95-702 Distributed Systems

Project 2

Assigned: Friday, February 1, 2013

Due: Monday, February 15, 2013

Principles

This project naturally leads to discussions concerning non-functional characteristics of distributed systems such as protocol reliability (via positive acknowledgement with retransmission), symmetric key encryption, unreliable networks, interoperability, marshaling and external data representation, naming, separation of concerns and design patterns (including the proxy design).

Review

In Project 1 we worked with J2EE servlets and Java Server Pages using the Glassfish web container.

In this project we will be working at a lower level. That is, we will not have the Glassfish runtime environment to rely on. You may, however, continue to use Netbeans for most of this work.

This project has two parts:

1. The first part exposes the student to both TCP and UDP sockets. In this part we are interested in exploring issues associated with protocol reliability over unreliable networks. External representation and marshaling is also explored.
2. The second part exposes the student to issues associated with remote method invocation. Code is provided that is scaffolding upon which the student builds a more significant application. The scaffolding includes a client side proxy and server side skeleton. Using that code as a model to work from, the student writes a simple registry and an object server and client. The program uses sockets but is careful to illustrate separation of concerns and the use of the proxy design pattern. The student must implement ideas from the Colouris text, such as remote object references and request reply messages.

Discussion of UDP and TCP

In the first part of this project you will work with both UDP and TCP over IP. UDP is a simpler protocol than TCP and therefore usually requires more work from the application programmer. TCP presents the application programmer with a stream abstraction and is busier behind the scenes. TCP, unlike UDP, tries its best to make sure packets are delivered to the recipient. The underlying network we are using may, on occasion, drop packets. So, how can TCP provide for reliable delivery of packets? TCP uses the fundamental principal of "positive acknowledgement with retransmission".

A simplified example of "positive acknowledgement with retransmission" looks like the following. In this scenario, no packets are lost. These notes are adapted from "Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture" by Douglas E. Comer

```
Sender                          Service
=====================================================

  Send packet 1
  Start timer
                                  Receive packet 1
                                  Send ACK for packet 1
  Receive ACK 1
  Cancel timer

  Send packet 2
  Start timer
                                  Receive packet 2
                                  Send ACK for packet 2
  Receive ACK 2
  Cancel timer
```

Here is an example where the first packet is lost.

```
Sender                          Service
=====================================================

  Send packet 1
  Start timer
                  Packet lost


  Time expires
  Send packet 1
  Start timer
                                  Receive packet 1
                                  Send ACK for packet 1

  Receive ACK 1
  Cancel timer

  Send packet 2
  Start timer
                                  Receive packet 2
                                  Send ACK for packet 2
  Receive ACK 2
  Cancel timer
```

Here is an example where the first ACK is lost.

```
Sender                          Service
=====================================================

  Send packet 1
  Start timer
                                  Receive packet 1
                                  Send ACK 1
                  ACK 1 lost

  Time expires
  Send packet 1
  Start timer
                                  Receive packet 1 a second time
                                  Send ACK for packet 1

  Receive ACK 1
  Cancel timer

  Send packet 2
  Start timer
                                  Receive packet 2
                                  Send ACK for packet 2
  Receive ACK 2
  Cancel timer
```

The acknowledgement may be replaced with the result of the service. Here is another example.

```
Sender                          Service
=====================================================

  Send packet 1
  Start timer
                                  Receive packet 1
                                  Send response
                 Response
                 Lost

  Time expires
  Send packet 1
  Start timer
                                  Receive packet 1 a second time
                                  Send response again

  Receive response
  Cancel timer

  Send packet 2
  Start timer
                                  Receive packet 2
                                  Send response for packet 2
  Receive response
  Cancel timer
```

Task 1A

In Figures 4.3 and 4.4 of the Coulouris text, two short UDPClient and UDPServer programs are presented. The code for these can also be found at:

```
http://www.cdk5.net/wp/extra-material/source-code-for-programs-in-the-book
```

Make modifications to the UDPClient and UDPServer programs so that the client may ask the server to perform simple integer arithmetic. You need to implement addition, subtraction, multiplication, exponentiation and division of integers. You may assume that you have a well behaved user and all input is correct. You may also assume that the user uses spaces to separate the command line arguments. As mathematical operators use '+', ' –','/', '^' for exponentiation and 'X' for multiplication (trying to use '*' presents many problems when used on a command line).

The execution of the client program will look like the following:

```
java UDPClient 2 ^ 3
8
```

In Netbeans, command line arguments can be set by choosing Run/Set Project Configuration/Customize.

UDPClient.java and UDPServer.java will be placed in a project called Project2Task1 and submitted to Blackboard.

