МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №8

по дисциплине: Компьютерные сети
тема: «Программирование протокола HTTP»

Выполнил: ст. группы ПВ-223
Игнатьев Артур Олегович

Проверили:
Рубцов Константин Анатольевич

Белгород 2025 г.

Цель работы: изучить протокол HTTP и составить программу согласно заданию.

Краткие теоретические сведения

HTTP (Hyper Text Transfer Protocol – протокол передачи гипертекста) – протокол прикладного уровня стека протоколов TCP/IP, предназначенный для передачи данных по сети с использованием транспортного протокола TCP. Текущая версия протокола HTTP v1.1, его спецификация приводится в документе RFC 2616.

Протокол HTTP может использоваться также в качестве «транспорта» для других протоколов прикладного уровня, таких как SOAP или XML-RPC. Основой HTTP является технология «клиент-сервер». HTTP-клиенты отсылают HTTP-запросы, которые содержат метод, обозначающий потребность клиента. Также такие запросы содержат универсальный идентификатор ресурса, указывающий на желаемый ресурс. Обычно такими ресурсами являются хранящиеся на сервере файлы. По умолчанию HTTP запросы передаются на порт 80. HTTP-сервер отсылает коды состояния, сообщая, успешно ли выполнен HTTP-запрос или же нет.

Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т.д. Именно благодаря возможности указания способа кодирования сообщения клиент и сервер могут обмениваться двоичными данными, хотя данный протокол является текстовым.

Унифицированный идентификатор ресурса представляет собой сочетание унифицированного указателя ресурса (Uniform Resource Locator, URL) и унифицированного имени ресурса (Uniform Resource Name, URN). Например:

URI = http://iitus.bstu.ru/to_schoolleaver/230400

URL= http://iitus.bstu.ru/

URN= to_schoolleaver/230400

Метод протокола HTTP – это команда, передаваемая HTTP-клиентом HTTP серверу. В табл. 8.1. перечислены некоторые методы, определенные в протоколе HTTP v1.1. Полный список методов HTTP v1.1. содержится в документе RFC 2616.

Основные методы HTTP v1.1 Метод	Назначение
GET	Используется для запроса содержимого ресурса, на который указывает URL, содержащийся в запросе
HEAD	Используется для извлечения метаданных или проверки наличия ресурса, на который указывает URL, содержащийся в запросе
POST	Применяется для передачи данных заданному ресурсу. Данный метод предполагает, что по указанному URL будет производиться обработка передаваемого клиентом содержимого
PUT	Применяется для передачи данных заданному ресурсу. Данный метод предполагает, что передаваемое клиентом содержимое соответствует находящемуся по данному URL ресурсу
OPTIONS	Используется для определения возможностей HTTP сервера или параметров соединения для конкретного ресурса
DELETE	Применяется для удаления ресурса, на который указывает URL

Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. Название метода чувствительно к регистру. Каждый сервер обязан поддерживать как минимум методы GET и HEAD. Если сервер не распознал указанный клиентом метод, то он должен вернуть статус 501 (NotImplemented). Если серверу метод известен, но он неприменим к конкретному ресурсу, то возвращается сообщение с кодом 405 (MethodNotAllowed). В обоих случаях серверу следует включить в сообщение ответа заголовок Allow со списком поддерживаемых методов.

Код состояния HTTP представляет собой целое число из трех цифр. Первая цифра указывает на класс состояния:

• информационные сообщения;

• успешное выполнение;

• переадресация;

• ошибка клиента;

• ошибка сервера.

Полный список статусов HTTP v1.1. содержится в документе RFC 2616.

Примеры:

201 Webpage Created

403 Access allowed only for registered users

Каждое HTTP-сообщение состоит из трех частей, которые передаются в следующем порядке:

1. Стартовая строка – определяет тип сообщения;

2. Заголовки – характеризуют тело сообщения, параметры передачи и прочие сведения;

3. Тело сообщения – непосредственно данные сообщения.

Стартовые строки HTTP-сообщения различаются для запроса и ответа.

Стартовая строка HTTP-запроса имеет следующий формат:

Метод URL HTTP/Версия, где метод - название запроса, URL определяет путь к запрашиваемому документу, версия - пара разделённых точкой арабских цифр. Стартовая строка HTTP-ответа имеет следующий формат: HTTP/Версия КодСостояния Пояснение. Заголовок HTTP представляет собой строку в HTTP-сообщении, содержащую разделенную двоеточием пару параметр-значение. Заголовки должны отделяться от тела сообщения хотя бы одной пустой строкой. Все заголовки разделяются на четыре основных группы:

1. Основные заголовки (General Headers) – должны включаться в любое сообщение клиента и сервера.

