ITS 332 – Assignment

1 Tasks

This assignment requires you to develop your own client/server applications in Ubuntu Linux. You have three tasks:

1. Develop a simple web server and command line based client that can retrieve pages from the server.
2. Enhance the client and server in Part 1 to allow for password based authentication.
3. Enhance the client and server in Part 1 to allow a proxy server to be used.

You should complete the tasks in this order.

2 Deadline

The deadline for submission (via email – see Section 8) is before the start of your last lab:

- Section 1 (IT): 9am Wednesday 13 February 2008
- Section 2 (IT): 1pm Wednesday 13 February 2008
- Section 3 (CS): 9am Monday 18 February 2008

You will have two lab classes to complete the assignment. It is expected during these 6 hours you can at least complete Part 1 and Part 2. The remaining must be completed in your own time before the deadline.

3 Plagiarism

Plagiarism includes: copying other peoples code; showing your code or significant part of it to other people; and telling other people what the code should be.

This is an individual assignment, and plagiarism will be heavily penalised. You must do the design and implementation on your own.

4 Marking

The allocation of marks to tasks is as follows:

- Part 1: 35%
- Part 2: 35%
- Part 3: 30%

Within each part, you will receive marks for:

- Successful tests: we will run several tests with your programs – they should produce the expected results. The tests will be run automatically, so it is important that you follow the instructions in this assignment – for example, naming of files, command line arguments, log output and message formats.
- Correct implementation: we will check your code to see if you have implemented the applications correctly.
- Well written code: we will check your code to see if you have structured your code well, including comments, appropriate use of constants and variable names, and good design.

5 Part 1 – Simple Web Server and Client

5.1 Task

Create a simple web server and a command line web page retriever.

The command line web page retriever must take three parameters:

- web server (hostname or IP address)
- port number of web server
- requested page

For example, if the web server is running on port `50001` on `192.168.1.3`, the client can be run as:

```
./client 192.168.1.3 50001 index.html
```

The client only requests a single page, and then exits. The output of the client may be:

1. If the requested page exists on the web server, then the client must save the received HTML as the requested file. In the above example, the received HTML (but not the 200 OK status code) would be saved as index.html. The client must also print a message to the screen:

    ```
    index.html retrieved and saved
    ```

    Then the client exits.
2. If the requested page does not exist on the web server, then the client prints a message to the screen:

    ```
    index.html not found on server
    ```

    Then the client exits.

The requests from the client will be for HTML web pages, and will follow a basic HTTP GET Request format:

```
GET url HTTP/1.1 server
```

where `url` is a file in the same directory as the web server, and `server` is the IP address (or host name/domain name) of the web server. We will not test the ability to handle directories (except in Part 2). Examples of valid values of `url` are:

```
index.html
file.html
hello.txt
```

Examples of invalid `url` (that is, values that we will NOT test, and hence you do not have to support) are:

```
/index.html
/directory/file.html
hello.jpg
```

The web server should receive requests from your client and send responses back.

The web server must take one parameter:

1. Port number of web server

For example, if listening on port `50001`:

```
./server 50001
```

The responses will include the HTTP Status Code message followed by the actual HTML web page (if it exists). There can be two different responses from the web server.

1. If the requested file exists (e.g. `index.html`), then the web server response will be of the format:

   ```
   HTTP/1.1 200 OK
   HTML of requested page
   ```

   where the actual HTML of the requested page is included.

2. If the requested file does not exist, then the web server response will be of the format:

   ```
   HTTP/1.1 404 File Not Found
   ```

The server must be able to handle connections from multiple clients, and will run continuously (that is, until the user hits Control-C).

The server must produce a log of events, and print them to the screen. The format of the server log is, for each request:

```
Time: <date and time of request>
Client: <IP address of requesting client>
Request: <the full HTTP request>
Response: <HTTP status code in response>
---
```

The trace later shows an example output. Note that the Request must include the entire request, whereas the Response only include the status code, e.g. “200 OK” or “404 File Not Found” - the HTML page is not included in the log.

