Programming Assignment #2
(due Thursday, October 28th, **7:00PM**)

Distributed File Sharing

*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 LMS Discussion Forum to post questions so that other students can also answer/see the answers.*

In this homework you are to implement a peer-to-peer system inspired by BitTorrent. The system consists of a *directory service* and multiple *file servers*. The Directory Service keeps track of the files located in the File Servers. A *client* can interact with the Directory Service to upload files—which the Directory Service decomposes into parts and distributes among the File Servers,—or to request information on a file so the Client may download the file parts directly from the File Servers to reconstruct the file locally.

Your implementation is to use the actor model for programming concurrent file access in either SALSA or Erlang. The Directory Service, each of the File Servers, and the Client are actors.

The Directory Service
The Directory Service is responsible for handling the following tasks:

1) When a new File Server is created, it contacts the Directory Service. The Directory Service updates its list of known File Servers and keeps it sorted alphabetically. In practice, sorting would not be necessary, but we require it in this assignment for deterministic file part storage.

2) When a client tells the Directory Server to create a file (for this assignment, we are using text files), the Client gives the file to the Directory Server, which splits the file into chunks of 64 characters each. Each of the chunks is sent to a File Server (that the Directory Service knows about) in a round-robin fashion following alphabetical order until all chunks have been assigned. The Directory Server updates its file meta-data to include where each chunk is located.

3) When a client wants to download a file, it asks the Directory Service for information on that file. The Directory Service provides the client with information on where each part of the file it wants is located (which File Server has each file part).

4) When a client sends a quit command to the Directory Service, it ends execution, including ending the execution of all its known File Servers. File parts remain in the File Servers' file system.

The File Server
The File Server is responsible for handling the following tasks:

1) When being assigned a *file part* from the Directory Service, it stores the file part within its own file system (see required directory/file structure below).

2) When a client requests a file, it returns the contents of that file along with what part (e.g. Part 4) of the file it is.

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

The Client
The client is to provide different functionality, given as the following commands. Notice that in Erlang, `<DirService>` is a UAL; in SALSA, it is a UAN.

DirService) Creates a Directory Service actor (with a unique UAN name, if in SALSA) at some (UAL) location. This will always be called first in any test case. Sufficient time will be given (e.g. 1 second) for the Directory Service to start so that no File Servers try to send messages to it before it is online.

```
d [UAN] <UAL>
```

FileServer) Creates a File Server actor (with a unique `fsUAN` name, if in SALSA) at some (`fsUAL`) location. The reference to the Directory Service this server will use is passed as the first argument.

```
f <DirService> [fsUAN] <fsUAL>
```

Create) Sends a file from an `input` folder to the Directory Service and the Directory Service will split that file into chunks and distribute it to the File Servers it knows about.

```
c <DirService> <FileName.txt>
```

Get) Requests file info from the Directory Service. Once it has that, it makes **concurrent** requests for each file part to the respective File Servers. Upon receiving responses from the File Servers, it reconstructs the file and saves it to a `downloads` folder (see required directory/file structure below).

```
g <DirService> <FileName.txt>
```

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.

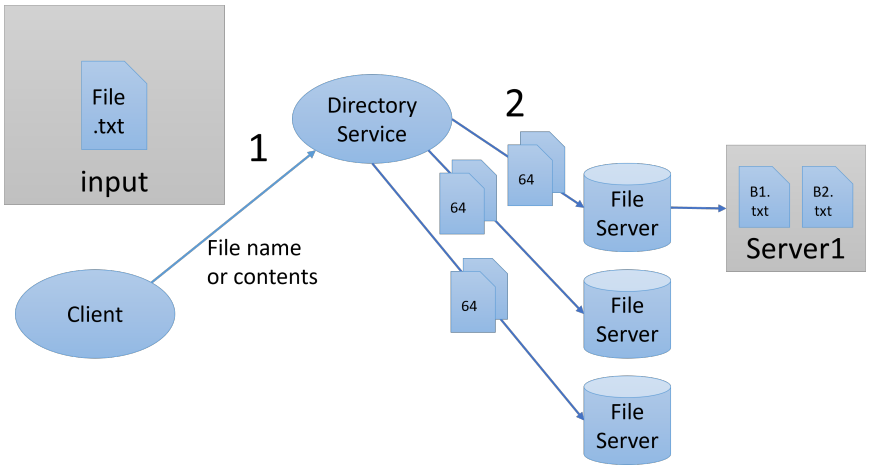
```
q <DirService>
```

Directory/File Structure

```
pa2
├── downloads
│   ├── 1.txt
│   └── 3.txt
├── input
│   ├── 1.txt
│   ├── 2.txt
│   ├── 3.txt
│   ├── 4.txt
│   ├── 1.script
│   ├── 2.script
│   └── 3.script
└── servers
    ├── fs1
    │   ├── 1_1.txt
    │   ├── 1_4.txt
    │   ├── 2_1.txt
    │   └── 3_1.txt
    ├── fs2
    │   ├── 1_2.txt
    │   ├── 1_5.txt
    │   ├── 2_2.txt
    │   └── 3_2.txt
    └── fs3
        ├── 1_3.txt
        └── 2_3.txt
```

