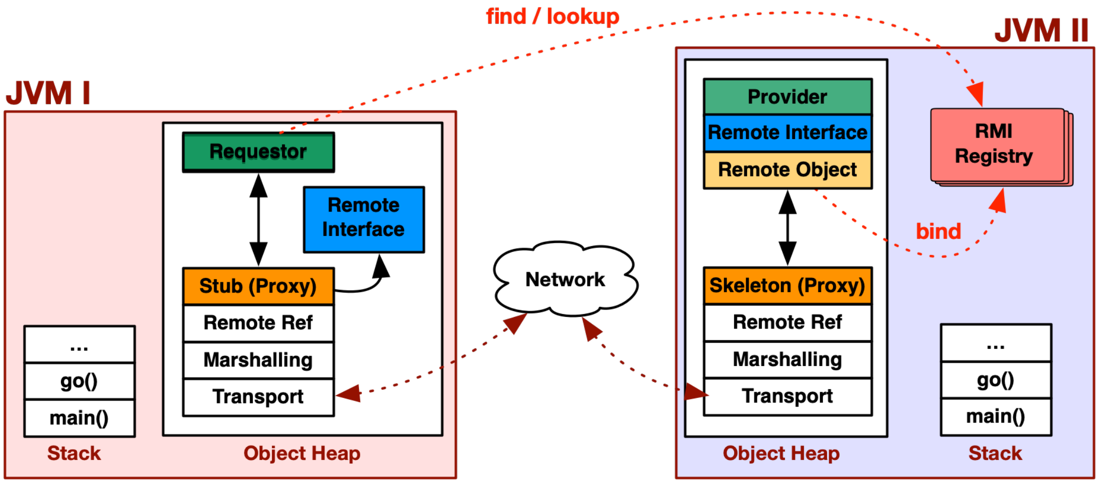
Department of Computer Science & Applied Physics

Distributed Computing with Proxies

Proxies play a major role in the implementation of many distributed computing frameworks, including Unicode-centric technologies such as RESTful services in JSON / XML and binary microservices such as the **Hessian** web service protocol and **Apache Thrift**. These technologies allow platform and language neutral inter-operation by marshalling service calls and responses in standardised formats with well described language bindings.



In this lab we will use the **Java Remote Method Invocation (RMI)** framework, which is part of the **java.rmi** module, to illustrate how proxies can be used in an inter-process communication (IPC) framework. Unlike the IPC frameworks mentioned above, all of which are **heterogeneous**, RMI is Java-centric and **homogeneous** and uses Java serialization to marshal and return method parameters and return types between JVMs.

One of the benefits of using a **homogeneous** IPC framework is that we can quickly and easily develop powerful distributed applications that are fully polymorphic. The exercises below demonstrate how an object or object graph can be passed around a distributed system **by-value** using serialization or **by-reference** using proxies. You should note the key concepts of **location transparency** and **local-remote transparency** that are promoted by the use of proxies.

Part 1: Configure and Execute the Pass-by-Value Example

- **Download the Zip archive** containing the source code for the project from the VLE under the heading "*Distributed Computing with Proxies (Source Code)*".

- Open Eclipse and create a new Java project called ***MessageService*** with a module called **atu.software**.

- Create a new package called **ie.atu.sw** and copy all the extracted source files into the package. In order to use the RMI framework in our module, you should add the following two declarations to *module-info.java*.

  requires java.rmi;

```
exports ie.atu.sw;
```

- The package **ie.atu.sw** should now contain the following compiled Java classes:

Class Name	Role
Message	A serializable POJO. This class will be **passed by value** as a serialized copy between the service requestor and the service provider.
MessageService	Represents the **remote interface or service interface** and exposes the public service methods that may be invoked by a remote object.
MessageServiceImpl	The **remote object or service provider** and the ultimate target for a remote method invocation. A **dynamic remote proxy called a stub** will be automatically generated from this class by the runtime environment and will control access to an instance of this class.
MessageServer	A servant class responsible for creating an instance of the remote object and **binding** it to a naming server with a human-readable name. When a service requestor asks the naming server for a remote handle on an object, the RMI registry will return an initialised instance of the remote interface in the form of a stub, which is really a dynamic remote proxy.
MessageClient	The **client or service requestor** of the remote message service. The client is loosely coupled with the remote object and does not know the actual class name of the service provider.

- Execute the class *MessageServer*. You should see the message *"[INFO] Server ready..."* in the console and then execute the *MessageClient* in a different console window. The line *System.out.println(message.**message**());* is the actual remote method invocation. You should see a message like *"Message from <your name>"*. The client invocation was dispatched through the stub proxy, over the network, to a skeleton proxy in a remote JVM and then delegated to the actual remote object. A serialized copy of the instance of *Message* was returned to the client. This is a remote **pass-by-value**.

