COMP 6411: PROJECT DESCRIPTION

Important Info:

1. *Projects can be done either individually or in a group of 2 or 3 students (extra work is required for group submissions).*

2. *Feel free to talk to other students about the project. That's not a problem. However, when you (or your group) write the code, you must do this yourself. Source code cannot be shared under any circumstance. This is considered to be plagiarism. Do NOT put yourself in this position.*

3. *Projects must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate. . You **MUST** verify the contents of your assignment AFTER you submit. You will be graded on the version submitted at the deadline – no other version will be accepted at a later date.*

4. *The graders will be using a standard distribution of Erlang (the current version in the docker distribution that will be used for grading is OTP 25, but any recent version should be fine). You cannot use any Erlang libraries and/or components that are not found in the standard distribution. The graders will not have these libraries on their systems and, consequently, your programs will not run properly.*

OBJECTIVE: For the final project, we will take a look at Erlang. Note that while Erlang is a functional language, 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.

Briefly, your objective is to model a simple banking environment. Specifically, you will be given a small number of customers, each of whom will contact a set of banks to request a number of loans. Eventually, they will either receive all of the money they require or they will terminate without completely meeting their original objective. The application will display information about the various banking transactions before it finishes. That's it.

DETAILS: So now for the details. To begin, you will need a handful of customers and banks. These will be supplied in a pair of very simple text files - one for customers and one for banks. While Erlang provides many file primitives for processing disk files, the process is not quite as simple as Clojure's `slurp()` function. So your two input files will contain records that are already pre-formatted. In other words, they are ready to be read directly into standard Erlang data structures.

An example customer file might be:

```
{jill,450}.
{joe,157}.
{bob,100}.
```

An example bank file might be:

```
{rbc,800}.
{bmo,700}.
{ing,200}.
```

In other words, each file simply contains a set of Erlang *tuples*. The first element of each tuple is an *atom* (note that atoms start with a lower-case letter). The atoms will represent the names of the customers and banks respectively (so no string processing is required). You will see that each atom/label is associated with a number. For customers, this is the total funds that they are hoping to obtain. For banks, the number represents their total financial resources that can be used for loans.

To read these files, we use the `consult()` function in the `file` module. This will load the contents of either file directly into an Erlang structure (i.e., a list of tuples). Note that NO error checking is required. The text files are guaranteed to contain valid data.

Because you (and the graders) will want to run the program with different input files, we don't want to hard-code the file names. Instead, they will be passed as command line parameters. While this isn't especially difficult to do, I don't want you to waste time trying to get this to work. So after compiling your source files with `erlc`, you will run your program with the following invocation on the command line:

```
erl -noshell -run money start c1.txt b1.txt -s init stop
```

This will run a program in which the main module (i.e., source file) is called `money`. Within `money`, the initial function (e.g., like *main* in Java) will be called `start`. The start function will accept a list of two parameters, in this case *c1.txt* and *b1.txt*. These, of course, are the names of the customer file and bank file. You can call them anything you like (the graders will provide their own test files).

Inside the `money.erl` source file, you will have the following code at the top of the file:

```
-module(money).
-export([start/1]).

start(Args) ->
      CustomerFile = lists:nth(1, Args),
      BankFile = lists:nth(2, Args),

      {ok, CustomerInfo} = file:consult(CustomerFile),
      {ok, BankInfo} = file:consult(BankFile),
      …
```

Note that the … at the end simply means that the `start` function will have more logic after this point. This is just the code to read the data files. In summary, the code above provides the module name for the current source file (`money`), and it indicates that the module exports a single function called `start`. The `start` function will take one parameter which, in this case, is going to be a list of the two command line args: `["c1.txt", "b1.txt"]`. We assign "c1.txt" to `CustomerFile` and "b1.txt" to `BankFile`. We then use the `consult` function in the `file` module to read the content of each file and bind it to labels. So `CustomerInfo` will be a list of customer tuples `[{jill,450}, {joe,157}, {bob,100}]`, while `BankInfo` will be a list of bank tuples `[{rbc,800}, {bmo,700}, {ing,200}]`. At this point, you have all of your input data and you are ready to proceed.

So your job now is to take this information and create an application that models the banking environment. Because customers and banks are distinct entities in this world, they will be modeled as separate tasks/processes. When the application begins, it will therefore generate a new process for *each* customer and *each* bank. Because you do not know how many customers or banks there will be, or even their names, you cannot "hard code" this phase of the application.

