

## ОПИСАНИЕ ИТОГОВОГО ЗАДАНИЯ К МОДУЛЮ 20.

### ПРОСТОЙ КОД ПЕРЕДАЧИ ДАННЫХ МЕЖДУ КЛИЕНТОМ И СЕРВЕРОМ В WINDOWS

За основу программы был взят код чат-мессенджера, реализованный в домашних заданиях к модулям 15, 16 и 18. Однако, поскольку была поставлена задача прежде всего реализовать передачу сообщений между двумя экземплярами программы – клиентом и сервером, код был предельно упрощен, из всех исходных классов был оставлен только класс создания сообщений Message, состоящий из заголовка и файла реализации.

Написание исходного кода сервера и клиента для ОС Windows было выполнено по материалам сайта <https://habr.com/ru/articles/582370/>, в котором было показано программирование сокетов на основе библиотек `WinSock2.h` и `WS2tcpip.h`. Изначально весь код находился в блоке `main`, а методы приема и передачи сообщений выглядели следующим образом:

#### Для сервера:

```
std::vector<char> servBuff(BUFF_SIZE), clientBuff(BUFF_SIZE);
short packet_size = 0;
while (true)
{
    packet_size = recv(ClientConn, servBuff.data(), servBuff.size(), 0);
    std::cout << "Сообщение клиента: " << servBuff.data() << std::endl;
    std::cout << "Ответ сервера: ";
    fgets(clientBuff.data(), clientBuff.size(), stdin);
    packet_size = send(ClientConn, clientBuff.data(), clientBuff.size(), 0);

    if (packet_size == SOCKET_ERROR)
    {
        std::cout << "Невозможно отправить сообщение клиенту. Ошибка " << WSAGetLastError() << std::endl;
        closesocket(ServSock);
        closesocket(ClientConn);
        WSACleanup();
        return 1;
    }
}
```

#### И для клиента:

```
std::vector<char> servBuff(BUFF_SIZE), clientBuff(BUFF_SIZE);
```

```

short packet_size = 0;
while (true)
{
    std::cout << "Ваше сообщение серверу: ";
    fgets(clientBuff.data(), clientBuff.size(), stdin);
    packet_size = send(ClientSock, clientBuff.data(), clientBuff.size(), 0);
    if (packet_size == SOCKET_ERROR)
    {
        std::cout << "Невозможно отправить сообщение. Ошибка " << WSAGetLastError() << std::endl;
        closesocket(ClientSock);
        WSACleanup();
        return 1;
    }
    packet_size = recv(ClientSock, servBuff.data(), servBuff.size(), 0);
    if (packet_size == SOCKET_ERROR) {
        std::cout << "Невозможно получить ответ сервера. Ошибка " << WSAGetLastError() << std::endl;
        closesocket(ClientSock);
        WSACleanup();
        return 1;
    }
    else
        std::cout << "Ответ сервера: " << servBuff.data() << std::endl;
}

```

Т.е. непосредственно сообщение записывалось из консоли строкой кода, выделенной красным цветом, и передавалось в массив `std::vector<char> servBuff(BUFF_SIZE)` для сервера и `std::vector<char> clientBuff(BUFF_SIZE)` для клиента, после чего указанные массивы передавались для отправки в функции `send()` и `recv()`.

Но таким образом приведенный выше код мог передавать и принимать только простые текстовые строки. А была поставлена задача передавать и принимать сообщения чата, представленного в предыдущих модулях, и представляющие собой объекты класса `Message` с полями типа `std::string`. Напрямую этот тип данных практически не поддается сериализации (по крайней мере, мне не удалось найти приемлемый способ без привлечения библиотек `boost` и `Qt`). Конечно, можно было бы заменить строковые поля массивами `char*` заданной длины, но такое преобразование не подходило для функции получения текущей даты и времени, использованной в предыдущих реализациях чата. Поэтому было

решено преобразовывать сообщения-экземпляры класса в текстовые файлы, используя методы класса `Message`, и передавать уже их. Методы для передачи файлов по протоколу TCP/IP были найдены в видеоматериале по адресу <https://yandex.ru/video/preview/9443398785359368320> - это функции `void send_file(SOCKET* sock, const std::string& file_name)` (передача) и `void recv_file(SOCKET* sock)` (прием).

