CS 485G: Systems Programming **Fall 2017**

Project 5: Network Programming and System I/O

Assigned: Nov. 17, 2017 *Due: Dec. 1, 2017, 11:59pm*

1 Introduction

Cloud computing has become very popular over the last decade. The underlying technologies for cloud computing include Internet communication and virtualization. One of the most popular services is cloud file storage; the ability to upload your data to the cloud and then to download it whenever your need it to any device that needs it.

In this project, you will write your own cloud file storage service. The goal of this project is to help you understand the basics about network programming using the client-server model. You will write both the server and the client.

Your code for this project must be written in C or C++ and must use the native Unix TCP/IP socket interface (i.e., socket calls) or the RIO library calls provided by the textbook (e.g., `open_clientfd`, `open_listenfd, Rio_writen, Rio_readn`) which can be found in the files csapp.c and csapp.h on the book's code web page: `http://csapp.cs.cmu.edu/3e/code.html`

2 The Interface

The project consists of two parts:

- Mycloud Client: it sends requests to store, retrieve, delete and list files on the Mycloud Server.
- Mycloud Server: it accepts requests from Mycloud Client and responds based on the requests from the client.

2.1 Mycloud Client

Mycloud Client runs as

```
mycloudclient ServerName TCPport SecretKey
```

where `ServerName` is the host name of Mycloud Server, `TCPport` is the port number the server listens to, and `SecretKey` is a 32-bit unsigned integer shared by the client and the server.

The client prompts "> " and takes commands from the standard input. It will process these commands and make requests to Mycloud Server. The client can process five commands:

- `cput FileName`

  The client will store the file named `FileName` in the local directory to Mycloud Server. The client sends a `STORE` message to the server.

- `cget FileName`

  The client will retrieve the file named `FileName` from Mycloud Server and store it in the local directory. The client sends a `RETRIEVE` message to the server.

- `cdelete FileName`

  The client will delete the file named `FileName` at Mycloud Server. The client sends a `DELETE` message to the server.

- `clist`

  The client will display a list of the files currently in Mycloud Server. The client sends a `LIST` message to the server. The server will provide the list of all files in the current directory the server is running.

- `quit` or `Ctrl-D`

  The client will terminate the current session with the server by closing the TCP connection to the server and exit the program.

The client will check whether the commands are well-formed and display the status (success/failure) of the result.

2.2 Mycloud Server

Mycloud Server runs as

`mycloudserver TCPport SecretKey`

where `TCPport` is the port number it listens to, and `SecretKey` is a 32-bit unsigned integer shared by the client and the server.

The server should listen for incoming TCP connections using the IPaddress `INADDR_ANY` at the port specified by `TCPport`. The server should only respond to requests that include a `SecretKey` - i.e., an integer number used to prevent unauthorized access to your server.

The server should accept incoming STORE, RETRIEVE, DELETE, or LIST requests from clients. All requests from a client should use the same TCP connection to the server. The details of the protocol used to exchange information between the client and the server will be described in the next section. You must verify the SecretKey for every request. The server should close the TCP connection if the key is not correct.

Your server should print out the following information for every request it receives:

- Secret Key = `SecretKey`, where `SecretKey` is the secret key contained in the request.
- Request Type = `type`, where `type` is one of store, retrieve, list, and delete.
- Filename = `filename`, where `filename` is the name of the file specified in the request (or 'NONE' if it is a list request).
- Operation Status = `status`, where `status` is either success or error.

2.3 Assumptions

Here are some assumptions you should assume when writing your code.

- Filenames will never be more than 80 characters long
- Filenames can contain any ASCII characters except `\0`.
- Files can contain binary or textual data.

- Files will not be longer than 100 KB.
- Your server will not provide service unless the client includes your server's SecretKey in its requests. The server and the client will get the SecretKey from the command line.
- The SecretKey will be an unsigned integer value in the range 0 to $2^{32} - 1$.

3 The Protocol

All requests from a client to Mycloud Server will use the same TCP connection over which you will send the request and receive a reply. The protocol (i.e., format of the messages sent between Mycloud Client and Mycloud Server) is described below for each type of requests/responses.

3.1 STORE Message

Store Request:

- Bytes 0-3: A 4 byte unsigned integer containing the type of request which is one of STORE (1), RETRIEVE (2), LIST (3), or DELETE (4) - in this case the value will be 1. Sent in network byte order.
- Bytes 4-7: A 4 byte unsigned integer containing `SecretKey` (stored in network byte order)
- Bytes 8-87: An 80 byte buffer containing the Filename (stored as a null terminated character string starting at the beginning of the buffer).
- Bytes 87-91: A 4 byte unsigned integer containing the number of bytes in the file (in network byte order). Assume the size of the file is n.
- Bytes 92-N: n bytes of file data, where $N = 91 + n$ and the size of the message being sent is $N + 1$.

Store Response:

- Bytes 0-3: A 4 byte integer containing the return status of operation Status. Status can be 0 (success) or -1 (error). Sent in network byte order.

3.2 RETRIEVE Message

Retrieve Request:

- Bytes 0-3: A 4 byte unsigned integer containing the type of request. It should be 2 (RETRIEVE) in this case. Sent in network byte order.
- Bytes 4-7: A 4 byte unsigned integer containing `SecretKey` (stored in network byte order)
- Bytes 8-87: An 80 byte buffer containing the Filename (stored as a null terminated character string starting at the beginning of the buffer).