The customer and bank tasks will then start up and wait for contact. You may want to make each new customer sleep for 200 milliseconds or so, just to make sure that all the bank tasks have been created and are ready to be used - this can be done trivially with an expression like `timer:sleep(200)`. Otherwise, the application may crash if a customer tries to contact a bank that does not yet exist.

The banking mechanism itself works as follows:

1. Each customer wants to borrow the amount listed in the input file. At any one time, however, they can only request a maximum of 50 dollars. When they make a request, they will therefore choose a random dollar amount between 1 and 50 for their current loan.
2. When they make a request, they will also randomly choose one of the banks as the target.
3. Before each request, a customer will wait/sleep a random period between 10 and 100 milliseconds. This is just to ensure that one customer doesn't take all the money from the banks at once.
4. So the customer will make the request and wait for a response from the bank. It will not make another request until it gets a reply about the current request.

5. The bank can accept or reject the request. It will reject the request if the loan would reduce its current financial resources below 0. Otherwise, it grants the loan and notifies the customer.
6. If the loan is granted, the customer will deduct this amount from its total loan objective and then randomly choose a bank (possibly the same one) and make another request. As noted, the loan requests can never be greater than 50 dollars. However, once the customer's remaining loan amount is less than 50 dollars, then the next loan request cannot be greater than that amount. In other words, the loan request is actually a random number between 1 and [50 or the remaining loan objective, whichever is smaller].
7. If the loan is rejected, however, the customer will remove that bank from its list of potential lenders, and then submit a new request to one of the remaining banks. Just to be completely clear, a customer never sends another (smaller) request to a bank that has denied a previous request.
8. This process continues until customers have either received all of their money or they have no available banks left to contact.

And that's it.

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, in turn, *spawns* processes for each of the customers and each of the banks. So, in our little example above, there will be 7 processes in total: the master, 3 customers, and 3 banks.

To confirm the validity of the program, we need a series of info messages to be printed to the screen. This will work as follows:

- When a customer requests a loan amount, they will send a message to a bank with the relevant info.
- The customer will also send a message to the master process, providing the details of the loan request.
- The master process will then print this info to the screen so that the user (you or the grader) can see the loan requests as they happen.
- When a bank responds to a loan request, it will send the approval/denial to the relevant customer.
- The bank will also send this information to the master process.
- The master process will then display the bank's response to the screen.

In effect, the loan request/response info will create a real-time transaction log as the program is running.

IMPORTANT: The "master" process is the only process that should display anything to the screen. Customer and Bank processes NEVER do any I/O themselves. In production applications this would also be true since (1) concurrent I/O from multiple tasks would get interleaved together, creating an unreadable mess, and (2) large applications would often use networked/distributed nodes that would not even use the same console/screen. The graders will check your source code to ensure that all printing is done from the master.

Of course, the transaction log is very detailed and it can be hard to determine if the banking transactions are being done properly. So once all of the customer/bank transactions are finished, the master process will display a final report that summarizes the transaction results in a simple way.

