Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчет по лабораторным работам

Сети ЭВМ и телекоммуникации

Программирование сокетов протоколов TCP и UDP

Работу выполнила:
Городничева Л.В.
Группа: 43501/3
Преподаватель:
Алексюк А.О.

Санкт-Петербург
2017

1. Цель работы

Изучение принципов программирование сокетов с использованием протоколов TCP и UDP.

2. Программа работы

TCP:

- Реализация простейшего TCP сервера и клиента на ОС Linux и Windows.
- Реализация многопоточного обслуживания клиентов на сервере.
- Реализация собственного протокола на основе TCP для индивидуального задания (сервер на Windows, клиент на Linux).

UDP:

- Модификация сервера и клиента для протокола UDP на ОС Windows и Linux.
- Реализация собственного протокола на основе UDP для индивидуального задания (сервер на Linux, клиент на Windows).
- Обеспечение надежности протокола UDP, посредством нумерации пакетов и посылки ответов

Дополнительное задание:

Исследовать реальные прикладные протоколы. Необходимо “притвориться” клиентом и подключиться к одному из существующих общедоступных серверов. Использовать утилиты nc, telnet, PuTTY или другие им подобные, чтобы подключиться к серверу. Нельзя использовать специализированные клиенты для конкретного протокола! Необходимо вручную отправлять команды серверу в соответствии с протоколом. Нужно выполнить следующие опыты:

- Подключиться к HTTP-серверу и запросить веб-страницу
- Подключиться к FTP-серверу, запросить список файлов в директории и получить файл
- Подключиться к SMTP-серверу и попробовать отправить письмо
- Подключиться к POP3-серверу, попробовать проверить почту и получить письмо

В отправляемом и принимаемом письме должно быть мое имя и фамилия В ходе опытов необходимо попробовать как незащищенное plaintext-подключение, так и шифрованное TLS-подключение.

3. Простейший TCP сервер и клиент

Для создания сокета в библиотеках BSD-socket и WinSock имеется системный вызов socket:

Листинг 1: Вызов socket

```
1 int socket(int domain, int type, int protocol);
```

Для установления TCP-соединения используется вызов connect:

Листинг 2: Вызов connect

```
2 int connect(int s, const struct sockaddr* serv_addr, int addr_len);
```

Результатом выполнения функции является установление TCP-соединения с TCP-сервером.

Передача и приём данных в рамках установленного TCP-соединения осуществляется вызовами send и recv:

Листинг 3: Вызов send и recv

```
3 int send(int s, const void *msg, size_t len, int flags);
4 int recv(int s, void *msg, size_t len, int flags);
```

Параметр s – дескриптор сокета, параметр msg – указатель на буфер, содержащий данные (вызов send), или указатель на буфер, предназначенный для приёма данных (вызов recv). Параметр len – длина буфера в байтах, параметр flags – опции посылки или приёма данных.

Возвращаемое значение – число успешно посланных или принятых байтов, в случае ошибки функция возвращает значение -1.

Завершение установленного TCP-соединения осуществляется в библиотеке BSD-socket с помощью вызова shutdown:

Листинг 4: Завершение соединения

```
5 int shutdown(int s, int how);
```

По окончании работы следует закрыть сокет, для этого в библиотеке BSD-socket предусмотрен вызов close:

Листинг 5: Вызов close

```
6 int close(int s);
```

Структура TCP-сервера:

1. Создание сокета с помощью вызова socket. При этом локальные адрес и порт сокету назначаются из числа свободных;

2. Привязка сокета к удаленному адресу с помощью вызова bind (для сервера устанавливается адрес INADDR_ANY, который позволяет получать данные с любых адресов);

3. Перевод сокета в состояние прослушивания соединений с помощью вызова listen. При этом данные передавать через сокет нельзя, единственная его задача – получение запросов на соединение;

4. Прием соединений клиентов с помощью вызова accept. Данный вызов блокирует выполнение потока, пока не придет соединение от клиента, в результате которого создастся новый сокет, связанный с адресом клиента;

5. Далее происходит прием/передача данных с помощью recv/send.

6. Закрытие сокета с помощью shutdown и close

Структура TCP-клиента:

1. Создание сокета с помощью вызова socket. При этом локальные адрес и порт сокету назначаются из числа свободных;

2. Установка соединения с помощью connect

3. Далее происходит прием/передача данных с помощью recv/send.

4. Закрытие сокета с помощью shutdown и close

Простейшее клиент-серверное приложение, реализующее описанные структуры и использующее стандартные функции управления сокетами, приведены в приложении (лис.7 и лис.8).

В Windows вместо вызовов write и read происходят вызовы send и recv.

4. Простейший UDP сервер и клиент

В случае установленного адреса по умолчанию для протокола UDP (вызов connect) функции для передачи и приёма данных по протоколу UDP можно использовать вызовы send и recv. Если адрес и порт по умолчанию для протокола UDP не установлен, то параметры удалённой стороны необходимо указывать или получать при каждом вызове операций записи или чтения. Для протокола UDP имеется два аналогичных вызова sendto и recvfrom:

Листинг 6: Передача и прием данных

```
7 int sendto(int s, const void *buf, size_t len, int flags, struct sockaddr *to,
    ↪ int* tolen);
8 int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, int
    ↪ * fromlen);
```

Структура UDP-сервера:

1. Создание сокета с помощью вызова socket. При этом локальные адрес и порт сокету назначаются из числа свободных;
2. Привязка сокета к удаленному адресу с помощью вызова bind (для сервера устанавливается адрес INADDR_ANY, который позволяет получать данные с любых адресов);
3. Далее происходит прием/передача данных с помощью recvfrom/sendto.
4. Закрытие сокета с помощью close

Структура UDP-клиента:

Структура UDP-клиента ещё более простая, чем у TCP-клиента, так как нет необходимости создавать и разрывать соединение.

1. Создание сокета с помощью вызова socket. При этом локальные адрес и порт сокету назначаются из числа свободных;
2. Привязка сокета к удаленному адресу с помощью вызова bind
3. Установка соединения с помощью connect
4. Далее происходит прием/передача данных с помощью recv/send.
5. Закрытие сокета с помощью shutdown и close

Простейшее клиент-серверное приложение, реализующее описанные структуры и использующее стандартные функции управления сокетами, приведено в приложении (лис.9 и лис.10).

5. Многопоточное обслуживание клиентов на сервере

Для организации работы сервера с несколькими клиентами необходимо создавать новый сокет для каждого из клиентов, а старый созданный сокет сделать слушающим: сокет будет использоваться только в listen и accept для подключения новых клиентов.

Подключение клиентов необходимо сделать в цикле. После подключения каждого нового клиента выделять отдельный поток для общения с ним. В этом потоке должна вызываться функция работы с клиентом.

