Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт кибербезопасности и защиты информации

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 2

«МЕХАНИЗМЫ СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ»

по дисциплине «Операционные системы»

Выполнил
студент гр. 4851003/10002 Галкин К. К.

Руководитель
К. н. т Крундышев В. М.

Санкт-Петербург
2023

1. ЦЕЛЬ РАБОТЫ

Цель работы — изучить программный интерфейс сетевых сокетов, получить навыки организации взаимодействия программ при помощи протоколов Internet и разработки прикладных сервисов.

2. ХОД РАБОТЫ

Перед выполнением работы были поставлены следующие задачи:

- Написать TCP-клиент под ОС Windows и TCP-сервер под ОС Linux с механизмом параллельного обслуживания poll.
- Написать UDP-клиент под ОС Linux и UDP-сервер под ОС Windows с использование механизма WSAEvent().
- Продумать тесты для программ, рассматривающие “нормальные” и граничные случаи.
- Протестировать разработанное решение.

Блок – схемы серверов представлены ниже

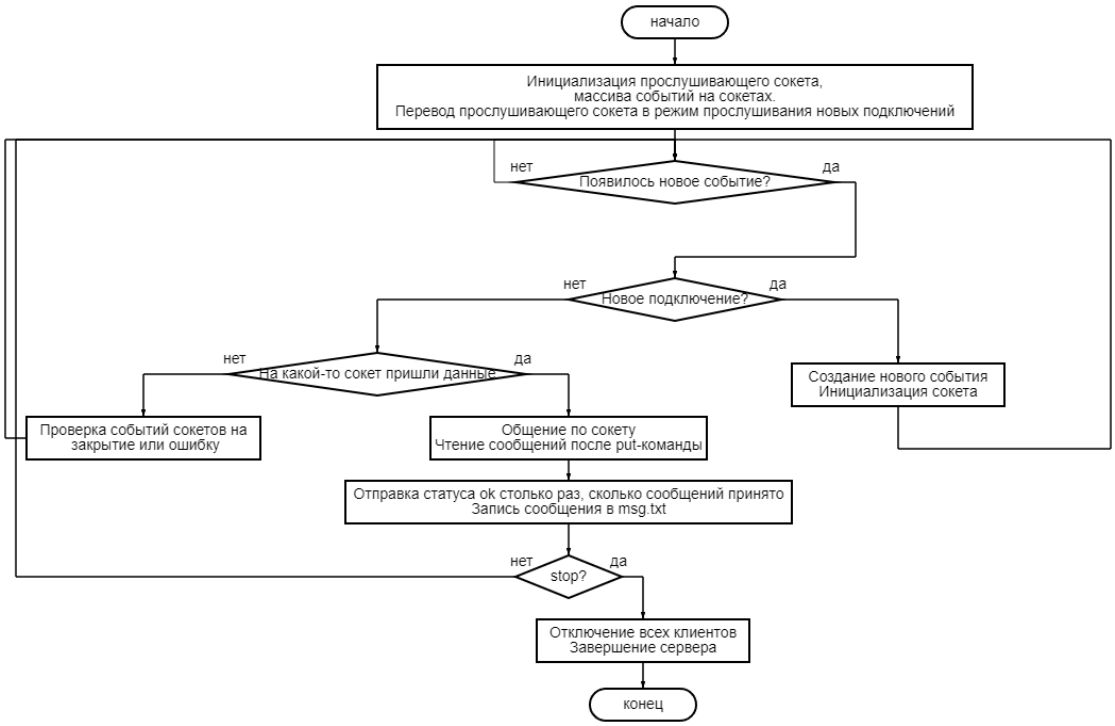


Рисунок 1 Блок-схема работы TCP-сервера

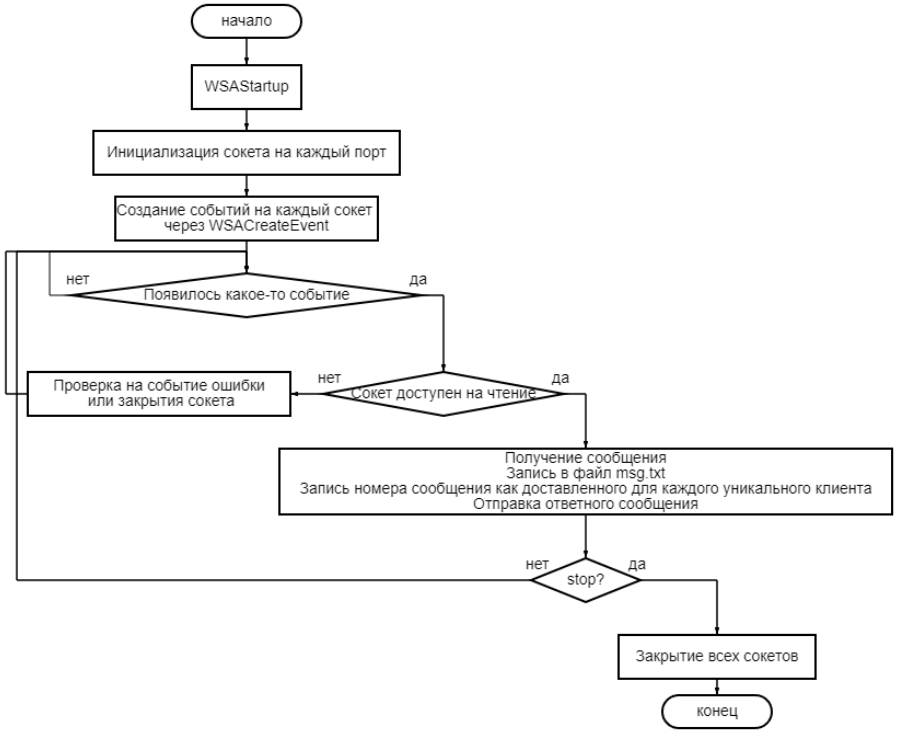


Рисунок 2 Блок-схема работы UDP-сервера

Таблица для описания структур и используемых функций представлена ниже:

Функция/Структура	Назначение
pollfd pfd[MAX_SOCKETS + 1]	Массив структур с описанием состояний сокетов. Нужна для обнаружения новых событий
union convert, short_ints	Объединения для конвертации чисел с обратным порядком байтов. Используются два поля: одно – число с необходимым размером в байтах, второе – буфер определенного размера байтов.
string findClient(struct sockaddr_in *addr)	Функция для возврата строки вида IP_SOURCE:PORT из структуры addr
int writeToFile(string buffer, string client)	Функция запись полученного сообщения в файл. Возвращает -1, если обнаружено сообщение stop
void disconnectClient(int index, sockaddr_in *addr)	Функция отключения клиента, освобождение сокета
static int readPacket(int sock, sockaddr_in *addr, int index)	Функция последовательного чтения данных из пакета. Возвращает значение функии writeToFile()

Таблица с описанием структур и функций UDP-сервера:

