Distributed Objects and Components

From **Coulouris, Dollimore, Kindberg and Blair**

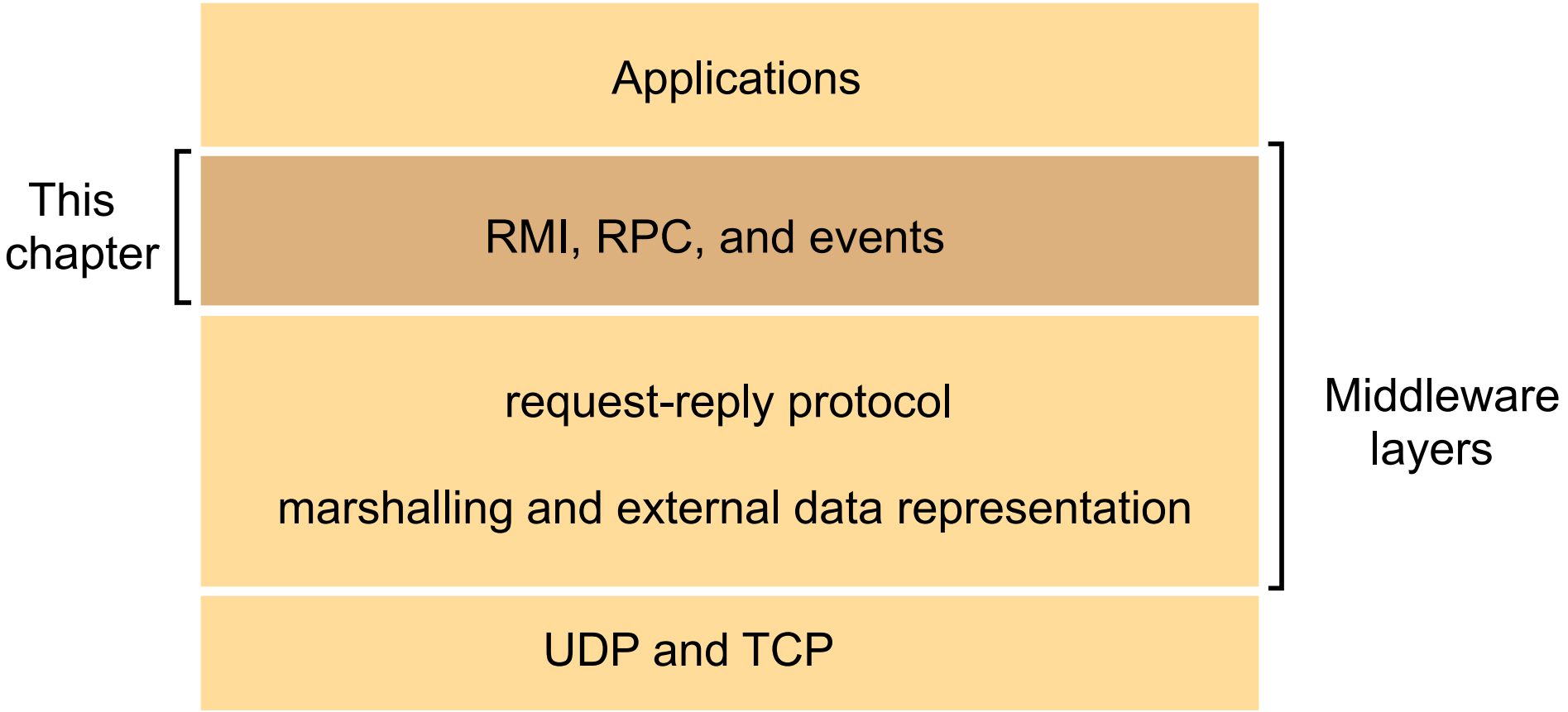
Distributed Systems:
Concepts and Design

Edition 5, © Addison-Wesley 2012

Distributed object middleware

- The key characteristic of distributed objects is that they allow you to adopt an object-oriented programming model for the development of distributed systems
- Communicating entities are represented by objects which communicate mainly using remote method invocation
  - The encapsulation
  - Data abstraction (programmers deal in terms of interfaces and not with implementation details)
  - More dynamic and extensible solutions
- Component-based middleware solves several limitations
  - Implicit dependencies, Programming complexity, Lack of separation of distribution concerns, No support for deployment

OUTLINE

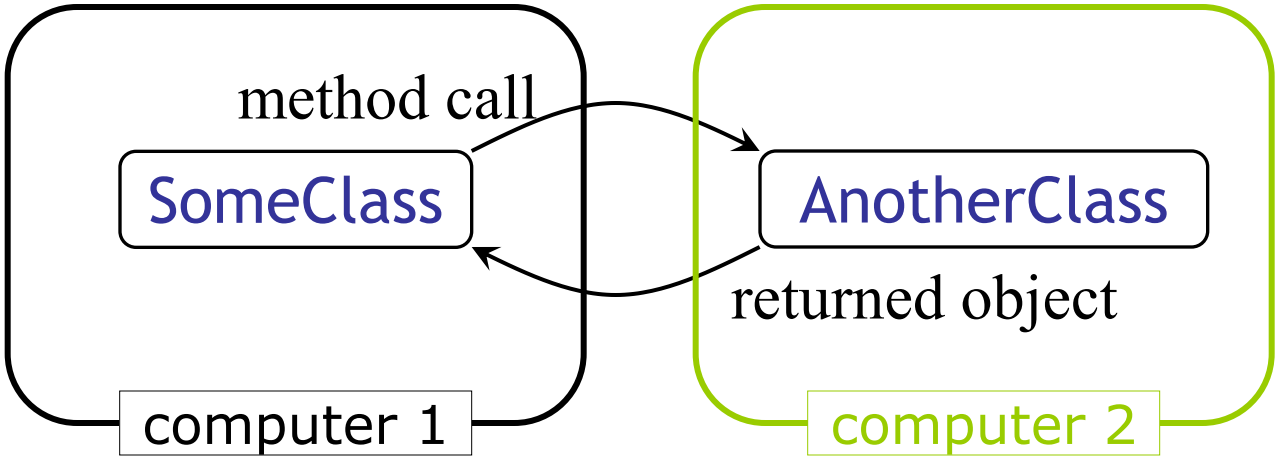

Applications
This chapter
RMI, RPC, and events
request-reply protocol
marshalling and external data representation
Middleware layers
UDP and TCP

What's in a Name?

