CSCI.4430 Programming Languages Fall 2024
Programming Assignment #2

Due Date: 7:00PM October 28th (Monday)

Fault Tolerant File System using a Ring of Servers

*This assignment is to be done either **individually** or **in pairs**. **Do not show your code to any other group** and do not look at any other group's code. Do not put your code in a public directory or otherwise make it public. However, you may get help from the mentors, TAs, or the instructor. You are encouraged to use the Submitty Discussion Forum to post questions so that other students can also answer/see the answers.*

In this homework, you are to implement a fault tolerant file storage system. The system consists of a directory service and multiple file servers connected in a ring. The Directory Service keeps track of which files are located in which File Servers. A client can interact with the Directory Service to upload files to or download files from the File Servers.

To make the system fault tolerant, when data is uploaded to the system it will be replicated across R File Servers. The Directory Service determines which initial File Server to upload the data to via hashing the filename and provides the File Server with the file to upload, along with a replication parameter r. The File Servers will save the file locally and then forward the file to the next File Server in the ring, this will continue until r File Servers have a copy of the uploaded file. The Client can also download a file from the system by contacting the Directory Service and requesting a file. The Directory Service will determine which File Server initially stored the file by hashing the name and will forward the request to that File Server so that it can respond to the Client directly.

Your implementation is to use the actor model for programming concurrent file access in either SALSA or Erlang. The three main behaviors of your program are the Client, the Directory Service, and the File Server. The Client is responsible for both initializing the program, and reading / running the commands of the input script. This client will act as a surrogate for web clients seeking to store and retrieve data from the distributed file system.

Agents

Below we outline the functionality expected for each agent in this assignment.

Client

The client is to provide different functionality, given as the following commands.

DirService Creates a Directory Service actor. This will always be called first in any test case. Creating the Directory Service must block the processing of further commands until it has finished creating the N File Server actors and connecting them into a ring.
Command: **d <N> <R>**

Create Sends a file from an input folder to the Directory Service and the Directory Service will first hash the provided filename into an integer value between 0 and N-1, where N is the number of File Servers. The Directory Service will then send the file to the selected File Server. The File Server that receives this file will then forward the file to the next File Server it is connected to. This will continue until the data is replicated on R File Servers.
Command: **c <FileName.txt>**

Get Requests file info from the Directory Service. The Directory Service will forward the request to a File Server, which will directly contact the client with the file contents.
Command: **g <FileName.txt>**

Inactive Requests that a File Server stop responding to Create and Get requests, simulating a local file system failure. The File Server will still forward requests to the next File Server in the ring. The File Server name will be in

the form: fs0 - fs(N-1).
Command: **i <File Server name>**

Quit In Erlang, sends a quit command to the Directory Service which in turn sends quit commands to all of the File Servers it knows about. In SALSA, the client removes the reference to the Directory Server, which then gets garbage-collected along with corresponding File Servers.
Command: **q**

Directory Service

The Directory Service is responsible for handling the following tasks:

1) When the Directory Service is created, it will be provided two arguments, N and $\boldsymbol{R}$. The Directory Service will be responsible for creating N File Server actors and providing them the replication parameter $\boldsymbol{R}$. The File Servers should be named fs0 - fs(N-1).

2) The Directory Service is responsible for connecting the File Servers into a ring topology. A list of the File Servers, sorted by name, should be created and each File Server in the sorted list should be connected, by reference, to the next File Server in the list. The last File Server should also be connected to the first.

3) When the Client tells the Directory Service to write a file, the Client gives the file to the Directory Service which will hash the file name and use the resulting key to assign a File Server to store the file. The File Server will also be provided with a value r which indicates how many File Servers need to store the file. The hash function will be provided to you in the starter code.

4) When the Client wants to download a file, it asks the Directory Service for the file. The Directory Service will forward the client request to the File Server that stores the file. The Client will then be directly contacted by the File Server with the file contents.

5) The Client may send an inactive command to the Directory Service along with the name of one of the File Servers. The Directory Service will forward the inactive request to the target File Server.

6) When a client send a quit command to the Directory Service, it ends execution, including ending the execution of all its known File Servers. The files already stored by each File Server stay on the system.

File Server

The File Server is responsible for handling the following tasks:

1) It will be assigned a next File Server from the Directory Service. These are the File Servers that we will forward requests to during Create and Get requests.

2) The Directory Service will send a Create request with a string filename, string file content, and an integer replication parameter r. When a File Server receives this request it will either store or not store the given file (if the File Server is active it will store it, if inactive it will not) then the File Server will check the replication parameter, if it is greater than 1 the File Server will forward the request to the next File Server. The replication parameter is decremented and sent with the forwarded request.