5.2 Assumptions

To make your program simple (and hence your life easier), you can (and should) make the follow assumptions:

1. The maximum request size will be 256 bytes/characters.
2. The maximum response size (status code and HTML page) will be 4000 bytes/characters.
3. The client will only send a correctly formatted GET request (e.g. GET url HTTP/1.1 server).
4. The server will only send a correctly formatted response: either HTTP/1.1 200 OK followed by the web page; or HTTP/1.1 404 File Not Found.
5. The username will not be longer than 10 characters.
6. The password will not be longer than 10 characters.

The above assumptions mean you do not have to test for these conditions. For example, you do not have to test that the response is less than 4000 bytes – you can assume it always will be. (And we will not run tests that exceed these limits, or break these assumptions).

These assumptions also apply for Part 2 and Part 3.

5.3 Example Messages

The following figure shows an example of the messages sent. The first two messages are the result of the client requesting a page on the server. The second two messages are a result of the client requesting a page that does not exist on the server.

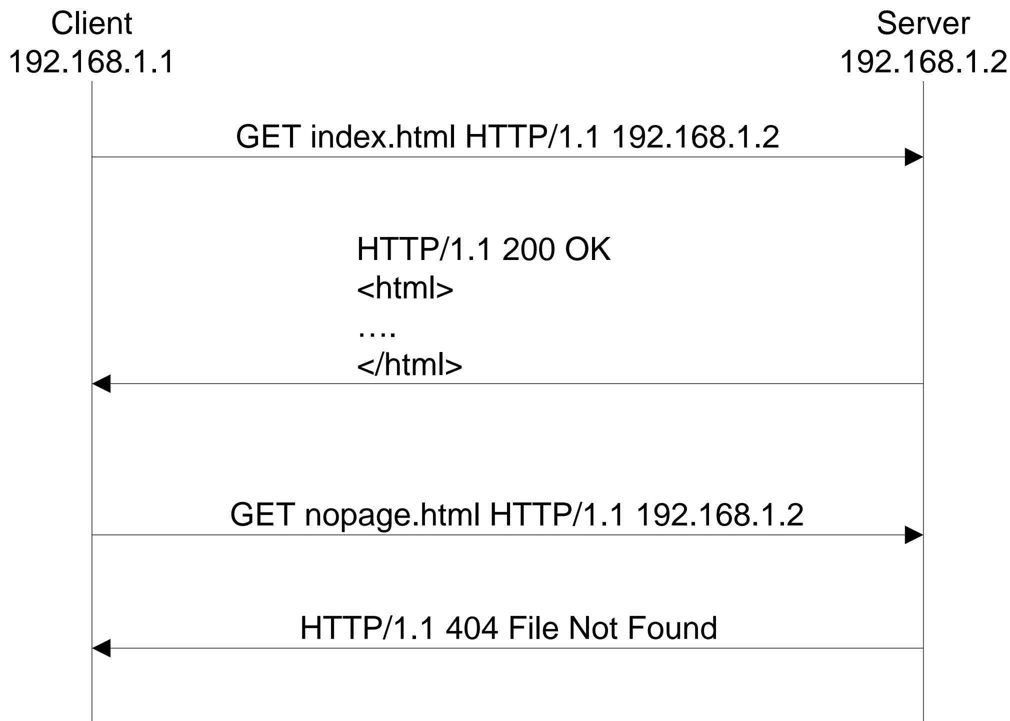

Figure 1

5.4 Deliverables

You must name your applications as follows:

- `client.c` – source code for client
- `client` – executable for client
- `client.log` – trace of the client output
- `server.c` – source code for server
- `server` – executable for server
- `server.log` – trace of the server output

Your log files should be traces from the command line, similar to those shown in Section 5.6. However, you do not have to show output from `ls` - only output from the programs running.

5.5 Running and Testing the Client and Server

In order to make your testing, and our testing (and marking) easier, you must follow the following instructions for developing your code.