Многопоточный TCP-сервер приведен в приложении (лис.11)

В приведенной выше программе видно вызов accept в бесконечном цикле, а также создание нового потока, который начинает работу с функции formulti, передавая ей newsockfd.

6. Индивидуальное задание

6.1. Задание

Разработать клиент-серверную систему дистанционного тестирования знаний, состоящую из централизованного сервера тестирования и клиентов тестирования.

6.2. Основные возможности

Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от клиентов
3. Поддержка одновременной работы нескольких клиентов через механизм нитей
4. Регистрация клиента, выдача клиенту результата его последнего теста, выдача клиенту списка тестов
5. Получение от клиента номера теста
6. Последовательная выдача клиенту вопросов теста и получение ответов на вопросы
7. После прохождения теста – выдача клиенту его результата
8. Обработка запроса на отключение клиента
9. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

1. Установление соединения с сервером
2. Посылка регистрационных данных клиента

3. Выбор теста
4. Последовательная выдача ответов на вопросы сервера
5. Индикация результатов теста
6. Разрыв соединения
7. Обработка ситуации отключения клиента сервером

6.3. Настройки приложений

Разработанное клиентское приложение должно предоставлять пользователю возможность введения идентификационной информации, настройки IP-адреса или доменного имени, а также номера порта сервера информационной системы.

Разработанное серверное приложение должно предоставлять пользователю возможность настройки начальной точки входа в информационную систему каждого пользователя.

6.4. Прикладной протокол TCP

После подключения клиента к серверу, сервер ждем команд от клиента. Существует 5 возможных команд от клиента.

Команда имеет вид:

<команда> <аргумент>

Для всех отправляемых сообщений установлена фиксированная длина в 20 байт. Если необходимо послать больше, то отсылается несколько пакетов по 20 байт.

Поддерживаемые команды клиента:

1. end - завершение работы клиента
2. register - регистрация клиента (выполняется в две команды, после нажатия enter клиенту необходимо ввести <логин> <пароль>)
3. show - выведение списка существующих тестов
4. getResult - выведение результата последнего пройденного теста
5. getTest <номер теста> - выведение списка вопросов определенного теста

Если клиентом введена несуществующая команда, то ничего не происходит.

Ответы сервера на команды клиента:

1. Нет сообщения от сервера
2. "You are successfully logged in! / Try another password or login! / You are successfully registered / Wrong number of argumenets. You can't create login and password with spaces"
3. Выводится список всех тестов, например "1 - Math test 2 - Phsycological test"
4. "Your last result was: [x out of y]"(где x - количество правильных ответов, y - количество всех ответов)
5. Выведение списка вопросов

   Действия сервера:

1. Сервер закрывает сокет
2. — / — / Регистрация клиента (запись логина и пароля в файл registered.txt)
3. Посылка списка вопросов из файла list_tests.txt
4. Посылка результата последнего теста клиента из файла registered.txt
5. Последовательная посылка списка вопросов из конкретного файла test<номер теста>.txt, а затем сверка ответов с правильными ответами теста из файла anstest<номер теста>.txt

Сообщения об ошибках:

1. recv failed with error: Ошибка при приеме данных
2. socket failed with error: Не удалось создать сокет
3. getaddrinfo failed with error: Не удалось получить адрес
4. ERROR reading from socket: Ошибка при приеме данных
5. accept failed with error: Ошибка при подтверждении подключения нового клиента
6. WSAStartup failed with error: Не удалось инициализировать структуру WSAStartup
7. bind failed with error: Ошибка привязки сокета к IP адресу
8. listen failed with error: Ошибка прослушивания сокета при ожидании новых клиентов
9. send failed with error: Ошибка при передаче данных

Формат содержимого файла registered.txt (логины, пароли и последние результаты тестирования клиентов):

<логин> <пароль> <x out of y>

Формат содержимого файла list_tests.txt (список тестов):

<номер теста> - <имя теста>

Формат содержимого файла test<номер теста>.txt (вопросы теста):

QUESTION:

<вопрос>

ANSWERS:

<номер ответа>) <ответ>

<номер ответа>) <ответ>

<номер ответа>) <ответ>

Формат содержимого файла anstest<номер теста>.txt(правильные ответы на определенный тест):

<ответ>

<ответ>

Поддерживаемые команды сервера:

1. end - завершение работы сервера
2. close - отключение определенного клиента
3. send - посылка сообщения клиенту
4. show - список подсоединившихся клиентов

6.5. Прикладной протокол UDP

В отличие от TCP, в прикладном протоколе UDP сервер поддерживает лишь одну команду: end - завершение работы сервера. В остальных командах отсутствует необходимость, т.к. в UDP не устанавливается соединение и все обмены со всеми клиентами происходят через один единственный сокет.

В начале каждого пакета записан его номер в текущем обмене. Далее через пробел следует само сообщение.

Сообщения об ошибках:

1. recv failed with error: Ошибка при приеме данных
2. socket failed with error: Не удалось создать сокет
3. getaddrinfo failed with error: Не удалось получить адрес

4. WSAStartup failed with error: Не удалось инициализировать структуру WSAStartup

5. bind failed with error: Ошибка привязки сокета к IP адресу

6.6. Описание архитектуры и особенности реализации TCP

Изначально происходит создание серверного сокета, который затем будет прослушивать подключения на опрделенном, заданном при создании, порту. Для того, чтобы сервер мог параллельно с установкой соединений самостоятельно писать и выполнять команды, создается новый поток (acceptThreadFunction), в котором как раз и осуществляется ожидание новых подключений. В основном потоке сервер принимает команды из командной строки, а затем осуществляет выполнение одной из следующих команд: завершение сервера, посылка сообщения определенному клиенту, закрытие определенного клиента и просмотр списка клиентов, подключенных в данный момент. Все операции с коллекцией (poolOfSockets), хранящей в настоящее време подключенные сокета, защищены мьютексами, так как к коллекции возможно одновременное обращение из нескольких потоков. В поточной функции acceptThreadFunction происходит ожидание новых подключений в цикле. Когда клиент подключается к серверу, клиентский сокет добавляется в коллекцию с активными сокетами (poolOfSockets), а на работу клиента выделяется отдельный поток(workingFlow), в который передается сокет для общения с клиентом. Поточная функция workingFlow ожидает команды от клиента, приведенные выше. В случае, если пришла строка об отключении клиента, клиентский сокет удаляется из коллекции сокетов и закрывается. Если же пришла строка о регистрации, выполняется функция регистрации, в которой изначально считываются строки из файла, чтобы была информация о существующих клиентах. Затем сервер ждет логин и пароль от клиента и сверяет их с уже существующими данными. Если такого логина, с которым пытается подключится клиент, не существует, создается новая запись в файле, а клиенту посылается сообщение о том, что он зарегистрирован. Если же логин, с которым клиент пытается зарегистрироваться, уже существует в регистрационном файле, то существует 2 варинта последующих действий. Если пароль совпадает с тем, что имеется в регистрационном файле, то клиент считается залогиненым и его сокет заносится в коллекцию залогиненых сокетов, если же пароль не совпал, то либо клиент ошибся в пароле, пытаясь залогинится, либо клиент при регистрации указал уже существующий логин. В последнем варианте клиенту посылается сообщение о том, что он ему стоит попробовать другой логин либо поробовать другой пароль. Считываение пароля с логином продолжается, пока клиент либо не зарегистрируется, либо не залогинится. Если пришла команда с запросом тестов, то название тестов считывается из определенного файла в одну строку. Может случится такая ситуация, что размер такой строки превысит длину единичной посылки, тогда такое сообщение разобьется на несколько посылок равно опредленной протоколом длины и передастся клиенту. Если сервер получил команду на запрос результата клиента, то мы изначально

