RMI – Remote Method Invocation

- Introduction
- What is RMI?
- RMI System Architecture
- How does RMI work?
- Distributed Garbage Collection
- RMI & the OSI Reference Model
- Security
- Programming with RMI

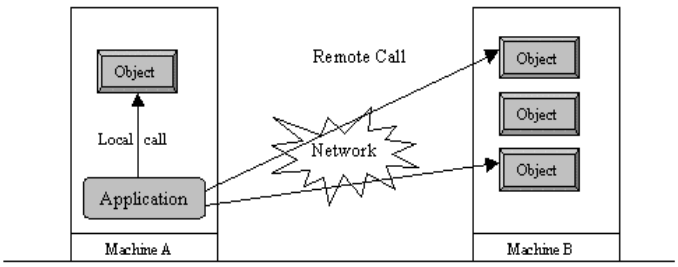
Introduction

- *Low-level* sockets can be used to develop client/server distributed applications
- But in that case, a protocol must be designed
- Designing such a protocol is hard and error-prone (how to avoid deadlock?)
- RMI is an alternative to sockets

What is RMI?

- A core package of the JDK1.1+ that can be used to develop distributed applications
- Similar to the RPC mechanism found on other systems
- In RMI, methods of remote objects can be invoked from other JVMs
- In doing so, the programmer has the illusion of calling a local method (but all arguments are actually sent to the remote object and results sent back to callers)

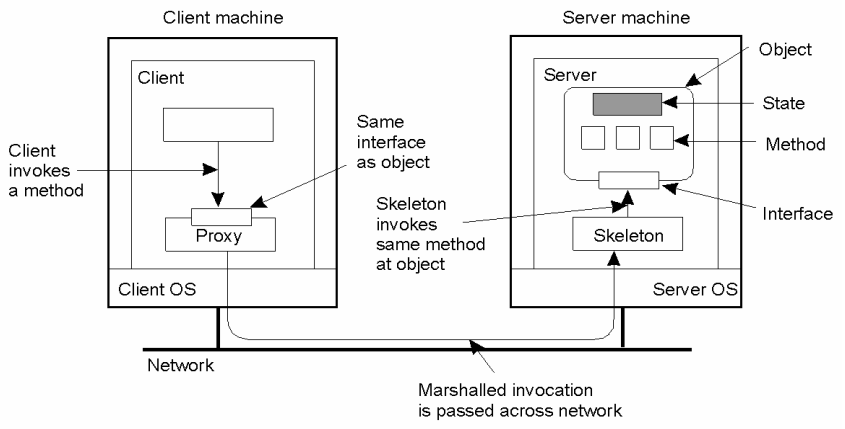
Local v. Remote method invocation



Goals and Features of RMI

- seamless object remote invocations
- callbacks from server to client
- distributed garbage collection
- NOTE: in RMI, all objects must be written in Java!

Distributed Objects and Client-Side Proxy



Parameters and Remote Objects

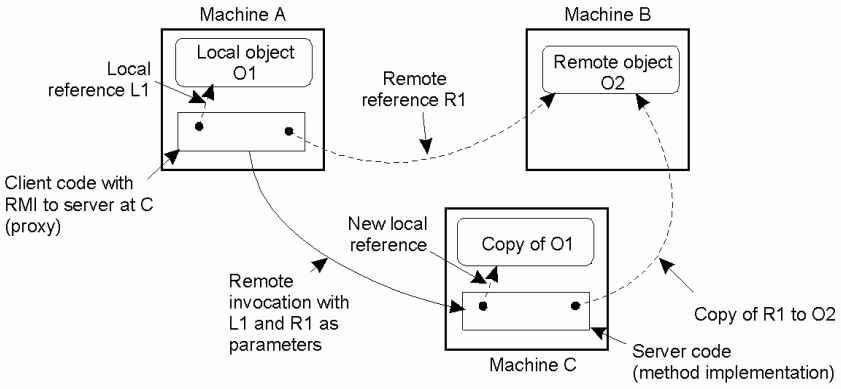
- regular parameters of method invocations are passed by value
  - they get serialized, sent over, and de-serialized, thus recreating a copy of the parameter
    - make sure your parameters are serializable!
  - as a result, modifications to such objects won't be noticed by the caller!
    - can be passed back as return value of the method