You must create a directory using your ID number (in the following example, assume mine is `123456789` and I am in the `/home/student/` directory). Create the following directory structure:

```
/home/student/123456789/clientdir/
/home/student/123456789/serverdir/
```

`clientdir/` should contain the source and executable for the client application, and `serverdir/` should contain the source and executable for the server, as well as the test HTML pages.

You may include other test files, like `page.html` and `hello.txt` in `serverdir/` – and these of course will be downloaded to the `clientdir/` when you run your client.

When we test your system, we will execute your client and server in the above directories. We may also test your client with our server, and test our client with your server.

5.6 Example Trace

The following shows actual output from running the client and server we developed. You should obtain the same output from running your client and server (although the directories, file names and times will be different). When we test and mark your assignment, we will be checking that you get similar output.

Trace from client (running on chilli, 192.168.1.2):

```
sgordon@chilli:~/123456789/clientdir$ ls -l
total 44
-rwx------ 1 sgordon sgordon 10607 2007-02-03 17:26 client
-rwx------ 1 sgordon sgordon  4291 2007-02-03 17:28 client.c
-rwx------ 1 sgordon sgordon 11225 2007-02-03 17:48 client_secure
-rwx------ 1 sgordon sgordon  5486 2007-02-03 17:48 client_secure.c
drwx------ 2 sgordon sgordon  4096 2007-02-03 17:52 protected
sgordon@chilli:~/123456789/clientdir$ ./client 192.168.1.3 50001 index.html
index.html retrieved and saved
sgordon@chilli:~/123456789/clientdir$ ./client 192.168.1.3 50001 page1.html
page1.html retrieved and saved
sgordon@chilli:~/123456789/clientdir$ ./client 192.168.1.3 50001 page2.html
page2.html not found on server
sgordon@chilli:~/123456789/clientdir$ ls -l
total 52
-rwx------ 1 sgordon sgordon 10607 2007-02-03 17:26 client
-rwx------ 1 sgordon sgordon  4291 2007-02-03 17:28 client.c
-rwx------ 1 sgordon sgordon 11225 2007-02-03 17:48 client_secure
-rwx------ 1 sgordon sgordon  5486 2007-02-03 17:48 client_secure.c
-rw------- 1 sgordon sgordon  1103 2007-02-03 17:54 index.html
-rw------- 1 sgordon sgordon  1103 2007-02-03 17:54 page1.html
drwx------ 2 sgordon sgordon  4096 2007-02-03 17:52 protected
sgordon@chilli:~/123456789/clientdir$
```

Trace from server (running on basil, 192.168.1.3):

```
sgordon@basil:~/123456789/serverdir$ ls -l
total 52
-rw------- 1 sgordon sgordon  1103 2007-02-03 13:05 index.html
-rw------- 1 sgordon sgordon  1103 2007-02-03 13:05 page1.html
drwxr-xr-x 2 sgordon sgordon  4096 2007-02-03 17:06 protected
-rwxr-xr-x 1 sgordon sgordon 10771 2007-02-03 17:46 server
-rw------- 1 sgordon sgordon  4758 2007-02-03 17:45 server.c
-rwxr-xr-x 1 sgordon sgordon 11290 2007-02-03 17:46 server_secure
-rw------- 1 sgordon sgordon  6210 2007-02-03 17:45 server_secure.c
sgordon@basil:~/123456789/serverdir$ ./server 50001
Time: Sat Feb  3 17:48:24 2007
Client: 192.168.1.2
Request: GET index.html HTTP/1.1 192.168.1.3
Response: 200 OK
```

```
---
Time: Sat Feb  3 17:48:32 2007
Client: 192.168.1.2
Request: GET page1.html HTTP/1.1 192.168.1.3
Response: 200 OK
---
Time: Sat Feb  3 17:48:35 2007
Client: 192.168.1.2
Request: GET page2.html HTTP/1.1 192.168.1.3
Response: 404 File Not Found
---

sgordon@basil:~/123456789/serverdir$
```

6 Part 2 – Authenticated File Access

6.1 Task

Modify your web server and client to allow the server to offer password protected files (similar to HTTP Authentication).