Весь код сокетов был вынесен в отдельные заголовочные файлы `serv_socket.h` и `klient_socket.h` (для клиента), в которых код сокетов был обернут в функции `void servSocket()` и `void klientSocket()`. В этих же файлах была записана и реализация указанных выше методов работы с файлами. Вызов же методов осуществлялся внутри функций `void servSocket()` и `void klientSocket()`, причем он сопровождался вызовом методов класса `Message`. Для сервера эти вызовы выглядели так:

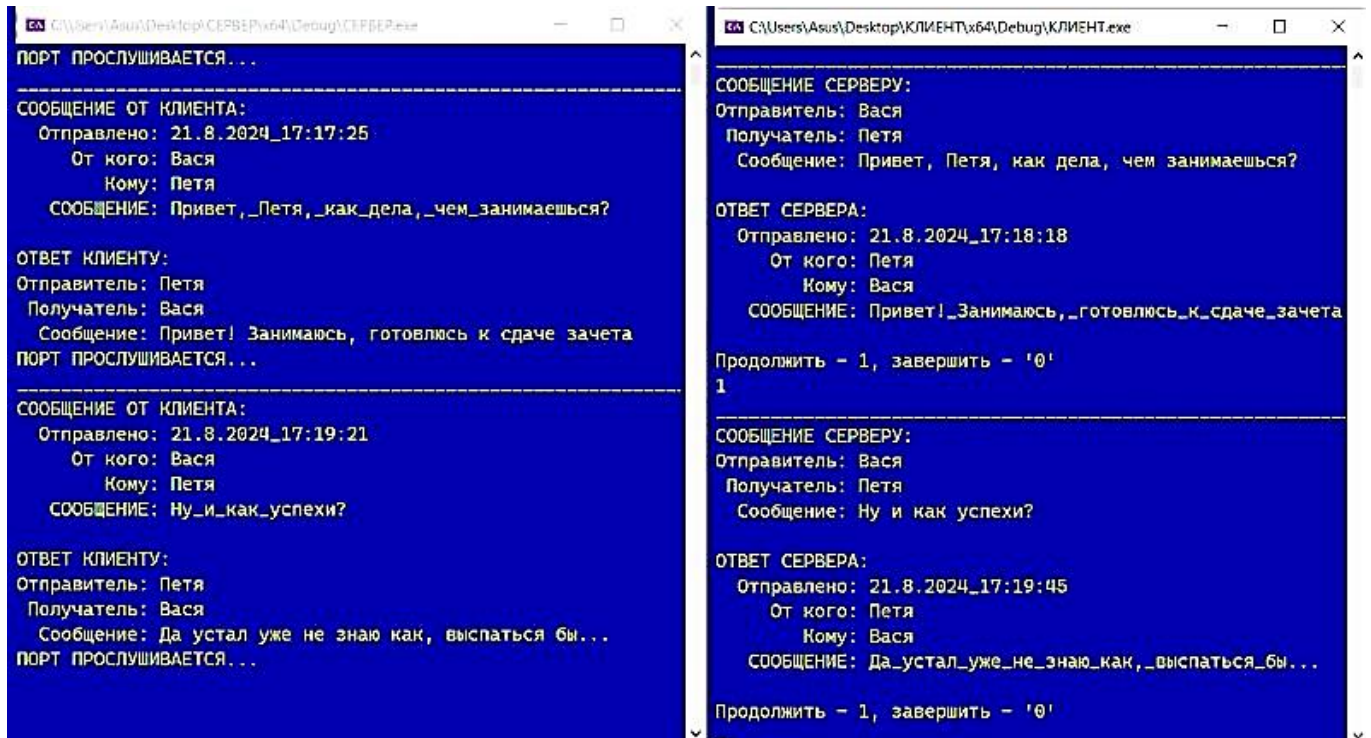
```
std::cout << "СООБЩЕНИЕ ОТ КЛИЕНТА: " << '\n';
    recv_file(&ClientConn); //прием файла
    ms.readPersonalMessages(); //восстановление сообщения из полученного текстового
    файла "Messages.txt";
    std::cout << '\n';
std::cout << "ОТВЕТ КЛИЕНТУ: " << '\n';
    ms.createMessages(); //создание сообщения
    std::string path = "Messages.txt";
    send_file(&ClientConn, path); //отправка файла
```