Distributed Objects:

Software **modules** (objects) that are designed to **work together** but reside in **multiple computer systems** throughout an organization. A program in one machine **sends a message** to an object in a **remote** machine to perform some processing. The results are sent back to the **local** machine.

“The Network is the Computer”



What exactly do we mean by “objects”?

Objects are units of data with the following properties:

- ***typed and self-contained***
  - Each object is an instance of a *type* that defines a set of *methods* (signatures) that can be invoked to operate on the object.

- ***encapsulated***
  - The only way to operate on an object is through its methods; the internal representation/implementation is hidden from view.

- ***dynamically allocated/destroyed***
  - Objects are created as needed and destroyed when no longer needed, i.e., they exist outside of any program scope.

- ***uniquely referenced***
  - Each object is uniquely identified during its existence by a name/OID/reference/pointer that can be held/passed/stored/shared.

Distributed objects

Objects	*Distributed objects*	*Description of distributed object*
Object references	Remote object references	Globally unique reference for a distributed object; may be passed as a parameter.
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure.
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

Why are objects useful?

The properties of objects make them useful as a basis for defining persistence, protection, and distribution.

- **Objects are self-contained and independent.**
- Objects are a useful granularity for persistence, caching, location, replication, and/or access control.

- **Objects are self-describing.**
- Object methods are dynamically bound, so programs can import and operate on objects found in shared or persistent storage.

- **Objects are abstract and encapsulated.**
- It is easy to control object access by verifying that all clients invoke the object's methods through a legal reference.
- Invocation is syntactically and semantically independent of an object's location or implementation.

Issues Vital to Distributed Object Systems

1. Can we use distributed objects as a basis for **interoperability** among software modules written in **different languages**?

