Concurrent programming, Autumn 2006, Project: -Part A and C-

Team:
RMI Consulting Group

Team members:

Yi Ding
Jouni Kleemola
Jaakko Nyman
Arturs Polis

Foreword 1
Part A 1
1 questions & answers 1
Chapter 4..1
Chapter 8..1
PART C..2
1 Overview of RMI usage and purpose 2
1.1 Basic idea...3
1.2 Java RMI Architecture...3
2 Similarities and differences compared with course material and other RMI resources 8
3 User manual 9
3.1 Using RMI 9
Evaluation Part..23

Foreword

This document can be found at :
http://www.cs.helsinki.fi/u/kleemola/rio/GroupWork/Documentation.doc

Part A

1 questions & answers

Chapter 4

This question can be found at
http://db.cs.helsinki.fi/~kerola/php/practice.php?file=http://www.cs.helsinki.fi/u/kleemola/rio/rio_practiseproblem_ch4_s06_fi.xml

Which of the following is true?

1. In concurrent programs errors could easily be discovered by debugging and related corrections can be checked by testing.

Answer: Sorry, it is not correct. Although debugging and testing are widely used in solving error problems for sequential programs, due to the characteristics of concurrent programs, it is not possible to discover errors hidden in the program and that is why we use temporal logics to verify programs.

2. An invariant is true once we have proven - a) it holds in the initial state; b)under inductive hypothesis, it is true in one possible successor to the current state.

Answer: Sorry, it is not correct. Because in a concurrent program, there may be more than one successors and inductive steps must be proved for each one of the possible states following concurrent one.

3. It is easy to prove a liveness property.

Answer: Sorry, it is not correct. For a liveness property, all possible computational scenarios must be checked and this requires more complex theory together with sophisticated proof rules.

4. In model checking, once a falsifying state is found during constructing incremental diagram, the construction need not be carried further more.

Answer: Yes, it is true. And this is the principal of model checking
which makes possible the verification of some concurrent programs.

Chapter 8

This question can be found at

http://db.cs.helsinki.fi/~kerola/php/practice.php?file=http://www.cs.helsinki.fi/u/kleemola/rio/rio_practiseproblem_ch8_s06_fi.xml

1. In asynchronous communication buffers are not needed in channels.

Answer: No, because the sender may send several messages and the receiver may check the channel only at time to time.

2. Asymmetric addressing is preferred for programming client-server systems.

Answer: Yes, the client has to know the name of the service it requests while the server can be programmed without knowledge of its clients.

3. Remote Procedure Call involves active participation of two processes in synchronous communication.

Answer: No, the client requests a service and just waits for return values while for example Rendezvous does indeed involve active participation between the processes.

4. Channels enable the developer to construct decentralized concurrent programs that do not necessarily share the same address space.

Answer: Yes, this is true. For example RMI client-server architectures use this approach. Also operating systems use channels called pipes to enable programs to be constructed by connecting an existing set of programs.

PART C

1 Overview of RMI usage and purpose

RMI is basically used as a framework for distributed computing. With RMI, separate parts of a single program can exist in multiple Java environments on multiple machines. This means that different functionalities can exist in separate locations and that these functionalities can be invoked from basically anywhere where Java clients exist.

There are many cases where a distributed environment is beneficial. As applications grow large and complex they are more manageable and more robust when they are distributed among multiple machines. Resources can be physically located where they make the most sense. Dedicated machines can be configured optimally to perform specialized operations. Heavy processing can be allocated to the best suited hardware, data processing can be handled by centralized data servers, and web servers can just serve web pages.

Our motivation for learning RMI was:

- To enhance the understandings of fundamental concepts in concurrent problems such as mutual exclusion, deadlock and starvation as well as the basics of distributed computing.

- To get familiar with utilizing Java to implement concurrency programs.

- To study concepts of job oriented methods, RMI mechanism and message passing on distributed system through implementation of programs to solve two classic concurrency problems.

1.1 Basic idea

RMI is one of the fundamental APIs that Enterprise Java Beans are built on.

The client is defined as the application which uses a remote object, and the server is defined as the machine which hosts the remote object. Communication is two way. Below is the standard simple diagram showing the relationship.

To both the client and server, the application appears to be local even though it is actually distributed among multiple Java environments. This is achieved by implementing interfaces which define the methods which can be invoked. The interfaces define the contract between the objects. What is actually passed back and forth are primitive data types (passed by value), remote references (objects which implement java.Remote), or copies of local objects (objects which implement java.io.Serializable).

