

Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и технологий  
Высшая школа интеллектуальных систем и суперкомпьютерных  
технологий

**Отчёт по лабораторным работам**  
**Дисциплина:** Технологии компьютерных сетей

**Работу выполнил:**

Д. В. Павлов

Группа:

3530901/70202

**Преподаватель:**

А. О. Алексюк

"\_\_" \_\_\_\_\_ 2020 г.

Санкт-Петербург  
2020

## Содержание

|                                    |    |
|------------------------------------|----|
| Лабораторная работа №1 . . . . .   | 1  |
| Формулировка задания . . . . .     | 1  |
| Теоретическая информация . . . . . | 2  |
| Программа работы . . . . .         | 3  |
| Тестирование . . . . .             | 9  |
| Лабораторная работа №2 . . . . .   | 10 |
| Формулировка задания . . . . .     | 10 |
| Программа работы . . . . .         | 11 |
| Тестирование . . . . .             | 18 |
| Лабораторная работа №3 . . . . .   | 19 |
| Формулировка задания . . . . .     | 19 |
| Теоретическая информация . . . . . | 20 |
| Программа работы . . . . .         | 21 |
| Тестирование . . . . .             | 38 |
| Лабораторная работа №4 . . . . .   | 39 |
| Формулировка задания . . . . .     | 39 |
| Программа работы . . . . .         | 40 |
| Тестирование . . . . .             | 48 |
| Заключение . . . . .               | 49 |

# Лабораторная работа №1

## Формулировка задания

Разработать чат

Общий канал, никнейм у каждого участника

Консольный UI (Пример сообщения: [Peter] Hello!)

Разработать протокол обмена, описание разместить в репозитории

Протестировать программы

## Теоретическая информация

Transmission Control Protocol (TCP) (протокол управления передачей) — один из основных сетевых протоколов Интернета, предназначенный для управления передачей данных в сетях и подсетях TCP/IP. Выполняет функции протокола транспортного уровня модели OSI. TCP — это транспортный механизм, предоставляющий поток данных, с предварительной установкой соединения, за счёт этого дающий уверенность в достоверности получаемых данных, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета

Передача потоковая — то есть с точки зрения клиент-сервера между ними есть «труба» в которую клиент закачивает данные, а сервер выкачивает. Данные хранятся в буфере, буфер есть передающий и принимающий. В ТСП есть особенность — данные кладутся в общий буфер и граница пакетов не существует (в юдп например когда посылаем пакет с помощью send, то ровно такую же порцию данных получает принимающая сторона выполнив receive). То есть если посылаем небольшие порции send это просто накапливает буфер, а потом все одним сегментов все улетает в сеть. Важно: нет деления на пакеты. Если это все же требуется — то это необходимо сделать на прикладном уровне

Как осуществляется передача? Нумеруем каждый передаваемый байт (уникальный номер) Сегмент — набор байтов и ему присваивается номер первого байта. Сегмент посылается в сеть и параллельно с этим копируется в буфер повторной передачи, затем включается таймаут и если мы вовремя получаем подтверждение о успешной, что сегмент передан успешно, то данные из буфера повторной передачи удаляются. Если таймаут прошел, то данные повторно передаются из буфера. То мы повышаем вероятность доставки.

Если пришло подтверждение, что правильно принят сегмент, то считаем, что приняты все его байты. Существуют задержки между сегментами (время, когда мы ничего не передавали в сеть тк не были уверены, что предыдущие данные были переданы успешно) в результате имеем проблемы с тем что имеются простои по времени. Когда походит квитанция то в ней приходит номер следующего ожидаемого байта! (например, первый сегмент 0-1023 был получен успешно, то в квитанции будет 1024).

## Программа работы

Протокол следующий:

В данном приложении используется способ передачи с указанием длины сообщения для обеспечения механизма выявления сообщений из потока байт. Для начала работы с сервером требуется лишь его запустить. Далее в нем будут отображаться новые подключения, получения сообщений, отключения пользователей. Время сервера – по Гринвичу. Для начала работы с клиентом после его запуска пользователю требуется ввести свое имя. После можно отсылать сообщения. Полученные сообщения отображаются в формате <HH:MM> [name]: message. Время клиента – локальное. Для завершения работы нужно отправить команду !exit.

В листингах сервера и клиента подробно прокомментированы детали реализации.

**Начнем с сервера:**

```
import socket
import threading
import time

CODE = 'utf-8'
HEADER_LEN = 16
IP = 'localhost'
PORT = 5001

# Лист подключенных clients - socket это ключ, user header и name это данные
clients_list = {}

# Для чтения данных используется функция recv,
# которой первым параметром нужно передать количество получаемых
# байт данных. Если столько байт, сколько указано, не пришло,
# а какие-то данные уже появились, она всё равно
# возвращает всё, что имеется, поэтому надо контролировать размер полученных данных.
# Тип возвращаемых данных — bytes. У этого типа есть почти все методы, что и у строк,
# но для того, чтобы использовать
# из него текстовые данные с другими строками
# (складывать, например, или искать строку в данных, или печатать),
# придётся декодировать данные (или их часть, если вы обработали байты и выделили строку)
# и использовать уже полученную строку.
```

```

def receive(socket):
    while True:
        try:
            # Получаем наш header, содержащий длину сообщения, размер константный
            msg_header = socket.recv(HEADER_LEN)
            # Если мы не получили данных, клиент корректно закрыл соединение
            #(socket.close () или socket.SHUT_RDWR)
            if not msg_header:
                return False
            # Метод strip() возвращает копию строки,
            в которой все символы были удалены с начала и конца (пробелы)
            msg_len = int(msg_header.decode(CODE).strip())
            # Возвращаем объект заголовка сообщения и данных сообщения
            return {"header": msg_header, "data": socket.recv(msg_len)}
        except:
            # Если мы здесь, клиент резко закрыл соединение, например, нажав ctrl + c
            return False

def server():
    # TCP почти всегда использует SOCK_STREAM, а UDP использует SOCK_DGRAM.
    # TCP (SOCK_STREAM) - это протокол, основанный на соединении.
    Соединение установлено, и обе стороны ведут
    # разговор идет пока соединение не будет прервано одной из сторон или сетевой ошибкой.
    # UDP (SOCK_DGRAM) - это протокол на основе дейтаграмм.
    Вы отправляете одну дейтаграмму и получаете один ответ,
    # а затем соединение разрывается.
    # socket.AF_INET — IPv4 usage

    serv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Решение проблемы с Address already in use, вроде не возникало, но добавил
    serv_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    serv_sock.bind((IP, PORT))

    # С помощью метода listen мы запустим для данного сокета режим прослушивания.
    # Метод принимает один аргумент — максимальное количество подключений в очереди.
    serv_sock.listen()

