ECE 463 Programming Project I

Fall 2020

Objective: You will create network applications on top of the transport layer using network programming (also called *socket programming)*. You will build a modified HTTP client and a modified HTTP server. 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	Thursday, September 10th 09:59 PM
Part 2 & 3	Thursday, September 17th 09:59 PM
Part 4	Thursday, September 24th 09:59 PM

Policies for this project

These policies are in addition to any other policies in the course syllabus.

Collaboration Policy: This project is to be done **individually. You must each independently complete the project and submit your own code.**

ABET Policy: You must score 40% on the project overall to meet the ABET outcome and obtain a passing grade in the course.

Testing Policy: Since different environments may use different versions of gcc, it is required that your code is tested on ECN computers. As your submitted code will be graded based off the version of gcc present on Purdue ECN computers, it is imperative that your code is tested within the same environment. You may use one of the methods outlined below for remote access, as these connect to ECN servers:

- ThinLinc Client Software
    - Download from here: https://www.cendio.com/thinlinc/download
    - Connect to server: ecegrid.ecn.purdue.edu
    - Log on using Purdue username and password
- ThinLinc Web Interface
    - Go to this website: https://ecegrid.ecn.purdue.edu/main/
    - Log on using Purdue username and password
- SSH:
    - Use SSH to connect to: ecegrid.ecn.purdue.edu
    - Log on using Purdue username and password

Port Binding Policy: All students should be testing their code on ECN computers. As this corresponds to a small number of Linux server machines, the issue that arises is that more than one student may attempt to bind to the same port on the same machine. This will cause an error. To avoid this problem, the following pattern should be used:

Port Number = 2000 + <your assigned class ID>

Please check Brightspace under 'My Grades' to receive your class ID.

If you attempt to bind to your port, and bind call fails, check the errno. If the result is "EADDRINUSE" or "The given address is already in use," please attempt increasing your port number by 1000, and repeat until an open port is found.

To check the errno if the bind call fails, please use the code provided below. perror will print the errno for you, and bind returns -1 upon failure:

```
if(bind(…) == -1){
     perror("Error binding to specified port!);
     return EXIT_FAILURE;
}
```

Eligibility to find partners for Project II: Project II to be released in mid-October will be done in groups of 2. However, you must score 60% on Project I to be eligible for a partner for Project II.

Cut-off time for answering questions: To discourage last minute work, we will stop answering questions one hour before the deadline on (i.e., 9 pm Thursdays on each of the days of the deadline). TAs will not be able to stay beyond the end of office hours even if there are students in the queue with unanswered questions. Anticipate long queues close to the deadline. Questions on Piazza may take upto 24 hours for a response on weekdays (and longer on weekends). Please plan accordingly and start early.

Late Policy

1. There will be a 11-hour grace period. That is, for the first 11 hours after the deadline, you may resubmit your project with no penalty imposed. To make use of the grace period, you **MUST** have submitted code with an appreciable amount of effort prior to the deadline. No questions will be answered during this grace period.
2. Beyond 11 hours, any submission will be subject to the late submission policy for programming projects described in the course syllabus. Barring serious medical situation or family emergencies (either situation must be accompanied by verification), late submissions require approval from Professor Rao prior to the deadline, and will incur stiff penalties based on the lateness of the submission (possibly resulting in a grade of zero). The hours late will be measured from the deadline date and time on the front page of the handouts.

Early Bird Policy: To incent students to start early and turn in strong submissions:

1. For Part I (HTTP Client), you can earn a 2.5% bonus on project 1 if you turn in the submission by Tuesday 11:59 pm September 8th, and get a grade above 90% of the score for Part I.

2. For Part 4 (full server), you can earn a 7.5% bonus on project 1 if you turn in the submission by Tuesday 11:59pm September 22nd and get a grade above 90% of the total score for the complete server code (i.e., code corresponding to Parts 2,3, and 4 together).

3. We won't be able to grade your code until the actual deadline is over. So, you should only do the early submission if you are confident that your code works well. You are allowed to submit a new version after the early bird deadline (until the actual deadline) – however, in this case we will use your latest code, and cannot award the bonus.