Issues Vital to Distributed Object Systems

2. Can objects interact across systems with different data formats?

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <meta http-equiv="X-UA-Compatible" content="ie=edge">
7       <title>Document</title>
8   </head>
9   <body>
10
11  </body>
12  </html>
```

Issues Vital to Distributed Object Systems

3. How can we **recover object state** after failures?

https://www.youtube.com/watch?v=vhuYpnqFNoA

Example: Emerald

Emerald is a classic and influential distributed object system.

Distribution is fully integrated into the language, its implementation, and even its type model.

- This is a strength and a weakness: combines language issues and system issues that should be separated.

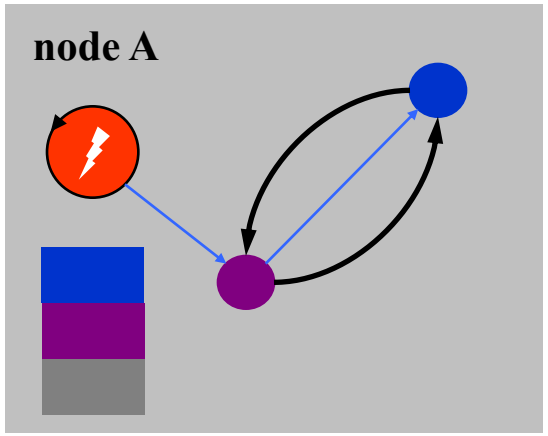
Objects can be freely moved around the network

- Programmers see a uniform view of local and remote objects.
- Moving objects “take their code and threads with them”.

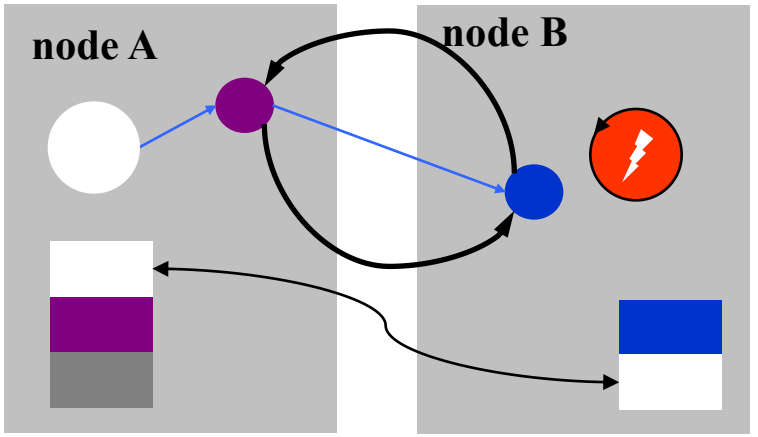
Local invocation is fast; remote invocation is transparent.

- supports pass-by-reference for RPC

Uniform Mobility: an Example



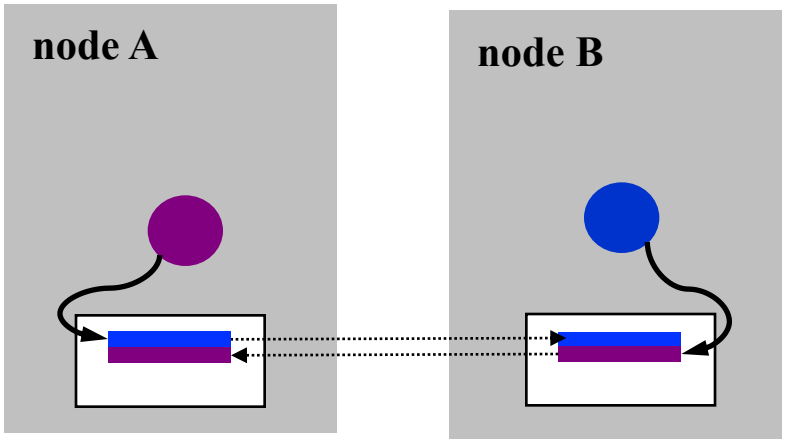
Step 1: a thread invokes a purple object on node A, which recursively invokes a blue object on the same node.



Step 2: the blue object moves to node B concurrently with the invocation.

How to preserve inter-object pointers across migration?
How to keep threads “sticky” with migrating objects?
How to maintain references in stack activation records?
How to maintain linkages among activation records?

Object References in Emerald