```

```
print("Server was started!")
```

```
while True:
```

```
    # мы можем принять подключение с помощью метода accept,
    который возвращает кортеж с двумя элементами:
```

```
    # новый сокет и адрес клиента. Именно этот сокет и будет
    использоваться для приема и отправки клиенту данных.
```

```
    client_socket, client_data = serv_sock.accept()
```

```
    client = receive(client_socket)
```

```
    if client:
```

```
        # Сохраняем имя пользователя и его заголовок
```

```
        clients_list[client_socket] = client
```

```
        print(f"Connection from {client_data[0]}:{client_data[1]} ;
```

```
        Nickname: {client['data'].decode(CODE)}")
```

```
        threading.Thread(target=handler, args=(client_socket, client,)).start()
```

```
def handler(socket, client):
```

```
    while True:
```

```
        message = receive(socket)
```

```
        local_time = str(int(time.time())).encode(CODE)
```

```
        l_time_header = f"{len(local_time):<{HEADER_LEN}}".encode(CODE)
```

```
        sender_time = {"header": l_time_header, "data": local_time}
```

```
        if not message or message['data'].decode(CODE) == "!exit":
```

```
            # Клиент отключился, удаляем его
```

```
            try:
```

```
                print(f"Connection was closed by {clients_list[socket]['data'].decode(CODE)}")
```

```
                del clients_list[socket]
```

```
                socket.shutdown(socket.SHUT_RDWR)
```

```
                socket.close()
```

```
                continue
```

```
            except:
```

```
                continue
```

```
server_time = time.strftime("%H:%M", time.gmtime())
```

```
print(
```

```
    f"Received message at {server_time} from {client['data'].decode(CODE)}:
```

```
    {message['data'].decode(CODE)}")
```

```
for client_sock in clients_list:
```

```

if client_sock != socket:
    # Мы повторно используем здесь заголовок сообщения,
    # отправленный отправителем, и сохраненный
    # Заголовок имени пользователя, отправленный пользователем при подключении
    client_sock.send(
        client['header'] + client['data'] + message['header'] + message['data'] + sender_time[
            'header'] + sender_time['data'])
server()

```

## Листинг 1 — Сервер

### Теперь клиент:

```

import socket
import threading
import time
import sys

CODE = 'utf-8'
HEADER_LEN = 16
IP = 'localhost'
PORT = 5001

def client():
    # Нам нужно кодировать имя пользователя в байтах
    nickname = input("Enter your nickname: ").encode(CODE)
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((IP, PORT))
    # Подготавливаем заголовок фиксированного размера,
    # который мы также кодируем в байтах
    nick_header = f"{len(nickname):<{HEADER_LEN}}".encode(CODE)
    client_socket.send(nick_header + nickname)
    threading.Thread(target=read, args=(client_socket,)).start()
    threading.Thread(target=send, args=(client_socket,)).start()

# receive обрабатывает части сообщения (ник, само сообщения, время отправки)
def receive(socket):

```



```

while True:
    # Теперь мы хотим перебрать полученные сообщения
    # (их может быть больше одного) и вывести их
    # Получим заголовок, содержащий длину имени пользователя (размер константный)
    header = socket.recv(HEADER_LEN)
    # print(header)
    # Если мы не получили данных, сервер корректно закрыл соединение
    # (socket.close () или socket.SHUT_RDWR)
    if not len(header):
        print("Connection was closed by the server")
        sys.exit()
    part_len = int(header.decode(CODE).strip())
    buf = b""
    while True:
        lbuf = socket.recv(part_len - len(buf))
        buf = buf + lbuf
        if not lbuf:
            break
    return buf.decode(CODE)

def read(socket):
    while True:
        try:
            nickname = receive(socket)
            message = receive(socket)
            msg_time = receive(socket)
            # производим учет времени отправителя и получателя
            user_time = time.strftime("%H:%M", (time.localtime(int(msg_time))))
            print(f'<{user_time}> [{nickname}]: {message}')

        except Exception as e:
            print('Error', str(e))
            sys.exit()

def send(socket):
    while True:
        try:
            message = input()
            # выход из чата после сообщения !exit
            # if message == "!exit":
            #     exit(0)

```

```

#     socket.shutdown(socket.SHUT_RDWR)
#     socket.close()
#     return

if message:
# Закодировать сообщение в байты. Подготовить заголовок и преобразовать в байты,
# как для имени пользователя ранее, затем отправить
    message1 = message
    message = message.encode(CODE)
    msg_header = f"{len(message):<{HEADER_LEN}}".encode(CODE)
    socket.send(msg_header + message)
    if message1 == "!exit":
        socket.shutdown(socket.SHUT_RDWR)
        socket.close()
        exit(0)
        return
except EOFError as e:
    print("Incorrect input")
    continue
except:
    exit(0)
    return
client()

```

Листинг 2 — Клиент

## Тестирование

Основная проблема была связана с передачей сообщений большой длины.

Были протестированы следующие моменты:

- 1) Сервер и клиент корректно завершаются - не зависают в процессе завершения, клиент получает и обрабатывает информацию о закрытии соединения
- 2) Сервер и клиент позволяют корректно отправлять большие сообщения. Для теста использовались случайные книги на `lib.ru` и их текст отправлялся в чат.
- 3) Клиенты работают независимо и при подключении не обрывают соединение друг для друга.

## **Лабораторная работа №2**

### **Формулировка задания**

Переписать задание 1 с использованием roll

Обязательно переписать сервер, желательно переписать клиент

## Программа работы

Протокол следующий:

В данном приложении используется способ передачи с указанием длины сообщения для обеспечения механизма выявления сообщений из потока байт. Для начала работы с сервером требуется лишь его запустить. Далее в нем будут отображаться новые подключения, получения сообщений, отключения пользователей. Время сервера – по Гринвичу. Для начала работы с клиентом после его запуска пользователю требуется ввести свое имя. После можно отсылать сообщения. Полученные сообщения отображаются в формате <HH:MM> [name]: message. Время клиента – локальное. Для завершения работы нужно отправить команду !exit.

В листингах сервера и клиента подробно прокомментированы детали реализации.

**Начнем с сервера:**

```
import socket
from select import select

CODE = 'utf-8'
HEADER_LEN = 10
IP = 'localhost'
PORT = 5001
# Лист подключенных clients - socket это ключ, user header и name это данные
clients_list = {}
sockets_list = []

# Для чтения данных используется функция recv,
# которой первым параметром нужно передать количество получаемых
# байт данных. Если столько байт, сколько указано, не пришло,
# а какие-то данные уже появились, она всё равно
# возвращает всё, что имеется, поэтому надо контролировать размер полученных данных.
# Тип возвращаемых данных — bytes. У этого типа есть почти все методы, что и у строк,
# но для того, чтобы использовать
# из него текстовые данные с другими строками
# (складывать, например, или искать строку в данных, или печатать),
# придётся декодировать данные (или их часть, если вы обработали байты и выделили строку)
# и использовать уже полученную строку.
```

```

def receive(socket):
    try:
        # Получаем наш header, содержащий длину сообщения, размер константный
        msg_header = socket.recv(HEADER_LEN)
        # Если мы не получили данных, клиент корректно закрыл соединение
        (socket.close () или socket.SHUT_RDWR)
        if not msg_header:
            return False
        # Метод strip() возвращает копию строки, в которой
        все символы были удалены с начала и конца строки (пробелы)
        msg_len = int(msg_header.decode(CODE).strip())
        # Возвращаем объект заголовка сообщения и данных сообщения
        return {"header": msg_header, "data": socket.recv(msg_len)}
    except:
        # Если мы здесь, клиент резко закрыл соединение, например, нажав ctrl + c
        return False

```

```

def new_client(socket):
    # мы можем принять подключение с помощью метода ассерт,
    который возвращает кортеж с двумя элементами:
    # новый сокет и адрес клиента. Именно этот сокет и будет использоваться
    для приема и отправки клиенту данных.
    client_socket, client_data = socket.accept()
    client = receive(client_socket)
    # Добавить принятый сокет в список select()
    sockets_list.append(client_socket)
    # Сохраняем имя пользователя и его заголовок
    clients_list[client_socket] = client
    print(f"Connection from {client_data[0]}:{client_data[1]} ;
    Nickname: {client['data'].decode(CODE)}")

