ECE 463 Programming Project I

Fall 2015

Objective

You will create network applications on top of the transport layer using network programming (also called *socket programming)*. You will build a simplified HTTP client and a simplified HTTP server that can be accessed by a typical web browser. After that, you will enhance your server to be able to respond to concurrent requests and make it able to handle not only HTTP requests, but also multiple services such as UDP pinging.

Deadline:

This project has multiple parts, and multiple deadlines.

Part #	*Due Date*
Part 1	Wednesday, September 16th 11:59 AM (Noon)
Part 2 & 3	Wednesday, September 23rd 11:59 AM (Noon)
Part 4	Wednesday, September 30th 11:59 AM (Noon)

Questions regarding the different project steps will not be answered after midnight the Tuesday before each due date. This is meant to discourage you from finishing the project at the last minute. **Start early!**

This project is to be done in groups of **two**.

Overview

HTTP (HyperText Transfer Protocol) (RFC1945)

HTTP has been in use by the World-Wide Web global information initiative since 1990. On the Internet, HTTP communications generally take place over TCP/IP connections. The default port is TCP 80, but other ports can be used.

There are two basic operations in HTTP: *request* and *respond*. A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource (GET, HEAD or POST. See RFC1945 for details), the identifier of the resource, and the protocol version in use. After receiving and interpreting a request message, a server responds in the form of an HTTP response message.

In this experiment, you will need to implement a simplified HTTP server and a simplified client. The server should be able to accept connections (one at a time) and to send a text file to the client on request, so you have to understand a minimal subset of messages exchanged by an HTTP client and an HTTP server, particularly the GET command mentioned below. On the client side, you have to know how to obtain a file from the server using the GET command:

```
GET <file path> HTTP/1.0<CR><LF><CR><LF>
```

This command will be sent to the server as simple text (bytes). In C, the control command for `<CR>` and `<LF>` (Carriage Return and Line Feed) are `\r` and `\n` respectively. Here is an example of an HTTP request: the client application sends the string

```
"GET /file.txt HTTP/1.0\r\n\r\n"
```

to the server meaning that you want to get the file `<server home directory>/file.txt` from the server using HTTP 1.0 protocol. In this lab, the home directory of the server you will write in question 2 is the directory where the server resides. We will use HTTP version 1.0 throughout the experiment. On the server side, the program has to interpret what it received from the client. For this experiment, you only need to interpret the request line starting with GET. Note that the server should be assumed as insensitive to upper/lower case (It should accept both GET and get). You can ignore all the

After the server has interpreted the GET request, it will first check if the file requested is available. If so, it will respond, by sending the file in simple text (bytes), to the client with a status code 200 (200 means OK) in the following format:

```
HTTP/1.0 200 OK<CR><LF><CR><LF>

(content of the file requested)……
```

If the file requested cannot be sent to the client, the server will respond with the appropriate error status. The most common ones are “403 Forbidden” for files which the client does not have access right to, and “404 Not found” for non-existing files. Details about the status codes can be found in RFC1945. In the server side, you will check if the file exists and has proper read permissions.

The URL (Uniform Resource Locator), which is commonly called the *address* of a website, contains information including *the protocol used*, *the site’s address (generic or IP)*, *the port where the HTTP server is sitting*, and *the pathname of the file* you request. For example:

http://shay.ecn.purdue.edu:80/~ece463

- **http://** is the protocol used to connect to the server. This can also be ftp:// , telnet:// or other defined protocols.
- **shay.ecn.purdue.edu** is the server hostname. You can use either an IP address or an hostname to identify the server.
- **:80** is the port the http server program is sitting on. It can be omitted if it is 80 (well known port for http). Some proxy servers use port 8000 or 8080 in order to differentiate themselves from http servers.
- **/~ece463/** is the path for the file. If no filename is given, the browser will try to find index.html. In this example, we are actually requesting the file /~ece463/index.html.

