COMP 6411: PROJECT DESCRIPTION

Important:

1] Projects must be submitted on time in order to receive full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied.

2] It is the student's responsibility to verify that the project has been properly submitted. You can NOT send an alternate version of the project at a later date, with the claim that you must have submitted the wrong version at the deadline. Only the original submission will be graded (though you can certainly re-submit multiple times before the due date)

3] These are individual projects. Feel free to talk to one another and even help others understand the basic ideas. But the source code that you write must be your own.

*4]*** This project will be written in Erlang and Java. As it the case with other languages, there are 3rd party Erlang (and Java) libraries that can be downloaded from the web. YOU CAN NOT USE ANY OF THOSE. The purpose of the assignment is to help you become familiar with the basic form of the language(Erlang in particular) – it is not for you to simply download libraries that someone else has written. So 100% of the code you write must work with the standard Erlang and Java distributions (both already include a huge number of supporting functions). The graders will NOT download any external Erlang or Java modules/packages in order to run your programs – if it doesn't run with the standard distribution, it will not be graded.*

DESCRIPTION: For the final project, we will take a look at Erlang. Note that while Erlang is a functional language, in the same general class as Clojure, our focus here is the *concurrency* model provided by Erlang. In particular, this assignment will require you to gain some familiarity with the concept of message passing. In fact, Erlang does this more effectively than any other modern programming language.

That said, there is a "twist" with the project. The course is called "Comparative Programming Languages" for a reason. The purpose is not just to learn something about other languages but to get a better sense of how problems can be solved differently, depending on the language used. Often, a well-chosen language can make the job much easier and much more intuitive.

So, in addition to implementing the application in Erlang, you will implement the same application using Java, arguably the most popular imperative language in use today. While Java will be far more comfortable for many of you, it is a general-purpose language that was not designed specifically for concurrency (though it has always provided support for this).

In short, the project emphasizes the "comparative" element in the course's title. Note, however, that this does not mean that the project is massive in size. The application itself is not large, so

neither the Erlang program nor the Java program will require a huge amount of source code. Instead, you will have to look at the problem differently in the two cases.

DETAILS: In the description below, we will describe the project in terms of Erlang. A short section on the Java version will be given at the end.

So your task is to provide an extremely simple communication network for a group of friends. It is so simple that all the friends will actually do is send a contact message to one or more people in the group, and then wait for a confirmation reply from that person. That's it.

You will, of course, need a list of a group of friends and the contact messages that will be sent by each person. This information will be read from a file called "calls.txt" that will be located in the same folder as the application code. While Erlang provides many file primitives for processing disk files, the process is not quite as simple as Clojure's slurp() function. So the "calls.txt" file will contain call records that are already pre-formatted. In other words, they are ready to be read directly into Erlang data structures. An example of a "calls.txt" file is:

```
{john, [jill,joe,bob]}.
{jill, [bob,joe,bob]}.
{sue,  [jill,jill,jill,bob,jill]}.
{bob,  [john]}.
{joe,  [sue]}.
```

Here, we have five "calling" tuples. The first field in each tuple contains the name of the person who will make the calls. The second field is a list of friends that this person will contact. So, for example, the first tuple indicates that john will contact jill, joe, and bob.

To read this file, all you simply have to use is the `consult()` function in the `file` module. This will load the contents into an Erlang list of 5 tuples (in this particular case). Note that NO error checking is required. The "calls.txt" file is guaranteed to exist and contain valid data. Each person will make at least one contact with another person, and all people in the contact list are guaranteed to exist and make at least one call.

So your job now is to take this information and make contact with the other people in each list. Once a contact request is received, each person must reply to the original person to indicate that they have received the contact request.

Of course, we need a way to demonstrate that all of this has worked properly. To begin, it is important to understand that this is a multi-process Erlang program. The "master" process will be the initial process that spawns one process for each of the people in the "calls.txt" file. So, in our little example above, there will be 6 processes in total: the master and 5 friends.

To confirm the validity of the program, each person receiving a contact request – either an initial request or a response – must send a message to the master process to inform it about the exchange.

IMPORTANT: The "master" process is the only process that should display anything to the screen. No "person process" should directly display anything, other than termination messages.

