МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ ПО ДИСЦИПЛИНЕ «Сетевое программирование»

для студентов направления подготовки 09.03.04 Программная инженерия
«Инженерия сложных программных систем»

Ставрополь, 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящее учебно-методическое пособие содержит описание лабораторного практикума по дисциплине «Сетевое программирование», предназначенного для бакалавров направления «Программная инженерия». Лабораторный практикум является ключевым элементом курса, обеспечивающим переход от теоретических знаний к устойчивым практическим навыкам разработки, отладки, тестирования и документирования клиент-серверных приложений.

Пособие структурировано по модульному принципу: 18 лабораторных работ объединены в пять тематических модулей, последовательно раскрывающих основные аспекты сетевого программирования – от базовых примитивов сокетов и анализа трафика до высокопроизводительных асинхронных серверов и подготовки проектной документации. Каждая работа содержит формулировку цели, перечень формируемых компетенций, теоретическое введение, подробное задание, методические указания, требования к содержанию отчёта, контрольные вопросы и чёткие критерии оценивания. Такой подход позволяет студентам самостоятельно осваивать материал в удобном темпе, а преподавателю – объективно контролировать качество выполнения.

Лабораторный практикум базируется на знаниях, полученных в предшествующих дисциплинах («Операционные системы», «Языки программирования», «Алгоритмы и структуры данных»), и служит фундаментом для последующих курсов по распределённым системам, высоконагруженным приложениям и безопасности программного обеспечения. Выполнение всех работ гарантирует не только успешную сдачу зачёта, но и формирование востребованных на рынке труда компетенций в области бэкенд-разработки, сетевой безопасности и DevOps-инжиниринга.

Авторский коллектив выражает надежду, что данное пособие поможет студентам «потрогать руками» механизмы взаимодействия программ через сеть и заложить прочную основу для дальнейшего профессионального роста.

Лабораторная работа №1

Тема: Знакомство с анализатором сетевого трафика Wireshark

Цель работы:
Научиться использовать инструмент Wireshark для захвата и анализа сетевого трафика. Освоить базовые приёмы фильтрации пакетов, изучить структуру стековых протоколов Ethernet, IP, TCP/UDP, а также детали протоколов HTTP и DNS. Выработать практический навык идентификации трёхэтапного установления TCP-соединения (TCP Handshake) и определения ключевых полей заголовков.

Формируемые компетенции (начальный уровень):

- ПК-4: умение применять анализатор трафика для отладки, тестирования и верификации сетевых взаимодействий.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Запускать захват трафика в Wireshark, корректно выбирая сетевой интерфейс.
2. Применять фильтры захвата и отображения (tcp, udp, http, dns, ip.addr и др.).
3. Идентифицировать на захваченных дампах трёхэтапное рукопожатие TCP: пакеты SYN, SYN, ACK, ACK.
4. Определять IP-адреса источника и назначения, номера портов, MAC-адреса, флаги TCP.
5. Описывать структуру кадра Ethernet, дейтаграммы IP, сегмента TCP/UDP, а также запросов и ответов HTTP и DNS.
6. Сохранять результаты захвата и экспортировать отдельные пакеты для отчёта.

2. Теоретические сведения

Wireshark – свободный анализатор сетевых протоколов, позволяющий перехватывать и интерактивно просматривать содержимое пакетов на выбранном сетевом интерфейсе. Пакеты отображаются в трёх секциях:

- список пакетов с краткими сводками;
- дерево протоколов (детализация заголовков);
- шестнадцатеричное представление содержимого.

Основные типы фильтров:

- *Capture filter* (BPF-синтаксис): ограничивает захват, например, port 80.

- *Display filter*: отображает только нужные пакеты из уже захваченного дампа, например, tcp.flags.syn == 1, http.request, dns.qry.name contains "example".

Краткая структура инкапсуляции:
Ethernet-кадр → IP-дейтаграмма → TCP-сегмент (или UDP-дейтаграмма) → данные прикладного уровня (HTTP, DNS).
Для TCP важно помнить флаги: SYN, ACK, FIN, RST, PSH и их комбинации. Процедура установки соединения: клиент посылает SYN, сервер отвечает SYN, ACK, клиент завершает квитирование пакетом ACK.

DNS использует преимущественно UDP (порт 53), запрос содержит поле Transaction ID, имя запрашиваемого домена; ответ добавляет ресурсные записи (A, AAAA, CNAME и т.д.). HTTP/1.1 поверх TCP показывает текстовые заголовки запросов и ответов.

3. Задание на выполнение работы

1. **Подготовка среды:**
   Убедиться, что Wireshark установлен и может захватывать трафик. При необходимости запустить от имени администратора/root.
2. **Захват трафика:**
   - Выбрать активный сетевой интерфейс.
   - Запустить захват без предварительного фильтра (или с фильтром port 80 or port 53 для ограничения объёма).
   - Сгенерировать учебный трафик:
     - открыть в браузере любой HTTP-сайт (например, http://example.com);
     - выполнить команду ping google.com (опционально);
     - выполнить nslookup ya.ru или аналогичный DNS-запрос.
   - Остановить захват через 30–60 секунд.
3. **Применение фильтров отображения:**
   - Отфильтровать только HTTP-трафик: http.
   - Отфильтровать только DNS-трафик: dns.
   - Для одного из HTTP-соединений выделить последовательность пакетов TCP Handshake с помощью фильтра tcp.stream eq N (номер потока определить по контекстному меню "Follow TCP stream").
4. **Анализ TCP Handshake:**
   Найти три пакета открытия соединения: [SYN], [SYN, ACK], [ACK]. Для каждого пакета зафиксировать:

- MAC-адреса (отправитель/получатель);
- IP-адреса (Source / Destination);
- TCP-порты;
- флаги и относительные/абсолютные sequence number (при необходимости включить отображение абсолютных номеров в настройках Wireshark).

5. **Анализ HTTP-запроса и ответа:**
   В рамках того же TCP-потока найти HTTP GET-запрос и соответствующий ответ. Просмотреть поля: метод, URI, версия HTTP, код ответа, заголовки Host, User-Agent, Content-Type и т.д.
6. **Анализ DNS-запроса/ответа:**
   С помощью фильтра dns найти пакет запроса к DNS-серверу. Определить:
   - IP-адрес DNS-сервера;
   - порт назначения (обычно 53);
   - имя запрашиваемого домена;
   - в ответе — IP-адрес(а), возвращённые в ресурсных записях типа A.
7. **Экспорт:**
   Сохранить отфильтрованный набор пакетов (или отдельные пакеты) в файл .pcapng для включения в отчёт (по требованию преподавателя). Сделать скриншоты основных шагов.

4. Методические указания по выполнению

- **Запуск захвата:**
  В главном окне Wireshark дважды щёлкнуть по активному интерфейсу (обычно с графиком трафика). Рекомендуется закрыть лишние сетевые приложения для уменьшения шума.
  Совет: можно задать capture filter tcp port 80 or udp port 53, чтобы сфокусироваться на нужных протоколах.
- **Фильтрация:**
  Display filter вводится непосредственно над списком пакетов. Полезные комбинации:
  - tcp.flags.reset == 1 – пакеты сброса;
  - http.request.method == "GET" – только GET-запросы;
  - dns.flags.response == 0 – только запросы.
    Для восстановления TCP-потока: ПКМ на любом пакете потока → Follow → TCP Stream.

- **Анализ TCP Handshake:**
  Убедитесь, что в колонке Info присутствуют флаги. Если отображаются только относительные sequence number, для наглядности отключите опцию: Edit → Preferences → Protocols → TCP → снять галочку "Relative sequence numbers". Обратите внимание, что флаг ACK может быть установлен во всех пакетах, кроме самого первого SYN.
- **Анализ HTTP:**
  Если трафик зашифрован (HTTPS), рукопожатие TCP видно, но HTTP-заголовки скрыты; используйте контрольный HTTP-ресурс. При использовании HTTP/2 может потребоваться включение расшифровки, но для целей ЛБ1 выберите классический HTTP/1.1.
- **Работа с DNS:**
  Для генерации запроса можно в командной строке выполнить nslookup example.com или использовать браузер. В списке пакетов запрос обычно помечен как "Standard query", ответ – "Standard query response".

5. Содержание отчёта

Отчёт должен быть оформлен в текстовом редакторе и содержать:

1. **Титульный лист** с номером и темой работы.
2. **Цель работы** (своими словами).
3. **Описание процесса захвата:** выбранный интерфейс, длительность, фильтры.
4. **Скриншоты и пояснения:**
   - Общий список пакетов с применённым фильтром http (или dns).
   - Три пакета TCP Handshake с видимыми флагами, IP-адресами и портами.
   - Развёрнутое дерево заголовков Ethernet, IP, TCP для одного из этих пакетов (подписать ключевые поля).
   - HTTP-запрос и ответ с заголовками.
   - DNS-запрос и ответ с выделенными полями Queries и Answers.
5. **Анализ результатов:** краткие выводы по структуре пакетов и порядку взаимодействия.
6. **Состав приложений:** файл(ы) захвата .pcapng (если требует преподаватель).

6. Контрольные вопросы

1. Какие основные протокольные уровни представлены в захваченном пакете HTTP-запроса? Опишите порядок инкапсуляции.
2. Что такое трёхэтапное рукопожатие TCP? Какие флаги и номера последовательности используются?
3. Чем отличаются фильтры захвата (capture) и отображения (display) в Wireshark? Приведите примеры.
4. Как определить, к какому TCP-потоку принадлежит конкретный пакет?
5. Какие поля содержит DNS-запрос и как в ответе возвращается IP-адрес?
6. Чем отличается HTTP от HTTPS с точки зрения видимости данных в Wireshark?
7. Каким образом в Wireshark можно посмотреть содержимое передаваемого файла или веб-страницы?

7. Критерии оценки

Оценка	Критерии
Зачтено	Студент продемонстрировал работающий захват трафика, корректно применил не менее двух разных фильтров отображения. В отчёте представлены скриншоты, подтверждающие: 1) идентификацию трёх пакетов TCP-квитирования с указанием IP-адресов и портов; 2) нахождение HTTP-запроса/ответа или DNS-запроса/ответа с детализацией заголовков. Студент способен устно объяснить назначение флагов SYN/ACK, различие TCP/UDP на примере захваченных пакетов, назвать основные поля IP-заголовка. Код не требуется.
Не зачтено	Захват трафика не выполнен, отчёт отсутствует либо состоит только из общих скриншотов без какой-либо аналитики. Студент не может показать пакеты рукопожатия, не понимает разницу между MAC-, IP-адресом и портом, не может ответить на контрольные вопросы.

Лабораторная работа №2

Тема: Простейший TCP-эхо-клиент

Цель работы:
Освоить базовые операции клиентского сетевого программирования с использованием TCP-сокетов: создание сокета, установка соединения, отправка и приём данных. Научиться обрабатывать типичные ошибки на стороне клиента и понимать особенности передачи байтовых данных. Закрепить навыки анализа сетевого трафика с помощью Wireshark, полученные в ЛР №1.

Формируемые компетенции (начальный уровень):

- ПК-4: разработка, отладка и тестирование сетевого приложения-клиента.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Создавать TCP-сокет средствами целевого языка (C/Python).
2. Выполнять подключение к серверу, задавая IP-адрес (127.0.0.1) и порт.
3. Отправлять произвольную текстовую строку и получать эхо-ответ, используя функции send (или эквивалент) и recv.
4. Корректно обрабатывать ошибки: невозможность соединения, разрыв связи, некорректный порт.
5. Осознавать различие между отправкой строк (последовательность символов) и бинарных байтовых буферов.
6. Захватывать трафик взаимодействия в Wireshark и идентифицировать отправленные и полученные данные.

2. Теоретические сведения

TCP-сокет – конечная точка соединения, идентифицируемая IP-адресом и номером порта. Для клиента последовательность системных вызовов (в нотации Berkeley Sockets) такова: socket() → connect() → send()/recv() → close().

Основные функции (C/POSIX):

- int socket(int domain, int type, int protocol) – создаёт сокет. Для TCP: AF_INET, SOCK_STREAM, 0.
- int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen) – устанавливает соединение с сервером. Структура sockaddr_in заполняется адресом (для localhost – inet_addr("127.0.0.1")) и портом (htons(port)).

- ssize_t send(int sockfd, const void *buf, size_t len, int flags) / ssize_t recv(int sockfd, void *buf, size_t len, int flags) – передача данных. Работают с байтовыми буферами, возвращают количество реально отправленных/принятых байт.
- close(sockfd) (Linux) / closesocket(sockfd) (Windows после вызова WSAStartup и с линковкой ws2_32).

В Python интерфейс аналогичен:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", port))
s.sendall(data.encode())   # или s.send()
response = s.recv(1024)
s.close()
```

Эхо-протокол подразумевает, что сервер отсылает обратно всё, что получил. Для тестирования эхо-клиента может использоваться:

- готовый эхо-сервер, предоставленный преподавателем;
- утилита ncat (Netcat) в режиме эхо: ncat -l 12345 --keep-open --exec /bin/cat (Linux) или ncat -l -p 12345 --keep-open --exec cmd /c "more < con" (Windows, творчески);
- простейший Python-сервер.

Обработка ошибок: каждая сетевая операция может завершиться неудачей. Необходимо проверять возвращаемые значения, использовать perror() / strerror(errno) (C) или перехватывать исключения socket.error (Python). Типичные ошибки: сервер недоступен (ECONNREFUSED), превышен таймаут, сброс соединения.

Отличие строки от байтового буфера: сетевая передача работает с октетами. Символьная строка должна быть преобразована в последовательность байт согласно кодировке (ASCII, UTF-8). При приёме возможна фрагментация: recv может вернуть меньше байт, чем ожидалось. Для учебного эхо-клиента на коротких строках это допустимо, но полезно знать.

3. Задание на выполнение работы

1. **Подготовка тестового окружения:**
    - Убедиться, что на компьютере запущен эхо-сервер, написанный преподавателем, или запустить простейший эхо-сервер самостоятельно (например, командой ncat -l 12345 --keep-open --echo в Linux или с использованием Python-скрипта echo_server.py). Зафиксировать порт (предположим, 12345).
2. **Разработка клиента (общая логика):**

- Запросить у пользователя строку для отправки.
- Создать TCP-сокет.
- Установить соединение с 127.0.0.1 и портом сервера.
- Преобразовать строку в байтовый массив (кодировка UTF-8 или ASCII).
- Отправить данные через send() (или sendall() в Python).
- Получить ответ в буфер фиксированного размера (например, 1024 байта).
- Вывести полученный ответ на экран (с преобразованием обратно в строку).
- Закрыть сокет.

3. **Обработка ошибок:**
   - Если соединение не удалось (неправильный порт, сервер не запущен) – вывести осмысленное сообщение и завершить программу без аварийного краша.
   - Если send или recv вернули ошибку – вывести диагностику.
   - Предусмотреть случай, когда сервер закрыл соединение (возврат 0 от recv).
4. **Взаимодействие с Wireshark:**
   - Запустить захват трафика на loopback-интерфейсе (обычно «Loopback: lo» в Linux, в Windows может потребоваться Npcap и установленный адаптер замыкания).
   - Выполнить несколько сеансов связи (отправить разные строки).
   - Применить фильтр tcp.port == 12345, найти свои сообщения. Убедиться, что передаваемые данные соответствуют отправленным.
5. **Дополнительное задание (по желанию, для углубления):**
   - Реализовать многократную отправку строк в цикле, пока пользователь не введёт "exit". Для каждой итерации использовать один и тот же сокет (постоянное соединение). Обеспечить корректное завершение.

4. Методические указания по выполнению

- **Для C (Linux/WSL):**
  Подключите заголовки <sys/socket.h>, <arpa/inet.h>, <unistd.h>. Пример создания сокета:

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) { perror("socket"); exit(1); }
struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(12345);
inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) { ... }
```

- **Для C (Windows):**
  Необходимо инициализировать Winsock вызовом WSAStartup(0x202, &wsaData), использовать closesocket(), линковать ws2_32.lib. Все функции аналогичны, но с SOCKET типом и INVALID_SOCKET. Документация MSDN содержит точные сигнатуры.
- **Для Python:**
  Не требуется линковка, используйте модуль socket. Рекомендуется with socket.socket(...) as s: для гарантированного закрытия, либо try: ... finally: s.close(). Для отправки всей строки надёжно использовать sendall(), для приёма – recv(1024). Кодировка: data.encode('utf-8'), response.decode('utf-8').
- **Запуск эхо-сервера для самопроверки:**
  Простейший вариант на Python (сохранить как echo_server.py):

```python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('127.0.0.1', 12345))
s.listen(1)
print("Echo server on port 12345")
conn, addr = s.accept()
while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

Тогда клиент должен корректно отрабатывать завершение сервером (закрытие после первого ответа или после обрыва).

- **Отладка:**
  Вставляйте отладочную печать переданных/принятых байт. Используйте Wireshark для проверки корректности отправки.

5. Содержание отчёта

1. **Цель работы** и краткое описание задачи.
2. **Листинг клиентской программы** с комментариями (на выбранном языке).
3. **Скриншоты, демонстрирующие:**
    - запуск программы и успешный обмен с эхо-сервером (консольный вывод);
    - ситуацию с ошибкой (например, неверный порт) и обработку этой ошибки;
    - захват трафика в Wireshark с фильтром по порту, с отображением отправленных и полученных данных.
4. **Описание тестов:** какие тестовые сценарии были выполнены (успешный обмен, неверный порт, остановка сервера во время сеанса и т.п.).
5. **Анализ результатов:** пояснение работы программы на основе показанных скриншотов; объяснение различия между отправкой строки и байтами на примере Wireshark.
6. **Выводы** (чему научились, какие трудности возникли).

6. Контрольные вопросы

1. Какие системные вызовы (функции) необходимы для реализации TCP-клиента, и в каком порядке они вызываются?
2. Какую роль играет функция connect? Что произойдёт, если сервер не прослушивает указанный порт?
3. В чём разница между функциями send и write (или send и sendall в Python) при работе с сокетами?
4. Почему после приёма данных функцией recv может быть принято меньше байт, чем было отправлено? Как с этим бороться в надёжном клиенте?
5. Каким образом клиент узнаёт, что сервер закрыл соединение? Как это обрабатывается в коде?
6. Для чего нужны функции преобразования порядка байт (htons, inet_pton)? Можно ли обойтись без них на localhost?
7. Как Wireshark помогает отладить взаимодействие клиента и сервера? Какие данные можно увидеть в полях TCP-пакета?

7. Критерии оценки (конкретизированные для ЛР №2)

Оценка	Критерии
Зачтено	Программа компилируется / запускается, устанавливает соединение с тестовым эхо-сервером (на localhost), отправляет пользовательскую строку, принимает корректный эхо-ответ и выводит его. При указании неверного порта или отсутствии сервера программа выводит осмысленное сообщение об ошибке и завершается, не вызывая краха системы. Код структурирован, содержит пояснительные комментарии. Отчёт включает все требуемые разделы и скриншоты (в т.ч. Wireshark). Студент защищает работу: объясняет назначение каждой сетевой функции, отвечает на контрольные вопросы, может модифицировать код по просьбе преподавателя (например, изменить порт, вывести длину полученных данных).
Не зачтено	Программа не запускается, выдаёт ошибки компиляции / синтаксиса. Соединение с сервером не устанавливается, студент не может исправить базовые ошибки. Отсутствует обработка ошибок (приложение «падает» без пояснений). Отчёт не оформлен, нет скриншотов трафика, студент не понимает разницы между send и recv, не может объяснить свой код.

Лабораторная работа №3

Тема: Итеративный TCP-эхо-сервер

Цель работы:
Изучить серверную часть сетевого взаимодействия на TCP. Освоить системные вызовы socket(), bind(), listen(), accept(), recv()/send() и close(). Научиться создавать сервер, способный последовательно обслуживать одного клиента за другим. Понять ограничения итеративной модели обслуживания.

Формируемые компетенции (продолжение):

- ПК-4: разработка серверного сетевого приложения, тестирование многопользовательского взаимодействия (последовательного), отладка сетевого кода.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Реализовать корректную последовательность вызовов для создания TCP-сервера: socket → bind → listen → accept → (recv/send в цикле) → close.
2. Связать (bind) сервер с конкретным портом на локальном адресе, предварительно настроив сокет на повторное использование адреса (SO_REUSEADDR).
3. Организовать бесконечный цикл приёма входящих соединений (accept), внутри которого выполнять чтение данных от клиента и отправку их обратно (эхо) до тех пор, пока клиент не разорвёт соединение.
4. Продемонстрировать на практике, что итеративный сервер не может одновременно обслуживать двух клиентов: пока обслуживается первое соединение, второе будет ожидать в очереди.
5. Обеспечить корректное завершение работы сервера по сигналу Ctrl+C (SIGINT) с освобождением слушающего сокета.
6. Протестировать сервер с помощью клиента из ЛР №2 либо стандартных утилит (telnet, netcat), подтвердить корректность эхо-обмена скриншотами и захватом трафика.

2. Теоретические сведения

Серверный сокет в TCP проходит через несколько состояний:

- socket() – создаёт конечную точку, аналогично клиенту.

- bind() – назначает сокету локальный адрес и порт. Используется структура sockaddr_in. Важно: после остановки сервера порт может оставаться в состоянии TIME_WAIT; флаг SO_REUSEADDR (через setsockopt) позволяет переиспользовать адрес без ожидания.
- listen() – переводит сокет в пассивный режим, задаёт максимальную длину очереди ожидающих соединений (backlog).
- accept() – извлекает очередное установленное соединение из очереди. Возвращает **новый сокет**, связанный именно с этим клиентом, и адрес клиента.
- Затем на новом сокете происходит обмен данными через recv()/send().
- После завершения работы с клиентом клиентский сокет закрывается, а слушающий сокет продолжает принимать новые соединения.
- Для завершения сервера слушающий сокет также закрывается (обычно в обработчике сигнала).

Итеративная модель означает, что сервер в каждый момент времени работает только с одним клиентом. После вызова accept() он входит во внутренний цикл обработки запросов клиента (например, эхо-повтор) и не возвращается к новому accept(), пока текущий клиент не отключится. Все остальные запросы на соединение в это время помещаются в очередь ожидания, размером до backlog. Это самый простой способ организации сервера, но он непригоден для параллельной работы с несколькими пользователями.

Корректное завершение: по Ctrl+C ядро посылает процессу сигнал SIGINT. По умолчанию процесс завершается, но сокеты могут остаться незакрытыми. Рекомендуется перехватывать сигнал и в обработчике выставлять флаг, по которому главный цикл аккуратно завершится и вызовет close() для слушающего сокета. В C это делается через signal() или sigaction(). В Python – через блок try: … except KeyboardInterrupt:.

Эхо-протокол: сервер читает данные, пока клиент не закроет соединение (recv возвращает 0), и каждую полученную порцию сразу отправляет обратно. Для коротких сообщений в учебном примере можно читать порциями до 1024 байт и отправлять.

3. Задание на выполнение работы

1. **Разработать итеративный эхо-сервер** (C или Python) со следующими деталями:
    - Порт задаётся в коде (например, 12345) или первым аргументом командной строки.
    - Перед bind() установить опцию SO_REUSEADDR.
    - Выполнить bind() на адрес INADDR_ANY (принимать соединения на любом интерфейсе) и указанный порт.

- Вызвать listen() с разумным backlog (например, 5).
- Войти в бесконечный цикл, где на каждой итерации вызывается accept().
- Получив новый клиентский сокет, вывести на консоль сообщение о подключении с адресом клиента.
- Во вложенном цикле читать данные от клиента (recv). Если получено 0 байт – клиент закрыл соединение, выход из внутреннего цикла. Иначе отправить полученные данные обратно (send).
- После выхода из внутреннего цикла закрыть клиентский сокет, вывести сообщение об отключении и продолжить главный цикл (accept).
- Предусмотреть обработку ошибок на всех этапах (с выводом осмысленных сообщений).

2. **Обеспечить корректное завершение по Ctrl+C:**
   - В C: установить обработчик SIGINT, который установит глобальную переменную volatile sig_atomic_t running = 0;. Главный цикл проверяет эту переменную; при её обнулении сервер закрывает слушающий сокет и завершается.
   - В Python: обернуть главный цикл в try: ... except KeyboardInterrupt: print("Shutting down"), после чего закрыть серверный сокет.
3. **Протестировать сервер:**
   - Запустить сервер в одном терминале.
   - Подключиться первым клиентом (telnet localhost 12345 или клиент из ЛР2). Отправить несколько строк, убедиться в эхо-ответе.
   - Не закрывая первую сессию, открыть второй терминал и попытаться подключиться вторым клиентом. Второй клиент должен зависнуть в ожидании (telnet покажет "Trying ...") или соединение будет принято, но ответа не будет до момента отключения первого. Зафиксировать это скриншотом.
   - Завершить сессию первого клиента (Ctrl+C в telnet или закрыть окно). Второй клиент сразу получит обслуживание. Проверить, что эхо работает и для второго.
   - Завершить сервер нажатием Ctrl+C в его терминале; убедиться, что порт освобождается и сервер можно немедленно перезапустить.
4. **Анализ трафика:**
   - Запустить Wireshark на loopback-интерфейсе, захватить трафик при тестировании. Найти пакеты установления соединения для двух клиентов,

обмен данными. Показать, что второй TCP-рукопожатие происходит (SYN идёт от клиента), но прикладной обмен данными начинается только после закрытия первого соединения (с точностью до буферизации TCP). В отчёте отразить эту последовательность.

4. Методические указания по выполнению

- **Установка SO_REUSEADDR:**
  В C:

```c
int opt = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

В Python:

```python
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Это необходимо для быстрого перезапуска сервера без ожидания таймаута TIME_WAIT.

- **Привязка к адресу:**
  Используйте INADDR_ANY (в Python пустую строку '' или '0.0.0.0'). Это позволит подключаться и с localhost, и с других хостов, если потребуется.
- **accept() и информация о клиенте:**
  В C accept() заполняет структуру sockaddr_in для клиента, из которой можно извлечь IP и порт с помощью inet_ntop и ntohs. В Python accept() возвращает кортеж (conn, address), где address – ('127.0.0.1', порт).
- **Внутренний цикл эхо:**
  В C:

```c
while ((n = recv(client_sock, buf, sizeof(buf), 0)) > 0) {
    send(client_sock, buf, n, 0);
}
if (n < 0) perror("recv");
close(client_sock);
```

  В Python:

```python
while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

Обратите внимание, что sendall в Python гарантирует отправку всех байт, а в C нужно проверять возвращаемое значение send и при необходимости организовывать цикл досылки, но для учебной работы допустимо send(fd, buf, n, 0) без досылки (при коротких сообщениях и локальном соединении фрагментация маловероятна).

- **Завершение по сигналу:**
    - C: пример обработчика:

```c
volatile sig_atomic_t keep_running = 1;
void handle_sigint(int sig) { keep_running = 0; }
// в main: signal(SIGINT, handle_sigint);
// главный цикл: while (keep_running) { ... accept (может быть прерван сигналом) }
// после цикла: close(server_sock);
```

Учтите, что accept – блокирующий вызов; если он ожидает соединения, сигнал прервёт его, но может потребоваться проверка errno == EINTR. Лучше после сигнала не вызывать accept снова. Простейший способ: в обработчике закрыть слушающий сокет, тогда accept вернёт ошибку и главный цикл завершится. Однако такой подход не всегда корректен; можно использовать флаг.