Let's look at the complete output for a small input example.

```
** The financial market is opening for the day **

Starting transaction log...

? sam requests a loan of 11 dollar(s) from the apple bank
$ The apple bank approves a loan of 11 dollar(s) to sam
? james requests a loan of 32 dollar(s) from the knox bank
$ The knox bank approves a loan of 32 dollar(s) to james
? karen requests a loan of 24 dollar(s) from the nova bank
$ The nova bank approves a loan of 24 dollar(s) to karen
? sam requests a loan of 20 dollar(s) from the knox bank
$ The knox bank approves a loan of 20 dollar(s) to sam
? sam requests a loan of 47 dollar(s) from the apple bank
$ The apple bank approves a loan of 47 dollar(s) to sam
? james requests a loan of 16 dollar(s) from the nova bank
$ The nova bank approves a loan of 16 dollar(s) to james
? sam requests a loan of 41 dollar(s) from the knox bank
$ The knox bank approves a loan of 41 dollar(s) to sam
? james requests a loan of 50 dollar(s) from the apple bank
$ The apple bank denies a loan of 50 dollar(s) to james
? karen requests a loan of 8 dollar(s) from the knox bank
$ The knox bank approves a loan of 8 dollar(s) to karen
? james requests a loan of 44 dollar(s) from the knox bank
$ The knox bank approves a loan of 44 dollar(s) to james
? sam requests a loan of 46 dollar(s) from the apple bank
? james requests a loan of 8 dollar(s) from the nova bank
$ The apple bank denies a loan of 46 dollar(s) to sam
$ The nova bank approves a loan of 8 dollar(s) to james
? karen requests a loan of 3 dollar(s) from the nova bank
$ The nova bank approves a loan of 3 dollar(s) to karen
? karen requests a loan of 41 dollar(s) from the knox bank
$ The knox bank approves a loan of 41 dollar(s) to karen
? james requests a loan of 26 dollar(s) from the nova bank
$ The nova bank approves a loan of 26 dollar(s) to james
? sam requests a loan of 8 dollar(s) from the nova bank
$ The nova bank approves a loan of 8 dollar(s) to sam
? karen requests a loan of 45 dollar(s) from the knox bank
$ The knox bank denies a loan of 45 dollar(s) to karen
? james requests a loan of 10 dollar(s) from the nova bank
$ The nova bank approves a loan of 10 dollar(s) to james
? karen requests a loan of 13 dollar(s) from the apple bank
$ The apple bank approves a loan of 13 dollar(s) to karen
? sam requests a loan of 9 dollar(s) from the knox bank
$ The knox bank approves a loan of 9 dollar(s) to sam
? james requests a loan of 45 dollar(s) from the nova bank
$ The nova bank approves a loan of 45 dollar(s) to james
? karen requests a loan of 30 dollar(s) from the apple bank
$ The apple bank denies a loan of 30 dollar(s) to karen
? sam requests a loan of 43 dollar(s) from the nova bank
```

```
$ The nova bank approves a loan of 43 dollar(s) to sam
? james requests a loan of 26 dollar(s) from the knox bank
$ The knox bank denies a loan of 26 dollar(s) to james
? karen requests a loan of 19 dollar(s) from the nova bank
$ The nova bank approves a loan of 19 dollar(s) to karen
? james requests a loan of 2 dollar(s) from the nova bank
$ The nova bank approves a loan of 2 dollar(s) to james
? karen requests a loan of 49 dollar(s) from the nova bank
$ The nova bank approves a loan of 49 dollar(s) to karen
? sam requests a loan of 20 dollar(s) from the nova bank
$ The nova bank approves a loan of 20 dollar(s) to sam
? sam requests a loan of 1 dollar(s) from the knox bank
$ The knox bank approves a loan of 1 dollar(s) to sam
? karen requests a loan of 22 dollar(s) from the nova bank
$ The nova bank approves a loan of 22 dollar(s) to karen
? james requests a loan of 2 dollar(s) from the nova bank
$ The nova bank approves a loan of 2 dollar(s) to james
? james requests a loan of 34 dollar(s) from the nova bank
$ The nova bank denies a loan of 34 dollar(s) to james
? karen requests a loan of 17 dollar(s) from the nova bank
$ The nova bank denies a loan of 17 dollar(s) to karen


** Banking Report **

Customers:
 sam: objective 200, received 200
 james: objective 300, received 185
 karen: objective 200, received 179
 -----
 Total: objective 700, received 564

Banks:
 apple: original 100, balance 29
 knox: original 200, balance 4
 nova: original 300, balance 3
 -----
 Total: original 600, loaned 564


The financial market is closing for the day...
```

Let's review the output. The first couple of lines simply indicate that the program is starting and we are about to see the transaction log. This may remain on the screen for a half second or so as the customers sleep for a moment to let the banks gets started.

Once the transactions start, they appear very quickly. The first two messages show that *sam* is asking for 11 dollars from the *apple* bank, and then the *apple* bank approves this loan. Loan requests are prefixed with a **?** symbol, while bank replies are prefixed with **$**. You should be able to go through the complete transaction log and find matching request/reply pairs. Note that a bank may sometimes reply to several customer requests at once so we may sometimes see several **$** messages in a row. This is perfectly OK.

Once all of the customers have either received their full loan amount or have no more banks from which to borrow, the master process will display the final report. Here we see a list of customers,

showing their initial loan objectives, along with the amount they actually received. In addition, we see the total of all loan objectives, along with a total for all money received. Note that the total received can NEVER be more than the total objectives.

Information for the banks is then provided. For each bank we see their original resources, along with their final balance. In many cases, these balances will be close to 0, indicating that the bank has essentially run out of money. Note that in these cases the balance does not have to be exactly 0, since a customer will stop contacting a bank as soon as that bank denies a loan request (e.g., the customer asks for $20 but the bank only has $4). The final output line shows the total original resources, along with the total actually loaned. Again, the amount loaned cannot be greater than the original resources.

IMPORTANT: All of the summary information must be consistent. Most importantly, the *total loaned* by the banks must be exactly the same as the *total received* by the customers. If it's not, your application is broken.

One final thing: Because of the random pause before each loan request, as well as the random loan amounts in the loan requests, the numbers in the final report should be slightly different each time you run the program, even for the same input files.

