Distributed Systems
Lab 1: Build your own middleware

Steven Latré, Jeroen Famaey and Tom De Schepper - University of Antwerp
(tom.deschepper@uantwerpen.be - M.G.215)

2015 – 2016

Group number:	1
Name student 1:	Armin Halilovic
Name student 2:	Josse Coen

1 Introduction

In this first lab session you will design and implement your own RMI-platform. As you can expect, this will not be a full-blown RMI implementation – this would lead us too far – but it should provide you with more insight into the basics of such a platform. This lab session contains four major tasks:

1. Creating an object to provide some straightforward functionality,
2. Designing and implementing the Server-side of the RMI platform,
3. Designing and implementing the Client-side of the RMI platform,
4. Creating an object to consume the functionality offered by the previous object and

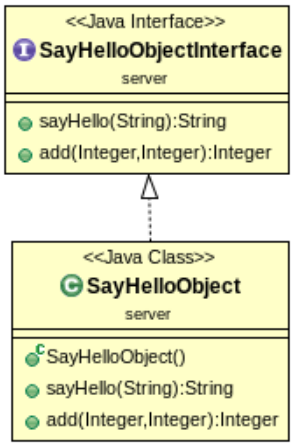
To complete these tasks, you will be guided by a number questions and implementation subtasks. The programming language chosen for this lab session is Java 7 and Eclipse is used as IDE.

2 Remote interface definition

The first task in this session is to create the object with a method which can be invoked by a client, and this by means of our to be developed RMI platform.

We shall start with a basic `HelloWorld` example, exposing one method. This `sayHello(String name)` procedure should take one input parameter (i.e. a name) and return a warm welcome to the client. Draw the complete UML Class Diagram of this server object below. Implement this in Eclipse as well.

Answer:



3 Server-side implementation of the RMI platform

3.1 Remote Reference Module

An important component of the server-side RMI platform is the Remote Reference Module. It manages the remote object references and translates between local and remote references, contains a remote object reference table and a table of local proxies. We will first create the `RemoteReferenceModule` class on the server side. Its most important function is to keep track of the server objects that have been registered in the platform. How will you uniquely identify a certain object in the system in each of the following cases:

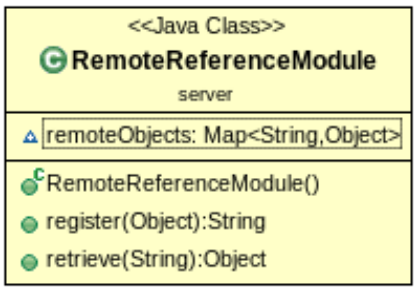
1. Only a single instantiation of a given class can be created (Singleton[1] pattern)
2. Multiple objects can be instantiated of a given class.

Which data structure will you use in your implementation and which methods need to be provided? Given the functionality that needs to be provided by the Remote Reference Module, methods will need to be defined for registering

[1]The Singleton pattern guarantees that only a single object of the Singleton class can be created. Additionally, it offers a global entry point towards that unique instance. More information can be found in Design Patterns: Elements of Reusable Object- Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the GangOfFour) ISBN 978-0201633610 , ISBN 0-201-63361-2 or on http://www.oodesign.com/singleton-pattern.html

objects and retrieving references towards registered objects. Draw the UML for both situations. However, you only need to implement in Eclipse the case for the Singleton pattern.

Answer:



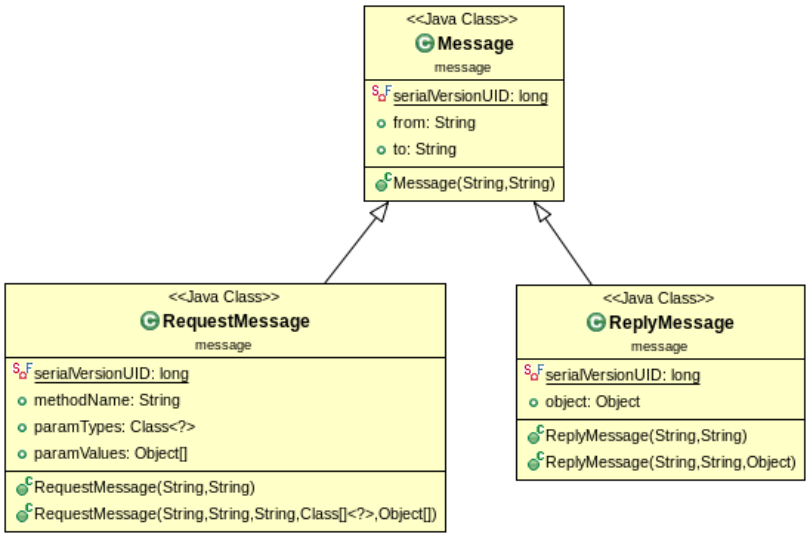
This structure supports both the Singleton pattern and multiple objects, the only difference lies in the implementation of the register method.

