CT414

Distributed Systems & Co-Operative Computing

Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

2025–01–15

Contents

1 Introduction

1.1 Client-Server Architectures

1.1.1 Two-Tier Architectures

A **two-tier client-server architecture** is a client-server architecture wherein a client talks directly to a server, with no intervening server. It is typically used in small environments ($\lesssim$ 50 users).

A common development error is to prototype an application in a small, two-tier environment, and then scale up by simply adding more users to the server: this approach will usually result in an ineffective system, as the server becomes overwhelmed. To properly scale to hundreds or thousands of users, it is usually necessary to move to a three-tier architecture.

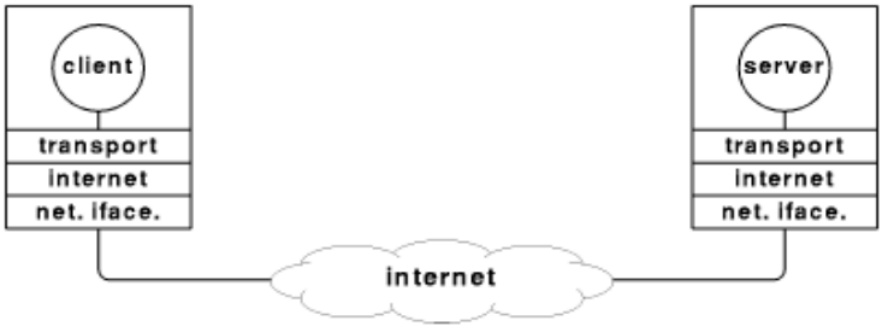

Figure 1: Client & server using TCP/IP protocols to communicate. Information can flow in either or both directions. The client & server can interact with a transport layer protocols.

1.2 Three-Tier Architecture

A **three-tier client-server architecture** introduces a server or **agent** (or **load-balancer**) between the client & the server. The agent has many roles:

- Translation services: such as adapting a legacy application on a mainframe to a client-server environment.
- Metering services: such as acting as a transaction monitor to limit the number of simultaneous requests to a given server.
- Intelligent agent services: as in mapping a request to a number of different servers, collating the results, and returning a single response to the client.

1.3 Network Programming Paradigms

Practically all network programming is based on a client-server model; the only real difference in paradigms is the **level** at which the programmer operates. The sockets API provides direct access to the available transport layer protocols. RPC is a higher-level abstraction that hides some of the lower-level complexities. Other approaches are also possible:

- Sockets are probably the best-known and most widely-used paradigm. However, problems of data incompatibility across platforms can arise.
- RPC libraries aim to solve some of the basic problems with sockets and provide a level of transport independence.
- Neither approach works very well with modern applications (Java RMI and other mdoern technologies, e.g., web services are better).

2 Java RMI

Remote Method Invocation (RMI) is a Java-based mechanism for distributed object computing. RMI enables the distribution of work to other Java objects residing in other processes or on other machines. The objects in one Java Virtual Machine (JVM) are allowed to seamlessly invoke methods on objects in a remote JVM. To call a method of a remote object, we must first get a reference to that object, which can be obtained from the registry name facility or by receiving the reference as an argument or return value of a method call. Clients can call a remote object in a server that itself is a client of another server. Parameters of method calls are passed as serialised objects:

- types are not truncated, and therefore, object-oriented polymorphism is supported;
- parameters are passed by value (deep copy) and therefore object behaviour can be passed.

The Java Object Model is still supported with distributed (remote) objects. A reference to a remote object can be passed to or returned from local & remote objects. Remote object references are passed by reference: therefore, the whole object is not always downloaded. Objects that implement the `Remote` interface are passed as a remote reference, while other objects are passed by value (using object serialisation).

