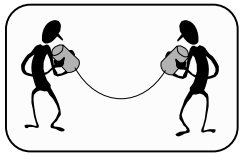
Object Interaction



- Object interaction vs RPC
- Java Remote Method Invocation (RMI)
- RMI Registry
- Security Manager

Introduction

- **Objective**
  To support interoperability and portability of distributed OO applications by provision of enabling technology

- **References**
- Latest Java documentation from http://java.sun.com/
  Java Remote Method Invocation Specification JDK 1.4
- Tutorials:
  - http://java.sun.com/docs/books/tutorial/rmi/index.html
  - http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html
- Couloris ch. 5, Boger ch. 4, Tanenbaum 2.3

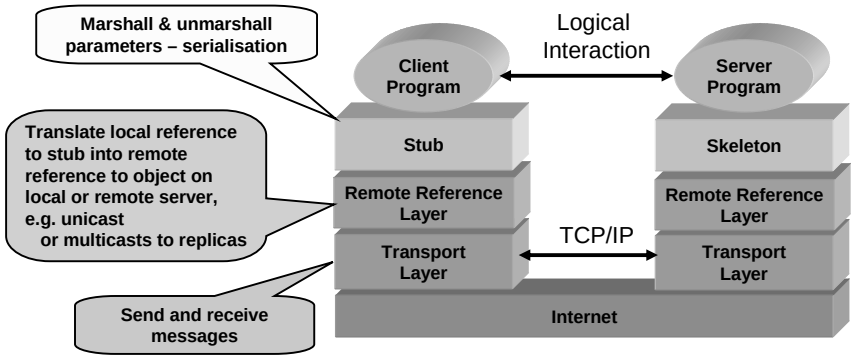
Object Interaction vs. RPCs

- Encapsulation via fine to medium grained objects
  (e.g. threads or C++ objects)
  Data and state only accessible via defined interface operations
  RPC based systems ➔ encapsulation via OS processes
- Portability of objects between platforms
  RPC clients and servers are not usually portable
- Typed interfaces
  Object references typed by interface ➔ bind time checking
  RPC interfaces often used in languages which do not support type checking
- Object can support multiple interfaces (depending on platform)
  RPC components have single interface
- Support for inheritance of interfaces
  Use inheritance to extend, evolve, specialise behaviour.
  New server objects with extended functionality (subtypes) can replace existing object and still be compatible with clients.
  RPC replacements must have identical interface
  ➔ usually no inheritance.

Object Interaction vs. RPCs (2)

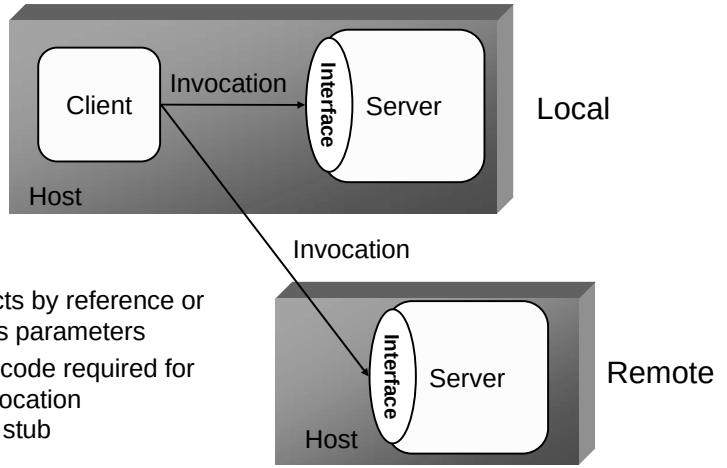
- Interaction Types
  Two-way synchronous invocation c.f. RPC – Java
- Pass objects as invocation parameters (Java only)
- Parameterised invocation exceptions ➔ Simpler error handling
- Location transparency
  Service use orthogonal to service location
- Access transparency
  Remote and co-located services accessed by same method invocation.
  RPC only used for remote access.
- Use invocations to create/destroy objects
  RPC systems (often) use OS calls to create/destroy processes

