CSS 434
Program 4: Distributed File System

Instructor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose

The final project is to design and implement a very simple distributed file system (DFS) in that a DFS client retrieves a file from a central DFS server and caches it in its /tmp directory for better performance. Our implementation uses Java RMI. Through this project, we will understand that (1) the design of both client and server is based on a state-transition diagram; (2) the server needs to maintain a directory of clients sharing the same file so as to implement delayed write and server-initiated invalidation; and (3) both server and client need to call each other's RMI functions.

2. DFS Model

Our DFS is based on the following schemes:

(1) One file cached in client's disk and multiple files cached in server's memory.
A DFS client program can cache just only one file in its local "/tmp" directory, whereas a DFS server can cache as many files as requested by DFS clients.

(2) Session semantics
A session in our system means a period of time while a file is being accessed or edited by a client's emacs text editor. In other words, a session starts from file download from a server with the read or the write mode, allows the file to be opened for read only or for modification with emacs, and completes when the emacs closes the file. A single writer but no multiple writer clients are allowed, regardless of the number of reader clients. Specifically, a client is guaranteed to modify a file exclusively, but may read a file being modified by someone else.

(3) Delayed write
A client can maintain a file it has modified in its local "/tmp" directory until it needs to download a different file from a server or the server sends the client a write-back request as indicating that there is someone else who wants to modify the same file. In either case, the client must upload (i.e., write back) the modified file to the server.

(4) Server-initiated invalidation
A server maintains a directory or a list of clients sharing the same file. If someone requests this file to download with the write mode, the server sends an invalidation message to all those clients that then invalidates their file copy. The server also sends a write-back message to the currently file-modifying client that then uploads the latest file contents to the server.

3. DFS Client Specification

Based on the DFS model discussed above, our DFS client program has the following specification.

(1) User input

Once invoked, the client program prints out the following message to start a file session.

```
FileClient: Next file to open
File name:
How(r/w):
```

A user types the name of a file he/she wants to access as well as the file access mode (such as read or write).

(2) File caching
Given such a user input, your client program first checks if it has cached this file locally in the "/tmp" directory. To be more specific, if your client program has cached a file, it has to maintain the file contents in "/tmp/youraccount.txt". For instance, if your account is mfukuda, the file contents should be cached as "/tmp/mfukuda.txt". The file attributes should be maintained internally in the client program. The attributes must include at least:

> **Name:** contains the name of a file whose contents are cached in /tmp/youraccount.txt
> **Access mode:** maintains the current file access mode such as read or write
> **Ownership:** indicates if this DFS client is owning (or exclusively writing) a cached file.
> **State:** indicates if this DFS client has no cached file, if a cached file is shared with other DFS clients, or if it is owned by another DFS client, etc.

The client program caches only one cache entry. In other words, it can cache up to one file at any point of time. The initial cached file state is "invalid".

(3) Downloading a new file
If the DFS client hasn't cached any file, it downloads the requested file from its DFS server. The request is performed through the server's RPC function:

```
	FileContents download( String myIpName, String filename, String mode );
```

As explained later, the DSF server needs to keep track of who has downloaded a given file, for which reason your client program must send the server its IP name as the first argument. The DFS client receives an instance of the FileContents class whose specification is as follows:

```
	import java.io.*;
	import java.util.*;

	public class FileContents implements Serializable {
	    private byte[] contents; // file contents
	    public FileContents( byte[] contents ) {
	       this.contents = contents;
	    }
	    public void print( ) throws IOException {
	       System.out.println( "FileContents = " + contents );
	    }
	    public byte[] get( ) {
	       return contents;
	    }
	}
```

The DFS client can then retrieve the file contents by calling the get( ) function of this FileContents instance. If the file access mode is “read”, the client must set the file state to “read_shared” in that the file is cached for read and may be shared with other clients. If the file access mode is “write”, the client must set the file state to “write_owned” in that the file is cached for write and can be modified exclusively while shared with other clients for reach.