3.2 The Dispatching Module

A second important server-side component is the Dispatching Module. It allows clients to invoke methods of remote objects by identifying the appropriate method in the remote object interface and passing the request message to the object. For this to be possible, we need to define a representation – i.e. a message – we can use for communication between the client and the server. The request message should carry all necessary information to describe a method call by the client on a remote object. The response message should carry the information that is returned by the method call. Design your own communi-

cation message below, using a complete UML Class Diagram. Also implement these `Message` classes in Eclipse.

Answer:



As mentioned previously, the main task of the Dispatching Module is to identify the appropriate method to invoke in the remote object and to pass the request message to it. We will implement this in one specific procedure of the Dispatching Module, namely `dispatchCall()`. This procedure takes two arguments, i.e. a message as defined in the previous task and a reference towards the object to be invoked. The return value should again be a communication message that can be sent back to the client. Implement this `DispatchingModule` class. Briefly describe below how you will do this.

Answer:

dispatchCall gets the object's class by calling getClass on it. Then, using Class's getMethod method and the requested method's signature (the method's name and parameter types, which the client needs to specify in a RequestMessage, as can seen above), the requested method is captured in a Method object.
By calling the Method object's invoke method,
we can invoke the requested method on the server-side object and get its return value. dispatchCall returns the result in a ReplyMessage

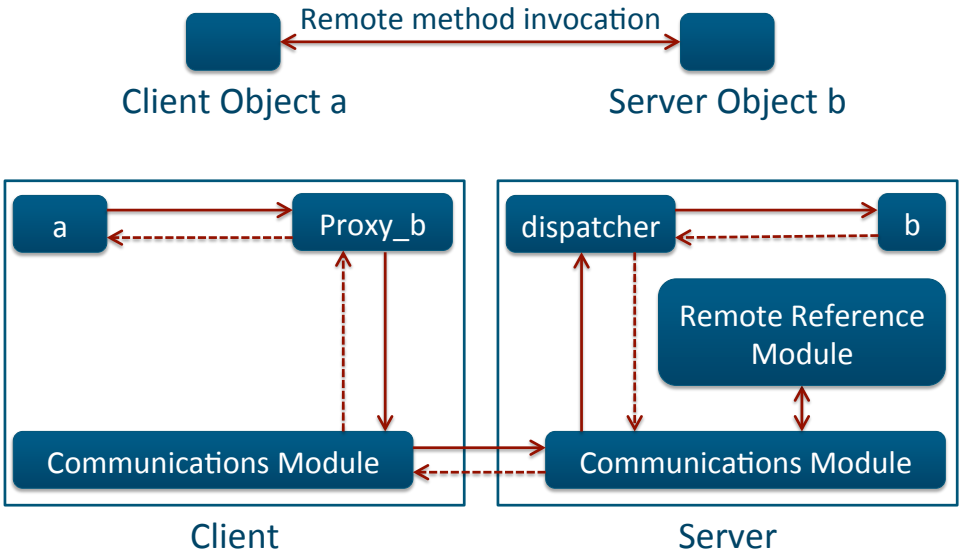


Figure 1: Schematic representation of the involved objects and modules for remote method invocation (client object a invokes server object b)

