CS 485, Fall 2014
Project 4: My Cloud File Storage
Due Date: December 5, 2014

Introduction

Over the past few years, the concept of a ``cloud'' where our programs and data can live has become very popular. What makes the cloud possible is a combination of the Internet communication (enabling communication with servers located anywhere in the world) and virtualization of resources (enabling providers to offer/sell storage and computation inside the network). The drivers for this shift include the economic savings and conveniences that come from moving our apps and data into the cloud. As a result, we are seeing companies race to put there services in the cloud, and users are rapidly embracing services in the cloud.

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. Today we see an seemingly endless list of cloud storage providers (e.g., amazon cloud drive, google drive, apple icloud, box, dropbox, MS skydrive, spideroak, etc.).

In this project, you will write your own cloud file storage service called *mycloud*. The goal of this project is to help you understand the basics about network programming. The project consists of two parts:

Mycloud Server:
: A *mycloud server* that accepts requests to store a file, retrieve a file, delete a file, and list the files stored on the server.

Mycloud Applications:
: Four client *mycloud applications* that will store, retrieve, delete, and list files on your mycloud server.

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_readnb*) which can be found on in the files csapp.c and csapp.h on the book's code web page:

> http://csapp.cs.cmu.edu/public/code.html

Groups

It is expected that you will work in groups of two on this project. You may select your own partner for this project. You may also decide to work alone on the project, but you must still do the entire project (i.e., there is no reduction in the amount of work you must do).

Assumptions

You should assume the following when writing your code:

- Machine Names (DNS names or dotted decimal IP addresses) will have a maximum length of 40 characters.
- 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.
- Files can be lost on the mycloud server if it exits or crashes. This is a horrible assumption, but

simplifies your server so that it can start out with an empty list of files every time is starts up.
- Your server will not provide service unless the client includes your server's *SecretKey* in its requests. The server and the clients will get the SecretKey from the command line.
- The SecretKey will be an unsigned integer value in the range 0 to $2^{32}-1$.

Mycloud Server

Your mycloud server should listen for incoming TCP connections using the IPaddress INADDR_ANY and a *TCPport* number specified as a command line parameter to your mycloud server. Your mycloud server should only respond to requests that include a *SecretKey* - i.e., a integer number used to prevent unauthorized access to your server (like a PIN number on your bank account). To run your mycloud server, you might type ``mycloud_server 5678 987654'' which would start your mycloud server listening for connections on the local machine at TCP port 5678, and would use 987654 as the SecretKey to prevent unauthorized access. See the example echo server from the notes and textbook as an example of how to create a server listening on a particular TCP port.

Your mycloud server should accept incoming file store, retrieve, delete, or list requests from clients. Each request will be a new connection to your server. In other words, your server should handle only one request (i.e., one store, or one retrieve, or one delete, or one list) per connection. The details of the protocol used to exchange information between your mycloud library and mycloud server is described below in Section 3. You must verify the SecretKey on every transaction.

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 get, put, del, or list.
- 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.

Mycloud Applications

There are four client-side applications you need to write. All of these programs (except mclist) take three command line arguments needed to communicate with the mycloud server (*MachineName*, *TCPport*, and *SecretKey*), followed by the *Filename* of the file to be stored, retrieved, or deleted.

The four programs you must write are:

mcput *MachineName TCPport SecretKey filename*

> The **mcput** program reads a file from standard input and then stores that data in a file called *filename* on the mycloud server identified by *MachineName* and *TCPport* using the number *SecretKey* as the password.

mcget *MachineName TCPport SecretKey filename*

> The **mcget** retrieves a file named *filename* from the mycloud server identified by *MachineName* and *TCPport* using the number *SecretKey* as the password. It then writes the file to standard output.

mcdel *MachineName TCPport SecretKey filename*

> The **mcdel** deletes the file named *filename* on the mycloud server identified by *MachineName* and *TCPport* using the number *SecretKey* as the password.

mclist *MachineName TCPport SecretKey*

The **mclist** retrieves a list of the filenames stored on the mycloud server identified by *MachineName* and *TCPport* using the number *SecretKey* as the password. It then writes the list of filenames to standard output, one filename per line of output.

The *MachineName* specifies the DNS name **or** the IP address (in dotted decimal format) of the machine where your mycloud server is running. *TCPport* specifies the TCP port number where your mycloud server is running. *SecretKey* specifies the number to use as the SecretKey when communicating with your mycloud server.