3) When a client requests a file, the File Server will either: Return the file if the File Server is active and is storing the file OR it will forward the request to the next File Server in the ring. The request will only be forwarded this way up to R times, where R is the replication parameter provided to the Directory Service when initializing. If a File Server receives a Get request where the $r = 0$, it will return an error file with the same name as the requested file, extended with ".err", indicating that the file was not found on the file system. For example: if foo.txt was requested by the client, and is not found by any of the R file servers who receive the request, the last File Server will send the client foo.txt.err, a file containing no text that indicates a failure to find foo.txt in the file system.

4) When receiving the Inactive command from the Directory Service, the File Server will become inactive. When in this state, the File Server will not store any new files nor will it return any files when requested. The only action an inactive File Server can take is to forward all Create and Get requests to the next File Server in the ring. The only exception to this is when the File Server receives a Get request with $r = 0$. In this case, it will send an error file to the client, as described in the previous list item.

5) When receiving a quit command from the Directory Service (Erlang) or when no longer referenced by the Directory Service (SALSA), it ends execution.

Data Replication

An important aspect to this assignment is having data be safe and reliable. To that end, when a file is created on our system, multiple File Servers will each maintain their own copy of the file. When the Directory Service is initialized it will be provided an argument R, which indicates how many redundant copies of stored files will exist in our system.

The File Servers in this assignment will be connected to each other in a ring topology, where File Server 0 maintains a reference to File Server 1. File Server 1 maintains a reference to File Server 2, etc. When a create request is received by a File Server, it will include an argument ***r*** that indicates how many more copies of the data need to be made in our system. If this value is greater than 0 the File server will forward the create request to the next server in the ring. If a File Server that receives a Create request is inactive it will not save the file and forward the request to the next file server as usual.

When a client requests a file from a File Server, it will be possible for the File Server to be 'inactive'. When inactive the File Server will not return the requested file, even if it is being stored by the File Server. Instead, if the replication parameter R is greater than 1, the File Server will forward the request to the next File Server in the ring. If the next file server is also inactive this will continue until we have forwarded the message R times. If no active File Server containing the file was reached by that point then an error file will be sent to the client. The error file has the same name as the requested file, except the file will be a .err instead of a .txt. The error file will be empty.

Actor Interaction Diagrams

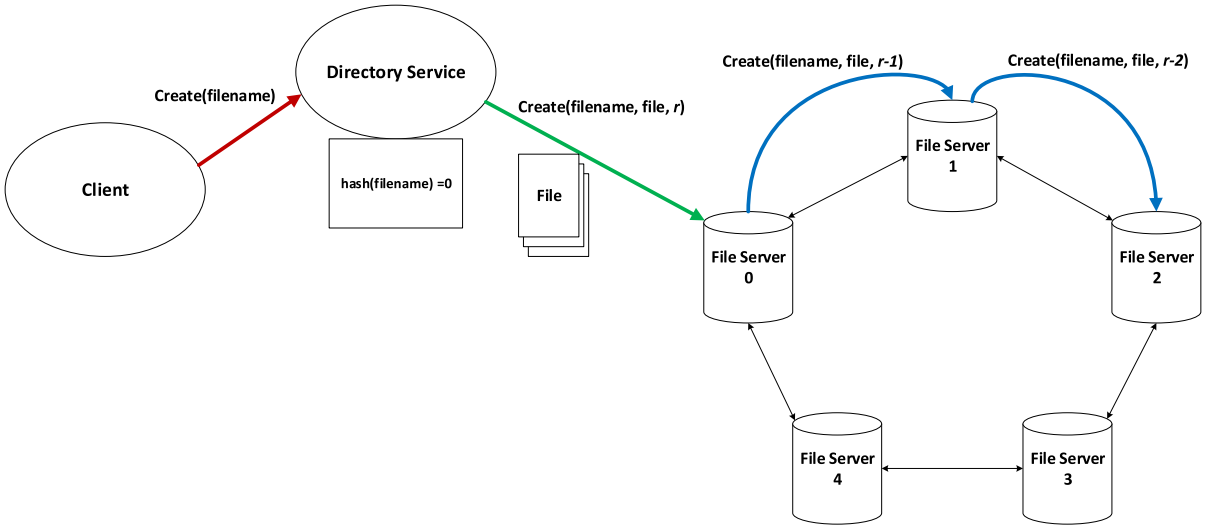

Figure 1: **Create Message Flow.** The client first contacts the directory service with a Create request and a file. The Directory Service runs the filename through a hash function that returns 0-(N-1) where N is the number of File Servers known by the Directory Service, and sends that File Server a Create(File, r) request where the r value indicates how many times the request should be forward to the next File Server in the chain. If any of the File Servers receiving the Create request is inactive, it will NOT store the given file, but will still forward the request to the next File Server and decrement the r parameter.