Experiment

Part 1 (Simplified HTTP Client):

Use socket programming to write a “simplified HTTP client”. This client should be able to retrieve a file (simple text) from “any” HTTP server. Your client need to establish a connection on the right port and then send a “GET” command to the server, check the server response status code, and print the complete server response (response header + content) on the client screen, regardless of the response code (even error codes). Remember to use the *relative path name* in your GET command. For example, to get the page http://enterprise.ecn.purdue.edu/ece463/lab1/test_short.txt, your GET command sent to port 80 of dtunes.ecn.purdue.edu should be:

“GET /ece463/lab1/test_short.txt HTTP/1.0\r\n\r\n”.

Command Line Arguments:

Your client (httpclient.<login1>.<login2>.c) must support command line arguments as specified:

./httpclient <servername> <serverport> <pathname>

<servername> is the name of the server

<serverport> is the port at which the web-server is running

<pathname> is the path of the file relative to the server home directory

For the example above, you would invoke your client as:

```
./httpclient enterprise.ecn.purdue.edu 80 /ece463/lab1/test_short.txt
```

> *Note*: throughout this document, when you see <login1>.<login2>, this means that you should replace it with your Purdue's login name. For example, if your login names are aturing and achurch then your client should be named
>
> httpclient.aturing.achurch.c

Output:

In your final submission, the client, when executed, must produce **ONLY** the server reply (i.e. HTTP reply header + content of the reply). **All other debugging statements of your own must be turned off upon submission. We suggest that you include a separate flag/command-line argument and debugging output is displayed only when this flag is turned on.**

Testing:

Two test files have been uploaded to a web server as well as to Blackboard (for your reference). They can be accessed at the following addresses:

http://enterprise.ecn.purdue.edu/ece463/lab1/test_short.txt

http://enterprise.ecn.purdue.edu/ece463/lab1/test_very_long.txt

To ensure that you are producing the correct output, it is advisable to compare the output your program creates against the original files. Assuming you have downloaded the files from Blackboard first, this can be done via the following commands:

```
./httpclient enterprise.ecn.purdue.edu 80 test_short.txt > output1.txt

diff output1.txt test_short.txt
```

The only difference between the two should be the HTTP header, which will be present in your output but not in the original text file (run `man diff` for more information on the output of this utility).

Pay special attention to the larger test file (test_very_large.txt) – bugs can arise when requesting a larger file that might not occur when testing with a smaller file.

Submission:

See last section.

Part 2 (Simplified HTTP Server):

Use socket programming to write your own version of a “simplified HTTP server” which allows one connection at a time. Your server should be able to accept a connection request, interpret the “GET” command, ignore other HTTP commands, and reply with one of the following 2 HTTP responses to the “simplified HTTP client” you have built in Part 1.

1. When the file is successfully found by the server, send back an HTTP 200 OK response as follows with the contents of the file specified in the HTTP absolute path, **but starting from the current directory instead of the top-level directory.** (e.g., /index.html should refer to the Unix file "./index.html")

```
HTTP/1.0 200 OK<CR><LF><CR><LF>
(content of the file requested)……
```

2. When the file is not found by the server, send back a 404 not found HTTP response.

```
HTTP/1.0 404 Not Found<CR><LF><CR><LF>
```

3. When the file doesn’t have public read permission, send back a 403 Forbidden HTTP response.

```
HTTP/1.0 403 Forbidden<CR><LF><CR><LF>
```

Your server should also be able to send the entire file requested by the client, if the file is available.

Command Line Arguments:

Your server (httpserver.<login1>.<login2>.c) must support the following command line argument:

./httpserver <serverport>

<serverport> is the port on which the server is running/listening

Testing:

If you implement the server correctly, you should be able to use a browser (Chrome, Firefox or IE) to connect to your server and view a file residing in the server's running directory by accessing http://host_name:port_number/*pathname_of_file*, where host_name is the name of the host your server is running on, and the port_number is the listening port of your server program.

