CS 270 Project 5: A file server

Introduction

Many services are provided over the Internet by means of communication between clients (anywhere) and servers (on specific machines). These services include the secure shell (used by the ***ssh*** client), the network time service, and e-mail. Many cloud services such as file storage also fit into this category.

This project is for you to write your own file service. The goal of this project is to help you understand the basics about network programming. The project consists of two parts.

- A *server* running the program ***filed*** that accepts a set of valid requests (listed below),
- Three client application programs that make various requests to the server. There is a fourth, extra-credit application as well.

Your code for this project must be written in C or C++ and must use the native Unix TCP/IP socket interface (socket syscalls) or the **`csapp`** library calls provided by the textbook (such as **`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

You may work in groups of two on this project. You may select your own partner for this project; you do not need to select the same partner you had before. You may also decide to work alone on the project, but you must still do the entire project; there is no reduction in the amount of work you must do.

Assumptions

You should assume the following when writing your code:

- machineName (DNS names or dotted decimal IPv4 addresses) have a maximum length of 40 characters.
- File data can be either binary or text.
- Your server does not provide service unless the client includes your server's secretKey in its requests. The server and the clients get the initial secretKey from the command line.
- The secretKey is an unsigned 32-bit integer (in the range 0 to 2^{32}-1).

File Server

Your file server should listen for incoming connections using the IP address **`INADDR_ANY`** and a port number specified as a command-line parameter to your server. Your server should only respond to requests that include a secretKey: an integer used to prevent unauthorized access to your server (like a PIN on your bank account). To run your server, you might type **`filed 5678 987654`**, which would start your file server listening for connections at port **`5678`** with **`987654`** as the secretKey. See the example *echo* server from the notes and textbook as an example of how to create a server listening on a particular port.

Your file server should accept incoming requests from clients. Each request is in a new connection to your server. Your server should handle only one request per connection. The details of the protocol used to exchange information between your clients and the server is described below. You must verify the secretKey on every transaction.

Your server should print the following information to standard output for every valid request it receives:

Output	Note
`Secret key = `*`secretKey`*	*secretKey* is the secret key contained in the request.
`Request type = `*`type`*	*type* is one of **`newKey`**, **`fileGet`**, **`fileDigest`**, and for extra credit: **`fileRun`**.
`Completion = `*`status`*	*status* is either **`success`** or **`failure`**
`Detail = `*`detail`*	*detail* is specific to the particular request type, as shown below.

If the secret key is wrong, it is only necessary to print out the secretKey line, and the server may immediately close the connection to the client.

Applications

You must program three client-side applications, each a separate program. These programs take command-line parameters needed to communicate with the server: *machineName*, *port*, and *secretKey*, followed by specifics detailing the command.

Parameter	**Purpose**
machineName	the DNS name (like `george.netlab.uky.edu`) or the IP address (like `128.163.146.12`) of the machine where your file server is running
port	the TCP port number where your file server is listening
secretKey	the number to use as the secretKey when communicating with your server

In each of the descriptions below, ***`value`*** can be any sequence of characters not exceeding 100 bytes. The four programs you must write are:

- **`newKey`** ***`machineName port secretKey value`***

  `newKey` asks the server to set the secret key to a new value, which must be a valid integer in the range 0 to 2^{32}-1. The command must include the current secret key. The client prints either `success`, `failure` (if the server rejects the command) or `invalid` (if the value is invalid).

- **`fileGet`** ***`machineName port secretKey fileName`***

  `fileGet` asks the server to respond with the first 100 bytes of the given file (or fewer if the file is shorter). The server reports failure if the file does not exist or is not accessible. The client prints the bytes it receives from the server (in `%02x` format), or the single word `failure`.

- **`fileDigest`** ***`machineName port secretKey fileName`***

  `fileDigest` asks the server to generate a cryptographic digest of the given file, the result of this invocation: `/usr/bin/sha256sum fileName`. The result is guaranteed to be 100 bytes or less. The client prints the result it receives from the server (as a string), or the single word `failure`.