Функция/Структура	Назначение

map<string, set<uint32_t, greater<uint32_t>>> clients;	Словарь всех клиентов и их номеров сообщений. Номера сообщений отсортированы.
SOCKET *sockets_desc	Массив всех сокетов
struct sockaddr_in *addr	Массив структур адресов для каждого сокета
union convert, short_ints	Объединения для конвертации чисел с обратным порядком байтов. Используются два поля: одно – число с необходимым размером в байтах, второе – буфер определенного размера байтов.
string findClient(struct sockaddr_in *addr)	Функция для возврата строки вида IP_SOURCE:PORT из структуры addr
int writeToFile(string buffer, string client)	Функция запись полученного сообщения в файл. Возвращает -1, если обнаружено сообщение stop
int analyseDatagram(SOCKET sock, int index)	Функция анализа полученного сообщения. Проверка на то, если клиент уникальный, добавление к множеству всех отправленных сообщений клиента нового значения из полученной датаграмы.
void configListeners(uint16_t port_start, uint16_t port_end)	Функция настройки сокетов для каждого порта. Идет проверка на то, что первый аргумент(порт) должен

	быть меньше второго, так же если port_start и port_end совпадают, то конфигурация идет только одного сокета.
void sendToClient(SOCKET s, int index)	Отправка клиенту ответного сообщения, в котором находятся номера отправленных клиентом сообщений.
void disconnectClient(int index, sockaddr_in *addr)	Функция отключения клиента, освобождение сокета

Клиенты работают примерно по одному и тому же принципу, независимо от используемой ОС:

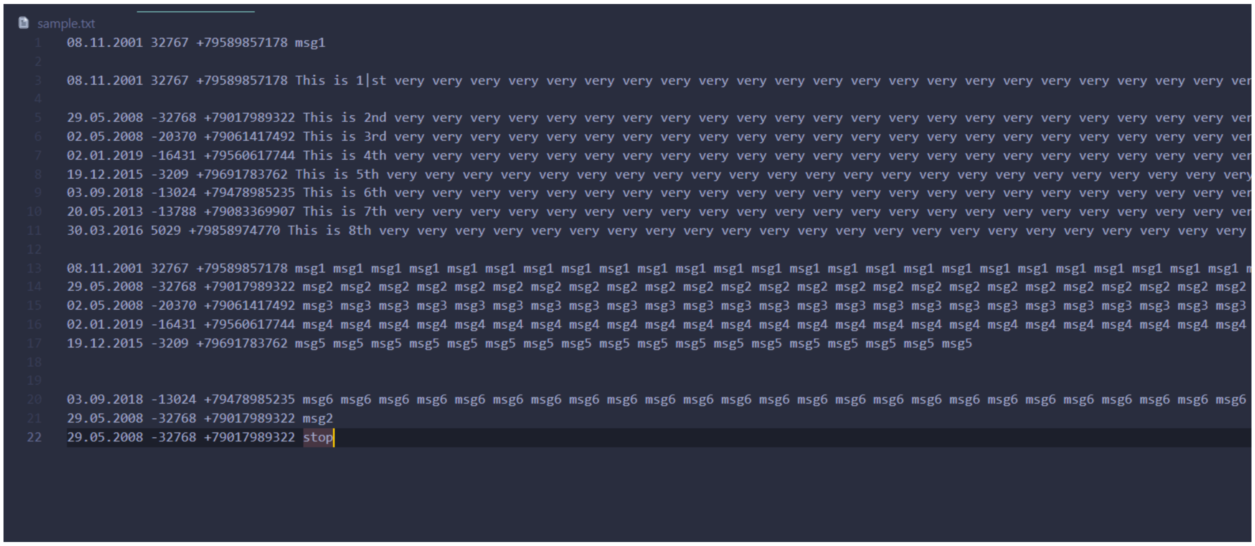
- Инициализация соединения. Попытка подключения через connect в течение 10 раз с разницей в 100мс.
- Чтения сообщений из файла в вектор строк.
- Для TCP – отправка команды put
    - После отправляется каждое сообщение из вектора msgs в отформатированном виде
    - Идет получение статуса ok на каждое сообщение
    - Если все сообщения доставлены – сокет закрывается
- Для UDP-клиента
    - Инициализация сокета
    - Чтение данных из файла в словарь строк allMsgs, где каждому сообщению присвоен индекс
    - Создается очередь из неотправленных сообщений.