проверяем залогинен ли данный клиент, если он залогинен, то результат считывается из специального файла по логину и паролю клиента. Если пришла команда о выдаче определенного теста, также происходит проверка, залогинен ли пользователь, затем по номеру тест выбирается определенный файл с вопросами и вариантами ответов. Вопрос может превысить длину сообщения, заданную протоколом, поэтому сообщение разибвается на несколько сообщений длиной определенной протоколом. После отправки вопроса и вариантов ответа, сервер ожидает ответ клиента. Приняв ответ, записывает ответ в вектор и посылает следующую порцию вопрос-варинты ответов. Отправив все вопросы, сервер считывает верные варинаты ответов из специального файл и сверяет с ответами клиента, формируя тем самым результат клиента. Этот результат записывается в файл с логином и паролем и посылается клиенту.

Когда сервер хочет завершить свою работу, ему необходимо отключить свой слушающий сокет, все клиентские сокеты и завершить все открытые потоки. Для завершения клиентских сокетов, необходимо выйти из цикла while. Поэтому для завершения сервера закрывается слушающий сокет, который как раз доступный из главного потока сервера, а затем начинаем ожидать завершения прослушивающего потока. После выхода из цикла прослушивания (так как из главного потока слушающий сокет был закрыт), закрываем все клиентские сокеты и ожидаем завершения потоков серверов. Так как произошло закрытие клиентских сокетов, то в потоке чтения и выполнения клиентских команд получается ошибка при выполнении команды readn, что приводит к выходу из цикла и завершению потока. Таким образом, после завершения всех клиентских потоков, завершается слушающий поток, а также происходит выход из while в главном потоке сервера, где производится финальная команда WSACleanup();

На сервере определенно 2 специальные функции считывания и отправки данных: readn и sendn. Первая гарантированно считывает n байт следующим образом: в цикле контрлируем количество реально принятых байт (k), затем в передвигаем указатель у буфера на k и уменьшаем общее число еще несчитанных байт (nLeft) на k. Вторая - посылает сообщение, длина (n) которого больше длины сообщения, которая задана в протоколе. Так посылаем первую посылку длиной равной протокольной длине и увеличиваем количество отосланных байт на это количество. Повторяем посылки в цикле, передвигая указатель в буфере на протокольную длину и увеличивая количество отосланных байт. В конце концов, количество отосланных байт станет больше или равно заданному количеству байт n. Отправка завершится.

6.7. Описание архитектуры и особенности реализации UDP

Основная логика взаимодействия клиента и сервера осталась такой же, как у TCP. Однако в данном случае нет необходимости создавать отдельные сокеты и потоки для каждого клиента. Обработка всех запросов происходит в основном потоке. Со стороны

клиента функция приема сообщения от сервера также была перенесена в основной поток. Для отправки и приема сообщений написаны функции sendOneTime и recvOneTime. Данные функции необходимы для осуществления контроля целостности данных. Функция sendOneTime перед отправкой сообщения вставляет в начало пакета его порядковый номер, а recvOneTime осуществляет проверку этого номера и вывод ошибки в случае несовпадения фактического номера пакета с ожидаемым.

7. Дополнительное задание

Исследуем реальные прикладные протоколы. Притворимся клиентом и подключимся к одному из существующих общедоступных серверов.

7.1. Подключение к HTTP-серверу и запрос веб-страницы

С помощью команды telnet подключаемся к HTTP-серверу:

```
1 telnet tproger.ru 80
```

tproger.ru - подключаемый сервер, 80 - номер порта.

Затем с помощью команды запрашиваем определенную веб-страницу:

```
1 GET /category/news/ HTTP/1.1
```

Веб-страница успешно выведена (рис.7.1).