Alternatively, you can test your server by using the client you created in Part 1. Assuming you are running both the client and the server on the same machine, you can access your server using the server name `localhost` and the port you specified when running it.

Once again, make sure to test using both test text files we have provided – as with the client, the server may encounter different bugs with the longer file than with the shorter one.

Make sure to test for every status code you implement. To test the 403 status code, you will need to remove read permissions from a file – this can be done using `chmod` (try `man chmod` for more information)

Output :

The server output is not critical – we will primarily evaluate whether the client can successfully receive the file.

Part 3: (Concurrent Web Server)

Use socket programming to write your own version of a "simplified HTTP server" as specified in Part 2. However, this server must also be able to handle *multiple client requests*, i.e., more than one client can connect to the server and be served simultaneously.

In order to be able to handle multiple clients, the server program must be able to create a new process for each connection, and manage the communication between each pair of processes. We recommend that you support this using `fork()`. Figure 1 shows the state of the server when `fork()` is called.

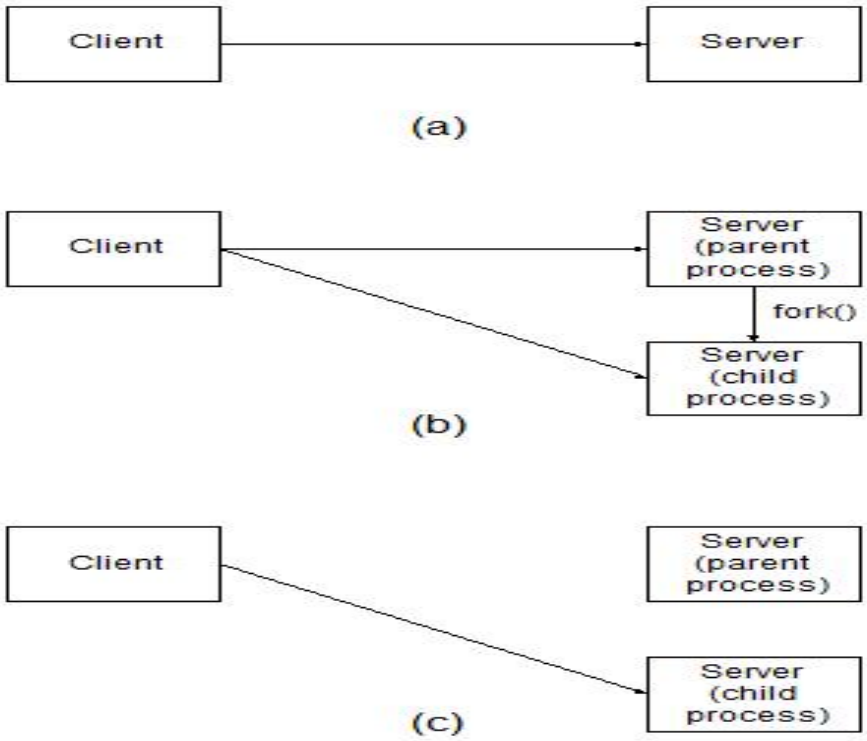


Fig. 1 The use of the fork() command to create child processes for a server allowing more than one connection at the same time. (a) The client is connected to the server; (b) the server calls fork() to create child process; (c) the parent process close the connection and wait for new connections while the child process serves the client.

Command Line Arguments:

Your concurrent server (httpserver_fork.<login1>.<login2>.c) must support the same argument as httpserver (i.e., port number).