- **`fileRun`** ***`machineName port secretKey programName`***

  This command is for extra credit. ***`fileRun`*** asks the server to run a specified program and send the first 100 bytes (or fewer if there are fewer) of its output back to the client. Only certain programNames are valid; the others must be rejected. The valid values are:

programName	**run this command**
inet	`/bin/ip address`
hosts	`/bin/cat /etc/hosts`
service	`/bin/cat /etc/services`
identity	`/bin/hostname`

  The client prints the result it receives from the server or the single word `failure`.

Creating a set of Library/API calls

Construct a *fileServer library* that could be used by any client, not just the three or four client programs that you write. Each of the library routines should take as parameters the *machineName*, *port*, and *secretKey* of the filer server. Library routines that you should write:

- `int newKey(const char *machineName, unsigned short port, unsigned int secretKey, unsigned int newKey)`
- `int fileGet(const char *machineName, unsigned int port, unsigned int secretKey, const char *fileName, char *result, unsigned int *resultLength)`
- `int fileDigest(char *machineName, unsigned short port, unsigned int secretKey, const char *fileName, char *result, unsigned int *resultLength)`
- For extra credit: `int fileRun(const char *machineName, unsigned short port, unsigned int secretKey, const char *request, char *result, unsigned int *resultLength)`

Each of these library routines returns 0 on success and -1 on failure.

These library routines should build a new connection for each request. The connection only lasts for the duration of the single request. The format to use for each request is described below.

File-server Protocol

Each request to your server is carried by a new connection over which the client sends a request and receives a reply. The protocol, that is, the format of the messages sent between clients and server, is described below for each type of request and response.

Messages from client to server

Numbers in all messages must be in network byte order. All messages from the client to the server have this format:

- Bytes 0-3: A 4-byte unsigned integer containing *secretKey*.
- Bytes 4-5: A 2-byte unsigned integer (a short) containing the type of request: newKey (0), fileGet (1), fileDigest (2), fileRun (3).
- Bytes 6-7: Two bytes of padding, with arbitrary values.

The remainder of the bytes are command-specific.

- newKey request
  - Bytes 8-11: the new secret key, as an unsigned 32-bit int, in network byte order.
- fileGet request
  - Bytes 8-107: a null-terminated file name, no longer than 100 characters.
- fileDigest request
  - Bytes 8-107: a null-terminated file name, no longer than 100 characters.
- Run request (extra credit)
  - Byte 8-23: a 16-byte string (null terminated) holding one of the valid values.

Messages from server to client

Numbers in all messages must be in network byte order. If a client sends a request with an invalid secret key, the server should close the connection without returning any message at all.

All messages from the server to the client have this format:

- Byte 0: A 1-byte return code: 0 for success, -1 for failure.
- Bytes 1-3: Three bytes of padding, with arbitrary values.

The remainder of the bytes are command-specific.

- newKey response
  - No further data
- fileGet, fileDigest, and (extra credit) fileRun response
  - Bytes 4-5: A 2-byte unsigned integer (short) giving the length of the value, which must not exceed 100, including the concluding null (for a string value).
  - Bytes 6 ..: The value itself. The server need not send any more than the number of bytes required.

Getting Started

You might want to use this Makefile.

You could start by modifying the echo client and server provided in the notes and in the textbook, but the file-server clients and server are quite different from the echo client and server. 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.

Each client application makes one connection and exits, so you can begin by modifying the code to send a single echo request. Another difference is that your clients send data across the connection (as opposed to lines read from a terminal). As a result, you should use `Rio_readn()` or `Rio_readnb()` rather than `Rio_readlineb()`, because `Rio_readlineb()` is designed to read text lines one at a time.

Your applications need to send the appropriate information across the connection based on the protocol description given above. On the server side, you may use the `system()` library call instead of lower-level calls like `execvp()`.

Credit and extra credit

You should complete all the client programs. However, you can get 90% of the full server score by only handling `newKey` and `fileGet`. Handling `fileDigest` is worth another 10%, and handling `fileRun` is worth a bonus of 5%.