Emerald represents inter-object references as pointers into an *object descriptor* in an *object table* hashed by a unique *object identifier* (OID).

The object table has a descriptor for every resident object, and for every remote object referenced by a resident object, and then some.

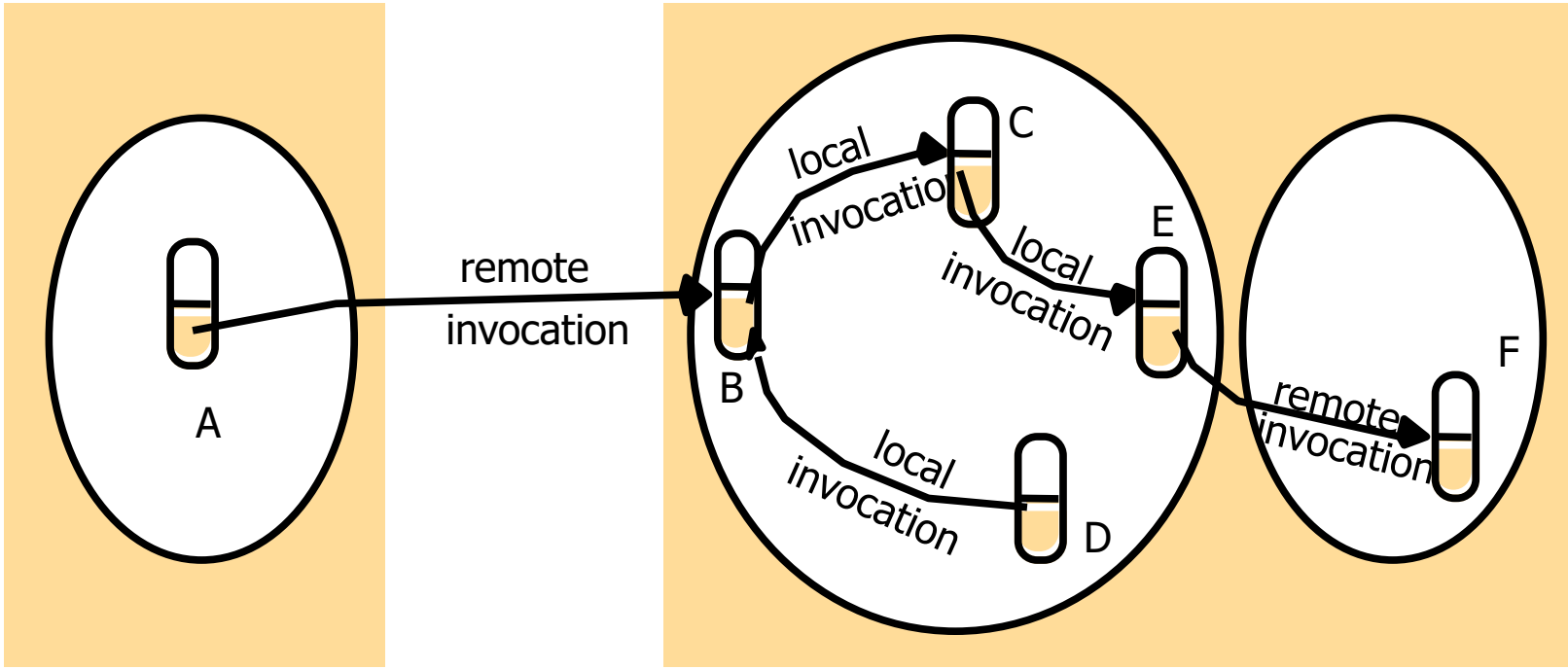
When an object moves, its containing references must be found (using its template) and updated to point to descriptors on the destination node.

References *to* the moving object need not be updated because they indirect through the object table.

Distributed Objects Model

- Distributed object systems adopt the client-server architecture:
  - Objects are managed by servers and their clients invoke their methods using remote method invocation.
  - The client's RMI request is sent in a message to the server managing the invoked method object.
  - The method of the object at the server is executed and its result is returned to the client in another message.
  - Objects in servers are allowed to become clients of objects in other servers.
- Distributed objects can be replicated and migrated to obtain the benefits of fault tolerance, enhanced performance and availability.
- Having client and server objects in different processes enforces encapsulation due to concurrency and heterogeneity of RMI calls.

Distributed Objects Model



Remote and local method invocations

Distributed Objects Model

- Each process contains a collection of objects, some of which can receive both local and remote invocations and others can receive only local invocations.
- Objects that can receive remote invocations are called *remote objects*.