Testing:

To check the working of your concurrent server, run two copies of your clients with both attempting to retrieve a large file. This file must be large enough such that the first client is still downloading the file when the second client begins to download it (test_very_large.txt should work just fine). The server works if both clients receive responses simultaneously. This can be checked by running the following command while your clients are downloading the files:

```
ps | grep httpserver_fork
```

As an example, if downloading a file using two clients simultaneously, the output should resemble the following (assuming your server is named httpserver_fork):

```
591 pts/43   00:00:00 httpserver_fork

7747 pts/43   00:00:00 httpserver_fork

31178 pts/43   00:00:00 httpserver_fork
```

Note – if you are having trouble running both clients simultaneously due to them downloading the test files too quickly, add the line

```
sleep(seconds);
```

to your server code after you process the client request but before you close the connection, where `seconds` is an integer. This function causes the process to wait for the specified amount of time – giving you more time to get your other client running**. Make sure to remove this from your server before submitting it!**

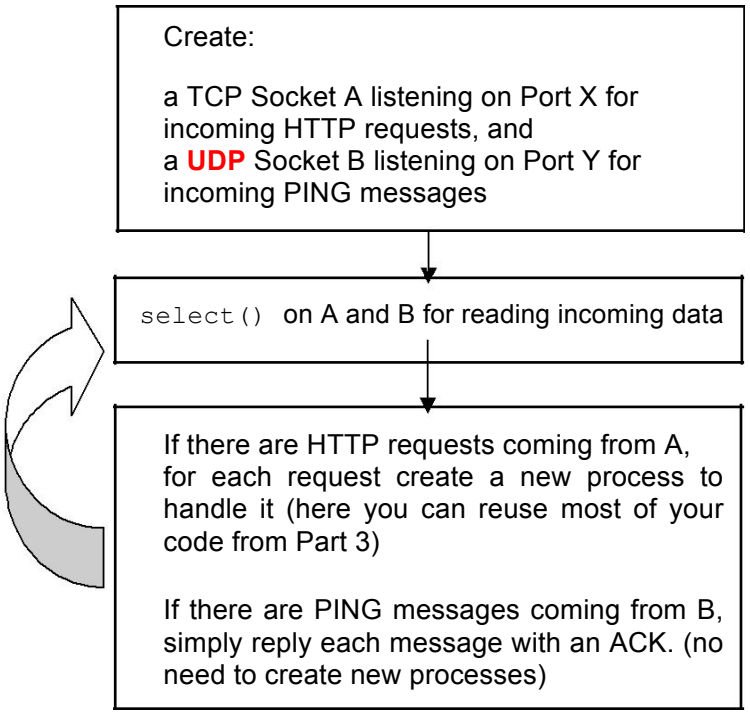
Part 4: (Multi-Service Server)

Use socket programming to write a “simplified multi-service server”. The server must be able to host 2 services simultaneously: HTTP web service and a **UDP** ping service. More specifically:

(1) The server must be able to handle multiple HTTP client requests on one port, as specified in Part 3.

(2) The server must be able to accept UDP PING packets and reply with PING_ACKs on another port. The details on formats of both messages are described below.

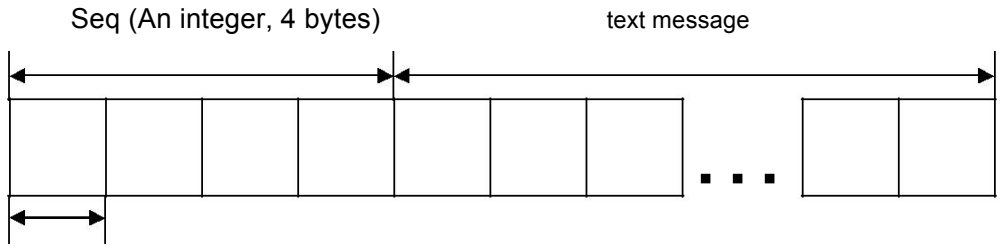
In order to be able to host 2 services at the same time, your server program (multi_service_server.<login1>.<login2>.c) must create two sockets, and concurrently listen on two different ports. We **REQUIRE** that you use **`select()`** to achieve that. So the structure of your program should be something like:



Note: We will grade your code only on Linux platform and we strongly encourage you to only use this platform for your development.