(4) Opening a file with emacs
To allow a user to access a requested file, its DFS client program uses “emacs”. First of all, if the client program has just received new file contents from the server, it first saves these contents into “/tmp/useraccount.txt”. If it re-opens a cached file, it can simply re-use the “/tmp/uesraccount.txt” file. If the access mode is “read”, change the file mode of “/tmp/useraccount.txt” into read only, (i.e., chmod 400 /tmp/useraccount.txt). If the access mode is “write”, change its file mode into read/write, (i.e., chmod 600 /tmp/useraccount.txt). Finally, the DFS client opens /tmp/useraccount.txt with emacs:

```
emacs /tmp/useraccoun.txt
```

(5) Accessing a cached file
If a DFS client has found in its cache what a user has requested, it must check the cache state before reusing it. If the state is “read_shared” and the user wants to access the file in the read mode, the client program can simply read the file contents cached in the “/tmp/useraccount.txt” file. The cache state will not change. If the states is “read_shared” but the user wants to access the file in the write mode, the client program must obtain the file ownership from the DFS server. For this purpose, the client program calls the download(myIpName, filename, “w”) function to re-download this file from the server as requesting this file’s ownership. The DFS client should change the file state in “write_owned”. It can then start modifying the file contents.

If a DFS client has found the cached file in the “write_owned” state, it can immediately access the file in “/tmp/useraccount.txt” regardless of the file access mode. In other words, a DFS client can read and write the cached file. The file state stays in “write_owned”.

(6) File replacement
If a user wants to access a file different from what its DFS client program has been caching, the client program must first start its file replacement procedure. If the cached file state is “read_shared”, its latest contents must be owned by someone else, and therefore a file in “read_shared” does not have to be uploaded back to the server. If the cached file state is “write_owned”, its latest contents are cached locally and thus must be uploaded back to the server. To upload a file, the DFS client must call the server’s RMI function as follows:

```
boolean upload( String myIpName, String filename, FileContents contents );
```

The server needs three parameters: the client Ip name, the name of file to upload, and the file contents. After the file upload, the DFS client program must reset the cached file state to "invalid", (i.e., no files cached locally). Thereafter, the DFS client program follows the operations specified in (3) downloading a new file.

(7) Server-initiated operations
While a DFS client is accessing a file with emacs, another client may open the same file for modification, in which case the DFS server sends a signal to those who have shared this file. There are two types of signals: file invalidation and file write-back. The former is sent to those clients that have shared the file for a read, (i.e., in the Read_Shared state), and invalidates their cached file, (i.e., resets their file state to "Invalid"). The latter is sent to the client that has owned this file for a write, (i.e., in the Write_Owned state), and prompts this client to upload the cached file to the server.

A DFS client prepares two RMI functions called from the DFS server so as to receive file invalidation and file write-back signals.

```
boolean invalidate( ); // set the DFS client's file state to "Invalid"
boolean writeback( );  // request the DFS client to upload its current cache.
```

These two functions return true if they completes in success, otherwise false. The DFS server may call these functions in parallel while a DFS client is still accessing its cached file with emacs. Because of this reason, the writeback( ) function must wait uploading the cached file until the DFS client close the current emacs session. To implement this delayed upload, writeback( ) function should simply change the file state from "Write_Owned" to "Release_Ownership". As soon as the DFS client completes the current emacs session, it uploads the cached file contents to the server through the server's upload( ) function.

In summary, a DFS client program can be implemented using the following state-transition diagram.

State	Remarks	Input	Output	Next State
Invalid	No valid file cached local	Read a file	Call the DFS server's download(myIpName, filename, "r");	Read_Shared
		Write a file	Call the DFS server's download(myIpName, filename, "w");	Write_Owned
Read_Shared	A file is cached for read and may be shared with other sites.	Read the same file	Simply use /tmp's local file	Read_Shared
		Write the same file	Call the DFS server's download(myIpName, filename, "w");	Write_Owned
		Perform file replacement	Do nothing	Invalid
		Receive file invalidation	Do nothing	Invalid

Write_Owned	A file is cached and owned for write and may be shared with other readers.	Read the same file	Simply use / tmp's local file	Write_Owned
		Write the same file	Simply user /tmp's local file	Write_Owned
		Perform file replacement	Call the DFS server's upload(myIpName, filename, fileContents);	Invalid
		Receive file write-back	Do nothing	Release_Ownership
Release_Ownership	A file will be uploaded to the server soon.	The current emacs session is completed.	Call the DFS server's upload(myIpName, filename, fileContents);	Read_Shared

4. DSF Server

Our DFS server program has the following specification.

