Advanced Distributed Systems

Project: Distributed File System
(Due by 11:59pm on 2 August 2019)

Objective

The overall goal of this project is to build a distributed file system using Sun RPC or Java RMI for distributed communication between the client and the server. Your file server will serve directory and file contents of the underlying Unix file system. You will not need to modify the kernel for this project. This is a group project with each group having no more than two students. A student who cannot find a partner needs to accomplish this project all by himself/herself. This project is worth 30 points towards your final grade

Problem statement

Distributed file systems allow local clients to access files and directories on remote servers. As discussed in class there are many issues in building a distributed file system and many approaches that can be taken to address these issues. For this project, you will be building a distributed file system client and server. These processes will communicate using either Sun RPC or Java RMI. The client will handle file system operations requests made by a user, translate them to appropriate remote procedure calls and print the result of each request. The initial current working directory for the server should be the Unix directory from which the server is started.

Operations

The project sets forth a number operations that users will make using a command line interface. Each operation will translate into one or more RPC (or RMI) calls. The following defines the set of directory and file operations that your client/server must support. The suggested RPC (or RMI) function call(s) for each operation are also shown in the form:

```
output_type function_name(input_type)
```

Directory Operations

All directory operations are with respect to the current directory (``.'') unless the directory_name begins with a ``/'' in which case a full path is specified. You need to implement the following directory operations:

- *getdir*

```
string getdir(void)
```

This function returns the current working directory as maintained by the remote server. Use *getwd()* if using C/C++ or the *getAbsolutePath* method of the File class if using Java.

- *cd directory_name*

```
boolean changedir(string)
```

This function changes the current working directory to the named directory on the server. The server should make sure the function succeeds. Use *chdir()* if using C/C++ or if using Java you will need to construct the new full path and test its validity with *getAbsolutePath*.

- *filecount*

```
int filecount(void)
```

This function returns the count of files and directories in the current directory. This count should include both visible and hidden (beginning with a ``.'') entries in the current directory.

- *ls [-l] [directory_name]*

```
boolean openlist(string)
direntry nextlist(void)
boolean closelist(void)
```

The *ls* and *ls -l* operations should produce output for all files in the current directory similar to the commands *ls* and *ls -l* (respectively) in Unix. Unlike the Unix *ls* command these operations will include all entries beginning with a ``.''. If the *directory_name* argument is missing then the current directory (``.'') should be used. If the *-l* option is omitted then the list of files should be given one per line. The *ls -l* operation should print the file name, its size in bytes and its last modified time. Both *ls* and *ls -l* should append a ``/'' to the end of the file name if it is a directory. It is not necessary to list the files in any particular order.

Your client should translate the *ls* operation into a series of function calls to the server (the server must maintain state between client calls). The `openlist()` call opens the directory for reading. If the `openlist()` call is successful then the `nextlist()` call should be repeatedly invoked to obtain the next entry in the directory. The information returned for each call should be a `direntry` structure that you define containing the entry name, a

flag indicating whether it is a directory, its size in bytes and its last modified time. When the last list entry has been read, your client should call `closelist`. If you are writing in C/C++, your server should use the Unix library calls *opendir()*, *readdir()* and *closedir()*. If you are writing in Java, your server should use the *File* class and its methods such as *isDirectory()*, *lastModified()*, *length()* and *list()*.

Sequential Access File Operations

Your program should support two file operations that access a local and remote file in sequential manner:

- *put localfile [remotefile]*

```
boolean openfiletowrite(string)
boolean nextwrite(block, int)
boolean closefile(void)
```

This operation should copy the contents of a local file (on the client) to a remote file on the server. If the optional *remotefile* name is not given then the remote file name should be the same as the local. Beware if the client and server are executing from the Unix same directory. The operation should be translated into a series of remote function calls to open the remote file for writing then write successive blocks of data read from a local file (assume a block is 512 bytes long and that is the maximum size that can be written in one call). When the local file contents have been transferred then the `closefile()` function should be called. Note that if using RPC you should not declare the `block` type as a `string`, but rather should use the type `opaque`, which will cause the RPC mechanism to not do run-time interpretation of the sent data.

- *get remotefile [localfile]*

boolean openfiletoread(string)
int,block nextread(void)
boolean closefile(void)

This operation is similar except that data is transferred from a remote file through a sequence of `nextread()` calls with each call needing the address of a block and the count of bytes returned. If the count is zero then the entire remote file has been read.

Random Access File Operations

Your program should support one file operation that allows random portions of a remote text file to be read:

- *randomread remotefile firstbyte numbytes*

boolean openfiletoread(string)
int,block randomread(int, int)
boolean closefile(void)

This operation can reuse the `openfiletoread()` and `closefile()` function calls, but introduces a `randomread()` call which allows up to 512 bytes to be read beginning at an arbitrary byte location. The results should be printed to the output.

