

CS 571 - Operating Systems

Fall 2003

Project - A Distributed File System with Session Semantics

Preliminary Design Document: Due Nov 17

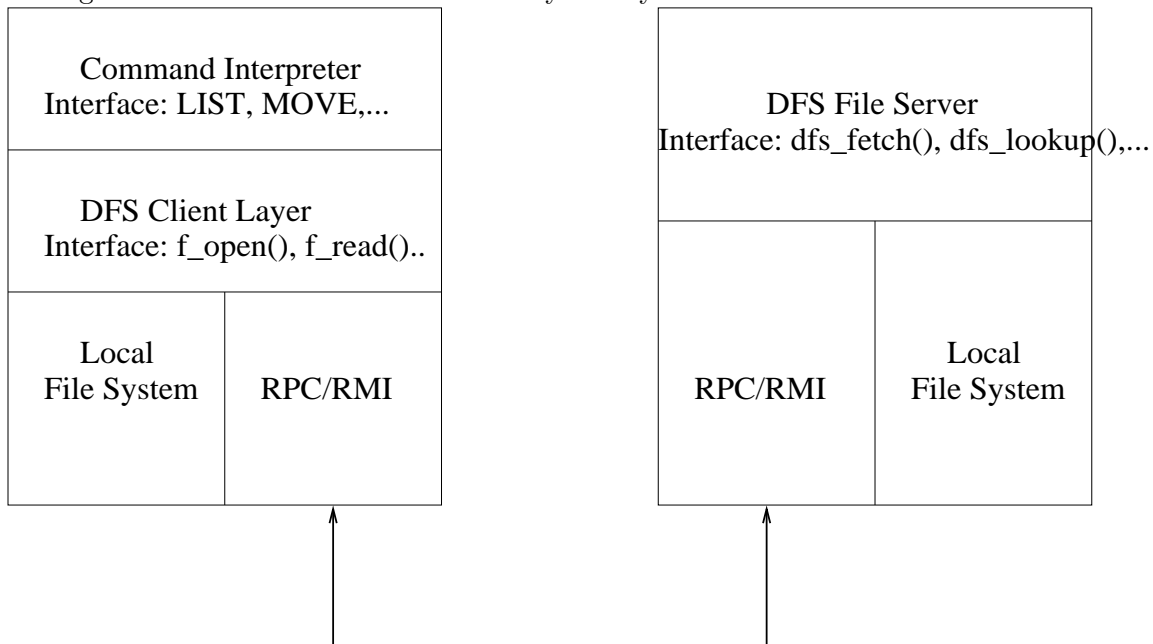
Working System: Due Dec 8

Your assignment is to implement a distributed file system and a simple interactive interface. You can use any programming language for this assignment. You may use any computing facilities that are available to you, however, you must be able to give me a demo in my office (using your own laptop) or the IT&E Lab. You can do this assignment in groups of two students.

You are to implement your distributed file system on top of the UNIX file system and Sun RPC/Java RMI. The distributed file system (DFS) should have **session semantics** and should use **whole file caching at the client**.

The interface and functionality of the file system are described later in this document. You are also to implement a simple, interactive command processing program which can be used to exercise and test the file system code.

The figure below illustrates the software layers in your DFS.



Your grade will be based on correctness of execution and on the quality of your design and code. At the end of the assignment, you will find a list of items you are expected to submit for grading.

The Distributed File System

The distributed file system has two components, the DFS Client and DFS Server, that communicate with each other using RPC/RMI.

DFS Server

You are to implement a distributed file system that is implemented on top of the UNIX file system. This means that the files in your DFS are stored as UNIX files – you do not have to write any software for mapping your files to disk blocks.

Your DFS should support a **flat file** and **directory service** such as that supported by AFS. This means that the files and directories in your DFS are stored in a single UNIX directory at the server. You may find the discussion in Chapter 8 of *Distributed Systems: Concepts & Design* by Coulouris *et al* on the design of a flat file system useful for this project.

The interface between the DFS Client and Server is left for you to design. However, a good starting point would be the flat file service and directory service in Chapter 8 of CDK3. Also, the fact that you are implementing session semantics should simplify the design.