Task 1B

Using the same UDP server that you wrote for Task 1A, write a new UDP client named UDPClientWithProxy.java that has a main routine that computes and displays the sum of the integers 1+2+3+...+100. The main routine of the client must be very clean and contain no socket level programming. All of the socket work will be done within a single method with the following signature:

```
public static int add(int x,int y);
```

The main routine will make 100 calls on the add method. The add method, however, will not perform any addition. Instead, it will send a UDP message to the server and receive the server's response. It will return that response as a simple int. This is a proxy design.

Complete the other methods on the client: sub, div, exp, and mul. Each will take two integers and send a request to the server. Each will return an integer as its return value.

UDPClientWithProxy.java will be placed in the same project (Project2Task1) as Task 1A.

Task 2

Build a new project called Project2Task2. Modify the UDPServer and create a new Java class called UDPServerThatIgnoresYou.java. Write the new server so that it randomly ignores 70% of requests. In other words, the new UDPServerThatIgnoresYou will contain code close to this:

```
// rnd is an object of the Random class

aSocket.receive(request);

if(rnd.nextInt(10) < 7) {
       System.out.println("Got request " +
            new String(request.getData())+
               " but ignoring it.");
       continue;
}
else {
       System.out.println("Got request" +
            new String(request.getData()));
       System.out.println("And making a reply");
}
```

Create a new client called UDPClientWithReliability.java. This new client is a modification of UDPClientWithProxy. After a request, it waits only 2 seconds for a reply. If the reply does not arrive after two seconds, the client tries again. It never gives up. The UDP receive will look something like this.

```
aSocket.setSoTimeout(2000);
aSocket.receive(reply);
```

See the above discussion on "positive acknowledgement with retransmission".

UDPClientWithReliability.java will have a main routine that computes and displays the sum of the integers 1+2+3+...+100. The main routine of the client must be very clean and contain no socket level programming. All of the socket work (and retry code) will be done within a single method with the following signature:

```
public static int add(int x,int y);
```

All files for this Task should be in a project named Project2Task2.

Task 3

In Figure 4.5 and 4.6 of the Coulouris text, two additional short programs are presented: TCPClient and TCPServer. You can also find these programs at:

```
http://www.cdk5.net/wp/extra-material/source-code-for-programs-in-the-book
```

The TCPClient program takes two string arguments: the first is a message to pass and the second is an IP address of the server (e.g. localhost). The server will echo back the message to the client.

Make modifications to the TCPClient and TCPServer programs to exchange ultra-secret messages between spies in the field and their boss in Intelligence Headquarters.

For example, when the client is run with the following arguments:

```
"spy on Sean Beggs" localhost
```

The "spy on Sean Beggs" message would be sent by TCPClient to TCPServer.

The TCPServer should return to TCPClient:

```
The plan to spy on Sean Beggs has been approved.
```

And finally TCPClient should print:

```
The reply from headquarters is: The plan to spy on Sean Beggs has
been approved.
```

Intelligence Headquarters is pretty incompetent, and they decide whether to approve or deny a plan based on a coin flip. (I.e. based on a random number generator which has equally likely chance of approval or rejection.) Therefore the response from the sever should always be of one of two forms:

```
"The plan to " +original-message+ "has been approved."
```

or

```
"The plan to " +original-message+ "has been rejected."
```

These are top secret plans, however, so Eve should not be able to intercept them. Therefore part of your marshaling and un-marshaling of the messages should be to encrypt and decrypt the messages using a Caesar Cipher. See Wikipedia. Our shift amount will be +3 – a symmetric key. In other words, the plaintext 'a' will be encrypted with the cipher text 'd'. Spaces and special symbols, like periods and commas, will be left unencrypted. The letter 'Z' will be encrypted as 'C'. Our cipher will only operate on letters ('a'..'z') and ('A'..'Z'). This style of encryption is further evidence of the incompetence of Intelligence Headquarters.

It is required that a proxy design be used. All of the socket level programming needs to be isolated. The encryption and decryption code also needs to be isolated.

Name this project Project2Task3. It will contain the files TCPServer.java, TCPClient.java, and CaesarCipher.java. Other files may be included as needed.

Task 4

Study the programs at the end of this document. These programs illustrate low level remote method invocation (RMI). That is, we are not yet working with Java RMI. We are implementing a simple RMI system using TCP sockets. If you study them closely, then everything that follows becomes easier.