- Other objects need to know the *remote object reference* in another process in order to invoke its methods.
  - A remote object reference is an identifier that can be used through a distributed system to refer to a particular unique remote object.
- Every remote object has a *remote interface* to specify which of its methods can be invoked remotely.
  - Objects in other processes can invoke only the methods that belong to the remote interface of the remote object.
  - Local objects can invoke the methods in the remote interface as well as other methods of the remote object.

Distributed Object Technologies

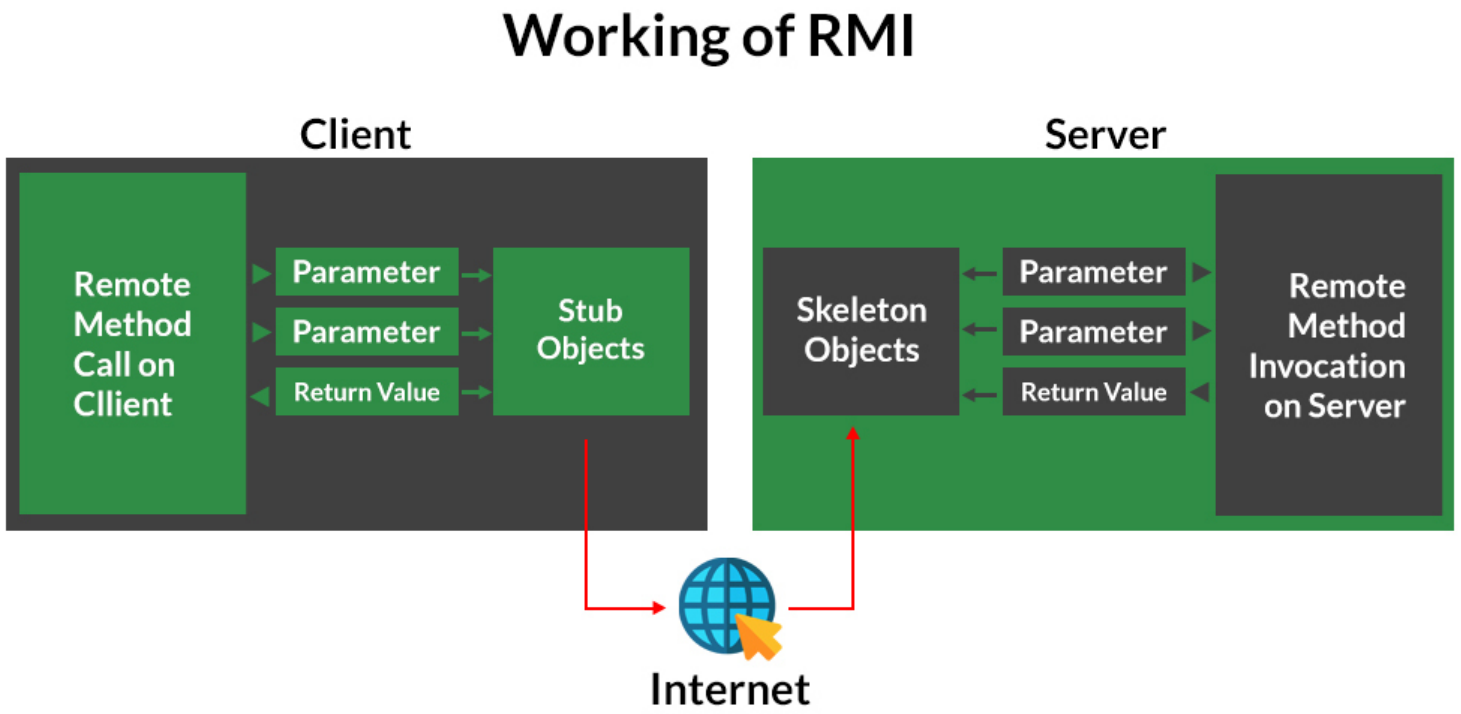
1. Remote Method Invocation (RMI)
   - API and architecture for distributed Java objects
2. Microsoft Component Object Model (COM/DCOM)
   - binary standard for distributed objects for Windows platforms
3. CORBA (Common Object Request Broker Architecture)
   - platform and location transparency for sharing well-defined objects across a distributed computing platform
4. Enterprise Java Beans (EJB)
   - CORBA-compliant distributed objects for Java, built using RMI
5. Web services and SOAP
   - protocols and standards used for exchanging data between applications

Example: RMI and Network Objects

Our goal now is to look at some current distributed object systems.

We start with systems that preserve the single-language model of Emerald, with uniform garbage collection:

- RMI for Java

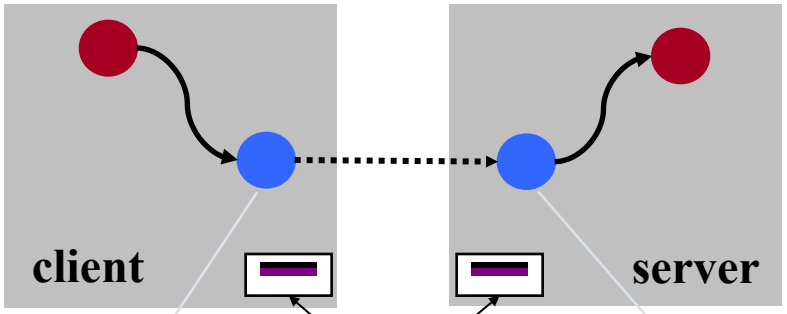


Stub/Surrogate Objects

Remote objects are referenced through *proxy* or *surrogate* objects, which "masquerade" as the actual remote object.