DFS Client

The DFS Client software is responsible for making the distributed nature of the file system transparent to the user. It implements a UNIX-like hierarchical file system using the flat file and directory services provided by the DFS server.

It also uses the local disk (via the UNIX file system) to implement a file cache for improved performance. Basically, the DFS Client acts as a virtual file system layer that implements the functions described below. These functions are invoked by the command interpreter. These functions, in turn, either invoke DFS services via RPC/RMI or local UNIX file system calls.

Your file system's interface should include the functions described below. In these descriptions, the *fd* argument is a file descriptor, returned by a previous call to *f_open()*.

int f_open(char *pathname,int create-flag,char *mode) Open the file specified by *pathname*, if it exists. If the file does not exist, the behavior depends on the *create-flag*, which may be 1 or 0. If *create-flag* is 1, a new file with the specified path name should be created and opened (provided all of the directories on the path already exist). Note that a new file should not be created if the parent directory is read-only. If *create-flag* is 0, no new file is created, and the call should return an error. If a file is successfully opened, a file descriptor should be returned to the caller. The *mode* parameter should be one of the following strings: “rw”, “r”, or “w” which indicates the mode in which the file is to be accessed. Your procedure should do the appropriate error-checking to disallow illegal modes, e.g., a read-only file cannot be opened in a read-write mode.

int f_opendir(char *pathname) Open the directory specified by *pathname*, if it exists. Otherwise return an error code. A call to *f_opendir* is necessary before invoking *f_readdir* (described below) to read the contents of a directory.

int f_read(int fd,char *buffer-pointer, int nbytes) Copy *nbytes* bytes from the specified regular file to the buffer pointed to by *buffer-pointer*. If successful, the current file pointer for *fd* should be incremented by *nbytes*.

int f_write(int fd,char *buffer-pointer, int nbytes) Copy *nbytes* bytes from the buffer pointed to by *buffer-pointer* to the specified regular file. If successful, the current file pointer for *fd* should be incremented by *nbytes*.

int f_seek(int fd,int offset) Set the current file pointer of the specified regular file to the specified offset. The system should allow seeks at most one byte beyond the end of the file.

int f_readdir(int fd, char *buffer-pointer) Copy the next entry from the specified directory file into the specified buffer. The information copied into the specified buffer should consist only of the file name component of the directory entry. Any other information in the entry (such as an i-node number) should not be copied. Successive calls to this function should return successive directory entries. A return value of zero should be used to indicate that there are no more entries in the specified directory.

int f_close(int fd) Indicates that the specified file descriptor is no longer needed.

int f_closedir(int fd) Indicates that the specified directory file descriptor is no longer needed.

int f_rename(char *old-pathname, char *new-pathname) If the file or directory specified by *old-pathname* exists, change its name to *new-pathname*. All components of *new-pathname* (except the last) must already exist, and the parent directory of the renamed file should not be read-only.

int f_mkdir(char *pathname) Create a new directory with the specified name. All components of *pathname* (except the last) must already exist, and the new directory's parent should not be read-only.

int f_delete(char *pathname) Delete the specified file or directory, if it exists. The parent directory of the deleted file must not be read-only. Directories must be empty to be deleted. This call should return an error if *pathname* specifies a directory that is not empty.

int f_perm(char *pathname) Toggle the write permission for the specified file or directory, i.e., if write permission was on, turn it off, if it was off, turn it on.

info_t *f_getinfo(char *pathname) Report information about the specified file or directory in a structure of type *info_t* (that you have defined). The information should include the size of the file, its type (directory or regular) and its write permission (on or off).

int f_initialize() Initialize the device to support a new file system. Any existing files on the device are destroyed by this call. The new file system should have a single, empty root directory. The root directory should be in the normal mode (readable and writable).

Your file system procedures should return **negative** integers to report errors. All procedures, except *f_open*, *f_opendir*, *f_readdir*, and *f_getinfo*, should return 0 on successful completion. *f_open* and *f_opendir* should return positive integers corresponding to file and directory descriptors on successful completion. *f_readdir(dird, buf)* should return a positive number on successful completion (while placing the next entry in the directory *dird* in the character buffer *buf*) but when there are no more entries in the directory, it should return 0. A successful call to *f_getinfo()* should return a struct (of type *info_t*) that YOU define, and that is meaningful to the calling procedure in your implementation.