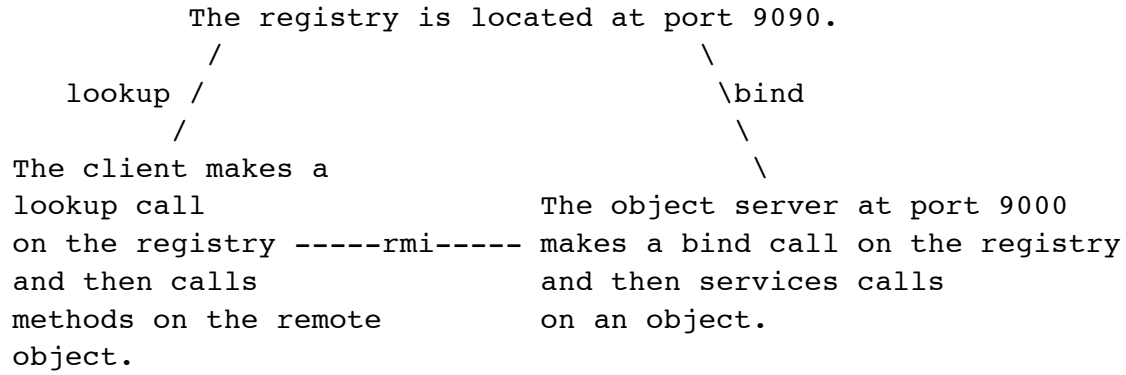
Program documentation is needed here. Be sure to add documentation to this working system so that it is clear to the reader that you know exactly what is going on.

Get these programs running. The terms "skeleton" and "stub" are used and you should understand exactly what kind of objects these terms refer to. Get the programs working in two separate Netbeans projects. Two separate projects works best (Project2Task4Client and Project2Task4Server) because it clearly separates the client code from the server code. The interface file (Person.java) must be placed on the server side as well as the client side. When the server and client are run, Netbeans will present two different console windows - one for the client and one for the server. There is a box on the console window that may be used to stop the server process.

Name these projects Project2Task4Client and Project2Task4Server. Submit these well- documented projects to Blackboard. The use of Javadoc is optional.

Task 5

This task asks you to build a distributed object server, a registry and a client.



The registry is used to illustrate an implementation of the brokerage design pattern.

Create three new Netbeans projects named Project2Task5Client, Project2Task5ObjectServer, and Project2Task5Registry.

a. Write a new Java class called RemoteObjectReference that implements Serializable. This class will encapsulate the private fields found in Figure 4.13 - a generic

representation of a remote object reference found in the Coulouris text. This class will have getter and setter methods for each field and a small main routine used as a test driver. The "interface of remote object" field will be defined as a String object. The IPAddress field is an array of 4 bytes. The other two fields are simple integers. This class will be present in Project2Task5Client, Project2Task5ObjectServer, and Project2Task5Registry projects.

b. Write a new Java class called RequestReplyMessage that implements Serializable. This class will encapsulate the private fields found in Figure 5.4 – a generic request reply message structure found in the Coulouris text. Note that a RequestReplyMessage object has a RemoteObjectRef member. This must be the same class as defined in the previous step. Setter and getter methods need to be defined.

Note that a RemoteObjectReference object contains enough information to specify a remote object. The RequestReplyMessage contains, in addition, information on the method that will be invoked (or was invoked) on that object as well as an array of bytes specifying the arguments to or return values from the method.

The RequestReplyMessage class will be present on the client and the object server only.

c. Write the following Java classes for the client project:

PersonClient.java
PersonClient creates a Binder_Stub object so that it may make lookup calls on the registry. It will call lookup("Mike") and receive a RemoteObjectReference object from the registry. PersonClient will then create a Person_Stub object with the RemoteObjectReference as a parameter to the Person_Stub constructor. It will then make calls on the stub to retrieve the name and ID in the remote object.

Binder_Stub.java
The client needs to speak to the registry.
This class implements the Binder interface and contains bind and lookup methods. The client only makes use of the lookup method.

Binder.java
This interface defines the lookup method as taking a string argument and returning a RemoteObjectReference value. It also defines the bind method that takes a string and a RemoteObjectReference as input. Its return type is void.

Person_Stub.java
The client needs to speak with the server. This class implements the Person interface and contains getID and getName methods. When making a call on the server, it creates a RequestReplyMessage from the information found in the RemoteObjectReference and sends this message to a TCP socket.

When receiving a reply, it reads a RequestReplyMessage from the socket, extracts bytes, and returns to the caller either the name or ID.

Person.java
This is the same interface as shown below.

RemoteObjectReference.java and RequestReplyMessage.java.
See above for a description of these classes.

d. Write the following Java classes for the registry project.

Binder_Servant.java
This servant object holds a mapping of names to RemoteObjectReference objects. This class implements the Binder interface. A Java TreeMap will be used to hold the mappings.

Binder.java
See above for a description.