ADDING A CHALLENGE: If you are able to get the above to work, using the same simple customer and bank information, then you already have the basic functionality required to get a passing grade. So what is required to get a full grade?

In case it's not immediately obvious, there are things that can go very wrong (apart from basic syntax errors). The main issue is that in order for the program to complete properly, all individual tasks must also complete properly. If a task hangs (i.e., is left waiting for a message that never arrives), then the program will never produce a final report. If a task terminates prematurely, then the program will probably crash when a message is later sent to that process (which doesn't exist anymore). Remember, the tasks really are independent – they don't automatically know if other tasks are still active or not.

In a simple configuration like the one above, this is not so hard to address. Erlang provides a timeout mechanism (using *after milliseconds*) in the request block to allow a task to stop waiting for messages if it hasn't received one in a while. So a bank could use such a timeout to shutdown, if it seems like the customers must have finished their requests. This works fine with a small example. But as the input sets get bigger and more complex, the communication patterns become much more irregular and it's harder to tell if the other tasks are completely done or not. So a very simple timeout mechanism might fail periodically, causing the program to crash. Consequently, you may have to use slightly more sophisticated communication logic to ensure that your program always ends properly and produces the proper final results.

Let's look at a couple of other examples. First, we'll look at a case where we have more customers and banks and one customer in particular wants a huge amount of money. In this case, all of the other customers will finish, but this one customer will hang around for a long time. The final summary info is below:

```
** Banking Report **

Customers:
 sam: objective 200, received 200
 james: objective 32150, received 2453
 karen: objective 200, received 200
 ahmed: objective 450, received 450
 sue: objective 3200, received 2772
 bing: objective 2650, received 2603
 cyrus: objective 230, received 230
 danie: objective 1900, received 1900
 gaston: objective 2000, received 2000
 carol: objective 450, received 450
 bruce: objective 180, received 180
 juan: objective 6460, received 2692
 ruby: objective 190, received 190
 emile: objective 120, received 120
 kira: objective 240, received 240
 -----
 Total: objective 50620, received 16680

Banks:
 apple: original 1000, balance 4
 knox: original 2000, balance 4
 nova: original 260, balance 1
 ing: original 3250, balance 4
 rbc: original 1805, balance 1
 cibc: original 4250, balance 3
 state: original 640, balance 0
 bmo: original 3500, balance 8
 -----
 Total: original 16705, loaned 16680
```

Here, all of the banks were effectively bankrupted, but some of the customers didn't receive all of their money. In particular, *james* wanted $32150, so he drained the banks almost by himself. Note, however, that the total loaned and total received still match exactly.

In the next example, we take the opposite approach and we include a bank with huge resources. The final report is below:

```
** Banking Report **

Customers:
 sam: objective 200, received 200
 james: objective 8150, received 8150
 karen: objective 200, received 200
 ahmed: objective 450, received 450
 sue: objective 3200, received 3200
 bing: objective 2650, received 2650
 cyrus: objective 230, received 230
 danie: objective 1900, received 1900
 gaston: objective 2000, received 2000
 carol: objective 450, received 450
 bruce: objective 180, received 180
```

```
 juan: objective 6460, received 6460
 ruby: objective 190, received 190
 emile: objective 120, received 120
 kira: objective 240, received 240
 mike: objective 1250, received 1250
 cammy: objective 4150, received 4150
 kim: objective 80, received 80
 mera: objective 460, received 460
 riba: objective 1905, received 1905
 felix: objective 410, received 410
 silvie: objective 940, received 940
 -----
 Total: objective 35815, received 35815

Banks:
 apple: original 1000, balance 0
 knox: original 2000, balance 1
 nova: original 260, balance 0
 ing: original 3250, balance 7
 rbc: original 1805, balance 14
 cibc: original 4250, balance 0
 state: original 640, balance 7
 bmo: original 350000, balance 343814
 honda: original 11405, balance 4757
 ford: original 2000, balance 4
 qubit: original 8150, balance 1836
 scam: original 1500, balance 5
 -----
 Total: original 386260, loaned 35815
```

In this case, the *bmo* bank had $350000, so loan requests would always be granted if they were sent to *bmo*. As a result, the total loans received by the customers are exactly equivalent to their original objective. In other words, they got every single dollar they wanted. On the other hand, multiple banks were completely wiped out, but a couple of the larger ones still had cash remaining. As always, however, the total loan amount is exactly equivalent to the total borrowed.

OTHER THINGS: As noted, this is an assignment that focuses on concurrency, not general programming issues. Consequently, I want to minimize the time spent on other things. So please keep the following things in mind.

