

Final

by Scott Roberts

Submission date: 20-May-2025 01:36PM (UTC+0100)

Submission ID: 259917144

File name: 49384_Scott_Roberts_Final_664977_799671379.docx (939.41K)

Word count: 1111

Character count: 5820

Socket Programming & Concurrency: Web Server Report

Scott Roberts – 2306895

Contents

Introduction	3
Design Principles	3
Overview of Functionality.....	3
Speaking Hypertext Transfer Protocol	5
Code Walkthrough	6

Introduction

This report explains the features and design decisions of the application built to meet the requirements of the Socket Programming and Concurrency Implementation. The chosen application is a minimal implementation of a Hypertext Transfer Protocol server (RFC2616). The application is built using C++ and uses Object Oriented principles. The application uses code written by Tim Bashford (NetworkCore), and Jochen Ulrich (HttpStatusCodes).

Design Principles

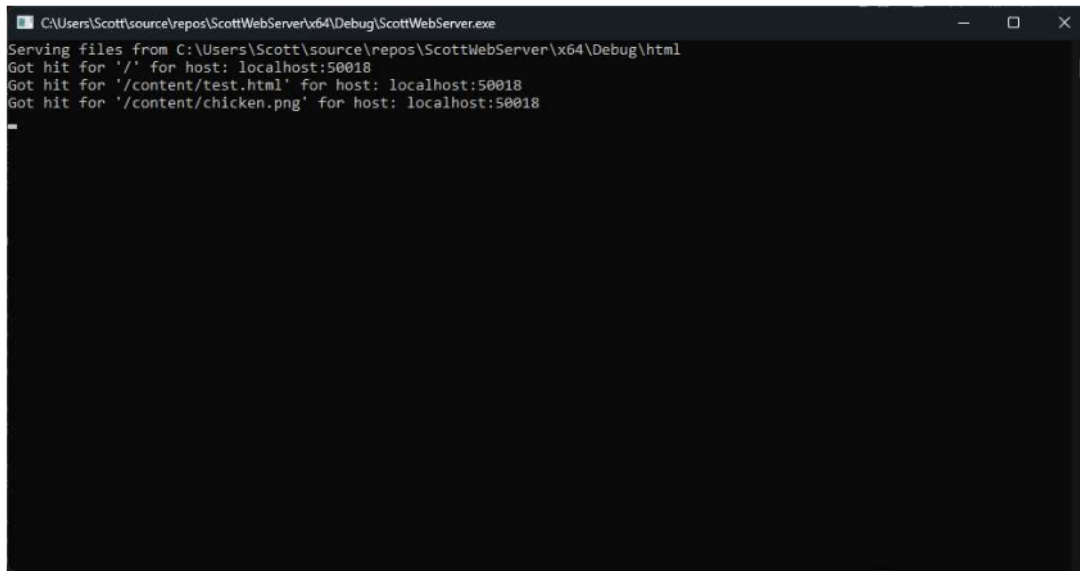
While not fully implemented, the application was built to be compliant with RFC2616, the specification which describes the Hypertext Transfer Protocol. It is important that the application adheres to this specification, to be compatible with most web browsers. The specification was therefore studied heavily during the development of the application.

The application is multithreaded and uses an IO dispatch model to allow multiple simultaneous connections. This is a desirable trait of web servers, which must be able to serve many clients at a time. A slow connection from one client must not affect connections to other clients.

The final key design principle behind the application is that it must be relatively simple to read and understand the source code. To that end, it makes use of Object-Oriented programming to simplify common concepts such as HTTP requests and responses.

Overview of Functionality

When the application is opened, it listens for HTTP requests on port 50018. It serves content from the 'html' directory, in the same folder as the executable. When a client requests a resource with a GET command, the application searches the content directory for a matching file. If the file is found, it is served to the client with the appropriate MIME type, which is then rendered by the browser. If no matching file is found, the application responds with a 'Not Found' error message. Requested resources are printed to the console output. Figure 1 shows an example of the running console output. Figure 2 shows an example of the 'test.html' file being rendered in a web browser.