Retrieve Response:

- Bytes 0-3: A 4 byte integer containing the return status of operation Status. Status can be 0 (success) or -1 (error). Sent in network byte order. If status is 0, it will contain the next two parts; otherwise this is the only field in the message and the total size of the message is 4 bytes.
- Bytes 4-7: A 4 byte unsigned integer containing the number of bytes in the file (in network byte order). Assume the size of the file is n.
- Bytes 8-N: n bytes of file data, where $N = 7 + n$.

3.3 DELETE Message

Delete Request:

- Bytes 0-3: A 4 byte unsigned integer containing the type of request. It should be 3 (DELETE) in this case. Sent in network byte order.
- Bytes 4-7: A 4 byte unsigned integer containing `SecretKey` (stored in network byte order)
- Bytes 8-87: An 80 byte buffer containing the Filename (stored as a null terminated character string starting at the beginning of the buffer).

Delete Response:

- Bytes 0-3: A 4 byte integer containing the return status of operation Status. Status can be 0 (success) or -1 (error). Sent in network byte order.

3.4 LIST Message

List Request:

- Bytes 0-3: A 4 byte unsigned integer containing the type of request. It should be 4 (LIST) in this case. Sent in network byte order.
- Bytes 4-7: A 4 byte unsigned integer containing `SecretKey` (stored in network byte order)

List Response:

- Bytes 0-3: A 4 byte integer containing the return status of operation Status. Status can be 0 (success) or -1 (error). Sent in network byte order. If status is 0, it will contain the next two parts; otherwise this is the only field in the message and the total size of the message is 4 bytes.
- Bytes 4-7: A 4 byte unsigned integer containing the number of the file listed (in network byte order). Assume the number of files is n. Note n is not the size of the next field.
- Bytes 8-N: $n * 80$ bytes of list data, where $N = 7 + n * 80$. It contains n file names, with each file name stored in an 80 byte buffer as a null terminated character string starting at the beginning of the buffer. [1]

[1] It can also be formatted as a list of file names separated by a newline character. However, it will need to calculate the total length of these file names.

4 Hints

1. We recommend that you start by modifying the echo client and the echo server provided in the notes and in the textbook. You can find a copy of the code online under the directory "netp" on the web page: `http://csapp.cs.cmu.edu/3e/code.html`

   The code you need to download include: `netp/echo.c,netp/echoclient.c, netp/echoserveri.c,include/csapp.h,src/csapp.c`. You can also use the functions such as `eval()` and `parseline()` from the last project.

2. One difference is that you will be sending binary data across the TCP connection (as opposed to lines read from a terminal). As a result you will want to use `Rio_readn()` and `Rio_writen()` rather than `Rio_readlineb()` because `Rio_readlineb()` is designed to read text lines one at a time and will have problems reading binary data. You will then need to send the appropriate information across the TCP connection based on the protocol description in the previous section.

3. You can send a string in the buffer over a TCP connection in the echo server/client code. It can be used to send the content from a binary file as well, except that you cannot use operations for string, such as `strlen(), strcpy()`,etc.

4. To send an integer `int x;` over a TCP connection, you can call `Rio_writen(fd, &x, 4)`. To receive an integer `int y;` over a TCP connection, you can call `Rio_readn(fd, &y, 4)`. Similarly, to send a structure $s1$ defined below, you can do `Rio_writen(fd, &s1, 12)`. Similarly, to receive a structure $s2$ defined below, you can do `Rio_readn(fd, &s2, 12)`. You can even just send the first 8 bytes of the structure by calling `Rio_writen(fd, &s1, 8)` and receive the first 8 bytes of the structure by calling `Rio_readn(fd, &s2, 8)`.

   ```
   struct message_t {
       int type;
       unsigned int code;
       unsigned int status;
       } s1, s2;
   ```

5. To send an integer in the network byte order, the convention is that you always convert an integer into the network byte order before sending. To send `int x;`, you can do `x = htonl(x); Rio_writen(fd, &x, 4);`. To get the correct value of an integer received from the network, you can convert it into host byte order after receiving it, i.e., `Rio_readn(fd, &y, 4); y = ntohl(y);`. This is necessary when you want your code to run without errors on two machines with different architectures with regard to byte order.

6. To read/write file in the current directory, you can use `fopen(), fread(), fwrite()` and `fclose()`. To find the size of a file, you can use `stat()` or use `fseek(), ftell(), rewind()`.

7. On the server side, you can assume the current directory in which the server is running as the cloud storage for the client. In order to list the files in the current directory, you can take a look at `readdir(), opendir(), closedir()` functions.

8. To delete a file, you can use the `remove()` function.

9. The secretkey is passed as an argument to the program in format of a string (either argv[3] or argv[2]). To convert it into an integer `int x;`, you can do `x=atoi(argv[3]);`.

5 Testing

Your code must compile and run on your VM. If you want to write your code on another machine and port it to your VM later, that is up to you, but it is your responsibility to get the code running on your VM before you submit it. You will not receive any credit if your code does not run on your VM - even it it runs elsewhere.

You can run your server on your VM and you can run your client on your VM as well. If you run both the client and the server on the same machine, you should run them from two different directories. Multilab machines are similar enough to your VM so that your code should run there without problems. You can test your programs using both your VM and Multilab machines to see whether they work over the network.

6 What to Submit?

You should submit your program as a single tarball, including .h and .c (.cc) file(s), Makefile, and README to the CSPortal at `https://www.cs.uky.edu/csportal`. The README file should give a general description about your program and state any limitations of the implementation. Robustness is one of important aspects when your code is tested. Comments are required.