

Programming Assignment: HTTP Client

You have not submitted.

 It looks like this is your first programming assignment. [Learn more](#)

Deadline Pass this assignment by Sep 21, 12:59 AM EDT

Instructions

My submission

Programming Assignment 1: HTTP Client

The purpose of this programming assignment is to learn basic network programming and protocols, and to exercise your background in C/C++ programming. You'll also become familiar with the environment to be used in the class and the procedure for handing in machine problems.

You will write, compile and run a simple network program, and implement an HTTP client program that can load web pages and objects. For this assignment, you will work on the virtual machine provided. The instructions for installing the virtual machine can be found in the handout named "VM installation instructions".



VM Installation Instructions.pdf
PDF File



assignment1_files.zip

Task 0: Socket-Oriented Programming (Optional)

This step is optional, it requires no coding and it is intended for you to get familiar with compiling and testing network applications on Linux machines. If you have already coded a network application in C, feel free to skip to the next section.

In the Assignment 1 files, you will find a folder named MP0, which contains the programs **client.c**, **server.c**, **talker.c**, and **listener.c** from Beej's Guide to Network Programming (<http://beej.us/guide/bgnet/>). Figure out what these programs are supposed to accomplish. Reading Beej's guide itself is very helpful, if you can tolerate his sense of humor. Compile the files using the GNU C compiler to create the executable files client, server, talker, and listener. For example, to create the executable file client, you'd execute the following command:

- `gcc -o client client.c`

Learn how to use the **make** command. The file **Makefile** in your assignment folder is configured to compile the example code provided. Here are some specific make commands you can run:

Purpose	Command(s)
Compile the programs individually	make client
	make server
	make talker
	make listener
Compile all programs	make all
Revert the folder to its original state by removing any file created by previous calls to make	make clean

Once you have compiled the code, open two terminal windows, and execute **client** in one and **server** in the other. This establishes a TCP connection. Execute **talker** in one terminal and **listener** in the other. This sends a UDP packet. Note that the connection-oriented pair (server and client) use a different port than the datagram-oriented pair (listener and talker). Try using the same port for each pair. Do the pairs of programs interfere with each other? Why or why not?

No submission is required for Task 0.

Task 1: HTTP Client

In this task, you will implement a simple HTTP client. The client should be able to GET files correctly from standard web servers.

HTTP uses TCP, so you can use Beej's client.c and server.c as a base.

HTTP Communication Basics

- The client sends an HTTP GET request to the server.
- The server replies with a response based on the request of the client (e.g. acknowledge the validity of the request with an OK response OR reply with a file not found error response etc.).
- Based on the response from the server, the client takes further action (e.g. store the message content in a file OR redirect to a different address with a new GET request, etc.)

HTTP GET Requests

Here's the very simple HTTP GET request generated by *wget*:

```
1 GET /test.txt HTTP/1.0
2 User-Agent: Wget/1.15 (linux-gnu)
3 Accept: */*
4 Host: localhost:3490
5 Connection: Keep-Alive
```

Here's the purpose of each line:

- **GET /test.txt** instructs the server to return the file called test.txt in the server's top-level web directory.
- **User-Agent** identifies the type of client program.
- **Accept** specifies what types of files are desired – the client could say “I only want audio”, or “I want text, and I prefer html text”, etc. In this case it is saying “anything is fine”.
- **Host** is the URL that the client was originally told to get from – exactly what the user typed. This is useful in case a single server has multiple domain names resolving to it (maybe www.cs.illinois.edu and www.math.illinois.edu), and each domain name actually refers to different content. This could be a bare IP address, if that's what the user had typed. The 3490 is the port – this server was listening on 3490, so I called “wget localhost:3490/test.txt”.
- **Connection: Keep-Alive** refers to TCP connection reuse for future requests.

Note that the newlines are technically supposed to be CRLF - “`\r\n`”. Only the first line is essential for a server to know what file to give back, so your HTTP GETs can be just that first line. HTTP specifies that the end of a request should be marked by a blank line, so be sure to have two newlines at the end. (This demarcation is necessary because TCP presents you a stream of bytes, rather than packets.)

You may use either HTTP 1.0 or 1.1. However, notice that if you are using HTTP/1.0, the Host header is *not required*. If you are using HTTP/1.1, the Host header is *required*. But be aware that the Host header may still be necessary for some URLs even in HTTP/1.0.

HTTP Responses

Here's an example response from an HTTP server:

```
1 HTTP/1.1 200 OK
2 Date: Sun, 10 Oct 2010 23:26:07 GMT
3 Server: Apache/2.2.8 (Ubuntu) mod_ssl/2.2.8 OpenSSL/0.9.8g
4 Last-Modified: Sun, 26 Sep 2010 22:04:35 GMT
5 ETag: "45b6-834-49130cc1182c0"
6 Accept-Ranges: bytes
7 Content-Length: 13
8 Connection: close
9 Content-Type: text/html
10
11 Message is Hello world!
12
```

Per the HTTP standard, again an empty line (two CRLF, “`\r\n\r\n`”) marks the end of the header, and the start of message body.

Some of the common response headers are:

- “HTTP/1.0 200 OK” for correctly returning the requested document.
- “HTTP/1.0 404 Not Found” when the requested file does not exist.
- “HTTP/1.0 301 Moved Permanently” when the server was moved to a different address.

Client Requirements

You should write your code in C or C++. Note that C++11 is not supported by our autograder. This program should not print anything to stdout. Your code should compile to generate an executable named `http_client`, with the usage specified as follows:

- `./http_client http://hostname[:port]/path_to_file`

For example:

- `./http_client http://illinois.edu/index.html`
- `./http_client http://localhost:5678/somedir/somefile.html`

Your client should send an HTTP GET request based on the first argument it receives. Your client should then write the message body of the response received from the server to a file named **output**.

The file can be any type of content that you could get via HTTP, including text, HTML, images, and more. Note that you don't need to specify the content-type in your GET request, but you need to be able to receive the file in different types. The file size may vary from a few bytes up to hundreds of MBs.

Optional: your client should also be able to handle HTTP 301 redirections (not graded).

Nonexistent Files

If the server returns a 404 error for a non-existent file, your client should only write "FILENOTFOUND" to the output file.

Invalid Arguments

The client should be able to handle any invalid arguments as follows. First, if the protocol specified is not HTTP, the program should write "INVALIDPROTOCOL" to the output file. For example, the following command should be handled as an invalid protocol argument:

- `./http_client htpt://localhost/file`

Second, if the client is unable to connect to the specified host, the program should write "NOCONNECTION" to the output file.

Testing your client

You can test the correctness of your client by running a local http server using the following python command line script on a separate terminal window in the relevant directory:

- `python3 -m http.server 8000`

where 8000 is the port number that the server will run on.

For your client program, the hostname for the server will be `localhost:8000` and the executable will be run as:

- ./http_client http://localhost:8000/example_file

Submission Instructions

For this programming assignment, you must submit only the code file for your implementation in the submission tab on coursera. The autograder assumes your implementation is contained in a single file, ending either in .c or .cpp, depending on whether you programmed in C or C++, respectively. Otherwise, the name of the file does not matter. Once you have submitted your code, the autograder will run it and display feedback within minutes. The feedback will include your overall score and the results of each test case for that submission. Before the assignment deadline, you can submit as many times as you want and the submission with the highest score will be counted towards your course grade.

How to submit

When you're ready to submit, you can upload files for each part of the assignment on the "My submission" tab.