Remote objects are listed in the rmiregistry of the server. A client will query the rmiregistry for an object by its handle and will receive a reference to the remote object. The client then invokes methods on the remote object. What actually is happening is that the client is invoking methods locally on the stub which carries out the interaction.

Likewise on the server side, the server interacts with the skeleton. The operations of the Remote Reference Layer and the Transport Layer are hidden from both sides. This allows different implementations of the Java Virtual Machine, which may be on different platforms, to integrate seamlessly.

1.2 Java RMI Architecture

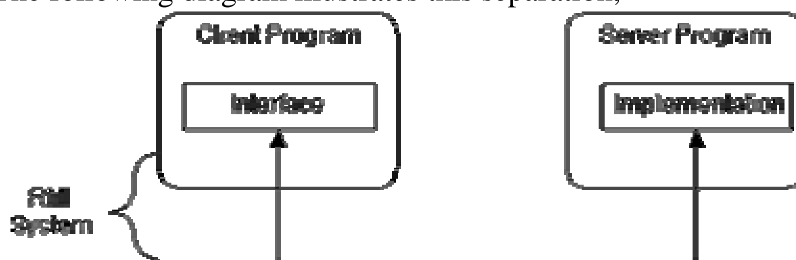
The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI creates a system that extends the safety and robustness of the Java architecture to the distributed computing world.

1.2.1 Essential features

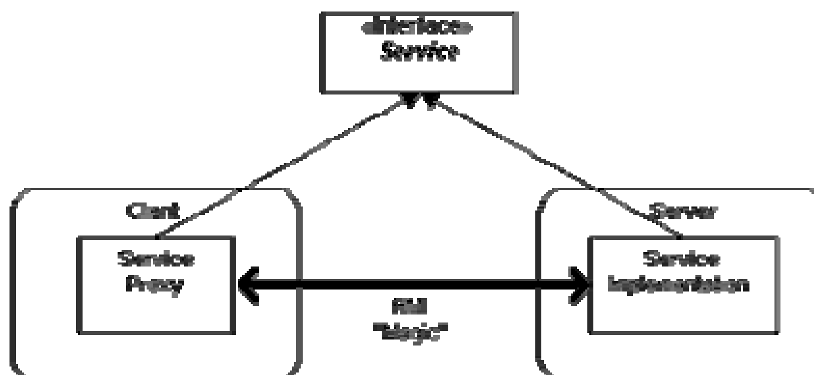
The RMI architecture is based on one important principle: the definition of behaviour and the implementation of that behaviour are separate concepts. RMI allows the code that defines the behaviour and the code that implements the behaviour to remain separate and to run on separate JVMs.

In RMI the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that interfaces define behaviour and classes define implementation.

The following diagram illustrates this separation,



One has to remember that a Java interface does not contain executable code. RMI supports two classes that implement the same interface. The first class is the implementation of the behaviour, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client. This is shown in the following diagram.



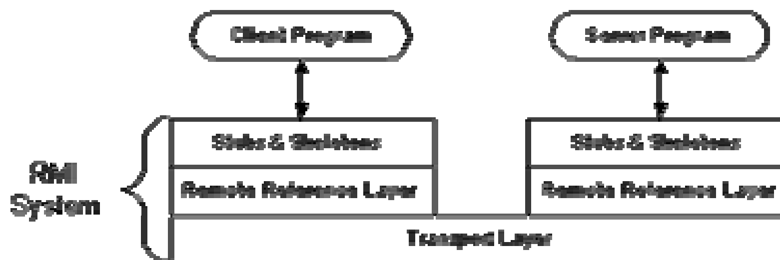
A client program makes method calls on the proxy (stub) object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

1.2.2 RMI Architecture Layers

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service. The Stub works on client side and the Skeleton on server side.

The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via Remote Object Activation.

The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

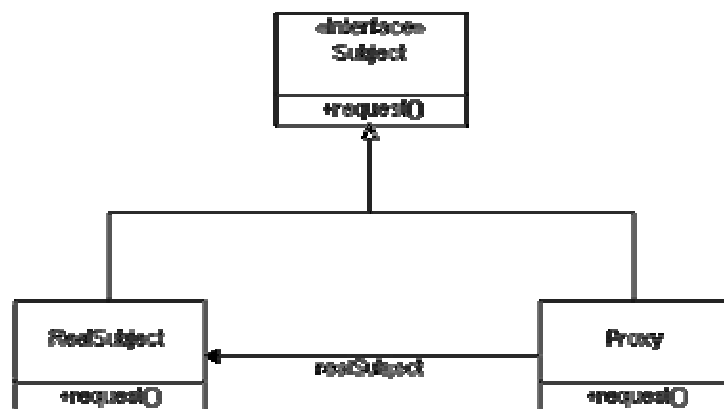


By using a layered architecture each of the layers could be enhanced or replaced without affecting the rest of the system. For example, the transport layer could be replaced by a UDP/IP layer without affecting the upper layers.