- Попытка отправить пачку отформатированных данных
- Проверка последнего пакета на номера полученных сообщений
- Полученные номера попадают во множество msgArchive
- Очередь для отправки обновляется до тех пор, пока не останется сообщений.

Для тестирования такого рода приложений стоит учитывать:

- Максимальный размер тела пакета для UDP(учитывается так же для TCP) – 65507 байтов. То есть нужно проверить отправку строки большого размера
- В данных из файла может быть только одна строка с сообщением stop.
- Проверять последовательность получения сообщений в UDP нет смысла в силу гонки обработки данных на сокетах.
- Стоит проверить набор данных с пустыми строками в файле
- Стандартная проверка с обычным сообщением.
- Для UDP клиента нужно проверить граничные случаи инициализации сокетов на диапазон портов: порт указан на самом деле один, портов слишком много, сначала указан конец диапазона, потом начало.
- Нужно сделать тесты на проверку аварийного завершения программы, если не получилось настроить конфигурацию сокета или поставить сокет на прослушивание.

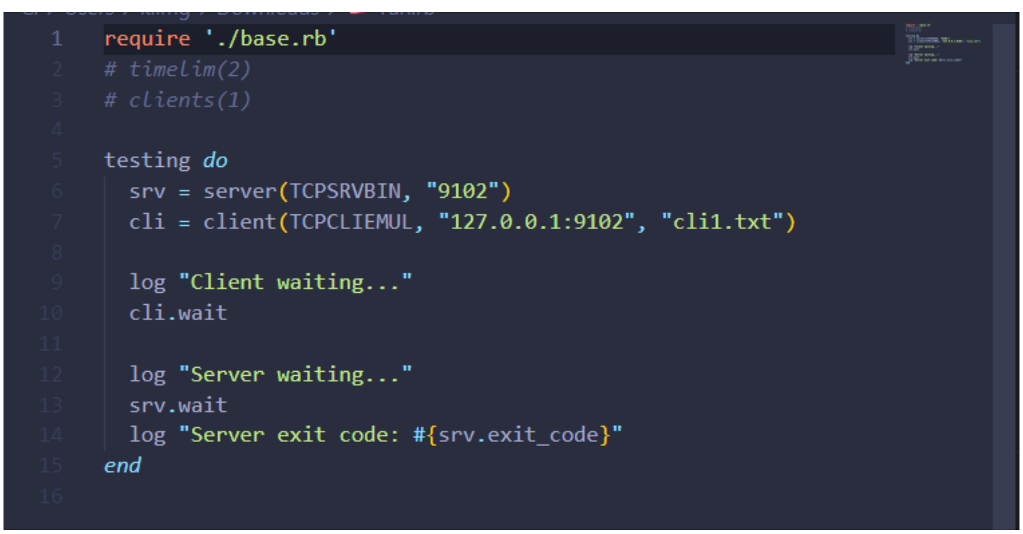
Вот простой пример входных сообщений для программ-клиентов:



```
sample.txt
08.11.2001 32767 +79589857178 msg1

08.11.2001 32767 +79589857178 This is 1st very very very very very very very very very very very very very very very very very very very very very very very ver

29.05.2008 -32768 +79017989322 This is 2nd very very very very very very very very very very very very very very very very very very very very very very very ver
02.05.2008 -20370 +79061417492 This is 3rd very very very very very very very very very very very very very very very very very very very very very very very ver
02.01.2019 -16431 +79560617744 This is 4th very very very very very very very very very very very very very very very very very very very very very very very ver
19.12.2015 -3209 +79691783762 This is 5th very very very very very very very very very very very very very very very very very very very very very very very very
03.09.2018 -13024 +79478985235 This is 6th very very very very very very very very very very very very very very very very very very very very very very very ver
20.05.2013 -13788 +79083369907 This is 7th very very very very very very very very very very very very very very very very very very very very very very very ver
30.03.2016 5029 +79858974770 This is 8th very very very very very very very very very very very very very very very very very very very very very very very very

08.11.2001 32767 +79589857178 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 msg1 m
29.05.2008 -32768 +79017989322 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2 msg2
02.05.2008 -20370 +79061417492 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3 msg3
02.01.2019 -16431 +79560617744 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4 msg4
19.12.2015 -3209 +79691783762 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5 msg5


03.09.2018 -13024 +79478985235 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6 msg6
29.05.2008 -32768 +79017989322 msg2
29.05.2008 -32768 +79017989322 stop
```