- With Erlang, you send messages to other process by using their process IDs (this is the value returned by the `spawn` function. In some situations, you can just bind this value to a label and then use this later in a send. However, when processes have been spawned in another task, you don't necessarily know what their IDs are. Erlang provides a very simple mechanism for this. Specifically, you can *register* a name/processID pair using the `register(name, ID)` function. Later, you can use the `whereis(name)` lookup function - in another process - to get the processID of your target process.
- The `self()` function gives the process ID of the current process

- Erlang uses *if* for its conditional processing. Note that at least one of the conditional checks MUST return true; otherwise, you will get a runtime error. If needed, you can use something like the expression below. Basically, this says that if the first condition(s) isn't true, the default is just to match "true" and essentially do nothing.
```
if
  X > 3 -> do_something();
  true -> false
end.
```
- The `lists:foreach` function is a way to quickly process a list by performing a function on each of the values in the list.
- The `lists:foldl` function is Erlang's equiavalent to Clojure's `reduce` function (i.e., reduce all values in a list to a single value with a function like *sum*).
- While not absolutely required, it is good practice to send your message content as a tuple. Use an atom/label as the first element of the tuple to easily distinguish one message from another!
- Erlang has modules for *lists* and *maps* (and other things). Each contains many functions for manipulating the associated data structures. The online docs give many examples of their use.
- Erlang's *list comprehensions* can be used to easily build lists from other lists. For example, the code below makes a new list from the second element of each tuple in the original list:
`FruitList = [Fruit || {_ , Fruit} <- GroceryList]`
- Erlang allows you to use many expressions in a single function. They just execute one after another. You simply have to separate each expression with a comma. The function itself will end with a period.
- Some expressions – like `receive` and `if` expressions – use a semi-colon to end each block (though no semi-colon is used after the final block).
- The `rand` module is used for random number generation. `rand:seed(exsss)` can be used to initialize the generator and `rand:uniform(myInteger)` can be used to get a random number from 1 to myInteger.
- The function `length(myList)` can be used to get the length of a list. `lists:nth(n, myList)` can be used to extract the value at position `n`. Note, however, that list indexing in this case begins at 1, not 0.
- Basic printing can be done with `io:fwrite()` or `io:format()`. These functions work very much like `printf()` in the C language. You provide a list of values that are mapped into a formatted string. Again, the online docs provide many examples.

I think that covers most of the obvious issues. If you build on these ideas and utilize the documentation on the main Erlang website, which is pretty good, you should be able to mostly focus on the logic of the communicating tasks.

GROUP VERSION If you are working in a group, you will create the same Erlang application. However, you will also be creating a second version of the application in Java. The comparable 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 banking/customer data. Once you have the data, you will replicate the functionality of the Erlang program.

In this case, you will use Java's basic *thread* mechanism to create individual threads to represent each customer and bank. So just like the Erlang app, this will be a multi-threaded program. Each bank/customer entity will be constructed as a thread, and the threads will exchange the same info (e.g., loan requests, loan decisions) with both each other and the master thread, using the same logic/order described in the Erlang description above. Output to the screen (log and final report) will also take the same form and will be written from the main thread.

Important: 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 The graders will provide their own simple customer and bank text files. A very simple pair of files will be used to assess basic functionality (and award most of the point value). Then some larger examples will be used to see if your application performs well under different conditions. As noted, the test files will ALWAYS use valid data. No error checking of any kind is required on the input. All loan and bank values will use positive, non-zero integers as the loan and resource amounts. Basically, if your code works well in your own testing, it should work properly for the graders.

Note that Erlang automatically uses the current folder when searching for your modules. So nothing special has to be done to find them.

Finally, you do not have to provide any error checking on the file names passed on the command line. It is the markers' job to correctly input the names of their test files.

DELIVERABLES: Your Erlang submission will have just 3 source files. The "main" file will be called `money.erl` and will correspond to the master process. The second file will be called `customer.erl` and will include the code associated with the "customer" processes. The final file, of course, will be `bank.erl` and will represent the bank processes. Module names will be identical to the file names (minus the .erl extension). Do not include any data files, as the markers will provide their own. For group submissions, you will use the same set of three files.

Before preparing your submission, please create a simple README.txt file. Inside, indicate whether this is a solo project or a group project. For group projects, include the name(s) of the other member(s). Only one submission on Moodle will be made for the group and each person in the group will receive the same grade.

Once you are ready to submit, place the three .erl source files into a zip file (plus .java files for groups). 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" (for groups, the last name will coincide with the name of the person doing the actual submission). 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