Assignment 2
Due: 24th November, 11:59 pm

In this assignment, you will build a simple distributed file system (DFS).
It is similar in spirit to the document sharing systems that we all use (e.g., Google Docs). However, it is very simple and provides much less functionality, as we are going to see.

The system allows users to access files from multiple locations in a transparent manner. A key aspect of this system is that it allows users to specify different types of consistency guarantees that they may require. The system consists of a DFS client and a server. The server address and port should be configurable (e.g., through command line parameters) and not hard coded into your programs.

The client runs on the machines of users and serves one or more user programs. The client implements key operations that are typically supported on files, such as opening a file or writing/reading a file. In order to complete these operations, it communicates with the server – the precise details of the protocol between the client and server, is left to you (we will have marks for the efficiency and effectiveness of your design). The server stores a copy of the file and plays a key role in implementing various consistency models. In the assignment we will assume that file size is limited to 16kB.

The DFS supports the following five functions:

createFile(char * filename)
The user will make this call to create a new file. File name length could be up to 30 characters. As a result of this call, the server will create a file with the specified file name in its local directory. If successful, this call will return 1; If a file with this name already exists, or there is some other problem the call will return -1.

openFile (char * filename, int mode)
The user will make this call to open a file. The mode could be either read (mode = 0) or write (mode = 1). Opening a file basically means creating a local copy of the file at the client and performing all subsequent reads on the local copy. The write operation depends on the type of write as we explain later. The system only allows one client to open a file in write mode at a given time. All subsequent requests to open a write return an error message.

If the file was successfully opened, the call returns a positive value, which is called the file descriptor. This file descriptor is subsequently used to perform other operations.

The call returns -1 if the file doesn't exist and -2 if the file is already opened in write mode at some other client.

readFile(int fd, char * buffer)

This call fills the buffer with the content of the file and returns the size of the file. You should do proper error handling/reporting – e.g., invalid file descriptor, etc.

writeFile(int fd, char * buffer, int length, int mode)

This call will write the file to the server/local copy depending on the mode. The file is stored in the *buffer* and its size is specified by the *length*.

Mode takes three options:

i) Blocking (mode = 0): In this mode, the file will be written to the server as well as all the active replicas – active replicas are all the clients who have currently opened the file in read mode. Once the server and replicas have the latest write then the call returns a positive value.

ii) Non-Blocking (mode = 1): In this mode, the write call returns as soon as the server completes the write operation. Note that the server still needs to update all the clients who have opened the file in read mode.

iii) Disconnected (mode = 2): In this mode, write is written to the local copy of the client. This is suitable for a mobile client who may not be connected to the server all the time. The server and other replicas are only updated when the user closes the file.

You should again do proper error handling/reporting for the write call (e.g., doing a write on a file descriptor that was only opened for reading should return an error).

closeFile(int fd)

As a result of this call, the reader/writer will close the file. In case of a reader, this will mean that future writes would not be communicated to the replica hosted at the client. So if the same user opens the file subsequently, then the file needs to be fetched again from the server (of course, you can implement optimizations where this overhead is minimized).

In case of a write user, if the client has a pending write (due to using disconnected mode earlier) then the write will first be made to the server and all the active replicas and only then the file close operation will take place.

In order to properly test the above functions, you will also need to supplement the client with a ‘user’ code that will be making these calls. In practice, the user of the DFS will be separate from the DFS client, but for this assignment you can have the user code as part of the client.

Implement all the five functions.

Each client must show this interactive menu:

----------MENU----------

1- Create a File.
2- Open a File.
3- Read File.
4- Write File.
5- Close File.
6- Quit.

Following should happen depending on the choice:

1- Ask for a filename and call **createFile** function. Display the output depending on the result returned by the function.
2- Ask for the filename and mode (read/write). Call the **openFile** function. Display the output depending on the result returned by the function. (Store the file Descriptors accordingly as they will be needed for other operations).
3- Ask for fileName. Call **readFile** function here. Display the content of the file or error message depending on the result returned by the function. The user must have opened this file in read or write mode, otherwise give error.
4- Ask for filename, length and mode. Create buffer according to the length. Prompt for input. Store that input in buffer and call **writeFile** function. The write must be according to the mode entered. The modes are very well explained in the description of the function. The user must have opened this file in write mode, otherwise give error.
5- Ask for filename. Call **closeFile** function here. IF that file was written in disconnected mode, take proper action. Remove the file – user association (if file was opened by user X in write mode, free the file now so that it may be opened by others waiting to open it in write mode).
6- The user quits. Call **closeFile** function on every file opened by the user irrespective of the mode.

Finally, both the client and server should be able to handle operations on multiple files at a given time. For example, a single user could open 3 files, perform various operations and then close them at the end.