EE122: Introduction to Computer Networks — Fall 2007

# Project 1: Build a Web Server*

**Milestone 2 - DUE 11:00 PM OCT 31, 2007**

Prof. Vern Paxson / Lisa Fowler / Daniel Killebrew / Jorge Ortiz

## Introduction

The goal of this project is to build a functional server and client for a subset of HTTP/1.0, plus a couple of extensions. Your server will be able to serve content to web browsers and clients anywhere in the world. Similarly, your web client will be able to receive content from any web server. This project will teach you the basics of distributed programming and client/server architectures, and allow you to gain a functional understanding of the HTTP protocol. This effort will be broken down into smaller stages, and will be due in two milestones. This is the project description for the second milestone.

As a reminder, a web server can be seen abstractly as a program that executes a logically infinite loop wherein it receives and processes commands, structured something like the following:

*Loop forever:*
*Listen for incoming connections*
*- Accept new client connection*
*- Parse HTTP/1.0 request*
*- Ensure well-formed request (return error otherwise)*
*- Determine if target file exists and if permissions are set properly (return error otherwise)*
*- Transmit contents of file to client (by performing reads on the file and writes on the socket)*
*- Close the connection*

## Milestone 2: HTTP Server/Client, Concurrency, and Wireshark

You will extend your work from the first milestone. You should ensure that any errors from the previous milestone have been addressed and corrected before beginning this milestone. Note that reference implementations for the first phase are available from `http://inst.eecs.berkeley.edu/%7Eee122/fa07/projects/Project1A-ref/client/` and `http://inst.eecs.berkeley.edu/%7Eee122/fa07/projects/Project1A-ref/server/`. Feel free to use elements of this code for your second phase implementation (note: we do not promise it is bug-free!). You will likely benefit considerably in terms of learning by not using it in its entirety for your second-phase starting point instead of your own code.

*Writeup Version 4, Friday Oct 26: due date extended to Oct 31. Writeup Version 3, Saturday Oct 20: Clarified that traces don't need to be turned in for Wireshark steps 1–19. Version 2, Monday Oct 15: Due date postponed a week to Oct 29. Added pointer to reference implementations for first phase. Clarified that code **403** needs to be supported too. Version 1, Tuesday Oct 2: original version.

## HTTP Server/Client

We now modify the client/server system you built before in order to make the system more robust and function more like a real web server, instead of naively repeating the GET request back to the client.

### Server Behavior

Your web server shall no longer output the components of the GET request, and instead will retrieve files from its local filesystem according to any valid requests it may have received. It is important that your server properly communicate with the client at the application level. It will do so by generating HTTP/1.0 responses with the appropriate status codes.

At this point, your server will receive HTTP requests and validate those requests as well-formed. Instead of merely sending the request back to the client as we did before, your server will now parse the request and create the appropriate status line, headers, and optional body that it will then send to the client as the complete HTTP response.

*HTTP/1.0 Response Messages*

An informal description of the full HTTP response message is:

```
Status-Line
Headers
CRLF
Body
```

Your server will generate and send messages with the above format to the connected client.

*HTTP/1.0 Response Messages: Status-Line*

The grammar for the HTTP/1.0 Status-Line message is:

`Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF`

In our case, `HTTP-Version` will be `HTTP/1.0`. The `Reason-Phrase` can be arbitrary (any printable character, not including CRLF) since the client is not required to process any information contained therein. The possible options for the `Status-Code` are enumerated in Section 6.1.1 and Section 9 of RFC 1945, however you are only required to support `Status-Code SP Reason-Phrase` combinations of: `200 OK`, `400 Bad Request`, `403 Forbidden`, `404 Not Found`, and `501 Not Implemented`.