2. Заголовки запроса (Request Headers) – используются только в запросах клиента.

3. Заголовки ответа (Response Headers) – только для ответов от сервера.

4. Заголовки сущности (Entity Headers) – сопровождают каждую сущность сообщения.

Полный список заголовков HTTP v1.1. содержится в документе RFC 2616.

Примеры заголовков:

Content-Type: text/plain; charset=windows-1251

Content-Language: ru

Тело HTTP-сообщения, если оно присутствует, используется для передачи данных, связанных с запросом или ответом.

Чтобы понять, как работает протокол HTTP, рассмотрим пример получения

HTML-страницы с HTTP-сервера.

HTTP-запрос:

GET /to_schoolleaver/230400 HTTP/1.1 Host: iitus.bstu.ru User-Agent:

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0) Accept:

text/html Accept-Language: ru Connection: close

HTTP-ответ:

HTTP/1.1 200 OK Date: Fri, 16 Dec 2012 13:45:00 GMT Server: Apache Last

Modified: Wed, 11 Feb 2009 11:20:59 GMT Content-Language: ru

Content-Type: text/html; charset=utf-8 Content-Length: 1234 Connection: close (далее следует запрошенная HTML-страница)

Протокол HTML позволяет достаточно легко создавать клиентские приложения. Возможности протокола можно расширить благодаря внедрению своих собственных заголовков, с помощью которых можно получить необходимую функциональность при решении нетривиальной задачи. При этом сохраняется совместимость с другими клиентами и серверами: они будут просто игнорировать неизвестные им заголовки.

Протокол HTTP устанавливает отдельную TCP-сессию на каждый запрос; в более поздних версиях HTTP было разрешено делать несколько запросов в ходе одной TCP-сессии, но браузеры обычно запрашивают только страницу и включённые в неё объекты (картинки, каскадные стили и т. п.), а затем сразу разрывают TCP-сессию. Для поддержки авторизованного (неанонимного) доступа в HTTP используются cookies; причём такой

способ авторизации позволяет сохранить сессию даже после перезагрузки клиента и сервера.

При доступе к данным по FTP или по файловым протоколам тип файла (точнее, тип содержащихся в нём данных) определяется по расширению имени файла, что не всегда удобно. HTTP перед тем, как передать сами данные, передаёт заголовок «Content-Type: тип/подтип», позволяющую клиенту однозначно определить, каким образом обрабатывать присланные данные. Это особенно важно при работе с CGI-скриптами, когда расширение имени файла указывает не на тип присылаемых клиенту данных, а на необходимость запуска данного файла на сервере и отправки клиенту результатов работы программы, записанной в этом файле (при этом один и тот же файл в зависимости от аргументов запроса и своих собственных соображений может порождать ответы разных типов — в простейшем случае картинки в разных форматах).

Кроме того, HTTP позволяет клиенту прислать на сервер параметры, которые будут переданы запускаемому CGI-скрипту. Для этого же в HTML были введены формы.

Перечисленные особенности HTTP позволили создавать поисковые машины (первой из которых стала AltaVista, созданная фирмой DEC), форумы и Internet-магазины. Это коммерциализировало Интернет, появились компании, основным полем деятельности которых стало предоставление доступа в Интернет (провайдеры) и создание сайтов.

В данной лабораторной работе необходимо использовать функции работы с Winsock из лабораторной работы №4. Подробное их описание можно найти на MSDN.

Анализ применяемых функций

Перевод сокета в состояние “прослушивания” (для TCP) осуществляется функцией listen (SOCKET s, int backlog), где s – дескриптор сокета; backlog – максимальный размер очереди входящих сообщений на соединение. Используется сервером, чтобы информировать ОС, что он ожидает (“слушает”) запросы связи на данном сокете. Без этой функции всякое требование связи с сокетом будет отвергнуто.

• Функция connect (SOCKET s, const struct sockaddr FAR* name, int namelen) нужна для соединения с сокетом, находящимся в состоянии “прослушивания” (для TCP). Она ипользуется процессомклиентом для установления связи с сервером. В случае успешного

установления соединения connect возвращает 0, иначе SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

• Функция accept (SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen) служит для подтверждения запроса на соединение (для TCP). Функция используется для принятия связи на сокет. Сокет должен быть уже слушающим в момент вызова функции. Если сервер устанавливает связь с клиентом, то данная функция возвращает новый сокет дескриптор, через который и производит общение клиента с сервером. Пока устанавливается связь клиента с сервером, функция блокирует другие запросы связи с данным сервером, а после установления связи "прослушивание" запросов возобновляется.