```

```

def server2():
    # TCP почти всегда использует SOCK_STREAM, а UDP использует SOCK_DGRAM.
    # TCP (SOCK_STREAM) - это протокол, основанный на соединении.
    Соединение установлено, и обе стороны ведут

```

```

# разговор идет пока соединение не будет прервано одной из сторон или сетевой ошибкой.
# UDP (SOCK_DGRAM) - это протокол на основе дейтаграмм.
Вы отправляете одну дейтаграмму и получаете один ответ,
# а затем соединение разрывается.
# socket.AF_INET — IPv4 usage

serv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Решение проблемы с Address already in use, вроде не возникало, но добавил
serv_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
serv_sock.bind((IP, PORT))

# С помощью метода listen мы запустим для данного сокета режим прослушивания.
# Метод принимает один аргумент — максимальное количество подключений в очереди.
serv_sock.listen()
sockets_list.append(serv_sock)
print("Server was started!")

# Функция select() даёт нам возможность одновременной проверки нескольких сокетов,
чтобы увидеть, если у них данные,
# ожидающие recv() или можете ли вы send() данные в сокет без блокирования.
Данная функция работает в режиме
# блокировки, пока либо не произойдут события,
связанные с появлением возможности чтения или записи в сокеты,
# либо не истечет время тайм аута, задаваемое для этого вызова.
Аргументы функции select() имеют следующий смысл:
# fd_set *readfds, *writefds, *exceptfds - указатели на наборы дескрипторов сокетов,
# предназначенных для операций чтения, записи и исключительных ситуаций

while True:
    sockets, _, exceptions = select(sockets_list, [], sockets_list)
    for sock in sockets:
        if sock == serv_sock:
            new_client(sock)
        elif message := receive(sock):
            # Клиент отключился, удаляем его
            if message['data'].decode(CODE) == "!exit":
                try:
                    print(f"Connection was closed by {clients_list[sock]['data'].decode(CODE)}")
                    sockets_list.remove(sock)

```

```

        del clients_list[sock]
        sock.shutdown(sock.SHUT_RDWR)
        sock.close()
        continue
    except:
        continue

    # Получить пользователя по уведомленному сокету, чтобы мы знали,
    # кто отправил сообщение
    client = clients_list[sock]
    for client_sock in clients_list:
        if client_sock != sock:
            # Мы повторно используем здесь заголовок сообщения,
            # отправленный отправителем, и сохраненный
            # Заголовок имени пользователя, отправленный пользователем при подключении
            print(f"Received message from {client['data'].decode(CODE)}:
                  {message['data'].decode(CODE)}")
            client_sock.send(client['header'] + client['data'] + message['header'] + message['data'])
    elif message is False:
        try:
            print(f"Connection was closed by {clients_list[sock]['data'].decode(CODE)}")
            sockets_list.remove(sock)
            del clients_list[sock]
            sock.shutdown(sock.SHUT_RDWR)
            sock.close()
            continue
        except:
            continue

    # Обработка некоторых исключений сокетов
    for sock in exceptions:
        sockets_list.remove(sock)
        del clients_list[sock]

server2()

```

### Листинг 3 — Сервер

**Теперь клиент:**

```
import socket
```



```

import threading
import time
import sys

CODE = 'utf-8'
HEADER_LEN = 16
IP = 'localhost'
PORT = 5001

def client():
    # Нам нужно кодировать имя пользователя в байтах
    nickname = input("Enter your nickname: ").encode(CODE)
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((IP, PORT))
    # Подготавливаем заголовок фиксированного размера,
    который мы также кодируем в байтах
    nick_header = f"{len(nickname):<{HEADER_LEN}}".encode(CODE)
    client_socket.send(nick_header + nickname)
    threading.Thread(target=read, args=(client_socket,)).start()
    threading.Thread(target=send, args=(client_socket,)).start()

# receive обрабатывает части сообщения (ник, само сообщения, время отправки)
def receive(socket):
    while True:
        # Теперь мы хотим перебрать полученные сообщения (их может быть больше одного)
        # и вывести их
        # Получим заголовок, содержащий длину имени пользователя (размер константный)
        header = socket.recv(HEADER_LEN)
        # print(header)
        # Если мы не получили данных, сервер корректно закрыл соединение
        # (socket.close () или socket.SHUT_RDWR)
        if not len(header):
            print("Connection was closed by the server")
            sys.exit()
        part_len = int(header.decode(CODE).strip())
        buf = b""
        while True:

```

```

lbuf = socket.recv(part_len - len(buf))
buf = buf + lbuf
if not lbuf:
    break
return buf.decode(CODE)

```

```

def read(socket):
    while True:
        try:
            nickname = receive(socket)
            message = receive(socket)
            msg_time = receive(socket)
            # производим учет времени отправителя и получателя
            user_time = time.strftime("%H:%M", (time.localtime(int(msg_time))))
            print(f'<{user_time}> [{nickname}]: {message}')

        except Exception as e:
            print('Error', str(e))
            sys.exit()

```

```

def send(socket):
    while True:
        try:
            message = input()
            # выход из чата после сообщения !exit
            # if message == "!exit":
            #     exit(0)
            #     socket.shutdown(socket.SHUT_RDWR)
            #     socket.close()
            #     return

            if message:
                # Закодировать сообщение в байты. Подготовить заголовок и преобразовать в байты,
                # как для имени пользователя ранее, затем отправить
                message1 = message
                message = message.encode(CODE)
                msg_header = f"{len(message):<{HEADER_LEN}}".encode(CODE)

```

```

        socket.send(msg_header + message)
    if message1 == "!exit":
        socket.shutdown(socket.SHUT_RDWR)
        socket.close()
        exit(0)
    return
except EOFError as e:
    print("Incorrect input")
    continue
except:
    exit(0)
    return
client()

```

Листинг 4 — **Клиент**

## Тестирование

Основная проблема была связана с передачей сообщений большой длины.

Были протестированы следующие моменты:

- 1) Сервер и клиент корректно завершаются - не зависают в процессе завершения, клиент получает и обрабатывает информацию о закрытии соединения
- 2) Сервер и клиент позволяют корректно отправлять большие сообщения. Для теста использовались случайные книги на `lib.ru` и их текст отправлялся в чат.
- 3) Клиенты работают независимо и при подключении не обрывают соединение друг для друга.

## **Лабораторная работа №3**

### **Формулировка задания**

Разработать собственную реализацию (Trivial File Transfer Protocol)

## Теоретическая информация

UDP (англ. User Datagram Protocol — протокол пользовательских дейтаграмм) — это транспортный протокол для передачи данных в сетях IP без установления соединения. Он является одним из самых простых протоколов транспортного уровня модели OSI. Его IP-идентификатор — 0x11. В отличие от TCP, UDP не подтверждает доставку данных, не заботится о корректном порядке доставки и не делает повторов. Поэтому аббревиатуру UDP иногда расшифровывают как Unreliable Datagram Protocol (протокол ненадёжных датаграмм). Зато отсутствие соединения, дополнительного трафика и возможность широковещательных рассылок делают его удобным для применений, где малы потери. Не имеет специальных средств подтверждения надежности доставки и надежность гарантируется только с помощью контрольной суммы, поэтому протокол относительно не надёжный.