You must determine which response message is appropriate to send to the client by performing validation on the request:

- You must ensure that the request itself is in fact valid (*i.e.*, it matches the grammar as noted in the first milestone). If there is a problem with the request, return to the client the following errors where appropriate:
  `HTTP/1.0 400 Bad Request: ERROR -- Invalid Method token: <`*method token*`>`
  `HTTP/1.0 400 Bad Request: ERROR -- Invalid Request-URI token: <`*request-URI*`>`
  `HTTP/1.0 400 Bad Request: ERROR -- Invalid HTTP-Version token: <`*version token*`>`

- Your server is only required to support directory listings and files of types: `.txt`, `.html`, `.gif`, and `.jpg`. File types will be interpreted from case-insensitive filename extensions. Handling requests for directory listings will be described later. If the file type is not one of the above, your server will return an error message listing the requested URI, in the following form:

  `HTTP/1.0 501 Not Implemented: /oops.avi`

- The file must exist, according to the tests described in the section on "*Resolving Filenames and Directories.*" If not, report an error message of the form:

  `HTTP/1.0 404 Not Found: /spam/is-tasty.html`

- Report any additional errors on the server (other than those that prevent communication with the client) with: `HTTP/1.0 400 Bad Request -- Error: <`*text of exception message*`>`

*Handling Input From The Client*

Along with the method, URI and version in the first line, the incoming request from the client might include headers as well. Your server should print these headers to standard output but otherwise ignore the data (except as mentioned below), to provide better compatibility for using your server with actual Web browsers.

If the request is valid and your server can respond to the request with the desired file contents, your server must generate and return a valid HTTP response message, along with both `Content-Type` and `Content-Length` header messages containing the appropriate values. For example, the general format of an HTTP response to a valid GET request for an existing `.html` file is:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1234

<the 1,234 bytes of the file's contents>
```

`Content-Type` values are managed by IANA and can be found at `http://www.iana.org/assignments/media-types`. The media type is comprised of the content type and its subtype in the form: `<`*`content-type`*`>/<`*`subtype`*`>`, such as `text/html`.

After the HTTP response message, the order of the header messages is unimportant. A blank line (containing only `CRLF`) must delimit the header messages from the start of the file contents. A blank line must also follow all response messages such as `404` and `501` messages.

*Resolving Filenames and Directories*

Your server will treat requested URIs *within the context of the server's working directory*. Let's refer to this directory as *XYZ*. Using `GET /foo HTTP/1.0` as an example, your server will attempt to resolve URIs in the following order:

1. Exactly as is, *e.g. XYZ*`/foo`
2. Append `/index.html` to the URI, *e.g. XYZ*`/foo/index.html`
3. Treat the URI as a directory, *e.g. XYZ*`/foo/`

If the requested URI has a trailing `"/"`, you should skip to step 2. If the file cannot be resolved by any of the above, generate an appropriate `404` response. Otherwise, if the file or directory was found, handle the

request as described in the previous section. In the case of a directory, you will generate "file contents" by returning the results of running `ls` on that directory (as if you ran `ls` in the directory, piped it into a file, and then sent the file to the client). You may find the standard `system`() or `popen`() library functions useful for this.

Your server needs to detect when it is unable to read a file or directory due to file-system permissions. In this case, generate an error message of the form:

`HTTP/1.0 403 Forbidden: /secret/diary.html`

*Additional Error Handling*

For any errors that cannot be communicated to the client (*i.e.* socket failures or a bad port number), print the following error message to standard output.

`ERROR -- Incommunicable Error: <`*text of error message*`>`

*Connection Maintenance*

By default, once your server has delivered the requested content, *it should close the connection* and await further connections. However, **if** the client includes the following header in its request:

```
Connection: Keep-Alive
```

then the server must leave the connection open. In this case, the client might send additional requests over the connection, which the server should process in the same fashion as above, including closing the connection after replying if a subsequent request does not include the `Connection: Keep-Alive` header. (Note, this functionality is not part of HTTP/1.0; however, it was a popular extension that was used prior to the standardization of HTTP/1.1, which provides a similar mechanism.)

*Chunking*

Finally, your server must take an optional flag **-c *size***. If specified, *size* gives a *chunking size*. Any time the item returned by the server exceeds *size* bytes, the server does the following. First, it includes the header:

```
Transfer-Encoding: chunked
```

in its reply, and in this case it does *not* include a `Content-Length` header.

It then uses "chunking" to transfer the item it returns. This type of transfer is defined in Section 3.6.1 of RFC 2616. We will use a somewhat simpler definition: you do not need to worry about *chunk-extension* or *trailer* components. Also, note that in the grammar given in the RFC, the term *chunk-data = chunk-size*(OCTET) means that a *chunk-data* component is comprised of exactly *chunk-size* bytes ("octets").

When your server uses chunking, it divides up the object it receives into chunks of size *size* (as specified by **-c**), except that the last chunk may be smaller. You may find this website helpful as a working example of chunking: `http://www.httpwatch.com/httpgallery/chunked/`

(Note that when using **-c**, your server is no longer compliant with HTTP/1.0; nor does it have enough HTTP/1.1 functionality to be compliant with that version, either. So if you want to use your server to send items to an actual browser, you should not use the **-c** option.)

In summary, the command line for the server looks like this:

`http_server [-c `*`chunk_size`*`] [`*`listen_port`*`]`

where items in square brackets [] are optional. The default *listen_port* is 7788.

**Client Behavior**

You will extend your previous client to save the content it has received from the server. The HTTP response message and any associated headers should be printed to standard output without any additional processing. If the request succeeds (`200 OK`), the client will save the file contents into a file in the current working directory (see below for naming the file). For `200 OK` replies from the server, by default the client shall obtain the contents of the reply by simply reading from the socket until the server closes the connection.

Once the client has retrieved an item from the server (via a `200 OK` reply), it constructs a local file name into which to store the item. The new file name is the simple name of the URI, i.e., all the characters following the last '/'. This handles all but a few special cases. In these special cases, your client must create these files according the simple names listed below.

| | |
|---|---|
| `/` | In this case, the simple name will be `dir` |
| `foo/` | In this case, the simple name will be `foo` |
| `foo/.` | In this case, the simple name will be `dot` |
| `foo/..` | In this case, the simple name will be `dotdot` |

If a file with the same name already exists, replace it.

If the request does not succeed, your client should print the status code and associated text, such as:

```
403 Forbidden: /secret/diary.html
```

After processing the response received from the server, which includes reading from the socket until the server closes it, the client should read the next full URL from standard input. When end-of-file (`CTRL+D`) is reached on standard input, the client program should terminate.

For any network errors that occur, print the following message on standard output:

`ERROR -- Network Error:` <*text of exception message*>

We also extend the client to support the additional features added to the server:

*Connection Maintenance*

Your client must support an optional **-p** flag. If set, your client must use a "persistent" connection to the server. It indicates this by including the header:

```
Connection: Keep-Alive
```

in its request, as already discussed above.

In this case, your client must find the end of the server's reply by counting for a number of bytes equal to that specified in the `Content-Length` header returned by the server (or by interpreting chunking—see next item); it can no longer simply read until the server closes the socket, because in this case the server keeps the socket open after sending the item.

In addition, each time your client reads a new URL from standard input, it checks whether the URL refers to the same host as the server to which it is currently connected. If so, then it sends the corresponding request using the already-opened socket, again specifying `Connection: Keep-Alive`. If not, however, then it

closes the connection to the old server before opening another connection to the new server.

*Chunking*

Your client must also be prepared for the server to return a chunked transfer, as indicated by a header in the server's reply of the form:

```
Transfer-Encoding: chunked
```

Chunked transfers can occur regardless of whether the client's **-p** flag is specified.

The client's command line looks like:

```
http_client [-p]
```

## Notes

- Both clients and servers should be aware that either party may close the connection prematurely, due to user action, automated time-out, or program failure, and should handle such closing by printing out the appropriate error message (`Incommunicable` or `Network Error`). In any case, the closing of the connection by either or both parties always terminates the current request, regardless of its status.
- You may use `<regex.h>` or lex & yacc to assist with your parsing tasks. It's up to you to figure out how to use these tools if you choose to use them. (Note though that `<regex.h>` is generally significantly easier to learn than lex & yacc, for those unfamiliar.)

## Examples

Here we give example runs of the server and client. Note that these do not reflect a comprehensive test case for your code. Instead, the purpose is to give you an idea of what the traffic between client and server will look like, and what to print to standard output.

We start the server on *ilinux3.eecs.berkeley.edu* with the command line:

```
http_server -c 512 7788
```

We then start the client with:

```
http_client -p
```

The user types into the client's command line:

```
http://ilinux3.eecs.berkeley.edu:7788/index.html
```

Now the client opens a connection and sends to the server:

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
```
*<a blank line here>*

where *a blank line* is a CRLF by itself. (Each of the lines before it is also terminated by a CRLF, but they contain some additional text too.)