Note on automated grading:

Please make sure to thoroughly test your code as we will be using an automated test script and cannot look at your code to see where the error is. Due diligence in testing to capture bugs is part of the requirements for the project. Last minute poorly tested submissions may not pass our tests and could get a low score.

We require that you run your code on Thinlinc before submitting even if the code is running on your own machine. We require that you use Valgrind to check for potential memory leaks. The command below should be used for Valgrind.

```
$ valgrind --leak-check=full ./httpclient <command line arguments>
```

Any regrade appeal will only be considered if your code fully passes the valgrind test.

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. A traditional HTTP GET request will request a file from a server using the format listed below:

```
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 Part I of the project, you will implement a variant of the traditional HTTP client that we refer to as a two-stage HTTP client. The client uses the same GET command as the traditional client described above, but there are other changes that we describe later.
In parts 2-4, you will create a variant of the traditional HTTP server that we refer to as a letter-shift server. To interact with this server will require a variant of the traditional HTTP client (different than the one implemented in Part 1) that we refer to as a letter-shift client. We do not require that you rewrite your client to incorporate this 'letter-shift,' and will instead provide you with a letter-shift client binary file.

Below is the updated GET command for letter-shift clients and servers:

```
"GET <file path> <shift number> HTTP/1.0<CR><LF><CR><LF>"
```

The home directory of the server you will write in parts 2-4 is the directory where the server resides. 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 other information. For details about HTTP requests, please refer to RFC1945.

After the server has interpreted the GET request, it will first check if the file requested is available. If so, it will open the file, appropriately encrypt the file given the shift number, and send 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>
(encrypted 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://dtunes.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.
- **dtunes.ecn.purdue.edu** is the server hostname. You can use either an IP addressor 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

Part 1 (Two-Stage HTTP Client):

Use socket programming to write a “two-stage HTTP client” that will retrieve a file (simple text) from “any” HTTP server in two steps. In the first step, a GET request is sent to the server for a first file (say File1). The content of File1 is the name of a second file, File2. The client then sends a second GET request to retrieve File2. It is expected that your client:

1. Establishes a connection to the correct server port
2. Sends a “GET” command to the server.
3. Prints the server response.
4. If the response code is not 200 (HTTP OK), terminate the client program. Otherwise, continue to step 5.
5. Sends a second “GET” request to the server using the extracted content as the new file name
6. Prints the complete server response (response header + content)

Note: In HTTP 1.0, the server will **always** close the client connection after sending a response. Because of this, your two-stage client will have to reconnect to the server after the first GET request.

Your client must use the relative pathname in the GET command. For example, to get the page http://dtunes.ecn.purdue.edu/ece463/lab1/path_short.txt, your GET command sent to port 80 of dtunes.ecn.purdue.edu should be:

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

Note: Make sure to add \r\n (Carriage Return and Line Feed) at the end of the command otherwise the HTTP server won’t be able to send the response properly.

Command Line Arguments:

Your client (httpclient.<login>.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 dtunes.ecn.purdue.edu 80 /ece463/lab1/path_short.txt
```

Note: Throughout this document, when you see <login>, this means that you should replace it with your Purdue's login name. For example, if your login name is aturing then your client should be named

httpclient.aturing.c

Output

In your final submission, the client, when executed, must produce the server reply, header and content of the first request. If the server replied with an HTTP code of 200, it is expected that you should have sent a second GET request. Your client must also output the server's response to your second GET command. **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:

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

http://dtunes.ecn.purdue.edu/ece463/lab1/path_short.txt
http://dtunes.ecn.purdue.edu/ece463/lab1/path_very_long.txt
When testing, the files listed above are what should be entered into the command line arguments under 'pathname'.

When path_short.txt is retrieved from the server, its file contents will contain ONLY the name of the next file you should retrieve, which will be a short text file.

path_very_large.txt's file contents are the name of another file on the server, which you will need to retrieve. This will be a long text file.