1.2.3 Stub and Skeleton Layer

The stub and skeleton layer of RMI lie just beneath the view of the Java developer. In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. The following class

diagram illustrates the Proxy pattern.



In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the RealSubject.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object. You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

1.2.4 Remote Reference Layer

The remote reference layer is a middle layer between the stub/skeleton layer and the transport layer. This layer is responsible for providing the ability to support varying remote reference or invocation protocols.

This layer provides a RemoteRef object that represents the link to the remote service implementation object. The stub objects use the invoke() method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services.

The JDK 1.1 implementation of RMI provides only one way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system. (If it is the primary service, it must also be named and registered in the RMI Registry).

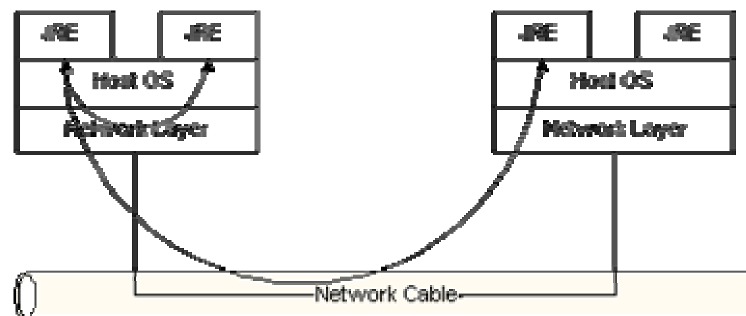
The Java 2 SDK implementation of RMI adds a new semantic for the client-server connection. In this version, RMI supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote

service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects.

1.2.5 Transport Layer

The Transport Layer makes the connection between JVMs. The transport layer is responsible for setting up connections to remote address spaces, managing connections, listening for incoming calls, maintaining a table for an incoming call, and locating the dispatcher for the target of the remote call and passing the connection to this dispatcher.

All connections are stream-based network connections that use TCP/IP.
Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. The following diagram shows the unfettered use of TCP/IP connections between JVMs.



In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially specified is now in two versions. The first version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server. The second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes. (Note that some alternate implementations, such as BEA Weblogic and NinjaRMI do not use JRMP, but instead use their own wire level protocol. ObjectSpace's Voyager does recognize JRMP and will interoperate with RMI at the wire level.) Some other changes with the Java 2 SDK are that RMI service interfaces are not required to extend from java.rmi.Remote and their service methods do not necessarily throw RemoteException.

The RMI transport layer is designed to make a connection between clients and server, even in the face of networking obstacles.

While the transport layer prefers to use multiple TCP/IP connections, some network configurations only allow a single TCP/IP connection between a client and server (some browsers restrict applets to a single network connection back to their hosting server).

In this case, the transport layer multiplexes multiple virtual connections within a single TCP/IP connection.

1.2.6 Naming Remote Objects

Clients find remote services by using a naming or directory service. So in fact the client locates a service by using a service. A naming or directory service is run on a well-known host and port number.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI). RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

On a host machine, a server program creates a remote service by first creating a local object that implements that service. Next, it exports that object to RMI. When the object is exported, RMI creates a listening service that waits for clients to connect and request the service. After exporting, the server registers the object in the RMI Registry under a public name.

On the client side, the RMI Registry is accessed through the static class Naming. It provides the method lookup() that a client uses to query a registry. The method lookup() accepts a URL that specifies the server host name and the name of the desired service. The method returns a remote reference to the service object. The URL takes the form:
"rmi://<host_name>[:<name_service_port>]/<service_name>"

where the host_name is a name recognized on the local area network (LAN) or a DNS name on the Internet. The name service_port only needs to be specified only if the naming service is running on a different port to the default 1099.

2 Similarities and differences compared with course material and other RMI resources

The course book does not cover that much of RMI. Of course RMI is a programming language specific thing and as such does not serve a good enough purpose to be thoroughly covered in the book. Remote procedure calls are also just briefly discussed in the material so there is not much to report about differences or similarities.