- Python:

```python
try:
    while True:
        conn, addr = s.accept()
        ...
except KeyboardInterrupt:
    print("Server shutting down.")
finally:
    s.close()
```

- **Демонстрация итеративности:**
  Запустите telnet в двух окнах. В первом отправьте строку, оставьте сессию открытой. Во втором telnet попытайтесь подключиться – либо соединение не установится (telnet будет ждать на этапе connect), либо будет принято (почти сразу) из-за backlog, но эхо работать не будет, пока первый не отключится. Важно объяснить, почему так происходит: сервер занят в recv для первого клиента, и не может выполнить новые recv для второго.

5. Содержание отчёта

1. **Цель работы** и описание реализованного сервера.
2. **Листинг кода сервера** с подробными комментариями.
3. **Инструкция по сборке и запуску** (команды компиляции/интерпретации).

4. **Результаты тестирования:**
    - Скриншот консоли сервера, показывающий подключение одного клиента, эхо-обмен и отключение.
    - Скриншот двух клиентов: первый активен, второй ожидает / не может обменяться данными до отключения первого.
    - Скриншот корректного завершения сервера по Ctrl+C без зависаний.
    - Захват Wireshark с отображением двух TCP-потоков (первый активен, второй – только рукопожатие или очередь). Подписи к пакетам, иллюстрирующие «бутылочное горлышко».
5. **Анализ работы:** объяснение итеративной модели и её ограничений; на основе скриншотов Wireshark показать разницу во времени обслуживания.
6. **Выводы** (чему научились, какие проблемы возникли, как можно улучшить сервер).

6. Контрольные вопросы

1. Перечислите основные системные вызовы для создания TCP-сервера и поясните назначение каждого.
2. Для чего нужна опция SO_REUSEADDR? Что произойдёт, если её не использовать?
3. Что делает функция listen и что означает параметр backlog?
4. Чем сокет, возвращаемый accept, отличается от слушающего сокета? Можно ли на нём вызвать listen?
5. Почему итеративный сервер не может обслуживать нескольких клиентов одновременно? Как бы вы это объяснили на примере очереди в магазине?
6. Как сервер узнаёт IP-адрес и порт подключившегося клиента? Покажите на примере своего кода.
7. Каким образом сервер определяет, что клиент закрыл соединение?
8. Как обработать сигнал Ctrl+C так, чтобы сервер завершил работу без утечки сокетов? Опишите два подхода.

7. Критерии оценки (конкретизированные для ЛР №3)

Оценка	Критерии
Зачтено	Сервер компилируется/запускается, успешно связывается с портом,

Оценка	Критерии
	принимает входящие соединения и реализует эхо-функцию. Продемонстрирована итеративная природа: при активном первом клиенте второй либо не может подключиться, либо не получает немедленного обслуживания. Сервер корректно обрабатывает обрыв связи клиентом и продолжает работу, завершается по Ctrl+C с закрытием слушающего сокета (нет необходимости вручную убивать процесс). Код снабжён комментариями, отчёт оформлен в соответствии с требованиями. Студент на защите объясняет назначение каждого системного вызова, умеет модифицировать код (например, изменить размер буфера, выводить адрес клиента в другом формате), иллюстрирует работу с помощью Wireshark.
Не зачтено	Программа не запускается, выдаёт ошибки на этапе компиляции/интерпретации. Сервер падает при подключении или отправке данных. Отсутствует обработка ошибок (нет проверки возвращаемых значений у socket/bind/listen). Сервер не завершается по Ctrl+C (приходится убивать через kill -9), не используется SO_REUSEADDR, после перезапуска порт занят. Студент не понимает разницы между bind и connect, не может объяснить, почему второй клиент ждёт. Отчёт не содержит скриншотов или они неинформативны.

Лабораторная работа №4

Тема: Многосоединный TCP-эхо-сервер (многопроцессная и многопоточная модели)

Цель работы:
Изучить способы параллельной обработки нескольких клиентов в рамках одного TCP-сервера. Освоить создание процессов (fork) в ОС Linux и потоков (pthread / std::thread в C++, threading в Python), а также их использование для одновременного обслуживания входящих соединений. Научиться защищать разделяемые ресурсы (консольный вывод) с помощью примитивов синхронизации и понимать ограничения модели «один клиент – один поток/процесс».

Формируемые компетенции (углубление):

- ПК-4: разработка многозадачных сетевых приложений, предотвращение состояний гонки, оценка ресурсоёмкости архитектурных решений.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Реализовать TCP-сервер, который при поступлении входящего соединения (accept) порождает отдельный процесс (fork) или поток (средствами целевой ОС/языка).
2. Передать созданному процессу/потоку дескриптор клиентского сокета и реализовать в нём эхо-логику (чтение – отправка обратно – закрытие при разрыве).
3. Обеспечить параллельное обслуживание не менее трёх клиентов одновременно; продемонстрировать, что они обмениваются данными независимо друг от друга.
4. Защитить общий ресурс – вывод на консоль сервера – с помощью мьютекса (критической секции) или эквивалентного механизма, чтобы исключить перемешивание сообщений от разных потоков/процессов.
5. Проанализировать и объяснить недостатки модели «один клиент – один поток/процесс» с точки зрения потребления памяти, накладных расходов на создание/переключение, масштабируемости.
6. Провести тестирование с тремя клиентами, захватив трафик в Wireshark и подтвердив параллельность обслуживания.

2. Теоретические сведения

Для одновременной работы с несколькими клиентами итеративный сервер непригоден. Альтернативой является создание **нового процесса** (системный вызов fork() в UNIX) или **нового потока** (функции pthread_create в POSIX, _beginthread/CreateThread в Windows, std::thread в C++11, модуль threading в Python).

- **Модель с fork()** (Linux/macOS):
  После успешного accept() процесс-родитель создаёт дочерний процесс вызовом fork(). Дочерний процесс наследует копию дескриптора клиентского сокета. Родитель **обязан закрыть** свою копию клиентского сокета сразу после fork(), чтобы не удерживать лишние ресурсы. Дочерний процесс работает с клиентом и завершается, закрывая сокет. Родитель продолжает принимать следующие соединения. Недостатки: создание процесса – дорогая операция, изоляция памяти не всегда нужна, неудобно организовать обмен данными между обработчиками.

- **Модель с потоками (pthread / std::thread / threading):**
  При новом соединении создаётся поток, которому в параметр передаётся дескриптор клиентского сокета (или указатель на структуру). Потоки разделяют адресное пространство процесса, поэтому важно синхронизировать доступ к общим данным. Обычно критическим ресурсом выступает консольный вывод: если несколько потоков одновременно вызывают printf/cout/print, вывод может быть перемешан. Для защиты применяют **мьютекс** (pthread_mutex_t, std::mutex, threading.Lock). Вокруг вывода на консоль ставятся блокировки lock/unlock. Потоки легковеснее процессов, но всё равно требуют отдельного стека и могут создавать заметные накладные расходы при тысячах одновременных клиентов (в этой работе ограничимся десятками).

- **Обработка зомби-процессов:**
  При использовании fork() родитель должен предотвращать появление зомби: либо игнорировать сигнал SIGCHLD (signal(SIGCHLD, SIG_IGN)), либо асинхронно вызывать waitpid(-1, NULL, WNOHANG) в цикле или обработчике. В учебной работе допустимо упростить до signal(SIGCHLD, SIG_IGN) для Linux, но нужно объяснить ограничения.

- **Переносимость:**
  В Windows вызов fork() отсутствует, поэтому рекомендуется использовать потоки. В Linux доступны оба варианта; допустимо реализовать сервер один раз с потоками (наиболее универсально). В Python кроссплатформенная реализация на threading также проще, но может быть ограничена GIL – для учебного эхо-сервера это не критично.

- **Ресурсоёмкость модели:**
  Каждый процесс/поток потребляет память под стек и служебные структуры ядра. Создание 1000 клиентов в модели 1:1 может исчерпать память или привести к резкому падению производительности из-за переключений. Альтернативы (пул потоков, асинхронный ввод-вывод) изучаются в последующих работах.

3. Задание на выполнение работы

1. **Разработать многопоточный (или многопроцессный) TCP-эхо-сервер** со следующими требованиями:
    - Использует знакомую из ЛР3 последовательность: socket() → setsockopt(SO_REUSEADDR) → bind() → listen ().
    - В бесконечном цикле вызывает accept(). При получении нового соединения:
        - для **процессной модели**: fork(), в дочернем процессе закрыть слушающий сокет (он не нужен), работать с клиентом, завершиться; родитель закрывает клиентский сокет и продолжает цикл.
        - для **поточной модели**: создать поток, передав ему дескриптор клиентского сокета (или структуру с ним и мьютексом для вывода). Поток выполняет функцию-обработчик, в которой ведётся эхо-цикл. Родительский поток не ждёт завершения, а сразу переходит к следующему accept. (В C для потоков pthread_detach или использовать std::thread с последующим detach(), в Python – thread.start() без явного ожидания).
    - Эхо-логика потока/процесса: чтение от клиента в цикле, отправка обратно; при recv == 0 закрытие сокета и завершение.
    - Предусмотреть обработку ошибок.
2. **Защита консольного вывода:**
    - Каждое сообщение о подключении, отключении или ошибке, выводимое из обработчика, должно быть защищено мьютексом.
    - Реализовать функцию-обёртку для вывода (например, safe_print(const char* msg)), которая захватывает мьютекс, выводит сообщение и освобождает мьютекс.
    - В C использовать pthread_mutex_t (с инициализацией) или std::mutex; в Python – threading.Lock.
3. **Обработка зомби (для fork):**
    - Если выбран процессный подход в Linux, перед главным циклом вызвать signal(SIGCHLD, SIG_IGN) или предусмотреть периодический сбор waitpid.
4. **Тестирование параллельности:**
    - Запустить сервер.
    - Открыть три (или более) окна терминала и одновременно подключиться к серверу через telnet (или клиент из ЛР2).

- В каждом окне отправить несколько строк, чередуя отправки, чтобы показать, что сервер обрабатывает всех независимо.
- На серверной консоли должны быть видны чёткие, не перемешанные строки сообщений о подключениях и, возможно, о полученных данных (если сервер логирует полученные эхо-строки).
- Захватить трафик Wireshark на loopback, отфильтровать по порту, продемонстрировать несколько параллельно существующих TCP-потоков с обменом данными.

5. **Дополнительный анализ:**
   - Оценить, сколько примерно потоков/процессов может создать такая модель на вашем компьютере до исчерпания ресурсов (например, попробовать в цикле создавать соединения и наблюдать за памятью). Кратко описать в выводах.

4. Методические указания по выполнению

- **Выбор языка и модели:** Рекомендуется C или Python на Linux; по согласованию с преподавателем – C++ с std::thread или Windows с _beginthread. В любом случае придерживайтесь общей структуры.
- **Пример многопоточного сервера на C (pthread):**

```c
#include <pthread.h>
#include <stdio.h>
// ... другие заголовки
pthread_mutex_t console_lock = PTHREAD_MUTEX_INITIALIZER;

void *client_handler(void *arg) {
    int client_fd = *(int*)arg;
    free(arg);  // если выделяли память под дескриптор
    char buf[1024];
    int n;
    while ((n = recv(client_fd, buf, sizeof(buf), 0)) > 0) {
        send(client_fd, buf, n, 0);
    }
    close(client_fd);
    pthread_mutex_lock(&console_lock);
    printf("Client disconnected\n");
    pthread_mutex_unlock(&console_lock);
    return NULL;
}

int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    // ... setsockopt, bind, listen ...
    while (1) {
        struct sockaddr_in client_addr;
        socklen_t len = sizeof(client_addr);
        int client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &len);
        if (client_fd < 0) { perror("accept"); continue; }
        pthread_mutex_lock(&console_lock);
        printf("New client connected\n");
        pthread_mutex_unlock(&console_lock);
        int *pfd = malloc(sizeof(int));
        *pfd = client_fd;
        pthread_t tid;
        pthread_create(&tid, NULL, client_handler, pfd);
        pthread_detach(tid);   // чтобы не ждать join
    }
}
```

Примечание: не забывайте линковать с -pthread.

- **Пример на Python (threading):**

```python
import socket, threading

lock = threading.Lock()
def handle(conn, addr):
    with lock: print(f"Connected {addr}")
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.sendall(data)
    conn.close()
    with lock: print(f"Disconnected {addr}")

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(('', 12345))
s.listen(5)
try:
    while True:
        conn, addr = s.accept()
        t = threading.Thread(target=handle, args=(conn, addr))
        t.daemon = True
        t.start()
except KeyboardInterrupt:
    s.close()
```

- **Для fork (Linux, C):**
  После accept вызвать fork(). Если возврат равен 0 (дочерний процесс): закрыть серверный сокет (close(server_fd)), выполнить эхо-цикл с client_fd, закрыть клиентский сокет и вызвать exit(0). Родительский процесс закрывает client_fd и продолжает. Не забудьте signal(SIGCHLD, SIG_IGN) в main до цикла. Обратите внимание: дочерний процесс не должен возвращаться в цикл accept.
- **Синхронизация вывода:**
  В Python используйте with lock: print(...). В C для printf обязательно блокировать мьютекс до вызова и разблокировать после. В C++ – std::lock_guard<std::mutex> lock(cout_mutex); cout << ... << endl;.
- **Тестирование:**
  Откройте три вкладки терминала. В каждой выполните telnet 127.0.0.1 12345. Убедитесь, что во всех можно печатать и сразу видеть эхо. Команды должны отрабатывать без задержек, вызванных другими клиентами.

5. Содержание отчёта

1. **Цель работы** и выбранная модель (потоки / процессы) с обоснованием.
2. **Листинг сервера** с комментариями; отдельно выделены функции синхронизации.
3. **Инструкции по сборке и запуску.**
4. **Результаты тестирования:**
    - Скриншот консоли сервера с сообщениями о подключении/отключении трёх клиентов без перемешивания строк.
    - Скриншоты (или фрагменты) трёх клиентов, демонстрирующие независимый эхо-обмен.
    - Wireshark-скриншот, показывающий несколько параллельных TCP-потоков с обменом данными (можно выделить цветом).
5. **Анализ:** описание, как именно достигается параллельность; объяснение использования мьютекса; замер или оценка ресурсоёмкости модели.
6. **Выводы** (что освоено, достоинства и недостатки модели, возможные улучшения).

6. Контрольные вопросы

1. Какие два основных способа параллельной обработки клиентов вы знаете? В чём их принципиальные различия?
2. Что такое состояние гонки? Приведите пример гонки данных, которая могла бы возникнуть в сервере без мьютекса при выводе на консоль.
3. Как вы обеспечиваете корректное освобождение ресурсов после завершения потока/процесса? Что будет, если не отсоединить поток (pthread_detach / join)?
4. Почему при использовании fork() важно закрывать клиентский сокет в родительском процессе сразу после ветвления?
5. Как вы предотвратили появление процессов-зомби в случае fork-реализации?
6. Сравните модели «один клиент – один поток» и «один клиент – один процесс» по быстродействию и памяти. Какие альтернативы существуют для высоконагруженных серверов?
7. Объясните, как Wireshark подтверждает параллельность обслуживания. Какие фильтры вы применили?
8. В чём ограничение Python GIL для многопоточного сервера? Влияет ли оно на эхо-сервер?

7. Критерии оценки

Оценка	Критерии
Зачтено	Сервер корректно принимает соединения, создаёт поток/процесс для каждого клиента, реализует эхо-функцию в параллельном режиме. Демонстрируется одновременная работа не менее трёх клиентов (по скриншотам или на защите). Консольный вывод сервера защищён мьютексом, строки не накладываются. Студент может объяснить, как организована синхронизация, как обрабатываются зомби (если fork) или отсоединяются потоки. Код структурирован, ошибки обрабатываются. В отчёте присутствуют все требуемые элементы, включая Wireshark-трафик с несколькими потоками. Студент способен модифицировать код (например, добавить логирование в файл с тем же мьютексом) и отвечает на контрольные вопросы.
Не зачтено	Сервер не запускается или падает при подключении более одного клиента. Отсутствует синхронизация вывода, наблюдается перемешивание сообщений. Нет подтверждения параллельной работы трёх клиентов (например, третий клиент зависает). Утечка ресурсов: потоки не отсоединяются, зомби не устраняются. Студент не может объяснить работу мьютекса, разницу между процессом и потоком, не осознаёт ограничения модели. Отчёт неполный, скриншоты отсутствуют или не отражают параллельность.

Лабораторная работа №5

Тема: Чат на базе UDP-ретранслятора

Цель работы:
Освоить программирование дейтаграммных сокетов (UDP) с использованием функций sendto() и recvfrom(). Понять особенности ненадёжной передачи данных без установления соединения. Реализовать простой чат-сервер, ретранслирующий сообщения между несколькими клиентами, и клиентское приложение, способное одновременно отправлять и принимать сообщения.

Формируемые компетенции (углубление):

- ПК-4: разработка, отладка и оценка характеристик сетевых приложений на базе UDP.
- ПК-9: оформление результатов тестирования и написание отчёта.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Создать UDP-сокет (SOCK_DGRAM) и выполнить привязку к адресу на стороне сервера (bind).
2. Использовать sendto() для отправки датаграммы конкретному адресату и recvfrom() для получения датаграммы с информацией об отправителе.
3. Реализовать сервер-ретранслятор, который накапливает список адресов клиентов (по первому полученному от каждого сообщению) и при получении новой датаграммы пересылает её всем остальным зарегистрированным клиентам.
4. Разработать чат-клиент с возможностью одновременного ввода сообщений с клавиатуры и приёма входящих датаграмм от сервера (с использованием двух потоков либо асинхронного ввода-вывода).
5. Корректно обрабатывать ошибки вызовов sendto/recvfrom и нетативные ситуации (закрытие сокета, прерывание по Ctrl+C).
6. Сравнить сетевой обмен в UDP и TCP, объяснить смысл «ненадёжности»: потенциальные потери, дублирование и нарушение порядка датаграмм.
7. Захватить трафик в Wireshark, продемонстрировать структуру UDP-датаграмм и отсутствие рукопожатия.

2. Теоретические сведения

UDP (User Datagram Protocol) – транспортный протокол без установления соединения. Он не гарантирует доставку, сохранение порядка и защиту от дублирования, но обеспечивает минимальные накладные расходы и низкую задержку. Каждая датаграмма – независимое сообщение.

При программировании UDP-сокетов используются:

- socket(AF_INET, SOCK_DGRAM, 0) – создание дейтаграммного сокета.
- sendto(sockfd, buf, len, flags, dest_addr, addrlen) – отправка данных по указанному адресу.
- recvfrom(sockfd, buf, len, flags, src_addr, &addrlen) – получение датаграммы с заполнением структуры адреса отправителя.

В отличие от TCP, connect() для UDP не обязателен (можно вызвать для фиксации удалённого адреса, но это ограничит связь одним собеседником). Сервер обычно использует bind() для фиксации своего порта, клиент может не вызывать bind – ядро назначит случайный порт при первой отправке.

Сервер-ретранслятор должен запоминать адреса активных клиентов. Простейший способ: при получении первой датаграммы от клиента его адрес добавляется в список (если ещё не добавлен). При получении очередного сообщения оно пересылается всем клиентам, кроме отправителя. Адреса хранятся в виде массива структур sockaddr_in (или эквивалента в Python). Такой сервер не требует многопоточности – он обрабатывает датаграммы последовательно.

Клиент чата должен одновременно читать ввод пользователя и принимать сообщения от сервера. Прямолинейный подход – использовать два потока:

- Поток-отправитель ожидает ввод с клавиатуры и вызывает sendto().
- Поток-приёмник в бесконечном цикле вызывает recvfrom() и выводит полученные сообщения на экран.
  Необходима синхронизация доступа к консольному выводу (мьютекс), чтобы сообщения не перемешивались с вводимым текстом (в учебных целях можно выводить с новой строки и допускать небольшие артефакты). Альтернатива – использование системного вызова select() для одновременного ожидания событий на stdin и сокете (будет подробнее рассматриваться в ЛР9).

Ненадёжность UDP проявляется в реальной сети, но на локальной петле потери практически исключены. Для демонстрации можно симулировать потерю пакетов: на сервере перед пересылкой генерировать случайное число и с вероятностью 10% не выполнять sendto. Это позволит увидеть пропажу сообщений и обсудить необходимость контроля целостности в реальных приложениях.

3. Задание на выполнение работы

1. **Реализовать UDP-сервер ретрансляции:**
    - Создать дейтаграммный сокет, привязать к порту (например, 12345) и адресу 0.0.0.0.
    - Поддерживать динамический список зарегистрированных клиентов (массив структур адресов).
    - В бесконечном цикле принимать датаграммы recvfrom().
    - Если адрес отправителя отсутствует в списке – добавить его. Логировать подключение нового клиента в консоль.
    - Полученное сообщение (текст) переслать с помощью sendto() всем клиентам из списка, **кроме отправителя**.
    - Предусмотреть корректное завершение по Ctrl+C: закрыть серверный сокет, вывести список активных адресов (опционально).
    - (Опционально) Для демонстрации ненадёжности добавить случайную потерю датаграмм при пересылке (с вероятностью 10–20 %), выводя предупреждение «packet dropped».
2. **Реализовать чат-клиент:**
    - Создать UDP-сокет (привязка к порту не обязательна).
    - Запустить поток приёма: бесконечно вызывать recvfrom() и выводить полученный текст на экран вместе с отправителем (например, IP:порт).
    - В главном потоке в цикле считывать строки с клавиатуры и отправлять их на сервер через sendto(). Выход по пустой строке или специальной команде /quit.
    - При завершении отправки закрыть сокет (поток приёма должен корректно завершиться, получив ошибку или пустую датаграмму).
    - Для защиты консоли от перемешивания ввода и вывода использовать мьютекс либо применить простой приём: перед выводом печатать символ \r и очищать текущую строку (на усмотрение студента).
3. **Протестировать чат:**
    - Запустить сервер.
    - Запустить двух клиентов (в разных терминалах).
    - Отправить несколько сообщений с каждого клиента; убедиться, что каждое сообщение от одного клиента отображается у другого.
    - Проверить, что сообщение не возвращается отправителю.

- При включённой симуляции потерь зафиксировать случаи, когда сообщение не доставлено.
- Захватить трафик в Wireshark на loopback-интерфейсе, применить фильтр udp.port == 12345, выделить датаграммы, просмотреть поля UDP-заголовка и отсутствие подтверждений. Сравнить с TCP-дампом из предыдущих работ.

4. **Сравнить с TCP:**
   В выводах к работе объяснить, почему в UDP не требуется рукопожатие, как отсутствие соединения сказывается на поведении сервера (сервер не знает, «жив» ли клиент), и почему модель сервер-ретранслятор не требует многопоточности.

4. Методические указания по выполнению

- **Создание UDP-сокета (C):**

```c
int sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) { perror("socket"); exit(1); }
```

Для сервера выполнить bind(). Пример заполнения структуры адреса:

```c
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(12345);
bind(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
```

- **Приём и отправка:**

```c
struct sockaddr_in client_addr;
socklen_t addr_len = sizeof(client_addr);
char buf[1024];
int n = recvfrom(sock, buf, sizeof(buf)-1, 0,
                 (struct sockaddr *)&client_addr, &addr_len);
if (n > 0) {
    buf[n] = '\0';
    // переслать другим клиентам ...
    sendto(sock, buf, n, 0, (struct sockaddr *)&target_addr, sizeof(target_addr));
}
```

- **Хранение адресов клиентов (C):**
  Можно использовать массив фиксированного размера (например, struct

sockaddr_in clients[10]) и переменную-счётчик num_clients. Для простоты сравнения адресов сравнивать sin_addr.s_addr и sin_port. Не забывать о порядке байт.

- **Клиент с потоками (Python):**

```python
import socket, threading, sys

def receiver(sock):
    while True:
        try:
            data, addr = sock.recvfrom(1024)
            print(f"\n[From {addr}] {data.decode()}")
        except:
            break

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server = ('127.0.0.1', 12345)
t = threading.Thread(target=receiver, args=(sock,))
t.daemon = True
t.start()
for line in sys.stdin:
    msg = line.strip()
    if msg == '/quit': break
    sock.sendto(msg.encode(), server)
sock.close()
```

В C аналогично используются pthread.

- **Симуляция потери (сервер):**
  Перед пересылкой добавить:
  if (rand() % 10 < 2) { printf("Simulated drop\n"); continue; }
  Убедиться, что генератор инициализирован srand(time(NULL)).
- **Завершение сервера:**
  Обработать SIGINT, выставить флаг, закрыть сокет, завершить цикл.
- **Wireshark:**
  Фильтр udp.port == 12345. Сравнить длину заголовков UDP и TCP (8 байт против 20). Обратить внимание на отсутствие пакетов SYN/ACK.

5. Содержание отчёта

1. **Цель работы** и краткое описание UDP-чата.
2. **Листинги сервера и клиента** с пояснительными комментариями; указание особенностей работы с UDP.

3. **Инструкции по сборке и запуску.**
4. **Результаты тестирования:**
    - Скриншоты консолей двух клиентов, показывающие успешный обмен сообщениями.
    - Логи сервера с отметками о подключении/отключении.
    - При реализации симуляции потерь – скриншоты с пропавшими сообщениями.
    - Захват Wireshark с отфильтрованными UDP-датаграммами; выделение полей UDP-заголовка.
5. **Сравнительный анализ** UDP vs TCP на основе проведённого эксперимента: надёжность, порядок, дублирование, накладные расходы, сценарии применения.
6. **Выводы** (освоенные навыки, проблемы, идеи по улучшению протокола).

6. Контрольные вопросы

1. Чем различаются системные вызовы sendto()/recvfrom() и send()/recv()? Почему для UDP требуется указывать адрес в каждой операции?
2. Каким образом UDP-сервер различает клиентов? Как он узнаёт их адреса?
3. Что произойдёт, если две датаграммы от одного клиента придут в изменённом порядке? Гарантирует ли UDP правильный порядок?
4. Какие проблемы может вызвать потеря датаграммы в реальном чате? Как бы вы предложили их решать на уровне приложения?
5. Объясните структуру UDP-заголовка (порты, длина, контрольная сумма). Какую роль выполняет поле «длина»?
6. Почему UDP-серверу не требуется многопоточность для обслуживания сотен клиентов (в отличие от TCP)?
7. Можно ли использовать connect() для UDP-сокета? Что это даёт?

7. Критерии оценки (конкретизированные для ЛР №5)

Оценка	Критерии
Зачтено	Сервер ретранслирует сообщения между двумя и более клиентами, не возвращая их отправителю. Клиенты реализуют одновременную отправку и

Оценка	Критерии
	приём (потоки или select) без фатальных ошибок. Код обрабатывает типичные ошибки вызовов. Студент демонстрирует захват трафика, идентифицирует UDP-датаграммы и объясняет отсутствие подтверждений. На защите отвечает на вопросы о ненадёжности UDP, отличиях от TCP, структуре заголовка. Отчёт полный, содержит скриншоты обмена и анализ. При включённой симуляции потерь студент может объяснить поведение.
Не зачтено	Программа не запускается или не обеспечивает обмен (сообщения не доходят, сервер падает при подключении). Клиент не способен одновременно принимать и отправлять (зависает на вводе, не получая сообщений). Студент не понимает назначение sendto/recvfrom, не может исправить ошибки адресации. Отсутствует или неверен анализ UDP-трафика, нет сравнения с TCP.