[SOS system, Marc Shapiro, The Proxy Principle (1986)]

Proxy objects are type-equivalent with their remote objects, but their methods are marshaling stubs.



Skeletons/guards may perform access checks as well as marshaling and method dispatch.

stub, surrogate, or *proxy*

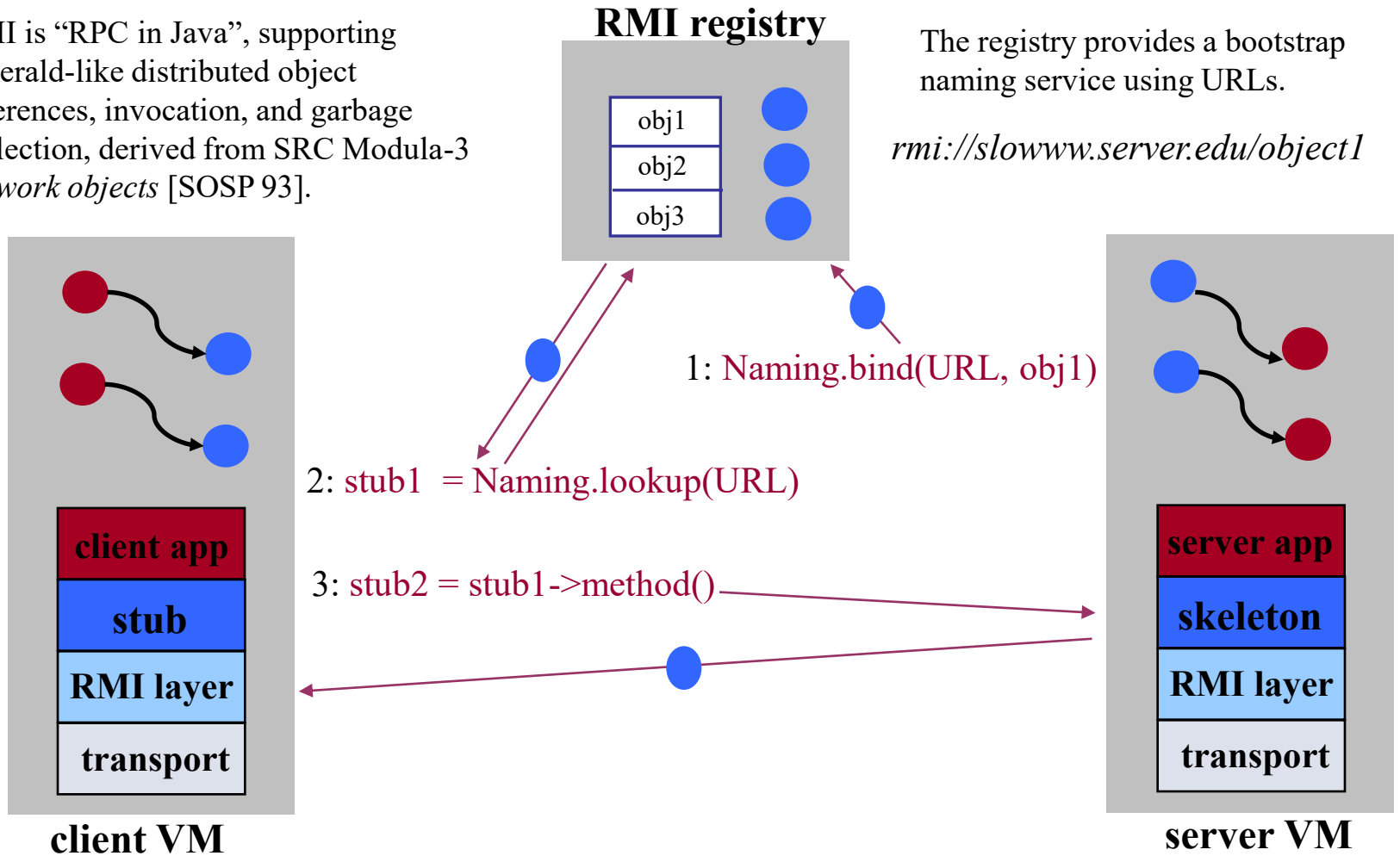
Per-process object tables hash stubs and skeletons by external OID (passed on the wire).

skeleton or *guard*

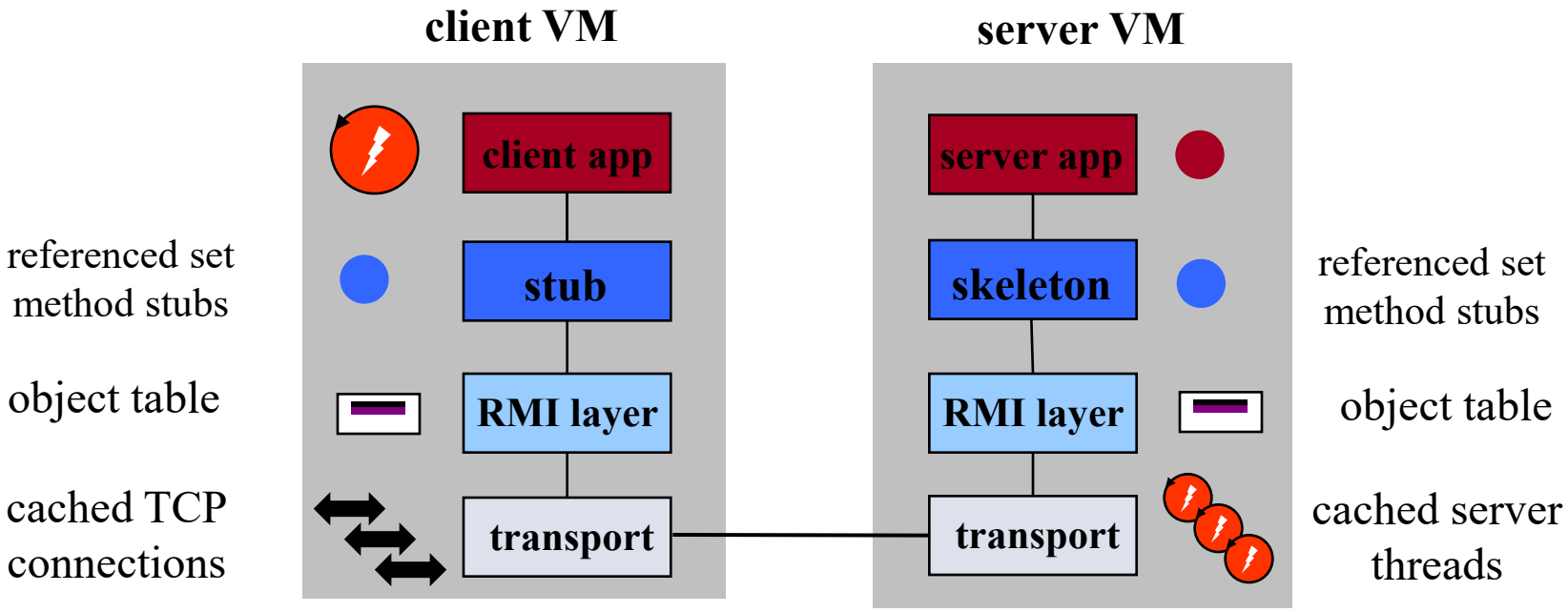
Proxy/stub objects can enscapsulate caching, replication, or other aspects of distribution that are best kept hidden from the client (also cf. *subcontracts* [Hamilton et. al., SOSP 93]).

Remote Method Invocation (RMI)

RMI is "RPC in Java", supporting Emerald-like distributed object references, invocation, and garbage collection, derived from SRC Modula-3 *network objects* [SOSP 93].



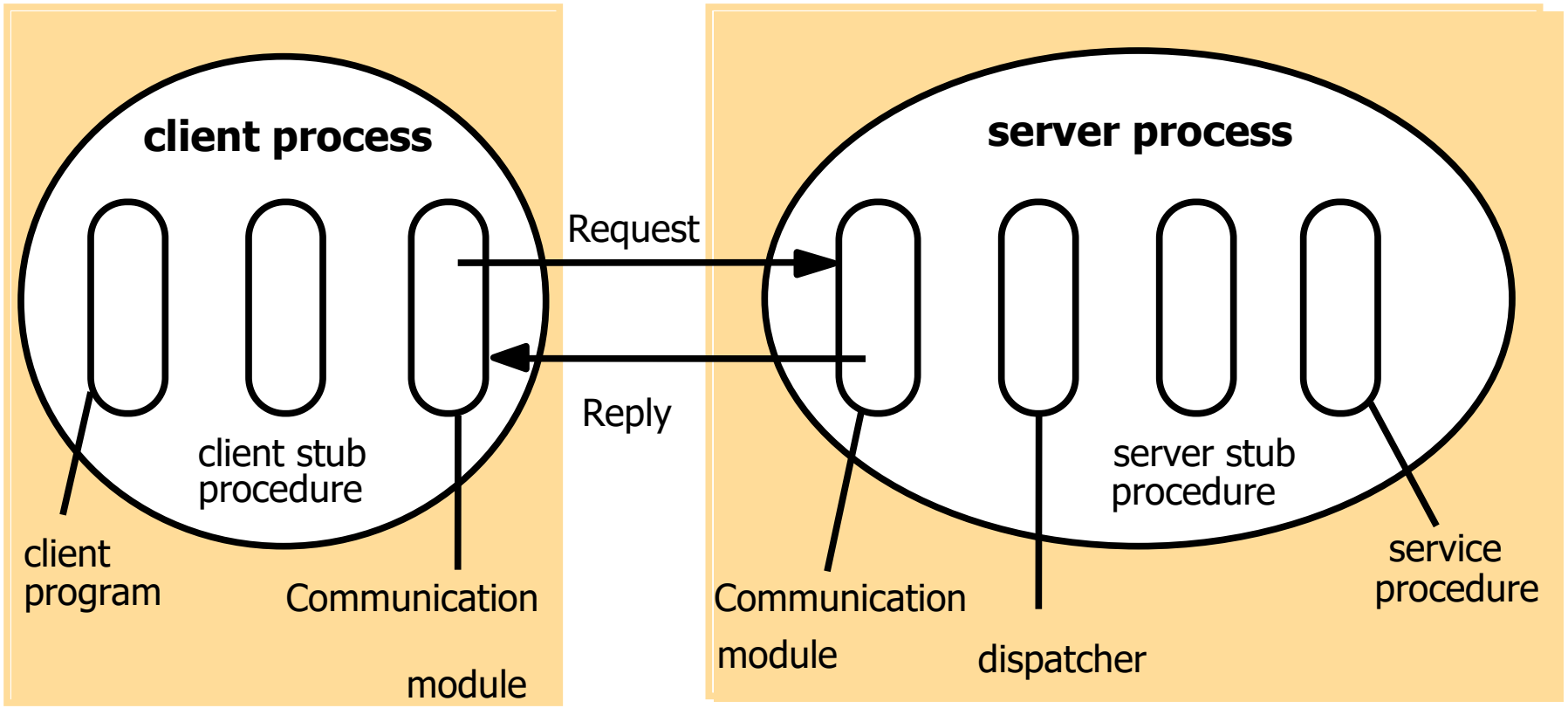
The RMI Stack



Remote Procedure Call (RPC)

- Very similar to a remote method invocation
  - A client process calls a procedure in a server process.
  - Servers may be clients of other servers to allow chains of RPCs.
  - Implemented to have one of the choices of invocation semantics.
  - Implemented over a request-reply protocol .
- The contents of request and reply messages are the same as RMI except that the object reference field is omitted.
- The supported software is similar except that no remote reference modules are required and client proxies and server skeletons are replaced by client and server *stub* procedures.
- The *service interface* of the server defines the procedures that are available for calling remotely.
