Remote Procedure Calls

Many distributed programming languages support Remote Procedure Calls (RPCs), where a call to the procedure \( P \) causes another process (often a server) to execute \( P \) as if in the caller’s address space.

RPCs provide
- higher level distribution than sockets
- architecture/OS neutral passing of simple & complex data types
- common distributed functionalities, e.g. name service, security etc.

RPC is supported by
- many languages, e.g. C, C++, ...
- many operating systems, e.g. Linux, Windows, ...

Remote Method Invocation

Distributed object-oriented languages like Java support Remote Method Invocation (RMI), analogous to RPC.

Java RMI supports
- Transparent interaction with objects on a remote JVM
- Callbacks from servers to clients
- reliability and safety of JVM

Java RMI uses a Distributed Object Model (JDOM), distinct from the single-location Object Model (JOM). More info in F24NU2 Network Technology and Security.
Using Java RMI

1: Server

A Java RMI server is created & registered as follows

1. Create an interface defining the methods to be implemented.
   - The interface must extend java.rmi.Remote, and
   - Every method must throw a java.rmi.RemoteException

For example, consider a broadcast ping utility.

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

2. Create a class that implements the new interface and is derived from java.rmi.server.UnicastRemoteObject

3. Create and install security manager for remote loading

4. Create instance(s) of the server object

5. Register the instance(s) with the registry using the Naming class. A registry is required to construct open systems, it enables new services to be registered, and clients to locate existing services.

```java
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
//2.
public class PingServer extends UnicastRemoteObject implements PingPong {
    // Constructor
    public PingServer() throws RemoteException {
    }
    public String doit() {
        String host;
        try {
            host = InetAddress.getLocalHost().getHostName();
        }
        catch(Exception e) { host = "unknown"; }
        return host;
    }

    public static void main (String[] argv) {
        //3. Create & install sec. manager
        System.setSecurityManager(new RMISecurityManager());
        try {
```
2: RMI Client

1. Looks up remote (server) object using registry
2. Casts remote object as the required server object
3. Can freely invoke methods of the remote object.

Building RMI Systems

1. Compile java source
   
   javac *.java

2. Run RMI stub generator rmic on the server implementation to generate
   
   • a stub PingServer_Sstub.class that acts as client-side proxy for remote object: forwarding RMI calls to remote skeleton and catching return values.
   
   • a skeleton PingServer_Skel.class that is server-side proxy for remote object, forwarding RMI calls to the local instance of the remote object & returning replies.

   rmic PingServer
3. Start RMI registry
kali% rmiregistry &

4. Start Server
kali% java PingServer &

5. Start Client at a different location
lambada% java PingClient ushas bartok ncc1705
pinging ushas.macs.hw.ac.uk ... at ushas time= 2ms
pinging bartok.macs.hw.ac.uk ... at bartok time= 3ms
pinging ncc1705.macs.hw.ac.uk ... at ncc1705 time= 2ms
lambada%

Java RMI Classification

Java RMI

- is location independent: the location of the remote object is not specified.
- can be used to build open systems: use naming class to register new services and to lookup existing services.
- used for small-scale distribution: every interacting program must be a Java program.

Java RMI Summary

Offers significantly higher-level distribution than sockets.

Distributed system built as multiple programs: at least a server and a client.

Once located remote objects are manipulated transparently, i.e. as if local.

Requires elaborate compilation: stub/skeleton generation, and launch: registry, server & client.