Лабораторная работа №6

Тема: Надёжная передача файла по TCP с контролем целостности

Цель работы:
Разработать клиент-серверное приложение для передачи файла по протоколу TCP, обеспечивающее надёжную доставку и проверку целостности данных с помощью контрольной суммы CRC32. Освоить технику поблочной передачи файла, цикличной досылки данных, корректного закрытия соединения и обработку ошибок ввода-вывода на больших объёмах данных.

Формируемые компетенции (углубление):

- ПК-4: разработка надёжных сетевых протоколов, реализация контроля целостности (CRC32), обработка краевых случаев при передаче данных.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Разработать протокол передачи файла, включающий этапы: отправка метаданных (размер файла, контрольная сумма CRC32, имя файла) и передача содержимого файла блоками фиксированного размера.
2. На стороне клиента: открыть и прочитать файл в бинарном режиме, вычислить CRC32 всего файла, передать метаданные и содержимое серверу, гарантируя отправку каждого блока полностью (цикл send).
3. На стороне сервера: принять метаданные, затем принять ровно указанное количество байт содержимого и сохранить в файл; вычислить CRC32 полученных данных, сравнить с переданным значением и сообщить клиенту результат проверки.
4. Реализовать корректную обработку частичных отправок и приёмов: функции send и recv могут вернуть меньше байт, чем запрошено; необходимо в цикле доотправлять/дополучать данные.
5. Проверить работоспособность на файлах разного размера (от нескольких байт до нескольких мегабайт), продемонстрировав совпадение контрольных сумм; при намеренном искажении файла на сервере (невозможно при правильной TCP-доставке, но можно смоделировать изменением буфера) показать обнаружение ошибки.
6. Захватить трафик в Wireshark, проанализировать сегментацию данных на уровне TCP, подтвердить, что весь файл передан без искажений.

2. Теоретические сведения

Надёжная передача файла по TCP базируется на свойствах самого протокола TCP: гарантированная доставка без потерь и сохранение порядка. Однако TCP не различает границы сообщений – это поток байт. Поэтому прикладной протокол должен предусматривать способ выделения из потока начала и конца файла.

Типичный протокол:

1. Клиент посылает заголовок фиксированного формата, содержащий, например:
    - 4 байта – длина имени файла (N);
    - N байт – имя файла (в ASCII или UTF-8);
    - 8 байт – размер файла в байтах (в сетевом порядке);
    - 4 байта – контрольная сумма CRC32 исходного содержимого (в сетевом порядке).
2. Затем клиент отправляет содержимое файла ровно file_size байт.
3. Сервер, получив заголовок, ожидает ровно file_size байт данных, записывает их в файл, вычисляет CRC32 полученного потока и сравнивает с переданным. При совпадении отправляет клиенту ответ "OK", иначе "FAIL".

Вычисление CRC32:

- Контрольная сумма CRC32 широко используется для быстрой проверки целостности данных.
- В C/C++ удобно использовать библиотеку zlib (функция crc32): unsigned long crc = crc32(0L, Z_NULL, 0); crc = crc32(crc, buffer, length);.
- В Python есть встроенный модуль zlib и функция zlib.crc32(data), возвращающая целое без знака.
- Для передачи CRC32 по сети следует привести к фиксированному 4-байтовому значению в сетевом порядке байт (htonl / ntohl в C, в Python – модуль struct).

Циклическая отправка/приём:

- Системный вызов send(sock, buf, len, 0) может вернуть значение меньше len, если буфер отправки заполнен. Необходимо сдвигать указатель и уменьшать оставшийся размер:

```
int total = 0;
while (total < len) {
    int sent = send(sock, buf + total, len - total, 0);
    if (sent < 0) { /* ошибка */ break; }
    total += sent;
}
```

- Аналогично для recv: при чтении ожидаемого количества байт может потребоваться несколько вызовов. Полезно написать вспомогательную функцию recv_all(sock, buf, len).

Сохранение файла сервером: после приёма всех байт следует открыть файл на запись в бинарном режиме и записать полученный буфер (или писать порциями прямо во время приёма, если файл очень большой). Для упрощения можно накапливать данные в буфере, если размер файла разумен (например, до 10 МБ), но корректнее – писать блоками сразу, не занимая лишнюю память.

Завершение соединения: после отправки ответа (OK/FAIL) сервер должен закрыть сокет, клиент после получения ответа – закрыть свой. Возможна также инициатива клиента: отправить файл, принять ответ, закрыть.

3. Задание на выполнение работы

1. **Спроектировать протокол:**
    - Зафиксировать структуру заголовка:
        - длина имени файла (2 или 4 байта, беззнаковое целое, network byte order);
        - имя файла (байты);
        - размер файла (8 байт, uint64_t, big-endian);
        - CRC32 (4 байта, uint32_t, big-endian).
    - (Или упрощённый вариант: сразу фиксировать размер файла и CRC32, без имени – сервер сохраняет в заранее заданное имя, например received.dat).
2. **Реализовать клиент:**
    - Принимает аргументы командной строки: IP сервера, порт, имя передаваемого файла.
    - Открывает файл, вычисляет его размер и CRC32.
    - Соединяется с сервером.
    - Отправляет заголовок с использованием цикличного send.
    - В цикле читает файл блоками (например, 4096 байт) и отправляет каждый блок, гарантируя полную отправку.
    - После отправки всех данных ждёт ответ от сервера (несколько байт), выводит результат проверки целостности.
    - Закрывает соединение.

3. **Реализовать сервер:**
    - Слушает порт (аргумент), принимает соединение.
    - Принимает заголовок, используя цикличный recv для всех полей.
    - Извлекает размер файла и CRC32, имя файла.
    - Создаёт/открывает файл для записи (бинарный режим).
    - Принимает ровно file_size байт данных, записывая их в файл (можно писать сразу при приёме, чтобы не занимать оперативную память для больших файлов).
    - Вычисляет CRC32 полученных данных (либо на лету, либо после сохранения прочитать файл и вычислить).
    - Сравнивает вычисленный CRC32 с полученным от клиента, выводит результат в консоль.
    - Отправляет клиенту ответ: "OK" или "FAIL" (например, 2 байта). Закрывает клиентский сокет.
    - Продолжает ожидать следующее соединение (многоразовый сервер, но для ЛБ6 достаточно однократного или с последовательной обработкой).
4. **Дополнительное задание (для усложнения):**
    - Реализовать возможность передачи нескольких файлов подряд в одном соединении.
    - Реализовать искусственное внесение ошибки на сервере (например, инвертировать один байт в принятых данных) для проверки реакции клиента.
5. **Тестирование:**
    - Подготовить текстовый файл и бинарный файл небольшого размера, передать их. Убедиться, что сервер сообщает «OK».
    - Передать файл достаточно большого размера (> 1 MB), проверить корректность.
    - Захватить трафик Wireshark, проанализировать TCP-сегменты; показать пакеты с заголовком и начало передачи данных.
    - Проверить устойчивость к частичным отправкам (можно искусственно уменьшить буфер сокета).

4. Методические указания по выполнению

- **Работа с бинарными данными:**
  Все поля заголовка должны быть упакованы в бинарном виде с фиксированным порядком байт. В C используйте htons, htonl, htobe64 (если доступно) или самодельную упаковку. В Python применяйте struct.pack('!I', value) для big-endian 4 байта, '!Q' для 8 байт.

- **Цикличная отправка:**
  Определите функцию send_all(int sock, const void *buf, size_t len), которая возвращает 0 при успехе, -1 при ошибке. Аналогично recv_all. В C пример:

```c
int send_all(int s, const char *buf, int len) {
    int total = 0, ret;
    while (total < len) {
        ret = send(s, buf + total, len - total, 0);
        if (ret < 0) return -1;
        total += ret;
    }
    return 0;
}
```

В Python можно использовать sock.sendall() (но для понимания принципа можно и свой цикл написать).

- **Вычисление CRC32 в C с zlib:**

```c
#include <zlib.h>
uLong crc = crc32(0L, Z_NULL, 0);
while ((n = fread(buf, 1, sizeof(buf), file)) > 0) {
    crc = crc32(crc, (const Bytef*)buf, n);
}
uint32_t crc_net = htonl((uint32_t)crc);
```

- **Приём файла сервером в потоковом режиме:**
  Вместо выделения буфера размером с файл, выделите буфер фиксированного размера (например, 8192) и читайте данные в цикле, сразу записывая в файл:

```
FILE *fp = fopen("received.dat", "wb");
uint64_t remaining = file_size;
while (remaining > 0) {
    size_t chunk = (remaining < BUFSIZE) ? remaining : BUFSIZE;
    if (recv_all(client_fd, buf, chunk) < 0) { /* ошибка */ break; }
    fwrite(buf, 1, chunk, fp);
    // одновременно обновляем CRC32, если делаем на лету
    crc = crc32(crc, (const Bytef*)buf, chunk);
    remaining -= chunk;
}
fclose(fp);
```

Это позволяет обрабатывать файлы произвольного размера.

- **Структура сообщений:**
  Убедитесь, что клиент и сервер одинаково интерпретируют размер полей. Лучше использовать типы фиксированной длины: uint16_t, uint32_t, uint64_t из <stdint.h>. При передаче размера файла 8 байт хватит для файлов до 16 ЭБ.
- **Имя файла:**
  Безопасность: не допускать выхода за пределы каталога (path traversal). Ограничьте имя базовым именем (basename). В учебной работе можно просто сохранять в received/ или текущую директорию, не используя переданное имя целиком, или проверять наличие /.
- **Wireshark:**
  Для наглядного отображения можно включить расшифровку данных как Raw или попытаться увидеть структуру заголовка в поле данных TCP. Примените фильтр tcp.port == <порт> и Follow TCP Stream.

5. Содержание отчёта

1. **Цель работы** и описание реализованного протокола передачи файла.
2. **Листинги клиента и сервера** с комментариями; выделены функции send_all, recv_all, вычисление CRC.
3. **Инструкция по сборке и запуску.**
4. **Результаты тестирования:**
   - Скриншоты консоли сервера и клиента при успешной передаче с сообщением «OK».
   - Скриншот с намеренной ошибкой (если реализована инверсия бита) и сообщением «FAIL».

- Wireshark-дамп с выделенным заголовком и фрагментом данных; объяснение пакетов.
- (По желанию) график сравнения времени передачи для файлов разного размера.

5. **Анализ:** обсуждение надёжности TCP + CRC32, возможные сценарии повреждения данных (в памяти, на диске), важность циклической досылки, практическая значимость контрольных сумм.
6. **Выводы** (чему научились, проблемы и способы их устранения).

6. Контрольные вопросы

1. Почему при использовании TCP всё равно нужна контрольная сумма на уровне приложения? Какие угрозы она помогает обнаружить?
2. Объясните, как работает ваш протокол: какие поля содержит заголовок и в каком порядке они передаются. Почему длину имени файла передают отдельно?
3. Для чего нужны функции циклической отправки и приёма? В каких случаях send может вернуть значение меньше запрошенного?
4. Как вы обеспечиваете независимость от порядка байт (endianness) в вашем протоколе? Приведите пример кода.
5. Как сервер понимает, что принял весь файл? Как он обрабатывает ситуацию, если клиент разорвал соединение досрочно?
6. В чём преимущество потоковой записи файла на сервере (без загрузки всего файла в память)? Когда это критично?
7. Какая функция библиотеки zlib используется для вычисления CRC32? Как инициализировать и накапливать сумму?
8. Предложите способы защиты от атаки «отказ в обслуживании» при передаче файлов (например, передача очень большого заявленного размера).

7. Критерии оценки (конкретизированные для ЛР №6)

Оценка	Критерии
Зачтено	Клиент и сервер корректно передают файл любого разумного размера, сервер сохраняет файл и проверяет CRC32. Результат проверки возвращается клиенту. В коде реализованы надёжные циклы отправки/приёма, исключающие потерю данных из-за частичных операций. Код комментирован, обработка ошибок присутствует. Студент демонстрирует передачу файла > 1 MB, совпадение CRC.

Оценка	Критерии
	На защите объясняет структуру протокола, причины использования циклических операций, может модифицировать код (например, добавить поле версии протокола). Отчёт содержит Wireshark-дамп с разбором полей заголовка и подтверждением целостности.
Не зачтено	Программа не компилируется или не передаёт файл целиком (теряются байты). Отсутствует проверка CRC32, сервер не сравнивает контрольные суммы. Не реализованы циклы досылки, в результате передача больших файлов обрывается. Сервер падает при попытке сохранить файл. Студент не понимает назначения CRC32 и не может объяснить формат заголовка. Отчёт неполный, нет никаких доказательств работы.

Лабораторная работа №7

Тема: Таймауты и обработка ошибок в сетевых приложениях

Цель работы:
Научиться управлять временными параметрами сокетных операций, устанавливать таймауты на приём и передачу данных, корректно интерпретировать коды ошибок и реализовывать устойчивое к разрывам поведение клиента (автоматическое переподключение). Сформировать понимание различий между временными (повторяемыми) и фатальными ошибками сетевого взаимодействия.

Формируемые компетенции (углубление):

- ПК-4: отказоустойчивость, обработка исключительных ситуаций в сетевых программах, повышение надёжности клиент-серверных систем.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Использовать опции SO_RCVTIMEO и SO_SNDTIMEO для задания предельного времени ожидания операций recv/send.
2. Обрабатывать ошибки с кодами EAGAIN/EWOULDBLOCK (таймаут) и отличать их от других ошибок (ECONNRESET, EPIPE, ETIMEDOUT).
3. Модифицировать TCP-клиент (на основе ЛР №2 или №6) таким образом, чтобы при обнаружении разрыва связи или истечения таймаута он предпринимал повторные попытки соединения с настраиваемой задержкой, ограничивая число попыток.
4. Настроить сервер так, чтобы он закрывал неактивные соединения после заданного таймаута ожидания данных.
5. Протестировать поведение системы в условиях искусственно создаваемых разрывов и задержек, фиксируя диагностическую информацию.
6. Обосновать различия между временными ошибками (допускают повтор операции) и фатальными (требуют пересоздания сокета).

2. Теоретические сведения

Необходимость таймаутов:
По умолчанию операции ввода-вывода на сокетах являются блокирующими. Если сервер зависает или сеть теряет связь, recv или send могут заблокировать процесс на неопределённое время. Установка таймаутов позволяет ограничить ожидание и перейти к альтернативной логике (повтор, закрытие, оповещение пользователя).

Опции таймаутов:

- SO_RCVTIMEO – задаёт максимальное время ожидания данных для операций чтения (recv, read). Если за указанное время не поступило ни одного байта, вызов возвращает ошибку (в Linux – EAGAIN/EWOULDBLOCK, но сокет остаётся в блокирующем режиме).
- SO_SNDTIMEO – аналогично для операций отправки. Если буфер отправки остаётся заполненным дольше указанного времени, send завершается с ошибкой.

Значение времени задаётся структурой struct timeval:

```
struct timeval tv;
tv.tv_sec = 5;   // секунды
tv.tv_usec = 0;  // микросекунды
setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

В Python аналогично:

```
sock.settimeout(5.0)  # таймаут в секундах на чтение и запись
```

При этом сокет остаётся блокирующим, но с ограниченным временем ожидания. Это отличается от неблокирующего режима, где операции возвращают EAGAIN немедленно, если данные недоступны.

Классификация ошибок:

- **Временные (восстанавливаемые):** EAGAIN / EWOULDBLOCK (таймаут или неблокирующий сокет), EINTR (прерывание сигналом). Эти ошибки не нарушают соединение, операцию можно повторить.
- **Фатальные (требующие переустановки соединения):** ECONNRESET (сброс партнёром), EPIPE (запись в закрытое соединение), ETIMEDOUT (истечение таймаута на уровне TCP при установке соединения – для connect), EBADF (недопустимый дескриптор). При их возникновении сокет становится непригоден, его необходимо закрыть и создать заново.

Логика повторного подключения:

- Клиент при обнаружении фатальной ошибки или возврате 0 из recv (плановый разрыв) закрывает сокет.
- Затем он в цикле пытается выполнить socket() → connect() с задержкой между попытками (например, 3 секунды). Количество попыток ограничено (5–10). При успешном соединении продолжается нормальная работа.
- Сервер также может применять таймаут: если клиент долго не присылает данные, сервер закрывает соединение, чтобы не тратить ресурсы.

Особенности connect() **с таймаутом:**
Напрямую установить SO_RCVTIMEO на connect нельзя (в POSIX). Типичный метод –

перевести сокет в неблокирующий режим, вызвать connect, и если он возвращает EINPROGRESS, использовать select или poll для ожидания готовности сокета к записи с заданным таймаутом. В учебной работе можно упростить: использовать цикл переподключения без таймаута на сам connect, но с задержкой между попытками, чтобы не перегружать систему.

3. Задание на выполнение работы

1. **Модифицировать клиент (например, эхо-клиент из ЛР №2 или клиент передачи файла):**
    - После создания сокета
      установить SO_RCVTIMEO и SO_SNDTIMEO (например, 5 секунд).
    - Реализовать отдельную функцию safe_recv() и safe_send(), которые обрабатывают ошибки: если errno == EAGAIN – выводить предупреждение «timeout, retrying» и повторять операцию (не более 2–3 раз), если ошибка фатальная – возвращать код ошибки.
    - В основном цикле после отправки запроса и ожидания ответа проверять возвращаемое значение: если recv вернул 0 или фатальную ошибку, переходить в режим переподключения.
    - **Режим переподключения:**
        - Закрыть старый сокет.
        - В цикле (до 5 попыток, с интервалом 3 секунды) создавать новый сокет и вызывать connect(). При успехе выходить из цикла и продолжать нормальную работу.
        - Выводить информационные сообщения о попытках.
    - Предусмотреть обработку сигнала Ctrl+C для корректного выхода даже во время ожидания переподключения.
2. **Модифицировать сервер (итеративный или многопоточный эхо-сервер):**
    - После accept для клиентского сокета установить SO_RCVTIMEO (например, 10 секунд).
    - В цикле чтения: если recv возвращает -1 с EAGAIN, вывести сообщение «read timeout, closing connection», закрыть сокет и продолжить ожидание следующего клиента.
    - Если recv возвращает 0 – корректное отключение клиента, просто закрыть сокет.
    - В случае других ошибок – также закрыть сокет и продолжить работу.

3. **Тестирование:**
    - Запустить сервер и клиент. Убедиться в обычном обмене данными.
    - Во время активного сеанса **остановить сервер** (Ctrl+C). Клиент должен обнаружить разрыв (recv вернёт ошибку или 0), вывести диагностику и начать попытки переподключения.
    - Снова запустить сервер; клиент должен успешно переподключиться и возобновить работу.
    - Искусственно создать задержку без разрыва: например, на сервере добавить sleep перед ответом дольше установленного таймаута клиента. Клиент должен зафиксировать таймаут (EAGAIN), повторить попытку (или выйти по превышению числа повторов), но не считать это фатальной ошибкой.
    - Захватить трафик Wireshark: найти пакеты RST при разрыве сервера, последующие попытки SYN от клиента. Показать разницу во времени между попытками.
    - Отчёт оформить с соответствующими скриншотами.

4. Методические указания по выполнению

- **Установка таймаута в C:**

```
struct timeval tv;
tv.tv_sec = 5;
tv.tv_usec = 0;
if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0)
    perror("setsockopt SO_RCVTIMEO");
```

После этого recv будет ждать максимум 5 секунд; если данных нет, вернёт -1 и errno == EAGAIN.

- **Безопасная функция приёма с учётом таймаута:**

```c
int recv_with_retry(int sock, char *buf, int len, int max_retries) {
    int retries = 0;
    int n;
    while (retries < max_retries) {
        n = recv(sock, buf, len, 0);
        if (n > 0) return n;  // успех
        if (n == 0) { /* соединение закрыто партнёром */ return 0; }
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            printf("Timeout, retry %d/%d\n", retries+1, max_retries);
            retries++;
            continue;
        }
        // другие ошибки
        perror("recv");
        return -1;
    }
    return -1; // превышено число повторов
}
```

- **Цикл переподключения клиента:**

```c
for (int attempt = 1; attempt <= MAX_ATTEMPTS; attempt++) {
    printf("Reconnecting attempt %d...\n", attempt);
    sleep(RECONNECT_DELAY);
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) { perror("socket"); continue; }
    // установка таймаутов на новый сокет
    if (connect(sock, ...) == 0) {
        printf("Reconnected successfully.\n");
        break;
    }
    close(sock);
    if (attempt == MAX_ATTEMPTS) {
        fprintf(stderr, "Max attempts reached, exiting.\n");
        exit(1);
    }
}
```

- **В Python:**

```
sock = socket.socket(...)
sock.settimeout(5.0)  # единый таймаут на recv/send (но можно отдельно через setsockopt)
try:
    data = sock.recv(1024)
except socket.timeout:
    print("recv timed out")
except ConnectionResetError:
    # переподключение
```

Для цикла переподключения:

```
import time
for i in range(1, 6):
    try:
        sock = socket.socket(...)
        sock.settimeout(5)
        sock.connect(('127.0.0.1', port))
        break
    except Exception as e:
        print(f"Attempt {i} failed: {e}")
        time.sleep(3)
else:
    sys.exit(1)
```

- **Различие временных и фатальных ошибок:**
  В C проверяйте errno. В Python – типы исключений (socket.timeout, ConnectionRefusedError, ConnectionResetError, BrokenPipeError). socket.timeout – аналог EAGAIN при таймауте.
- **Тестирование таймаута сервера:**
  Запустите сервер, подключитесь клиентом, но не отправляйте данные. Через 10 секунд сервер должен закрыть соединение.

5. Содержание отчёта

1. **Цель работы** и краткое описание реализованных механизмов.
2. **Листинги клиента и сервера** с акцентом на участки установки таймаутов, обработки ошибок и переподключения.
3. **Инструкция по сборке и запуску.**
4. **Результаты тестирования:**

- Скриншоты консоли клиента при нормальной работе, при разрыве сервера и последующем успешном переподключении (с видимыми сообщениями о попытках).
- Скриншот консоли сервера, показывающий детектирование таймаута клиента и закрытие соединения.
- Захват Wireshark: пакеты, иллюстрирующие разрыв (RST), таймауты, повторные SYN.

5. **Анализ:** таблица или описание классификации ошибок, встреченных в экспериментах; объяснение, почему повторное подключение улучшает надёжность.
6. **Выводы.**

6. Контрольные вопросы

1. Для чего нужны таймауты на сокетных операциях? Чем грозит их отсутствие?
2. Как установить таймаут на чтение из сокета? Опишите системный вызов и структуру.
3. Что означает ошибка EAGAIN (или EWOULDBLOCK) при использовании SO_RCVTIMEO? Является ли она фатальной?
4. Какие ошибки указывают на необратимый разрыв соединения? Как должен реагировать на них клиент?
5. Опишите алгоритм автоматического переподключения клиента. Почему между попытками вводится задержка?
6. Как сервер может защититься от «молчаливых» соединений, занимающих ресурсы?
7. В чём разница между установкой таймаута через setsockopt и использованием неблокирующих сокетов с select?
8. Можно ли применить SO_RCVTIMEO для connect? Если нет, то как реализовать таймаут подключения?
9. Объясните поведение Wireshark, которое вы наблюдали при разрыве и переподключении.

7. Критерии оценки

Оценка	Критерии
Зачтено	Клиент корректно устанавливает таймауты (либо через setsockopt, либо эквивалент), различает временные и фатальные ошибки. При разрыве соединения или таймауте клиент предпринимает повторные попытки подключения с контролируемой задержкой и ограничением числа попыток, что демонстрируется в ходе теста. Сервер обрабатывает таймаут ожидания и закрывает неактивные соединения. Программа не «зависает» при пропадании сети/сервера. Код содержит обработку ошибок и пояснения. Отчёт включает все необходимые скриншоты с подтверждением переподключения и анализ ошибок. Студент на защите правильно объясняет классификацию ошибок, может модифицировать параметры таймаутов и продемонстрировать результат.
Не зачтено	Программа не реагирует на разрыв (зависает). Таймауты не установлены или не работают, ошибки игнорируются (нет проверки возвращаемых значений). Отсутствует логика переподключения – после первой ошибки клиент просто завершается. Сервер не освобождает ресурсы при молчании клиента. Студент путает EAGAIN и ECONNRESET, не может объяснить, как работают таймауты. Отчёт не содержит скриншотов переподключения.

Лабораторная работа №8

Тема: Протокол с фиксированным бинарным заголовком (длина + тип)

Цель работы:
Спроектировать и реализовать собственный прикладной протокол, инкапсулирующий сообщения в кадры с фиксированным бинарным заголовком. Освоить сериализацию/десериализацию многобайтовых полей в сетевом порядке байт, корректное отделение сообщений от потока TCP, обработку ситуаций склейки и фрагментации пакетов, а также поддержку нескольких типов сообщений.

Формируемые компетенции (углубление):

- ПК-4: проектирование надёжных протоколов прикладного уровня, обработка неполных данных, обеспечение целостности сообщений.
- ПК-9: документирование структуры протокола.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Спроектировать формат кадра протокола, состоящий из фиксированного заголовка (например, 3 или 4 байта: длина тела + тип сообщения) и тела переменной длины.
2. Реализовать корректную упаковку полей заголовка в сетевой порядок байт (big-endian) на стороне отправителя и распаковку на стороне получателя.
3. Разработать клиент, который формирует и отправляет сообщения нескольких типов (текстовое эхо, запрос статуса, другое).
4. Разработать сервер, который в цикле читает из сокета заголовок фиксированного размера, извлекает длину и тип, затем принимает ровно указанное количество байт тела и выполняет действие в соответствии с типом сообщения.
5. Обеспечить обработку «склейки» пакетов (когда в буфере оказываются данные нескольких сообщений) и «разрыва» пакета (когда данные приходят частями, недостаточными для полного заголовка или тела).
6. Проверить работу протокола с помощью клиента и сервера, захватить трафик и показать структуру кадров в Wireshark.

2. Теоретические сведения

Проблема границ сообщений в TCP:
TCP – потоковый протокол, передающий непрерывную последовательность байт. Если приложение отправляет несколько сообщений подряд, они могут быть приняты получателем как одно склеенное целое или, наоборот, одно сообщение может быть

разбито на несколько фрагментов. Для восстановления границ сообщений прикладной протокол должен использовать один из методов кадрирования:

- фиксированная длина сообщений;
- разделители (например, \r\n в HTTP);
- заголовок с указанием длины (Length-prefixed framing).

В данной работе используется **заголовок с длиной** – самый гибкий и надёжный подход.

Структура фиксированного заголовка (пример):

Смещение (байты)	Размер	Поле	Описание
0	2	Length	Длина тела сообщения в байтах (без заголовка)
2	1	Type	Тип сообщения (0x01 – текстовое эхо, 0x02 – запрос статуса, 0x03 – ответ статуса и т.п.)

Таким образом, заголовок занимает фиксированные 3 байта (можно сделать 4 байта, если длина занимает 2 или 4 байта и тип 2 байта; выбирается студентом). Сервер всегда читает сначала эти 3 байта, извлекает Length, затем читает ровно Length байт тела.

Порядок байт:
Для переносимости многобайтовые целые (Length) передаются в сетевом порядке (big-endian). Используются функции htons/ntohs (для 2-байтовых) или htonl/ntohl (для 4-байтовых) в C, или модуль struct с префиксом ! в Python.

Техника надёжного чтения:
Нельзя полагаться, что recv сразу вернёт ожидаемое количество байт. Необходимо реализовать функцию, читающую ровно N байт (аналогичную recv_all из ЛБ6). Она вызывается дважды: сначала для заголовка (3 байта), затем для тела (Length байт). Это автоматически решает проблему «разрыва» пакета. После считывания одного сообщения можно снова читать заголовок следующего – что решает проблему «склейки».

Поддержка нескольких типов сообщений:
Сервер, получив значение Type, может вызвать соответствующую функцию-обработчик. Например:

- Type = 0x01: отправить тело обратно клиенту (эхо).
- Type = 0x02: вернуть клиенту сообщение со строкой «Server status: running», тип ответа 0x03.
  Это демонстрирует механизм команд.

Обработка ошибок:
Если при чтении заголовка или тела партнёр закрыл соединение (recv вернул 0) или произошла ошибка – сервер разрывает соединение. Клиент также должен уметь интерпретировать ответные сообщения сервера в том же формате.

3. Задание на выполнение работы

1. **Определить формат заголовка:**
    - Выберите размер поля длины (рекомендуется 2 байта, позволяющие тело до 65535 байт) и типы сообщений (минимум 2 типа: текстовое эхо и запрос/ответ статуса).
    - Опишите спецификацию протокола (в виде таблицы).
2. **Реализовать TCP-клиент:**
    - Устанавливает соединение с сервером.
    - В цикле предлагает пользователю выбрать тип сообщения (например, 1 – эхо, 2 – запрос статуса) и ввести текст (если нужно).
    - Формирует кадр: упаковывает длину тела (2 байта, big-endian) и тип (1 байт) в буфер, затем добавляет тело. Отправляет получившийся буфер целиком (можно sendall).
    - После отправки ожидает ответ от сервера в том же формате: читает заголовок (3 байта), извлекает длину и тип, затем читает тело и выводит его пользователю.
    - Предусмотреть корректное завершение (команда /quit или аналогичная).
3. **Реализовать TCP-сервер:**
    - Принимает соединение, входит в цикл обработки сообщений.
    - Реализует функцию recv_exact(sock, buffer, n), гарантирующую получение ровно n байт.
    - Для каждого сообщения:
        - Читает 3 байта заголовка с помощью recv_exact. Если соединение закрыто – выход.
        - Извлекает Length (первые 2 байта, преобразуя из сетевого порядка в хостовый) и Type (третий байт).
        - Проверяет Length на допустимость (например, не более 64 КБ).
        - Читает Length байт тела с помощью recv_exact.

        - Обрабатывает команду согласно Type:
            - 0x01: вывести полученный текст в консоль сервера и отправить клиенту кадр с тем же типом и телом (эхо).
            - 0x02: отправить клиенту кадр с типом 0x03 и телом «OK».
            - По умолчанию: отправлять кадр с типом 0xFF и сообщением «Unknown type».
    - После обработки продолжает ожидание следующего заголовка.
    - Предусмотреть корректное завершение сервера.
4. **Проверка целостности:**
    - Сервер должен проверять, что перед чтением тела заголовок полностью получен, и не доверять непроверенному значению Length (ограничение максимальной длины).
    - Если клиент отправляет некорректный заголовок (например, неверная длина), сервер должен разорвать соединение или вернуть ошибку.
5. **Тестирование:**
    - Запустить сервер. Подключить клиент.
    - Отправить несколько текстовых сообщений, убедиться в эхо-ответах.
    - Отправить запрос статуса, получить ответ.
    - Смоделировать склейку: клиент должен иметь возможность отправить несколько сообщений подряд без задержки. Сервер должен корректно разобрать их по порядку.
    - Захватить трафик Wireshark, выделить структуру кадра (заголовок и тело), показать шестнадцатеричное представление.
    - Проверить работу с большим сообщением (например, несколько килобайт текста), чтобы убедиться в правильной обработке фрагментации.

4. Методические указания по выполнению

- **Функция** recv_exact**:**
  В C:

```c
int recv_exact(int sock, void *buf, size_t len) {
    size_t total = 0;
    ssize_t n;
    while (total < len) {
        n = recv(sock, (char*)buf + total, len - total, 0);
        if (n <= 0) {
            if (n == 0) fprintf(stderr, "Connection closed\n");
            else perror("recv");
            return -1;
        }
        total += n;
    }
    return 0;
}
```

В Python можно обойтись sock.recv в цикле, так как recv в Python не гарантирует возврат ровно запрошенного количества.

- **Формирование заголовка (C, отправитель):**

```c
uint16_t len_host = (uint16_t)body_len;
uint16_t len_net = htons(len_host);
uint8_t type = 0x01;
// запись в буфер
memcpy(buf, &len_net, 2);
buf[2] = type;
memcpy(buf + 3, body, body_len);
send_all(sock, buf, 3 + body_len);
```

- **Разбор заголовка (C, получатель):**

```c
unsigned char header[3];
if (recv_exact(sock, header, 3) < 0) break;
uint16_t len_net;
memcpy(&len_net, header, 2);
uint16_t length = ntohs(len_net);
uint8_t type = header[2];
if (length > MAX_BODY_SIZE) { /* ошибка, закрыть */ break; }
char *body = malloc(length);
if (recv_exact(sock, body, length) < 0) { free(body); break; }
// обработка body в зависимости от type
free(body);
```

- **В Python with struct:**

```python
import struct
header = recv_exact(sock, 3)
length, msg_type = struct.unpack('!H B', header)  # ! - big-endian, H - uint16, B - uint8
body = recv_exact(sock, length)
```

Функция recv_exact для Python:

```python
def recv_exact(sock, n):
    data = b''
    while len(data) < n:
        chunk = sock.recv(n - len(data))
        if not chunk:
            raise ConnectionError("Socket closed")
        data += chunk
    return data
```

- **Обработка нескольких сообщений в одном буфере TCP:**
  Благодаря побайтовому чтению заголовка и тела мы естественным образом обрабатываем склейку: после завершения обработки одного сообщения цикл продолжает читать следующий заголовок из потока, даже если данные следующего сообщения уже пришли и буферизовались ОС.
- **Ограничение максимальной длины тела:**
  Для предотвращения атак и ошибок следует задать разумный максимум (например, 65535 байт, или меньше, если используете 2 байта на длину – это уже лимит). Если принятая длина превышает предел, сервер должен разорвать соединение.
- **Wireshark:**
  Для демонстрации можно применить фильтр по порту, затем в окне "Packet Bytes" выделить 3 байта заголовка и следующие Length байт тела. Можно также отметить, что в нешифрованном трафике видны наши структуры.

5. Содержание отчёта

1. **Спецификация протокола:** схема заголовка, описание типов сообщений, порядок байт.
2. **Листинги клиента и сервера** с выделением функций recv_exact, упаковки/распаковки заголовка.
3. **Инструкция по сборке и запуску.**
4. **Результаты тестирования:**
   - Скриншот консоли клиента с эхо-обменом и запросом статуса.

- Скриншот консоли сервера, показывающий получение команд разных типов.
- Wireshark: захват с фильтром, демонстрация двух кадров подряд (склейка), отображение шестнадцатеричного дампа заголовка и тела.
- Пример с длинным сообщением (фрагментация TCP).

5. **Анализ:** обсуждение проблем фрагментации и склейки, как они решены; сравнение с передачей файла (ЛБ6).
6. **Выводы.**

6. Контрольные вопросы

1. Почему в TCP необходимо явно задавать границы сообщений? Чем это отличается от UDP?
2. Объясните формат заголовка вашего протокола. Какие поля он содержит и в каком порядке байт передаются?
3. Как функция recv_exact решает проблему частичного приёма заголовка или тела?
4. Каким образом ваш сервер обрабатывает ситуацию, когда в одном TCP-сегменте пришло несколько сообщений (склейка)?
5. Что произойдёт, если клиент отправит значение Length, превышающее реальный размер тела? Как предотвратить чтение за пределами буфера?
6. Для чего используется ограничение максимальной длины тела? Какую уязвимость оно предотвращает?
7. Какие ещё методы кадрирования сообщений, кроме заголовка с длиной, вы знаете? Назовите примеры протоколов, их использующих.
8. Как бы вы расширили протокол для поддержки строковых команд (например, авторизация)?

7. Критерии оценки

Оценка	Критерии
Зачтено	Клиент и сервер реализуют оговоренный протокол с фиксированным бинарным заголовком, корректно обрабатывают сообщения разных типов. Сервер использует надёжное чтение recv_exact для заголовка и тела, извлекает длину в сетевом порядке байт, проверяет ограничения.

Оценка	Критерии
	Демонстрируется работа с эхо-сообщениями, запросом статуса, а также корректная обработка склейки (несколько сообщений подряд) и фрагментации (длинное сообщение). Код комментирован, ошибки обрабатываются. Отчёт включает спецификацию протокола, скриншоты и Wireshark-дамп с разбором. Студент на защите объясняет назначение каждого байта заголовка, может изменить формат (например, добавить поле) и продемонстрировать работоспособность без ошибок.
Не зачтено	Клиент/сервер не работает, не может разобрать заголовок, использует неправильный порядок байт (не конвертирует). Отсутствует recv_exact, в результате сервер путает сообщения или теряет данные. Обработка типов сообщений не реализована (все сообщения обрабатываются одинаково). Студент не понимает разницу между host и network byte order, не может объяснить, как сервер находит границы сообщений. Отчёт не содержит спецификации или демонстрации склейки.

Лабораторная работа №9

Тема: Мультиплексирование ввода-вывода с помощью select(). Однопоточный TCP-чат-сервер

Цель работы:
Освоить механизм мультиплексирования ввода-вывода select() для одновременного обслуживания множества клиентов в рамках одного потока. Реализовать однопоточный чат-сервер, способный принимать новые соединения и обрабатывать сообщения от всех активных клиентов, и сравнить такую архитектуру с многопоточной моделью по производительности и масштабируемости.

Формируемые компетенции (углубление):

- ПК-4: применение системного вызова select() для создания высокопроизводительных сетевых приложений, оценка ограничений и преимуществ модели неблокирующего ввода-вывода.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Настроить слушающий сокет в неблокирующем режиме (или использовать блокирующий сокет с select, что предпочтительнее для учебных целей).
2. Создать главный цикл обработки событий на основе select(): добавлять в набор fd_set слушающий сокет и все активные клиентские сокеты; на каждой итерации передавать в select максимальный дескриптор + 1 и три набора (чтение, запись, исключения) с таймаутом.
3. При готовности слушающего сокета к чтению — принять новое соединение (accept), перевести клиентский сокет в неблокирующий режим (обязательно для корректной работы) и добавить его в множество отслеживаемых.
4. При готовности клиентского сокета к чтению — прочитать доступные данные в пользовательский буфер, извлечь целые сообщения (например, строки, заканчивающиеся символом \n) и ретранслировать их всем остальным клиентам.
5. Обеспечить корректное удаление клиента из множества, если соединение разорвано (recv вернул 0) или произошла ошибка.
6. Реализовать однопоточный чат-сервер без создания потоков и процессов, обслуживающий не менее 10 одновременных клиентов.
7. Сравнить модель select с многопоточной моделью (ЛБ4): обсудить потребление памяти, накладные расходы на переключение контекста, ограничение FD_SETSIZE, применимость для тысяч соединений.

8. Протестировать сервер с несколькими telnet-клиентами, захватить трафик, показать одновременную работу.

2. Теоретические сведения

Системный вызов select() позволяет одному потоку отслеживать состояние нескольких файловых дескрипторов (сокетов) в ожидании событий: доступность для чтения, доступность для записи или возникновение исключительной ситуации. Прототип:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- nfds – наибольший номер дескриптора во всех наборах плюс 1.
- Наборы fd_set модифицируются: на входе содержат дескрипторы, которые нужно отслеживать; на выходе – те, на которых произошли события.
- Макросы для работы: FD_ZERO, FD_SET, FD_CLR, FD_ISSET.
- Возвращает количество готовых дескрипторов, 0 при таймауте, -1 при ошибке.

Для корректной работы с select клиентские сокеты **обязательно** должны быть установлены в неблокирующий режим. Иначе возможна ситуация, когда select сообщил о готовности сокета к чтению, но последующий recv может заблокироваться (если данные были прочитаны конкурирующим потоком или в редких случаях). Неблокирующий режим задаётся флагом O_NONBLOCK через fcntl:

```
int flags = fcntl(sock, F_GETFL, 0);
fcntl(sock, F_SETFL, flags | O_NONBLOCK);
```

В Python эквивалент — sock.setblocking(False), но при использовании select.select() можно оставить блокирующий режим, однако лучше перевести в неблокирующий для надёжности.

Ограничения select()**:**

- Максимальное количество дескрипторов ограничено константой FD_SETSIZE (обычно 1024).
- Наборы передаются в ядро и обратно при каждом вызове, что создаёт накладные расходы при тысячах соединений.
- Альтернативы: poll, epoll (Linux), kqueue (BSD/macOS), которые рассматриваются в ЛБ16.

Архитектура однопоточного чат-сервера:

- Сервер хранит массив структур для каждого клиента, включающий дескриптор и буфер приёма.

- В главном цикле пересобираются наборы fd_set для чтения: слушающий сокет + все клиентские сокеты.
- Вызывается select.
- Если слушающий сокет готов — accept нового клиента, добавление в массив.
- Для каждого клиентского сокета, который готов к чтению:
    - recv в кольцевой буфер (или просто в динамический буфер).
    - Если recv вернул 0 или ошибку — закрыть сокет, удалить клиента.
    - Если получены данные — накопить их, извлечь все завершённые строки (по \n).
    - Каждую полную строку ретранслировать другим клиентам (например, с помощью send).
- Ретрансляция может осуществляться немедленно, но при неблокирующих сокетах нужно быть готовым к тому, что send не сможет отправить всё сразу — однако для чата с небольшими сообщениями и локальной сетью это маловероятно; можно использовать send без дополнительных проверок или добавлять в набор для записи при необходимости.

Сравнение с многопоточной моделью:
Мультиплексирование позволяет обслуживать тысячи клиентов одним потоком, экономя память (каждый поток требует стек > 1 МБ) и процессорное время на переключение контекстов. Однако программирование сложнее из-за ручного управления состояниями и буферами. Для высоконагруженных серверов (например, nginx) модель событий (epoll) является предпочтительной.

3. Задание на выполнение работы

1. **Реализовать однопоточный TCP-чат-сервер на основе** select **(или** select **из Python):**
    - Привязать слушающий сокет к порту (например, 9000).
    - Основной цикл:
        - Очистить и заполнить fd_set слушающим сокетом и всеми активными клиентскими сокетами.
        - Вызвать select с таймаутом (например, 1 секунда) для возможности обработки прерываний.
        - Если select вернул > 0:
            - Проверить слушающий сокет FD_ISSET → принять новое соединение, перевести клиентский сокет в неблокирующий

режим, добавить в список клиентов (структура с сокетом и динамическим буфером).

- Для каждого клиентского сокета из списка: если FD_ISSET, то выполнить чтение данных и обработку сообщений (см. ниже).

- Если select вернул -1 и errno == EINTR — продолжить (обработка сигнала).

- Обработка чтения от клиента:
  - Выделить/использовать буфер для данного клиента.
  - Вызвать recv в неблокирующем режиме:
    - Если получено 0 — клиент отключился, закрыть сокет, удалить из списка.
    - Если ошибка и errno == EAGAIN/EWOULDBLOCK — данных пока нет (но select сказал обратное, это может быть ложным срабатыванием, продолжаем).
    - Если получены данные: добавить их в конец буфера клиента.
  - Проверить, есть ли в буфере символ \n. Если есть — извлечь полные строки (до \n включительно), для каждой строки выполнить ретрансляцию: отправить эту строку (или её содержимое без \n) всем остальным клиентам (с добавлением идентификатора, например, Client N: message).
  - После ретрансляции сдвинуть оставшиеся данные в начало буфера.
- При ретрансляции, если send клиенту возвращает ошибку (или отправлено меньше), можно временно накапливать данные для отправки; в простом варианте можно считать, что send всегда удаётся (для локального тестирования допустимо).

2. **Клиент для тестирования:**
   - Использовать telnet (telnet localhost 9000) либо написать простейшего клиента аналогично ЛБ2, поддерживающего отправку строк и вывод полученных.
3. **Тестирование:**
   - Запустить сервер.
   - Открыть 3–4 сессии telnet. Убедиться, что сообщения от одного клиента видны всем остальным, но не возвращаются отправителю.
   - Подключить и отключить несколько клиентов; сервер должен корректно управлять списком.

- Захватить трафик в Wireshark, проанализировать, как множество соединений обслуживаются одним потоком (чередование пакетов).
- Сравнить потребление памяти/потоков с многопоточным сервером из ЛБ4 (например, запустить оба и посмотреть в диспетчере задач количество потоков).

4. **Дополнительно:**
   - Реализовать таймаут неактивности: если клиент не отправлял данные более 60 секунд, сервер отключает его.
   - Продемонстрировать обработку до 1024 сокетов (можно симулировать, создав много клиентов через скрипт, но обязательно с учётом FD_SETSIZE).

4. Методические указания по выполнению

- **Подготовка** fd_set**:**
  В C необходимо на каждой итерации пересоздавать набор для чтения, заполняя его заново всеми активными дескрипторами. Используйте массив для хранения клиентов и перебирайте его для FD_SET.
- **Пример главного цикла (C):**

```c
fd_set read_fds;
int max_fd;
while (1) {
    FD_ZERO(&read_fds);
    FD_SET(server_fd, &read_fds);
    max_fd = server_fd;
    for (int i = 0; i < num_clients; i++) {
        int fd = clients[i].fd;
        FD_SET(fd, &read_fds);
        if (fd > max_fd) max_fd = fd;
    }
    int activity = select(max_fd + 1, &read_fds, NULL, NULL, &timeout);
    if (activity < 0 && errno != EINTR) { perror("select"); break; }
    if (FD_ISSET(server_fd, &read_fds)) {
        // accept новый клиент
    }
    for (int i = 0; i < num_clients; i++) {
        if (FD_ISSET(clients[i].fd, &read_fds)) {
            // чтение и обработка
        }
    }
}
```

- **Неблокирующий режим для клиентского сокета:**
  Сразу после accept:

```c
int flags = fcntl(new_fd, F_GETFL, 0);
fcntl(new_fd, F_SETFL, flags | O_NONBLOCK);
```

- **Буферизация строк:**
  Для каждого клиента поддерживайте динамический буфер (например, char *buffer и int buf_len). При чтении добавляйте данные в конец буфера, ищите \n. В C удобно использовать realloc. В Python можно использовать bytearray и искать b'\n'.
- **Пример извлечения строк и ретрансляции (C):**

```c
char *newline = memchr(client->buffer, '\n', client->buf_len);
while (newline) {
    int line_len = newline - client->buffer + 1; // включая \n
    // отправить всем остальным (предварительно можно убрать \n или добавить тег)
    for (int j = 0; j < num_clients; j++) {
        if (clients[j].fd != client->fd) {
            send(clients[j].fd, client->buffer, line_len, 0);
        }
    }
    // сдвинуть буфер
    memmove(client->buffer, client->buffer + line_len, client->buf_len - line_len);
    client->buf_len -= line_len;
    newline = memchr(client->buffer, '\n', client->buf_len);
}
```

При этом нужно быть готовым, что send может отправить не все байты; для простой лабораторной допустимо проигнорировать частичную отправку (в локальной сети она маловероятна). Для более строгого подхода можно вести очередь для отправки, но это усложнение.

- **Python с** selectors **или напрямую** select.select**:**
  Рекомендуется использовать selectors.DefaultSelector для кроссплатформенности, но для учебных целей можно использовать select.select. Не забудьте установить неблокирующий режим для клиентских сокетов (sock.setblocking(False)).
  Пример с select.select:

```python
import select, socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind(('', 9000))
server.listen(5)
server.setblocking(False)  # не обязательно, но рекомендуется
inputs = [server]
clients = {}  # fd -> (socket, address, buffer)
while True:
    readable, _, _ = select.select(inputs, [], [], 1.0)
    for s in readable:
        if s is server:
            conn, addr = s.accept()
            conn.setblocking(False)
            inputs.append(conn)
            clients[conn.fileno()] = (conn, addr, b"")
        else:
            try:
                data = s.recv(1024)
                if data:
                    # добавление в буфер и извлечение строк
                else:
                    # отключение
            except BlockingIOError:
                pass
```

- **Ограничение** FD_SETSIZE**:**
  В Linux по умолчанию 1024. Можно пересобрать ядро или использовать poll/epoll, но в данной работе нужно продемонстрировать понимание этого ограничения. При тестировании до 1024 дескрипторов можно упереться в лимит открытых файлов процесса (ulimit -n), что тоже важно.

5. Содержание отчёта

1. **Цель работы** и описание реализованного сервера (однопоточный, мультиплексирование).
2. **Листинг сервера** с подробными комментариями к блоку select и буферизации.
3. **Инструкция по сборке и запуску.**
4. **Результаты тестирования:**

- Скриншоты консолей трёх клиентов (telnet), показывающие, что сообщения ретранслируются всем, кроме отправителя.
- Скриншот консоли сервера с сообщениями о подключении/отключении (если реализовано).
- Wireshark-дамп с несколькими TCP-потоками; показ чередования пакетов от разных клиентов.
- Сравнение потребления ресурсов (потоки/память) с многопоточным сервером из ЛБ4 для 100 подключений (таблица).

5. **Анализ:** преимущества и недостатки select по сравнению с потоками, ограничения FD_SETSIZE, буферизация строк.
6. **Выводы.**

6. Контрольные вопросы

1. Что делает системный вызов select? Опишите его параметры. Какой параметр указывает верхнюю границу дескрипторов?
2. Почему для работы с select клиентские сокеты рекомендуется переводить в неблокирующий режим?
3. Как сервер обрабатывает ситуацию, когда клиент отправляет несколько сообщений подряд, и они склеиваются в один TCP-сегмент?
4. Каким образом сервер узнаёт, что клиент закрыл соединение?
5. В чём заключается ограничение FD_SETSIZE? Что произойдёт, если попытаться отслеживать больше дескрипторов?
6. Сравните модели «один поток на клиент» и «один поток + select» по потреблению памяти и поведению при 1000 ожидающих соединений.
7. Какие альтернативы select существуют в Linux и Windows? Чем epoll лучше select?
8. Почему в вашем сервере не требуется мьютекс для защиты общих данных?

7. Критерии оценки

Оценка	Критерии
Зачтено	Сервер работает в одном потоке, использует select() (или select.select в Python) для одновременной обработки нескольких клиентов. Клиентские сокеты переведены в неблокирующий режим. Сервер корректно принимает новые

Оценка	Критерии
	соединения, ретранслирует сообщения другим клиентам (чат), удаляет отключившихся. Демонстрируется работа как минимум с тремя клиентами одновременно. Код содержит обработку ошибок и корректную буферизацию для извлечения строк. Студент на защите объясняет устройство главного цикла, назначение FD_ZERO/FD_SET и т.д., может модифицировать код (например, изменить таймаут). В отчёте приведён сравнительный анализ с многопоточной моделью, есть скриншоты и Wireshark-дамп.
Не зачтено	Программа не использует select (например, применяет потоки или процессы). Сервер не может обслуживать более одного клиента одновременно, либо при подключении второго клиента первый перестаёт получать сообщения. Не настроен неблокирующий режим, из-за чего сервер зависает на операциях recv. Отсутствует буферизация — строки теряются или разрезаются. Студент не понимает работу select, путает наборы дескрипторов, не может объяснить, как сервер определяет готовность сокета. Отчёт не соответствует требованиям.

Лабораторная работа №10

Тема: Простой HTTP-сервер для отдачи статических файлов

Цель работы:
Изучить протокол передачи гипертекста HTTP/1.0 и HTTP/1.1 на примере реализации сервера, способного принимать и разбирать GET-запросы, отдавать статические файлы из файловой системы и формировать корректные HTTP-ответы с соблюдением требований безопасности. На практике закрепить навыки работы с TCP-сокетами и мультиплексирования ввода-вывода.

Формируемые компетенции (углубление):

- ПК-4: реализация сервера прикладного протокола HTTP, обработка ошибок, обеспечение безопасного доступа к файловой системе.
- ПК-9: документирование поддерживаемых методов и формата ответов сервера.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Реализовать TCP-сервер, который прослушивает порт (например, 8080) и принимает входящие соединения.
2. Для каждого соединения принять HTTP-запрос, разобрать (parse) начальную строку: метод, URI, версию протокола.
3. Обрабатывать только метод GET; для всех остальных методов возвращать ответ 501 Not Implemented.
4. Транслировать URI запроса в путь к файлу внутри заданной корневой директории (docroot), обеспечивая защиту от выхода за пределы корневого каталога (path traversal).
5. Проверить существование файла:
    - если файла нет — вернуть 404 Not Found;
    - если файл существует, но нет прав на чтение — вернуть 403 Forbidden;
    - в противном случае — отдать содержимое файла с кодом 200 OK.
6. Формировать обязательные HTTP-заголовки ответа: Server (опционально), Content-Type (определять по расширению файла), Content-Length, Connection (close для HTTP/1.0 или по умолчанию keep-alive для HTTP/1.1, но в учебном варианте можно всегда close).

7. (Рекомендуется) Использовать мультиплексирование (select/poll) или многопоточность для одновременного обслуживания нескольких клиентов; минимально допустимо последовательное обслуживание (один клиент за раз) с явным указанием этого ограничения в отчёте.
8. Продемонстрировать работу сервера с реальным веб-браузером и утилитой curl, захватить трафик в Wireshark и изучить структуру HTTP-сообщений.

2. Теоретические сведения

HTTP (HyperText Transfer Protocol) – основной протокол обмена данными в Web. В данной работе рассматривается HTTP/1.0 (RFC 1945) и упрощённо HTTP/1.1 (RFC 2616). Каждое взаимодействие состоит из запроса клиента и ответа сервера, передаваемых в текстовом виде поверх TCP.

Формат HTTP-запроса (упрощённо):

```
GET /index.html HTTP/1.1\r\n
Host: localhost:8080\r\n
User-Agent: Mozilla/5.0\r\n
Accept: text/html\r\n
\r\n
```

Начальная строка: Метод URI Версия. Методы: GET, POST, HEAD и др. В работе реализуется **только GET**. URI может содержать параметры (после ?), но для статического сервера они игнорируются. Версия определяет возможности соединения.

Формат HTTP-ответа:

```
HTTP/1.1 200 OK\r\n
Server: MyHttpServer/1.0\r\n
Content-Type: text/html\r\n
Content-Length: 1234\r\n
Connection: close\r\n
\r\n
<html>...</html>
```

Обязательны: статусная строка (HTTP/версия Код Пояснение), заголовки, пустая строка, тело (если есть).

Коды состояния (статус-коды):

- 200 OK – успешный запрос.
- 400 Bad Request – синтаксическая ошибка в запросе.
- 403 Forbidden – доступ запрещён.

- 404 Not Found – ресурс не найден.
- 501 Not Implemented – метод не поддерживается.
- 505 HTTP Version Not Supported – версия не поддерживается (опционально).

Заголовки:

- Content-Type – MIME-тип содержимого
  (например, text/html для .html, image/jpeg для .jpg, text/plain для .txt, application/octet-stream для прочих).
- Content-Length – размер тела в байтах.
- Connection – управление соединением (close – закрыть после ответа, keep-alive – сохранить для последующих запросов). В HTTP/1.0 по умолчанию close, в HTTP/1.1 – keep-alive.

Безопасность пути:
Никогда нельзя напрямую конкатенировать корневую директорию и URI. Пример:
URI /../etc/passwd может дать доступ к системному файлу. Нужно нормализовать путь:

1. Отбросить GET-параметры (?).
2. Декодировать URL-encoded символы (%20).
3. Преобразовать к каноническому виду с удалением ...
4. Проверить, что итоговый абсолютный путь начинается с корневой директории сервера.
   В учебном проекте достаточно использовать realpath() (POSIX) или аналоги и сравнить префикс.

Мультиплексирование:
Применение select() (из ЛР №9) предпочтительно для обслуживания десятков клиентов без создания потоков. Однако в данной работе можно начать с итеративной обработки, а затем добавить select как развитие.

3. Задание на выполнение работы

1. **Разработать HTTP-сервер** со следующими характеристиками:
   - Запускается с указанием порта и корневой директории (аргументы командной строки). По умолчанию порт 8080, директория ./www.
   - Создаёт TCP-сокет, ожидает подключения.
   - При подключении читает данные от клиента до обнаружения пустой строки \r\n\r\n (окончание заголовков). Время чтения ограничить таймаутом (например, 5 секунд) во избежание зависаний.

- Разбирает начальную строку. Если она не соответствует формату МЕТОД URI ВЕРСИЯ – возвращает 400 Bad Request.
- Поддерживает только GET. Для остальных методов – 501 Not Implemented.
- Преобразует URI в локальный путь к файлу:
  - Удалить ведущий /.
  - Заменить / на разделитель ОС (если нужно).
  - Выполнить нормализацию и проверку на выход за пределы корневой директории.
  - Если путь оканчивается на /, дополнить элементом index.html (обработка индекса по умолчанию).
- Пытается открыть полученный файл.
  - Если файла нет – 404.
  - Если нет прав на чтение – 403.
  - Иначе – 200, вычислить размер файла, определить Content-Type по расширению, отправить заголовки и содержимое файла.
- После ответа закрыть соединение (заголовок Connection: close), если только не реализован keep-alive (по желанию).
- Обрабатывать сигнал SIGINT для корректного завершения.

2. **Требования к обработке:**
   - Заголовок Content-Length обязателен.
   - Content-Type должен назначаться по расширению (минимум html, txt, jpg, png, css, js). Для неизвестных – application/octet-stream.
   - В ответе сервер должен включать собственную строчку Server: StudentHTTPServer/1.0.
3. **Тестирование:**
   - Создать директорию www с файлами: index.html, test.txt, image.jpg, поддиректорию с файлами.
   - Запустить сервер.
   - Открыть в браузере http://localhost:8080/ – должна отобразиться index.html.
   - Проверить http://localhost:8080/test.txt – отображается текст.
   - Проверить http://localhost:8080/nonexistent.html – страница 404.
   - С помощью curl -v зафиксировать полный ответ сервера, включая заголовки.

- Попытаться обратиться к /../etc/passwd – убедиться, что доступ запрещён (скорее всего, 403 или 404 после нормализации).
- Запустить сервер с поддержкой параллельных клиентов (select или потоки), открыть два браузерных сеанса одновременно и убедиться, что они обслуживаются.
- Захватить трафик Wireshark и показать HTTP-запрос и ответ, декодирование заголовков.

4. Методические указания по выполнению

- **Парсинг запроса:**
  В C: читайте в буфер, ищите \r\n\r\n. В Python: используйте socket.makefile() для построчного чтения заголовков, или читайте всё и разбивайте. Пример на Python:

```
request = b""
while b"\r\n\r\n" not in request:
    chunk = client.recv(4096)
    if not chunk: break
    request += chunk
# разделить строки
```

Начальная строка: request.splitlines()[0]. Декодировать UTF-8.

- **Нормализация пути:**
  Python: os.path.realpath(os.path.join(root_dir, uri.lstrip('/'))), затем проверить, что результат начинается с root_dir + os.sep. В C: можно использовать realpath(), но следить за возможными отсутствующими файлами — для проверки на .. можно самому пройтись по компонентам пути.
- **Определение Content-Type:**
  Простейший способ – словарь расширений:

```
MIME = {
    '.html': 'text/html',
    '.txt': 'text/plain',
    '.jpg': 'image/jpeg',
    '.png': 'image/png',
    '.css': 'text/css',
    '.js': 'application/javascript'
}
```

Получить расширение os.path.splitext. Если нет – application/octet-stream.

- **Формирование ответа:**
  Собираем строку статуса, заголовки, пустую строку. В Python:

```
response_headers = f"HTTP/1.0 200 OK\r\n"
response_headers += f"Server: StudentHTTPServer/1.0\r\n"
response_headers += f"Content-Type: {content_type}\r\n"
response_headers += f"Content-Length: {file_size}\r\n"
response_headers += f"Connection: close\r\n\r\n"
client.sendall(response_headers.encode())
# затем отправить тело файла
with open(full_path, 'rb') as f:
    while chunk := f.read(8192):
        client.sendall(chunk)
```

- **Поддержка параллельных клиентов:**
  - Через select: взять за основу сервер ЛБ9, заменить обработку эхо на HTTP-обработчик. Для каждого клиента нужно накапливать запрос до нахождения \r\n\r\n, затем обработать и сразу отправить ответ, после чего либо закрыть, либо если keep-alive – сбросить буфер и продолжить. Продвинутый вариант: после отправки ответа закрывать сокет (в HTTP/1.0). С select такой короткоживущий обработчик также укладывается в модель.
  - Через потоки: допустимо, но обязательно сравнить с select в выводах.
- **Обработка ошибок:**
  - Таймаут на чтение (чтобы клиент не держал соединение бесконечно).
  - Проверка на слишком длинный URI (защита от переполнения буфера).
  - Возврат 400 Bad Request, если запрос не соответствует формату.
  - Возврат 414 Request-URI Too Long (опционально).

5. Содержание отчёта

1. **Цель работы** и краткое описание реализованного HTTP-сервера.
2. **Листинг сервера** (основные функции: разбор запроса, формирование ответа, безопасная работа с путём).
3. **Инструкция по сборке и запуску** (аргументы командной строки).
4. **Результаты тестирования:**
   - Скриншоты браузера с успешной загрузкой HTML-страницы, картинки, текстового файла.

- Вывод curl -v для каждого сценария (200, 404, 403, проверка заголовков).
- Скриншот Wireshark с раскрытыми полями HTTP-запроса и ответа, фильтр tcp.port == 8080.
- Демонстрация параллельной работы: два терминала с curl или два окна браузера, скриншот с временем выполнения.

5. **Анализ:** обсуждение особенностей HTTP/1.0 и 1.1, возможные улучшения (keep-alive, кэширование, поддержка HEAD), безопасность (защита от path traversal).
6. **Выводы** (чему научились, сложности, сравнение с предыдущими протоколами).

6. Контрольные вопросы

1. Опишите структуру HTTP-запроса и HTTP-ответа. Какие обязательные элементы они содержат?
2. Какие методы HTTP вы знаете? Для чего они используются? Почему ваш сервер поддерживает только GET?
3. Какие коды состояния HTTP реализованы в вашем сервере? Поясните смысл каждого.
4. Как сервер определяет MIME-тип файла? Какие ещё способы существуют (например, файл /etc/mime.types, заголовок X-Content-Type-Options)?
5. В чём разница между HTTP/1.0 и HTTP/1.1 в отношении управления соединением? Как ваше приложение обрабатывает заголовок Connection?
6. Каким образом вы предотвращаете атаку «выход за пределы корневого каталога» (Path Traversal)? Приведите пример кода.
7. Почему важно ограничивать размер загружаемого запроса? Какую уязвимость это предотвращает?
8. Как вы тестировали параллельную работу сервера? Какую модель (select/потоки) вы выбрали и почему?
9. Для чего нужен заголовок Content-Length? Что произойдёт, если его не отправить?

7. Критерии оценки (конкретизированные для ЛР №10)

Оценка	Критерии
Зачтено	Сервер успешно принимает GET-запросы, корректно отдаёт статические файлы из указанной директории, формирует правильные HTTP-заголовки

Оценка	Критерии
	(Content-Type, Content-Length, Connection). Реализована обработка ошибок: 400, 403, 404, 501 как минимум. Обеспечена защита от Path Traversal (невозможно прочитать файлы вне разрешённой директории). Сервер способен обслуживать нескольких клиентов одновременно (продемонстрировано: два браузера или два curl одновременных запроса не задерживают друг друга). Код структурирован и снабжён комментариями. Отчёт содержит все требуемые скриншоты и анализ. Студент на защите объясняет устройство парсера, порядок формирования ответа, может модифицировать директиву ответа (например, добавить новый MIME-тип).
Не зачтено	Программа не запускается, или при запросе отдаёт неверный контент (например, HTML-страницу сохраняет в текстовом формате). Отсутствуют обязательные заголовки (Content-Length или Content-Type). Не реализована проверка на выход за пределы директории (демонстрируется чтение /etc/passwd). Сервер «падает» при любом некорректном запросе или при одновременном подключении нескольких клиентов. Студент не может объяснить разбор строки запроса и назначение кодов состояния.

Лабораторная работа №11

Тема: HTTP-клиент с поддержкой методов GET и POST

Цель работы:
Детально разобрать формат HTTP-запросов и ответов, научиться программно формировать GET- и POST-запросы, обрабатывать коды состояния, заголовки ответов, а также реализовать базовую обработку перенаправлений и механизма передачи данных chunked transfer encoding. Полученные умения закрепят понимание протокола HTTP и подготовят к взаимодействию с веб-сервисами.

Формируемые компетенции (углубление):

- ПК-4: реализация клиента прикладного протокола, обработка нестандартных ситуаций в HTTP-диалоге.
- ПК-9: документирование поддерживаемых возможностей клиента.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Разобрать URL, переданный в командной строке, выделив схему (http), хост, порт, путь и (для GET) параметры запроса.
2. Установить TCP-соединение с хостом на нужном порту (80 по умолчанию).
3. Сформировать корректный HTTP-запрос методом GET или POST:
    - для GET: параметры передаются в строке запроса (URL-кодированные);
    - для POST: данные передаются в теле запроса в формате application/x-www-form-urlencoded, с обязательным заголовком Content-Length.
4. Отправить запрос, принять ответ, разобрать статусную строку (код состояния и пояснение) и заголовки.
5. В зависимости от заголовков ответа:
    - если Content-Length присутствует, прочитать ровно указанное количество байт тела;
    - если Transfer-Encoding: chunked, прочитать тело в виде последовательности чанков (размер в шестнадцатеричном виде, данные, \r\n);
    - если код состояния 301, 302, 307, 308 и присутствует заголовок Location, выполнить переход по новому адресу (с ограничением количества редиректов);
    - для кодов 4xx/5xx вывести диагностическое сообщение.

6. Вывести полученное тело ответа на консоль или сохранить в файл (по выбору).
7. Протестировать клиент на внешних тестовых серверах (например, httpbin.org) и на собственном HTTP-сервере (из ЛР №10), захватить трафик и проанализировать.

2. Теоретические сведения

HTTP-запрос состоит из стартовой строки, заголовков, пустой строки и опционального тела.

Стартовая строка для GET:
GET /path?param1=val1&param2=val2 HTTP/1.1\r\n
Для POST:
POST /path HTTP/1.1\r\n
Обязательный заголовок Host (в HTTP/1.1 обязателен), для POST – Content-Type: application/x-www-form-urlencoded и Content-Length: длина_тела.

Тело POST-запроса содержит данные в формате ключ=значение, пары разделены &, пробелы заменены на +, специальные символы закодированы через %XX (URL encoding).

HTTP-ответ: статусная строка (HTTP/1.1 200 OK), заголовки, пустая строка, тело.

Обработка тела ответа:

- Если есть заголовок Content-Length, тело читается фиксированной длины.
- Если Transfer-Encoding: chunked, тело разбито на части, каждая предваряется строкой с размером чанка в шестнадцатерике и завершается \r\n. Последний чанк имеет размер 0. Подробнее:

```
chunk_size\r\n
data (chunk_size bytes)\r\n
...
0\r\n
\r\n
```

После последнего чанка могут идти трейлеры (опциональные заголовки), но в рамках учебной работы допустимо просто пропустить их или читать до \r\n\r\n после последнего чанка.

Перенаправления (Redirection):
Если сервер возвращает статус 301, 302, 307, 308 и заголовок Location: новый_URL, клиент должен повторить запрос по новому адресу. Важно ограничить количество редиректов (например, 5), чтобы избежать бесконечных циклов.

URL-кодирование:
Клиент должен уметь кодировать имена и значения параметров для передачи в GET-строке или POST-теле. Простейший способ – заменять не-ASCII и специальные символы

на %XX, пробелы на +. В стандартных библиотеках есть готовые функции (Python: urllib.parse.urlencode; C: можно реализовать упрощённо или использовать libcurl, но в рамках лабораторной работы предпочтительнее ручная реализация для обучения).

Заголовки, обязательные для обработки:
Content-Length, Transfer-Encoding, Location, Connection (close/keep-alive). Клиент может поддерживать только Connection: close (завершать соединение после чтения всего ответа) или реализовать keep-alive (переиспользование сокета), но в учебной работе достаточно закрывать соединение после каждого запроса.

3. Задание на выполнение работы

1. **Разработать консольный HTTP-клиент**, принимающий аргументы:
    - метод (GET или POST);
    - URL (например, http://example.com/resource);
    - для POST – дополнительно строку с параметрами в формате key1=value1&key2=value2 (можно через stdin или аргумент --data).
    - опционально – флаг сохранения тела в файл.
2. **Алгоритм работы клиента:**
    - Разобрать URL (схема, хост, порт, путь, query). Если схема не http, завершить с ошибкой.
    - Для GET: параметры (если есть) добавить к пути как ?кодированная_строка.
    - Создать TCP-сокет, подключиться к хосту:порт.
    - Составить запрос:

```
<МЕТОД> <путь> HTTP/1.1\r\n
Host: <хост>\r\n
Connection: close\r\n
[Content-Type: application/x-www-form-urlencoded\r\n для POST]
[Content-Length: длина тела\r\n]
\r\n
[тело для POST]
```

    - Отправить запрос (sendall).
    - Принять ответ: читать данные, пока не встретится \r\n\r\n (конец заголовков). Разобрать статусную строку и заголовки (например, сохранить в словарь).
    - Проанализировать статус:

        - 200 OK – продолжить чтение тела.
        - 301/302/307/308 – если есть заголовок Location, и количество редиректов не превышено, повторить запрос по новому URL с тем же методом (но для 302 лучше перейти на GET – по стандарту, но в учебных целях можно упростить: использовать тот же метод). Для нового URL повторить шаги начиная с разбора URL.
        - 4xx/5xx – вывести сообщение об ошибке и статус, тело (если есть) всё равно прочитать и вывести.
    - Чтение тела:
        - Если есть Transfer-Encoding: chunked, перейти в режим чтения чанков.
        - Иначе, если есть Content-Length, прочитать ровно указанное число байт.
        - Если ни того, ни другого – читать до закрытия соединения (для ответов без тела).
    - Вывести тело в консоль (или сохранить в файл, если указан флаг -o).
3. **Реализовать поддержку chunked encoding:**
    - Для каждого чанка:
        - Прочитать строку до \r\n, преобразовать в целое число (шестнадцатеричное) – размер чанка.
        - Прочитать ровно столько байт данных (+ завершающие \r\n).
        - Повторять, пока размер чанка не станет 0.
        - После последнего чанка прочитать трейлеры (строки до \r\n\r\n, их можно проигнорировать).
    - Данные чанков склеить и представить как тело.
4. **Тестирование:**
    - Использовать тестовые сервера: httpbin.org/get, httpbin.org/post, httpbin.org/status/404, httpbin.org/redirect/2 (2 редиректа).
    - Проверить GET-запрос, разбор ответа, вывод заголовков и тела.
    - Проверить POST-запрос с параметрами, убедиться, что сервер (httpbin) возвращает отправленные параметры в JSON-ответе.
    - Проверить перенаправление: клиент должен автоматически перейти по цепочке и получить конечный 200 OK.

- Проверить 404: клиент должен вывести статус 404 и содержимое страницы ошибки.
- Захватить трафик Wireshark и проанализировать HTTP-диалог (запрос, ответ, возможные редиректы).

4. Методические указания по выполнению

- **Разбор URL (Python, C):**
  В Python можно использовать urllib.parse.urlparse. В C можно реализовать вручную: найти ://, выделить хост (до первого :, если порт) или до первого /. Порт по умолчанию 80. Путь – всё начиная с /.
- **Формирование запроса (Python):**

```
request = f"GET {path} HTTP/1.1\r\nHost: {host}\r\nConnection: close\r\n\r\n"
sock.sendall(request.encode())
```

- **Чтение заголовков ответа:**
  Читать из сокета, пока не встретится последовательность b"\r\n\r\n". Затем разделить на строки. Первая строка – статус, остальные – заголовки. В Python можно создать словарь заголовков, сохраняя их имена в нижний регистр.
- **Чтение тела по Content-Length:**
  Использовать функцию recv_exact из предыдущих работ, чтобы прочитать ровно N байт.
- **Чтение chunked тела (псевдокод на Python):**

```
body = b''
while True:
    # читать строку размера чанка до \r\n
    size_line = b''
    while not size_line.endswith(b'\r\n'):
        size_line += sock.recv(1)
    size = int(size_line.strip(), 16)
    if size == 0:
        # последний чанк, прочитать завершающие \r\n и трейлеры (если есть)
        # трейлеры до пустой строки
        break
    # прочитать size байт данных + \r\n
    chunk_data = recv_exact(sock, size)
    sock.recv(2)  # прочитать \r\n
    body += chunk_data
```

Обратите внимание на необходимость обработки ситуации, когда сервер использует расширения чанков (например, ;комментарий после размера), что в упрощённой реализации можно проигнорировать.

- **Обработка редиректов:**
  После получения кода 301/302/307/308 извлечь заголовок Location. Если URL относительный, преобразовать в абсолютный относительно исходного запроса. Увеличить счётчик редиректов. Если счётчик превысил лимит (5), выдать ошибку и прекратить. Затем повторить запрос по новому URL (возможно, с GET, если был POST и код 302 – по стандарту меняется, но для простоты можно всегда использовать GET при 302, но это не обязательно). В учебных целях можно повторять тот же метод, но нужно указать это в отчёте.
- **URL-кодирование:**
  Для GET-параметров и POST-тела нужно закодировать спецсимволы. В Python используйте urllib.parse.urlencode или напишите простой аналог: re.sub(r'[^a-zA-Z0-9_]', lambda m: '%%%02X' % ord(m.group(0)), s) и заменить пробелы на +.
- **Тестирование на httpbin:**
  GET: http://httpbin.org/get?foo=bar – проверить, что ответ содержит JSON с полем args.foo == "bar".
  POST: http://httpbin.org/post с данными key1=value1&key2=value2 – тело ответа должно содержать form с переданными парами.
  Редирект: http://httpbin.org/redirect/2 – клиент должен пройти 2 редиректа и в итоге получить ответ от /get.
  404: http://httpbin.org/status/404 – получить код 404.

5. Содержание отчёта

1. **Цель работы** и описание функциональности HTTP-клиента.
2. **Листинг программы** с комментариями (разбор URL, формирование запроса, чтение ответа, обработка чанков и редиректов).
3. **Инструкция по сборке и запуску** (аргументы командной строки).
4. **Результаты тестирования:**
   - Вывод клиента для GET-запроса к httpbin.org/get (со скриншотом консоли).
   - Вывод клиента для POST-запроса с параметрами, подтверждение, что параметры доставлены.
   - Скриншот или логи, демонстрирующие обработку редиректа (например, вывод промежуточных статусов).
   - Скриншот обработки ошибки 404.

- Wireshark-дамп с разбором HTTP-диалога, выделение запросов/ответов.
- (Опционально) Пример работы с chunked encoding (можно получить с httpbin, добавив ?chunked=1 или через свой сервер).

5. **Анализ:** обсуждение сложностей, связанных с разбором чанков, редиректами, отличия Content-Length и chunked, безопасность редиректов.
6. **Выводы.**

6. Контрольные вопросы

1. Опишите структуру HTTP-запроса методами GET и POST. В чём отличие размещения параметров?
2. Какие заголовки ответа отвечают за указание длины тела? Объясните принцип работы Transfer-Encoding: chunked.
3. Как ваш клиент обрабатывает ситуацию, когда ответ не содержит ни Content-Length, ни Transfer-Encoding?
4. Что такое URL-кодирование и зачем оно нужно? Приведите пример кодирования строки «hello world».
5. Какие коды состояния относятся к перенаправлениям? Как клиент должен себя вести при получении 301 и 302? Зачем нужно ограничивать количество редиректов?
6. Какие проблемы безопасности могут возникнуть при автоматическом следовании редиректам?
7. Чем HTTP/1.1 отличается от HTTP/1.0 при работе с соединениями? Как заголовок Connection влияет на поведение клиента?
8. Какие утилиты можно использовать для отладки HTTP-клиента помимо Wireshark? (например, netcat, curl -v)
9. Как ваш клиент поступает, если сервер закрывает соединение до того, как тело полностью передано?

7. Критерии оценки (конкретизированные для ЛР №11)

Оценка	Критерии
Зачтено	Клиент реализует оба метода (GET и POST), корректно формирует запросы и разбирает ответы (статус, заголовки). Поддерживается обработка кодов 200,

Оценка	Критерии
	404, а также автоматический переход по редиректам (301/302/307/308) с ограничением их количества. Реализована базовая поддержка chunked transfer encoding (клиент успешно скачивает тело, переданное чанками). Тело ответа корректно извлекается и отображается. Программа обрабатывает ошибки сети и HTTP-ошибки без аварийного завершения. Код снабжён комментариями. Отчёт содержит скриншоты всех сценариев, включая Wireshark-дамп. Студент на защите объясняет механизм чанков, обработку редиректов, формирование POST-запроса, может модифицировать код (например, добавить поддержку заголовка User-Agent).
Не зачтено	Программа не может выполнить простой GET-запрос с разбором ответа. Отсутствует формирование POST-запроса. Не обрабатываются коды ошибок (программа падает при 404). Редиректы игнорируются или вызывают зацикливание. Chunked encoding не поддерживается (тело не загружается). Студент не понимает различия между GET и POST, не может объяснить назначение заголовков, не способен прочитать и объяснить свой код. Отчёт неполный или отсутствует.

Лабораторная работа №12

Тема: Нагрузочное тестирование сетевого сервера с помощью Apache Bench и JMeter

Цель работы:
Освоить методику нагрузочного тестирования сетевых приложений с использованием инструментов Apache Bench (ab) и Apache JMeter. Научиться измерять ключевые метрики производительности (RPS, время ответа, пропускная способность), выявлять узкие места в архитектуре сервера и сравнивать эффективность различных моделей обслуживания клиентов (итеративная, многопоточная, мультиплексированная). Закрепить навыки документирования результатов с построением графиков.

Формируемые компетенции (углубление):

- ПК-4: тестирование производительности, отладка, анализ и устранение «бутылочных горлышек» в сетевом ПО.
- ПК-9: оформление отчёта с метриками и визуализацией.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Установить и настроить Apache Bench (входит в пакет apache2-utils) и/или Apache JMeter на рабочей станции.
2. Подготовить тестовый стенд: запустить один из разработанных ранее TCP- или HTTP-серверов (из ЛБ №3, №4, №9, №10) с различными моделями обслуживания.
3. Сконфигурировать нагрузочный инструмент для отправки запросов к серверу с варьируемым уровнем параллелизма (concurrency) и общим количеством запросов.
4. Провести серию тестов для каждого типа сервера (однопоточный итеративный, многопоточный, основанный на select) при разных значениях параллельных клиентов (1, 10, 50, 100 и т.д.).
5. Зафиксировать полученные метрики: количество запросов в секунду (RPS), среднее/медианное время обработки запроса, пропускную способность (Transfer rate), процент ошибок.
6. Проанализировать зависимость метрик от числа конкурентных соединений, выявить пределы масштабируемости каждой архитектуры.
7. Построить графики (RPS от concurrency, время ответа от concurrency) и подготовить отчёт с выводами.

2. Теоретические сведения

Нагрузочное тестирование (load testing) – процесс подачи искусственной рабочей нагрузки на систему для оценки её производительности и стабильности. Основные цели: определить максимальную пропускную способность, измерить время отклика при разных нагрузках, выявить деградацию сервера и его «точку насыщения».

Ключевые метрики:

- **RPS (Requests Per Second)** – количество запросов, успешно обработанных за секунду. Характеризует пропускную способность сервера.
- **Time per request (mean)** – среднее время, затраченное на обработку одного запроса с учётом ожидания в очереди (если конкурентность > 1). Выражается в миллисекундах.
- **Transfer rate** – объём данных, передаваемых в единицу времени (КБ/с). Важно при оценке сетевого канала и эффективности использования буферов.
- **Failed requests** – доля запросов, завершившихся ошибками (таймауты, разрыв соединения, некорректные ответы).

Инструменты тестирования:

- **Apache Bench (ab)** – консольная утилита, входящая в дистрибутив Apache. Проста в использовании, но предназначена в основном для HTTP/HTTPS. Позволяет задать общее число запросов (-n) и уровень параллелизма (-c). Пример:
  ab -n 1000 -c 50 http://localhost:8080/index.html
- **Apache JMeter** – графический инструмент с широкими возможностями. Поддерживает не только HTTP, но и TCP-сокеты (TCP Sampler), UDP, FTP и др. Позволяет создавать сложные сценарии с переменными, таймерами и сбором статистики. Для тестирования TCP-эхо-сервера можно использовать «TCP Sampler», который устанавливает соединение и отправляет/принимает данные.

Планирование эксперимента:
При тестировании необходимо менять нагрузку (concurrency) и наблюдать, как ведёт себя RPS и время ответа. Для итеративного сервера (один процесс) конкурентные запросы будут вставать в очередь, время ответа резко возрастёт, а RPS останется на низком уровне. Многопоточный сервер способен обрабатывать запросы параллельно, но при очень большом числе потоков могут сказаться накладные расходы на переключение контекста и выделение памяти. Сервер на основе select способен эффективно распределять время между многими сокетами, но если обработка запроса занимает процессорное время, то один поток также станет узким местом. Сравнение этих моделей составляет основу лабораторной работы.

Рекомендации по подготовке тестов:

- Для чистоты эксперимента клиент и сервер желательно запускать на разных хостах или, как минимум, изолировать сетевое взаимодействие через loopback (127.0.0.1) с учётом того, что loopback обладает высокой пропускной способностью.
- Исключить влияние дисковых операций: для эхо-серверов это неактуально, для HTTP-сервера отдавать статику из памяти (или очень маленькие файлы), чтобы измерить именно сетевую/процессорную производительность.
- Убедиться, что система не ограничивает число открытых файловых дескрипторов (ulimit -n), особенно при большом числе параллельных соединений.

3. Задание на выполнение работы

1. **Подготовка тестового окружения:**
    - Установить Apache Bench: sudo apt install apache2-utils (Linux) или использовать встроенный в macOS. Для Windows можно установить через WSL или скачать бинарные файлы Apache.
    - Установить JMeter (потребуется Java): скачать с jmeter.apache.org, распаковать, запустить jmeter.bat / jmeter.sh.
    - Выбрать (или доработать) два сервера из числа предыдущих работ:
        - **Сервер А:** итеративный TCP-эхо-сервер (ЛБ №3) или однопоточный HTTP-сервер (ЛБ №10 без select).
        - **Сервер Б:** многопоточный TCP-эхо-сервер (ЛБ №4) или HTTP-сервер с многопоточностью / select-мультиплексированием (ЛБ №9, №10 с select).
    - Оба сервера должны реализовывать простой протокол, чтобы время обработки было минимальным (эхо, отдача фиксированного HTML).
2. **Проведение нагрузочного тестирования HTTP-сервера (если используется):**
    - Запустить сервер.
    - С помощью ab провести замеры:

```
ab -n 5000 -c 1 http://localhost:8080/index.html
ab -n 5000 -c 10 http://localhost:8080/index.html
ab -n 5000 -c 50 http://localhost:8080/index.html
ab -n 5000 -c 100 http://localhost:8080/index.html
```

(при необходимости подбирать число запросов, чтобы тест длился не менее 10 секунд)

    - Зафиксировать метрики из вывода утилиты: Requests per second, Time per request (mean, across all concurrent requests), Transfer rate, Failed requests.

- Повторить аналогичные замеры для второго типа сервера (многопоточного/select).

3. **Проведение нагрузочного тестирования TCP-эхо-сервера (альтернативный вариант):**
    - Сконфигурировать JMeter с потоком «TCP Sampler», который устанавливает соединение, отправляет тестовую строку, получает ответ и закрывает соединение (или сохраняет keep-alive). Задать различное число потоков (пользователей) и циклов.
    - Пример настройки: «Thread Group» с числом потоков = конкурентности, «TCP Sampler» с адресом и портом, отправляемой строкой "Hello", ожиданием ответа.
    - Запустить тесты для тех же уровней параллелизма (1, 10, 50, 100) и собрать статистику через «Summary Report» или «Aggregate Report» (среднее время, пропускная способность).

Примечание: если JMeter недоступен, студент может написать собственный многопоточный нагрузочный генератор на Python, но в отчёте необходимо обосновать выбранный подход и описать методику сбора метрик; тем не менее, приоритет отдаётся использованию стандартных инструментов.

4. **Сравнительный анализ:**
    - Свести результаты в таблицу: для каждой конфигурации (тип сервера, конкурентность) указать RPS, среднее время ответа, процент ошибок.
    - Построить графики (желательно с помощью Python matplotlib, gnuplot или Excel):
        - График 1: RPS от числа конкурентных клиентов для обоих серверов.
        - График 2: Среднее время ответа от числа конкурентных клиентов.
    - Определить точку насыщения для каждого сервера и объяснить причины различий.
5. **Выявление узких мест:**
    - Обратить внимание на использование CPU и памяти во время тестов (можно с помощью htop или диспетчера задач). Объяснить, почему RPS падает или перестаёт расти.
    - Обсудить влияние лимитов ОС (максимальное число дескрипторов, портов).
6. **Оформление отчёта:**
    - Включить снимки экрана с выводом утилит, таблицы, графики, анализ.

4. Методические указания по выполнению

- **Использование Apache Bench:**
    - Ключ -n задаёт общее число запросов, -c – количество одновременных соединений (конкурентность).
    - Для HTTP/1.0 серверов обязательно наличие заголовка Host: в запросе (ab добавляет автоматически).
    - Если тестирование проводится на локальной машине, убедитесь, что сервер не лимитирован портами (быстро исчерпаются клиентские эфемерные порты при большом количестве соединений). Для этого можно добавить задержку между запросами или уменьшить число запросов.
    - Пример интерпретации вывода ab:

```
Concurrency Level:      50
Time taken for tests:   2.345 seconds
Complete requests:      5000
Failed requests:        0
Requests per second:    2132.31 [#/sec] (mean)
Time per request:       23.45 [ms] (mean)
Transfer rate:          1234.56 [Kbytes/sec] received
```

- **Использование JMeter:**
    - Создать Test Plan:
        1. Thread Group (Number of Threads = N, Loop Count = M, чтобы всего запросов было N*M).
        2. Добавить Sampler → TCP Sampler. В настройках указать IP, порт, текст запроса (например, "ping\n"), установить флаги «Close connection» – если тестируется эхо с закрытием после каждого ответа.
        3. Добавить Listener → Aggregate Report и View Results Tree (для отладки).
    - Запустить тест. В Aggregate Report отобразятся среднее время, медиана, пропускная способность.
- **Тестирование итеративного сервера:**
  При подаче параллельных запросов с -c 50 все они будут устанавливать TCP-соединения (SYN), сервер их примет в очередь backlog, но обрабатывать будет по одному. В результате общее время теста резко возрастёт, а RPS упадёт. Возможны ошибки при переполнении очереди. В таком случае фиксируется значительная разница по сравнению с многопоточностью.

- **Построение графиков:**
  Данные из таблицы экспортировать в CSV и построить график в любом удобном инструменте. Обязательно подписать оси: ось X – "Concurrency", ось Y – "RPS" или "Response Time". На одном графике изобразить две кривые для разных серверов разными цветами.
- **Анализ ошибок:**
  Если часть запросов завершается ошибкой (Failed requests), необходимо классифицировать их (таймаут, сброс соединения) и принять меры: увеличить лимиты, снизить нагрузку. В отчёте отразить причины и влияние на результаты.
- **Дополнительный эксперимент (по желанию):**
  Провести тест на длительную стабильность (длительностью 60 секунд) и оценить, не деградирует ли сервер со временем (утечки памяти и т.п.).

5. Содержание отчёта

1. **Цель работы** и краткое описание тестируемых серверов.
2. **Методика тестирования:** описание инструментов, параметры тестов (число запросов, конкурентность), конфигурация серверов.
3. **Результаты:**
   - Таблица со сводными метриками для каждого сервера при разных уровнях параллелизма.
   - Скриншоты вывода ab или JMeter, подтверждающие цифры в таблице.
   - Графики зависимости RPS и времени ответа от числа конкурентных клиентов.
4. **Анализ:** интерпретация графиков, сравнение архитектур, идентификация узких мест (процессор, очередь, лимиты ОС).
5. **Выводы** (какой сервер показал лучшую производительность и почему, как можно улучшить слабые модели).

6. Контрольные вопросы

1. Что измеряет метрика RPS? Почему она падает при достижении предела пропускной способности сервера?
2. Чем отличается «Concurrency Level» от общего числа запросов в Apache Bench? Как эти параметры влияют на результаты теста?

3. Почему однопоточный итеративный сервер показывает низкую производительность при большом числе параллельных клиентов? Приведите пример из ваших тестов.
4. Какие системные ограничения могут повлиять на нагрузочное тестирование (file descriptors, ephemeral ports)? Как их обойти?
5. В чём преимущества и недостатки использования JMeter по сравнению с ab для тестирования TCP-серверов?
6. Как вы интерпретируете метрику «Time per request (mean)» в выводе ab при наличии 50 одновременных соединений? Отличается ли она от среднего времени обработки на стороне сервера?
7. Какую роль играет заголовок Connection: close при нагрузочном тестировании HTTP-сервера?
8. Предложите способы повышения масштабируемости вашего сервера, основываясь на результатах тестирования.

7. Критерии оценки

Оценка	Критерии
Зачтено	Студент продемонстрировал умение настраивать и применять ab и/или JMeter для нагрузочного тестирования. Проведена серия тестов не менее чем для двух разных архитектур серверов (итеративный vs многопоточный/select) с варьированием конкурентности. Получены численные метрики (RPS, время ответа), которые занесены в таблицу. Построены графики, иллюстрирующие разницу в производительности. В отчёте представлен грамотный анализ результатов, выявлены узкие места (например, падение RPS при исчерпании ресурсов). Студент на защите объясняет взаимосвязь между конкурентностью и метриками, может повторить эксперимент с другими параметрами и предсказать результат.
Не зачтено	Тестирование не проведено или проведено формально (одна конфигурация сервера, нет вариации параллелизма). Метрики не собраны либо неверно интерпретированы (например, RPS путается с количеством потоков). Отсутствуют графики или таблицы. Студент не может объяснить, как работают инструменты тестирования, не понимает причин различий в производительности. Сервер во время тестов падает, и не предпринято попыток диагностики.

Лабораторная работа №13

Тема: Отладка сетевых приложений с использованием Wireshark и логирования

Цель работы:
Освоить системный подход к отладке сетевого программного обеспечения, сочетающий инструментальное логирование ключевых этапов обмена данными и анализ сетевого трафика в Wireshark. Научиться выявлять типичные ошибки: нарушение целостности сообщений (склейка/разрывы), неверную сериализацию многобайтовых полей, преждевременное закрытие соединения, потерю данных из-за отсутствия цикличной отправки. Выработать навык сопоставления событий приложения с пакетами в дампе и целенаправленного исправления дефектов.

Формируемые компетенции (углубление):

- ПК-4: отладка и диагностика сетевых приложений, применение анализаторов трафика, верификация корректности протоколов.
- ПК-9: документирование процесса отладки, составление отчёта с доказательствами исправлений.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Интегрировать в существующий клиент-серверный код (на основе протокола с фиксированным заголовком из ЛБ №8) подробное логирование с временными метками, уровнями (отправка, приём, ошибки) и идентификацией потоков.
2. Намеренно внести в код одну или несколько типичных ошибок (или получить готовый дефектный вариант от преподавателя) – например, отсутствие цикла send для отправки больших сообщений, игнорирование порядка байт, отсутствие обработки склейки пакетов, некорректное закрытие сокета.
3. Запустить приложение и выполнить тестовый обмен, одновременно захватывая трафик в Wireshark на loopback-интерфейсе.
4. Проанализировать логи сервера и клиента, сопоставить их с данными из Wireshark, выявить расхождения и идентифицировать корневую причину ошибки.
5. Предложить и реализовать исправление, после чего повторить тест и продемонстрировать устранение дефекта (до/после).
6. Оформить процесс отладки в отчёте с иллюстрациями: фрагменты логов, скриншоты Wireshark с пояснениями.

2. Теоретические сведения

Комплексная отладка сетевых программ базируется на двух источниках информации:

- **Собственные логи приложения**, в которых разработчик фиксирует временные метки, содержимое переданных/принятых буферов, номера ошибок и важные состояния соединения.
- **Внешний наблюдатель трафика** (Wireshark), дающий объективную картину того, что действительно ушло в сеть.

Типичные ошибки, эффективно обнаруживаемые при таком подходе:

1. **Частичная отправка/приём:**
   Системный вызов send может вернуть значение меньше запрошенного, если буфер сокета заполнен. Разработчик забывает организовать цикл досылки, и часть данных остаётся неотправленной. В Wireshark видно, что TCP-сегмент короче ожидаемого, а последующий анализ тела сообщения показывает обрыв. Лог на сервере фиксирует «принято N байт, ожидалось M».

2. **Нарушение порядка байт (endianness):**
   Многобайтовые поля длины или кода типа записываются в хостовом порядке, а другая сторона ожидает сетевой (big-endian) или наоборот. В результате получатель считывает астрономическую длину
   (например, 0x04000000 вместо 0x04), что приводит к попытке чтения гигантского тела или, наоборот, к чтению меньшего количества байт. Wireshark показывает фактические байты заголовка, которые можно пересчитать. Лог помогает зафиксировать, какое значение извлёк парсер.

3. **Склейка и разрывы сообщений:**
   При использовании потокового протокола (TCP) без чёткого механизма выделения границ (например, заголовок с длиной используется, но не реализована функция recv_exact, или состояние парсера не учитывает возможность прихода нескольких заголовков в одном блоке данных). Склейка приводит к тому, что сервер воспринимает второе сообщение как продолжение тела первого. В Wireshark можно вручную разбить байтовый поток по ожидаемым границам и убедиться в наличии второго сообщения. Логи покажут, что после обработки первого сообщения не было попытки чтения следующего заголовка.

4. **Преждевременное закрытие (RST):**
   Если сторона закрывает сокет вызовом close (в Linux) без завершения передачи (например, после ошибки, без shutdown), ядро может отправить сегмент с флагом RST, обрывая соединение. Другая сторона теряет непрочитанные данные. В Wireshark явно виден пакет с RST. В логах можно заметить неожиданное завершение чтения с ошибкой «Connection reset by peer».

5. **Некорректная обработка ошибок (утечка дескрипторов):**
   При таймаутах или ошибках соединения сокет не закрывается, что со временем исчерпывает лимит дескрипторов. Wireshark может показать аномально большое

число соединений в состоянии TIME_WAIT или попытки SYN без ответа. Логирование ошибок со своевременным закрытием сокета помогает выявить такие места.

Методика отладки:

- **Гипотеза** → **Логирование** конкретных участков → **Тестовый прогон** с захватом трафика → **Сопоставление логов и дампа** → **Выявление несоответствия** → **Исправление** → **Повторный тест с подтверждением**.

Инструменты:

- fprintf(stderr, …) / print(…, file=sys.stderr) – для вывода логов в отдельный поток.
- Библиотеки логирования: syslog, log4c, Python logging. В учебных целях достаточно добавлять вывод с текущим временем (time_t или datetime) и идентификатором потока.
- Wireshark: фильтры по IP-адресам и портам, отслеживание TCP-потока (Follow TCP Stream), анализ шестнадцатеричного дампа.

3. Задание на выполнение работы

1. **Подготовка дефектного прототипа:**
    - Возьмите за основу рабочий код сервера и клиента из ЛБ №8 (протокол с заголовком «длина + тип»).
    - Внесите в код **минимум две ошибки** из следующего списка (по указанию преподавателя или на свой выбор, зафиксировав в отчёте):
        - a) В клиенте функция отправки send вызывается один раз без проверки возвращаемого количества байт; если сообщение больше ~4 КБ, оно обрезается.
        - b) В сервере при извлечении длины из заголовка не применяется ntohs (используется напрямую прочитанное значение), что на little-endian хосте даёт огромную длину.
        - c) Сервер после обработки одного сообщения не вызывает повторное чтение заголовка следующего, а пытается читать новое тело из оставшихся данных в буфере (склейка не обрабатывается корректно).
        - d) Клиент после отправки запроса сразу закрывает сокет без ожидания ответа (или вызывает close до полного чтения ответа), из-за чего на стороне сервера может возникать RST.
    - Убедитесь, что ошибка стабильно воспроизводится.

2. **Добавьте подробное логирование:**
    - В клиенте и сервере логируйте каждую операцию отправки/приёма с указанием количества байт и содержимого (в шестнадцатеричной или сокращённой строковой форме). Лог должен содержать временную метку (с точностью до миллисекунд).
    - Для сервера также выводите сообщения о подключении, отключении клиента, обнаруженной длине и типе пакета.
    - Пример строки лога:
      [14:23:05.123] Server: recv 3 bytes header: [00 05 01]
      [14:23:05.124] Server: parsed length=5, type=1
3. **Запустите тестовый обмен с захватом трафика:**
    - Запустите Wireshark на loopback-интерфейсе, задайте фильтр tcp port <ваш_порт>.
    - Запустите сервер, перенаправив его лог в файл.
    - Запустите клиента с тестовыми командами, вызывающими проявление ошибки.
    - Остановите захват, сохраните файл .pcapng.
4. **Анализ и исправление:**
    - Изучите лог сервера и клиента. Найдите запись, где поведение отклоняется от ожидаемого.
    - В Wireshark найдите соответствующие пакеты (по времени или по содержимому). Используйте «Follow TCP Stream», чтобы увидеть весь диалог. Проверьте, соответствует ли реально переданное количество байт заявленному в заголовке, правильно ли интерпретированы поля.
    - На основе анализа определите корневую причину.
    - Внесите исправление в код (например, добавьте цикл send_all, исправьте порядок байт, реализуйте повторное чтение заголовка).
    - Повторно запустите тест и убедитесь, что ошибка исчезла, а лог и дамп Wireshark подтверждают корректность.
5. **Задокументируйте результаты:**
    - Сделайте скриншоты Wireshark «до» и «после» с комментариями.
    - Приведите фрагменты логов с демонстрацией ошибочного и исправленного поведения.

- Опишите каждую найденную ошибку, её проявление, метод диагностики и внесённое исправление.

4. Методические указания по выполнению

- **Запись логов с временными метками:**
  В C используйте gettimeofday или clock_gettime для получения времени с микросекундами. В Python – datetime.datetime.now().strftime('%H:%M:%S.%f').
  Логи лучше выводить в unbuffered режим (setbuf(stdout, NULL) или sys.stderr) или принудительно fflush после каждой записи, чтобы в случае краха информация не потерялась.
- **Сопоставление логов и Wireshark:**
  Включите в Wireshark отображение абсолютного времени (View → Time Display Format → Date and Time of Day). Так вы сможете синхронизировать логи и захват.
- **Анализ ошибки a (частичная отправка):**
  - Ожидаемый симптом: клиент отправляет сообщение длиной, скажем, 5000 байт; в логах сервера видно получение заголовка с длиной 5000, но recv_exact (если он есть) зависает или получает меньше.
  - В Wireshark в поле TCP сегмента видно, что отправлено меньше 5000 байт прикладных данных в первом же пакете, остальное не досылается.
  - Исправление: написать функцию send_all, которая в цикле вызывает send до полной отправки всех байт.
- **Анализ ошибки b (порядок байт):**
  - Пример: заголовок содержит 0x00 0x05 (сетевое представление 5). На little-endian машине memcpy в uint16_t даст 0x0500 = 1280, если не применён ntohs. Сервер попытается прочитать 1280 байт тела, хотя реально данных нет.
  - В логах сервера видна неправдоподобно большая длина. Wireshark показывает корректные два байта 00 05.
  - Исправление: применить length = ntohs(len_net); после копирования из заголовка.
- **Анализ ошибки c (склейка, необработанный следующий заголовок):**
  - Сервер после прочтения тела первого сообщения не ищет следующий заголовок, а, допустим, ожидает disconnect. Клиент же отправляет второе сообщение в том же TCP-потоке. Сервер может воспринять данные второго заголовка как тело первого.

- В логах: после первого успешного сообщения нет попытки чтения ещё одного заголовка. В Wireshark видно, что второе сообщение пришло, но не обработано.
- Исправление: убедиться, что после обработки тела код возвращается в цикл, читающий заголовок, а не закрывает соединение.

- **Анализ ошибки d (преждевременное закрытие, RST):**
  - Клиент вызывает close сразу после отправки, несмотря на то, что должен ждать ответ. Сервер отправляет ответ, но клиент уже закрыл сокет, и его TCP-стек отвечает RST. Сервер получает ошибку при отправке ответа.
  - В логах сервера: ошибка «Broken pipe» или «Connection reset by peer» при send. Wireshark: после ответа сервера следует RST от клиента.
  - Исправление: клиент должен читать ответ и только потом закрывать соединение.
- **Общие рекомендации:**
  - Делайте diff-файлы исправлений, чтобы на защите можно было наглядно показать изменения.
  - При использовании Python следите за исключениями (ConnectionResetError).
  - Для сервера используйте обработку сигнала SIGINT для корректного завершения, чтобы избежать ложных RST.

5. Содержание отчёта

1. **Цель работы** и перечень анализируемых ошибок.
2. **Исходное состояние:** краткое описание протокола, листинг дефектных участков кода до исправления (с указанием вносимых ошибок).
3. **Методика отладки:** как настроено логирование, параметры захвата Wireshark.
4. **Процесс отладки для каждой ошибки:**
   - Описание симптома.
   - Фрагмент лога, иллюстрирующий ошибку.
   - Скриншот Wireshark с комментариями (стрелки, выделенные байты).
   - Объяснение корневой причины.
   - Фрагмент кода после исправления.
   - Доказательство исправления (лог и Wireshark «после»).

5. **Выводы** (чему научились, ценность комбинированного подхода, какие ещё ошибки могут быть обнаружены таким способом).

6. Контрольные вопросы

1. Чем полезно сочетание логирования и анализа трафика при отладке сетевых приложений? Приведите пример, когда одна только Wireshark не дала бы достаточно информации.
2. Как в Wireshark отличить нормальное завершение соединения (FIN) от обрыва (RST)? К каким последствиям для приложения приводит приход RST?
3. Какие ошибки могут быть скрыты в случае, если размер отправляемых данных всегда меньше размера буфера сокета? Как их спровоцировать?
4. Каким образом вы можете проверить, что много-байтовое поле в заголовке передано в правильном порядке байт, имея на руках только дамп Wireshark?
5. Опишите алгоритм действий, если сервер периодически теряет часть сообщения от клиента (например, вторую половину длинного текста). Какие проверки вы предпримете?
6. Почему при добавлении логирования важно использовать небуферизированный вывод или своевременно сбрасывать буфер?
7. Какие метрики времени следует включать в лог для точной корреляции с захваченными пакетами?

7. Критерии оценки

Оценка	Критерии
Зачтено	Студент продемонстрировал умение находить и исправлять реальные сетевые ошибки, пользуясь комбинацией логов и Wireshark. Внесены и затем исправлены не менее двух существенных дефектов (частичная отправка, порядок байт, склейка, RST и т.п.). Каждое исправление подтверждено логами и скриншотами дампов «до» и «после». Код снабжён адекватным логированием ключевых событий. Отчёт содержит подробный анализ процесса отладки. На защите студент объясняет методику поиска ошибок, интерпретирует показанный фрагмент дампа, может повторить отладку на новой заложенной ошибке.
Не	Студент не может найти ошибку или исправление неверно. Отсутствуют логи

Оценка	Критерии
зачтено	или скриншоты Wireshark. Устранение ошибок не подтверждено повторным тестированием. Студент путает типы ошибок, не может объяснить, как Wireshark помог в диагностике. Отчёт не отражает реального процесса отладки.

Лабораторная работа №14

Тема: Безопасность сетевого сервера: валидация входных данных

Цель работы:
Научиться выявлять и предотвращать типовые уязвимости сетевых приложений, связанные с некорректной обработкой входных данных. Освоить методы жёсткой валидации: ограничение длины сообщений, обнаружение и блокировка нулевых байтов в текстовых полях, защита от переполнения буфера. Сформировать понимание принципов построения безопасного сервера, устойчивого к атакам типа «отказ в обслуживании» (DoS), инъекциям и повреждению памяти.

Формируемые компетенции (углубление):

- ПК-4: обеспечение безопасности и надёжности сетевого программного обеспечения, валидация входных данных, предотвращение атак на этапе разработки.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Модифицировать существующий TCP-сервер (на основе протокола с фиксированным заголовком из ЛБ №8 или текстового чата из ЛБ №9), добавив строгую проверку всех поступающих от клиента данных.
2. Реализовать ограничение максимальной длины сообщения (тела) на уровне протокола — сервер должен отвергать запросы, превышающие установленный лимит, не допуская выделения огромных буферов.
3. Ввести запрет на наличие нулевых байтов (\0) в текстовых полях сообщений (если протокол подразумевает текстовые строки) для предотвращения атак типа Null Byte Injection.
4. Обеспечить защиту от переполнения буфера во всех операциях копирования и сборки данных, используя безопасные функции (strncpy, snprintf, memcpy с проверкой границ).

5. Смоделировать атаки: отправка запроса с запредельно большой заявленной длиной, вставка нулевого байта в середину текста, попытка вызвать переполнение буфера. Зафиксировать реакцию сервера (отбрасывание, разрыв соединения, логирование).
6. Провести анализ захваченного трафика в Wireshark, подтверждающий предотвращение уязвимости.
7. Описать в отчёте основные классы атак (Buffer Overflow, DoS, Null Byte Injection, Path Traversal, Format String) и методы защиты от них.

2. Теоретические сведения

Проблема безопасности входных данных:
Большинство уязвимостей сетевых приложений возникает из-за недостаточной проверки данных, поступающих от клиента. Классические угрозы:

- **Переполнение буфера (Buffer Overflow):** если программа копирует данные в буфер фиксированного размера без контроля длины, злоумышленник может записать произвольные данные за границей буфера, потенциально перезаписывая адреса возврата или служебные структуры. В сетевом коде это часто проявляется при использовании strcpy, sprintf, gets или неаккуратного memcpy.
- **Отказ в обслуживании (DoS — Denial of Service):** возможность исчерпать ресурсы сервера (память, процессор, дескрипторы). Типичный пример — отправка запроса с огромным заявленным размером тела, что вынуждает сервер попытаться выделить соответствующий буфер и вызвать исчерпание памяти.
- **Null Byte Injection (инъекция нулевого байта):** если сервер на языках C/C++ полагается на функции работы со строками (которые считают \0 концом строки), вставка нулевого байта в середину данных может привести к обходу проверок (например, валидация пути файла может пройти успешно для разрешённого расширения, а реальный доступ будет к иному файлу из-за преждевременного окончания строки). Даже в бинарных протоколах, где допустимы произвольные данные, нужно внимательно определять, какие поля считаются строками.
- **Атаки форматной строки (Format String):** если данные клиента напрямую передаются в функции типа printf без указания форматной строки, можно прочитать или записать в произвольные области памяти. Для предотвращения всегда следует использовать printf("%s", user_input) или не допускать вывода пользовательских данных в качестве управляющей строки.

Принципы безопасной обработки:

- **Валидация длины:** все входящие сообщения должны иметь явное ограничение сверху (например, максимальная длина тела — 4096 байт). Если заявленная длина превышает лимит, соединение следует разорвать или прочитать и отбросить лишние данные без сохранения.

- **Контроль содержимого:** для строковых данных необходимо гарантировать отсутствие нулевых байтов до проверяемой длины. Проверку можно выполнить с помощью memchr(buffer, '\0', length). При обнаружении — отбрасывать пакет и разрывать соединение.
- **Безопасные функции копирования:** использовать strncpy вместо strcpy, snprintf вместо sprintf, memcpy т олько после проверки, что целевой буфер достаточен. Постоянно отслеживать оставшееся свободное место в буфере.
- **Использование безопасных библиотек:** в C11 появились strcpy_s, memcpy_s; в Linux можно использовать strlcpy (при наличии). Но для переносимости важно хотя бы вручную контролировать размеры.
- **Разрыв при несоответствии:** сервер не должен пытаться «угадать» намерения клиента. При любой аномалии во входных данных лучше закрыть соединение и записать инцидент в лог, чем рисковать стабильностью системы.

Особенности протоколов с заголовком:
В протоколе, где длина тела передаётся в заголовке, сервер сначала читает длину. Здесь критически важно:

1. Ограничить допустимый диапазон длины (например, 0..MAX_BODY_LEN).
2. Если значение некорректно (слишком большое), не пытаться читать тело, а сразу закрыть сокет (с предварительным чтением и отбрасыванием данных может возникнуть риск приёма бесконечного потока; проще сделать close).

Имитация атак:
Для проверки защиты студент пишет «агрессивного» клиента, который намеренно посылает некорректные кадры: завышенную длину, кадр с \0 в середине текста. Сервер должен оставаться стабильным.

3. Задание на выполнение работы

1. **Выбрать базовый сервер:**
   Рекомендуется взять TCP-сервер с протоколом из ЛБ №8 (фиксированный заголовок: тип + длина). В качестве обрабатываемых типов сообщений ввести текстовое эхо (тип 0x01) и, возможно, передачу бинарных данных (тип 0x02). Для текстовых сообщений требуется валидация на отсутствие null-байтов. Для бинарных — только проверка длины.
2. **Реализовать жёсткую валидацию на стороне сервера:**
   - Ввести глобальную константу MAX_BODY_SIZE = 4096 (или разумное значение).

- При чтении заголовка и извлечении длины (length): если length > MAX_BODY_SIZE или length == 0 (если протокол не предусматривает пустые тела), то зафиксировать ошибку, логировать «Invalid body length», **немедленно закрыть соединение** (клиентский сокет) без попытки чтения тела.
- Если длина допустима, выделить буфер ровно под тело (или использовать стековый буфер, если длина мала). Прочитать тело с помощью recv_exact.
- Для типов сообщений, определённых как текстовые (например, TYPE_ECHO_TEXT), после успешного приёма тела проверить, что в буфере нет нулевых байтов (использовать memchr(buf, '\0', length)). Если найден – вывести предупреждение «Null byte detected», разорвать соединение (не обрабатывать сообщение).
- При записи логов или формировании ответа не вставлять полученные от клиента данные в форматную строку напрямую. Всегда использовать printf("Received: %s", buf) и т.п. (для C).
- Во всех местах, где происходит копирование данных (например, при сохранении в буфер клиента для чата или при формировании ответа), использовать функции с ограничением длины:
  - strncpy(dest, src, dest_size - 1); dest[dest_size - 1] = '\0';
  - snprintf(dest, dest_size, "...");
- При накоплении данных от recv в цикле обязательно проверять, что не вышли за размер выделенного буфера. Лучше иметь отдельную переменную для текущей позиции и перед каждым recv вычислять remaining = buf_size - current_pos. Если remaining == 0, то прекращать чтение и разрывать соединение (хотя по логике протокола такого быть не должно).

3. **Модифицировать клиент для тестирования уязвимостей:**
   - Разработать (или дополнить существующий) клиент, позволяющий отправить произвольный кадр. Например, возможность указать желаемую длину тела, которая может не соответствовать реальным данным, или вставить null-байт в тело.
   - Режим «Testing»: клиент может сформировать заголовок с длиной 0xFFFF (очень большой), либо отправить текстовое сообщение со встроенным \0.
4. **Провести тестирование и зафиксировать поведение:**
   - Обычный обмен: отправить несколько легитимных текстовых сообщений, убедиться, что сервер обрабатывает и возвращает эхо.

- Атака большой длиной: отправить запрос с length = 100000. Сервер должен прочитать заголовок, обнаружить превышение лимита, закрыть сокет.
- Атака нулевым байтом: отправить запрос типа текст с телом "Hello\0World". Сервер должен обнаружить \0 и разорвать соединение.
- Проверка на переполнение буфера: имитировать получение данных, превышающих ожидаемый размер, путём отправки вручную с помощью сырых сокетов (например, Python с send напрямую). Удостовериться, что сервер не крашится.
- Записать логи сервера, сделать снимки Wireshark, где видны отправленные некорректные кадры и ответная реакция (RST или просто закрытие соединения).

5. **Заполнить отчёт с описанием потенциальных атак и методов защиты:**
   - Таблица с классами атак (Buffer Overflow, DoS, Null Byte Injection, Format String), примерами и реализованными в лабораторной контрмерами.
   - Обсуждение важности валидации.

4. Методические указания по выполнению

- **Ограничение длины:**
  В функции обработки заголовка:

```c
#define MAX_BODY_LEN 4096
uint16_t length = ntohs(header.length);
if (length > MAX_BODY_LEN) {
    fprintf(stderr, "[SECURITY] Body length %u exceeds limit\n", length);
    close(client_sock);
    return;
}
```

После этого можно безопасно выделить char *body = malloc(length) (или использовать VLA, если допустимо и длина мала). Если malloc возвращает NULL — ошибка, закрыть сокет.

- **Обнаружение нулевого байта:**
  Для текстовых сообщений:

```c
if (type == TYPE_TEXT && memchr(body, '\0', length) != NULL) {
    fprintf(stderr, "[SECURITY] Null byte in text message from %s:%d\n", client_ip, client_port);
    free(body);
    close(client_sock);
    return;
}
```

- **Безопасное копирование:**
  Например, при сохранении имени файла в HTTP-сервере (из ЛБ10) или при формировании ответа:

```
char response[1024];
snprintf(response, sizeof(response), "Echo: %.*s", (int)length, body);
```

Использование формата %.*s с явной длиной предотвращает интерпретацию \0 как конца строки и не выходит за границы.

- **Защита от форматных строк:**
  Никогда не делать printf(user_input). Только printf("%s", user_input). В коде сервера, если выводятся логи, строго следить за этим.
- **Имитация атаки большой длины в клиенте:**
  Можно отправить заголовок с length = 50000, не отправляя тело. Сервер после чтения заголовка проверит длину и закроет соединение. Если сервер сначала пытается прочитать тело, он зависнет или упадёт при выделении памяти, поэтому проверку надо делать до выделения.
- **Wireshark анализ:**
  Фильтр по порту, найти кадр с аномальной длиной, показать, что сервер ответил RST (или FIN). В случае нулевого байта — в «Follow TCP Stream» будет видна только часть строки до \0, но сервер по логам покажет, что факт обнаружен и соединение разорвано.

5. Содержание отчёта

1. **Цель работы** и краткое описание модифицированного сервера.
2. **Принципы валидации:** список добавленных проверок с пояснениями.
3. **Листинг ключевых фрагментов кода** (обработка заголовка с лимитами, детектор null-байтов, безопасные функции).
4. **Описание тестовых атак:**
   - Сценарий с превышением длины: код атакующего клиента, скриншоты консоли сервера с сообщениями о нарушении, Wireshark-дамп до/после.
   - Сценарий с нулевым байтом: аналогично.
   - Проверка переполнения буфера: описание попытки и доказательство стабильности.
5. **Анализ угроз:** таблица (класс атаки → метод защиты → реализовано в коде / не реализовано).

6. **Выводы** (чему научились, важность Secure Coding, дальнейшие шаги по безопасности сетевого ПО).

6. Контрольные вопросы

1. Что такое переполнение буфера в сетевом контексте? Какие функции наиболее опасны и чем их заменить?
2. Каким образом злоумышленник может использовать нулевой байт для обхода проверки длины или расширения файла? Приведите пример для HTTP-сервера.
3. Почему при получении заведомо некорректной длины тела (например, 0xFFFF) сервер должен разрывать соединение, а не пытаться прочитать и отбросить эти данные?
4. Какие ещё атаки (кроме рассмотренных) могут угрожать простому эхо-серверу? Опишите хотя бы две.
5. Как использование функции snprintf вместо sprintf помогает предотвратить уязвимость форматной строки и переполнение буфера?
6. В чём разница между валидацией на стороне клиента и сервера? Почему нельзя полагаться только на клиентскую проверку?
7. Предложите, как можно ограничить частоту запросов от одного клиента для противодействия DoS.
8. Почему проверка на null-байт важна именно для строковых полей, а для бинарных не обязательна?

7. Критерии оценки (конкретизированные для ЛР №14)

Оценка	Критерии
Зачтено	Сервер реализует все предписанные проверки: ограничение максимальной длины тела, обнаружение null-байтов в текстовых сообщениях, применение безопасных функций копирования. При попытке атаки (завышенная длина, null-байт) сервер не падает, не выделяет чрезмерную память; соединение разрывается с записью в лог. Код комментирован, демонстрируется отсутствие уязвимостей с помощью специально созданных «вредоносных» клиентов. Отчёт содержит описание нескольких классов атак, скриншоты Wireshark и логов. Студент на защите может объяснить, как работает защита, и модифицировать лимиты или добавить проверку нового поля.

Оценка	Критерии
Не зачтено	Сервер не содержит проверок входных данных; успешно обрабатывает запрос с длиной тела 10 МБ, выделяя память и вызывая замедление или крах. Null-байты в тексте не детектируются, возможно злоупотребление. Используются небезопасные функции (strcpy, sprintf) без контроля. Студент не понимает опасности переполнения буфера и не может показать, как предотвратить. Отчёт не содержит анализа уязвимостей.

Лабораторная работа №15

Тема: Прикладной Keep-Alive и обнаружение мёртвых соединений

Цель работы:
Научиться реализовывать механизм контроля активности клиентов на прикладном уровне с помощью сообщений PING/PONG, а также освоить применение таймеров в серверных приложениях. Понять ограничения системного TCP Keep-Alive и преимущества собственного протокольного мониторинга для надёжного обнаружения «мёртвых» (бездействующих или аварийно отключившихся) соединений и своевременного освобождения ресурсов.

Формируемые компетенции (углубление):

- ПК-4: разработка отказоустойчивых серверов, управление ресурсами, реализация периодических проверок и таймаутов на прикладном уровне.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Модифицировать существующий многоклиентский сервер (на базе select/потоков) для поддержки прикладного Keep-Alive с использованием специализированных типов сообщений PING и PONG в рамках протокола с фиксированным заголовком (например, расширить протокол из ЛБ №8).
2. Реализовать на сервере механизм таймеров: для каждого подключённого клиента отслеживается время последней активности; при превышении интервала ожидания сервер отправляет клиенту запрос PING.
3. Обеспечить обработку ответов PONG (или любого другого сообщения) от клиента, сбрасывая счётчик неудачных проверок. Если ответ не получен в течение заданного таймаута, увеличивать счётчик пропущенных проверок.

4. При достижении порога последовательных неудачных проверок (например, 3 попытки) сервер принудительно закрывает соединение, освобождая ресурсы, и фиксирует событие в логе.
5. Реализовать клиент, поддерживающий автоматический ответ на PING сообщением PONG (или модифицировать существующий), для проверки нормальной работы; продемонстрировать корректное освобождение ресурсов при принудительном «убийстве» клиента без закрытия сокета.
6. Изучить опцию SO_KEEPALIVE на уровне сокета, сравнить её характеристики с прикладным Keep-Alive и обосновать преимущества/недостатки обоих подходов.

2. Теоретические сведения

Необходимость обнаружения мёртвых соединений
В долгоживущих соединениях (чат, уведомления, стриминг) клиент может отключиться некорректно: пропадание питания, обрыв сети, падение процесса без отправки FIN. Сервер будет бесполезно удерживать сокет и связанные структуры (буферы, память), что ведёт к утечке ресурсов и потенциальному DoS.

TCP Keep-Alive (системный уровень)
Опция SO_KEEPALIVE включает периодическую отправку пустых ACK-сегментов на транспортном уровне. Если удалённая сторона не отвечает, ядро может закрыть соединение. Однако стандартные интервалы очень велики (обычно tcp_keepalive_time = 7200 секунд, tcp_keepalive_intvl = 75 секунд, tcp_keepalive_probes = 9 попыток в Linux). Изменение системных параметров требует привилегий и влияет на все сокеты. Кроме того, Keep-Alive пакеты могут отфильтровываться NAT-ами, не отражая реальную доступность приложения. Прикладной контроль лишён этих недостатков.

Прикладной Keep-Alive (PING/PONG)
Сервер и клиент договариваются о специальных сообщениях протокола. В нашем случае можно расширить набор типов (например, TYPE_PING = 0x10, TYPE_PONG = 0x11). Сервер следит за временем, прошедшим с момента получения последних данных от клиента. Если интервал превысил KEEPALIVE_IDLE, сервер посылает клиенту кадр с типом PING и пустым телом. Клиент обязан ответить кадром PONG (или любым другим сообщением, что тоже сбрасывает таймер). Сервер ожидает ответа в течение KEEPALIVE_TIMEOUT. Если ответа нет, счётчик ошибок увеличивается; после KEEPALIVE_MAX_FAILS попыток соединение разрывается.

Реализация таймеров в однопоточном select-сервере
При использовании select() мы имеем возможность передать таймаут, который обычно используется просто для периодического выхода из блокировки. Этот же таймаут можно использовать для отслеживания Keep-Alive интервалов. После каждого выхода из select (по таймауту или по событиям) вычисляем текущее время и для каждого клиента проверяем, не истекло ли время бездействия. Если истекло и PING ещё не отправлен,

отправляем PING и запоминаем время. Если PING уже был отправлен и прошёл KEEPALIVE_TIMEOUT, увеличиваем счётчик. Это избавляет от необходимости создавать отдельный поток-наблюдатель.

Хранение состояния клиентов
Структура данных на каждого клиента дополняется полями:

- time_t last_active – метка последнего получения данных (или отправки PONG).
- int ping_sent – флаг, отправлен ли PING без ответа.
- time_t ping_time – время отправки последнего PING.
- int fail_count – количество последовательных неудач.

При получении любого корректного сообщения (включая PONG) сбрасываем fail_count = 0, ping_sent = 0, обновляем last_active.

3. Задание на выполнение работы

1. **Расширить протокол:**
   Добавить в спецификацию протокола (из ЛБ №8) два новых типа сообщений:
   - 0x10 – PING (запрос проверки, тело отсутствует).
   - 0x11 – PONG (ответ, тело отсутствует).
     (Либо можно использовать любой неиспользованный код.)
2. **Модифицировать сервер (на базе** select**-сервера или многопоточного):**
   - В структуру клиента добавить поля для Keep-Alive.
   - При каждом успешном приёме любого сообщения от клиента обновлять last_active, сбрасывать ping_sent и fail_count.
   - В главном цикле (после возврата из select или в случае таймаута) для каждого клиента проверять:
     - Если текущее время
       минус last_active превышает KEEPALIVE_IDLE (например, 15 секунд) и ping_sent == 0:
       - Отправить клиенту кадр с типом PING.
       - Установить ping_sent = 1, записать ping_time = now.
     - Если ping_sent == 1 и прошло более KEEPALIVE_TIMEOUT (например, 5 секунд) с ping_time:
       - Увеличить fail_count.
       - Если fail_count >= MAX_FAILS (например, 3):

            - Закрыть сокет клиента, удалить из списка, вывести лог "Client dead, disconnected".
        - Иначе: снова отправить PING (повторная попытка), обновить ping_time.
    - При реализации многопоточного сервера можно воспользоваться отдельным потоком-таймером или (сложнее) синхронизировать через общие переменные с основным потоком; однако методически проще выполнить задание на однопоточном select-сервере (из ЛБ №9).
3. **Модифицировать клиент:**
    - Добавить в клиент обработку входящих сообщений: если пришёл кадр типа PING, немедленно отправить кадр типа PONG. (Обычные эхо-сообщения также принимаются.)
    - Предусмотреть возможность «грубого» отключения клиента (Ctrl+C мгновенно или kill -9), чтобы проверить реакцию сервера.
4. **Настройка параметров (дефайны или конфигурация):**
    - KEEPALIVE_IDLE = 15 секунд (период бездействия до первой проверки).
    - KEEPALIVE_TIMEOUT = 5 секунд (ожидание ответа на PING).
    - MAX_FAILS = 3 (допустимое число последовательных неудач).
5. **Тестирование:**
    - Запустить сервер и несколько клиентов (telnet или свои программы).
    - Провести обычный обмен сообщениями, убедиться, что Keep-Alive не мешает активному диалогу (PING не отправляется, пока идёт активность).
    - Прекратить отправку данных от одного клиента и подождать ~15 секунд; в логах сервера должно появиться сообщение об отправке PING. Клиент должен ответить PONG, и соединение останется активным.
    - «Убить» клиента (kill -9) и наблюдать за сервером: после трёх неудачных PING с интервалами по 5 секунд сервер должен закрыть сокет и зафиксировать событие.
    - Продемонстрировать, что TCP Keep-Alive (если вообще включён) не срабатывает так быстро (можно просто сравнить времена).
    - Захватить трафик в Wireshark, показать кадры PING, PONG и последующий RST при разрыве.
6. **Сравнительный анализ (в отчёте):**

- Описать опцию SO_KEEPALIVE, как её включить (setsockopt), и указать системные значения таймаутов по умолчанию.
- Сделать вывод о преимуществах прикладного подхода.

4. Методические указания по выполнению

- **Добавление типов в протокол:**
  В заголовочном файле или в начале кода определить:

```
#define TYPE_PING 0x10
#define TYPE_PONG 0x11
```

- **Отправка PING сервером:**
  Сформировать буфер заголовка (3 байта): длина тела = 0, тип = TYPE_PING.
  Отправить клиенту с помощью send_all.
- **Обработка входящих сообщений на клиенте (Python пример):**

```
if msg_type == 0x10:  # PING
    send_frame(sock, 0x11, b"")
    continue
```

- **Реализация таймеров в select-сервере:**
  Установить таймаут select в 1 секунду (для отзывчивости к Keep-Alive).
  После select проверить текущее время (time(NULL) или clock_gettime). Пройти по всем клиентам, проверить условия. Важно не блокировать select на долгое время, чтобы иметь возможность отслеживать таймауты.

Псевдокод:

```c
struct timeval tv = {1, 0}; // 1 sec timeout
int ret = select(max_fd+1, &read_fds, NULL, NULL, &tv);
time_t now = time(NULL);
for (int i = 0; i < num_clients; i++) {
    client_t *c = &clients[i];
    if (c->fd == -1) continue;
    if (c->ping_sent) {
        if (now - c->ping_time >= KEEPALIVE_TIMEOUT) {
            c->fail_count++;
            if (c->fail_count >= MAX_FAILS) {
                close(c->fd);
                c->fd = -1;
                log("Dead client removed");
                continue;
            }
            // resend PING
            send_ping(c->fd);
            c->ping_time = now;
        }
    } else {
        if (now - c->last_active >= KEEPALIVE_IDLE) {
            send_ping(c->fd);
            c->ping_sent = 1;
            c->ping_time = now;
        }
    }
}
```

- **Обновление last_active:**
  При любом успешном чтении данных от клиента: c->last_active = time(NULL); c->ping_sent = 0; c->fail_count = 0;.
- **Тестирование грубой остановки клиента:**
  В Linux выполнить kill -9 <pid> клиента. Сервер перестанет получать ACK на PING, и через 3 * 5 = 15 секунд закроет соединение. В Wireshark можно увидеть, как сервер отправляет PING, но TCP-стек отвечает RST (так как на стороне клиента сокет закрыт и система посылает RST при получении данных). В этом случае сервер может сразу получить ошибку ECONNRESET при send или recv, что даже быстрее, чем исчерпание счётчика. Важно показать, что даже если RST не пришёл (например, из-за обрыва сети), наш счётчик сработает.
- **Сравнение с SO_KEEPALIVE:**
  Продемонстрируйте код на C: как включить системный TCP Keep-Alive на сокете:

```c
int optval = 1;
setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof(optval));
```

Объясните, что интервалы меняются в /proc/sys/net/ipv4/tcp_keepalive_* глобально, что непрактично.

5. Содержание отчёта

1. **Цель работы** и описание проблемы «мёртвых» соединений.
2. **Разработанный механизм:** спецификация расширения протокола (типы PING/PONG), алгоритм таймеров.
3. **Листинги сервера и клиента** с выделением функций обработки Keep-Alive и таймеров.
4. **Результаты тестирования:**
    - Скриншоты логов сервера при активном обмене (показывающие отсутствие лишних PING).
    - Логи и Wireshark-дамп при бездействии клиента: последовательность PING → PONG.
    - Логи и дамп при «убийстве» клиента: три PING, затем закрытие.
    - Сравнительная таблица характеристик TCP Keep-Alive и прикладного PING/PONG.
5. **Анализ:** обсуждение выбора интервалов, влияние на нагрузку, возможность использования в реальных приложениях.
6. **Выводы.**

6. Контрольные вопросы

1. В чём разница между системным TCP Keep-Alive и прикладным PING/PONG? Каковы типичные параметры TCP Keep-Alive в Linux?
2. Почему одного лишь SO_KEEPALIVE недостаточно для своевременного обнаружения обрыва связи?
3. Каким образом реализованы таймеры в вашем однопоточном сервере на основе select? Как часто вызывается select?
4. Как сервер отличает ситуацию, когда клиент просто не шлёт данные, от ситуации обрыва? Какие значения параметров Keep-Alive вы выбрали и почему?

5. Что произойдёт, если клиент получил PING, но его ответ PONG потерялся? Как это влияет на стабильность?
6. Какие ещё способы обнаружения разрыва можно реализовать помимо PING/PONG? (например, heartbeat, таймаут на чтение)
7. Как предотвратить ложные срабатывания Keep-Alive при кратковременных задержках сети?
8. Каким образом модифицировать сервер, чтобы он восстанавливал счётчик неудач при получении любого сообщения, а не только PONG?

7. Критерии оценки

Оценка	Критерии
Зачтено	Сервер реализует прикладной Keep-Alive с сообщениями PING/PONG, корректно отслеживает время бездействия и отправляет PING. После превышения лимита неудачных ответов (3) соединение разрывается, ресурсы освобождаются. Клиент отвечает PONG автоматически. Продемонстрировано обнаружение «мёртвого» клиента (имитация kill). Код содержит таймерную логику, не блокирует основной цикл. Отчёт содержит сравнение с TCP Keep-Alive, скриншоты и дампы. Студент на защите объясняет выбор таймаутов, может изменить лимиты и показать реакцию системы, обсуждает ограничения обеих методик.
Не зачтено	Сервер не реализует собственный Keep-Alive, либо реализация неверна (PING не отправляются, таймеры не срабатывают). Счётчики не обновляются при приёме данных, что ведёт к ложным разрывам. При обрыве клиента сервер не закрывает соединение, удерживая ресурс бесконечно. Студент не понимает разницы между системным и прикладным Keep-Alive. Отчёт не содержит подтверждений работоспособности.

Лабораторная работа №16

Тема: Высокопроизводительный асинхронный сервер на основе epoll (Linux)

Цель работы:
Познакомиться с механизмом асинхронного мультиплексирования ввода-вывода epoll, обеспечивающим высокую производительность и масштабируемость сетевых приложений в ОС Linux. Освоить создание сервера в режиме edge-triggered (ET), способного обслуживать тысячи одновременных соединений в одном процессе. Закрепить понимание преимуществ epoll перед select/poll с точки зрения потребления памяти, процессорного времени и удобства управления большим числом дескрипторов.

Формируемые компетенции (углубление):

- ПК-4: применение современных системных вызовов для построения высоконагруженного сетевого ПО, анализ производительности, оценка масштабируемости.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Использовать системные вызовы epoll_create1, epoll_ctl и epoll_wait для организации цикла событий.
2. Перевести слушающий и клиентские сокеты в неблокирующий режим и настроить edge-triggered (EPOLLET) отслеживание событий чтения и записи.
3. Реализовать однопоточный TCP-эхо-сервер, способный принимать и обслуживать более 5000 одновременных соединений без деградации производительности.
4. Решить проблему «голодания» и корректно обрабатывать частичное чтение/запись в неблокирующем режиме, реагируя на ошибки EAGAIN.
5. Сравнить архитектуру epoll с ранее изученными моделями (select, многопоточность) по использованию CPU и памяти в условиях массовой нагрузки (возможно, с помощью нагрузочного тестирования из ЛБ №12).
6. Оформить отчёт, содержащий сравнение характеристик epoll, select и poll, а также результаты стресс-тестирования.

2. Теоретические сведения

Недостатки select **и** poll**:**

- select ограничен FD_SETSIZE (обычно 1024) и требует пересоздания наборов дескрипторов при каждом вызове.
- poll не имеет фиксированного лимита, но также перебирает весь массив отслеживаемых структур при каждом вызове, что приводит к O(n)-сложности.
- Оба механизма при каждом возврате вынуждают приложение проходиться по всем дескрипторам, проверяя готовность, даже если активных мало.

Модель epoll**:**
epoll введён в Linux 2.6 и использует структуры в ядре, которые сохраняют зарегистрированные дескрипторы между вызовами. Приложение не пересылает весь список каждый раз. Функции:

- int epoll_create1(0) – создаёт экземпляр epoll, возвращает файловый дескриптор.
- epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event) – добавляет fd в наблюдение с заданными событиями.
- int epoll_wait(epfd, events, maxevents, timeout) – ожидает готовность, возвращает **только** готовые дескрипторы (структуры epoll_event). Количество обрабатываемых записей не зависит от общего числа зарегистрированных fd.

Режимы работы:

- **Level-triggered (LT)** – поведение аналогично poll: событие будет сообщаться до тех пор, пока данные доступны. Более прост в обработке.

- **Edge-triggered (ET)** – событие генерируется только при изменении состояния (например, при поступлении новых данных). Требует неблокирующих сокетов и полного чтения/записи до получения EAGAIN, иначе последующие данные могут быть пропущены. ET позволяет дополнительно снизить накладные расходы, но усложняет кодирование.

В данной работе предпочтительно использовать **Edge-triggered (EPOLLET)** для демонстрации высокопроизводительного подхода.

Неблокирующие операции и обработка EAGAIN:
При использовании ET после получения уведомления о доступности на чтение необходимо в цикле вызывать read/recv, пока не будет возвращена ошибка EAGAIN (или EWOULDBLOCK). Аналогично для записи, если отслеживаются события записи (EPOLLOUT). Игнорирование этого правила может привести к «зависанию» соединений, потому что новых уведомлений не будет.

Масштабируемость:
epoll позволяет обслуживать десятки тысяч соединений в одном процессе, так как затраты CPU прямо пропорциональны числу активных соединений, а не общему числу открытых. Память расходуется на буферы приёма и структуры ядра для каждого сокета.

3. Задание на выполнение работы

1. **Реализовать однопоточный TCP-эхо-сервер на epoll (Linux):**
    - Создать слушающий сокет, перевести в неблокирующий режим, выполнить bind и listen.
    - Создать экземпляр epoll: int epfd = epoll_create1(0);
    - Добавить слушающий сокет в epoll с флагами EPOLLIN | EPOLLET.
    - В главном цикле вызывать epoll_wait с таймаутом (например, 1000 мс).
    - Для каждого готового дескриптора:
        - Если это слушающий сокет — в цикле вызывать accept, пока не вернёт EAGAIN. Каждый полученный клиентский сокет сделать неблокирующим, добавить в epoll с флагами EPOLLIN | EPOLLET, выделить для него буфер (или структуру состояния).
        - Если это клиентский сокет и событие EPOLLIN — в цикле читать данные в буфер. При получении EAGAIN прекратить чтение. Если прочитано > 0 — отправить обратно (эхо). Если прочитано 0 — клиент закрыл соединение: удалить из epoll и закрыть сокет.

        - Если событие EPOLLOUT (при необходимости, если данные не удалось отправить сразу) — в цикле дописывать накопленные данные до EAGAIN.
    - Для каждого клиента поддерживать буфер для отправки (в случае, если отправка не может быть выполнена целиком). При возникновении необходимости отправки и ситуации, когда send возвращает меньше или ошибку, сокет регистрируется на EPOLLOUT.
    - Реализовать корректное удаление клиента: при закрытии (close) удалить дескриптор из epoll (не обязательно вручную, но лучше явно epoll_ctl с EPOLL_CTL_DEL).
    - Предусмотреть обработку сигнала SIGINT для корректного завершения с освобождением ресурсов.
2. **Настройка системы для большого числа соединений:**
    - Увеличить лимит файловых дескрипторов: ulimit -n 10000 (или более).
    - Убедиться, что диапазон эфемерных портов и параметры ядра не препятствуют созданию 5000+ соединений (при тестировании на одном хосте).
3. **Тестирование (стресс-тест):**
    - Запустить сервер.
    - Создать нагрузочного клиента, который устанавливает множество соединений (например, 5000) и периодически отправляет сообщения. Можно использовать модифицированный клиент из ЛБ2, написанный на Python с потоками, или специализированные утилиты (если возможно — JMeter с TCP Sampler, или самописный генератор).
    - Измерить утилизацию CPU и памяти сервера (htop).
    - Захватить трафик для небольшого подмножества соединений Wireshark (фильтр по порту), чтобы показать обмен.
4. **Сравнение с select/poll (в отчёте):**
    - Использовать результаты ЛБ9 (или провести дополнительные тесты) для сравнения поведения сервера под нагрузкой в 1000+ соединений.
    - Сделать вывод о преимуществах epoll: отсутствие копирования больших наборов дескрипторов, O(1) по активным, отсутствие ограничения FD_SETSIZE.

4. Методические указания по выполнению

- **Создание epoll и добавление слушающего сокета (C):**