Рисунок 3 - Набор сообщений для тестирования

Проверка работоспособности сетевых функций делается путем написания скриптов, которые либо запускают клиентов параллельно, либо последовательно на один сокет.



```
require './base.rb'
# timelim(2)
# clients(1)

testing do
  srv = server(TCPSRVBIN, "9102")
  cli = client(TCPCLIEMUL, "127.0.0.1:9102", "cli1.txt")

  log "Client waiting..."
  cli.wait

  log "Server waiting..."
  srv.wait
  log "Server exit code: #{srv.exit_code}"
end
```

Рисунок 4 - Пример теста для проверки сетевых функций

Тексты всех программ представлены в приложений.

3. ВЫВОД

В ходе выполнения лабораторной работы были изучены механизмы сетевого взаимодействия программ, параллельного обслуживания клиентов, спецификации современных протоколов.

Были написаны программы-клиенты и программы-серверы для протоколов TCP и UDP под разные ОС с разными механизмами обслуживания. В ходе написания кода возникали проблемы аварийных завершений программы до закрытия сокета, непонимания технического задания, некоторые функции работали некорректно (как, например, memcpy).

Для разработанных решений были описаны тестовые ситуации, которые позволяют покрыть большую часть функционала программ.

4. ПРИЛОЖЕНИЕ

a. TCP-клиент

```
#define _CRT_SECURE_NO_WARNINGS
#define WIN32_LEAN_AND_MEAN
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#pragma comment(lib, "ws2_32.lib")
#include <winsock2.h>
#include <ws2tcpip.h>
#include <string>
#include <vector>
#include <stdint.h>
```

```
#define TCP_MAX_SIZE 65507
using namespace std;

vector<string> msgs;

int init()
{
   WSADATA wsaData;
   return (0 == WSAStartup(MAKEWORD(2, 2), &wsaData));
}

void readFile(char *fileName)
{
   ifstream fdesc(fileName);
   string msg;
   while (getline(fdesc, msg))
   {
      if (!msg.empty())
      {
         msgs.push_back(msg);
      }
      msg.clear();
   }
   fdesc.close();
}

void append_chunk(string &s, const uint8_t *chunk, size_t chunk_num_bytes)
{
   s.append((char *)chunk, chunk_num_bytes);
```

```
}

void transferData(SOCKET s)
{
    char ok[3] = {0};
    int okCount = 0;
    int msgCount = msgs.size();
    while (okCount != msgCount)
    {
        string currentMsg = msgs[okCount], send_msg, token;
        vector<string> Tokens;
        istringstream _stream(currentMsg);
        while (std::getline(_stream, token, ' '))
        {
            if (!token.empty())
            {
                Tokens.push_back(token);
            }
            token.clear();
        }
        vector<string> dateParts;
        stringstream date(Tokens[0]);
        string part;
        while (std::getline(date, part, '.'))
        {
            dateParts.push_back(part);
            part.clear();
        }
        uint32_t index_in_message = htonl(okCount);
```

```
uint8_t day = (uint8_t)atoi(dateParts[0].c_str());
uint8_t month = (uint8_t)atoi(dateParts[1].c_str());
uint16_t year = htons(atoi(string(dateParts[2].begin(), dateParts[2].end()).c_str()));
int16_t AA = htons(atoi(Tokens[1].c_str()));

/* Phone Number Process */
char *phone = (char *)Tokens[2].c_str();
/* Packing message process */
send_msg.reserve(currentMsg.size() + 4);
append_chunk(send_msg, (const uint8_t *)&index_in_message, sizeof(uint32_t));
append_chunk(send_msg, (const uint8_t *)&day, 1);
append_chunk(send_msg, (const uint8_t *)&month, 1);
append_chunk(send_msg, (const uint8_t *)&year, sizeof(uint16_t));
append_chunk(send_msg, (const uint8_t *)&AA, sizeof(signed short));
append_chunk(send_msg, (const uint8_t *)phone, strlen(phone));

/* Pacling all data after phone */
size_t data_pos = currentMsg.find("+") + 13;
string data = currentMsg.substr(data_pos);
append_chunk(send_msg, (const uint8_t *)data.c_str(), data.length());
send_msg[send_msg.size()] = '\0';
int status = send(s, send_msg.c_str(), send_msg.size() + 1, 0);
int r;
while ((r = recv(s, ok, 2, 0)))
{
   if (strcmp(ok, "ok") == 0)
   {
      okCount++;
      break;
```

```
        }
      }
    }
}

int main(int argc, char **argv)
{
    bool isConnected = false;
    if (argc != 3)
    {
        cout << "Usage ./tcpclient IP:PORT TEXTFILE" << endl;
        return 0;
    }
    char *ip = strtok(argv[1], ":");
    char *port = strtok(NULL, "\0");
    struct sockaddr_in addr;
    init();
    SOCKET s = socket(AF_INET, SOCK_STREAM, 0);

    if (s < 0)
    {
        cout << "socket() function error" << endl;
        return 0;
    }

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(port));
    addr.sin_addr.s_addr = inet_addr(ip);
```

```
for (int connAttempt = 0; connAttempt < 10 && !isConnected; connAttempt++)
{
   if (connect(s, (struct sockaddr *)&addr, sizeof(addr)) != -1)
   {
      int startMessaging = send(s, "put", 3, 0);
      if (startMessaging < 0)
      {
         cout << "Sending problems..." << endl;
         break;
      }
      isConnected = true;
      cout << "Connection complete!" << endl;
   }
   else
   {
                              cout << "Attempt to connect..." << endl;
      Sleep((DWORD)100);
   }
}
if (!isConnected)
{
   cout << "Connection refused" << endl;
   closesocket(s);
   return 0;
}

readFile(argv[2]);
transferData(s);
```

```
    closesocket(s);
    WSACleanup();
    return 0;
}
```