It is imperative that you **DO NOT** hard code the 'next' file you should request based off of the inputted pathname. During grading, we will be changing the names of files and their file contents respectively. Additionally, we will be grading with other files for which their names are not known to you. This means that any hard-coded values will **NOT** work. It is critical that you extract the 'next' file to request from the file contents of the original GET request.

We have included a make file with instructions and comments to help you compile and run the client code. It is highly advised to compare the output file contents using the linux command diff. Assuming you have downloaded the files from Brightspace first, this can be done via the following commands:

```
./httpclient dtunes.ecn.purdue.edu 80 /ece463/lab1/path_short.txt >
output1.txt
diff output1.txt test_short.txt
```

There will be two differences between the compared files:

1. Your output file will contain the header and file contents of the first HTTP GET. This will simply be the HTTP header and a string of the next file you were supposed to receive.
2. Your output file will also contain the HTTP header of the second GET command.

If you are confused about the result of running diff, in your terminal, type `man diff` for more information on the output of this utility.

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

Submission:

See last section.

Part 2 (Letter-Shift HTTP Server):

Use socket programming to write your own version of a “letter-shift HTTP server”. This server must be able to accept a single connection at a time. There is no need to support multiple connections at this time. Additionally, your server must be able to interpret the modified “GET” command:

“GET <file path> <shift number> HTTP/1.0<CR><LF><CR><LF>”

Your server should ignore other HTTP commands. Your server must also encrypt files with Caesar Cipher using the shift number passed to the server in the GET command. Lastly, your server must reply with one of the following 3 HTTP responses described below. You will be required to ‘encrypt’ the requested file using the shift number supplied in the GET command and send the appropriate HTTP header and encrypted file contents back to the client. Note that you should only encrypt the file contents; **DO NOT** encrypt the HTTP header.

1. When the file is successfully found by the server, send back an HTTP 200 OK response as follows with the encrypted 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 encrypted file requested by the client, if the file is available.

Command Line Arguments:

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

./httpserver <serverport>

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

As all students will be testing their code on a common platform, the issue arises that students may attempt to bind to the same port on the same machine. Thus, it is expected that all students follow the policy outlined in the 'Port Binding Policy' defined above.

Caesar Cipher Examples:

Briefly, a Caesar Cipher is a shift based encryption that moves letters 'shift number' times from their original location. For example, with a shift number of 5, the letter 'f' becomes 'a'. When using Caesar cipher, you should only shift letters. Do not attempt to shift any numbers, special characters, line returns, etc. Be sure to account for both lowercase and uppercase letters. We will only be doing single digit shifts, in the range of 0 to 9 inclusive. Below are a few examples of shifts, where the top row is pre-shift, and the bottom row is post shift. Please note that the shifting essentially 'wraps' around the alphabet, such that shifting 'a' by 1 will result in 'z'.

Shift Number: 1

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y

Shift Number: 5

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u

Provided below is a copy of The Gettysburg Address in plain text, and then encrypted with Caesar Cipher using a shift number of 4. Note that none of the special characters are different, nor any numbers. Also note that all characters kept their original case:

Plaintext:

"Fourscore and seven years ago our fathers brought forth, on this continent, a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal." – Abraham Lincoln, November 19, 1863

Caesar Cipher Encrypted with Shift Number of 4:

"Bkqnoykna wjz oaraj uawno wck kqn bwpdano xnkqcdp bknpd, kj pdeo ykjpejajp, w jas jwpekj, ykjyaeraz ej hexanpu, wjz zazeywpaz pk pda lnklkoepekj pdwp whh iaj wna ynawpaz amqwh." – Wxnwdwi Hejykhj, Jkraixan 19, 1863

Testing:

A binary file called shift_client will be provided that will serve as a tool for testing your server. This binary file expects command line inputs in the following manner:
./shift_client <servername> <serverport> <pathname> <shift cipher number>

- <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.
- <shift cipher number> is the shift number for the above discussed Caesar Cipher

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 all test files we have provided – as with the client, the server may encounter different bugs with the larger 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.

Submission:

See last section.