TFTP — это очень удобный и простой протокол, применяемый для копирования файлов с одного устройства, поддерживающего протоколы TCP/IP, на другое. В протоколе TFTP в качестве транспортного протокола используется UDP, а это означает, что на транспортном уровне TFTP является ненадежным. Но для обеспечения надежности в протоколе TFTP применяется собственная система подтверждения. При передаче файла с помощью TFTP этот файл разбивается на блоки по 512 байтов и передается в виде блоков. Таймауты в TFTP работают достаточно просто. Мы отправляем пакет с данными и ждем подтверждения (acknowledgement). Если ack приходит за указанное время (например, 5 секунд), то мы отправляем следующий блок данных, если не приходит, то мы заново отправляем последний блок. В отличие от TCP, где есть плавающее окно переменного размера, в TFTP неподтвержденным может быть только один пакет.

Алгоритм, следующий: 1) Поток, работающий на отправку, отправляет данные в сеть 2) Поток, работающий на приём, получает ack и кладёт информацию о нём в очередь (например, номер блока или даже просто сам факт получения ack) 3) Поток, работающий на отправку, достаёт информацию из очереди. На случай, если ack не придёт или придёт не сразу, устанавливается таймаут. 4) Если попытка достать элемент не удалась из-за превышения таймаута, то возвращаемся к п. 1, если всё получилось, то готовим новую порцию данных и повторяем последовательность действий. 5) Если после нескольких попыток отправить пакет подтверждение так и не пришло, то процесс передачи можно остановить.

## Программа работы

Протокол следующий:

Для начала работы с сервером требуется лишь его запустить. Для начала работы с клиентом после его запуска пользователю требуется последовательно ввести айпи адрес сервера, команду (PUT or GET), и имя файла: 127.0.0.1 PUT iii.txt. После будет производится передача. Для завершения работы нужно отправить команду !exit.

В листингах сервера и клиента подробно прокомментированы детали реализации.

**Начнем с сервера:**

```
Server started at ('127.0.0.1', 69)
Waiting for connection...
Connected to ('127.0.0.1', 64956)
Received packet from ('127.0.0.1', 64956)
packet data: b'\x00\x02iii.txt\x00octet\x00'
Received packet from ('127.0.0.1', 64956)
packet data: b'\x00\x03\x00\x01127.0.0.1 PUT
TRANSMISSION ENDED
file path on server: iii.txt
125 bytes transmitted
Waiting for connection...
Connected to ('127.0.0.1', 64959)
Received packet from ('127.0.0.1', 64959)
packet data: b'\x00\x02iii.txt\x00octet\x00'
File already exists
TRANSMISSION ENDED
file path on server: iii.txt
0 bytes transmitted
Waiting for connection...
```

Рисунок 0.1 — Принимаем небольшой файл от клиента

```

Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rK\x00\x01\x00\x00\x00C\x00\x00\x00\x01\x00\x00\x00H\x00\x00\x00\x01\x00\x00\x00F\x00\x00\x00\x01\x00
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rL\x00\x03\x00\x00\x00F\x00\x00\x00\x02\x00\x00\x00G\x00\x00\x00\x01\x00\x00\x00D\x00\x00\x00\x01\x00
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rM\x00\x01\x00\x00\x00D\x00\x00\x00\x01\x00\x00\x00F\x00\x00\x00\x01\x00\x00\x00H\x00\x00\x00\x01\x00
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rN\x00\x01\x00\x00\x00F\x00\x00\x00\x01\x00\x00\x00H\x00\x00\x00\x01\x00\x00\x00F\x00\x00\x00\x01\x00
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rO\x0c\xce\x00\x00\t\xfe\x00\x00\x08\x05\x00\x00\x08\x86\x00\x00\r\xbd\x00\x00\x13\xa1\x00\x00\n^\x00
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rP\r\xeb\x00\x00\x06\xd3\x00\x00\t\r\x00\x00\x0fq\x00\x00\x0b\x95\x00\x00\x11\xe3\x00\x00\x12\x06\x00
Received packet from ('127.0.0.1', 50824)
packet data: b''\x00\x03\rQ\x0c\x80\x00\x00\tq\x00\x00\x12*\x00\x00\x13\x10\x00\x00\x10\x8f\x00\x00\tT\x00\x00\t\xbb\x00\x00\
Received packet from ('127.0.0.1', 50824)
packet data: b''\x00\x03\rR\x0e\xc3\x00\x00\r\x07\x00\x00\x0e\x01\x00\x00\x0e\xed\x00\x00\x0c\xfc\x00\x00\x0c\xfc\x00\x00\x0c
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rS\xbb6\x00\x02\xca[\x00\x02\xda\x9d\x00\x02\xe9\x16\x00\x02\xf6d\x00\x03\x00\xe2\x00\x03\n'\x00\x03\
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rT\x0f4\x00\t\x86\xa9\x00\t\x94\xa8\x00\t\x9e\xcb\x00\t\xac\xb6\x00\t\xb4\t\x00\t\xbd\x96\x00\t\xcd\
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rU-\x15\x00\x10:"\x00\x10K\x00\x00\x10U'\x00\x10b'\x00\x10l\x18\x00\x10~\xc2\x00\x10\x92R\x00\x10\
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rV\xea\x83\x00\x16\xf3\xa2\x00\x17\x02b\x00\x17\x0f\xf9\x00\x17\x1f<\x00\x17,\xc3\x00\x17;D\x00\x17J\
Received packet from ('127.0.0.1', 50824)
packet data: b'\x00\x03\rW\x00\x00\x00\x00\x00\x00\x00\x00\x18\xff\xff\x00\x00\x00lesds\x00\x00\x00\x00\x03^\x00\x00\x00\
TRANSMISSION ENDED
file path on server: zhenya.mp4
1748088 bytes transmitted
Waiting for connection...

```

Рисунок 0.2 — Принимаем объемный файл от клиента

```

Waiting for connection...
Connected to ('127.0.0.1', 65249)
Received packet from ('127.0.0.1', 65249)
packet data: b'\x00\x01test.txt\x00octet\x00'
Received packet from ('127.0.0.1', 65249)
packet data: b'\x00\x04\x00\x01'
Received packet from ('127.0.0.1', 65249)
packet data: b'\x00\x04\x00\x02'
Received packet from ('127.0.0.1', 65249)
packet data: b'\x00\x04\x00\x03'
Received packet from ('127.0.0.1', 65249)
packet data: b'\x00\x04\x00\x04'
file path on server: test.txt
2046 bytes transmitted
Waiting for connection...