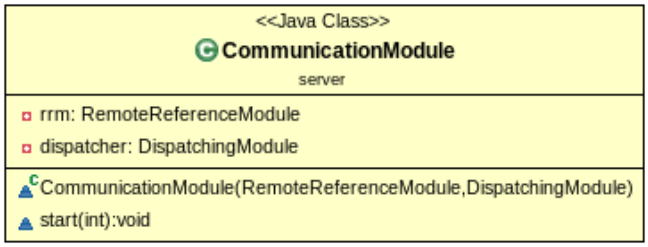
3.3 The Communication Module

3.3.1 The coordinator

The Communication Module runs a request-reply protocol and implements direct invocation semantics. As can be seen from Figure 1, the Communication Module needs to consult both the Remote Reference Module and the Dispatching Module when it receives a request from a client. Therefore, we will register both modules with the Communication Module. The Communication Module is also the responsible class for receiving incoming requests from clients. Therefore, we need some mechanism to allow connections from clients. We will use a `Socket` for this[2]. Implement the `start(int port)` method, which obviously starts a socket on a given port, but also provides the functionality for serving the incoming client. The details of how to serve the request will be completed in the following task. For now, create a separate Java class for the `CommunicationModule` and implement the necessary constructors, getters and setters, the `start(int port)` method and a placeholder for the implementation of the handling of an incoming request. Briefly describe below how you will implement the `start(int port)` method. You should keep in mind that this application is to be used in a multi-threaded setting, and accepting incoming client connection requests should be performed in a separate thread.

[2]More information on Socket programming can be found on Blackboard, or in the excellent tutorial on the Oracle website: `http://docs.oracle.com/javase/tutorial/networking/sockets/`

Answer:

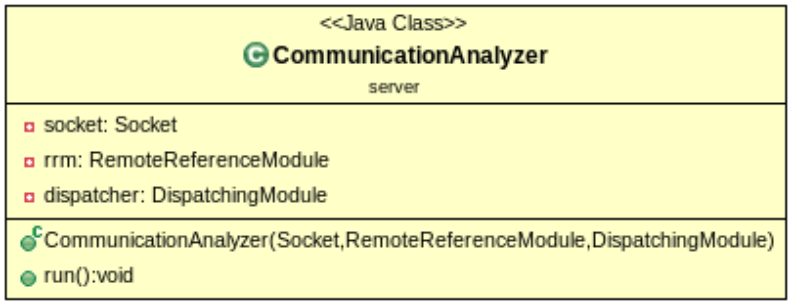


The start method will open a server socket that always listens to the given port. When a connection request arrives on that port, the method will create a CommunicationAnalyzer object that will handle the request in a different thread.

3.3.2 The Communication Analyser Thread

The Communication Analyser is responsible for reading and processing the incoming packets from the socket of a specific client and for delegating the incoming request towards the Dispatching Module. Subsequently, it will receive the return value of the invoked method on the local object, through the Dispatching Module, and will return this value to the client, using the same socket. Each Communication Analyser should run in its own thread, so that the server may handle requests from multiple clients simultaneously. Briefly describe you implementation below and implement your ideas in the `CommunicationAnalyser` class. Again, you only need to implement the case for the Singleton pattern. Keep in mind the `Message` types you created earlier in the lab session.

Answer:



The run method will use an ObjectSteam to send and receive messages to and from the client. It will receive RequestMessage objects, use the dispatcher to call the correct method based on the contents of the message, and send a ReplyMessage back to the client.

Now replace the placeholder in the `CommunicationModule` class with the code to create a `CommunicationAnalyser` and start its thread.

3.4 The Server

As a final task for the implementation of the server-side of our RMI-platform, you have to complete a `Server` class with a main method that creates, initialises and registers all previously implemented classes and finally makes sure that everything is booted. Describe the sequence of creation and invocation below.

Answer:

Initialize the server object (SayHelloObject), RemoteReferenceModule and DispatchingModule.
Register the server object into the RemoteReferenceModule.
Initialize the CommunicationModule with the RemoteReferenceModule and the DispatchingModule.
Start the CommunicationModule.

When a connection request arrives on the port given in the program arguments:
- Create a new socket to start a connection with the Client.
- Initialize the CommunicationAnalyzer.
- Start the CommunicationAnalyzer in a new thread.
- While the client is sending RequestMessages:
  - Read incoming RequestMessages.
  - Use the RemoteReferenceModule to find the object that Client needs.
  - Use the DispatchingModule to execute the requested command.
  - Send the result to the Client.

4 Client-side implementation

In the tasks from the previous section, you have created the server-side of the RMI platform. Your next task is now to design and implement the client-side, which will transparently handle and convert the invocations from an application on a client proxy object into remote procedure calls on the real server-side object.

Note: you may use a synchronous communication model between client and server. This means that (i) the client can wait on the server's response and (ii) you do not need to make a multi-threaded client.