```c
int epfd = epoll_create1(0);
if (epfd == -1) { perror("epoll_create1"); exit(EXIT_FAILURE); }
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;  // edge-triggered только для чтения
ev.data.fd = server_fd;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev) == -1) {
    perror("epoll_ctl: server_fd"); exit(EXIT_FAILURE);
}
```

- **Установка неблокирующего режима:**
  Включите fcntl(fd, F_SETFL, O_NONBLOCK) для слушающего и всех клиентских сокетов.
- **Цикл обработки готовности (псевдокод):**

```c
#define MAX_EVENTS 1024
struct epoll_event events[MAX_EVENTS];
while (running) {
    int nfds = epoll_wait(epfd, events, MAX_EVENTS, 1000);
    if (nfds == -1) {
        if (errno == EINTR) continue;
        perror("epoll_wait"); break;
    }
    for (int i = 0; i < nfds; i++) {
        int fd = events[i].data.fd;
        if (fd == server_fd) {
            // Принимаем все входящие соединения
            while (1) {
                struct sockaddr_in client_addr;
                socklen_t len = sizeof(client_addr);
                int client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &len);
                if (client_fd == -1) {
                    if (errno == EAGAIN || errno == EWOULDBLOCK) break;
                    perror("accept"); break;
                }
                set_nonblocking(client_fd);
                ev.events = EPOLLIN | EPOLLET;
                ev.data.fd = client_fd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &ev);
                // инициализация структуры клиента
            }
        } else {
            // Чтение данных
            if (events[i].events & EPOLLIN) {
                while (1) {
                    ssize_t n = recv(fd, buf, sizeof(buf), 0);
                    if (n > 0) {
                        // буферизация и эхо-ответ
                    } else if (n == 0) {
                        // соединение закрыто
                        close(fd);
                        epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL); // необязательно, ядро удалит пр
и close
                        break;
                    } else {
                        if (errno == EAGAIN || errno == EWOULDBLOCK) break;
                        perror("recv"); close(fd); break;
                    }
                }
            }
            if (events[i].events & EPOLLOUT) {
                // дописываем отложенные данные
                while (outstanding > 0) {
                    ssize_t n = send(fd, buf, outstanding, 0);
                    if (n > 0) { /* обновить буфер */ }
                    else if (n == -1 && (errno == EAGAIN)) break;
                    else { /* ошибка, закрыть */ }
                }
                // если всё отправлено, убрать EPOLLOUT
            }
        }
    }
}
```

- **Обработка записи (EPOLLOUT):**
  Для упрощения лабораторной работы можно не использовать EPOLLOUT, если сервер просто эхо-отвечает немедленно после чтения коротких сообщений, и буферов отправки достаточно. Однако для демонстрации полной картины edge-triggered подхода рекомендуется реализовать отложенную отправку и переключение на EPOLLOUT при необходимости.
- **Нагрузочное тестирование 5000 соединений:**
  Используйте самописного клиента на Python с использованием asyncio или потоков, который создаёт множество сокетов, подключается к серверу и периодически отправляет сообщения. Подсчитывайте успешные обмены. Альтернативно, можно использовать утилиту nping или ncat в скриптах, но это менее управляемо.
- **Сравнение с select:**
  Можно привести таблицу времени обработки 100, 500, 1000, 5000 соединений для select-сервера (ограниченного 1024) и epoll-сервера. При числе соединений больше FD_SETSIZE select просто не сможет работать.

5. Содержание отчёта

1. **Цель работы** и краткое описание технологии epoll.
2. **Схема работы сервера:** описание главного цикла с edge-triggered событиями.
3. **Листинг сервера** с выделением вызовов epoll.
4. **Нагрузочное тестирование:**
   - Скриншоты сервера (htop) при 5000+ открытых соединений, демонстрирующие низкую загрузку CPU.
   - Выводы клиентов-генераторов нагрузки (статистика успешных эхо-ответов).
   - Захват Wireshark для небольшой выборки соединений, подтверждающий корректный обмен.
5. **Сравнительный анализ:** таблица характеристик select, poll, epoll (пределы, сложность, механизм). Обсуждение результатов тестов.
6. **Выводы.**

6. Контрольные вопросы

1. В чём принципиальные отличия механизма epoll от select и poll?
   Почему epoll лучше масштабируется?

2. Что такое edge-triggered режим? Какие дополнительные требования он накладывает на код сервера?
3. Почему при использовании EPOLLET необходимо читать/писать данные в цикле до получения EAGAIN?
4. Каким образом вы реализовали обработку ситуации, когда send не может отправить все данные сразу?
5. Какие системные лимиты влияют на возможность обслуживания более 5000 одновременных соединений? Как их изменить?
6. Сравните работу epoll-сервера под нагрузкой с select-сервером. При каком количестве соединений select становится неприменим?
7. Можно ли использовать epoll в проектах для других ОС (Windows, macOS)? Какие аналоги существуют?
8. Как epoll помогает реализовать таймауты бездействия клиента (Keep-Alive)?

7. Критерии оценки (конкретизированные для ЛР №16)

Оценка	Критерии
Зачтено	Сервер использует epoll_create1, epoll_ctl, epoll_wait; слушающий и клиентские сокеты настроены на неблокирующий режим и edge-triggered (EPOLLET). Корректно выполняются операции чтения/записи в цикле до EAGAIN. Продемонстрирована стабильная работа с не менее чем 5000 одновременных соединений (подтверждено скриншотами и/или логами). Код обрабатывает ошибки, корректно удаляет клиенты. Отчёт содержит сравнение с select/poll, скриншоты тестов. Студент на защите объясняет устройство цикла epoll, может модифицировать параметры (изменить режим на LT, добавить таймауты) и отвечает на вопросы о масштабируемости.
Не зачтено	Программа не использует epoll, реализована через select/потоки. Edge-triggered обработка некорректна (нет цикла до EAGAIN), что приводит к «зависанию» соединений. Сервер не способен держать более нескольких десятков соединений, падает при нагрузке или не компилируется. Студент не понимает работу epoll и отличия от select. Отчёт отсутствует или не содержит доказательств тестирования.

Лабораторная работа №17

Тема: Документирование сетевого протокола и подготовка спецификации

Цель работы:
Сформировать навыки технического документирования сетевых протоколов. Научиться оформлять спецификацию протокола прикладного уровня в формате Markdown (с возможностью экспорта в PDF), включающую описание версий, транспортной привязки, форматов пакетов, типов сообщений, кодов ошибок и диаграмм последовательности. Закрепить умение переводить ключевые разделы документации на английский язык для международной аудитории.

Формируемые компетенции:

- ПК-9: подготовка технической документации на русском и английском языках, визуализация архитектуры протоколов, создание спецификаций для разработчиков.

1. Задачи работы

В результате выполнения лабораторной работы студент должен:

1. Выбрать реализованный ранее сетевой протокол (предпочтительно протокол с фиксированным бинарным заголовком из ЛБ №8 или протокол чата с PING/PONG из ЛБ №15) и составить его полную спецификацию.
2. Структурировать документ согласно предложенному шаблону: общее описание, версия и транспорт, формат кадра (с таблицей полей и байтовым представлением), перечень типов сообщений, коды ошибок, правила обработки.
3. Изобразить диаграмму последовательности (Sequence Diagram) для одного или нескольких ключевых сценариев взаимодействия (установка соединения, текстовое эхо, запрос статуса, процедура PING/PONG).
4. Перевести основные разделы спецификации (название, описание протокола, поля заголовка, типы сообщений) на английский язык, оформив их в виде двуязычного документа или отдельного англоязычного варианта.
5. Экспортировать итоговую документацию в PDF и включить её в отчёт.

2. Теоретические сведения

Назначение спецификации протокола

Спецификация служит единым источником истины для разработчиков клиентской и серверной частей, а также для сторонних реализаций. Она описывает:

- назначение протокола,

- допустимый транспорт (TCP/UDP, порты),
- модель соединения (запрос-ответ, потоковая, publish-subscribe),
- точный формат каждой единицы данных (кадра, сообщения),
- семантику полей,
- возможные коды ошибок и реакцию на них.

Хорошая спецификация не зависит от языка реализации, однозначно задаёт поведение и может быть использована для автоматической кодогенерации.

Структура документа (обычный подход):

1. **Введение (Introduction)** – цели протокола, область применения, версия.
2. **Транспорт и соединение (Transport and Connection)** – используемый протокол транспортного уровня (TCP), порт по умолчанию, поведение при установке/разрыве.
3. **Формат пакетов (Packet Format)** – описание кадра, таблица полей с указанием смещения, размера и назначения. Для бинарных протоколов пример:

Offset	Size	Field	Description
0	2	Length	Body length (big-endian)
2	1	Type	Message type identifier
3	N	Body	Variable-length payload

4. **Типы сообщений (Message Types)** – таблица с кодами, именами, направлением и кратким описанием.
5. **Коды ошибок (Error Codes)** – если протокол предусматривает ответы с ошибками, таблица кодов и значений.
6. **Диаграммы взаимодействия (Sequence Diagrams)** – визуализация основных сценариев.
7. **Соображения безопасности (Security Considerations)** – меры по валидации, ограничения длины, защита от инъекций (можно кратко).
8. **Приложения (Appendices)** – примеры дампов, ссылки.

Инструменты:

- Markdown – облегчённый язык разметки, легко конвертируется в PDF через pandoc или расширения VS Code.
- Диаграммы последовательности можно рисовать с помощью Mermaid (встроенная поддержка в GitHub/GitLab/VS Code) или онлайн-редакторов (например, sequencediagram.org) с экспортом изображений.

Требования к двухязычию:
Минимум – название протокола, заголовки разделов, названия полей и типов сообщений должны быть представлены на английском языке в скобках или в параллельной колонке. Желательно подготовить отдельную англоязычную версию спецификации.

3. Задание на выполнение работы

1. **Выбрать протокол для документирования.** Рекомендуется протокол из ЛБ №8 (длина + тип) или расширенный протокол с Keep-Alive из ЛБ №15. Можно взять протокол передачи файлов с CRC32 (ЛБ №6) или любой другой, реализованный в рамках практикума.
2. **Подготовить документ спецификации в формате Markdown.** Документ должен содержать следующие разделы (на русском языке):
    - **Название протокола**, версия (например, "Echo Protocol v1.0"), автор, дата.
    - **Введение** – назначение и краткое описание.
    - **Транспорт** – TCP, порт по умолчанию, установка соединения, политика закрытия.
    - **Формат пакета** – таблица с полями: смещение, размер, имя, описание. Для заголовка и тела указать байтовую структуру.
    - **Типы сообщений** – перечислить все коды, дать им имена и указать направление (клиент→сервер, сервер→клиент).
    - **Коды ошибок** – если протокол подразумевает ответы об ошибках, описать их.
    - **Основной сценарий** – диаграмма последовательности (sequence diagram) в виде рисунка (Mermaid-код или вставка изображения) для одного из взаимодействий, например, текстовый эхо-запрос и ответ.
    - **Дополнительные сценарии** (по желанию) – обработка PING/PONG, таймаут, ошибка.
3. **Перевести на английский язык следующие ключевые разделы:**
    - Название протокола, введение.
    - Таблица полей заголовка (Field Name, Size, Description).

- Таблица типов сообщений (Message Types).
- Подписи к диаграмме.
  Можно либо создать отдельную секцию "English Specification" внутри документа, либо параллельно представить англоязычный вариант. Перевод должен быть терминологически корректным.

4. **Визуализировать диаграмму последовательности:**
   - Использовать синтаксис Mermaid (блок sequenceDiagram). Пример:

```
sequenceDiagram
    participant Client
    participant Server
    Client->>Server: Header + Body (Type=0x01, "Hello")
    Server-->>Client: Header + Body (Type=0x01, "Hello")
```

   - Либо нарисовать в любом редакторе и вставить как изображение.
5. **Экспортировать документ в PDF:**
   - Использовать расширение Markdown PDF (VS Code) или pandoc -o spec.pdf spec.md. Убедиться, что форматирование корректно.
6. **Включить полученный PDF в отчёт** (в виде приложения) и кратко описать структуру документа.

4. Методические указания по выполнению

- **Шаблон Markdown:**
  Используйте заголовки #, ##, таблицы | | |, вставку изображений . Для Mermaid достаточно вставить блок ````mermaid` (если pandoc настроен с фильтром) или сделать скриншот из редактора.
- **Пример описания заголовка протокола:**

Offset	Size (bytes)	Field	Description
0	2	Length	Payload length, network byte order
2	1	Type	Message type
3	Length	Payload	Variable-length data

Английский дубликат: Field, Size, Description на английском.

- **Разработка диаграмм:**
  Mermaid-код можно сразу вставить в Markdown, а затем сконвертировать через pandoc с фильтром mermaid-filter или через онлайн-сервис mermaid.live, экспортировав изображение.

- **Перевод:**
  Рекомендуется использовать специализированные словари (RFC-терминология: payload – тело, header – заголовок, network byte order – сетевой порядок байт). Перевод должен быть грамматически верным. Проверить через сервисы проверки правописания.
- **Генерация PDF:**
  В VS Code установите расширение "Markdown PDF" и выполните команду markdown-pdf. В командной строке:
  pandoc spec.md --pdf-engine=xelatex -o spec.pdf (при необходимости установите pandoc и TeX). Убедитесь, что кириллица отображается корректно (добавить -V mainfont="DejaVu Serif").

5. Содержание отчёта

1. **Цель работы** и выбранный протокол.
2. **Содержание спецификации:** краткий обзор разделов документа со скриншотами (титульная страница, таблица полей, диаграмма).
3. **Листинг исходного Markdown-файла** (можно частично).
4. **Двуязычный фрагмент:** пример страницы с параллельным русским и английским описанием.
5. **Приложение:** итоговый PDF-документ спецификации.
6. **Выводы** (чему научились, важность документирования протоколов).

6. Контрольные вопросы

1. Какие разделы должна содержать спецификация сетевого протокола? Опишите содержание каждого.
2. Почему порядок байт в полях должен быть явно указан в спецификации? К чему может привести его отсутствие?
3. Какие инструменты позволяют создавать диаграммы последовательности в текстовом формате? Назовите преимущества такого подхода.
4. Зачем в спецификации перечислять коды ошибок? Как клиент должен реагировать на неизвестный код ошибки?
5. При переводе спецификации на английский язык какие термины требуют особого внимания (endianness, payload, handshake)?
6. Почему спецификация должна быть независимой от языка программирования? Приведите пример нарушения этого принципа.

7. Какие форматы документов удобны для хранения спецификации в системе контроля версий? Почему Markdown предпочтительнее Word?

7. Критерии оценки

Оценка	Критерии
Зачтено	Представлена полная спецификация протокола в формате Markdown и PDF. Документ содержит все обязательные разделы: транспорт, формат пакетов (с таблицей полей), типы сообщений, коды ошибок, диаграмму последовательности. Ключевые разделы переведены на английский язык (грамотно, терминологически точно). Диаграмма отражает реальное взаимодействие и оформлена аккуратно. В отчёте имеется скриншот или листинг Markdown, подтверждающий авторство. Студент на защите объясняет структуру протокола, может модифицировать спецификацию (добавить новый тип сообщения) и ответить на вопросы по документированию.
Не зачтено	Документация отсутствует, либо не соответствует реализованному протоколу. Спецификация неполна (нет типов сообщений, нет формата заголовка). Отсутствует перевод на английский язык либо он выполнен с грубыми ошибками, искажающими смысл. Диаграмма неверна или отсутствует. Студент не может объяснить, зачем нужна спецификация, и не ориентируется в структуре документа.

Лабораторная работа №18

Тема: Итоговый проект – разработка клиент-серверной системы с документацией и презентацией

Цель работы:
Интегрировать и продемонстрировать все практические навыки, полученные в ходе лабораторного практикума, путём самостоятельной разработки полноценной клиент-

серверной системы с элементами многопользовательского взаимодействия, надёжного протокола, тестирования и документирования. Отработать умение подготавливать технический отчёт и публичную презентацию, а также защищать результаты командной или индивидуальной работы перед аудиторией.

Формируемые компетенции (завершающий уровень):

- ПК-4: проектирование, реализация, тестирование и отладка сложных сетевых приложений с учётом требований надёжности, безопасности и производительности.
- ПК-9: оформление полного комплекта документации (спецификация протокола, руководство разработчика, описание тестов) на русском и английском языках, подготовка и проведение презентации.

1. Задачи работы

В результате выполнения итогового проекта студент должен:

1. Спроектировать и реализовать многопользовательскую клиент-серверную систему на основе TCP (или TCP/UDP в сочетании) – на выбор:
    - Многопользовательский текстовый чат с комнатами и приватными сообщениями.
    - Простая MUD-игра (Multi-User Dungeon) с перемещениями и взаимодействием.
    - Сетевые «крестики-нолики» или шашки с одновременными сессиями.
    - Иная тема, согласованная с преподавателем.
2. Применить не менее **пяти** ключевых тем курса:
    - TCP-сокеты и надёжная передача (ЛБ2, ЛБ3, ЛБ6);
    - Многопоточность или мультиплексирование select/epoll (ЛБ4, ЛБ9, ЛБ16);
    - Таймауты и обработка разрывов (ЛБ7);
    - Логирование и отладка (ЛБ13);
    - Безопасная валидация входных данных (ЛБ14);
    - Протокол с фиксированным заголовком или текстовым кадрированием (ЛБ8);
    - Прикладной Keep-Alive (ЛБ15) – поощряется, но не строго обязательно.
3. Подготовить **технический отчёт**, содержащий:
    - постановку задачи и описание архитектуры;

    - спецификацию протокола (по формату, освоенному в ЛБ17);
    - инструкции по сборке и запуску;
    - результаты функционального и нагрузочного тестирования (используя Wireshark, ab/JMeter, скриншоты);
    - анализ использованных технологий и обоснование решений.
4. Подготовить **презентацию** (10–12 слайдов), отражающую ключевые моменты проекта:
    - цель и обзор системы;
    - архитектурные решения (модель параллелизма, формат протокола);
    - демонстрацию работы (скриншоты, возможно короткое видео);
    - сложности и пути их преодоления;
    - выводы и возможные улучшения.
5. Перевести отдельные элементы документации и слайдов на английский язык (название протокола, описание сообщений, заголовки слайдов, выводы) – минимум 1 слайд полностью на английском или раздел в отчёте.
6. Публично защитить проект: выступить с презентацией, продемонстрировать работающее приложение, ответить на вопросы преподавателя и сокурсников, объяснить ключевые фрагменты кода.

2. Организация работы

- **Форма выполнения:** индивидуально или в парах (рекомендовано). При работе в паре каждый студент должен быть способен объяснить любой компонент системы и продемонстрировать личный вклад.
- **Сроки:** тема проекта утверждается преподавателем не позднее ЛБ №16. Защита – в последние две недели семестра по расписанию.
- **Исходный код** должен быть размещён в репозитории (GitHub/GitLab) с историей коммитов; в отчёте указывается ссылка на репозиторий. Код должен сопровождаться комментариями на русском или английском языке.
- **Отчёт** оформляется в электронном виде (PDF) и сдаётся вместе с исходным кодом до защиты. **Презентация** – в формате PDF или PPTX.

3. Требования к минимальной функциональности

Общие требования (для всех типов проектов):

- Сервер должен обслуживать не менее 10 одновременных клиентов без потери сообщений и без деградации времени ответа (подтверждается нагрузочным тестом).
- Обмен данными осуществляется по собственному прикладному протоколу с чётко определённым форматом сообщений (бинарный заголовок с длиной или текстовые строки с маркерами). Протокол должен быть документирован.
- Реализована обработка ошибок сети, восстановление после разрывов (автоматическое переподключение клиента или корректное удаление сессии на сервере).
- Ведётся лог работы сервера (подключения, отключения, ошибки, существенные события). Логи должны демонстрироваться на защите.
- Предусмотрена базовая защита от некорректных данных (ограничение длины, проверка типов, защита от crash).

По выбранной тематике дополнительно:

Чат:

- Комнаты (или общий чат) с возможностью отправки сообщений всем участникам.
- Уникальные имена пользователей, присоединение/отсоединение с оповещением.
- Приватные сообщения (опционально).

MUD-игра:

- Несколько локаций (комнат), перемещение между ними командами (north, south и т.д.).
- Возможность видеть других игроков в той же локации.
- Чат внутри игры (сказать, крикнуть).

Сетевые крестики-нолики:

- Создание игровых сессий, подключение двух игроков.
- Поочерёдная отправка ходов, проверка победы/ничьи.
- Обработка отключения соперника.

4. Методические указания по выполнению

- **Планирование:** начните с проектирования протокола на бумаге (как в ЛБ17), затем продумайте архитектуру сервера (многопоточность, epoll/select). Создайте каркас, который обменивается простыми сообщениями, и постепенно наращивайте функционал.

- **Многоразовое использование наработок:** допустимо использовать код из предыдущих лабораторных работ, но итоговая система должна быть цельным продуктом, а не механической суммой фрагментов. Переработайте общие компоненты (функции send_all, recv_exact, буферизацию) в отдельные модули.
- **Тестирование:** обязательно проведите не только ручное тестирование с несколькими клиентами, но и нагрузочное (ab для HTTP-подобных, либо самописный многопоточный генератор для собственного протокола). Зафиксируйте метрики в отчёте (число одновременных соединений, RPS, потребление памяти). Используйте Wireshark для выборочного контроля.
- **Логирование:** добавьте в сервер запись ключевых событий с временными метками (как в ЛБ13). Это очень поможет при отладке и при демонстрации.
- **Документирование:** спецификация протокола – обязательная часть отчёта. Следуйте структуре из ЛБ17: транспорт, формат пакетов, типы сообщений, коды ошибок, диаграмма последовательности для основных сценариев.
- **Презентация:** делайте слайды в деловом стиле. Минимизируйте текст, используйте диаграммы, скриншоты. Обязательно имейте резервный вариант демонстрации (записанное видео), на случай непредвиденных сетевых проблем. Репетируйте выступление, уложитесь в 7–10 минут.
- **Английский язык:** не менее одного слайда презентации должно быть полностью на английском (например, "Architecture" или "Conclusion"). В отчёте раздел спецификации протокола должен содержать англоязычную версию основных таблиц (как минимум, названия полей и типов сообщений). Это принципиальное требование, проверяющее ПК-9.

5. Содержание итогового отчёта

Отчёт должен включать следующие разделы:

1. **Титульный лист** с названием проекта, авторами, группой, датой.
2. **Введение** – обоснование выбора темы, цели проекта.
3. **Архитектура системы** – описание серверной и клиентской частей, модель параллелизма, структурная схема.
4. **Спецификация протокола** – полное описание в соответствии с требованиями ЛБ17 (включая двухязычные элементы).
5. **Инструкция по сборке и запуску** – зависимости, команды компиляции/интерпретации, настройка портов.
6. **Результаты тестирования:**
    - функциональное (сценарии использования, скриншоты клиентов);

    - нагрузочное (таблицы RPS и времени ответа, графики, скриншоты утилит);
    - анализ Wireshark (пример диалога с разбором пакетов).
7. **Заключение** – выводы, использованные темы курса, возможные улучшения.
8. **Приложения** – листинги ключевых модулей, полные диаграммы.

6. Контрольные вопросы (примерный перечень для защиты)

1. Почему вы выбрали именно такую модель параллелизма? Какие альтернативы рассматривались?
2. Как ваш протокол решает проблему границ сообщений? Покажите на примере из кода.
3. Какие меры безопасности были реализованы? Как сервер реагирует на слишком длинное сообщение?
4. Каким образом организован Keep-Alive или обнаружение мёртвых соединений?
5. Как вы тестировали производительность? Какие метрики считаете ключевыми?
6. Какая часть проекта была самой сложной, и как вы её преодолели?
7. Продемонстрируйте англоязычный фрагмент спецификации и объясните терминологию.
8. Если бы у вас было ещё две недели, что бы вы улучшили или добавили?

7. Критерии оценки

Оценка (зачёт/незачёт)	Критерии
Зачтено	Система функционирует, стабильно обслуживает не менее 10 одновременных клиентов. Применены не менее 5 тем курса (список в отчёте). Реализован собственный протокол с документированной спецификацией, включающей англоязычные элементы. Имеются логи и результаты тестирования (включая нагрузочное). Подготовлен полный отчёт и презентация (10–12 слайдов, один слайд полностью на английском). На защите студент(ы) демонстрируют работоспособность, уверенно отвечают на вопросы, объясняют архитектуру и ключевые фрагменты кода. Код закоммичен в репозитории с историей.

Оценка (зачёт/незачёт)	Критерии
Не зачтено	Система не запускается, падает при подключении нескольких клиентов. Отсутствует собственный протокол или документация к нему. Нагрузочное тестирование не проводилось или результаты неудовлетворительные (сервер не держит нагрузку). Презентация отсутствует, либо англоязычная часть не выполнена. Студент не может объяснить реализованную логику, путается в собственном коде. Значительная часть тем курса не отражена.