b. TCP-сервер.

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/poll.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>

#define FILE_LOGGER "msg.txt"
#define MAX_SOCKETS 255
#define TCP_MAX_SIZE 65507
using namespace std;

int s_c = 0;
int cs[MAX_SOCKETS];
pollfd pfd[MAX_SOCKETS + 1];
```

```
union
{
   uint32_t integer;
   char buffer[4];
} convert;

union
{
   char buffer[2];
   uint16_t integer;
   int16_t ninteger;
} short_ints;

string findClient(struct sockaddr_in *addr)
{
   char ip[16];
   char port[5];
   string client;
   inet_ntop(AF_INET, &addr->sin_addr, ip, sizeof(ip));
   client.clear();
   snprintf(port, 5, "%d", htons(addr->sin_port));
   client.append((char *)ip, strlen(ip));
   client += ':';
   client.append((char *)port, strlen(port));
   return client;
}

int writeToFile(string buffer, string client)
{
```

```
    ofstream file_desc;
    file_desc.open(FILE_LOGGER, ios::app);
    uint16_t day, month, year = 0;

    day = buffer[0];
    month = buffer[1];
    short_ints.buffer[0] = buffer[3];
    short_ints.buffer[1] = buffer[2];
    year = short_ints.integer;

    file_desc << client << " ";
    file_desc << setw(2) << setfill('0') << day << '.' << setw(2) << setfill('0') << month << '.' << year << " ";

    short_ints.buffer[0] = buffer[5];
    short_ints.buffer[1] = buffer[4];
    int16_t AA = short_ints.ninteger;
    file_desc << AA << " ";

    file_desc << string(buffer.begin() + 6, buffer.begin() + 18) << " " << string(buffer.begin() + 18,
buffer.end() - 1) << endl;
    file_desc.close();

    if (string(buffer.begin() + 18, buffer.end() - 1) == "stop")
    {
        return -1;
    }
    return 0;
}
```

```
void disconnectClient(int index, sockaddr_in *addr)
{
   string client = findClient(addr);
   cout << "  Peer disconnected: " << client << endl;
   cs[index] = -1;
   close(pfd[index].fd);
   pfd[index].fd = -1;
}

static int readPacket(int sock, sockaddr_in *addr, int index)
{
   char *buffer = new char[TCP_MAX_SIZE];
   int r = 0, write = 0;
   string b;
   while ((r = recv(sock, buffer, TCP_MAX_SIZE, 0)) > 0)
   {
      string client = findClient(addr);
      b.append((char *)buffer, r);
      write = writeToFile(b, client);
      int s_status = send(sock, "ok", 2, 0);
      b.clear();
   }
   //disconnectClient(index, addr);
   delete[] buffer;
   return write;
}

int main(int argc, char **argv)
{
```

```
if (argc != 2)
{
   cout << "Usage: ./tcpserver <port>" << endl;
   return 0;
}
int i;
uint32_t port = atoi(argv[1]);
sockaddr_in addr;

unsigned long mode = 1, opt = 1;
int s = socket(AF_INET, SOCK_STREAM, 0);

memset(&addr, 0, sizeof(addr));
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
int fl = fcntl(s, F_GETFL, 0);
fcntl(s, F_SETFL, fl | O_NONBLOCK);

if (s < 0)
{
   cout << "socket() error function" << endl;
   return 0;
}
if (bind(s, (sockaddr *)&addr, sizeof(addr)) < 0)
{
   cout << "bind() function error" << endl;
   return 0;
}
```

