SCHOOL OF COMPUTING, QUEEN'S UNIVERSITY

CISC324: Operating Systems

Lab 6

Due on **Monday, November** 30th

This lab may be your first time programming exercise in which you will be writing concurrent programs that run on different computers and that communicate and cooperate using messages sent over the network. You will be using Java RMI (Remote Method Invocation) and Java sockets as a message passing technology to complete this lab.

Overview of Java RMI and Java Sockets.

I. Remote Method Invocation

RMI (Remote Method Invocation) is a Java message passing technology developed by Sun Microsystems. It is implemented as a set of classes that allow the manipulation of objects on a remote computer in the same way as it happens in a local computer. RMI can be seen as an object oriented RPC (Remote Procedure Call) mechanism. The main purpose of RMI, is the allow a computer to invoke, execute, and receive results returned back from a method that was executed in a different Java virtual machine i.e., remote computer (viz., Figure 1). Note that, RMI can also be used on a localhost computer where one Java object invokes a method on another Java object through the local TCP/IP stack. RMI is used to build distributed applications. It is provided in the package `java.rmi`. Within the framework for CISC324, we are more concerned about having a thread running on one computer A and sharing data through message passing with another thread running on another computer B.

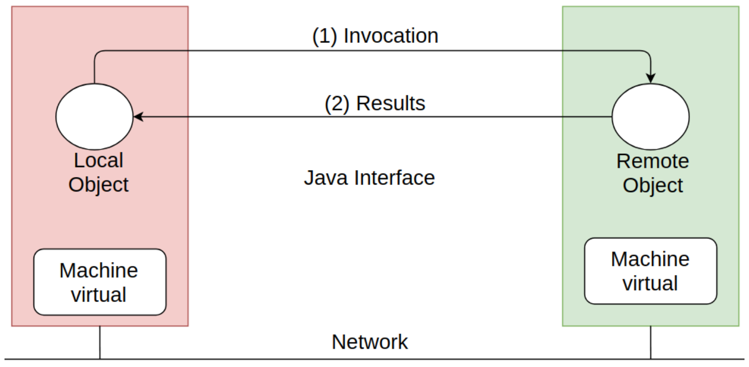

Figure 1. Remote Method Invocation Principle.

Conceptually, to use Java RMI, you need three main components that use `java.rmi` package:

1. An interface component: which presents what a remote server provides as services. Usually, it contains the prototypes of the different methods that the remote server implements. This interface takes the following form:

```
Import java.rmi.*;
public interface INTF extends Remote
{
```

```
data_type method_1_name(data_type p1, data_type p2, ..., data_type pn);
data_type method_2_name(data_type p1, data_type p2, ..., data_type pn);
...
data_type method_n_name(data_type p1, data_type p2, ..., data_type pn);
}
```

The interface should have a name (e.g., INTF in this case) and should extend the class `Remote`. It must import the `java.rmi` package.

2. A server component: which implements the remote server. The server implements the remote interface defined in point (1). It implements all the methods that have been declared in the interface. The server is a class that inherits from `UnicastRemoteObject` that contains all methods and attributes needed for processing a remote object.

```
Import java.rmi.*;
class Server_name extends java.rmi.server.UnicastRemoteObject implements INTF
{
//Define the constructor
public Server_name throws RemoteException { }
//Implements the methods
void method_1_name(data_type p1, data_type p2, ..., data_type pn)
{
...
}
void method_2_name(data_type p1, data_type p2, ..., data_type pn)
{
...
}
void method_n_name(data_type p1, data_type p2, ..., data_type pn)
{
...
}
//Defines the main method of the server (where server starts).
//It should instantiates the server object and register it by
//associating a name to it, so that clients can find it.

static public void main()
{
 int port = 1099; //server communication port
 try
 {
   Server_name S = new Server_name();
   Registry R = LocateRegistry.createRegistry(port);
   R.rebind(“ServerName”, S);
 }
 catch(Exception e) {}
}
}
```

3. A client component: which invokes the methods implemented on a remote computer. It consists of obtaining a reference on the remote object (which must be registered by the server) then invoke the desired method on that reference object.

```
Import java.rmi.*;
class Client_name
{
//Obtain a reference on the remote object, then invoke the method
static public void main()
{
  int port = 1099;
  try
  {
   INTF S;
   Registry R = LocateRegistry.getRegistry(“remote-IP-Address”,port);
   S = (Interface_name)(R.lookup(“ServerName”));
   data_type returned_result = S.method_k_name(p1,p2, ..., pn);
  }
  catch(Exception e) {}
}
}
```

The client has to first locate the server to obtain the server's reference by presenting the IP address of the remote server (if local then 127.0.0.1) and the communication port on which the server is listening (i.e., waiting for connections). Once obtained, the client can invoke remote methods on that object (which is of type INTF in this case). If the client wants to get back the results of some computations, then the remote method should have a return type.