(1) File caching

The DFS server maintains a list (or a Vector) of file caching entries, each having the following attributes:

> **Name:** a cached file name
> **Readers:** a list (or a Vector) of IP names, each identifying a different DFS client who is sharing this file for read.
> **Owner:** The IP name of the DFS client who owns this file for modification.
> **State:** indicates four different states: "Not_Shared", "Read_Shared", "Write_Shared", "Ownership_Change".
> Data: a byte array that stores the file contents.

When the server is booted up, this list of cache entries is null. Whenever a DFS client calls the server's download( ) function, the server scans this list. If the server does not find a requested file in this list, it then creates a new cache entry, reads the file contents into the entry from its working directory, and adds the entry to the list. Once a new file is cached on memory, the server does not have to write it back to disk, because we assume that memory is large enough to cache as many files as possible.

(2) Downloading a new file to a client

A DFS client calls the server's download( ) RMI function to download a new file from the server:

```
        FileContents download( String myIpName, String filename, String mode );
```

Upon a call, this function first scans the list of file cache entries. As described above, if the server does not find a requested file's cache entry in this list, it must creates entry, reads the file contents into the entry from its working directory, and adds the entry to the list. The entry state is initialized to "Not_Shared".

The server now has to register the requested DFS client in this cache entry. If the client is downloading the file for read, its IP name is added to the readers list of this entry. Then, the server changes the entry state from "Not_Shared" to "Read_Shared". If the client is downloading the file for write, its IP name is registered in the owner field of this cache entry. The entry state changes from “Not_Shared” to “Write_Shared”. Finally, the server will store the cached file contents in a FileContent object and passes the DFS client as the download( )’s return value.

```
import java.io.*;
import java.util.*;

public class FileContents implements Serializable {
    private byte[] contents; // file contents
    public FileContents( byte[] contents ) {
       this.contents = contents;
    }
    public void print( ) throws IOException {
       System.out.println( "FileContents = " + contents );
    }
    public byte[] get( ) {
       return contents;
    }
}
```

Clients in the reader list remain until a new client writes new data back to the corresponding file, (i.e., until it calls upload( ).)

(3) Downloading a cached file to a client

If the DFS server finds a requested file in its list of file cache entries, it does not read the file from its working directory. Instead, the server updates the corresponding entry’s attributes as shown below:

(a) If the entry state is “Read_Shared” and the client is requesting file for a read, the entry state remains in “Read_Shared”. The server adds this client’s IP name to the reader’s list of this entry.
(b) If the entry state is “Read_Shared” but the client is requesting the file for a write, the entry state must be changed in “Write_Shared”. The server registers this clinet’s IP name in the owner field of this entry.
(c) If the entry state is “Write_Shared” and the client is requesting the file for a read, the entry state remains in “Write_Shared”. The server adds this client’s IP name to the reader’s list of this entry.
(d) If the entry state is “Write_Shared” and the client is requesting the file for a write, the entry state must go to “Ownership_Change”. The server calls the current owner’s writeback( ) function. The server must suspends the download( ) function until the owner client calls the server’s upload( ) function so as to actually write back the latest file contents to the server.
(e) If the entry state is “Ownership_Change” and the client is requesting the file for a read, the entry state remains in “Ownership_Change”. The server adds this client’s IP name to the reader’s list of this entry.
(f) If the entry state is “Ownership_Change” and the client is requesting the file for a write, the entry state remains in “Ownership_Change”. At this time, another client has suspended the download( ) function in the above 3-(d) case. The server must suspends the download( ) function until the above 3-(d) case is completed and the entry state changes in “Write_Shared”.