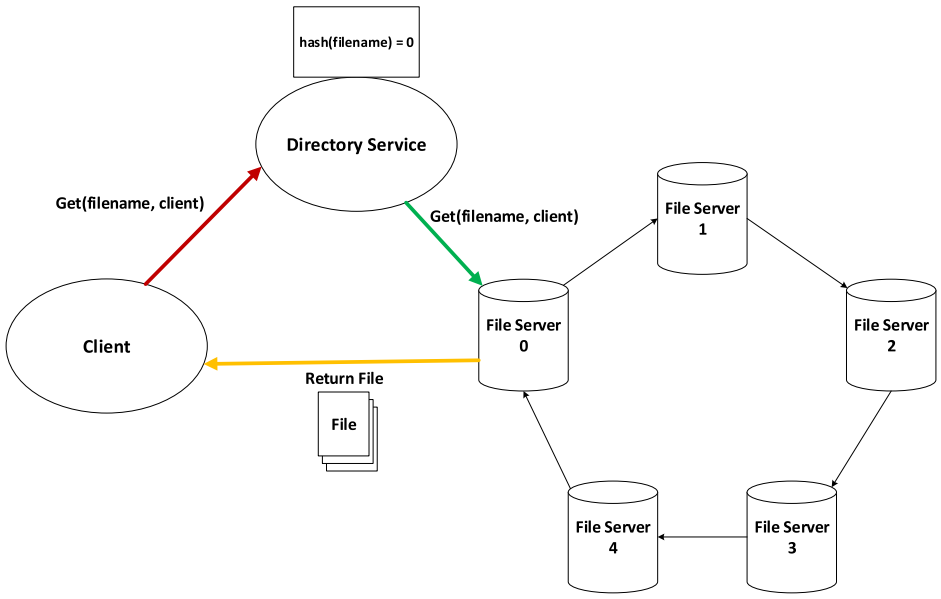

Figure 2: **Get Message Flow.** The Client sends a Get request for a given filename from the Directory Service. The Directory service hashes the filename and sends the corresponding File Server a Get request for the file. If the File Server is active and is storing the file, it will return the file to the client.

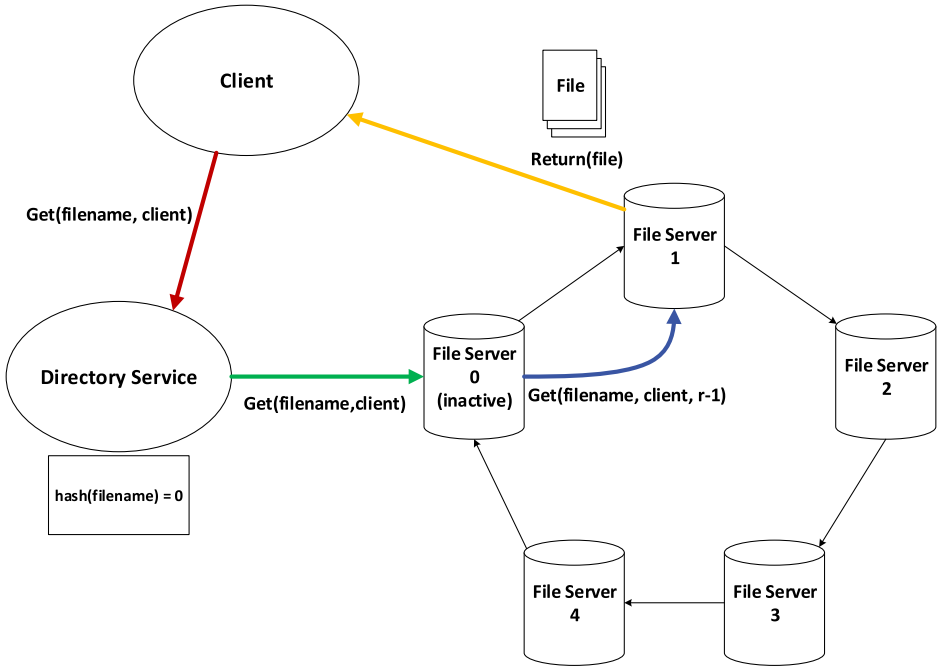

Figure 3: **Get Message Flow, inactive file server.** The Client sends a Get request for a given filename from the Directory Service. The Directory service hashes the filename and sends the corresponding File Server a Get request for the file. In this example the target File Server is inactive. Instead of returning the requested file, File Server 1 forwards the Get request to the next File Server in the chain (File Server 2). Since File Server 2 is active and contains the file it returns the file back to the client. NOTE: you will need to send a reference to the client actor with the Get message so that you can send the requested file back to the client in the case that the first contacted file server is inactive.

