Distributed Computing with Message Passing

Homework 5

Alex Lindberg
alex5@kth.se

March 3, 2023

Introduction

This is a short report detailing the fifth homework — *Distributed Computing with Message Passing*. Implementations for task **[1,3]** are included. All implementations were done in Java using RMI.

1. Distributed Pairing 1

This implementation uses a Server `Teacher` and a client, `Student`, with the remote interface

```
public interface PairingService extends Remote {
  public void requestPartner(String studentName)
        throws RemoteException;
  public String receivePartner(String studentName)
        throws RemoteException;
}
```

The "Teacher" (server) is exported and bound to the RMI registry under the alias `teacher`, after which the Servers is ready to receive connections. The client program creates a number of clients that locate the teacher in the registry and start by requesting a partner. The client the waits and calls to receive a partner while waiting. When the client has gotten assigned a partner the client prints and terminates. To handle an odd number of students the server keeps track of the amount of students and the amounts of pairs it has left to assign. If the amount of students is odd then the one making the request gets paired with themselves.

The server uses synchronized methods to ensure thread safety.

Result

Below is a sample of the output.

```
[ ID1217/L5 ] $ java pairing.Teacher 5
Unbound teacher
Teacher is ready and bound in the RMI registry.
Student 5 requests a partner
Student 4 requests a partner
Student 5 gets paired with Student 4
Student 1 requests a partner
Student 3 requests a partner
Student 1 gets paired with Student 3
Student 2 requests a partner
Student 2 is the odd one out
```

And for the student we get

```
[ ID1217/L5 ] $ java pairing.Student 5
Student 5  : my partner is Student 4!
Student 1  : my partner is Student 3!
Student 2  : my partner is Student 2!
Student 4  : my partner is Student 5!
Student 3  : my partner is Student 1!
```

3. Dining philosophers

This implementation of the Dining Philosophers problem was done as follows. Very similar to the first assignment we create a Server called butler and clients called philosopher. We use this remote interface:

```
public interface Fork extends Remote {
    public void pickUp(int philosopherId) throws RemoteException;
    public void putDown(int philosopherId) throws RemoteException;
    public void leave(int philosopherId) throws RemoteException;
}
```

The difference here is that we loop these methods until some criteria has been fulfilled after which we can call the `leave` function which will terminate the server when the last philosopher has eaten.

```
while (hunger > 0) {
    think();
    butler.pickUp(philosopherId);
    eat();
```

```
    butler.putDown(philosopherId);
}
System.out.format("P-[%d]: \tFinished%n", philosopherId);
butler.leave(philosopherId);
```

Other that the Server termination and the looping both assignment are mostly the same. One small difference might be that both `pickUp` and `putDown` are sort of request methods, meaning the pickup needs to wait for a `putDown` request before a thread can continue (if there are none available).

Result

The following are parts of a run for the butler...

```
[ ID1217/L5 ] $ java philosophers.Butler
Unbinding butler from registry...
No butler bound
Butler ready
^[[AP-[4]:  Give fork
P-[1]:  Give fork
P-[1]:  Get back fork
P-[4]:  Get back fork
P-[3]:  Give fork
P-[1]:  Give fork
...
P-[2]:  Leaving Table
P-[0]:  Get back fork
P-[0]:  Leaving Table
P-[3]:  Get back fork
P-[3]:  Leaving Table
Butler going home.
```

and the Philosophers...

```
[ ID1217/L5 ] $ java philosophers.Philosopher
Starting...
P-[0]:  Fetching registry
P-[0]:  Locating Butler
P-[1]:  Fetching registry
P-[0]:  Running...
...
P-[3]:  Eating, hunger: 1
P-[1]:  Eating, hunger: 0
P-[3]:  Thinking
P-[4]:  Eating, hunger: 0
```

```
P-[1]:  Finished
P-[2]:  Eating, hunger: 1
P-[2]:  Thinking
P-[2]:  Eating, hunger: 0
P-[4]:  Finished
```