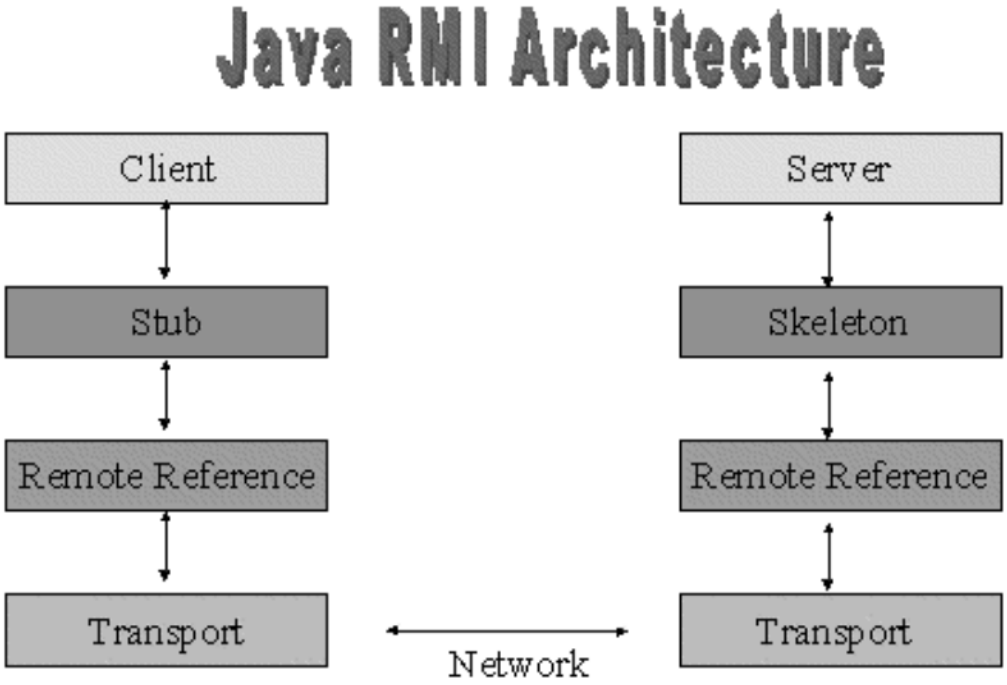

Figure 2: Java RMI Architecture

The client obtains a reference for a remote object by calling `Naming.lookup(//URL/registered_name)` which is a method which returns a reference to another remote object. Methods of the remote object may then be called by the client. This call is actually to the **stub** which represents the remote object. The stub packages the arguments (**marshalling**) into a data stream (to be sent across the network). On the implementation side, the skeleton unmarshals the argument, calls the method, marshals the return value, and sends it back. The stub unmarshals the return value and returns it to the caller. The RMI layer sits on top of the JVM and this allows it to use Java Garbage Collection of remote objects, Java Security (a security manager may be set for the server, now deprecated), and Java class loading.

2.1 Steps to Creating an RMI Application

1. Define the interfaces to your remote objects.
2. Implement the remote object classes.
3. Write the main client & server programs.

4. Create the stub & skeleton classes by running the *rmic* compiler on the remote implementation classes. (No longer needed in later Java versions).
5. Start the *rmiregistry* (if not already started).
6. Start the server application.
7. Start the client (which contains some initial object references).
8. The client application/applet may then call object methods in the remote (server) program.

2.2 Example Java RMI Program

```
// Remote Object has a single method that is passed
// the name of a country and returns the capital city.
import java.rmi.*;

public interface CityServer extends Remote
{
    String getCapital(String Country) throws
        RemoteException;
}
```

Listing 1: Example Java RMI Program

```
import java.rmi.*;
import java.rmi.server.*;

public class CityServerImpl
    extends UnicastRemoteObject
    implements CityServer
{
    // constructor is required in RMI
    CityServerImpl() throws RemoteException
    {
        super(); // call the parent constructor
    }

    // Remote method we are implementing!
    public String getCapital(String country) throws
        RemoteException
    {
        System.out.println("Sending return string now - country requested: " + country);
        if (country.toLowerCase().compareTo("usa") == 0)
            return "Washington";
        else if (country.toLowerCase().compareTo("ireland") == 0)
            return "Dublin";
        else if (country.toLowerCase().compareTo("france") == 0)
            return "Paris";
        return "Don't know that one!";
    }

    // main is required because the server is standalone
    public static void main(String args[])
```

```
    {
        try
        {
            // First reset our Security manager
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Security manager set");

            // Create an instance of the local object
            CityServerImpl cityServer = new CityServerImpl();
            System.out.println("Instance of City Server created");

            // Put the server object into the Registry
            Naming.rebind("Capitals", cityServer);
            System.out.println("Name rebind completed");
            System.out.println("Server ready for requests!");
        } catch(Exception exc)
        {
            System.out.println("Error in main - " + exc.toString());
        }
    }
}
```

Listing 2: Example Server Implementation

```
public class CityClient
{
    public static void main (String args[])
    {
        CityServer cities = (CityServer) Naming.lookup("//localhost/Capitals");
        try {
            String capital = cities.getCapital("USA");
            System.out.println(capital);
        } catch (Exception e) {}
    }
}
```

Listing 3: Example Client Implementation

No distributed system can mask communication failures: method semantics should include failure possibilities. Every RMI remote method must declare the exception `RemoteException` in its **`throw`** clause. This exception is thrown when method invocation or return fails. The Java compiler requires the failures to be handled.

When implementing a remote object, the implementation class usually extends the RMI class `UnicastRemoteObject`: this indicates that the implementation class is used to create a single (non-replicated) remote object that uses RMI's default sockets-based transport for communication. If you choose to extend a remote object from a non-remote class, you need to explicitly export the remote object by calling the method `UnicastRemoteObject.exportObject()`.

The main method of the service first needs to create & install a **security manager**, either the `RMISecurityManager` or one that you have defined yourself. A security manager needs to be running so that it can guarantee that the classes loaded do not perform "sensitive" operations. If no security manager is specified, no class loading for RMI classes is allowed, local or otherwise.

TO make classes available via a web server (or your classpath), copy them into your public HTML directory. Alternatively, you could have compiled your files directly into your public HTML directory:

```
javac -d ~/project_dir/public_html City*.java
rmic -d ~/project_dir/public_html  CityServerImpl
```

The files generated by `rmic` (in this case) are: `CityServerImpl_Stub.class` & `CityServerImpl_Skel.class`.

Polymorphic distributed computing is the ability to recognise (at runtime) the actual implementation type of a particular interface. We will use the example of a remote object that is used to computer arbitrary tasks:

- Client sends task object to compute server.
- Compute server runs task and returns result.
- RMI loads task code dynamically in the server.

This example shows polymorphism on the server, but it also works on the client, for example the server returns a particular interface implementation.