• В случае автоматического распределения адресов и портов узнать какой адрес и порт присвоен сокету можно при помощи функции getsockname (SOCKET s, struct sockaddr FAR* name, int FAR* namelen). Если операция выполнена успешно, возвращает 0, иначе возвращает SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

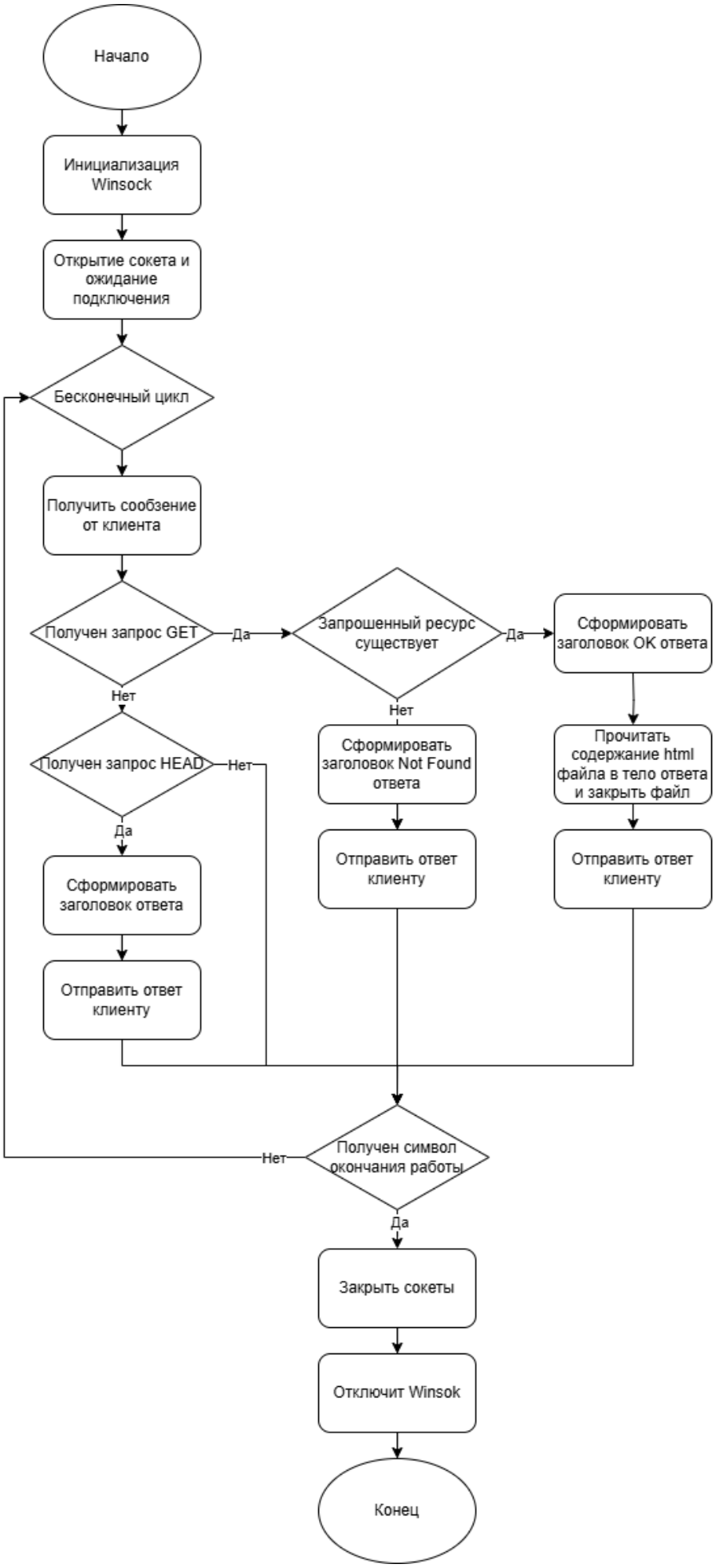
• Для передачи данных по протоколу UDP используется функция sendto (SOCKET s, const char FAR * buf, int len, int flags, const struct sockaddr FAR * to, int tolen). Если операция выполнена успешно, возвращает количество переданных байт, иначе возвращает SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

• Для передачи данных по протоколу TCP используется функция send (SOCKET s, const char FAR * buf, int len, int flags), где s - дескриптор сокета; buf - указатель на буфер с данными, которые необходимо переслать; len - размер (в байтах) данных, которые содержатся по указателю buf; flags - совокупность флагов, определяющих, каким образом будет произведена передача данных. Если операция выполнена успешно, возвращает количество переданных байт, иначе возвращает SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

• Для приема данных по протоколу UDP используется функция recvfrom (SOCKET s, char FAR* buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen). Если операция выполнена успешно, возвращает количество полученных байт, иначе возвращает SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.