More on Remote Objects

- on the other hand, parameters that are remote objects themselves are not copied!
  - RMI ensures that if such objects' methods are invoked by the remote object, they are in turn remote invocations
  - in fact, the argument type should be compatible with the remote interface of the object, not its implementation
  - this is how RMI achieves *callbacks (discussed later)*

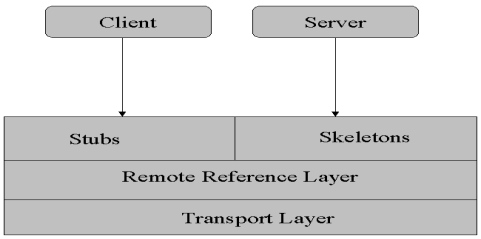
Parameter Passing

- The situation when passing an object by reference or by value.



RMI System Architecture

- Built in three layers (they are all independent):
  - Stub/Skeleton layer
  - Remote reference layer
  - Transport layer



The Stub/Skeleton layer

- The interface between the application layer and the rest of the system
- Stubs and skeletons are generated using the RMIC compiler
- This layer transmits data to the remote reference layer via the abstraction of marshal streams (that use object serialization)
- This layer doesn't deal with the specifics of any transport

The Stub/Skeleton I

- Client stub responsible for:
  - Initiate remote calls
  - Marshal arguments to be sent
  - Inform the remote reference layer to invoke the call
  - Unmarshaling the return value
  - Inform remote reference the call is complete
- Server skeleton responsible for:
  - Unmarshaling incoming arguments from client
  - Calling the actual remote object implementation
  - Marshaling the return value for transport back to client

The Remote Reference Layer

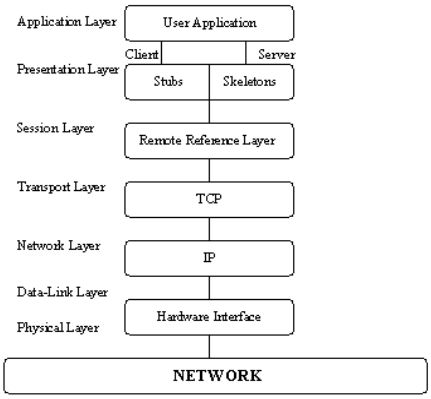
- The middle layer
- Provides the ability to support varying remote reference or invocation protocols independent of the client stub and server skeleton
- Example: the unicast protocol provides point-to-point invocation, and multicast provides invocation to replicated groups of objects, other protocols may deal with different strategies…
- Not all these features are supported….

The Transport Layer

- A low-level layer that ships serialized objects between different address spaces
- Responsible for:
  - Setting up connections to remote address spaces
  - Managing the connections
  - Listening to incoming calls
  - Maintaining a table of remote objects that reside in the same address space
  - Setting up connections for an incoming call
  - Locating the dispatcher for the target of the remote call

RMI and the OSI reference model

- How can RMI be described by this model?



How does RMI work?

- An invocation will pass through the stub/skeleton layer, which will transfer data to the remote reference layer
- The semantics of the invocation are carried to the transport layer
- The transport layer is responsible for setting up the connection

The Naming Registry

- The remote object must register itself with the RMI naming registry
- A reference to the remote object is obtained by the client by looking up the registry

Distributed Garbage Collection

- RMI provides a distributed garbage collector that deletes remote objects no longer referenced by a client
- Uses a reference-counting algorithm to keep track of live references in each Virtual Machine
- RMI keeps track of VM identifiers, so objects are collected when no local or remote references to them

Programming with RMI

- Anatomy of an RMI-based application
  - Define a remote interface
  - Provide an implementation of the remote interface
  - Develop a client
  - Generate stubs and skeletons
  - Start the RMI registry
  - Run the client and server

Define a Remote Interface

- It specifies the characteristics of the methods provided by a server and visible to clients
  - Method signatures (method names and the type of their parameters)
- By looking at the interface, programmers know what methods are supported and how to invoke them
- Remote method invocations must be able to handle error messages (e.g. can't connect to server or server is down)

Characteristics of Remote Interface

- Must be declared *public*
- To make an object a remote one, the interface must extend the *java.rmi.Remote* interface
- Each method declared in the interface must declare *java.rmi.RemoteException* in its *throws* clause.