Для клиента все выглядело так же, но в обратном порядке:

```
std::cout << "СООБЩЕНИЕ СЕРВЕРУ: " << '\n';
    ms.createMessages();
    std::string path = "Messages.txt";
    send_file(&ClientSock, path);
    std::cout << '\n';
std::cout << "ОТВЕТ СЕРВЕРА: " << '\n';
    recv_file(&ClientSock);
    ms.readPersonalMessages();
```

Ну и в блоке `main` осуществлялся вызов только функций `servSocket()` и `klientSocket()`. Первым запускаем сервер, включающий порт на прослушивание, затем запускаем клиент, в котором и пишем первое сообщение. Результат представлен на ри-

сунке 1. Как видно из рисунка, прием и передача сообщений происходит именно в том формате, в котором сообщения реализованы в классе `Message`.



```
C:\Users\Asus\Desktop\SERVER\Py64\Debug\SERVER.exe
ПОРТ ПРОСЛУШИВАЕТСЯ...
-----
СООБЩЕНИЕ ОТ КЛИЕНТА:
Отправлено: 21.8.2024_17:17:25
От кого: Вася
Кому: Петя
СООБЩЕНИЕ: Привет, _Петя, _как_дела, _чем_занимаешься?
-----
ОТВЕТ КЛИЕНТУ:
Отправитель: Петя
Получатель: Вася
Сообщение: Привет! Занимаюсь, готовлюсь к сдаче зачета
ПОРТ ПРОСЛУШИВАЕТСЯ...
-----
СООБЩЕНИЕ ОТ КЛИЕНТА:
Отправлено: 21.8.2024_17:19:21
От кого: Вася
Кому: Петя
СООБЩЕНИЕ: Ну_и_как_успехи?
-----
ОТВЕТ КЛИЕНТУ:
Отправитель: Петя
Получатель: Вася
Сообщение: Да устал уже не знаю как, выспаться бы...
ПОРТ ПРОСЛУШИВАЕТСЯ...

C:\Users\Asus\Desktop\КЛИЕНТ\Py64\Debug\КЛИЕНТ.exe
СООБЩЕНИЕ СЕРВЕРУ:
Отправитель: Вася
Получатель: Петя
Сообщение: Привет, Петя, как дела, чем занимаешься?
-----
ОТВЕТ СЕРВЕРА:
Отправлено: 21.8.2024_17:18:18
От кого: Петя
Кому: Вася
СООБЩЕНИЕ: Привет! Занимаюсь, _готовлюсь_к_сдаче_зачета
-----
Продолжить - 1, завершить - '0'
1
-----
СООБЩЕНИЕ СЕРВЕРУ:
Отправитель: Вася
Получатель: Петя
Сообщение: Ну и как успехи?
-----
ОТВЕТ СЕРВЕРА:
Отправлено: 21.8.2024_17:19:45
От кого: Петя
Кому: Вася
СООБЩЕНИЕ: Да_устал_уже_не_знаю_как, _выспаться_бы...
-----
Продолжить - 1, завершить - '0'
```

Рисунок 1. Образец работы сервера и клиента

Как видим, метод выхода из цикла написания сообщений предусмотрен только для клиента, а на сервере цикл оставлен бесконечным, во избежание конфликтов и нарушений работы системы. Т.е. вначале выходим из клиента выбором «0», закрываем окно клиента, после чего закрываем окно сервера.

Интересно, что такой простой код позволяет передавать и принимать файлы любых других форматов – например, форматы .png, .jpg, .docx и .xls (другие не пробовал). Для этого перед функцией `send_file(&ClientSock, path)` нужно ввести полное имя файла – `std::cin>>path` (с расширением, и в имени файла не должно быть пробелов), предварительно скопировав файл в папку «Клиент». Или прописать полный путь к файлу.

Но, к большому сожалению, данный код категорически не пожелал компилироваться в `Linux`, несмотря на все `#if defined` и `#endif`. Как видно, программирование сокетов в `Linux` (как это приведено в учебном пособии) осуществляется совершенно по-другому, и под `Linux` весь код нужно переписывать заново, с другими методами передачи файлов.