Testing your Client Applications and the Server

Your code must compile and run on your VM. You may write your code on another machine and port it to your VM later, but your code must run properly on your VM.

You can run both your server and the clients on your VM. If you would like to run your client applications on other machines, try running from your laptop (if it is running Linux) or a MultiLab machine such as **vio.cs.uky.edu**, **iri.cs.uky.edu**, or **pen.cs.uky.edu** (they are similar enough to your VMs that your code should run there without problems).

You can get working copies of the programs here (a ZIP file). You should test your clients and your server against the working program.

To test your client-side applications, you might type commands like the following and get responses like those shown. The responses depend, of course, on the particular machine running the server and its environment.

```
% host=`hostname`
% port=6564
% key=1411223
% newKey=1226631232
% ./newKey $host $port $key $newKey
success
% ./newKey $host $port $newKey $newKey
success
% ./newKey $host $port $key $key
failure
% ./fileGet $host $port $newKey /etc/hosts
3132372e302e302e31206c6f63616c686f73740a0a232054686520666f6c6c6f77696e67206c696e657320617265206465
% ./fileGet $host $port $newKey /bin/ls
7f454c46020101000000000000000000003003e0001000000050580000000000004000000000000000a003020000000000000
% ./fileDigest $host $port $newKey /etc/services
75d36dfa5c3f829d23f64750d6451641963c5a1e0c6e5d85fb83525755b1450c  /etc/services
% ./fileDigest $host $port $newKey /etc/sorvices
failure
% ./fileRun $host $port $newKey service
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a sing
% ./fileRun $host $port $newKey inet
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link
```

For that client input, the server might output as follows:

```
  % ~/CS270/project5/filed 6564 1411223
  Secret key = 1411223
  Request type = newKey
  Detail = 1226631232
  Completion = success
```

```
--------------------------
Secret key = 1226631232
Request type = newKey
Detail = 1226631232
Completion = success
--------------------------
Secret key = 1411223
Request type = newKey
Detail = 1411223
Completion = failure
--------------------------
Secret key = 1226631232
Request type = fileGet
Detail = /etc/hosts
Completion = success
--------------------------
Secret key = 1226631232
Request type = fileGet
Detail = /bin/ls
Completion = success
--------------------------
Secret key = 1226631232
Request type = fileDigest
Detail = /etc/services
Completion = success
--------------------------
Secret key = 1226631232
Request type = fileDigest
Detail = /etc/sorvices
Completion = failure
--------------------------
Secret key = 1226631232
Request type = run
Detail = service
Completion = success
--------------------------
Secret key = 1226631232
Request type = run
Detail = inet
Completion = success
--------------------------
```

Some hints

1. The clients and the server need to send and receive data. They don't need to send data all at once; they can send part of the required information and then send the rest. Likewise, they don't need to receive all the data at once; they can separate receiving information into multiple requests.
2. To capture the result of running a program, the server can redirect the program to a file such as **`/tmp/foo`**, then read that file to prepare for sending a result to the client. Fancier, but somewhat higher quality, is to use **`popen(3)`**.
3. If you are writing in C++, to include the **`csapp.h`** header:

```
extern "C"
{
  #include "csapp.h"
}
```

What to Submit

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

Only one person from your group should submit the code. Remember to list both people from your group in the `README` file. The other member of the group should submit a file saying "worked with (name your partner here)".

You should 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: `tar czf proj5.tgz` ***`projectdirectory`***. The tar package should include the following:

- `README` — listing all the files you are submitting along with your names, and documentation. The documentation should be a brief description of your project, including the algorithms you used. It should also include a description of any special features or limitations of your project. If you do not document a limitation, we will assume you did not know about it, and we will consider it a bug. **Do not submit MS Word, PostScript, or PDF files**.
- `Makefile` — we will type `make`, and we expect *`make`* to compile your program.
- all your C/C++ files
- all your header files

Upload your tar file as usual to https://www.cs.uky.edu/csportal. You may upload your project as many times as you like. The csportal timestamps your submission with the date/time of the last submission.