A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\Scott\source\repos\ScottWebServer\x64\Debug\ScottWebServer.exe. The window contains the following text:

```
Serving files from C:\Users\Scott\source\repos\ScottWebServer\x64\Debug\html  
Got hit for '/' for host: localhost:50018  
Got hit for '/content/test.html' for host: localhost:50018  
Got hit for '/content/chicken.png' for host: localhost:50018
```

Figure 1

Test Page



Image by [Hung Nguyen](#) from [Pixabay](#)

[Go back to the main page](#)

Status	Method	Domain	File
200	GET	localhost:50018	test.html
200	GET	localhost:50018	chicken.png
404	GET	localhost:50018	favicon.ico

Figure 2

Speaking Hypertext Transfer Protocol

Figure 3 shows an example of a raw HTTP request. Request behavior is defined in section 5 of RFC2616. The first line of the request follows the following syntax:

Method SP Request-URI SP HTTP-Version CRLF

The application creates an 8,192 byte buffer, and then reads the request into the buffer, stopping when it reaches an empty line or the buffer size. It then extracts the method (GET) and the URI (/content/test.html), as well as the remaining headers.

```
GET /content/test.html HTTP/1.1
Host: localhost:50018
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:137.0) Gecko/20100101 Firefox/13
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Referer: http://localhost:50018/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Priority: u=0, i
```

Figure 3

By now, the application knows which resource the client is looking for, and can send the requested file. To do this, the application follows the behavior defined in section 6 of RFC2616. Figure 4 shows an example of the response headers the application sends to the client, according to the following syntax:

HTTP-Version SP Status-Code SP Reason-Phrase CRLF

CRLF

[Message Body]

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 556
```

Figure 4

Once the requested resource has been received by the client, the connection is closed, freeing the application to accept new connections.

Code Walkthrough

The program defines the variables `HTTP_SERVER_HEADER_BUFFER_SIZE` and `HTTP_ROOT_DIRNAME`. `HTTP_SERVER_HEADER_BUFFER_SIZE` defines the maximum size of an incoming HTTP request, and also the size of the buffer used to hold incoming HTTP requests. `HTTP_ROOT_DIRNAME` defines the name of the folder relative to the program executable that the program will look for HTML files to serve.

```
#define HTTP_SERVER_HEADER_BUFFER_SIZE 8192
#define HTTP_ROOT_DIRNAME "html"
```

The `ciToString` function takes a pointer to a `ConnectionInstance` instance and returns a string containing the raw HTTP request. It first creates a char buffer the size of

HTTP_SERVER_HEADER_BUFFER_SIZE, reading the request into the buffer. It then constructs a string from the buffer to be returned to the calling function.

```
// Read the HTTP request headers into a string (stop when encountering an empty line as per RFC2616 4.1)
std::string ciToString(ConnectionInstance* ci) {
    //This code could probably be more concise, but seems robust enough for the time being
    char recvbuff[HTTP_SERVER_HEADER_BUFFER_SIZE];
    char* currentByte = recvbuff;
    int newlinesReadInARow = 0;
    int bytesRead = 0;
    std::string headers = "";

    while (true) {
        if (bytesRead > HTTP_SERVER_HEADER_BUFFER_SIZE) {
            std::cout << "Header size limit reached. Not reading any more!" << endl;
            break;
        }
        ci->Readn(currentByte, 1);
        if (*currentByte == '\n' || *currentByte == '\r') {
            newlinesReadInARow++;
        }
        else {
            newlinesReadInARow = 0;
        }
        if (newlinesReadInARow > 3) {
            //std::cout << "Got end of headers!" << std::endl;
            break;
        }
        //std::cout << *currentByte;
        headers += *currentByte;
        currentByte++;
        bytesRead++;
    }

    return headers;
}
```

The simpleMessage function takes a pointer to a ConnectionInstance instance, an integer status code, and a string error message. It then instantiates a new HttpResponseHeaders instance and writes the HTTP response back to the client. This function is used for writing simple messages back to the client in case of error, or in the case where it would not be appropriate to return a document or file.