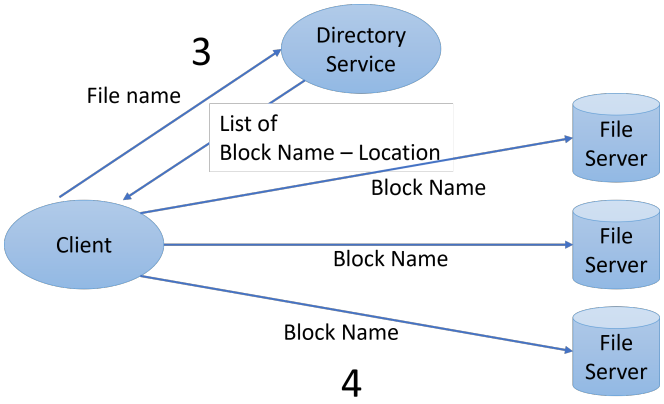
input contains all of the files that the client can upload to the Directory Service. Each file server will have its own folder within **servers**. Server files start getting populated when the client calls a *Create* command. **downloads** contains files obtained when the client calls the *Get* command. 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** have names the same as in **input**. The file parts in **servers must** have **_num** appended to the name where the number indicates the part of the file (**starting at 1**). Each file part will be exactly 64 characters (except for the last one which may be shorter). At the start of program execution, only **input** will have anything in it while **downloads** and **servers** will be empty.

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



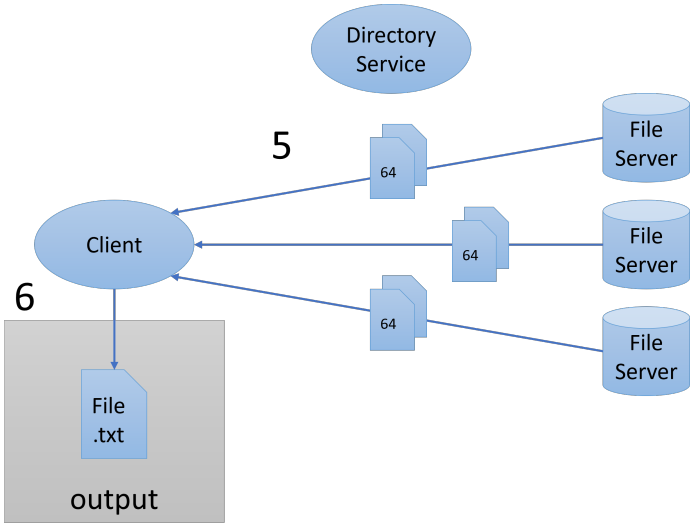
`Create File`:

1. File name and contents are sent to the directory service.
   The file must be located in a folder named `input`.
2. Directory server splits file into 64 character blocks and distributes it to the file servers.
   The servers save their blocks to directories named after themselves.



`Get File`:

3. The client requests a particular file from the directory service.
   The directory service replies with the block names and their locations.
4. The client requests a particular block from each file server concurrently.



`Get File` (cont.):

5. The file servers respond with the requested blocks.
6. The client reassembles them into the original file.
   The file must be written to a folder named `downloads`.

Erlang-specific Instructions

To run your program as a distributed system, first start an appropriate number of Erlang VMs:

```
$ erl -sname ds@localhost -setcookie foo
$ erl -sname fs1@localhost -setcookie foo
$ erl -sname fs2@localhost -setcookie foo
```

During testing, it will be useful for you to have multiple terminal windows with one VM per window. After running an Erlang VM, you can compile code by doing `c(main).` assuming you have a file named `main.erl`. You can then, in ds@localhost, start the directory service by calling `main:start_dir_service().` and start the file servers in their respective terminal windows (e.g. fs1@localhost), by calling `main:start_file_server(ds@localhost).`
After this, you can create another Erlang VM for the client (e.g. client@localhost), following the above format. From this VM you can call the client functions, `main:create(ds@localhost,"1.txt").`, `main:get(ds@localhost,"1.txt").`, `main:quit(ds@localhost).`

Erlang Tips

We highly recommend going through the Erlang tutorial before beginning the assignment. The tutorial can be found here (documentation can also be found on this site). 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 in this homework.