Directory / File Structure

```
pa2
├── downloads
│   ├── 1.txt
│   └── 4.txt
├── logs
│   ├── namingServer.log
│   ├── theater1.log
│   └── theater2.log
├── input
│   ├── 1.txt
│   └── 4.txt
└── servers
    ├── fs0
    │   ├── 1.txt
    │   └── 4.txt
    ├── fs1
    │   └── 1.txt
    ├── fs2
    │   └── 1.txt
    ├── fs3
    │   └── 4.txt
    └── fs4
        └── 4.txt
```

input contains all of the files that the client can upload to the Directory Service. Each file server will have its own folder within **servers**. Server files start getting populated when the client calls a *Create* command. **downloads** contains files obtained when the client calls the *Get* command. Notice the naming convention for the files. This naming convention **must** be used in your code in order for us to grade your assignment. The files in **downloads** must have the same names as the corresponding files in **input**. The **logs** directory will contain the log files for each theater that is started by the run.sh script provided. At the start of program execution, only the **input** directory will have anything in it, while the **downloads** and **servers** directories will be empty.

Files will be UNIX style (no carriage returns), though this should not affect your implementation.

Example Program Input

The following is an example of the output expected by your program. In this example, foo.txt and bar.txt will be created on your file servers. When hashing foo.txt, File Server 2 is selected to store the file initially and bar.txt is hashed to return File Server 3. Note a .err file is returned when all file servers that could store the file are inactive.

Files in the input directory:

```
input/foo.txt
input/bar.txt
```

Contents of the input script:

```
d 4 2
c foo.txt
c bar.txt
g bar.txt
i fs2
i fs3
g foo.txt
q
```

Program Output:

```
servers/fs0/bar.txt
servers/fs2/foo.txt
servers/fs3/foo.txt
```

```
servers/fs3/bar.txt
downloads/foo.txt.err
downloads/bar.txt
```

SALSA Instructions

Begin by downloading the Salsa starter code here. Your solution will be implemented in the Client.salsa, DirectoryService.salsa, and FileServer.salsa files. You have been provided a utility script that handles reading the input script and functions to read and write files. Additionally, you have been provided a hashing function in the DirectoryService.salsa file.

For convenience, the starter code includes the file run_salsa.sh, this file is provided to more easily compile and execute your code. The only argument to the script is a path that you should point to your pa2 directory (which contains an input folder). This input folder should contain both an input.script and a theaters.txt file.

```
/** Run Code */
bash run_salsa.sh path/to/pa2/directory
```

If you would like to manually compile and run your solution you can compile your code with:

```
/** Compile the salsa and java code in local directory, assumes the file "salsa1.1.6.jar" is in local
    directory */
$ salsac DirectoryServer.salsa
$ salsac FileServer.salsa
$ salsac Client.salsa

/** You can start your program with: */
$ salsa Client path/to/input.script path/to/theaters.txt
```

For the above snippet, salsac and salsa are UNIX aliases or Windows batch scripts that run java and javac with the expected arguments: See .cshrc for UNIX, and salsac.bat / salsa.bat for Windows.

Make sure to run all these scripts in the root directory of the assignment, i.e. pa2.

To run your distributed implementation, you will first need to run the name server and theaters:

```
/** Run the nameserver */
[host0:dir0]$ wwcns [port number 1]

/** Start theaters */
[host1:dir1]$ wwctheater [port number 2]
[host2:dir2]$ wwctheater [port number 3]
...

/** Run Code */
```

Where wwcns and wwctheater are UNIX aliases or Windows batch scripts. See .cshrc for UNIX and wwcns.bat and wwctheater.bat for Windows. Make sure that the theaters are run where the actor behavior code is available, that is, the pa2 directory should be visible in directories host1:dir1 and host2:dir2. Then, run the distributed program as you would for the concurrent implementation. During testing, it will be useful for you to have multiple terminal windows open, with one VM per window.