Important. *You should always start running the server before running the client as the server has to wait for the client which initiates a connection.*

II. Sockets

Network sockets are message passing tools used to establish a connection between two distant computers connected over a telecommunication network (or within the localhost). They allow a computer to send and receive messages (containing data) to and from a remote computer. Within the framework for CISC324, sockets can be used to allow a process or thread running on a computer A, to communicate and share data with another process or thread running on another computer B.

Sockets work on client/server architectures, which means, if sockets are to be used between computer A and computer B to exchange messages, then one of the computers takes the role of a client (running a client socket) whereas the other takes the role of the server (running a server socket). The socket on the server side (server socket) waits for incoming connections from client sockets running on remote computer (clients). Sockets that run on the client computer, start establishing a connection with a remote computer that runs the server socket which accepts the connections. Once connected, the two computers can exchange messages.

In Java, there are two types of sockets: Client socket (provided in `java.net.Socket`) and Server socket (provided in `java.net.ServerSocket`). A client socket is a socket which starts establishing a connection with a remote server socket to request a service from the server. The server socket is a socket that listen and waits for an incoming connection from a client socket on a specific communication port (viz., Figure 2).

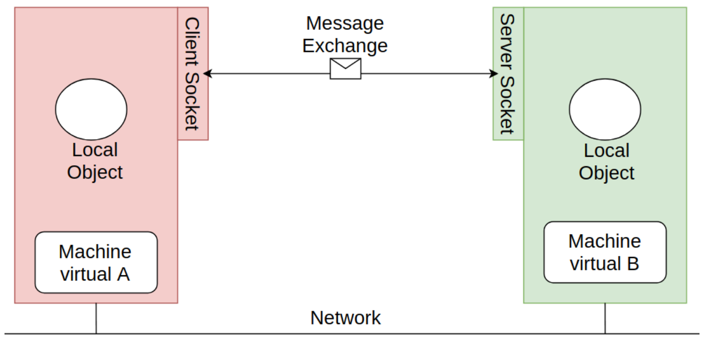

Figure 2. Socket Principle.

Server socket. The server which runs the server socket will be implemented as follow:

```
Import java.net.*;
class Server_name
{
static public void main()
{
  try
  {
  //Create a server socket on port 1234 with a maximum of 10 simultaneous connections
   ServerSocket ss = new Socket(1234, 10);
  //When the server detects the connection, it accepts or rejects it
  //It creates a new socket ssc on the same port to hold the connection and
  //continue listening for other connections
   Socket ssc = ss.accept();
  //Start exchanging messages
   <Exchange messages>;
  }
  catch(Exception e) {}
}
}
```

Client socket. The client which runs the client socket will be implemented as follow:

```
Import java.net.*;
class Client_name
{
static public void main()
{
  try
  {
  //Create a client socket on port 1234 and remote Ip address 192.168.1.13
  //This assumes that the server in the other side has 192.168.1.13 as its
  //Ip address and is listening on port 1234.
   Socket cs = new Socket(‘‘192.168.1.13’’, 1234);
  //Start exchanging messages
   <Exchange messages>;
  }
  catch(Exception e) {}
}
}
```

To exchange messages, Java sockets provide two types of data streams. The Input stream (provided by `java.io.ObjectInputStream`) used to receive messages and the Output stream (provided by `java.io.ObjectOutputStream`) used to send messages. A very important point is that: when a thread is sending a message using the output stream entity, the remote thread must be using the input stream entity to receives the message. If the message is sent while the remote thread was not waiting for the message (i.e., synchronized with the sender) on the input stream entity, the message will be lost and this may end up in a deadlock.