```

Рисунок 0.3 — Отсылаем файл клиенту

```

import os
import enum

```



```
import socket
import struct
import time
```

```
class TftpProtocol(object):
```

```
    """
```

```
    получаем udp пакет
```

```
    отсылаем пакет, который нужно записать в сокет
```

```
    Вход и выход - ТОЛЬКО байтовые массивы.
```

```
    Выходные пакеты сохраняются в буфере в этом классе,
```

```
    функция get_next_output_packet возвращает следующий пакет,
```

```
    который нужно отправить.
```

```
    Этот класс также отвечает за чтение/запись файлов на диск.
```

```
    Несоблюдение этих требований приведет к ошибке.
```

```
    """
```

```
class TftpPacketType(enum.Enum):
```

```
    RRQ = 1
```

```
    WRQ = 2
```

```
    DATA = 3
```

```
    ACK = 4
```

```
    ERROR = 5
```

```
def __init__(self):
```

```
    self.client_port = 0
```

```
    self.file_path = ''
```

```
    self.client_address = None
```

```
    self.file_block_count = 0
```

```
    self.fail = False
```

```
    self.sent_last = False
```

```
    self.ignore_current_packet = False # игнорируем пакет если у него другой порт
```

```
    self.tftp_mode = 'octet' # default mode
```

```
    self.request_mode = None # 'RRQ' или 'WRQ'
```

```
    self.fileBytes = []
```

```
    self.reached_end = False
```

```
    self.packet_buffer = []
```

```
    self.err = 0
```

```

def process_udp_packet(self, packetData, packetSource):
    """
    packet data - данные в bytearray
    packet source - информация об адресе отправителя
    """
    print(f"Received packet from {packetSource}")
    print('packet data:', packetData)
    self.ignore_current_packet = False
    if self.ignore_current_packet: # не добавляем текущий пакет в буффер
        return
    outputPacket = self.handle_received_packet(packetData)
    if outputPacket == []: # последний пакет в файле ACK
        return
    self.packet_buffer.append(outputPacket)

def generate_error_packet(self, error_code, error_message=''):
    # пакет ошибки в формате 2 байт, opcode 5. 2 байта под error code, error_msg
    error_packet = struct.pack('!HH', TftpProtocol.TftpPacketType.ERROR.value, error_code)
    error_packet += struct.pack('!{}sB'.format(len(error_message)), error_message.encode(), 0)

    return error_packet

def handle_received_packet(self, inputPacket):
    """
    создаем пакет который будет далее занесен в буффер
    """
    opcode = struct.unpack('!H', inputPacket[0:2])[0]
    packetTypes = {1: 'RRQ', 2: 'WRQ', 3: 'DATA', 4: 'ACK', 5: 'ERROR'}
    try:
        packetType = TftpProtocol.TftpPacketType(opcode)
    except ValueError: # несуществующий opcode
        self.reached_end = True
        err_msg = 'Illegal TFTP Opcode'
        print(err_msg)
        # возвращаем пакет с opcode = 5, error code = 4, сообщением ошибки
        return self.generate_error_packet(error_code=4, error_message=err_msg)

    if packetType == TftpProtocol.TftpPacketType.RRQ or

```

```

packetType == TftpProtocol.TftpPacketType.WRQ:
    self.fileBytes = []
    self.request_mode = packetTypes[opcode]
    separatorId = 2 + inputPacket[2:].find(0)
    # получаем id конца поля имени файла
    # + 2 поскольку индекс, возвращаемый поиском,
    # относится к подписке, начальный индекс 2:
    filenameBytes = inputPacket[2:separatorId]

    fmt_str = '!{s}'.format(len(filenameBytes))
    # распаковываем байты и получаем путь к файлу из кортежа
    self.file_path = struct.unpack(fmt_str, filenameBytes)[0]
    # если доступ к файлу сервера запрещен
    if str(self.file_path, encoding='ascii') == os.path.basename(__file__):
        self.reached_end = True
        self.fail = True
    return self.generate_error_packet(error_code=0, error_message="Access Forbidden")

    self.tftp = str(inputPacket[separatorId + 1:-1], 'ascii').lower()

if packetType == TftpProtocol.TftpPacketType.ACK and self.sent_last:
    # последний пакет acknowledged

    self.sent_last = False
    # конец передачи
    self.reached_end = True
    return []

if packetType == TftpProtocol.TftpPacketType.RRQ: # RRQ
    err = self.read_file()
    # проверяем существует ли файл на сервере
    if err:
        # error code =1, opcode for error = 5
        # формируем пакет ошибки
        error_code = 1
        self.err = 1
        err_msg = 'File not found.'
        self.reached_end = True
        print(err_msg)

```

```

return self.generate_error_packet(error_code=error_code, error_message=err_msg)

if packetType == TftpProtocol.TftpPacketType.WRQ: # WRQ
    # если файл не существует возвращаем ACK с блоком под номером 0
    if os.path.exists(self.file_path): # проверяем существует ли файл на сервере
        error_code = 6
        self.err = 6
        err_msg = 'File already exists'
        self.reached_end = True
        print(err_msg)
    return self.generate_error_packet(error_code=error_code, error_message=err_msg)

    outputPacket = struct.pack('!HH', TftpProtocol.TftpPacketType.ACK.value, 0)
elif packetType == TftpProtocol.TftpPacketType.DATA: # Data
    block_num = struct.unpack('!H', inputPacket[2:4])[0]

    if len(inputPacket) > 4: # последней пакет может иметь 0 байт
        len_data = len(inputPacket[4:])
        if len_data != 512:
            self.sent_last = True
            self.reached_end = True
            if self.tftp_mode == 'octet':
                fmt_str = '!{}B'.format(len_data)
            else: # netascii
                fmt_str = '!{}s'.format(len_data)
            unpacked_data_bytes = struct.unpack(fmt_str, inputPacket[4:])
            # вставляем байты полученного блока в файл чтобы далее записать
            self.fileBytes.extend(unpacked_data_bytes)
        else: # конец передачи
            self.reached_end = True

    outputPacket = struct.pack('!HH', TftpProtocol.TftpPacketType.ACK.value, block_num)

elif packetType == TftpProtocol.TftpPacketType.ERROR:
    self.reached_end = True
    err_msg = 'Not defined :' + str(inputPacket[4:-1], encoding='ascii')
    print(err_msg)
    # возвращаем ERROR пакет с opcode = 5, error code = 0, error message
    return self.generate_error_packet(error_code=0, error_message=err_msg)

```

```

if packetType == TftpProtocol.TftpPacketType.ACK or
packetType == TftpProtocol.TftpPacketType.RRQ:
    # ответить на RRQ с первым блоком и ACK с другими блоками
    if packetType == TftpProtocol.TftpPacketType.RRQ:
        block_num = 1
    else:
        block_num = struct.unpack('!H', inputPacket[2:4])[0] + 1
    # получаем блок данных после ack пакета, или первый если это rrq
    data_blocks = self.get_next_data_block(block_num)

    len_data = len(data_blocks)
    if len_data > 0: # проверяем есть ли еще блоки для отправки
        format_char = ''
        if self.tftp_mode == 'octet':
            format_char = '!B'
        elif self.tftp_mode == 'netascii':
            format_char = '!s'
        # блоки данных конвертируются в требующийся тип
        outputPacket = struct.pack('!HH', TftpProtocol.TftpPacketType.DATA.value,
            block_num)
        for byte in list(data_blocks):
            outputPacket += struct.pack(format_char, byte)
    else: # если размер файла кратен 512, то последний пакет не будет иметь данных
        outputPacket = struct.pack('!HH', TftpProtocol.TftpPacketType.DATA.value,
            block_num)
    return outputPacket

def get_next_data_block(self, block_num):
    # индексируем блоки
    startId = (block_num - 1) * 512
    endId = startId + 512

    if endId > (self.file_block_count): # если размер последнего блока меньше 512
        # конец передачи
        self.sent_last = True
        return self.fileBytes[startId:]
    elif endId == self.file_block_count: # отправляем пустой блок в конце передачи,
        # если размер кратен 512

