Computer Networks and Distributed Systems (CNDS)
Assessed Coursework 2 : Distributed Systems

Due Date: Monday, 3rd March 2025, 5.00pm

(You can work individually or in pairs)

The purpose of this exercise is to gain experience programming with RMI and UDP, compare them for relative reliability and ease of use and acquire an intuition of their performance when running in a local area network.

The system to be implemented comprises *Sensor*s that use UDP message passing to send a number of measurements to a *FieldUnit*. The latter, in turn, uses RMI to send a number of statistics to a *CentralServer* as shown in the Figure below. Each message contains: a message sequence number, the measurement recorded from the sensor (a non-integer value) and the total number of messages to be sent.

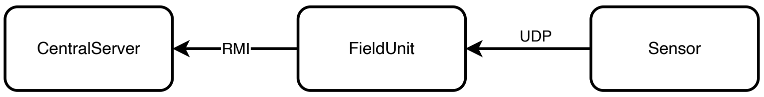

Figure 1: Layout of the distributed system

The *FieldUnit* keeps track of the messages received from the *Sensor* and, when there are no more messages to be received, performs the following operations in order.

1. It outputs a summary of the number of messages received and enumerates lost messages.
2. Given the measurements received from the *Sensors*, it computes an array of simple moving averages[1] using the previous $k = 7$ points. Each element e_i of the resulting array is computed using the formula in equation 1, where m_j is the j-th element of the vector of measurements.

$$e_i = \begin{cases} m_i & i < 7 \\ \frac{1}{k}\sum_{j=i-k+1}^{i} m_j & i \geq 7 \end{cases} \tag{1}$$

3. Using Java RMI, the *FieldUnit* sends the elements of the moving average vector separately, each as an individual invocation, to the *CentralServer*. Each message contains a message sequence number, a non integer value and the total number of messages to be sent.

When all the information has been received by the *CentralServer*, the server performs a summary of the number of messages received and of the messages that have been lost in the communication between the *FieldUnit* and the *CentralServer*, if any.

Your solutions should deal with exceptions appropriately.

1 What to do

Download the template source code and scripts from Scientia. You are not obliged to use these; however, they will make it simpler to develop the solution. The *MessageInfo* class (in the *common* folder)

[1]https://en.wikipedia.org/wiki/Moving_average

provides a container for the data to be sent and also has a constructor that extracts the data from a string representation.

Template code for each of the components can be found in the *central-server*, *field-unit* and *sensor* folders. For each component, the implementation of the methods outlined below must be provided.

Sensor: The Sensor implements the *ISensor* interface shown below. It sends a specified number of messages (given as input to the program) to the *FieldUnit*. (You can use the *getMeasurement()* method provided to simulate a measurement operation.)

```
public interface ISensor {
  /* sends N measurementes to the Field Unit*/
  public void run(int N, int timeout) throws InterruptedException;

  /* Send the message 'msg' to 'address' on port 'port' */
  public void sendMessage(String address, int port, MessageInfo msg);

  /* Simulate one measurement */
  public float getMeasurement();
}
```

Provide the implementation of the *constructor*, *sendMessage* and *run* methods and the *main* method of the program in the *Sensor.java* classfile.

After the build (see notes at the end of the coursework for more detailed instructions), you can execute the *Sensor* from the command line by calling:

```
./sensor.sh <field_unit_address> <field_unit_port> <number of messages>
```

Field Unit: The *FieldUnit* implements the *IFieldUnit* interface below. It connects via RMI to the *CentralServer*. Then, it listens on a UDP port for incoming messages from the Sensors. When all messages have been received, the *FieldUnit* shows if any messages have been lost and calls the *sMovingAverage* method to compute the moving averages on the received messages. Finally, the *FieldUnit* sends the averages to *CentralServer* through RMI.

```
public interface IFieldUnit {
  /* Save message into local data structure */
  public void addMessage (MessageInfo m);

  /* Compute the k-points moving averages of all messages */
  public void sMovingAverage (int k);

  /* Listen on UDP port UNTIL there is no more to receive */
  public void receiveMeasures(int port, int timeout) throws SocketException;

  /* Set up RMI client */
  public void initRMI (String address);

  /* In this function, we call CentralServer.receiveMsg() to send the message to the
       Central Server via RMI */
  public void sendAverages ();

  /* Print Stats */
  public void printStats ();
}
```

Provide the implementation of the constructor, methods of the *FieldUnit* interface below and the main method of the program in the *FieldUnit.java* classfile.

CentralServer: The *CentralServer* implements the *ICentralServer* interface below. The *FieldUnit* uses the *receiveMsg()* method to send messages to the server through RMI.

```
public interface ICentralServer extends Remote {
  /* Receive Message. Called by the client to send a value to the CentralServer
     through RMI*/
  public void receiveMsg(MessageInfo m) throws RemoteException;
}
```

Provide the implementation of the Central Server including its *receiveMsg()* method defined in the interface below, the *main()* function of the *CentralServer.java* class file and the methods necessary to report the statistics (see template implementation).