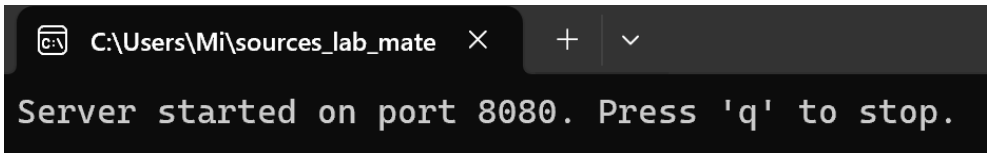
• Для приема данных по протоколу TCP используется функция recv (SOCKET s, char FAR* buf, int len, int flags). Если операция выполнена успешно, возвращает количество полученных байт, иначе возвращает SOCKET_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError

Разработка программы. Блок-схемы программы



Анализ функционирования программы

Работа программы



Программа работает корректно и предоставляет доступ к сайтам и умеет отвечать на GET и HEAD запросы. Тестовый сайт отображается корректно.

Вывод: на этой лабораторной работе был изучен протокол HTTP и составлена программа согласно заданию.

Код программы:

Файл HTTP.cpp

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <string>
#include <sstream>
#include <fstream>
#include <iostream>
#include <conio.h>

// Класс простого HTTP-сервера
class HttpServer {
private:
    static constexpr int PORT = 8080;                  // Порт сервера
    static constexpr int BUFFER_SIZE = 8192;           // Размер буфера для получения запроса
    static constexpr const char* DEFAULT_PAGE = "index.html"; // Страница по умолчанию
    SOCKET listen_socket = INVALID_SOCKET;             // Слушающий сокет

    // Структура, представляющая HTTP-запрос
    struct Request {
        std::string method; // Метод (GET, HEAD)
        std::string path;   // Путь к запрашиваемому ресурсу
        bool valid;         // Валидность запроса

        Request() : valid(false) {}
    };

    // Инициализация Winsock (сетевой подсистемы Windows)
    bool init_winsock() {
        WSADATA wsa_data;
        if (WSAStartup(MAKEWORD(2, 2), &wsa_data) != 0) {
            std::cerr << "Winsock initialization failed: " << WSAGetLastError() << std::endl;
            return false;
        }
        return true;
    }

    // Создание и настройка сокета для прослушивания подключений
    bool setup_socket() {
        addrinfo hints{}, *result = nullptr;
        hints.ai_family = AF_INET;        // IPv4
        hints.ai_socktype = SOCK_STREAM;  // TCP
        hints.ai_protocol = IPPROTO_TCP;
        hints.ai_flags = AI_PASSIVE;      // Используется для bind

        // Получаем информацию о локальном адресе (127.0.0.1:8080)
        if (getaddrinfo("127.0.0.1", std::to_string(PORT).c_str(), &hints, &result) != 0) {
            std::cerr << "Failed to resolve address: " << WSAGetLastError() << std::endl;
            return false;
        }

        // Создание сокета
        listen_socket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
        if (listen_socket == INVALID_SOCKET) {
            std::cerr << "Socket creation failed: " << WSAGetLastError() << std::endl;
            freeaddrinfo(result);
            return false;
        }

        // Привязка сокета к адресу
        if (bind(listen_socket, result->ai_addr, static_cast<int>(result->ai_addrlen)) ==
SOCKET_ERROR) {
            std::cerr << "Bind failed: " << WSAGetLastError() << std::endl;
            freeaddrinfo(result);
```

```
        closesocket(listen_socket);
        return false;
    }

    freeaddrinfo(result); // Освобождение памяти после использования адреса

    // Прослушивание подключений
    if (listen(listen_socket, SOMAXCONN) == SOCKET_ERROR) {
        std::cerr << "Listen failed: " << WSAGetLastError() << std::endl;
        closesocket(listen_socket);
        return false;
    }

    return true;
}

// Разбор строки HTTP-запроса
Request parse_http_request(const std::string& raw_request) {
    Request req;
    std::istringstream stream(raw_request);
    std::string first_line;

