NWEN303

Assignment 4

Actors

William Kilty
300473541

Task 1:

Alice:

Alice is an Actor who, upon receiving a message containing an ActorRef ‘r’, will ‘tell’ a new message to r, containing a new Wheat object, and say this message was sent by the Actor who sent the message to Alice in the first place. Alice will then ‘tell’ a message to itself containing the exact same information as the message received.

Alice acts as a sort of ‘middle-man’, creating wheat for an actor at another actor’s request.

Bob:

Bob is an Actor who, upon receiving a message containing an ActorRef ‘r’, will ‘tell’ a new message to r, containing a new Sugar object, and say this message was sent by the Actor who sent the message to Bob in the first place. Bob will then ‘tell’ a message to itself containing the exact same information as the message received.

Bob acts as a sort of ‘middle-man’, creating sugar for an actor at another actor’s request.

Charles:

Charles is an Actor containing a List of Wheat and a List of Sugar who, upon receiving a message containing either Sugar or Wheat, will look to see if there is any instance of the other in the two Lists. If not, Charles will add the Sugar or Wheat to the respective List, and return. Otherwise Charles will take a Wheat/Sugar out of the List, and use both the Wheat and the Sugar to create a Cake, and ‘tell’ the sender of the message a new message containing the Cake, referencing Charles itself as the sender.

Tim:

Tim is an Actor created with a specified int ‘hunger’, a boolean ‘running’ set to true, and an ActorRef ‘originalSender’ set to null.

Upon receiving a message containing a GiftRequest and if originalSender is still null, originalSender will be set as the sender of this message.

Upon receiving a message containing a Cake and if running is still true, reduce hunger by 1, print out "JUMMY but I'm still hungry " and Tim’s hunger value, and then - assuming hunger is less than 1 - set running to false and ‘tell’ a new message to originalSender containing a new Gift object and referencing itself as the sender.

OpenAkka:

OpenAkka is a class containing an alternative main method for use on other machines. If wanting to split the Actors across multiple machines, one would have one machine use Cakes.java, and all the others would run OpenAkka.

OpenAkka itself opens up a new ActorSystem under the port 2500 utilising AkkaConfig, and waits for the user to terminate the process after the system's input stream receives data after a \n is entered.

In the meantime, any Actor from Cakes.java on the main machine sent to the IP address this ActorSystem is listening to will be picked up and handled by the ActorSystem. This allows for data to be processed seamlessly over a network.

AkkaConfig:

AkkaConfig contains a single method which creates an ActorSystem for use in either Cakes.java or OpenAkka.java. The method will first look for valid functioning IP addresses on the current machine, and if there is exactly one present, will use it as the IP address to host the ActorSystem on.

A Config object will be created, which holds all the information necessary to set up the ActorSystem's functionality over a network using the chosen IP address and given port number.

It will then look through the map of given Actors to IP addresses and add to the Config object the knowledge to send those actors out to their intended addresses on the port 2500.

At last the ActorSystem is constructed, under the given name, and using the designated Config object. This ActorSystem is returned.

Cakes:

Cakes is a class containing the main method of the program.

The main method simply makes sure assertions are enabled across the program, calls the method 'computeGift' with 1000 as the input, and then if we got a successful Gift object out of the method, prints out the Gift.

The computeGift method takes in an int 'hunger'. It creates a new ActorSystem via AkkaConfig called 'Cakes', under the port 2501, with a map relating Tim, Bob, and Charles to different IP addresses. This is in order to tell the program to run those actors on different machines with those IP addresses under the port 2501.

It then creates - within this ActorSystem - Actors alice; bob; charles; and tim with constructors for themselves in case the program needs to initialise a new version of the Actor on a different machine, and with their corresponding names so the system recognises that Tim, Bob, and Charles should run on different machines.

The method then 'tells' alice and bob a message containing a reference to charles, from tim. A CompletableFuture is created by 'ask'ing tim with a new GiftRequest, and a timeout of 10,000,000 ms.

We then wait for the CompletableFuture to join (when tim's hunger gets to 0 and a Gift is sent to this Object by a now full tim). After this succeeds or times out, we kill all the actors and terminate the ActorSystem, allowing us to safely shutdown the program.

Task 3:

Designing the system for sugar taking 200 milliseconds to procure was a lengthy process. Using the system as it was in Task 2 led to each Bob being able to create multiple sugars in parallel, which undermines the point of the process taking 200 milliseconds. The only way to achieve the desired effect was to make the process blocking in some way. I tried a number of options including simply joining the future product in the MakeOne response. This and other options I felt didn't meet the 'feel' of Actor programming, so I settled on making the make() method for Bob synchronised. This has the desired effect without looking quite as bad from a coding standpoint.

Running the program with one Bob - taking 200 milliseconds to produce sugar - takes 205.689 seconds to consume 1,000 cakes. This is about what I would expect, as each sugar takes 200 milliseconds, and with 1,000 cakes, we end up with $200 * 1000/1000 = 200\, seconds$. 5 seconds of overhead plus a small amount for creating and consuming the cakes with the sugar is to be expected.

Running the program with four Bobs across different machines takes 53.589 seconds to consume the same amount of cakes. This is exactly as you would expect with four times the 'processing power' $200/4 = 50\, seconds$. With roughly 3 seconds of overhead and managing the other actors. This is an ideal result.

I measured performance by simply taking the time at the start of a test, and taking the time at the end of that test. This gives the number of seconds taken for that test to perform. Redoing each of the tests 5 times and averaging the scores provided a more accurate representation of how long the process takes. Given each test I took was within a few milliseconds of one another, and the times match up with what one would expect the times to be, I can be reasonably confident that the timing is accurate.