Java RMI Architecture



- See http://java.sun.com/docs/books/tutorial/rmi/index.html

Transparent Invocation



- Pass objects by reference or by value as parameters
- Download code required for remote invocation
  – dynamic stub

Java Interfaces

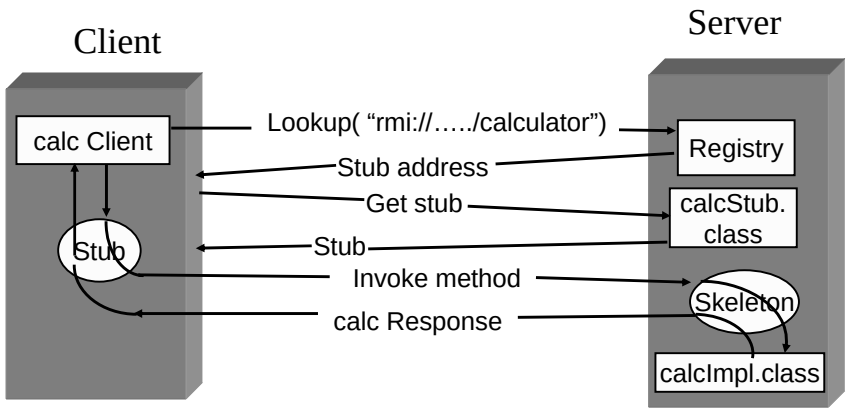
- Java is a *class-based* OO programming language
- Supports single inheritance
- A Java *interface* is essentially an abstract class and defines a new type
  - a collection of methods (and constant definitions)
  - methods and constants declared in an interface are implicitly public
- An interface may be derived from *one or more* further interfaces
- A class can implement *one or more* interfaces
  - as well as being derived from at most one other class

Remote Interface 1

- A type whose interfaces may be invoked remotely is defined as a remote interface
- A remote interface extends the **`java.rmi.Remote`** and must be public
- The methods of a remote interface must be defined to throw the exception **`java.rmi.RemoteException`** for comms failures

```
import java.rmi.*
public interface Calculator extends Remote {
    public long add(long a, long b)
        throws RemoteException;
    public long sub(long a, long b)
        throws RemoteException;
    public long mul(long a, long b)
        throws RemoteException;
    public long div(long a, long b)
        throws RemoteException;
}
```

Client Server Interaction



Note: skeleton not needed in later versions of Java

Remote Objects

- **Remote objects** are instances of classes that implement **remote interfaces** eg. **`CalculatorImpl`** implements **`Calculator`**
  Coulouris calls them **servants**
- A remote object class simply implements the methods defined in the remote interface
- Remote objects execute within a **server** which may contain multiple remote objects
- An object is implicitly exported if its class derives from **java.rmi.server.UnicastRemoteObject**
- Note operations invoked on remote objects, not on server containing them.

Remote Object Implementation

```
import javarmi.*
public class CalculatorImpl
    extends UnicastRemoteObject
    implements Calculator {


  public CalculatorImpl() throws RemoteException {
        super();
    }

    public long add(long a, long b) throws RemoteException {
        return a + b;
    }
    public long sub(long a, long b) throws RemoteException {
        return a - b;
    }
       . . .
  }
```

`UnicastRemoteObject` constructor exports the object as single server – not replicated

Call to super activates code in **`UnicastRemoteObject`** for RMI linking & object initialisation

Server Implementation

- A server program creates one or more remote objects as part of mainline code. For simple single object applications it is possible to combine server and object implementation.
- A server may advertise references to objects it hosts via the local RMI registry
- Registry allows a binding between a URL and an object reference to be made and subsequently queried by potential clients
- The server listens for incoming invocation requests which are dispatched to appropriate object.

- Note: there may be multiple servers and multiple clients within an application
- Client is not created within a server.