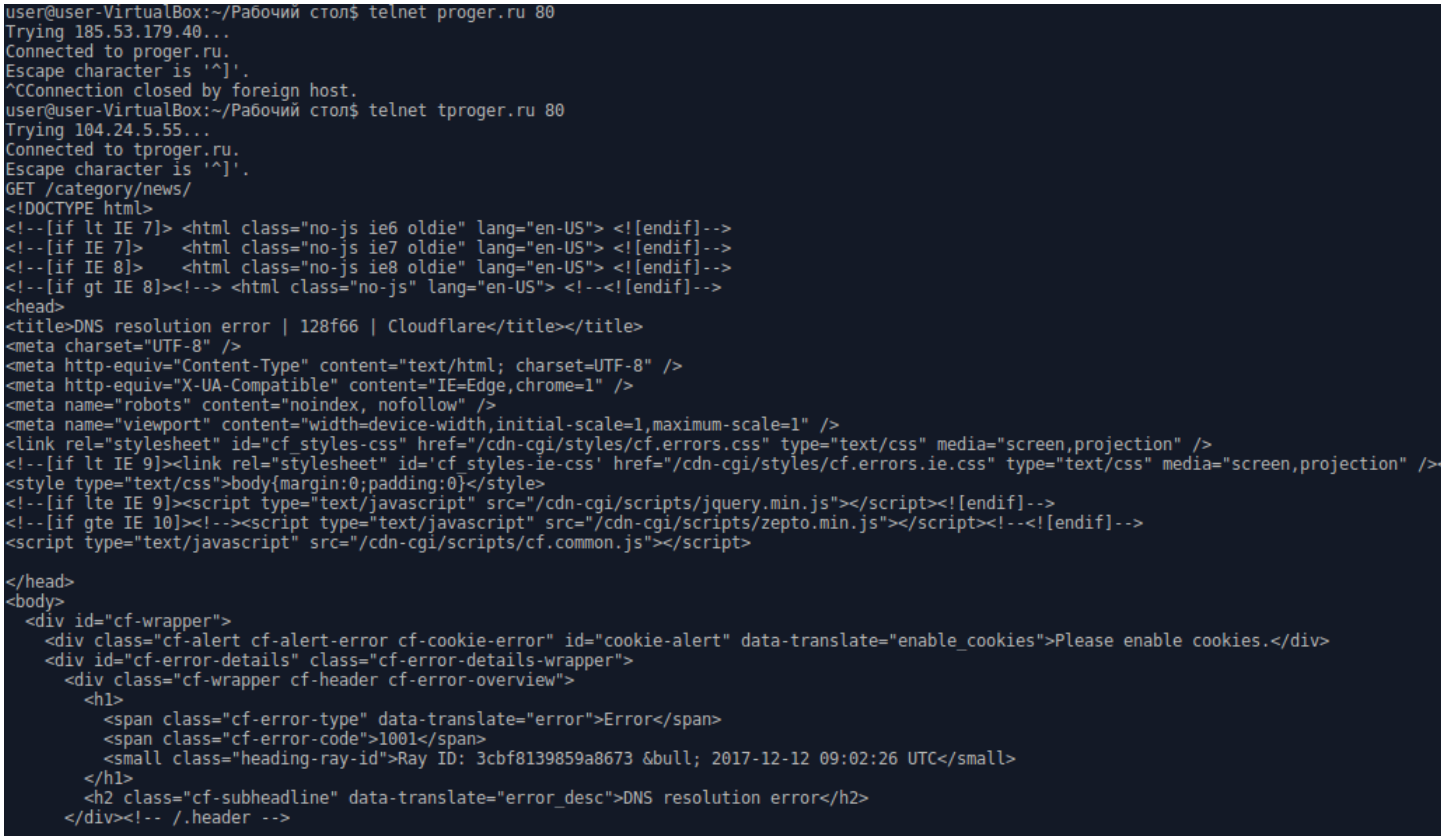

Рисунок 7.1. Подключение к HTTP-серверу

7.2. Подключение к FTP-серверу, запрос списка файлов в директории и получение файла

С помощью команды telnet подключаемся к FTP-серверу:

```
1 telnet ftp.stat.duke.edu 21
```

ftp.stat.duke.edu - ftp-сервер, к которому происходит подключение, 21 - порт.

Вводим имя пользователя и пароль. Затем необходимо перейти в пассивный режим с помощью команды pasv. С помощью цифр кодируется IP-адрес и порт для соединения. Создаем соединение к указанному IP-адресу через необходимый порт (порт высчитывается по формуле p1*256 + p2, где p1 и p2 - это два последних числа в присланном сообщении сервера). Стоит уточнить, что здесь используется одно соединение на одну команду. С помощью команды pwd и cwd переходим по папкам рис.7.2, пока не найдем необходимый файл. Для скачивания файла используем команду retr <имя файла>.

Файл успешно скачан (рис.7.3).

