clientThread = new Thread(new ClientHandler(clientSocket));

activeClients.put(clientId, clientSocket);

clientThread.start();

logInfo("[INFO]: Клиент подключился. Client id: %s".formatted(clientId));

}

} catch (IOException e) {

logError("[INFO]: Ошибка сервера: " + e.getMessage());

}

}

}

package chat;

import java.io.BufferedReader;

import java.io.Closeable;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;

import static chat.Logger.logError;

import static chat.Logger.logInfo;

public class ClientHandler implements Runnable {

private static final Set<PrintWriter> CLIENT\_WRITERS = ConcurrentHashMap.newKeySet();

private final Socket socket;

public ClientHandler(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try (

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(socket.getOutputStream(), true)

) {

CLIENT\_WRITERS.add(out);

String message;

while ((message = in.readLine()) != null) {

logInfo("[CHAT]: " + message);

if ("exit".equalsIgnoreCase(message)) {

break;

}

for (PrintWriter writer : CLIENT\_WRITERS) {

if (writer != out) {

writer.println(message);

}

}

out.println("Message '%s' received".formatted(message));

}

} catch (IOException e) {

logInfo("[INFO]: " + e.getMessage());

} finally {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

CLIENT\_WRITERS.removeIf(PrintWriter::checkError);

}

}

}