    // Извлекаем первую строку (например: GET / HTTP/1.1)
    if (!std::getline(stream, first_line)) return req;

    std::istringstream line_stream(first_line);
    std::string method, path, version;

    // Разбор метода, пути и версии протокола
    if (!(line_stream >> method >> path >> version)) return req;

    // Проверка поддержки метода и протокола
    if ((method == "GET" || method == "HEAD") && version.find("HTTP/") == 0) {
        req.method = method;
        req.path = path == "/" ? DEFAULT_PAGE : path.substr(1); // Убираем слэш в начале
        req.valid = true;
    }
    return req;
}

// Загрузка содержимого файла по указанному пути
std::string load_file(const std::string& path) {
    std::ifstream file(path, std::ios::binary);
    if (!file.is_open()) return "";

    std::ostringstream content;
    content << file.rdbuf();
    return content.str();
}

// Построение HTTP-ответа на основе запроса и содержимого
std::string build_response(const Request& req, const std::string& raw_request) {
    std::ostringstream response;
    std::string body;

    std::string file_content = load_file(req.path);
    if (file_content.empty()) {
        // Если файл не найден — ошибка 404
        body = "<html><body><h2>404 Not Found</h2><p>File not found.</p></body></html>";
        response << "HTTP/1.1 404 Not Found\r\n"
                 << "Content-Type: text/html; charset=utf-8\r\n"
                 << "Content-Length: " << body.size() << "\r\n"
                 << "Connection: close\r\n\r\n";
        if (req.method == "GET") response << body;
    } else {
        // Если файл найден — возвращаем содержимое и отладочную информацию
        body = file_content + "<h2>Request Details</h2><pre>" + raw_request + "</pre>";
```

```
            response << "HTTP/1.1 200 OK\r\n"
                     << "Content-Type: text/html; charset=utf-8\r\n"
                     << "Content-Length: " << body.size() << "\r\n"
                     << "Connection: close\r\n\r\n";
            if (req.method == "GET") response << body;
        }

        return response.str();
    }

    // Обработка одного клиента
    void process_client(SOCKET client_socket) {
        char buffer[BUFFER_SIZE] = {0};

        // Получение запроса от клиента
        int bytes_received = recv(client_socket, buffer, BUFFER_SIZE - 1, 0);
        if (bytes_received == SOCKET_ERROR) {
            std::cerr << "Receive error: " << WSAGetLastError() << std::endl;
            closesocket(client_socket);
            return;
        }
        if (bytes_received == 0) {
            // Клиент закрыл соединение
            closesocket(client_socket);
            return;
        }

        // Обработка запроса
        std::string request(buffer, bytes_received);
        Request req = parse_http_request(request);

        // Формируем ответ
        std::string response = req.valid ?
            build_response(req, request) :
            "HTTP/1.1 400 Bad Request\r\nContent-Type: text/html; charset=utf-8\r\nContent-
Length: 15\r\nConnection: close\r\n\r\nBad Request";

        // Отправка ответа клиенту
        if (send(client_socket, response.c_str(), response.size(), 0) == SOCKET_ERROR) {
            std::cerr << "Send error: " << WSAGetLastError() << std::endl;
        }

        // Закрытие соединения
        closesocket(client_socket);
    }

public:
    // Запуск сервера
    bool start() {
        if (!init_winsock() || !setup_socket()) {
            return false;
        }

        std::cout << "Server started on port " << PORT << ". Press 'q' to stop." << std::endl;

        // Основной цикл ожидания клиентов
        while (true) {
            SOCKET client_socket = accept(listen_socket, nullptr, nullptr);
            if (client_socket == INVALID_SOCKET) {
                std::cerr << "Accept error: " << WSAGetLastError() << std::endl;
                continue;
            }

            // Обработка подключения
            process_client(client_socket);

            // Прерывание работы сервера по клавише 'q'
```

```
            if (_kbhit() && _getch() == 'q') {
                break;
            }
        }

        // Очистка ресурсов
        closesocket(listen_socket);
        WSACleanup();
        return true;
    }
};

int main() {
    HttpServer server;
    if (!server.start()) {
        return 1;
    }
    return 0;
}
```

Файл site.html

```
<html>
<head><title>Welcome</title></head>
<body><h1>Hello from HTTP Server!</h1></body>
</html>
```