Creating a set of Library/API calls

We recommend that you write a *mycloud library* that could be used by any program, not just the four programs that you will write. Each of the library routines should takes as parameters the *MachineName*, *TCPport*, and *SecretKey* of the mycloud server. Example library routines that you might write would be:

int mycloud_putfile(char *MachineName, int TCPport, int SecretKey, char *Filename, char *data, int datalen)

This routine will send *datalen* bytes of data stored in the buffer pointed to by *data* to the mycloud server, where the mycloud server will somehow save the data and associated filename *Filename* for later getfile() requests. The return value will be 0 on success, and -1 for an error.

int mycloud_getfile(char *MachineName, int TCPport, int SecretKey, char *Filename, char *data, int datalen)

This routine will retrieve a maximum of *datalen* bytes from the file *Filename* stored on the mycloud server, and place the bytes retrieved into the buffer pointed at by *data*. The return value will be the number of bytes in the file, or -1 for an error.

int mycloud_delfile(char *MachineName, int TCPport, int SecretKey, char *Filename)

This routine will send a message to the mycloud server requesting that the file named *Filename* be deleted. The return value will be 0 on success, and -1 for an error.

int mycloud_listfiles(char *MachineName, int TCPport, int SecretKey, char *listbuf, int listbuflen)

This routine should send a message to the mycloud server requesting a listing of all the filesnames currently stored on the server. The listing should be stored into the buffer pointed to by *listbuf* which has a maximum size of *listbuflen*. The listing should list each file name on a separate (i.e., lines separated by `'\n'`) and the entire listing (a C string) should be terminated by a `'\0'`). The return value will be 0 on success, and -1 for an error.

Your mycloud library will create a new TCP connection for each putfile, getfile, delfile, or listfiles request. The connection will only last for the duration of the single request. The format to use for each request is described in Section 3 below. Your library will need to transmit the *SecretKey, Filename, datalen*, and *data* to the server for a putfile request. Getfile and delfile requests will only send the *SecretKey* and *Filename* to the server, with the *data* and *datalen* being returned in the case of a getfile request, and no *data* (just a success/failure value) returned for a delfile request. The listfiles request only sends the *SecretKey*, and gets back a buffer *listbuf* with a list of all the files stored on the server--one file name per line of the output.

Mycloud Protocol

Each request to your mycloud server will be a new TCP connection over which you will send the request and receive a reply. The protocol (i.e., format of the messages sent between you application and mycloud

server) is described below for each of the types of requests/responses.

1. Store Message Formats
   Store Request:
   - Bytes 0-3: A 4 byte unsigned integer containing *SecretKey* (stored in network byte order)
   - Bytes 4-7: A 4 byte unsigned integer containing the type of request which is one of get (0), put (1), del (2), or list (3) - in this case the value will be 1. Sent 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)
   - Bytes 92-N: N-92 bytes of file data, where N is the size of the message being sent.

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

2. Retrieve Message Formats
   Retrieve Request:
   - Bytes 0-3: A 4 byte unsigned integer containing *SecretKey* (stored in network byte order)
   - Bytes 4-7: A 4 byte unsigned integer containing the type of request which is one of get (0), put (1), del (2), or list (3) - in this case the value will be 0. Sent 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 unsigned integer containing the return status of operation *Status*. *Status* can be 0 (success) or -1 (error). Sent in network byte order.
   - Bytes 4-7: A 4 byte unsigned integer containing the *number of bytes in the file* (in network byte order)
   - Bytes 8-N: N-8 bytes of file data, where N is the size of the message being returned.

3. Delete Message Formats
   Delete Request:
   - Bytes 0-3: A 4 byte unsigned integer containing *SecretKey* (stored in network byte order)
   - Bytes 4-7: A 4 byte unsigned integer containing the type of request which is one of get (0), put (1), del (2), or list (3) - in this case the value will be 2. Sent 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 unsigned integer containing the return status of operation *Status*. *Status* can be 0 (success) or -1 (error). Sent in network byte order.

4. List Message Formats
   List Files Request:
   - Bytes 0-3: A 4 byte unsigned integer containing *SecretKey* (stored in network byte order)
   - Bytes 4-7: A 4 byte unsigned integer containing the type of request which is one of get (0), put (1), del (2), or list (3) - in this case the value will be 3. Sent in network byte order.