```

```

        self.sent_last = True
        return []
    return self.fileBytes[startId: endId]

def get_next_packet(self):
    return self.packet_buffer.pop(0)

def has_packets_to_send(self):
    return len(self.packet_buffer) != 0

def save_file(self):
    if not self.fail:
        with open(self.file_path, 'wb') as uploadFile:
            uploadFile.write(bytes(self.fileBytes))

def read_file(self):
    try:
        with open(self.file_path, 'rb') as f:
            self.fileBytes = list(f.read())
            self.file_block_count = len(self.fileBytes)
        return False
    except FileNotFoundError: # если файл не существует
        return True

def set_client_address(self, client_address):
    self.client_address = client_address
    self.client_port = client_address[1]

def get_file_path(self):
    return str(self.file_path, encoding='ascii')

def get_file_size(self):
    return len(self.fileBytes)

def setup_sockets(address):
    serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    serverSocket.bind(address)
    return serverSocket

```

```

def main():
    server_address = ("127.0.0.1", 69)
    server_socket = setup_sockets(server_address)

    print('Server started at', server_address)
    while True:
        print('Waiting for connection...')
        protocol = TftpProtocol()
        # получаем пакет, содержащий строку запроса (RRQ или WRQ)
        request_packet, clientAddress = server_socket.recvfrom(2048)
        # путь к файлу может быть самым большим блоком в пакете,
        # поэтому размер пакета не может превышать 2048 байт.
        protocol.set_client_address(clientAddress)
        print('Connected to ', clientAddress)
        protocol.process_udp_packet(request_packet, clientAddress)
        request_mode = protocol.request_mode

        if request_mode == 'RRQ' or request_mode == 'WRQ':

            while protocol.has_packets_to_send():
                nextPacket = protocol.get_next_packet()
                server_socket.sendto(nextPacket, clientAddress)

            if not protocol.reached_end:
                # получаем новый пакет, если не достигли конца передачи
                received_packet, received_client = server_socket.recvfrom(2048)
                protocol.process_udp_packet(received_packet, received_client)
            else:
                print('TRANSMISSION ENDED')
                while protocol.ignore_current_packet:
                    # если получен случайный пакет, игнорировать его получить другой пакет
                    received_packet, received_client = server_socket.recvfrom(2048)
                    protocol.process_udp_packet(received_packet, received_client)
                print('file path on server:', protocol.get_file_path())
                print(protocol.get_file_size(), ' bytes transmitted ')

        if request_mode == 'WRQ' and protocol.err != 6 and protocol.err != 1:

```

```

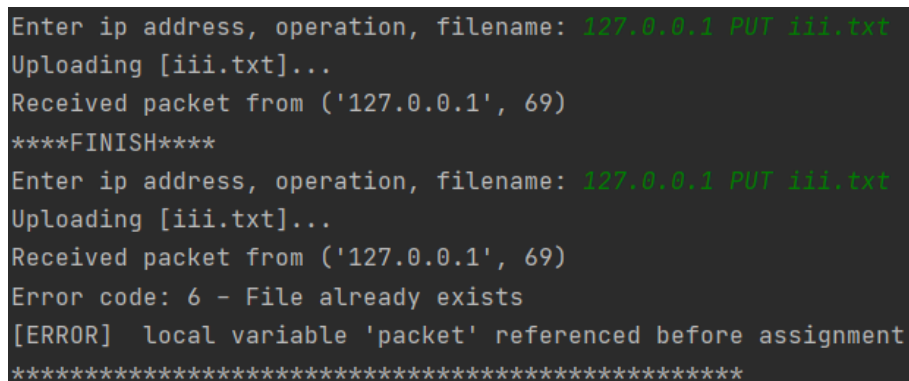
        # сохраняем файл после получения
        protocol.save_file()
    else:
        print('ERROR!')
        time.sleep(1)

if __name__ == "__main__":
    main()

```

### Листинг 5 — Сервер

Теперь клиент:



```

Enter ip address, operation, filename: 127.0.0.1 PUT iii.txt
Uploading [iii.txt]...
Received packet from ('127.0.0.1', 69)
****FINISH****
Enter ip address, operation, filename: 127.0.0.1 PUT iii.txt
Uploading [iii.txt]...
Received packet from ('127.0.0.1', 69)
Error code: 6 - File already exists
[ERROR] local variable 'packet' referenced before assignment
*****

```

Рисунок 0.4 — Передаем небольшой файл на сервер





```

"""
получаем udp пакет
отсылаем пакет, который нужно записать в сокет
Вход и выход - ТОЛЬКО байтовые массивы.
Выходные пакеты сохраняются в буфере в этом классе,
функция get_next_output_packet возвращает следующий пакет,
который нужно отправить.
Этот класс также отвечает за чтение/запись файлов на диск.
Несоблюдение этих требований приведет к ошибке.
"""

class TftpPacketType(enum.Enum):
    RRQ = 1
    WRQ = 2
    DATA = 3
    ACK = 4
    ERROR = 5

def __init__(self):
    self.packetBuffer = [] # буфер для хранения пакетов
    self.blockNumber = 0 # номер блока

    self.isFinished = False # старт/стоп протокола
    self.isReceiving = True # старт/стоп получения пакетов
    self.isMismatch = False # отправляем пакет еще раз в случае несовпадения

    self.errorCode = 0
    self.errorMessage = ""

def process_udp_packet(self, packetData, packetSource):
    """
    packet data - данные в bytearray
    packet source - информация об адресе отправителя
    """
    print(f"Received packet from {packetSource}")
    receivedOpcode = self.parse_udp_packet(packetData)
    outputPacket = self.handle_received_opcode(receivedOpcode)
    self.packetBuffer.append(outputPacket)

```

```

def parse_udp_packet(self, packetBytes):
    """
    получаем информацию о типе пакета и вытаскиваем данные
    """
    opcode = struct.unpack('!H', packetBytes[:2])[0]

    if opcode == self.TftpPacketType.DATA.value:
        self.blockNumber = struct.unpack('!H', packetBytes[2:4])[0]
        self.accessed_file.write(packetBytes[4:])
        if len(packetBytes) < (512 + 4):
            self.accessed_file.close()
            self.isFinished = True
            self.isReceiving = False

    elif opcode == self.TftpPacketType.ACK.value:
        block_number = struct.unpack('!H', packetBytes[2:4])[0]
        self.isMismatch = not (self.blockNumber == block_number)

    elif opcode == self.TftpPacketType.ERROR.value:
        _ = self.show_error(int(struct.unpack('!H', packetBytes[2:4])[0]))

    return opcode

def handle_received_opcode(self, receivedOpcode):
    """
    создаем пакет который будет далее занесен в буфер
    """

    if receivedOpcode == self.TftpPacketType.ACK.value:
        data = self.accessed_file.read(512)
        # проверяем последний ли это пакет
        if len(data) < 512:
            self.isFinished = True

        # проверяем, есть ли несоответствие в пакете, чтобы запросить его снова
        if self.isMismatch:
            packet = struct.pack('!HH', self.TftpPacketType.DATA.value, self.blockNumber)
                + data