Assume the web server stores password protected files in the subdirectory "`protected`". A request for the client for a URL `protected/file.html` will force the server to return a message saying "`401 Not Authorized`", so the client will then send the same request but with the username and password attached.

The message formats are now:

Initial request from client:

```
GET url HTTP/1.1 server
```

Response if requesting file in `protected/` directory:

```
HTTP/1.1 401 Not Authorized
```

Follow up request from client of a 401 Not Authorized was received:

```
GET url HTTP/1.1 server username password
```

Response from server if username/password correct and file sent.

```
HTTP/1.1 200 OK
```

The client does not have to cache username/password pairs. The command line interface to the client is the same as Part 1. However, if the `url` is for a file in the `protected/` directory, then after receiving a `401 Not Authorized` from the server, the client must prompt the user for a username and password, and then send the request to the server again.

You may allow multiple username/password pairs at the server, but you must allow at least the username `siit` and password `test`. (We will use that in testing your system).

If the username and password are incorrect, then the server will respond again with a HTTP/1.1 401 Not Authorized message, and the client will exit (showing a message "`Incorrect username/password for index.html`").

If the `url` is not for a file in the `protected` directory, then the client will perform the same as in Part 1.

If the `url` is for a file in the `protected` directory, but the file does not exist, the server returns a 404 message (instead of a 401 message).

6.2 Example Messages

Figure 2 shows the result of requesting a page in the `protected` directory.

Figure 2

Figure 3 shows an example of the user entering an incorrect password.

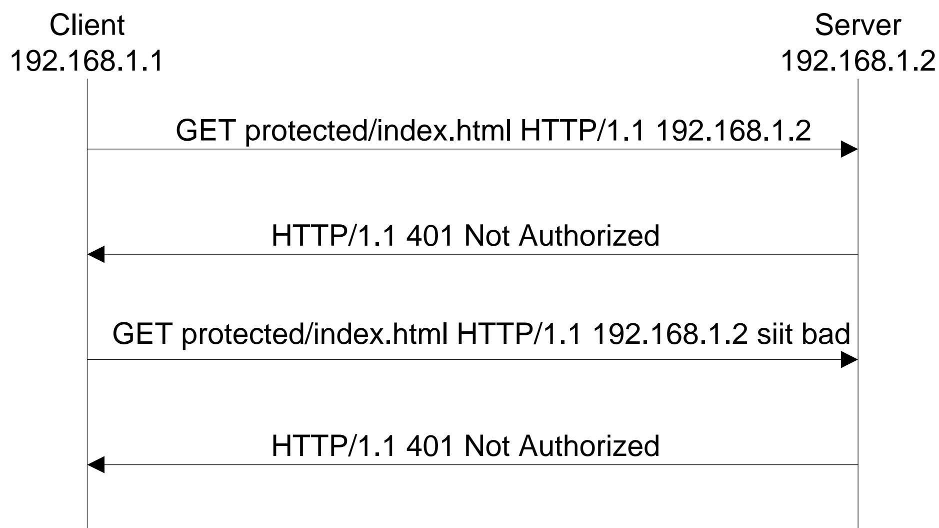

Figure 3

Figure 4 shows an example of the user requesting a page that does not exist in the protected directory.

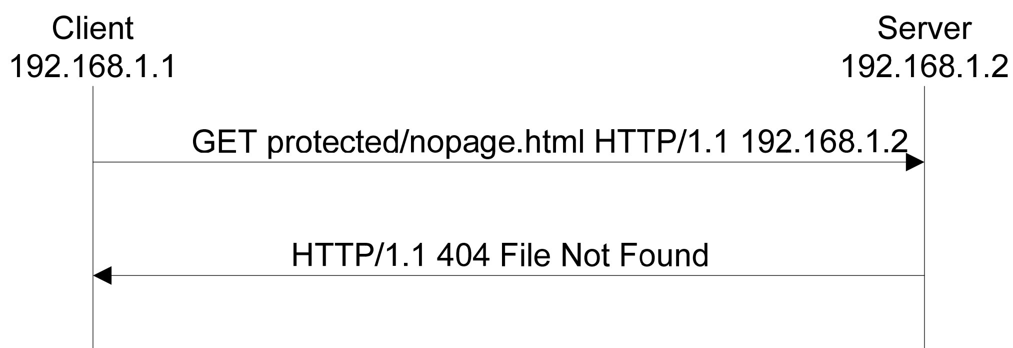

Figure 4

6.3 Deliverables

You must name your applications as follows:

- `client_secure.c` – source code for client
- `client_secure` – executable for client
- `client_secure.log` – trace of the client output
- `server_secure.c` – source code for server
- `server_secure` – executable for server
- `server_secure.log` – trace of the server output

6.4 Example Traces

Client Trace

```
sgordon@chilli:~/123456789/clientdir$ ls -l
total 44
-rwx------ 1 sgordon sgordon 10607 2007-02-03 17:26 client
-rwx------ 1 sgordon sgordon  4291 2007-02-03 17:28 client.c
-rwx------ 1 sgordon sgordon 11225 2007-02-03 17:48 client_secure
-rwx------ 1 sgordon sgordon  5486 2007-02-03 17:48 client_secure.c
drwx------ 2 sgordon sgordon  4096 2007-02-03 17:52 protected
sgordon@chilli:~/123456789/clientdir$ ls -l protected/
total 0
sgordon@chilli:~/123456789/clientdir$ ./client_secure 192.168.1.3 50001
protected/index.html
Username: siit
Password: test
protected/index.html retrieved and saved
sgordon@chilli:~/123456789/clientdir$ ls -l protected/
total 4
-rw------- 1 sgordon sgordon 1103 2007-02-03 18:02 index.html
sgordon@chilli:~/123456789/clientdir$ ./client_secure 192.168.1.3 50001
protected/index.html
Username: john
Password: test
Incorrect username/password for protected/index.html
sgordon@chilli:~/123456789/clientdir$ ./client_secure 192.168.1.3 50001
protected/test.html
protected/test.html not found on server
sgordon@chilli:~/123456789/clientdir$ ./client_secure 192.168.1.3 50001
index.html index.html retrieved and saved
sgordon@chilli:~/123456789/clientdir$
```

Server Trace

```
sgordon@basil:~/123456789/serverdir$ ls -l
total 52
-rw------- 1 sgordon sgordon  1103 2007-02-03 13:05 index.html
-rw------- 1 sgordon sgordon  1103 2007-02-03 13:05 page1.html
drwxr-xr-x 2 sgordon sgordon  4096 2007-02-03 17:06 protected
-rwxr-xr-x 1 sgordon sgordon 10771 2007-02-03 17:46 server
-rw------- 1 sgordon sgordon  4758 2007-02-03 17:45 server.c
-rwxr-xr-x 1 sgordon sgordon 11290 2007-02-03 17:46 server_secure
-rw------- 1 sgordon sgordon  6210 2007-02-03 17:45 server_secure.c
```

```
sgordon@basil:~/123456789/serverdir$ ls -l protected/
total 4
-rw------- 1 sgordon sgordon 1103 2007-02-03 17:06 index.html
sgordon@basil:~/123456789/serverdir$ ./server_secure 50001
Time: Sat Feb  3 17:55:53 2007
Client: 192.168.1.2
Request: GET protected/index.html HTTP/1.1 192.168.1.3
Response: 401 Not Authorized
---
Time: Sat Feb  3 17:55:58 2007
Client: 192.168.1.2
Request: GET protected/index.html HTTP/1.1 192.168.1.3 siit test
Response: 200 OK
---
Time: Sat Feb  3 17:56:11 2007
Client: 192.168.1.2
Request: GET protected/index.html HTTP/1.1 192.168.1.3
Response: 401 Not Authorized
---
Time: Sat Feb  3 17:56:16 2007
Client: 192.168.1.2
Request: GET protected/index.html HTTP/1.1 192.168.1.3 john test
Response: 401 Not Authorized
---
Time: Sat Feb  3 17:56:32 2007
Client: 192.168.1.2
Request: GET protected/test.html HTTP/1.1 192.168.1.3
Response: 404 File Not Found
---
Time: Sat Feb  3 17:56:42 2007
Client: 192.168.1.2
Request: GET index.html HTTP/1.1 192.168.1.3
Response: 200 OK
---

sgordon@basil:~/123456789/serverdir$
```