What was a bit lacking was the justification for RMI. From other sources we have learned that in RMI, resources can be physically located where they make the most sense. Dedicated machines can be configured optimally to perform specialized operations. Heavy processing can be allocated to the best suited hardware, data processing can be handled by centralized data servers, and web servers can just serve web pages. A large application can distribute the workload and achieve better efficiency and data isolation.

3 User manual

In this chapter a walkthrough about building RMI system is provided by using two different examples.

3.1 Using RMI

A working RMI system is composed of several parts.

- Interface definitions for the remote services
- Implementations of the remote services
- Stub and Skeleton files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A class file provider (an HTTP or FTP server)
- A client program that needs the remote services

For simplicity, you should use a single directory for the client and server code. By running the client and the server out of the same directory, you will not have to set up an HTTP or FTP server to provide the class files.

The following steps are required to build a system:
1. Write and compile Java code for interfaces
2. Write and compile Java code for implementation classes
3. Generate Stub and Skeleton class files from the implementation classes
4. Write Java code for a remote service host program
5. Develop Java code for RMI client program
6. Install and run RMI system

We will be using two examples to explain how to implement RMI. Dining Philosophers is implemented traditionally with command line execution. To the other example, Sleeping Barber, we have added a little twist using bat-files. This can be useful in Windows-environment.

3.1.1 Example case: Dining Philosophers

All the code to this example can be found at:
http://www.cs.helsinki.fi/u/kleemola/rio/GroupWork/philosopher/

The dining philosophers problem is a classic multi-process synchronization problem.

Five philosophers spend their time eating and thinking. They are around of a dining table with a large bowl of spaghetti in the centre of the table. There are five plates at the table and five forks set between the plates.

Each philosopher needs two forks to eat. The goal is of course to be able to eat in orderly manner.

RMI implementation of dining philosophers

Our aim is to attempt to solve the well known Dining Philosophers problem by utilizing Java's RMI technology.

This document will outline the basic structure of our solution. For additional technical implementation details please refer to the source code files: Forks.java (the interface) ForkServer.java (the RMI server), Philosophers.java (the client)

The structure of the solution is as the following: remote ForkServer issues and collects forks from the clients. The actual fork issuing algorithm is based upon Lehmann-Rabin's randomized solution to dining philosophers problem. http://www.springerlink.com/content/4h3kmxgr7umg7acr/. So a fork can be treated as reusable resource that can be held at most by one instance of Philosophers (our client).

Mutual exclusion

To ensure the mutual exclusion when checking or modifying the status of the fork we use simple Java semaphores, one semaphore per fork.

Freedom from deadlock

Deadlock happens when all the philosophers select a right hand fork or a left hand fork at the same time, and keep it. Following Lehmann-Rabin's solution, which ensures that if the philosopher doesn't get a second fork he returns both forks and tries again. Furthermore every time before a philosopher has to pick up the first fork, he tosses a coin with equal chances of selecting right hand or left hand fork. Our document will not go deeper into advanced proof of this elegant algorithm. Still, we refer you to the link http://www.springerlink.com/content/4h3kmxgr7umg7acr/ to get more information.

Freedom from starvation

Due to probabilistic nature of the algorithm the probability of starving approaches 0 as times passes by. Detailed discussion on probability part, see Lehmann-Rabin's proof.