- The client and server stub procedures and the dispatcher are generated by an interface compiler from the definition of the service interface.

Remote Procedure Call (RPC)

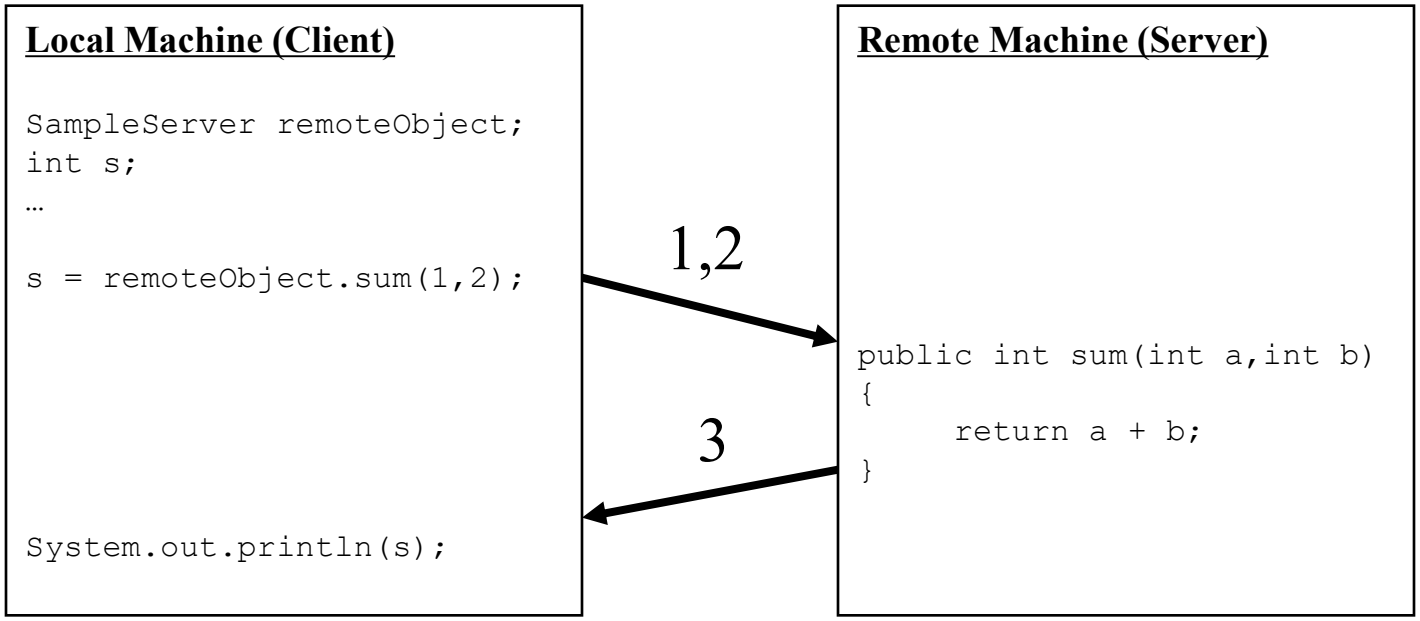


Role of client and server stub procedures in RPC

JAVA RMI Example

Introduction to Java RMI

Java RMI allowed programmer to execute remote function class using the same semantics as local functions calls.



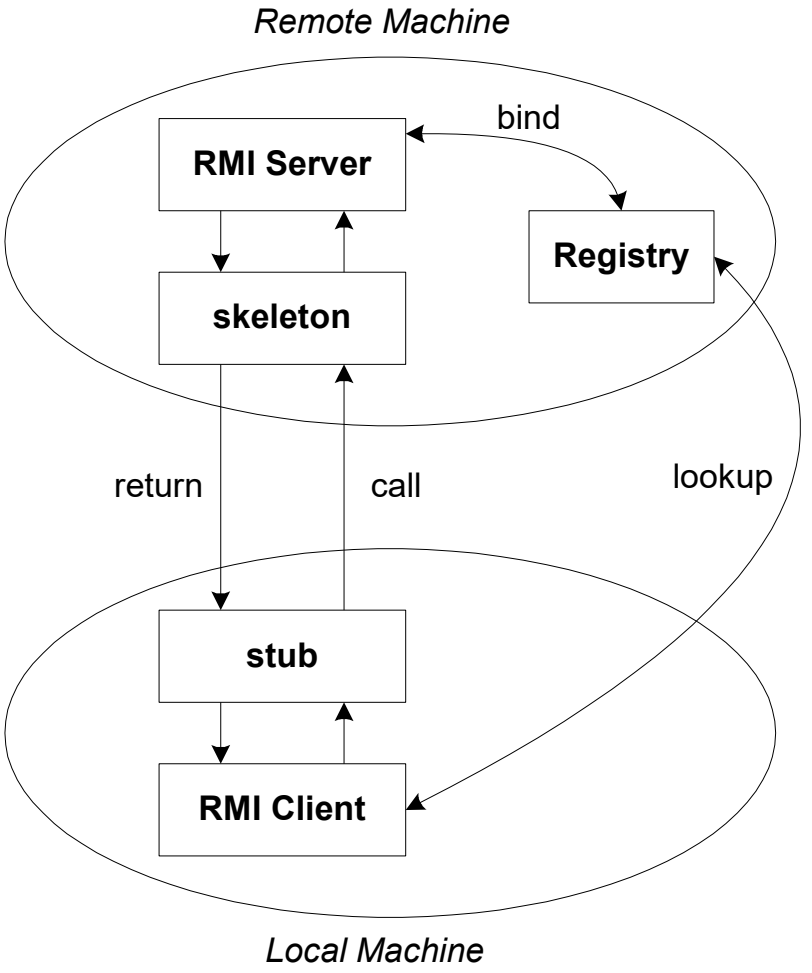
The General RMI Architecture

Local/Remote machines

Registry

Skeleton

Stub

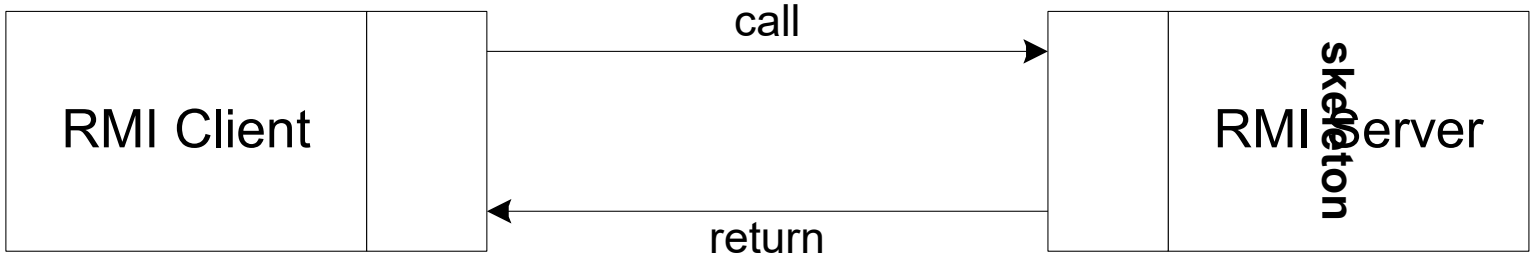


The Stub and Skeleton

A client invokes a remote method, the call is first **forwarded to stub.** The stub is responsible for sending the remote call over to the server-side skeleton

The stub opening a socket to the remote server, **marshaling** (encoding in standard, independent format) the object parameters and forwarding the data stream to the skeleton.

A skeleton contains a method that receives the remote calls, **unmarshals** the parameters, and **invokes** the actual remote object implementation.



Steps for Developing an RMI System

1. Define the remote interface
2. Develop the remote object by implementing the remote interface
3. Develop the client program
4. Compile the Java source files
5. Generate the client stubs and server skeletons
6. Start the RMI registry
7. Start the remote server objects
8. Run the client

Step 1: Defining the Remote Interface

To create an RMI application, the first step is the **defining of a remote interface between the client and server objects**.