```
// Generate a simple HTTP response with a message and status code
void SimpleMessage(ConnectionInstance* ci, int statusCode, std::string errorMessage)
{
    HttpResponseHeaders* response = new HttpResponseHeaders(statusCode, errorMessage.length(), "text/html");
    ci->SendString(response->serialize());
    ci->Writen(errorMessage.c_str(), errorMessage.length());
}
```

The HttpResponseHeaders class is a simple class which stores an integer status code, a string content type and an integer content length. The class is used to encapsulate a simple HTTP response. It features a serialization method, which when called returns the raw HTTP response as a string, which can be written back out to a client connection. The class relies on Jochen

Ulrich's `HttpStatusCodes` library to write the correct HTTP reason phrase into the serialized string.

```
#include "HttpResponseHeaders.h"
#include <sstream>

HttpResponseHeaders::HttpResponseHeaders(int statusCode, int contentLength, std::string contentType)
{
    this->statusCode = statusCode;
    this->contentType = contentType;
    this->contentLength = contentLength;
}

std::string HttpResponseHeaders::serialize()
{
    std::stringstream response;
    // Thanks to Jochen Ulrich <jochenulrich@t-online.de> for HttpStatusCodes
    response << "HTTP/1.1 " << this->statusCode << " "
        << HttpStatus::reasonPhrase(this->statusCode) << "\r\n"
        << "Content-Type: " << this->contentType << "\r\n"
        << "Content-Length: " << this->contentLength
        << "\r\n\r\n";
    return response.str();
}
```

Naturally, as there is a HTTP response class, it makes sense that there would be a complimentary HTTP request class:

```
class HttpRequest
{
private:
public:
    std::string method;
    std::string host;
    std::string userAgent;
    std::string requestUri;
    std::map<std::string, std::string> headers;

    HttpRequest(std::string method,
        std::string requestUri,
        std::map<std::string, std::string> headers);

    static HttpRequest* unserialize(std::string in);
};
```

The HTTP request class stores a number of string variables, as well as a map for storing generic headers. It can be instantiated via its constructor function, or via a static deserialization method. The 'unserialize' method takes a raw HTTP request and converts it into an instance of the `HttpRequest` class.

The main part of the code responsible for handling connections is the ClientHandler function. This function takes a pointer to a ConnectionInstance instance as a parameter. The first thing it does is call on the ciToString function to read the incoming request into a string, and deserializes the request using HttpRequest's 'unserialize' method.

```
// Read the request into a string
std::string headers = ciToString(ci);

// Unserialize the request into a HttpRequest instance
HttpRequest* httpRequest = HttpRequest::unserialize(headers);
```

It then check that the request is of 'GET' type, and sends an error using SimpleMessage if this is not the case. This is because the server only implements GET requests in its current form. It logs the request to the console, and appends 'index.html' to the URI if the client has not specified a filename in their request.

```
// If not a GET request, send 501
if (httpRequest->method != "GET") {
    SimpleMessage(ci, 501, "Not Implemented.");
    ci->Disconnect();
    return;
}

// Log the request to the console
std::stringstream logMessage;
logMessage << "Got hit for '" << httpRequest->requestUri << "' for host: " << httpRequest->headers["Host"] << std::endl;
std::cout << logMessage.str();

// If the client has asked for a directory (e.g. a URI ending in /) grab the default index.html file for that directory
if (httpRequest->requestUri[httpRequest->requestUri.length() - 1] == '/') {
    httpRequest->requestUri += "index.html";
}
```

The server then attempts to open the requested file. If the file failed to open, the program will send an error 404 message back to the client