Any system call that cannot execute properly should return an error status code (a negative integer). This code should indicate the source of the problem. For example, the *f_mkdir()* function may be unable to create a directory because the specified path does not exist, or because the file system is full, or for some other reason. It is your responsibility to develop a comprehensive set of descriptive error codes for your system calls to return.

NOTE: If you use Java for this assignment, you will need to adapt the API described above as appropriate.

The Interactive Command Line Interface

To exercise and test your file system, you should write an interactive command-line interpreter. This program should accept commands typed at the keyboard and should produce output on the terminal screen. To implement commands (except for EDIT), the interpreter should use the file system interface described in the previous section. It should **not** use the UNIX file system or the DFS interface directly. You must implement at least the commands listed below.

LIST pathname If *pathname* is a directory, print a list of the files (or subdirectories) in it. For each file, the listing should also indicate the file's size, whether it is read-only or not, and whether it is a directory or not.

MOVE oldpathname newpathname Change the name of the file or directory specified by *oldpathname* to *newpathname*.

COPY oldpathname newpathname Make a copy of the file specified by *oldpathname*. This must be a regular file (not a directory). The new file should be called *newpathname*.

EDIT pathname Allow modification of the specified file. The file must be a regular file (not a directory). This command can be implemented by invoking your favorite editor on the local cached copy of the specified file. If no such file exists, a new file should be created if possible.

PRINT pathname Display the contents of the specified file. The *pathname* must specify a regular file, not a directory.

DELETE pathname Delete the specified file or directory. A directory must be empty to be deleted.

NEWDIR pathname Create a new, empty directory with the specified name.

RW pathname Toggle the write permission of the specified file or directory.

INITFS Make a new, empty file system on the remote device. This command destroys any existing file system and creates a new one containing only an empty root directory. Before making the new file system, the user should be queried to ensure that this command was not entered accidentally.

TEST This command should invoke a routine called *f_test()* which I will provide to you. The file `fstest.c` available on the Web page for the project contains the code for *f_test()*. You have to link this file to your code and use it without modification. The function *f_test()* basically tests your file system for correctness and performance. It creates a file called `testing.txt` which logs the actions taken by *f_test()* and their results.

You should link this function with your code only after you have completed implementing your file system. You may be able to modify the function to run specific tests as your implementation proceeds but the correctness and the performance of your system will be judged based on the tests in the *f_test()* code provided to you. **NOTE: If you use Java for doing this assignment, you will need to modify the code for the routine *f_test()* accordingly.**

EXIT This command ends an interactive session.

If the requested command cannot be completed because of an error, the interpreter program should print a message stating the cause of the error. The cause should be determined from the error code(s) returned by the file system calls used by the interpreter.

What to Submit

You should submit the following items by the project due date.

Source Code Submit source code for both the file system and the command interpreter. Your code should be neat, clear and well-documented.

File System Design Document This should be a short (3-4 pages) document giving an overview of the the design of your file system. The final design document should contain the following:

1. a description of the DFS file service and directory service operations (i.e., the API used plus a brief description of what each remote procedure/ method does.)
2. a description of how the DFS client is implemented, i.e., how does the DFS client map file system calls to DFS operations or local UNIX file system calls.
3. description of (i) how your system uses local disk caching to improve performance, and (ii) how it maintains cache consistency to support session semantics.
4. pseudo-code corresponding to the functions *f_open()*, *f_rename()*, *f_delete()*, and the commands **COPY** and **LIST**.

The preliminary design document should contain at least items (1),(2), and (3) above.

Test Script This should be a printout of the file `testing.txt` produced when the command **TEST** is invoked.

This is the schedule you have to follow for this project

Preliminary Design Document	Nov 17
Final Report and Demo	Dec 1 (office hrs), Dec 8, Dec 9

I encourage each student (group) to meet with me to discuss their file system design around the time they submit the design document. Demos will be scheduled on Dec 8th and 9th (and potentially also on Dec 1st). You should plan to come to campus for your demo. Your demo will take around 20 minutes. Guidelines for the final demo will be provided later.