Interface

1. The first step is to write and compile the Java code for the service interface. The Forks interface defines all of the remote features offered by the service:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Forks extends Remote {
      int getForks(int philNum) throws RemoteException;
      int returnForks(int philNum) throws RemoteException;
}
```

Notice this interface extends Remote, and each method signature declares that it may throw a RemoteException object.

Copy this file to your directory and compile it with the Java compiler:
>javac Forks.java

2. Implementation

Next, you write the implementation for the remote service. Notice that in this example this class works also as a server . This is the ForkServer class:

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.concurrent.Semaphore;

public class ForkServer implements Forks {

        // Access control semaphores for 5 forks
        static Semaphore[] s = { new Semaphore(1),
                                                        new Semaphore(1),
                                                        new Semaphore(1),
                                                        new Semaphore(1),
                                                        new Semaphore(1) };

        // Critical section of random()
        static Semaphore rs = new Semaphore(1);

        /* Array elements of forks show which philosopher holds the given fork */
        private int[] forks = { -1, -1, -1, -1, -1 };

        public int getForks(int philNum) {

                if (philNum < 0 || philNum > 4)
                        return -1; //wrong argument

                System.out.println("Philosopher " + philNum + " trying to get
                                        forks...");

                while(true) {
                        /* 1:    Throw a coin selecting which fork to get first */
                        try { // we don't want concurrent access to random number
                            // generator
                        //because of the risk of deadlock
                                rs.acquire();
                        } catch (InterruptedException e) {}

                        int rnd = (int)(Math.random()*2);

                        rs.release();
                        /* 2:    If taken GOTO 1 else try to take the other fork */
```

```
                int fork1 = (philNum + rnd)%5;

                // reserve right to access the data on the first selected fork
                try {
                        s[fork1].acquire();
                } catch (InterruptedException e) {}

                if (forks[fork1] == -1) {

                        int fork2 = (philNum + (1 - rnd))%5;

                        /* 3:    If the other fork taken GOTO 1 */
                        // reserve right to access the data on the second selected
                          fork
                        try {
                                s[fork2].acquire();
                        } catch (InterruptedException e) {}

                        if (forks[fork2] == -1) {
                                forks[fork1] = philNum;
                                forks[fork2] = philNum;

                                System.out.println("Philosopher " + philNum +
                                                " can start to eat using forks " +
                                                 fork1 + " and " +fork2 + "...");
                                s[fork1].release();
                                s[fork2].release();

                                return 0;
                        }

                        s[fork2].release();
                /* 4:    If taken go to 1 */
                }
                s[fork1].release();
        }
}

public int returnForks(int philNum) {
        if (philNum < 0 || philNum > 4)
                return -1; //wrong argument

        int fork1 = philNum;
        int fork2 = (philNum+1)%5;

        try {
                s[fork1].acquire();
                s[fork2].acquire();
        } catch (InterruptedException e) {}
```

```
            System.out.println("Philosopher " + philNum + " has returned the
                                    forks " +fork1 + " and " + fork2 + "...");

            forks[fork1] = -1;
            forks[fork2] = -1;

            s[fork1].release();
            s[fork2].release();

            return 0;
      }

      public static void main(String[] args) {

            try {
                  String name = "Forks";
                  Forks server = new ForkServer();
                  Forks stub =
                        (Forks) UnicastRemoteObject.exportObject(server, 0);
                  Registry registry = LocateRegistry.getRegistry();
                  registry.rebind(name, stub);
                  System.out.println("ForkServer bound");
            } catch (Exception e) {
                  System.err.println("ForkServer exception");
                  e.printStackTrace();
            }
      }
}
```

Again, copy this code into your directory and compile it.

The implementation class uses UnicastRemoteObject to link into the RMI system. In the example the implementation class directly extends UnicastRemoteObject. This is not a requirement. A class that does not extend UnicastRemoteObject may use its exportObject() method to be linked into RMI. When a class extends UnicastRemoteObject, it must provide a constructor that declares that it may throw a RemoteException object. When this constructor calls super(), it activates code in UnicastRemoteObject that performs the RMI linking and remote object initialization.

3. Stubs and Skeletons

You next use the RMI compiler, rmic, to generate the stub and skeleton files. The compiler runs on the remote service implementation class file.

Notice!

Note: With versions prior to Java Platform, Standard Edition 5.0, an additional step was required to build stub classes, by using the rmic compiler. However, this step is no longer necessary for 5.0 version.

>rmic ForksServer

Try this in your directory. After you run rmic you should find the file ForksServer_Stub.class.