The program will know if it is running the concurrent versus the distributed version of the code based on the provided theater file. The theaters.txt file will be organized as a set of UALs, one per line of the file. When creating your Directory Service and File Server actors will each take one of the UALs as their own. The first line of this file, if it is not empty, will be the name server for SALSA, and therefore be used as the UAN.

To run the concurrent version of the program the theaters.txt file should be empty. To run distributed version of the program, the file should be populated with a name server and UALs for each actor. When running the code on Submitty, we will be providing the theaters.txt to be used during the tests.

SALSA Tips

- For reference, please see the SALSA webpage, including its FAQ. Read the tutorial and a comprehensive example illustrating distributed computing in SALSA.

- The module/behavior names in SALSA must match the directory/file hierarchical structure in the file system. e.g., the **DirectoryServer** behavior must be in a relative path **src/DirectoryServer.salsa** and should start with the line **module src;**.
- Messaging is asynchronous. **m1(...); m2(...);** does not imply **m1** occurs before **m2**.
- Notice that in the code **m(...)@n(...);**, **n** is sent after m is executed, but not necessarily after messages sent *inside* m are executed. For example, if inside **m**, messages **m1** and **m2** are sent, in general, **n** could happen before **m1** and **m2**.
- (Named) tokens **can only** be used as arguments to **messages** - they cannot be used in arbitrary expressions since that would cause the actor to block.
- Join statements can be used to group multiple return tokens (say, in a loop) into a single token array sorted by message order.
- **@currentContinuation** can be thought of as **return token;**.
- You are free to use any imports or custom Java classes as is convenient. SALSA does not support **generic** types, so you will have to use Vectors and type casting or custom class types compiled separately.

Erlang Instructions

First download your Erlang starter code. You will implement your solution in main.erl. Five functions have been provided that you need to fill in. Note that these function signatures can not be altered as this would lead to problems with the run.sh scripts. We have also provided a util.erl script which has several functions which will help you in your implementation. Note that we have also provided a hashFileName function which is essential as it is the hash function we will be using in our testing.

Run concurrent implementation

To run your Erlang concurrent program, execute the following:

```
bash run_concurrent.sh tests/test{TEST_NUMBER}.txt
```

Note that you can also manually test your functions by simply starting erl and executing the functions in your shell:

```
/** Launch the Erlang shell */
erl

/** In the erlang shell, compile all files, run the functions you would like */
c(main), c(util).
main:start_dir_service(1,1).
main:create('a.txt').
main:deactivate('fs0').
main:get('a.txt').
main:quit().
```

Run distributed implementation

To run your Erlang distributed program, execute the following:

```
bash run_distributed.sh tests/test{TEST_NUMBER}.txt
```

If you would like to manually test your functions you can do so:

```
/** Compile your main and util files **/
erlc main.erl util.erl

/** Start your file server nodes then run the directory service node with it's parameters */
erl -noshell -detached -sname fs0@localhost -setcookie foo
erl -noshell -detached -sname ds@localhost -setcookie foo -eval "main:start_dir_service(1, 1)"
```

```
/** Start your client, this will launch you into the erlang shell */
erl -sname client@localhost -setcookie foo

/** In the erlang shell, compile execute the functions you would like */
main:create('a.txt').
main:deactivate('fs0').
main:get('a.txt').
main:quit().
```

Erlang Tips

- For reference, please see the Erlang documentation on concurrency.
- We highly recommend going through the Erlang tutorial before beginning the assignment. The tutorial can be found here. Start with the sequential programming section to get a feel for the syntax and then move on to concurrent programming for some practice on what you will be doing for this homework.
- When you begin your implementation recommend starting with a concurrent implementation before moving on to handling adding distributed functionality.
- Once you are ready to add distributed functionality we have provided the function isDistributed() in the util.erl file which will return true if the script is being run on a node and false if there is no node being run.
- You would likely also like to cleanup your directory between tests and any nodes which could not be terminated. For this purpose we have provided a cleanup.sh script which will handle this for you.

Due date and submission guidelines

Due Date: Monday, 10/28th, 7:00pm

Grading: This assignment will be graded on the correctness of the files in the downloads directory and servers directory, based on the given test case. As always, code clarity / readability is important, and will be a factor in your final grade.

Submission Requirements: Please submit your code files (main.erl for Erlang and the three files Client.salsa, DirectoryService.salsa, FileServer.salsa for SALSA) and a README file to Submitty. README files must be in plain text; markdown is acceptable. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific features / bugs in your solution. You are limited to 20 submissions to Submitty, so wherever possible test your solution locally.

Do not include unnecessary files.