https://code-live.ru/post/cpp-http-server-over-sockets/

Веб-сервер на C++ и сокетах

- Создадим HTTP-сервер, который обрабатывает запросы браузера и возвращает ответ в виде HTML-страницы.
- Введение в HTTP
- Что будет делать сервер?
- О сокетах
- Создание сокета
- Привязка сокета к адресу (bind)
- Подготовка сокета к принятию входящих соединений (listen)
- Ожидание входящего соединения (accept)
- Получение запроса и отправка ответа
- Последовательная обработка запросов

Введение в HTTP

- **Введение в HTTP**
- Для начала разберемся, что из себя представляет HTTP. Это текстовый протокол для обмена данными между браузером и веб-сервером.
- **Пример HTTP-запроса:**
- **GET /page.html HTTP/1.1** Host: site.com Первая строка передает метод запроса, идентификатор ресурса (URI) и версию HTTP-протокола. Затем перечисляются заголовки запроса, в которых браузер передает имя хоста, поддерживаемые кодировки, cookie и другие служебные параметры. После каждого заголовка ставится символ переноса строки \r\n.
- У некоторых запросов есть тело. Когда отправляется форма методом POST, в теле запроса передаются значения полей этой формы.
- `POST /submit HTTP/1.1 Host site.com Content-Type: application/x-www-form-urlencoded name=Sergey&last_name=Ivanov&birthday=1990-10-05`
- Тело запроса отделяется от заголовков одной пустой строкой. Заголовок «Content-Type» говорит серверу, в каком формате закодировано тело запроса. По умолчанию, в HTML-форме данные кодируются методом «application/x-www-form-urlencoded».

- Иногда необходимо передать данные в другом формате. Например, при загрузке файлов на сервер, бинарные данные кодируются методом «multipart/form-data».
- Сервер обрабатывает запрос клиента и возвращает ответ.
- **Пример ответа сервера:**
- **HTTP/1.1 200 OK** Host: site.com Content-Type: text/html; charset=UTF-8 Connection: close Content-Length: 21 <h1>Test page...</h1>
- В первой строке ответа передается версия протокола и статус ответа. Для успешных запросов обычно используется статус «200 OK». Если ресурс не найден на сервере, возвращается «404 Not Found».
- Тело ответа так же, как и у запроса, отделяется от заголовков одной пустой строкой.
-

- Полная спецификации протокола HTTP описывается в стандарте rfc-2068. По понятным причинам, мы не будем реализовывать все возможности протокола в рамках этого материала. Достаточно реализовать поддержку работы с заголовками запроса и ответа, получение метода запроса, версии протокола и URL-адреса.

Что будет делать сервер?

- Сервер будет принимать запросы клиентов, парсить заголовки и тело запроса, и возвращать тестовую HTML-страничку, на которой отображены данные запроса клиента (запрошенный URL, метод запроса, cookie и другие заголовки).

Test page

This is body of the test page...

Request headers

```
GET / HTTP/1.1

Host: localhost:8000
Accept-Encoding: gzip, deflate, sdch
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Connection: keep-alive
Cache-Control: no-cache
Pragma: no-cache
Accept-Language: ru,en-US;q=0.8,en;q=0.6
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.90 Safari/537.36
```

О сокетах

- Для работы с сетью на низком уровне традиционно используют сокеты. Сокет — это абстракция, которая позволяет работать с сетевыми ресурсами, как с файлами. Мы можем писать и читать данные из сокета почти так же, как из обычного файла.
- В этом материале мы будем работать с виндовой реализацией сокетов, которая находится в заголовочном файле <WinSock2.h>.
- В Unix-подобных ОС принцип работы с сокетами такой же, только отличается API. Вы можете подробнее почитать о сокетах Беркли, которые используются в GNU/Linux.

Создание сокета

- Создадим сокет с помощью функции socket, которая находится в заголовочном файле <WinSock2.h>. Для работы с IP-адресами нам понадобится заголовочный файл <WS2tcpip.h>.

```
#include <iostream>
#include <sstream>
#include <string> // Для корректной работы freeaddrinfo в MinGW
// Подробнее: http://stackoverflow.com/a/20306451 #define _WIN32_WINNT 0x501
#include <WinSock2.h>
#include <WS2tcpip.h>
// Необходимо, чтобы линковка происходила с DLL-библиотекой
// Для работы с сокетам
#pragma comment(lib, "Ws2_32.lib")
using std::cerr;
```

```cpp
int main()
{
    // служебная структура для хранение информации
    // о реализации Windows Sockets
    WSADATA wsaData;

    // старт использования библиотеки сокетов процессом
    // (подгружается Ws2_32.dll)
    int result = WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Если произошла ошибка подгрузки библиотеки
    if (result != 0) {
        cerr << "WSAStartup failed: " << result << "\n";
        return result;
    }
```

```
    if (result != 0) {
        cerr << "getaddrinfo failed: " << result << "\n";
        WSACleanup(); // выгрузка библиотеки Ws2_32.dll
        return 1;
    }

    // Создание сокета
    int listen_socket = socket(addr->ai_family, addr-
>ai_socktype,
        addr->ai_protocol);
    // Если создание сокета завершилось с ошибкой, выводим
сообщение,
    // освобождаем память, выделенную под структуру addr,
    // выгружаем dll-библиотеку и закрываем программу
    if (listen_socket == INVALID_SOCKET) {
        cerr << "Error at socket: " << WSAGetLastError() <<
"\n";
        freeaddrinfo(addr);
        WSACleanup();
        return 1;
    }

    // ...
```