After that we need the code for the client:

```
/***************************************************************
 * Client for RMI dining philosophers.
 * It creates and runs 5 thread-instances of Philosophers class,
 * representing 5 philosophers.
 * It is possible to implement 5 philosophers as separate clients,
 * but we opted for single client / single terminal solution
 * for the sake of simplicity.
 * First run rmiregistry and ForkServer, then this program
 * Supply the hostname to connect to when running this client
 * e.g. "java Philosophers localhost"
 ***************************************************************/

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Timer;
import java.util.TimerTask;

public class Philosophers extends Thread {

        //
        // Adjustable parameters (times in millisecs)
        //

        private static int timeThink = 3000;
        private static int timeEat = 1500;
        private static int timeRun = 60000;

        //
        // End parameters
        //

        protected static class PhilTimer extends TimerTask {

                // Stop the the threads and print out status info
                public void run() {

                        Philosophers.runThread = false;
```

```
                System.out.println("The philosophers are done");

                for (int i=0; i<=4; i++) {
                        System.out.println("Philosopher " + i + " has eaten " +
                                        Philosophers.numMeals[i] + " times");
                }
                System.exit(0);
        }
}

// Our 5 philosophers
private static Philosophers[] phils = new Philosophers[5];

// ID of this philosopher
private int philNum;

// Server stub
private static Forks frk;

//This is set to false by the timer causing the client threads to stop
private static boolean runThread = true;

//Number of meals each philosopher has had
private static int[] numMeals = { 0, 0, 0, 0, 0 };


public Philosophers(int num, Registry registry) {

        philNum = num;

        this.start();

}

public void run() {

        while(runThread) {

                try {
                        this.sleep((int)(Math.random()*timeThink)); //Think

                        frk.getForks(philNum);

                        numMeals[philNum]++; //Record the meal

                        this.sleep((int)(Math.random()*timeEat)); //Eat

                        frk.returnForks(philNum);

                } catch (Exception e) {
```

```
                        System.err.println("Philosophers exception");
                        e.printStackTrace();
                }
        }
}

public static void main(String[] args) {

        if (java.lang.reflect.Array.getLength(args) != 1) {
                System.out.println("Please supply the host to connect to as a
                                        command line argument.");
                System.out.println("for example:   java Philosophers
                                        localhost");
                return;
        }

        try {
                Registry registry = LocateRegistry.getRegistry(args[0]);

                // Get the server object from the registry
                frk = (Forks) registry.lookup("Forks");

                for (int i = 0; i <= 4; i++) {
                        new Philosophers(i, registry);
                }

                Timer philTimer = new Timer();

                //Prints the results and exits
                philTimer.schedule(new PhilTimer(), timeRun);


        } catch (Exception e) {
                System.err.println("Philosophers exception");
                e.printStackTrace();
        }
}
}
```

Running the RMI System

You are now ready to run the system! You need to start three consoles, one for the server, one for the client, and one for the RMIRegistry.

Start with the Registry. You must be in the directory that contains the classes you have written. From there, enter the following:
start rmiregistry (in Linux just use rmiregistry)

If all goes well, the registry will start running and you can switch to the next console.

In the second console start the server hosting the service, and enter the following:
>java ForksServer

It will start, load the implementation into memory and wait for a client connection.

In the last console, start the client program.
> java Philosophers localhost.

If all goes well you will see things happening in the server console!

3.1.2 Example case: Sleeping barber

All the code to this example can be found at:
http://www.cs.helsinki.fi/u/kleemola/rio/GroupWork/SleepingBarber/

In computer science, the sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes. The problem is analogous to that of keeping a barber working when there are customers, resting when there are none and doing so in an orderly manner. The barber and his customers represent aforementioned processes.

The analogy is based upon a hypothetical barber shop with one barber, one barber chair, and a number of chairs for waiting customers. When there are no customers, the barber sits in his chair and sleeps. As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs. If all of the chairs are occupied, the newly arrived customer simply leaves.

In our implementation the salon runs as an RMI-server process and the barber and the customer run as their own RMI-client processes. Actual implementation of barbering and getting haircut is implemented at the salon-server. For example in order to get a hair cut the customer just have to call salons wantHairCut method over RMI.

The implementation follows closely to the common solution using three semaphores: one for any waiting customers, one for the barber (to see if he is idle), and a mutex. When a customer arrives, he attempts to acquire the mutex, and waits until he has succeeded. The customer then checks to see if there is an empty chair for him (either one in the waiting room or the barber chair), and if none of these are empty, leaves. Otherwise the customer takes a seat – thus reducing the number available (a critical section). The customer then signals the barber to awaken through his semaphore, and the mutex is released to allow other customers (or the barber) the ability to acquire it. If the barber is not free, the customer then waits.

However in our example also a fourth semaphore is used to avoid infinitely fast haircutting. In other words the barber waits until the client signals (by using a semaphore) the barber that the haircut is ready and the barber can stop cutting his hair. In our example a static number of clients(7) are used and so after a finished haircut the customer waits outside the barber shop for random time between 0-2000 ms

growing his hair. After the hair has grown enough the customer re-enters the salon and tries to get a new haircut taking another 0-2000 ms not including the waiting time of course.

Notice that some Java versions don't have sufficient access rights by default so a custom security policy must be used in order to run the clients. This is done automatically at the batch files barber.bat and customer.bat. Examine the batch files by text editor in order to run the clients at a Linux machine.