   List Files Response:
   - Bytes 0-3: A 4 byte unsigned integer containing the return status of operation *Status*. *Status* can be 0 (success) or -1 (error). Sent in network byte order.
   - Bytes 4-7: A 4 byte unsigned integer containing the *number of bytes in the file listing* (in network byte order)
   - Bytes 8-N: N-8 bytes of list data, where N is the size of the message.

Getting Started

We recommend that you start by modifying the echo client and 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/public/code.html

Unlike the echo server, your client applications will make one TCP connection and exit, so you can begin by modifying the code to send off a single echo request. Another 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 the Rio_readn() or Rio_readnb() 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 given in Section 3. On the server side, you will want to save the data along with the filename. The simplest way to save the data is to store it in a file, but you may save it in memory if you prefer. In either case, you need to be able to find the data when you receive a retrieve request.

Testing your Client Apps and Server

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 clients on your VM as well. If your server is saving files to the local directory using the filename passed in the request message, make sure that you do not run the client application and the server in the same directory as they might try to write the same file. If you would like to run your client applications on other machines to be sure you can read your files from anywhere in the Internet, you can run your client applications on the multilab machines (they are similar enough to your VMs so that your code should run there without problems).

To test your client side applications, you might type commands like the following:

```
bash $ alias runmcput='~/p4/client/mcput myloginid.netlab.uky.edu 6789 13579'
bash $ alias runmcget='~/p4/client/mcget myloginid.netlab.uky.edu 6789 13579'
bash $ alias runmcdel='~/p4/client/mcdel myloginid.netlab.uky.edu 6789 13579'
bash $ alias runmclist='~/p4/client/mclist myloginid.netlab.uky.edu 6789 13579'
bash $ cat /etc/services | runmcput UnixServices
bash $ cat /etc/issue | runmcput OSinfo
bash $ runmclist
UnixServices
OSInfo
bash $ runmcget OSinfo
Ubuntu 14.04.1 LTS \n \l

bash $ runmcdel OSinfo
bash $ runmclist
UnixServices
bash $ alias mclistbadkey='~/p4/client/mclist myloginid.netlab.uky.edu 6789 2468'
bash $ mclistbadkey
Error
bash $
```

The server might be run as follows:

```
bash $ ~/p4/server/mycloud_server 6789 13579
Secret Key = 13579
Request Type = put
Filename = UnixServices
Operation Status = success
--------------------------
Secret Key = 13579
Request Type = put
Filename = OSinfo
Operation Status = success
--------------------------
Secret Key = 13579
Request Type = list
Filename = NONE
Operation Status = success
--------------------------
Secret Key = 13579
Request Type = get
Filename = OSinfo
Operation Status = success
```

```
-------------------------
Secret Key = 13579
Request Type = del
Filename = OSinfo
Operation Status = success
-------------------------
Secret Key = 13579
Request Type = list
Filename = NONE
Operation Status = success
-------------------------
Secret Key = 2468
Request Type = list
Filename = NONE
Operation Status = error
-------------------------
^C
bash $
```

What to Submit

***** Your code must compile and run on the Ubuntu 12.04 VM that you have been assigned or you will receive no credit.**

Only one person from your group should submit the code. Consequently it is important the you remember to list both people from your group in the README file.

You will submit all your code plus a documentation file. You should **not** submit .o files or other binary files. To create your submission, tar and compress all files that you are submitting (e.g., tar czf proj4.tgz projectdirectory). You should submit the following:

- README file - listing all the files you think you are submitting along with your names. *Do not forget to list both people in your group, otherwise we will assume it was an individual project*.
- Documentation file - brief description of your projet including the algorithms you used. Your documentation file should also include a description of any special features or limitations of your shell. If you do not document a limitation, we will assume you did not know about it - and thus it will be considered a bug. The documentation file must be a text file. *Do not submit MS Word, postscript, or PDF files*.
- Makefile - we will type 'make', and we expect make will compile your program
- all your C/C++ files
- all your header files

Once you create the text file, go to https://www.cs.uky.edu/csportal to upload your file.

Note, you may upload your .tgz file as many times as you like. Note that the system timestamps your submission with the date/time of the last submission. The system does allow late submissions (which will be charged a late penalty).

James Griffioen 2014-12-04