CS416/CS518, Fall 2014 - Assignment #3

Topic: A Simple Network File System

Out: October 31, 2014, Due: By 11:00am on December 1, 2014

Post any questions that you have on this homework to Sakai forums. No late submissions will be accepted. This homework assignment requires you to develop a significant amount of code, so get started early. If you've never programmed with sockets before, you will have to familiarize yourself with socket programming as well. You *will* need time to debug your programs, so start early. Don't postpone things to the last few days before the due date.

Overview

This assignment will give you the opportunity to understand how a simple distributed file system works. You will be implementing a simple network file server and its corresponding client. The server implements the "file system" by writing files to disk, while the client is implemented as a library that can be used by applications that wish to access the file system. All application requests to access files go through the client library, and are processed by the file server.

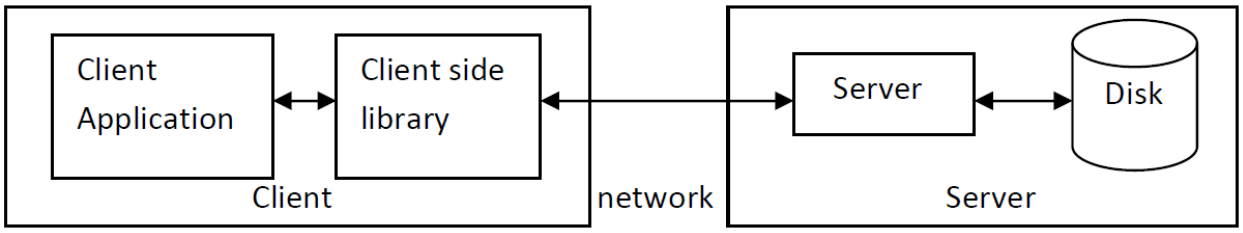
The overall architecture of the system that you will build will consist of the following three components:

- A server, serverSNFS, that listens to a port and executes the clients' requests. The server is multi-threaded and can serve multiple requests from clients at the same time.

- A client-side library, clientSNFS, that communicates with server. It handles accesses from the application and propagates them to the server. The application interface that the client library has to support is specified later in this specification.

- A simple client application program that will use all the APIs supported by the client library.

Pictorially, here is how your system should look when built:



The client library, the server, and the client application

We now describe the details of the API that the client library must export, the client application that you are required to implement, and the server.

Client Library API

The client library must present an API that client applications can use to access the file system via the file server. This API must export the following methods:

- `void setServer(char *serverIP, int port)` This method is called to provide the server IP and port addresses. The character string passed to this routine may contain a valid IP address (e.g., 192.168.1.1) or a hostname (e.g., scarlet.rutgers.edu) Your code must handle both cases. You can assume that the `setServer()` call is made before the `openFile()`.

- `int openFile(char *name)` takes the name of a file and returns a file descriptor or -1 on failure. If the file doesn't exist, it creates the file.

- `int readFile(int fd, void *buf)` attempts to read the whole file from file descriptor fd into the buffer starting at buf. On success, the number of bytes read is returned and -1 otherwise. You can assume file size will be less than 1KB.

- `int writeFile(int fd, void *buf)` writes the whole file from the buffer (pointed to by buf) to the file referred by the file descriptor fd. On success, the number of bytes written is returned and -1 otherwise.

- `int statFile(int fd, struct fileStat *buf)` returns information about a file. You need to define structure `fileStat`. It should at least contain file size, creation time, access time and modification time.

- `int closeFile(int fd)` relinquishes all control that the client had with the file. For example, after closing the file if client calls `readFile`, the request should fail.

The client library implements the API above and communicates with the server, which ultimately performs the file system accesses. The server address or name will be specified to the client library at startup time via the client API setServer(). You can use any port number for the client. The client library must be able deal with more than one open file at a time. For example, a client application can open two files simultaneously, reading from one file and writing into the other. To simplify the client library, there is no need to maintain cached copies of the data accessed.

A Sample Client Application

Your client library should be functional enough to support at least a simple client application, which does the following tasks:

- It should take the server IP, the server port and a valid file name as argument.

- Call `setServer` with the server information.

- Open a file named "`file.in`" using `openFile()` API.

- Copy the content of the file that is taken as argument, into the file "`file.in`" using `writeFile()`.

- Using `statFile()` print information of "`file.in`"
- Open another file "`reverse.in`".
- Read the content of the file "`file.in`" using `readFile()` API.
- Copy the content of the file "`file.in`" in reverse order into the file "`reverse.in`".
- Using `statFile()` print information of "`reverse.in`".
- Close the files.

Server details

The server emulates a file system by processing the requests from the client. The server is multi-threaded in the sense that whenever it receives a client request, it dynamically creates a new thread to service the request. After the request has been serviced, the thread kills itself.

Your server executable must accept and require these parameters:

- `-port port#`: TCP port # for fs communication.
- `-mount filepath` : place where files are to be stored by the server.