This example is short version of the previous example. Only the classes with comments are provided and a short description of usage.

Save the source code to the same folder and compile them. All the classes are found after these instructions.

javac Customer.java

javac Barber.java

javac Salon.java

javac SalonInterf.

rmic Salon

Make the policy file. Write the following text to file and save it as everything.policy.

```
grant
{
      permission java.security.AllPermission;
};
```

Instructions for Windows:

1. Start RMI registry:

rmiregistry

2. Start salon (server)

-Save a file salon.bat with this text included:

@java -Djava.security.policy=everything.policy Salon

-Click salon.bat

salon.bat

3. Start customer and barber (clients)

- Save a file barber.bat with this text included:

-@java -Djava.security.policy=everything.policy Barber

-Click barber.bat

-Save a file barber.bat with this text included:

@java -Djava.security.policy=everything.policy Customer

-Click customer.bat

The code for Sleeping barber

```
//Customer class

import java.util.Random;
import java.rmi.Naming;

/**
 * This class implements multiple clients for the barber.
 * Amount of clients is static and each client is running
 * on a separate thread.
 */
public class Customer implements Runnable
{
	// personal id for each of the clients
	private int id;

	// the salon attached to the customer
	private SalonInterf salonInterf;

	public Customer(int id, SalonInterf salonInterf)
	{
		// set customer id
		this.id = id;

		// start the client
		this.salonInterf = salonInterf;
		new Thread(this).start();
	}

	public void run()
	{
		while(true)
		{
			System.out.println("Customer " + this.id + ":
					growing hair...");
```

```
                  try {
                        // random delay between 0-2000ms
                        int delay = (int)((new
Random()).nextDouble()*2000);
                        Thread.sleep(delay);

                        System.out.println("Customer " + this.id + ":
wants a haircut");

                        // wait for a haircut (if there is free
chairs at the salon)
                        this.salonInterf.wantHairCut(id);
                  }
                  catch(Exception e)
                  {
                        e.printStackTrace();
                  }
            }
      }

      public static void main(String[] argv)
      {
            try
            {
                  // connect to the remote salon-server
                  SalonInterf salon =
(SalonInterf)Naming.lookup("//localhost/SalonServer");

                  // attach 7 customers with the salon
                  for(int i=1; i<=7; i++)
                        new Customer(i, salon);
            }

            catch(Exception e)
            {
                  e.printStackTrace();
            }
      }

}
//End of customer class

// Barber class
import java.rmi.Naming;

/**
 * Barber client for the SalonServer
 * only one barber per salon supported
 */
public class Barber
{
      public static void main(String[] argv)
      {
            try
            {
                  // Connect the RMI-server at
RMI://localhost/SalonServer
                  SalonInterf salon =
(SalonInterf)Naming.lookup("//localhost/SalonServer");

                  while(true)
```

```
                        {
                                // Cut hair infinitely
                                salon.wantToCut();
                        }
                }

                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }
}
// End of Barber class

// Salon interface
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * This is the remote interface for the Salon-server
 */
public interface SalonInterf extends Remote
{
        public void wantToCut()             throws RemoteException;
        public void wantHairCut(int id)     throws RemoteException;
}
// End of Salon interface

// Salon class
import java.util.concurrent.Semaphore;
import java.util.Random;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

/**
 * This class implements the salon for the sleeping barber-solution
 */

public class Salon extends UnicastRemoteObject implements SalonInterf
{
        // total number of chairs, reserved or free, at the salon
        private int numChairs;

        // semaphore for signaling the barber that customers are
waiting
        private Semaphore customers;

        // semaphore for signaling a client that the barber is ready
        private Semaphore barber;

        // mutex for avoiding interleaving of critical sections
        private Semaphore mutex;

        // semaphore for making sure that the customer sits until his
haircut is ready
        private Semaphore cutting;

        // number of customers currently waiting at the salon
        private int waiting;
        private String objectName;
```

```
	public Salon(String s) throws RemoteException
	{
		super();
		this.objectName = s;
		this.numChairs = 5;
		this.waiting = 0;
		this.customers = new Semaphore(0);
		this.barber = new Semaphore(0);
		this.mutex = new Semaphore(1);
		this.cutting = new Semaphore(0);
	}

	// implementation of making a haircut
	public void wantToCut()
	{
		System.out.println("Barber: waiting for customer...");
		this.customers.acquireUninterruptibly();
		this.mutex.acquireUninterruptibly();
		this.waiting--;
		this.barber.release();
		this.mutex.release();
		System.out.println("Barber: cutting hair...");
		this.cutting.acquireUninterruptibly();
		System.out.println("Barber: finished cutting");
	}

	// implementation of taking a haircut
	public void wantHairCut(int id)
	{
		this.mutex.acquireUninterruptibly();

		if(this.waiting < this.numChairs)
		{
			waiting++;
			System.out.println("Customer " + id + ": waiting
for haircut...");
			this.customers.release();
			this.mutex.release();
			this.barber.acquireUninterruptibly();
			System.out.println("Customer " + id + ": getting
haircut...");
			try {
				// random delay between 0-2000ms
				int delay = (int)((new
Random()).nextDouble()*2000);
				Thread.sleep(delay);
			}
			catch(InterruptedException e)
			{ }
			System.out.println("Customer " + id + ": haircut
ready!");
			this.cutting.release();
		}

		else
		{
			System.out.println("Customer " + id + ": room
full!");
			this.mutex.release();
		}
	}
```

```
        public static void main(String argv[])
        {
                // obligatory definition of the RMI security manager
                RMISecurityManager sm = new RMISecurityManager();
                System.setSecurityManager(sm);
                try
                {
                        // start the salon-server
                        Salon salon = new Salon("salonServer");
                        Naming.rebind("//localhost/SalonServer", salon);
                       System.out.println("SalonServer bound in registry");
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }
}

// End of Salon class
```