- **Add the following lines** as the last statements in the *main()* method of *MessageClient*:

```
System.out.println(ms);
System.out.println(ms instanceof MessageService);
System.out.println(ms.getClass().getName());
```

  The output should be similar to the following:

```
Message from <your name>
Proxy[MessageService,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[127.0.0.1:51022](remote),objID:[-78525f59:1754ac2cfb2:-7fff, -
8941753759756732735]]]]]
true
com.sun.proxy.$Proxy0
```

  Explain the output.

Part 2: Refactor and Execute a Pass-by-Reference

- Open the class *Message* in Eclipse and then select ***Refactor → Extract Interface*** and name the interface *RemoteMessage*. Make sure to select the option "*Use extracted interface where possible*". Check that the classes *MessageService*, *MessageServiceImpl*, *MessageServer* and *MessageClient* are all using the newly extracted interface.

- Change the interface *RemoteMessage* as follows to declare it as a remote / service interface:

```
import java.rmi.*;
public interface RemoteMessage extends Remote{
        public String message() throws RemoteException;
}
```

- Create a new class called *RemoteMessageImpl* as follows.

```
import java.rmi.*;
import java.rmi.server.*;
public class RemoteMessageImpl extends UnicastRemoteObject implements RemoteMessage,
                                                                        Unreferenced {
  private static final long serialVersionUID = 1L;
  private Message message;
  private static int refCount = 0;
  private int objNum = 0;

  public RemoteMessageImpl(String message) throws RemoteException {
    this.message = new Message(message);
    objNum = refCount;
    System.out.println("[RemoteMessage] #References: " + refCount + ". Object#" + objNum);
    refCount++;
  }

  public String message() throws RemoteException {
    return message.message();
  }

  public int getReferenceCount() throws RemoteException {
    return refCount;
  }

  public int getObjectNumber() throws RemoteException {
    return objNum;
  }

  public void finalize() throws Throwable {
    System.out.println("[RemoteMessage Finalize called. About to be GCd.]");
  }

  public void unreferenced() {
    System.out.println("[RemoteMessage Unreferenced, Object No: " + objNum + "]");
  }
}
```

Notice that *RemoteMessageImpl* has an instance variable of type *Message*. Any calls to the remote method ***message()*** are delegated to this object variable. The class *RemoteMessageImpl* is an implementation of the *RemoteMessage* interface, i.e. it implements the remote method *public String message() throws RemoteException*. This class also implements the ***java.rmi.server.Unreferenced*** interface. This means that our *unreferenced()* method will be called when no other references exist to our remote object, i.e. when the **Distributed Garbage Collection (DGC)** reference counter reaches zero. You can think of the unreferenced method as the RMI equivalent of the *finalize()* method.

A remote object is considered to have an **active remote reference** to it if it has been accessed within a certain time period, called a lease period. If all remote references have been explicitly dropped, or if all remote references have expired leases, then a remote object is available for distributed garbage collection. The lease expiry interval can be specified on the command line using the property ***-Djava.rmi.dgc.leaseValue=10000***, i.e. a 10 second lease.

The remote interface *MessageService* now declares a single remote method *public RemoteMessage getMessage() throws RemoteException*. As we **want to simulate a pass-by-reference**, the return type, *RemoteMessage*, is really a reference to another server-side remote object.

- Open the class *MessageServer* in Eclipse and change the following statement:

  RemoteMessage msg = new Message(s);

Change to:

RemoteMessage msg = new RemoteMessageImpl(s);

- To execute the programme, select ***Run Configurations*** → ***MessageServer*** → ***Arguments*** and enter the following lease value parameter into the ***VM Arguments*** dialog: -Djava.rmi.dgc.leaseValue=10000. You should see the following output:

```
[RemoteMessage] #References: 0. Object#0
[INFO] Server ready...
```

- Execute the *MessageClient*. Add the statement ***System.out.println(message);*** as the last line in *main()* of *MessageClient* and then execute the class. You should see the following additional console output:

```
Proxy[RemoteMessage,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[127.0.0.1:51136](remote),objID:[-57638f0a:1754adc9d84:-7fff,
1287053005129618111]]]]]
```

Explain what the proxy object is doing in this case.

Instead of returning a serializable *Message* type, we now return an instance of a *RemoteMessage* type, i.e. we return a proxy stub for the class *RemoteMethodImp*. Thus, the return type of our remote method is itself a reference to a remote object. You should note that the *MessageServer* and *MessageServiceImpl* classes do not require specialised coding to implement this behaviour.