```

```

        # получаем следующий пакет
    else:
        self.blockNumber += 1
    packet = struct.pack('!HH', self.TftpPacketType.DATA.value, self.blockNumber)
        + data

elif receivedOpcode == self.TftpPacketType.DATA.value:
    packet = struct.pack('!HH', self.TftpPacketType.ACK.value, self.blockNumber)

# # при ошибке завершаем соединение
# elif receivedOpcode == self.TftpPacketType.ERROR.value and
# self.errorCode != 6 and self.errorCode != 1:
#     sys.exit()
return packet

def has_packets_to_send(self):
    return len(self.packetBuffer) != 0

def get_next_packet(self):
    return self.packetBuffer.pop(0)

def download_file(self, file_name):
    # перед отправкой запроса проверяем, существует ли уже файл
    exists = os.path.exists(file_name)
    if exists:
        return self.show_error(6) # file already exists

    try:
        self.accessed_file = open(file_name, 'wb')
        mode = b'octet'
        opcode = self.TftpPacketType.RRQ.value
        file_name = file_name.encode('ascii')
        file = struct.pack('!H{sB}sB'.format(len(file_name), len(mode)), opcode, file_name,
                           0, mode, 0)
    except: # если случилась ошибка доступа, отправляем соответствующее сообщение
        file = self.show_error(2) # access violation

    return file

```

```

def upload_file(self, file_name):
    try:
        self.accessed_file = open(file_name, 'rb')
        mode = b'octet'
        opcode = self.TftpPacketType.WRQ.value
        file_name = file_name.encode('ascii')
        file = struct.pack('!H{sB{sB'.format(len(file_name), len(mode)), opcode, file_name,
                                0, mode, 0)
    except:
        exists = os.path.exists(file_name)
        if not exists: # если файл не существует
            file = self.show_error(1) # file not found
        else: # если случилась ошибка доступа, отправляем соответствующее сообщение
            file = self.show_error(2) # access violation
    return file

def show_error(self, errorCode):
    self.errorCode = errorCode
    self.isFinished = True
    self.isReceiving = False

    if errorCode == 0:
        self.errorMessage = b"Not defined, see error message (if any)"
    elif errorCode == 1:
        self.errorMessage = b"File not found"
    elif errorCode == 2:
        self.errorMessage = b"Access violation"
    elif errorCode == 3:
        self.errorMessage = b"Disk full or allocation exceeded"
    elif errorCode == 4:
        self.errorMessage = b"Illegal TFTP operation"
    elif errorCode == 5:
        self.errorMessage = b"Unknown transfer ID"
    elif errorCode == 6:
        self.errorMessage = b"File already exists"
    elif errorCode == 7:
        self.errorMessage = b"No such user"

    print("Error code:", self.errorCode, "-", self.errorMessage.decode('ascii'))

```

```

        packet = struct.pack('!HH{sB}'.format(len(self.errorMessage)),
                               self.TftpPacketType.ERROR.value, self.errorCode,
                               self.errorMessage, 0)
        return packet

def setup_sockets(address):
    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    serverAddress = (address, 69)

    return clientSocket, serverAddress

def do_socket_logic(protocol, address, request_packet):
    socket, server = setup_sockets(address)

    if request_packet:
        socket.sendto(request_packet, server)
        if protocol.isReceiving:
            packet, rev_addr = socket.recvfrom((512 + 4))

    while not protocol.isFinished:
        protocol.process_udp_packet(packet, rev_addr)
        if protocol.has_packets_to_send():
            socket.sendto(protocol.get_next_packet(), rev_addr)
            if protocol.isReceiving:
                packet, rev_addr = socket.recvfrom((512 + 4))
    print('****FINISH****')

def parse_user_input(address, operation, file_name=None):
    protocol = TftpProtocol()

    if operation == "PUT":
        print(f"Uploading [{file_name}]...")
        requested_file = protocol.upload_file(file_name)

        do_socket_logic(protocol, address, requested_file)

```

```

elif operation == "GET":
    print(f"Downloading [{file_name}]...")
    requested_file = protocol.download_file(file_name)
    do_socket_logic(protocol, address, requested_file)

else: # в случае некорректной tftp операции
    do_socket_logic(protocol, address, protocol.show_error(4))

def main():
    while True:
        try:
            (ip_address, operation, file_name) = input('Enter ip address, operation, filename: ').split(' ')
            parse_user_input(ip_address, operation, file_name)
        except Exception as e:
            print(f"[ERROR] ", e)
            print('*' * 50)

if __name__ == "__main__":
    main()

```

Листинг 6 — Клиент

## Тестирование

В ходе выполнения данной работы возникало множество проблем - то данные передавались с клиента, но не принимались на клиент, то наоборот. В ходе чего было принято решение максимально реплецировать код (чтобы сервер и клиент отличались лишь парой функций, а в остальном были максимально схожи), что устранило все проблемы.

Были протестированы следующие моменты:

- 1) Сервер и клиент корректно завершаются - не зависают в процессе завершения, клиент получает и обрабатывает информацию о закрытии соединения
- 2) Сервер и клиент позволяют корректно отправлять и принимать большие файлы. Для теста использовались различные текстовые документы, видео, аудио.
- 3) Для тестирования клиента также использовалась программа tftpd64.exe. Для тестирования сервера - клиенты однопользователей.
- 4) Отслеживать передачу пакетов помогала программа Wireshark.



## **Лабораторная работа №4**

### **Формулировка задания**

Работая в паре, реализовать клиент для калькулятора и сервер для дистанционной тестирующей системы.

## Программа работы

Начнем с сервера:

Протокол следующий:

Для начала работы с сервером требуется лишь его запустить.

```
Socket is created...
Socket is being used...
Binding is done...
The server is ready to receive
Vlud logged in
Vlud started Networks test
Dan logged in
Vlud give answer A for question 1 timestamp: 2021-01-18 17:21:15
Dan started Russian test
Dan give answer A for question 1 timestamp: 2021-01-18 17:21:33
Dan give answer B for question 2 timestamp: 2021-01-18 17:21:36
Dan give answer A for question 3 timestamp: 2021-01-18 17:21:38
Dan give answer A for question 4 timestamp: 2021-01-18 17:21:38
Dan give answer A for question 5 timestamp: 2021-01-18 17:21:39
Vlud give answer B for question 2 timestamp: 2021-01-18 17:21:42
Vlud give answer B for question 3 timestamp: 2021-01-18 17:21:43
Dan give answer A for question 6 timestamp: 2021-01-18 17:21:45
Dan answers for Russian test ['A', 'B', 'A', 'A', 'A', 'A']
Socket Information: 127.0.0.1:64370 | Username: Dan | Result:5/6 | Mark is 4
Vlud give answer A for question 4 timestamp: 2021-01-18 17:21:50
Vlud give answer A for question 5 timestamp: 2021-01-18 17:21:51
Vlud give answer B for question 6 timestamp: 2021-01-18 17:21:53
Vlud answers for Networks test ['A', 'B', 'B', 'A', 'A', 'B']
Socket Information: 127.0.0.1:64365 | Username: Vlud | Result:1/6 | Mark is 2
```