To send a message of type string, one of the threads (client or server) executes the following instructions:

```
  //Here the client is sending a message m of type string via socket cs
   ObjectOutputStream out = new ObjectOutputStream(cs.getOutputStream());
   out.writeObject(m);
   out.flush();
```

At the same time, the other side (e.g., server) must be waiting for the message on its input stream:

```
  //Here the server is waiting for message string m via its socket ssc
   ObjectInputStream out = new ObjectInputStream(ssc.getInputStream());
   m=readObject();
```

When the two threads terminate, the created sockets as well as the I/O streams should be closed:

```
in.close();     out.close();
ssc.close();     sc.close();     ss.close();
```

Important. *You should always start running the server before running the client as the server has to wait for the client which initiates a connection.*

Lab exercise.

In this lab, you are asked to realize an interprocess/interthread communication mechanism using Java Sockets and RMI to allow different processes/thread to communicate over a TCP/IP (*TCP/IP stands for Transmission Control Protocol/Internet Protocol, which is a protocol stack that allows heterogeneous computers to communicate over the Internet or a local network*). network. When a system is composed of multiple computers connected over a network and running different processes/threads that communicate and cooperate within the framework of a same application, the system is qualified as a distributed system. The set of all collaborative algorithms run by each process on each computer, constitutes a distributed algorithm.

For this lab, we propose to compare different page replacement algorithms in a centralized system that uses virtual memory, using a distributed system. To that end, we implement each page replacement algorithm on a particular computer server. We also suppose the existence of multiple processes/threads (called clients) which request access to a number of pages according to a local reference string. Each server implementing a page replacement algorithm, can hold up to `N` frames. When a client requests a page from a given server, then if the page is in the server frames, the server sends back the requested page to the client. Yet, if the page is not on the concerned server, a page fault occurs and the server requests the central server to swap in the missing page from the HDD (Hard Disk Drive) and send the page to the server. Once the server receives the missing page, it runs a page replacement algorithm to insert the new page in a selected frame. All the pages are considered “read only” pages, so no worries about dirty pages that have to be written back to the HDD.

For comparison purpose, each client broadcast (sequentially) its page request to each server implementing a page replacement algorithm. Because each client has its own reference string, only one request is sent by reference. To broadcast their requests, the set of clients adopt a sequential communication mechanism using a token (*Token: is a unique message that is shared and exchanged between a set of computers to determine which computer can exclusively access the network to send its messages and avoid collision*). In other words, the clients are initially organized in a unidirectional ring on which the token circulates. The client which receives (or holds) the token is the only client who has access to the network to send its request. Once the client has sent its request (which contains one reference e.g., page number), it sends the token to the next client in the ring and waits for the token to be back so that it can send its next request. This occurs till all the reference string is consumed.

To exchange the token between different clients, clients use Java sockets. Also, to send their requests to the page replacement servers, they use sockets as well. Each server is waiting for a connection from a client on a specific dedicated communication port. However, all page replacement servers communicate with a unique central server using Java RMI to request a missing page to be swapped in from the HDD.

We consider the system architecture depicted in Figure 3. The system is composed of two client threads running on different computers and communicating with each other using Java sockets. Each client has a reference string of twenty logical addresses. The system also uses three page replacement servers (thread servers): FIFO (First In First Out) server, LFU (Least Frequently Used) server, and MFU (Most Frequently Used) server. Each page replacement server has a local memory of three frames to hold logical pages. The three page replacement servers communicate with a unique central server (central thread) using Java RMI to request a missing page in the case where a server encounters a page fault. The central server swaps in the requested page from the HDD and sends it to the requesting page replacement server.