Evaluation Part

The project-team was formed after the first exercise session. The group consisted of 4 members, and the topics chosen were RMI implementations and two assigned practical questions based on Chapter 4 and Chapter 8.

Before all the members reached an agreement on the specific topic of RMI implementation, there was an intense discussion on which topic should be picked up and it seemed that inside the group members had their own interests because of various backgrounds. The discussion lasted for nearly one week and finally all members arrived to a consensus about the final topic.

At the beginning stage, the whole task was not divided into small parts for each members. Every one was assigned the same work load to ponder on finding the solution to the project. Later in the programming part, due to the tight schedules and time limits, the group was divided into two - each subgroup is responsible for implementation of one problem. The plan was then changed as detailed information was discussed inside subgroup and after the completion, one coordinator collected all the jobs done by subgroups and integrated them into a whole one. Feedback scheme was adopted after the first integration was distributed to each member and based on the feedbacks the final version was produced.

As the RMI and distributed system computing are topics which came quite late in the class, and some members did not have experiences in distributed computing and even some not that familiar with Java, it took quite a lot of time in reading the tutorials and RMI introductions separately. Text book does not provide us much information related to RMI methods, so searching for the necessary materials was another challenging job which was done mostly by the project coordinator. Through the implementation of RMI on concurrent problems, there are still some parts we are not sure such as we do not find a real distributed method to solve those two classic problems, what we do is to adopt RMI and put two essential entities on different end systems – client/server. The underlying method to solve the concurrency is still basic

mechanisms like semaphores.

During the proceeding of team project, all kinds of methods have been adopted to facilitate communications and cooperation among team members. The Wiki bulletin board offered by Moodle was used most frequently - every one put there fresh ideas they come up with and ask for comments from others. Through personal contributing of each member and group sharing of information, the project went on well and everybody was well informed during the whole process. E-mail and personal chat via instant messaging were also used often, to exchange ideas in a faster and more personal way to settle some disagreements. This happened between both parties and to get some more detailed information on how the project is going on together with proposing technical consulting and obtaining supports. A face to face meeting was held every week after exercise class, all members were trying their best to come, in spite of difficulties such as working issues and personal affairs. The meeting was used to adjust the project plan and served as a check point for the project progress in the past week. As every member had their own schedule, the original plan has been modified many times in order to meet the personal needs as well as that of group as a whole. In general, the project was getting on smoothly and efficiently, although coming across hurdles here and there. Everybody was actively participating in the completion of the project though at times the work load was not evenly distributed. Towards the end of the project the altogether workload per person was compensated accordingly.

References:

http://java.sun.com/developer/onlineTraining/rmi/RMI.html
http://java.sun.com/docs/books/tutorial/rmi/index.html
http://www.dynamic-apps.com/tutorials/rmi_part1.jsp
http://fragments.turtlemeat.com/rmi.php
Qusay H. Mahmoud - Distributed Programming with Java
Concurrent Programming: The Java Programming Language, Oxford University Press 1998