On server startup, if this directory already exists, the server should die with a message "directory already in use." If such a directory does not exist, it should create it. Do not remove the mount directory on server termination! For example, if the server is started with:

```
serverSNFS -port 12345 -mount /tmp/fs416
```

And a client opens file `"file1.txt"` using `openFile("file1.txt")`, the server should create the file: `/tmp/fs416/file1.txt.`

Information on Sockets

You have to use sockets for client-server communication. Below is a brief summary of the basic socket calls, the role they play in the client and the server, and the order in which they are called. The nformation is presented here to give you a starting place. You could also study the example code provided in the file SOCKETexample.tar.gz (in the Resources section on Sakai). While both of these approaches will help, there is no substitute for reading the manual pages. They are easily found using the man or xman commands.

The first important point is that the socket interface assumes that the client-server relationship holds between two processes. One is the server, which waits for requests at a fixed port address. The fixed address is the equivalent of waiting at a known phone number. The client requests service from the server by calling its number from anywhere it likes. The telephone analogy is not perfect, though, since the processes involved also execute system calls which are the equivalent of building and destroying the telephones, as well as establishing their numbers! The first system call of concern is the socket() call,

which creates a file descriptor attached to a socket structure. This creates the communication point in the process that calls it. Both the client and the server call socket(). Next, the server uses the bind() call to establish its address, and listen() to define the number of pending calls that may wait for attention from the server. The server then calls accept() when it wishes to accept a request for service. Note that accept() returns a new socket, unrelated to the original socket created by the server, which provides aconnection to the client making the request which has just been accepted.

After the client creates its socket using the socket()call, it needs to make a connection to the server. The client can first bind its socket to a specific address, although it need not do so. Whether it picks its port address or not, it requests a connection to the server using the connect() call. At this point the client and server are in communication. Each process reads and writes to the sockets as they would read and write to files. When finished, the client and server discard the sockets assuming the close() system call. The example code available from the web page above illustrates the use of sockets. The following pseudo code provides a summary of how the server and the client establish connections:

Server:

```
fd = socket();

bind(fd, server_address);

listen(fd, queue_length);

nfd = accept(fd, ...);

read requests; write answers;

close(nfd);

close(fd);
```

Client:

```
fd = socket();

[bind(fd, optional explicit client address)];

connect(fd, server_address);

write requests;

read answers;

close(fd);
```

Additional Notes

- Note that because of byte ordering conventions ("big endian" versus "little endian") on different machines, your client and server may not correctly interpret the contents of an exchanged message. If your client and server are running on different machines with different byte

- ordering conventions, they will not interpret the content of each others' structures correctly. You should use the htonl() and ntohl() functions to ensure that all machines interpret the byte orderings in the same manner.

- In writing your code, make sure to check for an error return from all system calls. If there is an error, the system declared global variable, errno, will give you information about the type of error that occurred:

  ```
  #include <errno.h>

  if ((sockid = socket(AF_INET, SOCK_STREAM, 0)) < 0) {

        printf("error creating client socket, error%d\n",errno);

        perror("meaning:"); exit(0);

  }
  ```

  See the man pages errno(2) and perror(2) for a description of the error codes and the use of perror.

- Make sure you close every socket (file descriptor) that you use in your program. If you abort your program, the socket may still hang around and the next time you try and bind a new socket to the port ID you previously used (but never closed), you may get an error. Also, please be aware that port ID's, when bound to sockets, are system-wide values and thus other students may be using the port number you are trying to use. With this in mind, I suggest you use a number as unique to yourself as is possible. An example would be to use the last four digits of your RUID and add it to 10,000 to create your port number. The reason for the addition is that port numbers below 5000 are reserved for system use, and those between 5,000-10,000 are used by lots of local applications. If you follow this rule the likelihood of colliding with another program using the same port at the same time is very low.

- Perform all time measurements within the server and client code as done in your warm-up homework. Alternative methods of time measurement (e.g., using `gettimeofday()`) are not acceptable.

- All development and measurements should be performed on the iLab cluster.

Deliverables

The following three deliverables are expected:

1. A tarball containing the source code (without executable files) for everything you've done, including all test files you used and scripts, etc. The more you comment the source files, the easier it will be for us to grade. At the very least, your source code should consist of 3 C files, serverSNFS.c, clientSNFS.c and clientapp.c. The tarball should also contain a Makefile that will allow us to build the executables for the server, the client library and the client application by simply typing 'make' on the command line.

2. A brief README file describing all the modules (.c files) of your source code. The README file should also contain a description of how to use, compile (flags needed, libraries used) and test your code.

3. A report describing what you did. The report should contain results showing the performance (throughput) of the server as a function of the number of threads that it uses (i.e., the number of concurrent requests that it receives). For these results, you will need to execute multiple client application programs at the same time. It is okay to change the file names in the client applications to prevent data sharing across applications. your report should explicitly state what exactly each member of the group did. It should contain your names, e-mails and student ID numbers. Your report should explain (in enough details for us to understand) what you've done, how you implemented and how you tested your work.