Example

Your client program must have at least one command line argument, the name of the remote machine to contact for the file server. If the next argument is the ``-f filename'' switch then your client program should read in commands from a file. If the ``-f'' option is absent, then the commands are to be read from standard input. You should provide the user with a command prompt, but only if input is from standard input. An example set of operations and appropriate responses are given below. In general, each operation must print a single line response message (except for ls and ls -l as shown below). The basic response line must begin ```operation [succeeded|failed]`'' with additional explanation optional.

```
% client serverhost
$ getdir
getdir succeeded with /users/csfaculty/cew
$ ls
./
../ foo/
letter.tex prog1.c
$ filecount
filecount succeeded with count of 5
$ ls -l
./     512    Mon    Nov    19     21:12:29     EST    2001
../    4096   Mon    Nov    19     21:12:29     EST    2001
foo/   512    Mon    Nov    19     21:14:48     EST    2001
letter.tex   2935   Mon    Nov    19     21:26:01     EST    2001
prog1.c      1170   Mon    Nov    19     21:16:11     EST    2001
$ cd foo
cd succeeded
$ getdir
getdir succeeded    with   /users/csfaculty/cew/foo
$ ls -l
./           512 Mon Nov 19 21:15:29   EST    2001
../          4096 Mon Nov 19 21:15:29  EST    2001
$ cd bar
cd failed
$ cd ..
cd succeeded
$ get prog1.c localprog1.c
get succeeded transferring 1170 bytes
$ get prog2.c localprog2.c
get failed prog2.c not found
$ randomread letter.tex 1 14
randomread succeeded transferring 14 bytes
```

```
\documentclass$
```
Each input line contains one operation with any needed parameters separated by spaces. Again, your program *must* give the operation name and ``succeeded'' or ``failed'' after each operation other than for *ls* and *ls -l*. The format of any output after the success or failure indication is up to you, but it must be on a single line. The order of your output for the *ls* operations may differ from the example. Note that the output from the *randomread* operation may not terminate with a newline character. This result will cause the prompt to be printed on the same line as the output as shown in the example.

Your Task

The name of your client executable should be *client* and the name of your server executable should be *server*. It is suggested that you proceed in the following manner for this project:

1. Concentrate on the directory operations first. As a starting point to ensure that you understand how to access the underlying directory information, you should create a *single* program that implements the directory operations using the suggested function calls. This program *should not* use RPC or RMI. A correctly working program that implements the directory operations is worth 10 points on the project.

2. Once this program is working, break it in two so that you have a client and server program. You will need to use Sun RPC or Java RMI for the client stub functions to call the appropriate server stubs. Correctly working client and server programs that implement the directory operations are worth 18 points on the project.

3. Next implement the two sequential access file operations. Correct implementation of these two operations in your client/server architecture are worth 24 points on the project.

4. Next implement the randomly access file operation. Correct implementation of it is worth 27 points on the project.

5. For the final 3 points of the project, you will need to implement a filtering mechanism for the *filecount* and *ls* commands. The filtering must be done on the *server-side*, using arguments passed from the client. As part of the command line for *filecount* and *ls* the user can include any one (but no more than one) of the following:

   ```
   size[<>=]value
   age[<>=]value
   type=[d|r]
   ```

   These arguments allow users to list of count files with particular size attributes, age (current - last modified time) attributes or type attributes.

You will need to create an argument to pass with `filecount()` and `nextlist()` that will include the filter if given. If no filter is specified then these routines should work as previously described. Examples of their use:

```
$ filecount size>400
filecount succeeded with count of 1
$ ls age<1d
letter.tex
$ filecount type=d
filecount succeeded with count of 3
```

The first example counts all files with a size greater than 400 bytes. The second example lists all files with an age less than one day. The value is given as an integer followed by s, m, h, d indicating a unit of seconds, minutes, hours or days. The last example shows a count of all directories. A value of ``r'' for type counts or lists regular files.

Submission

1. Turn in your program files, makefile (if any) and script files showing runs for the sample file; do not turn in any executable files. Indicate in header comments of your code what portions of the project you attempted and completed.
2. Submit a Readme text file that includes (1) the names of your group members and the contributions of each member (who did what); (2) list each file enclosed in your submission package and briefly explain its function and usage
3. Please email your submission to both Dr. Tao Xie (garyxie20@yahoo.com) and teaching assistant Ruoyu Wang (wangry@shanghaitech.edu.cn) before 11:59 PM on 2 August 2019. Late submissions will receive zero point.