Below, we see sample output for our "calls.txt" example:

```
** Calls to be made **
john: [jill,joe,bob]
jill: [bob,joe,bob]
sue: [jill,jill,jill,bob,jill]
bob: [john]
joe: [sue]

bob received intro message from jill [738000]
joe received intro message from john [741004]
bob received intro message from john [770008]
joe received intro message from jill [779007]
john received intro message from bob [736102]
john received reply message from joe [741004]
john received reply message from bob [770008]
jill received intro message from sue [737001]
bob received intro message from jill [816004]
bob received reply message from john [736102]
bob received intro message from sue [897005]
jill received intro message from john [739000]
jill received reply message from bob [738000]
john received reply message from jill [739000]
jill received intro message from sue [819004]
jill received reply message from joe [779007]
jill received intro message from sue [880460]
jill received reply message from bob [816004]
sue received intro message from joe [828004]
sue received reply message from jill [737001]
sue received reply message from bob [897005]
sue received reply message from jill [819004]
sue received reply message from jill [880460]
joe received reply message from sue [828004]
jill received intro message from sue [987009]
sue received reply message from jill [987009]

Process joe has received no calls for 5 seconds, ending...

Process john has received no calls for 5 seconds, ending...

Process bob has received no calls for 5 seconds, ending...

Process sue has received no calls for 5 seconds, ending...

Process jill has received no calls for 5 second, ending...

Master has received no replies for 10 seconds, ending...
```

Let's look at the output. When the program is run, the master process will first display a summary of the calls that will be made. Next, the master will start a process for each of the people in the calls.txt file.

Each time a contact message is received, information about the message will be passed to the master process, which will display the info. Again, the person process can NOT display this themselves (you will receive no point value for this criteria if you do this). The information message will include information about the sender and receiver, the type of message (initial contact or a reply), and it will include a timestamp for the initial contact message.

The timestamp will be created from the 3rd component of the erlang:now() function. now() returns a 3-element tuple of the {MegaSeconds, Seconds, MicroSeconds} since Jan 1, 1970. The MicroSeconds value serves as a nice timestamp for each message exchange. Note that newer versions of the Erlang compiler will typically say that the now( ) function is deprecated. That's fine. It still works and represents a simple mechanism to create a useful timestamp.

Let's see how this works. In our first call record, john contacts jill, joe, and bob. If you look at the 12th info message displayed by the master process, you will see:

```
jill received intro message from john [739000]
```

Note the timestamp 739000. If we look a little further, to the 14th info message, we see:

```
john received reply message from jill [739000]
```

Here, we can see the matching timestamp, indicating that this is the second half of the exchange. If you look at the call lists, you will see that there are 13 contact requests, and exactly 13 pairs of info messages in the output.

The final part of the output simply shows that each process shuts down once a period of time with no new messages has been identified (Note: In practice, all messages will be sent and received in less than 5 seconds.) This is the one place where the *person processes* will display to the screen.

One final thing: In theory, if you run this program multiple times, the message order would be slightly different. But with such simple functionality in each process, that might not happen. So, just before each process sends a message (intro or reply), it should sleep for a random amount of time, between 1 and 100 milliseconds. Erlang has a simple sleep() function, and random() and seed() functions that will take care of this in a couple of lines of code. Once this is done, you will see that multiple invocations of the program will produce slightly different results each time.

So that's it. As noted, the program isn't particularly long – you'll see this when you have completed the application. The difficult part is thinking about the logic in a new way.

Important: You are free to implement your Erlang code however you like (i.e., using any Erlang data structures). But again, you can only use Erlang modules contained in the standard Erlang distribution, NO external third party libraries.

JAVA VERSION: The comparative Java program will produce exactly the same result. You will begin by reading the same data files. In this case, you will use Java's IO classes to extract the calling data.

Once you have the data, you will replicate the messaging program. In this case, you will use Java's basic *thread* mechanism to create individual threads to represent each friend. So just like the Erlang app, this will be a multi-threaded program. Each friend will be constructed as a thread, and the friend threads will exchange messages, using the same logic/order as described in the Erlang description above. For the timestamp, you can use something like *System.currentTimeMillis( ) …* or a reasonable equivalent.

Important: Again, you are free to implement your Java code however you like. However, you can only use Java classes contained in the Java Standard libraries, NO external third party libraries, including any message frameworks that do the communication work for you. The real purpose of the Java component is to see how a general imperative language compares to a language designed for a particular purpose.

GRADING Please note that although two programs will be written, they will not be given equal weight in terms of grading. This is primarily an Erlang project and, as such, the Erlang application will receive 80% of the total grade. The remaining 20% will be associated with the Java program. So, if you want a good grade, you cannot simply do the Java version and quickly throw together a non-functional Erlang application.

DELIVERABLES: Your Erlang submission will have just 2 source files. The "main" file will be called `exchange.erl` and will correspond to the master process. The second file will be called `calling.erl` and will include the code associated with the "people" process.

You should use a function called "start()" in the exchange.erl file. This represents the main or starting function for the program. This will allow the graders to run your Erlang program by simply specifying that "start()" is the first function to be run in the application.

Module names will be identical to the file names. Do not include any data files, as the markers will provide their own.

The Java version can have multiple source files, depending on how you choose to organize your code. However, the main "driver" class must be called `exchange.java`. It will have the *main* method and will be the entry point for running and testing your code.

Once you are ready to submit, place the .erl and .java source files into a zip file. The name of the zip file will consist of "project" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called project_Smith_John_123456.zip". The final zip

file will be submitted through the course web site on Moodle. You simply upload the file using the link on the project web page.

Good Luck