Except (d) and (f) where the download( ) is suspended, the server finally composes a FileContent object from the data field of this cache entry, and returns it to the client.

(4) Uploading a file from a client
A DFS client program calls the server's upload( ) RMI function so as to write back its cached file contents to the server. This function call is invoked:

(a) When a client has modified its cached file contents and now completes its program.
(b) When a client has modified its cached file contents and received a writeback( ) function call from the server.

```
boolean upload( String myIpName, String filename, FileContents contents );
```

The server checks if the corresponding file has been cached in it with the "Write_Shared" or the "Ownership_Change" state. If not, it simply returns false to the client. Otherwise, the server accepts the uploaded file contents in the corresponding entry's data field. It then calls the invalidate( ) function of each client registered in the readers list of this entry. Therefore, all copies over the system are invalidated. The server also takes the following actions, depending on the current entry state:

(a) If the entry state is "Write_Shared", the server changes the entry state to "Not_Shared".
(b) If the entry state is "Ownership_Change", the server changes the entry state to "Write_Shared" as well as resuming the download( ) function that has tried to download the same file for a write, (i.e., resuming the client suspended in the above 3-(d) case.) The resumed client can now complete its download( ) function call to download the file contents for a write. If there are any clients waiting on the above 3-(f) case, the resumed client must wake up one of them.

In summary, the DFS server program can be implemented using the following state-transition diagram

State	Remarks	Input	Output	Next State
Not_Shared	A file is cached but no client shares it.	download(clientIpName, filename, "r");	Add the client to the readers list. Return a FileContents object to the client.	Read_Shared
		download(clientIpName, filename, "w");	Register the client to the owner fieild. Return a FileContents object to the client.	Write_Shared
		upload(clientIpNmae, filename, fileContents);	Return false.	Not_Shared
Read_Shared	A file is shared among one or more clients for a read.	download(clientIpName, filename, "r");	Add the client to the readers list. Return a FileContents object to the client.	Read_Shared
		download(clientIpName, filename, "w");	Register the client to the owner fieild. Return a FileContents object to the client.	Write_Shared

		upload(clientIpNmae, filename, fileContents);	Return false.	Not_Shared
Write_Shared	A file is shared among one or more clients for a read, and owned by a client for a write.	download(clientIpName, filename, “r”);	Add the client to the readers list. Return a FileContents object to the client.	Write_Shared
		download(clientIpName, filename, “w”);	Call the current owner’s writeback( ) function, and thereafter suspends this download( ) function.	Ownership_Change
		upload(clientIpNmae, filename, fileContents);	Update the entry data with the given fileContents. Invalidate a copy of each client registered in the readers list.	Not_Shared
Ownership_Change	A file will be uploaded to the server soon.	download(clientIpName, filename, “r”);	Add the client to the readers list. Return a FileContents object to the client.	Ownership_Change
		download(clientIpName, filename, “w”);	Immediately suspend this download( ) function call.	Ownership_Change
		upload(clientIpNmae, filename, fileContents);	Update the entry data with the given fileContents. Invalidate a copy of each client registered in the readers list. Resume download( ) that has been suspended in Write_Shared. In other words, register the reusmed client to the owner fieild. Return a FileContents object to this client. (Just before the return, resume one of download( ) calls that has been suspended in Ownership_Change.	Write_Shared

5. Statement of Work

Code the DFS client and server programs as specified in the above.

6. What to Turn in

The homework is due at the beginning of class on the due date. Although this is a group project, each of you has to submit the following materials in a soft copy to “CollectIt for Homework” individually. Your soft copy should include:

(1) Your report in PDF or MS Word (All team member names must be written.)
(2) Source code (either within your report or separate .java files)
(3) Execution outputs (either within your report or separate .jpg/.pdf/.tif/.txt files)
(4) A confidential evaluation of your partner.

The grader’s preference is all in one report.

Go through the following test scenario and take a snapshot for each step. Each snapshot must be leveled with the corresponding step number.