- Мы подготовили все данные, которые необходимо для создания сокета и создали сам сокет. Функция socket возвращает целочисленное значение файлового дескриптора, который выделен операционной системой под сокет.

Привязка сокета к адресу (bind)

- Следующим шагом, нам необходимо привязать IP-адрес к сокету, чтобы он мог принимать входящие соединения. Для привязки конкретного адреса к сокету используется фукнция bind.
- Она принимает целочисленный идентификатор файлового дескриптора сокета, адрес (поле ai_addr из структуры addrinfo) и размер адреса в байтах (используется для поддержки IPv6).

```
// Привязываем сокет к IP-адресу
result = bind(listen_socket, addr->ai_addr, (int)addr-
>ai_addrlen);

// Если привязать адрес к сокету не удалось, то выводим
сообщение
// об ошибке, освобождаем память, выделенную под структуру addr.
// и закрываем открытый сокет.
// Выгружаем DLL-библиотеку из памяти и закрываем программу.
if (result == SOCKET_ERROR) {
    cerr << "bind failed with error: " << WSAGetLastError() <<
"\n";
    freeaddrinfo(addr);
    closesocket(listen_socket);
    WSACleanup();
    return 1;
}
```

Подготовка сокета к принятию входящих соединений (listen)

- Подготовим сокет к принятию входящих соединений от клиентов. Это делается с помощью функции listen. Она принимает дескриптор слушающего сокета и максимальное количество одновременных соединений.
- В случае ошибки, функция listen возращает значение константы SOCKET_ERROR. При успешном выполнении она вернет 0.

```
// Инициализируем слушающий сокет
if (listen(listen_socket, SOMAXCONN) == SOCKET_ERROR)
{ cerr << "listen failed with error: " <<
WSAGetLastError() << "\n";
closesocket(listen_socket);
WSACleanup();
 return 1;
}
```

- В константе SOMAXCONN хранится максимально возможное число одновременных TCP-соединений. Это ограничение работает на уровне ядра ОС.

Ожидание входящего соединения (accept)

- Функция accept ожидает запрос на установку TCP-соединения от удаленного хоста. В качестве аргумента ей передается дескриптор слушающего сокета.
- При успешной установке TCP-соединения, для него создается новый сокет. Функция accept возвращает дескриптор этого сокета. Если произошла ошибка соединения, то возвращается значение INVALID_SOCKET.
- *// Принимаем входящие соединения*
- **int** client_socket = accept(listen_socket, NULL, NULL);
- **if** (client_socket == INVALID_SOCKET) {

  cerr << "accept failed: " << WSAGetLastError() << "\n";

  closesocket(listen_socket); WSACleanup();

  return 1;

  }

Получение запроса и отправка ответа

- После установки соединение с сервером, браузер отправляет HTTP-запрос. Мы получаем содержимое запроса через функцию recv. Она принимает дескриптор TCP-соединения (в нашем случае это client_socket), указатель на буфер для сохранения полученных данных, размер буфера в байтах и дополнительные флаги (которые сейчас нас не интересуют).
- При успешном выполнении функция recv вернет размер полученных данных. В случае ошибки возвращается значение SOCKET_ERROR. Если соединение было закрыто клиентом, то возвращается 0.
- Мы создадим буфер размером 1024 байта для сохранения HTTP-запроса