Part 3: (Concurrent Web Server)

Use socket programming to write your own version of a “letter-shift 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. The server must retain the same ability to interpret GET commands with a shift number and appropriately encrypt the requested file.

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.

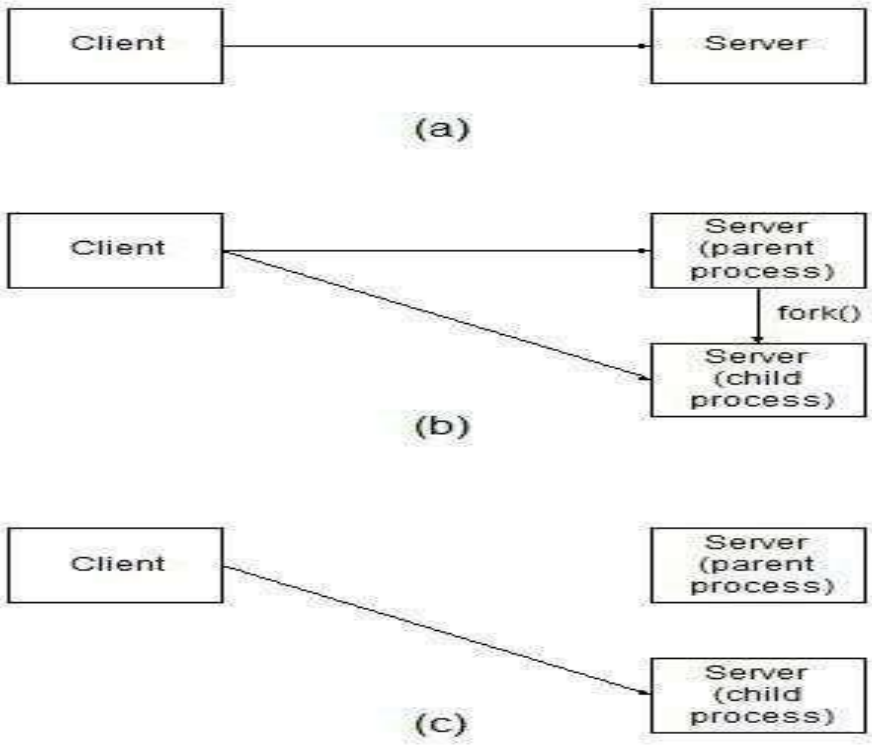


Fig. 1 The use of the fork() command to create child processes for a server allowing more than one connection at the same time. (a) The client is connected to the server; (b) the server calls fork() to create child process; (c) the parent process close the connection and wait for new connections while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

Command Line Arguments:

Your concurrent server (httpserver_fork.<login>.c) must support the same argument as httpserver from part 2 (i.e., port number):
./httpserver_fork <serverport>

Again, be sure to abide by the Port Binding Policy specified above when selecting your port to bind to.

Testing:

To check the working of your concurrent server, run two copies of the supplied shift_client 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 –ef | 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: Your server **should not** create more than 2 processes (you will see total three processes as shown above), if you are running two copies of shift_client simultaneously. If you see more than three then there is something wrong!

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!**

Submission:

See last section.

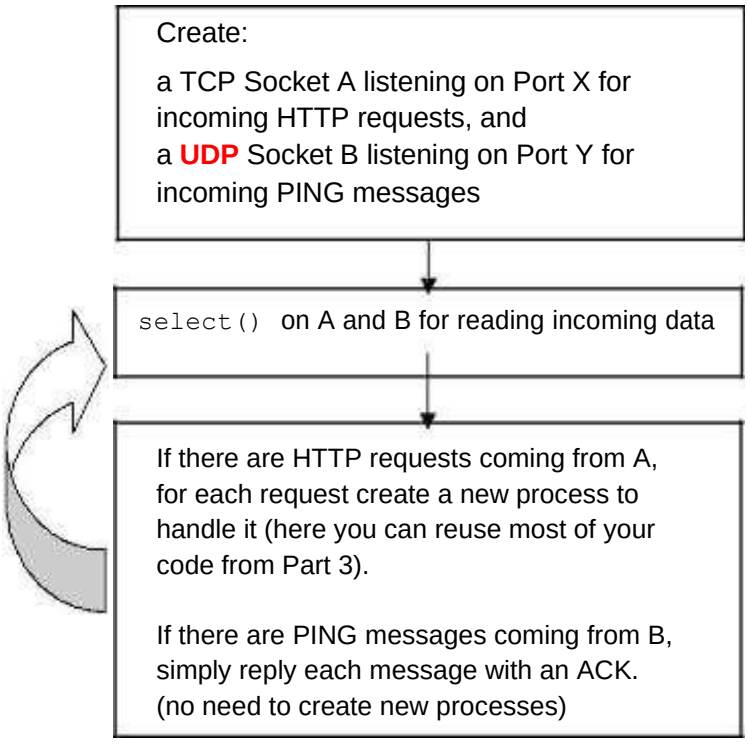
Part 4: (Multi-Service Server)