```
/* SampleServer.java */
import java.rmi.*;

public interface SampleServer extends Remote
{
  public int sum(int a,int b) throws RemoteException;
}
```

Step 2: Develop the Remote Object and its Interface

The server is a simple **unicast remote server**.

Create server by extending **java.rmi.server.UnicastRemoteObject**.

```
/* SampleServerImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class SampleServerImpl extends UnicastRemoteObject implements SampleServer
{
  SampleServerImpl() throws RemoteException
  {
     super();
  }
…
```

Implement the remote methods

```
/* SampleServerImpl.java */
  …
  public int sum(int a,int b) throws RemoteException
  {
     return a + b;
  }
}
```

Step 2: Develop the remote object and its interface

```
/* SampleServerImpl.java */
public static void main(String args[])
{
    try
    {
     //set the security manager
      System.setSecurityManager(new RMISecurityManager());

      //create a local instance of the object
      SampleServerImpl Server = new SampleServerImpl();

      //put the local instance in the registry
      Naming.rebind("SAMPLE-SERVER" , Server);

      System.out.println("Server waiting.....");
    }
    catch (java.net.MalformedURLException me)       {
      System.out.println("Malformed URL: " + me.toString());   }
    catch (RemoteException re)  {
       System.out.println("Remote exception: " + re.toString());  }
}
```

Step 3: Develop the client program

In order for the client object to invoke methods on the server, it must **first look up the name of server in the registry**. You use the java.rmi.Naming class to lookup the server name.

The server name is specified as URL in the form (rmi://host:port/name)

Default RMI port is 1099.

The name specified in the URL must exactly match the name that the server has bound to the registry. In this example, the name is “SAMPLE-SERVER”

The remote method invocation is programmed using the remote interface name (remoteObject) as prefix and the remote method name (sum) as suffix.

Step 3: Develop the client program

```
import java.rmi.*;
import java.rmi.server.*;
public class SampleClient
{
   public static void main(String[]  args)
   {
      // set the security manager for the client
      System.setSecurityManager(new RMISecurityManager());
      //get the remote object from the registry
      try
       {
         System.out.println("Security Manager loaded");
         String url = "//localhost/SAMPLE-SERVER";
         SampleServer remoteObject = (SampleServer)Naming.lookup(url);
         System.out.println("Got remote object");
         System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );
       }
      catch (RemoteException exc) {
         System.out.println("Error in lookup: " + exc.toString()); }
      catch (java.net.MalformedURLException exc) {
         System.out.println("Malformed URL: " + exc.toString());   }
      catch (java.rmi.NotBoundException exc)  {
         System.out.println("NotBound: " + exc.toString());
      }
```

Step 4 & 5: Compile the Java source files & Generate the client stubs and server skeletons

Once the interface is completed, you need to generate stubs and skeleton code. The RMI system provides an RMI compiler (rmic) that takes your generated interface class and procedures stub code on its self.

```
javac SampleServer.java
javac SampleServerImpl.java
rmic SampleServerImpl
javac SampleClient.java
```

Step 6: Start the RMI registry

RMI Registry must be started manually.

The **rmiregistry** us uses port **1099** by default. You can also bind rmiregistry to a different port by indicating the new port number as : rmiregistry <new port>

rmiregistry (Unix)

start rmiregistry (Windows)

Steps 7 & 8: Start the remote server objects & Run the client

Once the Registry is started, start the server.

Set the security policy

java –Djava.security.policy=policy.all SampleServerImpl

java –Djava.security.policy=policy.all SampleClient

More at http://www.neward.net/ted/Papers/JavaPolicy/