Binder_Skeleton.java
This skeleton is used to communicate with the server and the client. The serve method of this class is written like the serve method in Person_Skeleton shown below. On bind calls, it reads String objects and remote object reference objects from sockets and makes calls on its Binder_Servant. On calls to lookup, it writes a RemoteObjectReference object to the socket. Nothing is written to the socket on a call to bind.

BinderServer.java
This class starts up the registry. It creates a Binder_Servant and passes that servant to its new Binder_Skeleton and calls serve.

RemoteObjectReference.java

This class is described above.

e. Write the following Java classes for the object server:

Binder_Stub.java
See above.

Binder.java
See above.

Person_Servant.java
This class implements Person and is shown below.

Person_Skeleton.java
This class has a server method that reads RequestReplyMessages and writes RequestReplyMessages. In between the reading and writing it makes a call on the Person_Servant object.

Person.java
See below.

PersonServer.java
This class creates a Person_Servant object and a RemoteObjectReference. It uses the Binder_Stub to make a bind call on the registry. It creates a Person_Skeleton object and asks it to serve.

RemoteObjectReference.java
See above.

RequestReplyMessage.java
See above.

You need not communicate with the registry using a request reply message structure. To simplify things, on a call to bind, simply make the call using two objects (a string and a remote object reference.)

Summary

Task Number	Project Names	Summary
1a & 1b	Project2Task1	Intro to UDP
2	Project2Task2	UDP Reliability
3	Project2Task3	TCP and Cryptography
4	Project2Task4Client Project2Task4Server	Proxy Design Pattern
5	Project2Task5Client Project2Task5ObjectServer Project2Task5Registry	Brokerage and Proxy design pattern

Code

```
// file: Person.java on both the client and server side
public interface Person {
  public int getID() throws Exception;
  public String getName() throws Exception;
}
```

```
// file: Person_Stub.java found only on the client side
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;

public class Person_Stub implements Person {
  Socket socket;
  ObjectOutputStream o;
  ObjectInputStream i;
  public Person_Stub() throws Exception {
  }

  public int getID() throws Exception {
     socket = new Socket("localhost",9000);
     o = new ObjectOutputStream(socket.getOutputStream());
     o.writeObject("ID");
     o.flush();
     i = new ObjectInputStream(socket.getInputStream());
     int ret = i.readInt();
     socket.close();
     return ret;
  }
```

```
  public String getName() throws Exception {
     socket = new Socket("localhost",9000);
     o = new ObjectOutputStream(socket.getOutputStream());
     o.writeObject("name");
     o.flush();
     i = new ObjectInputStream(socket.getInputStream());
     String ret = (String)(i.readObject());
     socket.close();
     return (String)ret;
  }
}
```

```
// file: PersonClient.java exists only on the client side
public class PersonClient {
    public static void main(String args[]) {
      try {
              Person p = new Person_Stub();
              int id = p.getID();
              System.out.println("ID = " + id);
              String name = p.getName();
              System.out.println(name + " has ID number" + id);
      }
      catch(Exception t) {
          t.printStackTrace();
          System.exit(0);
      }
    }
}
```

```
// file: Person_Skeleton.java exists only on the server side
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;
import java.net.ServerSocket;

public class Person_Skeleton {
  Person myServer;

  public Person_Skeleton(Person s) {
     myServer = s;
  }
```

```
  public void serve()   {
    try {
     ServerSocket s = new ServerSocket(9000);
     while(true) {
       Socket socket = s.accept();
       ObjectInputStream i = new ObjectInputStream(socket.getInputStream());
       String method = (String)i.readObject();
       if(method.equals("ID")) {
         int a = myServer.getID();
         ObjectOutputStream o = new
                                ObjectOutputStream(socket.getOutputStream());
         o.writeInt(a);
         o.flush();
       } else if(method.equals("name")) {
         String n = myServer.getName();
         ObjectOutputStream o = new
                                ObjectOutputStream(socket.getOutputStream());
         o.writeObject(n);
         o.flush();
       }
     }
    } catch(Exception t) {
      System.out.println("Error " + t);
      System.exit(0);
    }
  }
}
```

```
// file: Person_Servant.java exists only on the server side
public class Person_Servant implements Person {

  int id;
  String name;
  public Person_Servant(String n, int i) {
    name = n;
    id = i;
  }

  public int getID() {
    return id;
  }
  public String getName() {
    return name;
  }
}
```

```
// file: PersonServer.java exists only on the server side
public class PersonServer {
  public static void main(String args[]) {
    Person p = new Person_Servant("Mike",23);
    Person_Skeleton ps = new Person_Skeleton(p);
    ps.serve();
  }
}
```