Рисунок 0.1 — Пример работы сервера

```
from socket import *
```

```
import datetime
```

```
import time
```

```
import threading
```

```
class ThreadedServer():
```

```
    def listenToClient(self, client, addr):
```

```
        userName = (client.recv(1024)).decode("utf-8")
```

```
        print(userName, "logged in")
```

```
        while True:
```

```
            counter = 0
```

```
            answers = []
```

```
            questions = []
```

```
            choice = (client.recv(1024)).decode("utf-8")
```

```
            if choice == "0":
```

```
                print(userName, "left the server")
```

```
                client.close()
```

```
                return
```

```

else:
    if choice == "1":
        testName = "Networks test"
    elif choice == "2":
        testName = "Russian test"
    elif choice == "3":
        testName = "Math test"
    print(userName, "started", testName)
    if choice == "1":
        questions = self.questions1
    elif choice == "2":
        questions = self.questions2
    else:
        questions = self.questions3
    for i in range(6):
        client.send(questions[counter].encode())
        message = client.recv(1024)
        if message == "exit":
            print(addr, " is closed")
            client.close()
        else:
            answers.append(message.decode("utf-8").upper())
            ts = time.time()
            st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
            print(userName, "give answer", answers[counter], "for question", counter + 1,
                  "timestamp:",
                  st)
            counter += 1
    self.assessment(addr, answers, userName, int(choice), client, testName)

def assessment(self, addr, answers, userName, choice, client, testName):

    point = 0
    print(userName, "answers for", testName, answers)
    if choice == 1:
        if (answers[0] == "A"):
            point += 1
        if (answers[1] == "A"):
            point += 1

```

```

    if (answers[2] == "A"):
        point += 1
    if (answers[3] == "C"):
        point += 1
    if (answers[4] == "D"):
        point += 1
    if (answers[5] == "A"):
        point += 1
elif choice == 2:
    if (answers[0] == "A"):
        point += 1
    if (answers[1] == "A"):
        point += 1
    if (answers[2] == "A"):
        point += 1
    if (answers[3] == "A"):
        point += 1
    if (answers[4] == "A"):
        point += 1
    if (answers[5] == "A"):
        point += 1
elif choice == 3:
    if (answers[0] == "B"):
        point += 1
    if (answers[1] == "B"):
        point += 1
    if (answers[2] == "B"):
        point += 1
    if (answers[3] == "B"):
        point += 1
    if (answers[4] == "B"):
        point += 1
    if (answers[5] == "B"):
        point += 1
if (point < 2):
    success_comment = "Mark is 2"
elif (point < 4):
    success_comment = "Mark is 3"
elif (point <= 5):

```

```

        success_comment = "Mark is 4"
    else:
        success_comment = "Mark is 5"

    client.send(("Your result of " + testName + " " + str(point) + "/6 | "
+ success_comment).encode())
    result = "Socket Information: " + str(addr[0]) + ":" + str(addr[1]) +
    " | Username: " + userName \
        + " | Result:" + str(point) + "/6 | " + success_comment
    print(result)
    return result

def __init__(self, serverPort):
    with open('questions.txt') as inp:
        self.sets = inp.read().split("FINISH")
        self.questions1 = self.sets[0].split("'''",")
        self.questions2 = self.sets[1].split("'''",")
        self.questions3 = self.sets[2].split("'''",")
    self.answers = []
    try:
        self.serverSocket = socket(AF_INET, SOCK_STREAM)
    except:
        print("Socket cannot be created!!!")
        exit(1)
    print("Socket is created...")
    try:
        self.serverSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    except:
        print("Socket cannot be used!!!")
        exit(1)
    print("Socket is being used...")
    try:
        self.serverSocket.bind(('', serverPort))
    except:
        print("Binding cannot de done!!!")
        exit(1)
    print("Binding is done...")
    try:
        self.serverSocket.listen(1)

```

```

except:
    print("Server cannot listen!!!")
    exit(1)
print("The server is ready to receive")
while True:
    connectionSocket, addr = self.serverSocket.accept()
    threading.Thread(target=self.listenToClient, args=(connectionSocket, addr)).start()

if __name__ == "__main__":
    serverPort = 5006
    ThreadedServer(serverPort)

```

### Листинг 7 — Сервер дистанционной тестирующей системы

#### Теперь клиент:

Протокол следующий:

Для начала работы с клиентом после его запуска пользователю требуется ввести числовое выражение. Поддерживаются операции  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $!$ ,  $\text{sqrt}$ . Для завершения работы нужно отправить команду `!exit`.

```
The IP address of the server: localhost , port: 5005
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 5-5
The answer is: 0
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 5*5
The answer is: 25
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 4+4
The answer is: 8
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 3/4
The answer is: 0.75
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 20!
The answer is: 2432902008176640000
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 3!
The answer is: 6
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: sqrt9
The answer is: 3.0
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 25a!
The answer is: ValueError
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: 7/0
You can't divide by 0, try again
Type your equation: arg|+,-,*,/|arg or arg! or sqrt|arg
Type exit! to quit
Input line: ■
```

Рисунок 0.2 — Пример работы клиента

```
from socket import *
```

```
serverName = "localhost"
```

```
serverPort = 5005
```

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

```

clientSocket.connect((serverName, serverPort))

operation = {
    '+': 0,
    '-': 1,
    '*': 2,
    '/': 3,
}

print("The IP address of the server:", serverName, ", port: ", serverPort)

while (True):
    valid = False
    while not valid:
        print("Type your equation: arg |+,*,/,| arg or arg! or sqrt|arg")
        print("Type exit! to quit")
        equ = input("Input line: ")
        if "!" not in equ and "sqrt" not in equ:
            line = equ.split(" ")
            if len(line) == 3:
                if line[1] in operation:
                    try:
                        arg1 = int(line[0])
                        arg3 = int(line[2])
                        valid = True
                        clientSocket.send((line[0] + line[1] + line[2]).encode())
                    except error:
                        print("wrong arguments")
                else:
                    print("wrong operation")
            else:
                print("wrong input. Example: 1 + 1 ; 2 * 2 ; 3 - 3 ; 4 / 4")
        else:
            valid = True
            clientSocket.send(equ.encode())

    try:

```



```

    result = clientSocket.recv(1024).decode()
except ConnectionResetError:
    print(f'Closed connection')
    clientSocket.shutdown(socket.SHUT_RDWR)
    clientSocket.close()
    exit(0)

if not result:
    print(f'Closed connection')
    clientSocket.shutdown(socket.SHUT_RDWR)
    clientSocket.close()
    exit(0)
if result == "exit!":
    print("You left the server")
    clientSocket.close()
    exit(0)
elif result == "ZeroDiv":
    print("You can't divide by 0, try again")
elif result == "MathError":
    print("There is an error with your math, try again")
elif result == "SyntaxError":
    print("There is a syntax error, please try again")
elif result == "NameError":
    print("You did not enter an equation, try again")
else:
    print("The answer is:", result)

```

Листинг 8 — Клиент калькулятор

## Тестирование

Были протестированы следующие моменты:

- 1) Сервер и клиент корректно завершаются - не зависают в процессе завершения, клиент получает и обрабатывает информацию о закрытии соединения
- 2) Сервер корректно обрабатывает запросы от клиента.
- 3) Все вычисления обрабатываются корректно. В случае некорректного ввода пользователя - высвечиваются соответствующие ошибки и работа сервиса не прекращается.

## Заключение

В ходе данной лабораторной работы были получены базовые знания в сфере компьютерных сетей.

Было изучено:

- 1) Принцип работы блокирующих сокетов в TCP протоколе на примере консольного чата.
- 2) Принцип работы не блокирующих сокетов в TCP протоколе на примере консольного чата.
- 3) Принцип передачи данных в UDP протоколе. Для этого был реализован TFTP протокол, используемый для передачи файлов между удаленными точками.
- 4) Программа WireShark для отслеживания трафика передачи пакетов в сети.
- 5) На основе полученных знаний в последней работе в паре были реализованы сервис для вычислений (калькулятор) и сервис для дистанционного тестирования знания.