7 Part 3 – Create a Proxy Server

7.1 Task

Create a proxy server, that will intercept and forward requests from a client to real web server. A proxy is often used to provide security and performance enhancements to web access for users. When a client sends a request for a web page, it goes to the proxy. The proxy then sends the request to the real web server. The response from the real web server is sent to the proxy, and then the proxy forwards the response to the client. This allows the proxy to control what requests are allowed and also cache responses.

In this task you have to create the proxy server, which forwards requests and responses between client and web server.

You should copy your original client (from Part 1) to `client_proxy.c` and modify it to allow extra command line arguments specifying the proxy:

```
./client_proxy 192.168.1.2 192.168.1.3 50001 index.html
```

where the real web server is running on `192.168.1.2` and the proxy server on `192.168.1.3`, port 50001. Note that the client no longer specifies the port number of the real web server – it is assumed that it is a well-known port number.

The server should be the same as the server used in Part 1 (you do not need to modify it).

You should create a new directory for the proxy: `proxydir/`

The proxy should receive GET requests from the client and then send them (unmodified) to the real server. The response from the real server will be sent by the proxy (unmodified) back to the client.

```
./proxy 50001 50000
```

where the proxy listens on port 50001 and the well-known port number of the real server is 50000.

The proxy does not have to cache (save) pages that it receives from the real web server.

The proxy should keep a log in the same format as a real server, but with an additional line which is printed when the proxy forwards a response to the client:

```
Proxy: response sent to client
```

7.2 Example Messages

Figure 5 shows an example a client requesting a page on the server. The request (and response) go via the proxy.

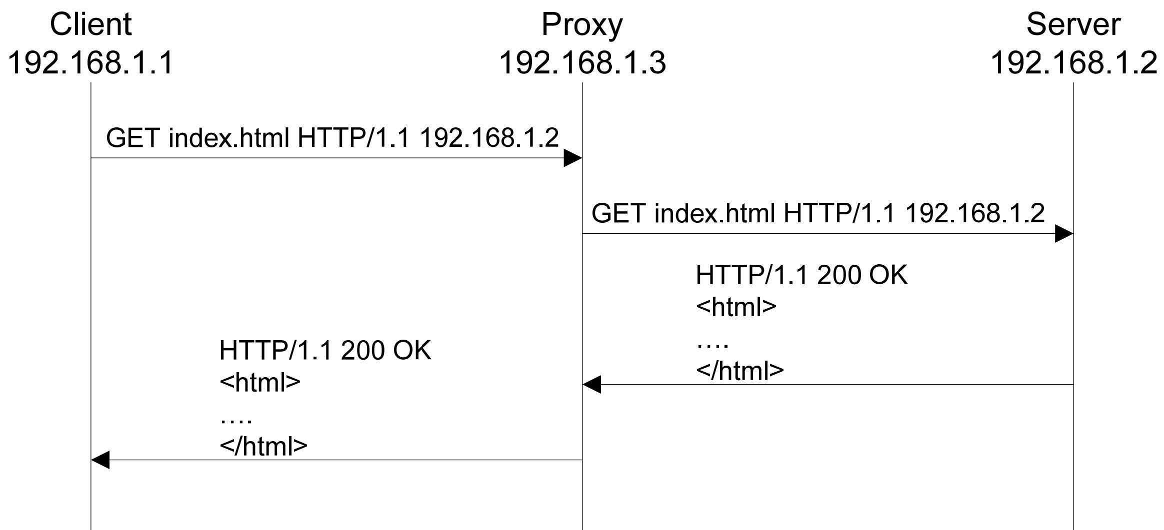

Figure 5

7.3 Deliverables

You must name your applications as follows:

- `client_proxy.c` – source code for client
- `client_proxy` – executable for client
- `client_proxy.log` – trace of the client output
- `proxy.c` – source code for server
- `proxy` – executable for server

- `proxy.log` – trace of the proxy output
- `server_proxy.log` – trace of the server output

Note that the server will be the same as that used in Part 1.