4.1 The Communication Module

As stated earlier, the Communication Module runs a request-reply protocol and implements direct invocation semantics. As can be seen from Figure 1, a Communication Module also exists on the client side and needs to consult the Remote Reference Module before it can invoke methods on a server object. In our platform we will not implement the client side Remote Reference Module, because we assume beforehand that all objects are available at the same server. Therefore, we can create the remote reference in a straightforward manner.

The Communication Module communicates with the server-side implementation by means of a `Socket`. Therefore, we will implement a `createCommunicationModule(socket)` method in the `CommunicationModule` class, so that a

new instance of the Communication Module is created and can be used by the client to handle the invocations on the server-side object. Implement the initialisation of the Socket towards the server-side RMI platform you created earlier and the `CommunicationModule` class itself. The thread may stop execution after reading one response message from the server. Briefly describe below how you will accomplish this.

Answer:

The communication module is given an open socket through the constructor and gets the input and output streams from it: specifically, we use ObjectInputStream and ObjectOutputStream in this case.
The run method will send the request message to the server through the output stream, and receive and return the response.

4.2 The proxy mechanism

The client-side requires a local representation of the remote server-side object. This local representation will be invoked transparently, without the application fully realising that this object is hosted on a different server. That is, it knows that it is a remote object, because it invokes the methods from the interface, but is unaware of the exact details where and how the communication is handled.

To do this we will use the Java reflection mechanism. Java reflection allows invoking methods by creating classes based on arguments instead of directly invoking the methods. Java reflection defines a class called `Proxy` , which provides static methods for creating dynamic proxy classes and instances. It is also the superclass of all dynamic proxy classes created by those methods. A dynamic proxy class is a class that implements a list of interfaces specified at runtime when the class is created. In other words, at runtime, your code will define what the methods of the proxy class are actually doing. Each instance of such a proxy class has an associated invocation handler object, which implements the interface `InvocationHandler`. A method invocation on a proxy instance will be dispatched to the `invoke` method of the instance's invocation handler. More information about Java reflection can be found at `http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html`.

Now implement an additional method in the `Communica- tionModule`, which can convert the invocation on the `Proxy` object into `Call- Messages`

which can be transferred over the wire to the other computer, hosting the server-side of the RMI platform. Implement this method, with the following signature:

```
public Object remoteInvocation(Object proxy, Method method,
        Object[] args)
```

Describe in the text area below how you've accomplished this task.

Answer:

We use the communication module as the invocation handler. Thus: CommunicationModule implements the InvocationHandler interface. In the concrete implementation of invoke (inherited from InvocationHandler) remoteInvocation is called, where the request message is constructed and is sent to the server.

Your next task is to implement a new class, called the `ProxyLookup`, responsible for creating the `Proxy` object towards the server-side actual implementation. This class should have one method:

```
public static Object lookup(Class<?> cl,
        CommunicationModule communicationModule)
```

It should pass the correct `Proxy` object for the given class in the first argument. Moreover, the method returns the object returned by the remote method invocation. You will also need to create the appropriate invocation handler that redirects calls to the proxy's methods to your Communication Module. Describe below how you've implemented the Proxy creation procedure.

Answer:

The lookup method creates and returns a new proxy using Proxy's static method newProxyInstance, given the class loader, the class itself, and the invocation handler (which is the communication module).

4.3 The client

Finally, create a `Client` class with a main method that creates the Communication Module, connects with the server, creates a a Hello World proxy object and calls its `sayHello` method. List the sequence of object creations and method invocations below.

Answer:

Get host IP address from the program arguments.
Create socket to connect to the given IP address.
Create the communication module and pass the socket to it.
Create proxy using ProxyLookup's lookup method by passing
the interface of the remote object and the communication module.
Call methods on the proxy object, causing remote invocations.
Close the socket.

5 Testing your code

Finally, we've reached the stage where we can test the implementation. Remotely invoke your server-side object by means of your recently created lightweight RMI platform.

6 Submitting your solution

The deadline for submitting your solution is Sunday November 1, 2015 23:59.
To submit your solution do the following:

- Compress your code into one ZIP file, name it **ds-middleware-groupXX.zip** (where XX is group number) and send it to me by mail (at tom.deschepper@uantwerpen.be).
- You can scan this document with your answers and add it to the ZIP file.