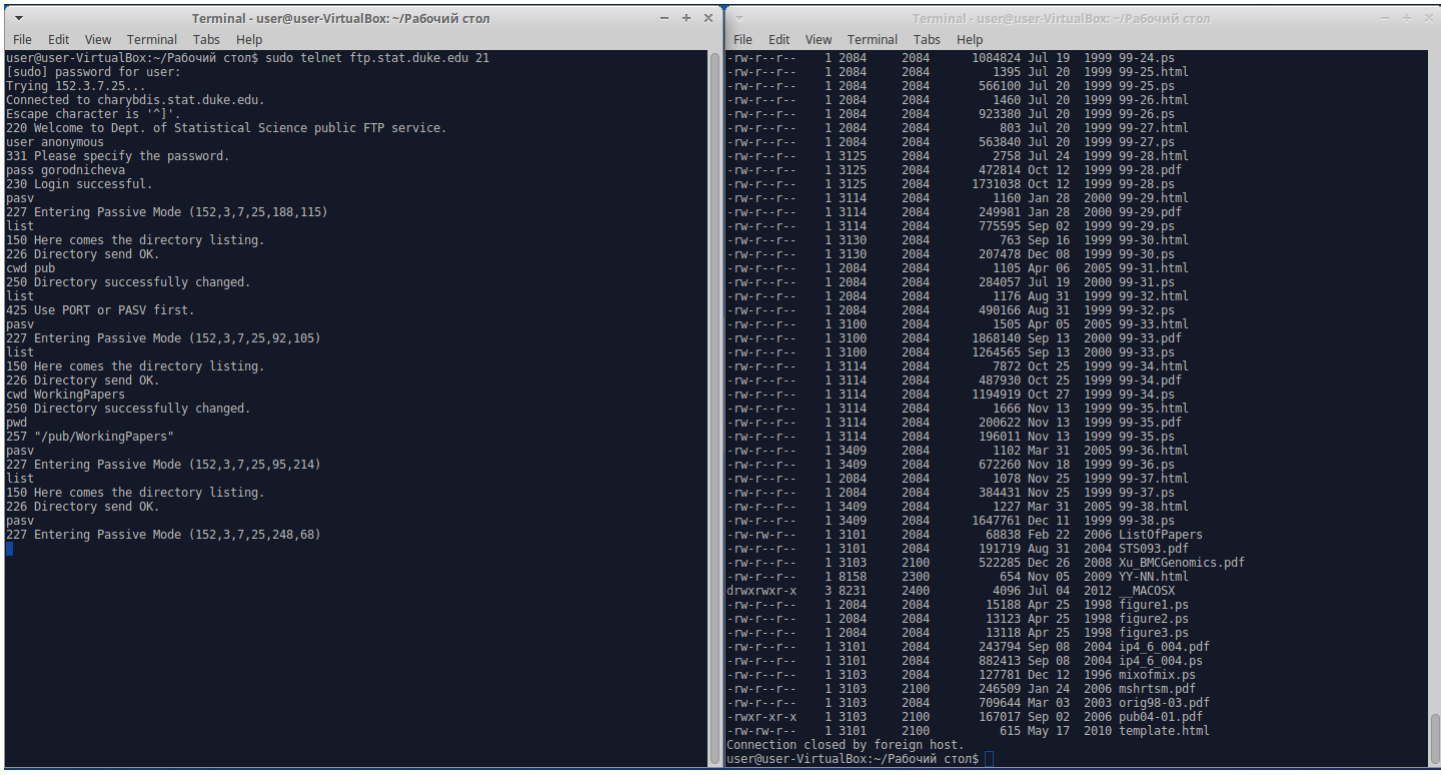

Рисунок 7.2. Подключение к FTP-серверу

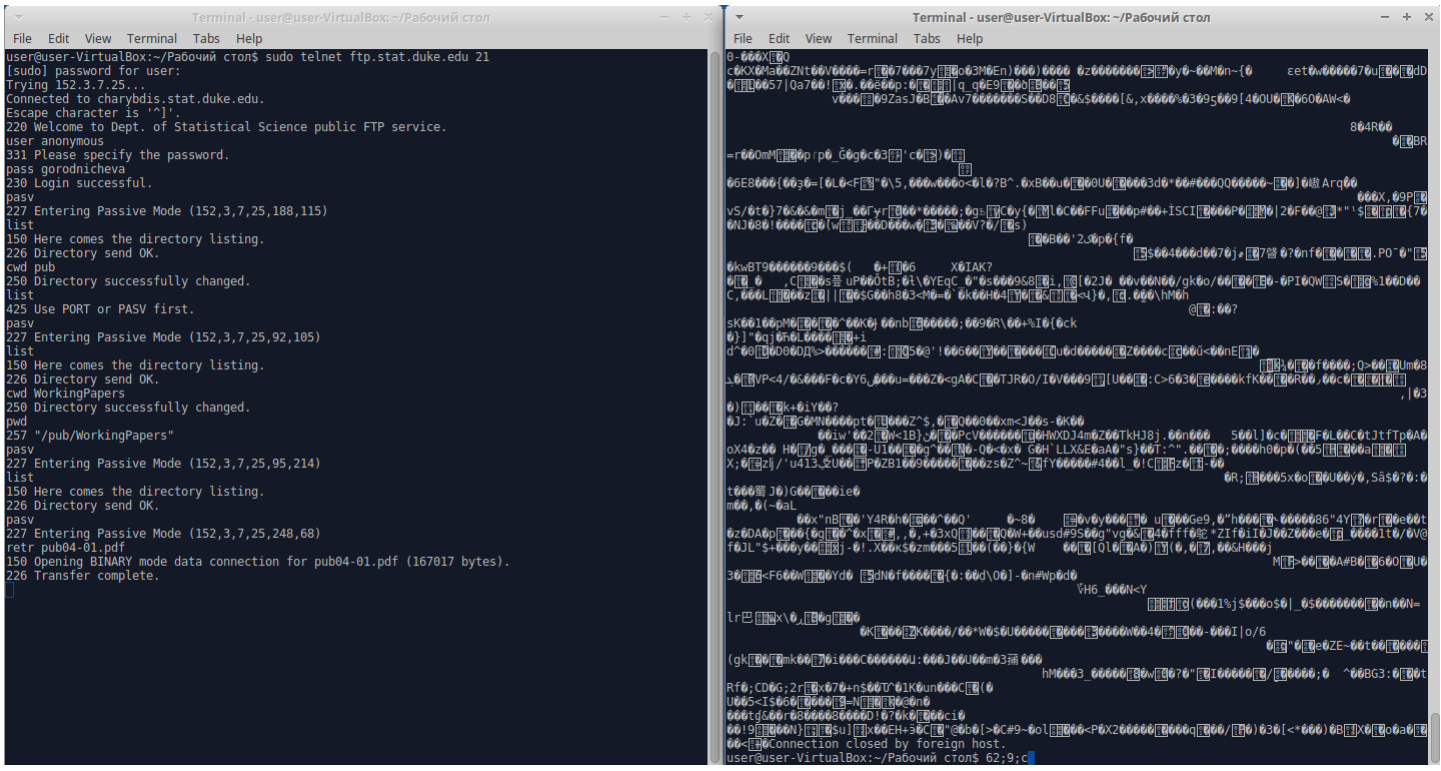

Рисунок 7.3. Скачивание файла с FTP-сервера

7.3. Подключение к SMTP-серверу и отправка письма

С помощью команды, указанной ниже, подключаемся к SMTP-серверу smtp.yandex.ru (рис. 7.4):

```
1 gnults-cli -p 465 smtp.yandex.ru
```

Рисунок 7.4. Подключение к SMTP-серверу

Данная команда обеспечивает защищенное TLS-подключение. Команды telnet/nc здесь не подходят по причине их работы только с нешифрованными соединениями.

SSL (Secure Sockets Layer) и TLS (Transport Level Security) - это криптографические протоколы, обеспечивающие защищенную передачу данных в компьютерной сети. Они активно используются при работе с электронной почтой. Соединение, защищенное протоколом TLS, обладает определенными свойствами: безопасностью, аутентификацией и целостностью.

Затем необходимо пройти аутентификацию:

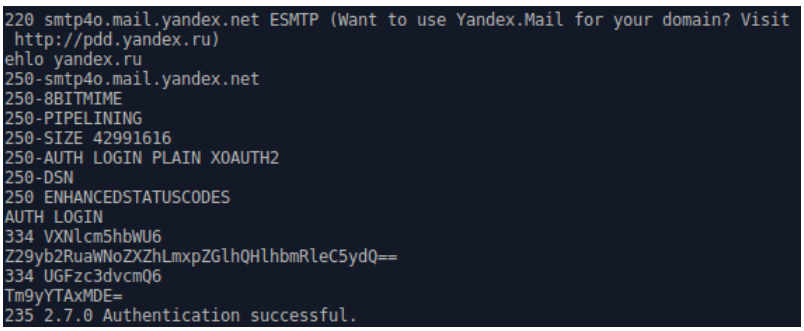

Рисунок 7.5. Аутентификация

Для аутентификации использовался BASE64, с помощью которого были закодированы логин и пароль (рис. 7.6):

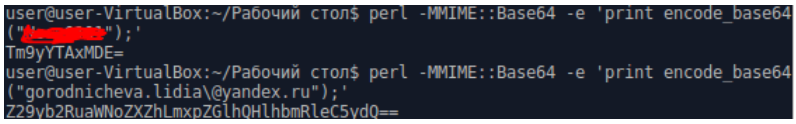

Рисунок 7.6. Кодирование логина и пароля

Используя команды MAIL FROM (от кого письмо), RCPT TO (кому письмо), DATA (собственно само письмо с указанием темы (Subject) и From (от кого), было отослано письмо с содержанием ‘Gorodnicheva Lidia” (рис. 7.7):

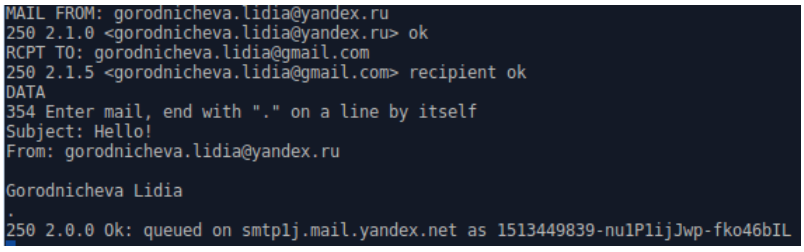

Рисунок 7.7. Посылка письма

7.4. Подключение к POP3-серверу, проверка почты и получение письма

Попробуем прочитать посланное письмо с помощью протокола POP3. Для начала подключимся к POP3-серверу (рис. 7.8):