- `const int max_client_buffer_size = 1024;`
- `char buf[max_client_buffer_size];`
- `result = recv(client_socket, buf, max_client_buffer_size, 0);`
- `std::stringstream response;`
- `// сюда будет записываться ответ клиенту`
  `std::stringstream response_body;`
- `// тело ответа`

```
if (result == SOCKET_ERROR) {
    // ошибка получения данных
    cerr << "recv failed: " << result << "\n";
    closesocket(client_socket);
} else if (result == 0) {
    // соединение закрыто клиентом
    cerr << "connection closed...\n";
} else if (result > 0) {
    // Мы знаем фактический размер полученных данных, поэтому
ставим метку конца строки
    // В буфере запроса.
    buf[result] = '\0';

    // Данные успешно получены
    // формируем тело ответа (HTML)
    response_body << "<title>Test C++ HTTP Server</title>\n"
        << "<h1>Test page</h1>\n"
        << "<p>This is body of the test page...</p>\n"
        << "<h2>Request headers</h2>\n"
        << "<pre>" << buf << "</pre>\n"
        << "<em><small>Test C++ Http Server</small></em>\n";
```

```
    // Формируем весь ответ вместе с заголовками
    response << "HTTP/1.1 200 OK\r\n"
        << "Version: HTTP/1.1\r\n"
        << "Content-Type: text/html; charset=utf-8\r\n"
        << "Content-Length: " << response_body.str().length()
        << "\r\n\r\n"
        << response_body.str();

    // Отправляем ответ клиенту с помощью функции send
    result = send(client_socket, response.str().c_str(),
        response.str().length(), 0);

    if (result == SOCKET_ERROR) {
        // произошла ошибка при отправле данных
        cerr << "send failed: " << WSAGetLastError() << "\n";
    }
    // Закрываем соединение к клиентом
    closesocket(client_socket);
}
```

- После получения запроса мы сразу же отправили ответ клиенту с помощью функции send. Она принимает дескриптор сокета, строку с данными ответа и размер ответа в байтах.
- В случае ошибки, функция возвращает значение SOCKET_ERROR. В случае успеха — количество переданных байт.
- Попробуем скомпилировать программу, не забыв предварительно завершить функцию main.

```
// Убираем за собой
closesocket(listen_socket);
 freeaddrinfo(addr);
 WSACleanup();
return 0;
```

- Если скомпилировать и запустить программу, то окно консоли «подвиснет» в ожидании запроса на установление TCP-соединения. Откройте в браузере адрес http://127.0.0.1:8000/. Сервер вернет ответ, как на рисунке ниже и завершит работу.

Test page

This is body of the test page...

Request headers

```
GET / HTTP/1.1

Host: localhost:8000
Accept-Encoding: gzip, deflate, sdch
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Connection: keep-alive
Cache-Control: no-cache
Pragma: no-cache
Accept-Language: ru,en-US;q=0.8,en;q=0.6
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.90 Safari/537.36
```

Test C++ server

Последовательная обработка запросов

Чтобы сервер не завершал работу после обработки первого запроса, а продолжал обрабатывать новые соединения, нужно зациклить ту часть кода, которая принимает запрос на установку соединения и возвращает ответ.

```
const int max_client_buffer_size = 1024;
char buf[max_client_buffer_size];
int client_socket = INVALID_SOCKET;

for (;;) {
    // Принимаем входящие соединения
    client_socket = accept(listen_socket, NULL, NULL);
    if (client_socket == INVALID_SOCKET) {
        cerr << "accept failed: " << WSAGetLastError() << "\n";
        closesocket(listen_socket);
        WSACleanup();
        return 1;
    }

    result = recv(client_socket, buf, max_client_buffer_size,
0);
```

```
    std::stringstream response; // сюда будет записываться ответ
клиенту
    std::stringstream response_body; // тело ответа

    if (result == SOCKET_ERROR) {
        // ошибка получения данных
        cerr << "recv failed: " << result << "\n";
        closesocket(client_socket);
    } else if (result == 0) {
        // соединение закрыто клиентом
        cerr << "connection closed...\n";
    } else if (result > 0) {
        // Мы знаем размер полученных данных, поэтому ставим
метку конца строки
        // В буфере запроса.
        buf[result] = '\0';
```

```
        // Данные успешно получены
        // формируем тело ответа (HTML)
        response_body << "<title>Test C++ HTTP Server</title>\n"
            << "<h1>Test page</h1>\n"
            << "<p>This is body of the test page...</p>\n"
            << "<h2>Request headers</h2>\n"
            << "<pre>" << buf << "</pre>\n"
            << "<em><small>Test C++ Http Server</small></em>\n";

        // Формируем весь ответ вместе с заголовками
        response << "HTTP/1.1 200 OK\r\n"
            << "Version: HTTP/1.1\r\n"
            << "Content-Type: text/html; charset=utf-8\r\n"
            << "Content-Length: " <<
response_body.str().length()
            << "\r\n\r\n"
            << response_body.str();
```

```
        // Отправляем ответ клиенту с помощью функции send
        result = send(client_socket, response.str().c_str(),
            response.str().length(), 0);

        if (result == SOCKET_ERROR) {
            // произошла ошибка при отправле данных
            cerr << "send failed: " << WSAGetLastError() <<
"\n";
        }
        // Закрываем соединение к клиентом
        closesocket(client_socket);
    }
}
```

Когда сервер закончит обработку запроса одного клиента, он закроет соединение с ним и будет ожидать нового запроса.