```
if (listen(s, 1) < 0)
{
   cout << "listen() function error" << endl;
   return -1;
}
for (size_t i = 0; i < MAX_SOCKETS; i++)
{
   cs[i] = -1;
}
pfd[0].fd = s;
pfd[0].events = POLLIN;

cout << "Listening in " << port << endl;
while (true)
{
   int res = poll(pfd, sizeof(pfd) / sizeof(pfd[0]), 100);
   if (res > 0)
   {
      for (size_t i = 1; i <= s_c; i++)
      {
         if (pfd[i].revents & POLLIN)
         {
            char *buffer = (char *)malloc(4);
            int r = recv(pfd[i].fd, buffer, 4, 0);
            if (strcmp(buffer, "put") == 0)
               break;
            else
            {
               int r_status = readPacket(pfd[i].fd, &addr, i);
```

```
            if (r_status == -1)
            {
                cout << "stop message detected. Powering off..." << endl;
                for (size_t i = 1; i <= s_c; i++)
                {
                    close(pfd[i].fd);
                }
                close(s);
                exit(0);
            }
        }
    }
    if (pfd[i].revents & POLLHUP || pfd[i].revents & POLLERR)
    {
        disconnectClient(i, &addr);
    }
}
if (pfd[0].revents & POLLIN)
{

    unsigned long mode = 1;
    unsigned int socklen = sizeof(addr);
    int new_connection = accept(pfd[0].fd, (struct sockaddr *)&addr, &socklen);
    string client = findClient(&addr);
    cout << "  Peer connected: " << client << endl;
    if (new_connection < 0)
    {
        break;
    }
```

```
            s_c++;
            for (size_t i = 1; i <= s_c; i++)
            {
               if (cs[i] == -1)
               {
                  if (i == s_c) s_c--;
                  cs[i] = new_connection;
                  pfd[i].fd = new_connection;
                  pfd[i].events = POLLIN | POLLOUT;
                  break;
               }
            }
         }
      }
   }
   for (size_t i = 0; i < s_c; i++)
   {
      close(pfd[i].fd);
   }
   close(s);
   return 0;
}
```

c. UDP-клиент

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <map>
#include <vector>
```

```
#include <set>
#include <iomanip>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <arpa/inet.h>
#include <stdint.h>

#define MAX_BODY_SIZE 65507
#define MAX_CONNECTIONS 65535
using namespace std;

map<uint32_t, string> allMsgs;
set<uint32_t> msgArchive;

set<uint32_t> recvData(int s, int sendCount);
string datagramCreate(uint32_t index);
static int sendData(int s, sockaddr_in *addr, set<uint32_t> msgNums);
void readFile(char *file_name);

int main(int argc, char **argv)
{
   if (argc != 3)
   {
      cout << "Usage: ./udpclient IP:PORT FILENAME" << endl;
      return 0;
   }
   char *ip = strtok(argv[1], ":");
```

```
    char *port = strtok(NULL, "\0");
    cout << ip << " " << port << endl;
    sockaddr_in addr;
    int s = socket(AF_INET, SOCK_DGRAM, 0);
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip);
    addr.sin_port = htons(atoi(port));

    readFile(argv[2]);
    set <uint32_t> startMsgs;
    set <uint32_t> memSave;
    for (auto it = allMsgs.begin(); it != allMsgs.end(); it++)
    {
        startMsgs.insert(it->first);
    }
    int sendCount = 0;
    sendCount = sendData(s, &addr, startMsgs);
    memSave = startMsgs;
    while ((allMsgs.size() <= 20 && msgArchive.size() != allMsgs.size()) || (allMsgs.size() > 20 &&
msgArchive.size() != 20))
    {
        set<uint32_t> newData = recvData(s, sendCount);
        if (newData.size() == 0)
        {
            newData = memSave;
        }
        sendCount = sendData(s, &addr, newData);
        memSave = newData;
```

```
    }
}