CSS434 Final Project Test Scenario

1. Open 4 windows: uw1-320-20, uw1-320-21, uw1-320-22, and uw1-320-23.
2. Compile with javac and rmic.
3. Create two empty files: demoA and demoB.
3. Start a server on uw1-320-20.
4. Start a client on uw1-320-21, uw1-320-22, and uw1-320-23.

File read test:
5. Read empty from demoA at uw1-320-21. (A: clt:rs, svr:rs)

File write test:
6. Write xyz to demoA at uw1-320-21.   (A: clt:wo, svr:ws)
7. Read xyz from demoA at uw1-320-21.   (A: clt:wo, svr:ws)

File replacement test:
8. Write 123 to demoB at uw1-320-21.   (A: clt:iv, svr:ns, B: clt:wo, svr:ws)
9. Read xyz demoA at uw1-320-22.   (A: clt:rs, svr:rs)
10. Read xyz demoA at uw-320-23.   (A: clt:rs, svr:rs)

File writeback test:
11. Write xyz?! to demoA at uw1-320-21. (A: clt:wo, svr:ws, B: clt:iv, srv:ns)
12. Keep emacs open at uw1-320-21.   (A: clt:wo, svr:ws, B: clt:iv, srv:ns)
13. Write to demoA at uw1-320-22.   (A: clt:suspended, svr:oc)
14. close emacs at uw1-320-21.   (A: clt:rs, svr:ws)
15. Write xyz?!abc to demoA at uw1-320-22(A: clt:wo, srv;ws)

Session semantics read test:
16. Read xyz?! from demoA at uw1-320-21.(A: clt:rs, svr:ws)
17. Read xyz?! from demoA at uw1-320-23.(A: clt:rs, svr:ws)

Multiple write test:
18. Write 123pqr to demoB at uw1-320-21.(A: clt:iv, svr:ws, B: clt:wo, svr:ws)
19. Keep emacs open at uw1-320-21.

20. Write 123pqr456 to demoB at uw1-320-22.(A: clt:iv, svr:rs, B: clt:suspended, svr:oc)
21. Write 123pqr456abc to demoB at UW1-320-23. (A: clt:iv, srv:ns, B: clt:suspended, svr:oc)
22. Close emacs at uw1-320-21. (A: clt:iv, svr:ws)
23. Close emacs at uw1-320-22. (A: clt:iv, svr:ws)
24. Close emacs at uw1-320-23. (A: clt:wo, svr:ws)

25. quit uw1-320-20, uw1-320-21, uw1-320-22, and uw1-320-23.
26. Check demoA and demoB with cat demoA == "xyz?!abc" demoB == "123"

CSS434 Final Project Grading Sheet

Criteria	Grade
Documentation Write in 1-2 pages about only your unique implementation that is different from the original specification, (e.g., data structures, algorithms, etc.)	4pts
Source code Adhere good modularization, coding style, and an appropriate amount of comments. (1) FileClient.java: 5pts i. Command prompt: file name, read/write, error checking (1pt) ii. download() function (1pt) iii. upload() function (1pt) iv. invalidate() function (1pt) v. writeback() function (1pt) (2) FileServer.java: 5pts i. File cache (1pt) ii. download() implementation (2pts) iii. upload() implementation (2pts) (3) Coding: 4pts i. Code completeness (2pt) ii. Coding style and readability (2pt)	14pts
Execution output Take a snapshot for each step of the test scenario. (1) Compilation: steps 1-4 (2pt) (2) File read test: step 5 (1pt) (3) File write test: steps 6 and 7 (2pt) (4) File replacement test: steps 8, 9, and 10 (3pt) (5) File writeback test: steps 11, 13, and 15 (3pt) (6) Session semantics read test: steps 16 and 17 (2pt) (7) Multiple write test: steps 18, 20, and 21 (3pt)	16pts
Discussion Write about possible functional and performance improvements of your program. (1) Functional improvements (3pts) (2) Performance improvements (3pts)	6pts
Total	40pts