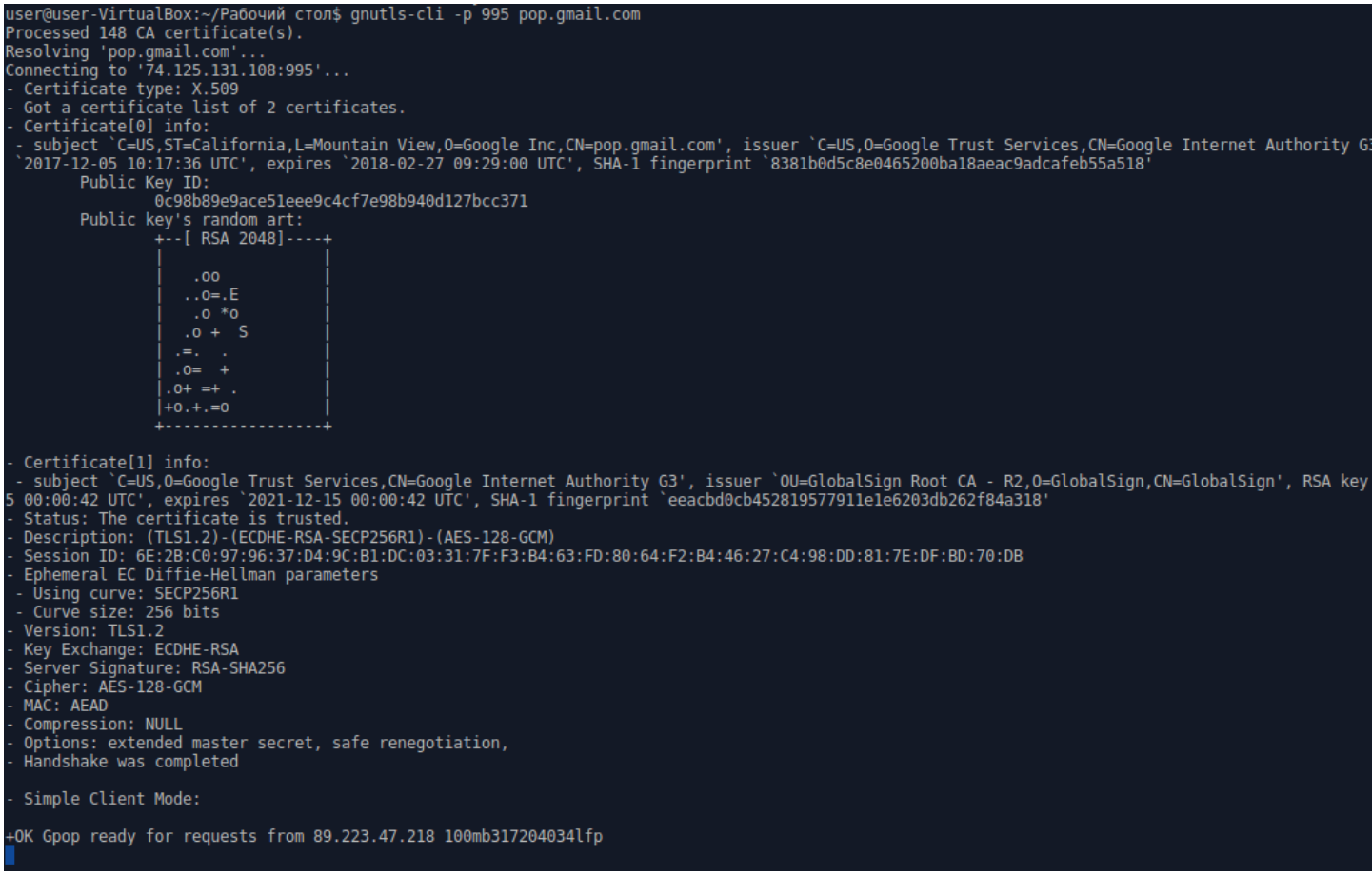

Рисунок 7.8. Подключение к POP3-серверу

Затем введем логин и пароль, используя команды USER и PASS (рис. 7.9):

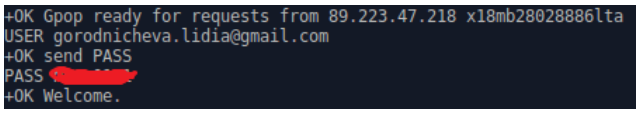

Рисунок 7.9. Аутентификация

С помощью команды LIST запрошен список всем писем (рис. 7.10), а команда RETR <номер письма> вывела полученное письмо (рис. 7.11):