Example of output:

The following is an example of the output you should obtain from your program:

Process 1: Central Server

```
username@edge01 cw % ./centralServer.sh
Central Server ready
[Central Server] Received message 1 out of 20. Measure = 22.58044
....
[Central Server] Received message 20 out of 20. Measure = 26.994593
Total Missing Messages = 0 out of 20
```

Process 2: Field Unit

```
username@curve07 cw % ./fieldUnit.sh 9999 localhost
[Field Unit] Listening on port: 9999
[Field Unit] Message 1 out of 20 received. Value =  22.58044
....
[Field Unit] Message 20 out of 20 received. Value =  31.654812
Total Missing Messages = 0 out of 20
===============================
[Field Unit] Computing SMAs
[Field Unit] Sending SMAs to RMI

[Field Unit] Listening on port: 9999
```

Process 3: Sensor

```
username@texel11 cw % ./Sensor.sh 127.0.0.1 9999 20
[Sensor] Sending message 1 out of 20. Measure = 22.58044
.....
[Sensor] Sending message 20 out of 20. Measure = 31.654812
```

2 Evaluate your findings

When the clients and servers are working, run some experiments on three computers *in different parts of the lab (i.e., not physically near each other)* sending an increasing numbers of messages (e.g., from 20-100 with increments of 20, then, 200, 300 and 400) from the Sensor. Then identify the situations in which messages are lost. If you are accessing the lab network remotely, you should pick three computers whose hostname differs significantly among each other (i.e. sprite1, edge1 and texel1, and sprite1, point2 and edge3 would be good combinations while choosing, for instance, sprite1, sprite2 and texel1 could lead to biased results).

Do not send an excessive number of messages e.g., more than about 4000 messages you may overload the lab network.

3 What to submit

You should hand in the following as a single pdf file. You should additionally include an archive (e.g., zip file) of your code:

1. A short summary (no more than 1 page) describing your findings from running the programs. You should address the following points:
   (a) For each communication, measure the amount of time that is needed to receive all the messages.
   (b) Identify which communication is faster and give the possible reasons for this.
   (c) For each communication, give the possible causes for any messages being lost?
   (d) Are there any patterns in the way messages are lost?
   (e) What is the relative reliability of the different communication mechanisms? When is one preferable to another?
2. Proof that both the RMI and UDP programs actually ran, e.g. console logs or screen dump plus an indication of which message numbers, if any, were lost.
3. A well formatted listing of the completed code for the 4 classes (2 client server pairs) which is easy to read. Start each of the 4 classes on a new page and avoid long lines (i.e. keep lines below <70 characters).

   You are encouraged to write your report in LaTeXand use the *lstlisting* package for the code.

Please provide program listings for all the classes you have changed or wrote (you do not need to include the templates that you have used but not modified i.e. MessageInfo). The following order would be preferred:

1. *Sensor*. Both interface and class
2. *FieldUnit*. Both interface and class.
3. *CentralServer*. Both interface and class.

Notes

Download the template source code and scripts from *Scientia*. You are not obliged to use these; however, they will make it simpler to develop the solution. Below are a few notes on the files provided:

The class *MessageInfo* (in the *common* folder) provides a container for the data to be sent and also has a constructor that extracts the data from a string representation. Outline code for each of the client/server pairs can be found in the *server*, *local-central* and *field-unit* folders.

The *Makefile* allows Linux users to use *make* to compile the various parts of the exercise. It can also be used to help configure your preferred development environment with the correct commands, flags and parameters. The shell scripts (e.g., *sensor.sh*, etc.) allows you to execute the various parts of the exercise.

Remote Access:

There are many options to access the College network and test your code on lab machines remotely. Detailed instructions can be found at the link ”SSH to Lab computers” under ”Resources” below. For the purpose of this coursework, you can either write the code on your home computer and then download and run the experiments on the lab computers develop the solutions in the lab.

To remotely use lab computers, you will need to:

1. Open a terminal window (or use PuTTY) and SSH into one of the gateways.
2. run */vol/linux/bin/freelabmachine* to get the hostname of a free lab machine.

3. SSH into the hostname returned (**it is important that you don't run anything on the gateway machine**).
4. copy (or download) your project into your home directory. You should remember that your home directory is unique across all college machines (i.e. you won't need to ssh necessarily into the same machine to access your project).

To run the experiments remotely, you will need to simultaneously access three different computers. To do so you can:

1. execute steps 1,2 and 3 above thrice.
2. cd into the project directory on the computers.
3. run the programs on the three computers.

If you connect through the College VPN and you already know the which lab computers you want to use (i.e. hostname), you can directly ssh into the lab computer (through the VPN) and skip steps 1,2.

You can use the *scp* utility to copy your project from your home computer to the lab computers. For further details see documentation at the link "Accessing and transferring files to lab computers" under "Resources" below.

Resources:

Sockets: http://docs.oracle.com/javase/tutorial/networking/sockets/
Datagrams: http://docs.oracle.com/javase/tutorial/networking/datagrams/
RMI: http://docs.oracle.com/javase/tutorial/rmi/
SSH into lab computers: https://www.imperial.ac.uk/computing/csg/guides/remote-access/ssh/
Accessing and transferring files to lab computers: https://www.imperial.ac.uk/computing/csg/guides/file-storage/scp/