Example Erlang Program Input
Files:

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

In the input script:

```
d ds@localhost
f ds@localhost fs1@localhost
f ds@localhost fs2@localhost
c ds@localhost 'foo.txt'
f ds@localhost fs3@localhost
c ds@localhost 'bar.txt'
g ds@localhost 'bar.txt'
g ds@localhost 'foo.txt'
q ds@localhost
```

Program Output
Files:

```
servers/fs1/foo_1.txt
servers/fs2/foo_2.txt
servers/fs1/foo_3.txt ... etc
(after server 3 is added)
servers/fs1/bar_1.txt
servers/fs2/bar_2.txt
servers/fs3/bar_3.txt
servers/fs1/bar_4.txt ... etc
(after they are requested)
downloads/foo.txt
downloads/bar.txt
```

SALSA-specific Instructions

You can compile your code with:

```
$ salsac src/DirectoryServer.salsa
$ salsac src/FileServer.salsa
$ salsac src/Client.salsa
```

To run your program as a distributed system, you must first start a name server and an appropriate number of theaters (actor VMs):

```
$ wwcns
$ wwctheater -p 4040
$ wwctheater -p 4041
$ wwctheater -p 4042
```

During testing, it will be useful for you to have multiple terminal windows with one VM per window.

Then, you can start the directory service by:

```
$ salsa src/Client
```

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

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.

Example SALSA Program Input
Files:

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

In the input script:

```
d uan://localhost:3030/ds rmsp://localhost:4040
f uan://localhost:3030/ds uan://localhost:3030/fs1 rmsp://localhost:4041
f uan://localhost:3030/ds uan://localhost:3030/fs2 rmsp://localhost:4042
c uan://localhost:3030/ds foo.txt
f uan://localhost:3030/ds uan://localhost:3030/fs3 rmsp://localhost:4043
c uan://localhost:3030/ds bar.txt
g uan://localhost:3030/ds bar.txt
g uan://localhost:3030/ds foo.txt
q uan://localhost:3030/ds
```

Program Output
Files:

```
servers/fs1/foo_1.txt
servers/fs2/foo_2.txt
servers/fs1/foo_3.txt ... etc
(after server 3 is added)
servers/fs1/bar_1.txt
servers/fs2/bar_2.txt
servers/fs3/bar_3.txt
servers/fs1/bar_4.txt ... etc
(after they are requested)
downloads/foo.txt
downloads/bar.txt
```

Provided Code
Starter code for SALSA is provided here and starter code for Erlang is provided here. The provided code mostly does file reading and writing for you, since that is not the focus of the assignment. Also included in the provided files is a bash script for running the program and checking the output in a single terminal along with several test scripts. To run it, type `.\run.sh "script.txt"` substituting the argument with the script you wish to run. This is what we will be doing to grade the assignments (with additional scripts) but for testing it may be more useful to set up each VM manually.

Notes for Other Programming Languages
You may use another actor programming language, such as Scala, to complete this assignment, but you should only use that language's actor programming aspects (e.g., Akka). Be certain you have a strong understanding of the language and do not use other concurrency mechanisms (e.g. threads or locks.) **If you intend to use another actor programming language, please get explicit approval from Professor Varela first.**

Fault Tolerance
Your program does not have to be particularly error tolerant. We will not give inputs formatted differently than those seen above. If a file is requested, it will have been previously provided and can be assumed to exist (though SALSA/Java requires exception handling). After a File Server or Directory Service is started, it will not prematurely terminate.

Grading
The assignment will be graded mostly on correctness, but also code clarity / readability. If something is unclear, explain it with comments!

We will be running your code with **run.sh**, start to finish, to verify that files are correctly reconstructed. We will also check that the files saved by the file servers contain the expected output. Lastly, we will be reading through your code for clarity. If your code doesn't immediately compile or pass our tests, it will likely result in a low score.

Submission Requirements
You may work alone, or with one other student. You are to submit your code via LMS. Only submit one assignment per pair. Please submit a ZIP file with your code, including a README file. In the README file, place the names of each group member (up to two). It should also have a list of specific features/known bugs in your solution. Your ZIP file must be named with your LMS user name(s) as the filename and contain the language you chose in the following format:

Ex.: `userid1_erlang.zip, userid1_userid2_salsa.zip, userid1_salsa.zip`

For Erlang submissions, include `main.erl` and `util.erl` along with any additional files you may have created. For SALSA submissions, include a `src` directory with files `Client.salsa`, `DirectoryService.salsa`, and `FileServer.salsa`. You may also have additional .salsa and .java files.