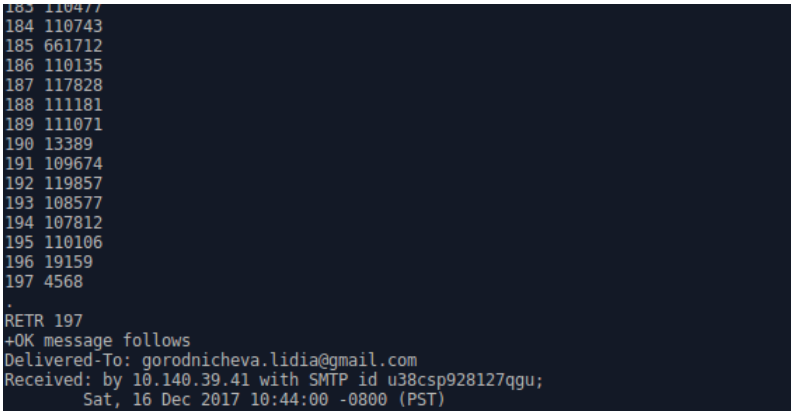

Рисунок 7.10. Список писем

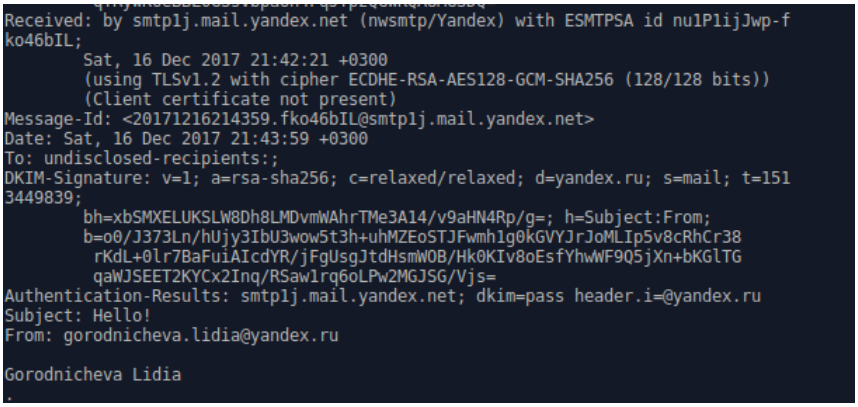

Рисунок 7.11. Полученное письмо

8. Выводы

В данной лабораторной работе изучены принципы программирования сокетов на основе TCP и UDP. Познакомились с архитектурой разработки многопоточных сетевых приложений: использование механизма нитей и средств синхронизации. Изучение происходило через разработку простейших клиент-серверных приложений на ОС Linux и Windows, а также реализацию собственного протокола на основе TCP и UDP для индивидуального задания (а именно дистанционного тестирования). Чтобы не усложнять задание и не создавать отдельную базу данных для пользователей и их результатов, были созданы отдельные текстовые файлы, в которых эти пользователи хранятся. Для вопросов и вариантов ответов также есть свои файлы. Архитектура TCP приложения была построена на основе 3 различных потоков для 3 разных видов блокирующих функций (read, accept, getline) , обеспечивая тем самым параллельную работу всей системы. Минус такой архитектуры в том, что для каждого клиента выделяется отдельный поток, что получается

весьма ресурсоемко. Решить такую проблему можно, например, при помощи пула сокетов (или портов завершения, которые имеются в Windows). Передача сообщений по TCP тоже реализована довольно просто: мы передавали определенное количество байт с каждой посылкой, а завершали сообщение меткой.

Для передачи сообщений длиной больше, чем длина, заданная в протоколе, была создана специальная функция, которая делит сообщение на куски заданной длины и передавает их поочередно.

При разработке UDP столкнулись с тем, что соединения не создается. Чтобы сохранить простую архитектуру TCP сервера, при получении сообщения от клиента создавали новый сокет. Так было сымитировано отдельное соединение. Чтобы понимать, что все посылки доставляются верно или неверно, инкапсулировали номер посылки в сообщение. Если номер не совпадал с ожидаемым, просто выдавали ошибку. Здесь для сохранения длины посылки полную длину посылки оставили той же, однако количество байт данных в ней каждый раз меняется, в зависимости от длины номера посылки. На приеме просто отделяем номер посылки.

Также были получены навыки работы с такими прикладными протоколами, как HTTP, FTP, SMTP и POP3. Эти навыки могут оказаться полезными при разработке серьезных сетевых приложений. Некоторые задачи данной лабораторной очень схожи с задачами, решаемыми этими протоколами. Так SMTP ожидает ответа на каждую операцию и присылает обратно код - выполнена ли операция или завершилась с какой-либо ошибкой. В нашем задании незарегистрированный и незалогиненый пользователь не может получить и пройти тест. В POP3, SMTP, а так же FTP перед тем, как выполнить какую-либо операцию, необходимо было произвести аутентификацию.

9. Приложение

Листинг 7: TCP сервер на Linux

```
1  #include <stdio.h>
2  #include <stdlib.h>
4
4  #include <netdb.h>
5  #include <netinet/in.h>
6  #include <unistd.h>
8
8  #include <string.h>
10
10 int readn(int sockfd, char *buf, int n){
11     int k;
12     int off = 0;
13     for(int i = 0; i < n; ++i){
```

```
14          k = read(sockfd, buf + off, 1);
15          off += 1;
16          if (k < 0){
17              printf("Error. reading from socket \n");
18              exit(1);
19          }
20      }
21      return off;
22  }
24  int main(int argc, char *argv[]) {
25      int sockfd, newsockfd;
26      uint16_t portno;
27      unsigned int clilen;
28      char buffer[256];
29      char *p = buffer;
30      struct sockaddr_in serv_addr, cli_addr;
31      ssize_t n;
33      sockfd = socket(AF_INET, SOCK_STREAM, 0);
35      if (sockfd < 0) {
36          perror("ERROR opening socket");
37          exit(1);
38      }
40      bzero((char *) &serv_addr, sizeof(serv_addr));
41      portno = 5001;
43      serv_addr.sin_family = AF_INET;
44      serv_addr.sin_addr.s_addr = INADDR_ANY;
45      serv_addr.sin_port = htons(portno);
47      if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
48          perror("ERROR on binding");
49          exit(1);
50      }
52      listen(sockfd, 5);
53      clilen = sizeof(cli_addr);
55      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
57      shutdown(sockfd, 2);
58      close(sockfd);
60      if (newsockfd < 0) {
```

```
61          perror("ERROR on accept");
62          exit(1);
63      }
66
66
66      bzero(buffer, 256);
68
68      n = readn(newsockfd, p, 255);
70
70      printf("Here is the message: %s\n", buffer);
72
72      n = write(newsockfd, "I got your message", 18);
74
74      if (n < 0) {
75          perror("ERROR writing to socket");
76          exit(1);
77      }
79
79      shutdown(newsockfd, 2);
80      close(newsockfd);
82
82      return 0;
83  }
```

Листинг 8: TCP клиент на Linux

```
1  #include <stdio.h>
2  #include <stdlib.h>
4
4  #include <netdb.h>
5  #include <netinet/in.h>
6  #include <unistd.h>
8
8  #include <string.h>
10
10 int readn(int sockfd, char *buf, int n){
11     int k;
12     int off = 0;
13     for(int i = 0; i < n; ++i){
14         k = read(sockfd, buf + off, 1);
15         off += 1;
16         if (k < 0){
17             printf("Error reading from socket \n");
18             exit(1);
19         }
20     }
21     return off;
```

```
22 }
24
24 int main(int argc, char *argv[]) {
25     int sockfd, n;
26     uint16_t portno;
27     struct sockaddr_in serv_addr;
28     struct hostent *server;
30
30     char buffer[256];
31     char *p = buffer;
33
33     if (argc < 3) {
34         fprintf(stderr, "usage %s hostname port\n", argv[0]);
35         exit(0);
36     }
38
38     portno = (uint16_t) atoi(argv[2]);
40
40     sockfd = socket(AF_INET, SOCK_STREAM, 0);
42
42     if (sockfd < 0) {
43         perror("ERROR opening socket");
44         exit(1);
45     }
47
47     server = gethostbyname(argv[1]);
49
49     if (server == NULL) {
50         fprintf(stderr, "ERROR, no such host\n");
51         exit(0);
52     }
54
54     bzero((char *) &serv_addr, sizeof(serv_addr));
55     serv_addr.sin_family = AF_INET;
56     bcopy(server->h_addr, (char *) &serv_addr.sin_addr.s_addr, (size_t) server->
    ↪ h_length);
57     serv_addr.sin_port = htons(portno);
59
59     if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    ↪ {
60         perror("ERROR connecting");
61         exit(1);
62     }
65
65
65     printf("Please enter the message: ");
66     bzero(buffer, 256);
```

```
67      fgets(buffer, 255, stdin);
69
69      n = write(sockfd, buffer, 255);
71
71      if (n < 0) {
72          perror("ERROR writing to socket");
73          exit(1);
74      }
76
76      bzero(buffer, 256);
77      n = readn(sockfd, p, 255);
79
79      printf("%s\n", buffer);
81
81      shutdown(sockfd, 2);
82      close(sockfd);
84
84      return 0;
85  }
```