void readFile(char *file_name)
{
    ifstream fdesc(file_name, ios::in);
    string fline;
    uint32_t index = 0;
    while(getline(fdesc, fline))
    {
        if (!fline.empty())
        {
            allMsgs.insert(make_pair(index, fline));
            index++;
        }
        fline.clear();
    }
}

void append_chunk(string &s, const uint8_t *chunk, size_t chunk_num_bytes)
{
    s.append((char *)chunk, chunk_num_bytes);
}

string datagramCreate(uint32_t index)
{
    string payload;
    string msgInfo = allMsgs[index].substr(0, allMsgs[index].find('+') + 12), token;
    string msgData = allMsgs[index].substr(allMsgs[index].find('+') + 13, allMsgs[index].size());
```

```
vector <string> Tokens;
istringstream _stream(msgInfo);
while (getline(_stream, token, ' '))
{
   if (!token.empty())
   {
      Tokens.push_back(token);
   }
   token.clear();
}
string date = Tokens[0];
istringstream _datestream(date);
vector <string> dateParts;
while(getline(_datestream, token, '.'))
{
   if (!token.empty())
   {
      dateParts.push_back(token);
      token.clear();
   }
}
/* Data processing */
uint32_t msgIndex = htonl(index);
uint8_t day = atoi(dateParts[0].c_str());
uint8_t month = atoi(dateParts[1].c_str());
uint16_t year = htons(atoi(dateParts[2].c_str()));
int16_t AA = htons(atoi(Tokens[1].c_str()));

append_chunk(payload, (const uint8_t *)&msgIndex, sizeof(uint32_t));
```

```
    append_chunk(payload, (const uint8_t *)&day, sizeof(uint8_t));
    append_chunk(payload, (const uint8_t *)&month, sizeof(uint8_t));
    append_chunk(payload, (const uint8_t *)&year, sizeof(uint16_t));
    append_chunk(payload, (const uint8_t *)&AA, sizeof(int16_t));
    append_chunk(payload, (const uint8_t *)Tokens[2].c_str(), 12);
    append_chunk(payload, (const uint8_t *)msgData.c_str(), msgData.size());
    return payload;
}

set <uint32_t> recvData(int s, int sendCount)
{
    struct timeval tv = {0, 100 * 1000};
    fd_set fds;
    FD_ZERO(&fds);
    FD_SET(s, &fds);
    set <uint32_t> nums;
    int32_t result = select(s + 1, &fds, 0, 0, &tv);

    if (result > 0)
    {
        sockaddr_in addr;
        socklen_t addrLen = sizeof(addr);
        uint32_t currentNums[20];
        memset(currentNums, 0, 20 * sizeof(uint32_t));
        int i = 0, r;
        while(i != sendCount - 1)
        {
            r = recvfrom(s, &currentNums, 20 * sizeof(uint32_t), 0, (sockaddr *)&addr, &addrLen);
            i++;
```

```
        }
        r = recvfrom(s, &currentNums, 20 * sizeof(uint32_t), 0, (sockaddr *)&addr, &addrLen);
        if (r > 0)
        {
           for (size_t i = 0; i < (int) r / 4; i++)
           {

              uint32_t rawInt = ntohl(currentNums[i]);
              if (msgArchive.find(rawInt) == msgArchive.end())
              {
                 msgArchive.insert(rawInt);
              }
           }
           for (size_t i = 0; i < allMsgs.size(); i++)
           {
              if (msgArchive.find(i) == msgArchive.end())
              {
                 nums.insert(i);
                 if (nums.size() == 20) break;
              }
           }
        }
     }
     return nums;
}

static int sendData(int s, sockaddr_in *addr, set <uint32_t> msgNums)
{
     cout << msgNums.size() << " message(s) in queue.." << endl;
```

```
    for (auto it = msgNums.begin(); it != msgNums.end(); it++)
    {
        string buffer = datagramCreate(*it);
        buffer.reserve(allMsgs[*it].size() + 4);
        int status = sendto(s, buffer.c_str(), buffer.size() + 1, 0, (sockaddr *)addr, sizeof(sockaddr));
    }
    return msgNums.size();
}
```

