Elisa Gonzalez Boix
email:egonzale@vub.ac.be
office: 10F732

3 Concurrent Programming: The Internet Cafe

Ambienttalk's tutorial and language reference are available at `http://soft.vub.ac.be/amop/`. The lab session material is available at `http://soft.vub.ac.be/amop/teaching/dmpp`

3.1 Idea

This exercise introduces AmbientTalk's concurrent building blocks: actors and asynchronous message passing. The idea is to implement an internet cafe, i.e a place where customers can use a computer with access to internet. The internet cafe and customers are going to be modeled as actors. The internet cafe consists of a computer room with a limited number of computers. Each computer has an id that identifies its position in the room. When customers ask for a computer they get back a computerId if there is room in the computer room. If a customer hasn't received a computerId within a certain amount of time, the customer leaves the internet cafe.

3.2 Implementing the internet cafe

We will implement the internet cafe application starting from a skeleton code shown below (included in the lab session material)

```
def MAX_COMPUTERS := 2;

def internetCafe( capacity := MAX_COMPUTERS) {
   actor: { |capacity|
      def computerRoom :=  ... //TODO

      def getRoom(){ computerRoom };
  };
};

def makeCustomer(name, internetCafe) {
  //TODO
};

def sessionModule := object:{
  def sessionTest(){
    //TODO
  };
};
```

An internet cafe is created by invoking a `internetCafe` function which returns a far reference to an actor. The actor provides a method `getRoom` so that customers can access the computer room. Customers are created by invoking `makeCustomer` Use the above skeleton to incrementally grow the internet cafe as follows:

a) Implement the data structure for computer room using a guarded object (found in `/at/lang/guards.at`). Recall that a guard is a predicate which must be evaluated to true in order to execute an asynchronous message sent to an object.

   You will need to implement in the following methods on the `computerRoom` object to manipulate the occupancy of the room:

   getComputer adds a customer in the room and returns a `computerId` of the position assigned to the customer. Remember that this method can only be executed as long as there is space in the computer room.

   freeComputer(computerId) releases the given position in the computer room.

b) Implement the `makeCustomer` function which returns a far reference to an actor whose behaviour implements a `askComputer` and a `leaveComputer` methods used to make a customer ask for a computer in the internet cafe, and leave the assigned computer, respectively.

   Recall: actors do not have access to the enclosing lexical scope!

c) Adapt your implementation to pass the `testAsyncOneCustomer` unit test.

   Hint: the `askComputer` method should return a future which is resolved with the customerId received from the computer room.

d) Extend your implementation so that customers leave the internet cafe when they don't receive a `computerId` within a certain amount of time (i.e. 10 seconds) when they ask for a computer.

   Hint: Take a look at `@Due` annotation in futures to put time boundaries to the delivery of asynchronous messages.

e) Implement `testAsyncFullOccupancy` which checks that the computer room is full after `customer` and `customer2` asked for a computer.

f) So far we assumed that customers receive by parameter the internet cafe to interact with. So both actors live in the same virtual machine. Add the necessary code to use the AmbientTalk's network facilities to discover in the network an internet cafe to interact with, i.e. turn your concurrent application into a distributed one.

   You will need to adapt your implementation to add service discovery code so that the internet cafe actor exports the services of the computer room. Once a customer discovers an internet cafe service, it asks the cafe for a computer.

   Recall: by default, AmbientTalk's network access is shut down!