Server Mainline code

```
import java.rmi.Naming;
public class CalculatorServer {
  public static void main(String args[]) {
    if System.getSecurityManager() == null {
        System.setSecurityManager (new RMIsecurityManager ());
    }
    try {
      Calculator c = new CalculatorImpl();
      Naming.rebind("rmi://localhost/CalcService", c);
       }
      catch (Exception e) {
      System.out.println("Trouble: " + e);
    }
  }
}
```

Create security manager

Create server object

Register it with the local registry: URL-reference binding

Calculator Client Implementation

```
import java.rmi.*;
import Calculator;
public class CalculatorClient {

   public static void main(String[] args) {
     try {
          if System.getSecurityManager() == null {
          System.setSecurityManager (new RMIsecurityManager ());
           Calculator c = (Calculator) Naming.lookup(
             "rmi://remotehost/CalcService");
          System.out.println( c.sub(4, 3) );
          . . .   other calls ;
        } catch (RemoteException e) {
          System.out.println(" Exception:" + e);
   }
}
```

Get ref to CalcServer stub from remote registry

Invoke sub operation on remote calculator

RMIRegistry

- Must run on every server computer hosting remote objects
- Advertises availability of Server's remote objects
- name is a URL formatted string of form **`//host:port/name`**
  Both host and port are optional
- **`lookup (String name)`** called by a remote client. Returns remote object bound to **`name`**
- **`bind(String name Remote obj)`** Called by a server – binds **`name`** to remote object **`obj`**
  Exception if **`name`** exists
- **`rebind(String name Remote obj)`** binds **`name`** to object **`obj`** discards previous binding of **`name`** (safer than bind)
- **`String [] list()`** returns an array of strings of names in registry
- **`unbind(String name)`** removes a name from the registry

Using Registry

- Server remote object making itself available

```
Registry r = Locateregistry.getRegistry();
r.rebind ("myname", this)
```

- Remote client locating the remote object

```
Registry r =
  LocateRegistry.getRegistry("thehost.site.ac.uk");
RemObjInterface remobj =
  (RemObjInterface) r.lookup ("myname");
remobj.invokemethod ();
```

RMI Security Manager

- Single constructor with no arguments
  `System.setSecurityManager(new RMISecurityManager());`
- Needed in server and in client if stub is loaded from server
- Checks various operations performed by a stub to see whether they are allowed eg
  - Access to communications, files, link to dynamic libraries, control virtual machine, manipulate threads etc.
- In RMI applications, if no security manager is set, stubs and classes can only be loaded from local classpath – protect application from downloaded code

Parameter Passing

- Clients always refer to remote object via remote interface type not implementation class type
- A reference to a remote object can be passed as a parameter or returned as a result of any method invocation
- Remote objects passed by reference – stub for remote object is passed
- Given two references, r1 and r2, to a remote object (transmitted in different invocations):
  - r1 == r2 is false ➔ different stubs
  - r1.equals (r2) is true ➔ stubs for same remote object
- Parameters can be of any Java type that is serializable
  - primitive types, remote objects or objects implementing `java.io.serializable`
  - Non-remote objects can also be passed and returned by value i.e. a copy of the object is passed

  ➔ new object created for each invocation

Garbage Collection of Remote Objects

- RMI runtime system automatically deletes objects no longer referenced by a client
- When live reference enters Java VM, its reference count is incremented
- First reference sends “referenced” message to server
- After last reference discarded in client “unreferenced” message sent to server.
- Remote object removed when no more local or remote references exist.
- Network partition may result in server discarding object when still referenced by client, as it thinks client crashed

Dynamic Invocation

- Single method interface
- Invocation identifies method to be called + parameters
- User programs marshalls/demarshalls parameters
- Optional invocation primitive for object environments such as CORBA and for Web services.

```
public byte[] doOperation (RemoteObjectRef o,
            int methodId, byte[] arguments)
```

  - Sends a request message to the remote object and returns the reply.
  - The arguments specify the remote object, the method to be invoked and the arguments of that method.
  - Server has to decode request and call method

Summary

- RMI provides access transparency, object oriented concepts for IDL specification, object invocations and portability.
- Inheritance supports reuse ➔ high level programming concepts
- High implementation overheads due to
  - Byte code interpretation in Java
  - Marshalling/Demarshalling of parameters
  - Data copying
  - Memory management for buffers etc.
  - Demultiplexing and operation dispatching