```
// Construct the relative file path of the file to serve
std::string relFilePath = HTTP_ROOT_DIRNAME + httpRequest->requestUri;

// Attempt to open the file
// https://cplusplus.com/doc/tutorial/files/
// Also, this is horribly insecure as we are taking unsanitized input from the client
// Not ideal... But works :)
std::ifstream file(relFilePath, ios::binary);

// Attempt to open the file. Send a 404 on failure, as the file probably doesn't exist
if (!file.is_open()) {
    SimpleMessage(ci, 404, "Not Found.");
    ci->Disconnect();
    return;
}
```

The server then selects the correct MIME type to send the file with. At present, only basic MIME types are supported. For files other than HTML or CSS, the generic 'application/octet-stream' type is used. The server also gets the size of the file, to be sent as the content-length header.

```
// Grab the file extension of the file to serve
// https://en.cppreference.com/w/cpp/filesystem/path/extension
std::string extension = fs::path(relFilePath).extension().string();

// Set the default MIME type
std::string contentType = "application/octet-stream";

// Explicit MIME types
// There are libraries available to help determine the MIME types automatically
// This should suffice for now though
if (extension == ".html" || extension == ".htm") {
    contentType = "text/html";
}
if (extension == ".css") {
    contentType = "text/css";
}

// Grab the size of the file by seeking to the end and measuring the distance from the start
// https://stackoverflow.com/a/18816228
file.seekg(0, std::ios::end);
std::streamsize size = file.tellg();
file.seekg(0, std::ios::beg);
```

Finally, the program writes HTTP headers to the connection, followed by the contents of the file buffer, before closing the connection and returning.

```
// Attempt to read the file into a buffer, send error 500 on failure
std::vector<char> buffer(size);
if (!file.read(buffer.data(), size)) {
    SimpleMessage(ci, 500, "Internal Server Error.");
    ci->Disconnect();
    return;
}

// Send the HTTP headers
HttpResponseHeaders* response = new HttpResponseHeaders(200, size, contentType);
ci->SendString(response->serialize());

// Send the binary data
ci->Writen(buffer.data(), static_cast<int>(size));
ci->Disconnect();
return;
```

The main function of the program organizes the high level functionality of the server. It first determines the location of the executable, before creating a new socket and listening for connections on port 50018. It writes a simple message to the console, and then waits for new connections to the server, offloading them to a separate thread.

```
int main(int argc, char** argv)
{
    // Create a buffer
    // Using wide chars here (Unicode) as PathCchRemoveFileSpec only supports wide chars by the looks of it
    // Wide chars aren't compatible with the usual cout, wcout needs to be used
    // There is an older function called pathremovefilespec but this is deprecated, vulnerable to overflow
    // It does not support emoji filenames :(
    wchar_t buffer[MAX_PATH];

    // Get the path to the currently running executable
    GetModuleFileNameW(NULL, buffer, MAX_PATH); // https://learn.microsoft.com/en-us/windows/win32/api/libl

    // Strip the filename from the path
    PathCchRemoveFileSpec(buffer, MAX_PATH); // https://learn.microsoft.com/en-us/windows/win32/api/pathcch

    // Change the working directory of the program
    // This is probably not required when running the exe file standalone.
    // However, my IDE insists on making the working directory the same as the source directory for some reason
    GetCurrentDirectory(buffer); // https://stackoverflow.com/a/3485200

    INet4Address* address = new INet4Address(50018);
    ServerConnection* server = new ServerConnection(address);

    server->StartServer(10);

    wcout << "Serving files from " << buffer << "\\\" << HTTP_ROOT_DIRNAME << std::endl;

    while (true)
    {
        ConnectionInstance* ci = server->AcceptClient();
        thread(ClientHandler, ci).detach();
    }

    system("pause");
}
```

Final

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

A very nice http server, well designed and implemented - great work, Scott.

88/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11