Use socket programming to write a “letter-shift multi-service server”. The server must be able to host 2 services simultaneously: HTTP web service with multiple connections at a time as specified in part 2 and 3, as well as 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. Again, the server must retain the ability to accept a shift number within the GET request and appropriately encrypt the requested file.

(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.<login>.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 an **IP address**, as a variable length string in dotted decimal notation. The IP address is not NULL terminated, and immediately followed by a randomly generated **sequence number**, which is a 32- bit unsigned integer. Users can specify the IP address in the command line (described below). *PING* is sent from the client to the server.

PING_ACK: Whenever your server gets a *PING*, it must reply with a *PING_ACK*. To accomplish this, the server needs to resolve the hostname of the IP address received in the ping. From here, it will construct a *PING_ACK* in which the head of the response is the hostname as a variable length string, immediately followed by the received 32 bit unsigned integer plus 1.

As an example, if the server got a *PING* with IP address '128.46.4.63' and sequence number *99999*, it then must send back a *PING_ACK* whose head is the resolved hostname, dynamo.ecn.purdue.edu, followed by the received sequence number + 1, 1*00000*. A block image of the received *PING* and the transmitted *PING_ACK* are shown below:

Received PING:

Value	128.46.4.63	99999
Description	IP Address in dotted formation (not NULL terminated)	Randomly generated sequence number
Data Type	Variable length string	32 bit unsigned integer

Transmitted PING_ACK:

Value	dynamo.ecn.purdue.edu	100000
Description	Resolved hostname (not NULL terminated)	Received sequence number plus one
Data Type	Variable length string	32 bit unsigned integer

Command Line Arguments:

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

Again, because of the nature of testing on a single linux server machine, it is important to abide by the Port Binding Policy specified above.

Output:

The server output is not critical.

Testing:

We provide you with a multi_service_client binary on the BrightSpace for your convenience to test your server. The client only runs on linux machines. Running it will perform two 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:

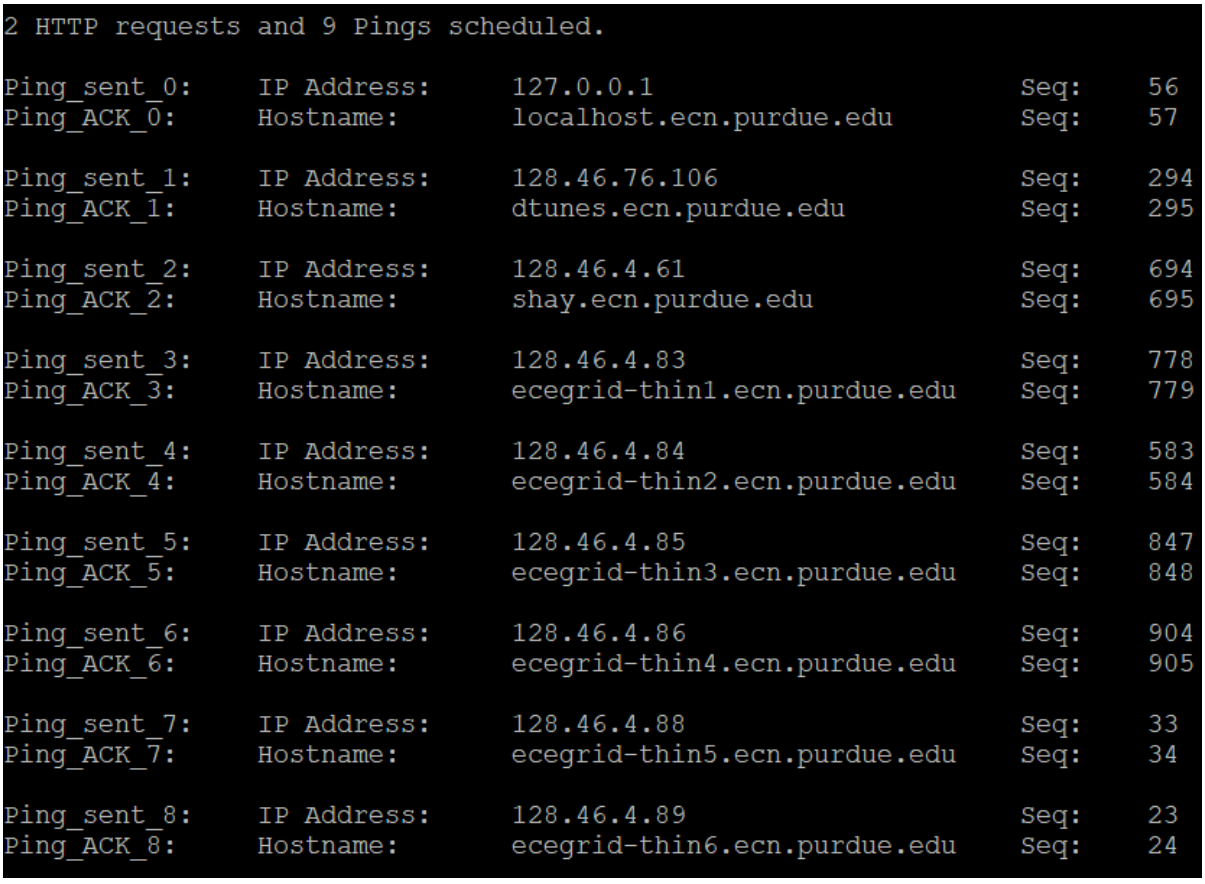
```
./multi_service_client <servername> <server http port> <pathname> <server ping port> <shift cipher number>
```

- <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
- <shift cipher number> is the shift number for Caesar encryption

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.

Below is an image of the result of running the provided multi_service_binary. Note that two HTTP GET requests will occur to test for concurrent service, and a random number of pings will be transmitted. sample_client will transmit anywhere from 2 to 9 pings with varying IP addresses corresponding to Purdue servers.



Deliverables:

For each part, please follow the following rules to submit your work: For part 1 (http client):

- Please use a filename "**httpclient.<login>.c**"
- Zip it, using the following command. "**zip httpclient httpclient.<login>.c**"
- Upload on Brightspace in in the folder "**Project submission" under "Lab 1 Part 1**"

For part 2 (simple http server):

- Please use a filename "**httpserver.<login>.c**"
- Zip it, using the following command. "**zip httpserver httpserver.<login>.c**"
- Upload on Brightspace in in the folder "**Project submission" under "Lab 1 Part 2**"

For part 3 (concurrent http server):

- Please use a filename "**httpserver_fork.<login>.c**"
- Please compress your file using the following command.
  "**zip httpserver_fork httpserver_fork.<login>.c**"
- Upload on Brightspace in in the folder "**Project submission" under "Lab 1 Part 3**"

For part 4 (Multi-Service Server):

- Please use a filename "**multi_service_server.<login>.c**"
- Please compress your file using the following command.
  "**zip multi_service_server multi_service_server.<login>.c**"
- Upload on Brightspace in in the folder "**Project submission" under "Lab 1 Part 4**"

Note: please use a **single** C file for each part.
We will only test your code on the **Linux** platform.

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 Brightspace.