Message formats:

***PING*:** The head of a *PING* is a randomly generated **sequence number**, which is a 32-bit integer, followed by a **text message** no more than 64 bytes. Users can specify the content of the text message in the command line (described below). *PING* is sent from the client to the server.



PING

PING_ACK: Whenever a server gets a *PING*, it must reply with a *PING_ACK*. The head of the *PING_ACK* is also a sequence number (a 32-bit integer), whose value is the sequence number of the *PING* plus 1; the body is the same text message as in the *PING*.

For example, if the server got a *PING* with sequence number *99999* and text message *abc123ABC*, it then must send back a *PING_ACK* whose sequence number must be *100000*, and text message must be *abc123ABC*.

Command Line Arguments:

```
./multi_service_server  <http service port>  <ping service port>
```

Output:

The server output is not critical.

Testing:

We provide you with a sample client binary on the blackboard for your convenience to test your server. The client only runs on linux machines. Running it will perform several HTTP requests and send several PING messages, and report replies from the server. Use it to check that both types of requests being implemented are handled properly and that the proper *PING_ACK* response is sent.

Note: if you cannot run the client binary you might need to change permissions of the file once unzipped using `chmod 777`

The sample client can be run as follows:

```
./sample_client <servername> <server http port> <pathname> <server
ping port> <text message>
```

- <servername> is the name of the server
- <server http port> is the port at which the server runs the HTTP web service
- <pathname> is the path of the file relative to the server home directory. The file will be requested via HTTP.
- <server ping port> is the port at which the server runs the Ping service
- <text message> is the text message of PINGs sent to the server.

To help you verify your server, our client provides the following output: it will first show how many PINGs and HTTP requests it will send to the server, and then display all the PINGs sent and PING_ACKs received. It will NOT display any HTTP replies; instead, it will save the HTTP replies as separate files named httpreplies1, httpreplies2, etc. So please remember to check those HTTP replies to make sure your server handle HTTP requests correctly.

The following is a sample output on client's screen if you implemented the server correctly and your server is listening on ports 8080 and 8090

```
-bash-3.00$ ./sample_client enterprise.ecn.purdue.edu 8080 /alice.txt 8090 PINGU


3 Ping(s) and 2 HTTP request(s) scheduled


PING0:     Seq =     18956 Text = PINGU

PING_ACK0:  Seq =     18957 Text = PINGU

PING1:     Seq =     19031 Text = PINGU

PING_ACK1:  Seq =     19032 Text = PINGU

PING2:     Seq =     16249 Text = PINGU

PING_ACK2:  Seq =     16250 Text = PINGU


Ping finished
```

Deliverables:

For each part, your C code file(s) and the text file **<login1>.<login2>.txt** should be submitted using the "turnin" command.

Your text file <login1>.<login2>.txt should describe the status of your submission (i.e. whether all parts are working, if not, which parts are working, and where you think the problem is in the parts that do not work).

We will only test your code on the **linux** platform.

- Submission instructions & turnin command:

Log into `ecegrid.ecn.purdue.edu`

Move to the same directory as the file you are going to submit.

Execute the right command according to which part you are going to submit:

Part #	*Command*
Part 1	`turnin -c ece463 -p lab1pt1 httpclient.<login1>.<login2>.c <login1>.<login2>.txt`
Part 2&3	`turnin -c ece463 -p lab1pt2pt3 httpserver.<login1>.<login2>.c httpserver_fork.<login1>.<login2>.c <login1>.<login2>.txt`
Part 4	`turnin -c ece463 -p lab1pt4 multi_service_server.<login>.c <login>.txt`

After you execute the command you should get a confirmation immediately. For example, after execute the command for Part 1 you should see a message saying "*Your files have been submitted to ece463, lab1pt1 for grading.*". If you don't see the confirmation please contact the TA ASAP with your files attached in the email.

Reference

[Ste90] R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.

[BFF96] T. Berners-Lee, R. Fielding and H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*, RFC 1945, May 1996.

For other references, see blackboard.