Листинг 9: UDP сервер на Windows

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <winsock2.h>
5 #include <stdint.h>
8
8
8 int main() {
10
10      WSADATA wsaData;
12
12      unsigned int t;
13      t = WSAStartup(MAKEWORD(2,2), &wsaData);
15
15      if (t != 0) {
16          printf("WSAStartup failed: %ui\n", t);
17          return 1;
18      }
20
20      int sockfd;
21      uint16_t portno;
22      int clilen;
23      char buffer[256];
24      struct sockaddr_in serv_addr, cli_addr;
26
```

```
26      /* First call to socket() function */
27      sockfd = socket(AF_INET, SOCK_DGRAM, 0);
29
29      if (sockfd < 0) {
30          perror("ERROR opening socket");
31          exit(1);
32      }
34
34      /* Initialize socket structure */
35      memset((char *) &serv_addr, 0, sizeof(serv_addr));
36      portno = 5001;
38
38      serv_addr.sin_family = AF_INET;
39      serv_addr.sin_addr.s_addr = INADDR_ANY;
40      serv_addr.sin_port = htons(portno);
42
42      /* Now bind the host address using bind() call.*/
43      if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
44          perror("ERROR on binding");
45          exit(1);
46      }
48
48      clilen = sizeof(cli_addr);
49      int rec = recvfrom(sockfd, buffer, 256, 0, (struct sockaddr *)&cli_addr, &
    ↪ clilen);
50      if (rec < 0) {
51        perror("ERROR");
52        exit(1);
53      }
55
55      struct hostent *hst;
56          hst = gethostbyaddr((char *)&cli_addr.sin_addr, 4, AF_INET);
57          printf("+%s [%s:%d] new DATAGRAM!\n", (hst) ? hst->h_name : "Unknown
    ↪ host", inet_ntoa(cli_addr.sin_addr), ntohs(cli_addr.sin_port));
59
59          printf("C=>S:%s\n", &buffer[0]);
61
61          sendto(sockfd, buffer, 256, 0, (struct sockaddr *)&cli_addr, sizeof(
    ↪ cli_addr));
62          close(sockfd);
63          WSACleanup();
65
65      return 0;
66  }
```

Листинг 10: UDP клиент на Windows

```c
#include <stdio.h>
#include <winsock2.h>
#include <stdint.h>


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

    WSADATA wsaData;

    unsigned int t;
    t = WSAStartup(MAKEWORD(2,2), &wsaData);

    if (t != 0) {
        printf("WSAStartup failed: %ui\n", t);
        return 1;
    }

    int sockfd, n;
    uint16_t portno;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];

    if (argc < 3) {
        fprintf(stderr, "usage %s hostname port\n", argv[0]);
        exit(0);
    }

    portno = (uint16_t) atoi(argv[2]);

    /* Create a socket point */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    if (sockfd < 0) {
        perror("ERROR opening socket");
        exit(1);
    }

    server = gethostbyname(argv[1]);

    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }

    memset((char *) &serv_addr, 0, sizeof(serv_addr));
```

```
48      serv_addr.sin_family = AF_INET;
49      memcpy((char *) &serv_addr.sin_addr.s_addr, server->h_addr, (size_t) server
        ↪ ->h_length);
50      serv_addr.sin_port = htons(portno);
52
52      printf("Please enter the message: ");
53      memset(buffer, 0, 256);
54      fgets(buffer, 255, stdin);
56
56      /* Send message to the server */
57      n = sendto(sockfd, buffer, 256, 0, (struct sockaddr *)&serv_addr, sizeof(
        ↪ serv_addr));
59
59      if (n < 0) {
60          perror("ERROR writing to socket");
61          exit(1);
62      }
64
64      /* Now read server response */
65      memset(buffer, 0, 256);
67
67      struct sockaddr_in serv_two;
68      int serv_two_size = sizeof(serv_two);
70
70      int rec = recvfrom(sockfd, buffer, 256, 0, (struct sockaddr *)&serv_two, &
        ↪ serv_two_size);
71      if (rec < 0) {
72          perror("ERROR");
73          exit(1);
74      }
76
76      printf("%s\n", buffer);
78
78      closesocket(sockfd);
79      WSACleanup();
81
81      return 0;
82  }
```

Листинг 11: Многопоточный TCP-сервер на Linux

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <netdb.h>
4 #include <netinet/in.h>
5 #include <unistd.h>
7
```

```
7 #include <string.h>
8 #include "pthread.h"
10
10 int readn(int sockfd, char *buf, int n){
11     int k;
12     int off = 0;
13     for(int i = 0; i < n; ++i){
14         k = read(sockfd, buf + off, 1);
15         off += 1;
16         if (k < 0){
17             printf("Error. reading from socket \n");
18             exit(1);
19         }
20     }
21     return off;
22 }
24
24 void* formulti (void* temp) {
25   int n = *((int*)temp);
26   char buffer[256];
27   char *p = buffer;
28     bzero(buffer, 256);
29     int k = readn (n, p, 255);
30     if (k > 0) {
31       printf("%s \n", buffer);
32       write(n, "I have got your message", 23);
33     }
34     shutdown(n, 2);
35     close(n);
36 }
40
40
40
40 int main() {
41     int sockfd, newsockfd;
42     uint16_t portno;
43     unsigned int clilen;
44     char buffer[256];
45     struct sockaddr_in serv_addr, cli_addr;
46     ssize_t n;
48
48     sockfd = socket(AF_INET, SOCK_STREAM, 0);
50
50     if (sockfd < 0) {
51         perror("ERROR opening socket");
52         exit(1);
53     }
```

```
55
55      bzero((char *) &serv_addr, sizeof(serv_addr));
56      portno = 5001;
58
58      serv_addr.sin_family = AF_INET;
59      serv_addr.sin_addr.s_addr = INADDR_ANY;
60      serv_addr.sin_port = htons(portno);
62
62      if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
63          perror("ERROR on binding");
64          exit(1);
65      }
67
67      while (1) {
68          listen(sockfd, 5);
69          clilen = sizeof(cli_addr);
71
71          newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
74
74
74          if (newsockfd < 0) {
75              perror("ERROR on accept");
76              exit(1);
77          }
80
80
80          pthread_t tid;
81          pthread_attr_t attr;
82          pthread_attr_init(&attr);
83          pthread_create(&tid, &attr, formulti, &newsockfd);
84          pthread_detach(tid);
85      }
87
87      return 0;
88  }
```