7.4 Example Traces

Client Trace

```
sgordon@chilli:~/123456789/clientdir$  ./client_proxy  localhost  chilli  50001
index.html
index.html retrieved and saved
sgordon@chilli:~/123456789/clientdir$
```

Proxy Trace

```
sgordon@chilli:~/123456789/proxydir$ ./proxy 50001 50000
Time: Sun Feb  4 10:21:56 2007
Client: 127.0.0.1
Request: GET index.html HTTP/1.1 localhost
Proxy: response sent to client
---

sgordon@chilli:~/123456789/proxydir$
```

Server Trace

```
sgordon@chilli:~/123456789/serverdir$ ./server 50000
Time: Sun Feb  4 10:21:56 2007
Client: 127.0.0.1
Request: GET index.html HTTP/1.1 localhost
Response: 200 OK
---

sgordon@chilli:~/123456789/serverdir$
```

8 Submission

After completion of all three tasks, your directory structure should be:

```
/home/student/123456789/clientdir/
                  client.c
                  client
                  client.log
                  client_secure.c
                  client_secure
                  client_secure.log
                  client_proxy.c
                  client_proxy
                  client_proxy.log
/home/student/123456789/serverdir/
                  index.html (you should include at least this test file)
                  server.c
                  server
                  server.log
                  server_secure.c
                  server_secure
                  server_secure.log
                  server_proxy.log
/home/student/123456789/proxydir/
                  proxy.c
                  proxy
                  proxy.log
```

You should archive all files using the following commands:

```
cd /home/student/

tar czvf its332-123456789.tgz 123456789/*
```

The result will be a single file in `/home/student` called `its332-123456789.tgz`. You can test that your archive worked by unpacking it using:

```
cd /home/student

mkdir test

mv its332-123456789.tgz test

cd test

tar xzvf its332-123456789.tgz
```

Attach the archive file to an email, and send the email to steve@siit.tu.ac.th with the subject:

`ITS332 Assignment 123456789`

Of course replace 123456789 with your ID number.

9 Useful C Hints

Below are some C functions and hints that may be useful for your applications.

`inet_ntoa()` takes an IP address in network byte order (the binary format used by the `sockaddr_in` structure) and converts it to a string.

If you include `<time.h>` you can use the following code to obtain the current date/time:

```
time_t now; /* define the variable now */
time(&now); /* store the current date/time in the now variable */
ctime(&now); /* converts the value in now to a string */
```

File manipulation can be done using `fopen` and `fclose`:

```
FILE fileptr; /* define a file pointer */
/* open a file for reading (r); for writing use "w" */
fileptr = fopen("filename.txt", "r");
/* Use fputs, fgets to write/read from file; or similar functions like fscanf
and fprintf */
feof(fileptr) /* will return 0 when you reach the end of a file */
fclose(fileptr); /* Close the file */
```

You may have to manipulate strings. There are many functions available, such as `strcpy()`, `strcmp()`, `strtok()`, `strcat()`: use `man string` to see the list of related functions.

`strtok()` may be useful to split a string into tokens. For example, for the string, `s = "My name is Steve"`, in the following code, `token` points to "`My`", then "`name`" and so on

```
char *token;

token = strtok(s," "); /* Now token points to "My" */
token = strtok(NULL," "); /* Now token points to "Name" */
token = strtok(NULL," "); /* Now token points to "is" */
token = strtok(NULL," "); /* Now token points to "Steve" */
token = strtok(NULL," "); /* Now token is NULL */
```

Be careful because the string `s` is modified for each call to `strtok`. If you want to use s later, then you should make a copy of if before applying `strtok`.