The server prints to standard out:

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
```
*<a blank line here>*

Suppose *index.html* in the server's working directory is 539 bytes in size. The server responds with:

```
HTTP/1.0 200 OK
Content-Type: text/html
Transfer-Encoding: chunked
```
*<blank line here>*
```
200
```
*<512 bytes of data, plus an additional CRLF>*
```
1b
```
*<27 bytes of data, plus an additional CRLF>*
```
0
```
*<blank line here>*

The client will print to standard output:

```
HTTP/1.0 200 OK
Content-Type:  text/html
Transfer-Encoding:  chunked
```
*<blank line here>*

The client will save the contents it received to a file *index.html* in its current directory. If the next URL also refers to the server on port 7788 of *ilinux3.eecs.berkeley.edu*, then the client uses the current connection to retrieve it (because we invoked the client using **-p**). Otherwise, it closes its current connection before opening one to the new server.

## Wireshark

In this project, we also want you to become familiar with using tools to analyze network behavior, both of protocols as others have implemented them and your own client and server. To this end, you need to:

1–19. Do the Wireshark Lab from the textbook, which you will find at `http://inst.eecs.berkeley.edu/%7Eee122/fa07/projects/Wireshark-HTTP.pdf`

If you use the instructional machines, you will need to capture packets using **tshark** (or **tcpdump -s 0**) and then visualize the traces using Wireshark, rather than capturing them directly with Wireshark. (A reminder: you will find these tools under */share/b/ee122/*. Unfortunately, other copies are present on the instructional machines which can fail due to not having permission to read the network device.)

The textbook authors provide sample traces to go along with this lab. It's fine for you to consult these to understand better what's going on, but for your final answers in this part of the project you need to analyze traces you captured yourself.

20. Fetch the US Bill of Rights document (section #3 of the Lab) using your client. Capture a packet trace of the retrieval and determine (1) how many total packets your client sent, (2) how many of the packets sent by your client were TCP SYN, FIN or RST control packets, (3) how many of its packets were TCP acknowledgments that didn't send any data to the server.

21. Store the US Bill of Rights document in a file where your server can access it. Retrieve it using your client, again capturing a packet trace. Answer question #13 in the Lab for this retrieval (how many

data-containing TCP segments).

22. Repeat the previous step, but this time run your server using the flag `-c 500` to instruct it to send the item in chunks of 500 bytes. Again answer question #13 for this retrieval.

## General Instructions

1. You may choose from C or C++ to build your web server and client, but you must do it in a Unix-like environment. You will use the following system calls to build your system: `socket(), select(), listen(), accept(), connect()`. This means no multithreaded servers.
2. The code must be clean, well-formatted, and well-documented.
3. The programs must compile and run on the UNIX instructional account machines. We will not be able to grade programs that work only on Windows.
4. Your program needs to be robust to ill-formed or otherwise stressful input.

### Submission Guidelines

1. You will need to submit the source code for both the server and the client, along with an appropriate Makefile to compile both the programs. The `gmake` command should work with your Makefile.
2. The client should compile to a binary called `http_client`.
3. The server should compile to a binary called `http_server`.
4. Include a `README.txt` file that documents how to use your `http_client` and `http_server` programs.
5. Include a `Wireshark-Lab.txt` writeup with your answers to the Wireshark Lab questions, including the additional steps. For each of the additional steps (but not for the main Lab), include a text dump of the relevant lines from your trace files.
6. Submission Steps:
   (a) As before, make sure you are registered with the EECS instructional account system.
   (b) Log in to your instructional account.
   (c) Create a directory called `proj1b`: `% mkdir proj1b`
   (d) Change to that directory: `% cd proj1b`
   (e) Copy all the files that you wish to submit to this directory.
   (f) Run the submit program: `% submit proj1b`
   (g) Make sure that no error messages are displayed. If some error occurs, retry. If it still occurs, contact the TAs.

## Grading

- This project milestone is worth **10%** of the course grade.
- Of the elements in the project, your client is worth 40%, your server 40%, and the Wireshark Lab is worth 20%.
- Your submission must satisfy all the specifications in order to receive full credit.
- Programs will be tested on `c199.eecs.berkeley.edu` (a Solaris machine), so your final submitted code should work on that machine.
- You will be penalized if the submitted code is shabbily written (difficult to extend or for us to understand, or brittle in terms of error handling).