d. UDP-сервер

```
#define _CRT_SECURE_NO_WARNINGS
#define WIN32_LEAN_AND_MEAN
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <windows.h>
#include <winsock2.h>
#include <WS2tcpip.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <map>
#include <set>
#include <iomanip>
#include <string.h>
#include <stdint.h>
#pragma comment(lib, "ws2_32.lib")

#define USHORT_MAX 0xffff
#define UDP_MAX_LEN 65507
#define FILE_LOGGER "msg.txt"
```

```
using namespace std;

map<string, set<uint32_t, greater<uint32_t>>> clients;
SOCKET *sockets_desc = new SOCKET[100];
struct sockaddr_in *addr = new struct sockaddr_in[100];

uint16_t last_index;
bool stop_hook = false;

union
{
   uint32_t integer;
   char buffer[4];
} convert;

union
{
   char buffer[2];
   uint16_t integer;
   int16_t ninteger;
} short_ints;

string findClient(struct sockaddr_in *addr)
{
   char ip[16];
   char port[5];
   string client;
   inet_ntop(AF_INET, &addr->sin_addr, ip, sizeof(ip));
   client.clear();
```

```
    _itoa(htons(addr->sin_port), port, 10);
    client.append((char *)ip, strlen(ip));
    client += ':';
    client.append((char *)port, strlen(port));
    return client;
}

int writeToFile(string buffer, string client)
{
    ofstream file_desc;
    file_desc.open(FILE_LOGGER, ios::app);
    uint16_t day, month, year = 0;
    day = buffer[0];
    month = buffer[1];
    short_ints.buffer[0] = buffer[3];
    short_ints.buffer[1] = buffer[2];
    year = short_ints.integer;
    file_desc << client << " ";
    file_desc << setw(2) << setfill('0') << day << '.' << setw(2) << setfill('0') << month << '.' << year << " ";

    short_ints.buffer[0] = buffer[5];
    short_ints.buffer[1] = buffer[4];
    int16_t AA = short_ints.ninteger;
    file_desc << AA << " ";
    file_desc << string(buffer.begin() + 6, buffer.begin() + 18) << " " << string(buffer.begin() + 18,
buffer.end() -1) << endl;
    file_desc.close();
    if (string(buffer.begin() + 18, buffer.end() -1) == "stop")
    {
```

```
        return -1;
    }
    return 0;
}

int analyseDatagram(SOCKET sock, int index)
{
    char *buf_input = new char[UDP_MAX_LEN];
    socklen_t addrLen = sizeof(addr[index]);
    int status = recvfrom(sock, buf_input, UDP_MAX_LEN, 0, (struct sockaddr *)&addr[index], &addrLen);
    char num_bytes[4];
    int number = 0, result = 0, i;
    string buf;
    string client;
    buf.append((char *)buf_input, status);
    for (i = 0; i < 3; i++)
    {
        number += buf_input[i] & 0xff;
        number = (number << 8);
    }
    number += buf_input[i] & 0xff;
    if (status > 0)
    {
        set<uint32_t, greater<uint32_t>> msgs;
        client = findClient(&addr[index]);
        if (clients.find(client) == clients.end())
        {
            cout << "New client connected: " << client << endl;
            clients.insert(make_pair(client, msgs));
```

```
        }
        if (clients[client].find(number) == clients[client].end())
        {
            clients[client].insert(number);
            result = writeToFile(string(buf.begin() + 4, buf.end()), client);
        }
    }
    buf.clear();
    delete[] buf_input;
    return (result == -1 ? -1 : 0);
}

void configListeners(uint16_t port_start, uint16_t port_end)
{
    uint16_t ports_count = 0;
    uint16_t step;
    if (port_start > port_end)
    {
        uint16_t tmp = port_start;
        port_start = port_end;
        port_end = tmp;
    }
    ports_count = port_end - port_start;
            if (port_end == port_start)
            {
                        ports_count = 1;
            }
    cout << "Listening on ports: ";
    for (step = 0; step <= ports_count; step++)
```

```
    {
        unsigned long mode = 1;
        SOCKET s = socket(AF_INET, SOCK_DGRAM, 0);
        if (s < 0) {
            cout << "socket() func error " << WSAGetLastError() << endl;
            return;
        }
        sockets_desc[step] = s;
        ioctlsocket(sockets_desc[step], FIONBIO, &mode);
        memset(&addr[step], 0, sizeof(addr[step]));
        addr[step].sin_family = AF_INET;
        addr[step].sin_addr.s_addr = htonl(INADDR_ANY);
        addr[step].sin_port = htons(port_start + step);
        if (bind(sockets_desc[step], (struct sockaddr *)&addr[step], sizeof(addr[step])) < 0)
        {
            cout << "bind() func error " << WSAGetLastError() << endl;
            return;
        }
        cout << port_start + step << ' ';
    }
    cout << endl;
}

void sendToClient(SOCKET s, int index)
{
    string client = findClient(&addr[index]);
    char *buffer = new char[80];
    memset(buffer, 0, 80);
    int i = 0;
```

```
    for (auto it = clients[client].begin(); it != clients[client].end(); it++)
    {
        convert.integer = htonl(*it);
        for (int j = 0; j < 4; j++)
        {
            buffer[i] = convert.buffer[j];
            i++;
        }
    }
    sendto(s, buffer, clients[client].size() * 4, 0, (struct sockaddr *)&addr[index], sizeof(addr[index]));
    delete[] buffer;
}

void disconnectClient(int index)
{
    string client = findClient(&addr[index]);
    clients.erase(client);
    cout << "Old client disconnected: " << client << endl;
}

int main(int argc, char **argv)
{
    if (argc != 3)
    {
        cout << "Usage: ./udpserver <port1> <port2>" << endl;
        return 0;
    }
    WSADATA ws;
    WSAStartup(MAKEWORD(2, 2), &ws);
```

```
uint16_t port1, port2, ports_count;
port1 = atoi(argv[1]);
port2 = atoi(argv[2]);
ports_count = port2 - port1;
configListeners(port1, port2);
WSAEVENT *event = new WSAEVENT[ports_count];
for (int i = 0; i < ports_count; i++)
{
                event[i] = WSACreateEvent();
   WSAEventSelect(sockets_desc[i], event[i], FD_READ | FD_WRITE | FD_CLOSE);
}
while (true)
{
                WSANETWORKEVENTS ne;
   DWORD dw = WSAWaitForMultipleEvents(ports_count, event, FALSE, 1000, FALSE);
                DWORD res = 0;
   for (int i = 0; i < ports_count; i++)
   {
      if ((res = WSAEnumNetworkEvents(sockets_desc[i], event[i], &ne)) == 0)
      {
         if (ne.lNetworkEvents & FD_READ)
         {
            int value = analyseDatagram(sockets_desc[i], i);
            sendToClient(sockets_desc[i], i);
            if (value == -1)
            {
               cout << "stop message detected. Power off...." << endl;
                                                for (int i = 0; i < ports_count; i++)
                                                {
```

```
                                        WSACloseEvent(event[i]);
                                        closesocket(sockets_desc[i]);
                                    }

                    WSACleanup();
                    delete[] sockets_desc;
                    return 0;
                  }
                }
                if (ne.lNetworkEvents & FD_CLOSE)
                {
                  disconnectClient(i);
                }
              }
            }
          }
          return 0;
}
```