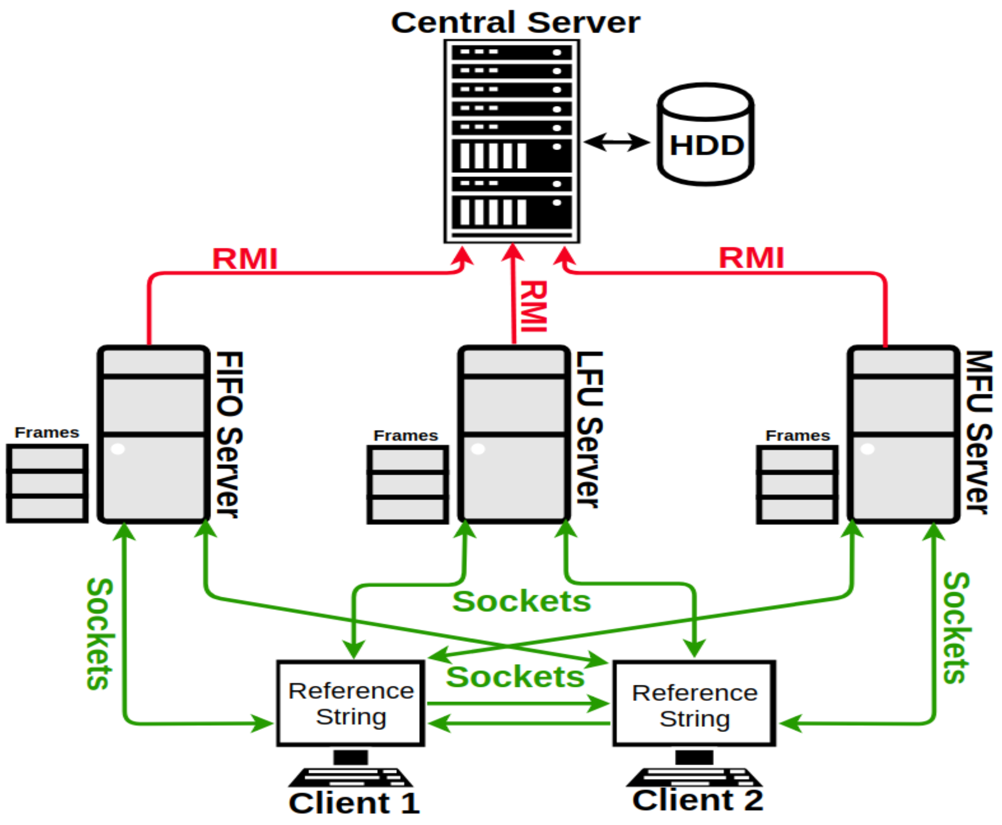

Figure 3. Lab 6 system architecture.

To experiment the system, we propose the following reference string for each client:

Client 1: `{7, 0, 1, 2, 4, 3, 0, 6, 2, 3, 0, 5, 2, 1, 0, 5, 1, 7, 0, 2}`

Client 2: `{3, 1, 4, 2, 3, 6, 7, 0, 1, 3, 6, 4, 2, 0, 1, 3, 2, 7, 0, 1}`

To start with, you are provided with two Java applications:

- ***Application 1.*** This application consists of two Java threads, one acting as a client (`Client.java`) and the other acting as a server (`Server.java`). The client and server threads use Java RMI (Remote Method Invocation) to communicate. The server thread provides a method to compute the square of a positive integer. The client generate a positive integer then sends it to the server thread over the network by remotely invoking the square method on the server using RMI mechanism. The interface that presents the services provided by the server is defined in the Java file `Server.Interface.java`. You can use this code as a building block for the lab whenever RMI is required to be used.

- ***Application 2.*** This application consists of two Java threads, both alternatively acting as a client and a server. These two threads (`T1.java` and `T2.java`) use Java Sockets to communicate and exchange messages. When one of the threads is acting as a server thread, it waits for the other thread which is acting as a client to initiates a connection and send an integer. The server increments that integer (which it receives from the client), becomes a client, and sends back the new integer to the old client which now is acting as a server (waiting for a connection). This ping-pong behavior is repeated for 5 rounds. You can use this code as a building block for the lab whenever sockets are required to be used.

What should be done and submitted

- Implement the system described in Figure 3.
- Create a Java file for each component:

```
Client_1.java, Client_2.java, FIFO_Server.java, MFU_Server.java,
LFU_Server.java, Central_Interface.java, and Central_Server.java
```

- Run the system for different server frame capacity i.e., `F=3` frames, `F=4` frames, and `F=5` frames. Take note of the number of faults per server and per client (reference string) for each value of `F`.
- Your codes should be well commented. You must add comments to explain why you have used a particular variable and what each piece of code performs.
- Use a ReadMe.txt file to explain the settings that you have used to produce each of your outputs. You should also include you outputs.
- Place everything inside one folder and compress the file using zip.