Implement the Remote Interface

- The implementation needs to:
  - Specify the remote interface being implemented
  - Define the constructor of the remote object
  - Implement the methods that can be invoked remotely
  - Create an instance of a remote object
  - Register it with the RMI registry

Develop a Client

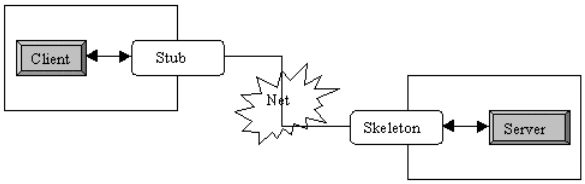
- Creates a variable of type “interface” and assigns the result of the remote object lookup (after typecasting…)
- Lookup is done using registry on server
  - Question: where is the class?
- Invoke remote interface methods as if object was local

Example Application

- See course website
- Compare to SUN-RPC example or XML-RPC example

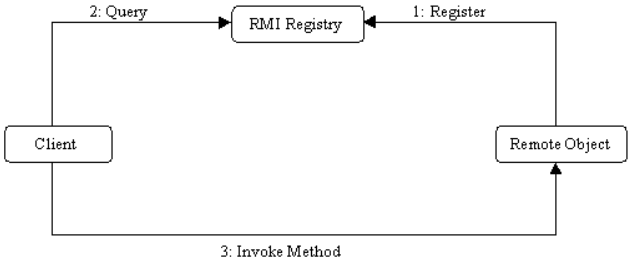
Generate stubs and skeletons

- Use the **rmic** compiler (in Java versions prior to Java 5, starting with Java 5 the Java Compiler does this)
- Place the remote interface and the stub class on the client side, and both the stub and skeleton on the server side.
  - alternately, some of these files could be dynamically loaded (will see this later)



Start the RMI registry

- It is a naming service that allows clients to obtains references to remote objects



Run the server and client

- Run the rmi registery (default port: 1099)
- Run the server
- Run the client

More on the Registry

- it is an RMI application too!
  - needs access to stubs
  - set the `classpath` accordingly!
- binding methods can only be called by code executing on the same host
- can launch programmatically
  - `LocateRegistry.createRegistry(int port)`
- can list all registered objects
  - `String[] Naming.list(String address)`

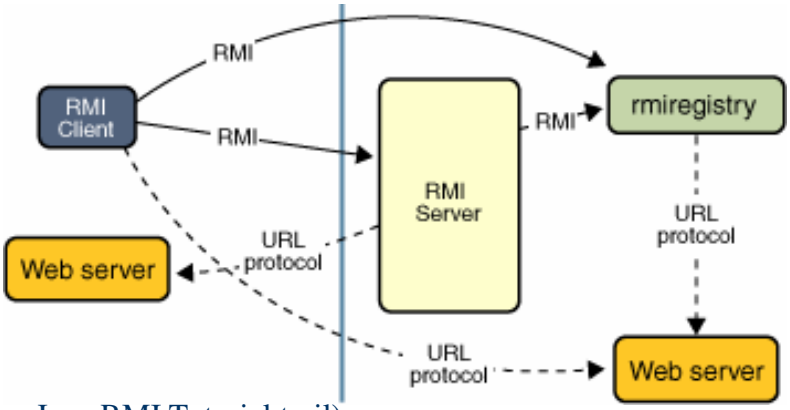
Callbacks

- to make callbacks possible:
  - pass a reference of the object obj1 as a parameter of a remote method invocation, say on object obj2
  - obj1 must implement a remote interface, and provide a stub to obj2, just like obj2 did to obj1
  - no need to register obj1, since obj2 obtains the reference directly through the method invocation
    - also remember, you can't register an object on a remote registry
  - the bind() method of the registry is a good example of that

Automatic Class File Distribution

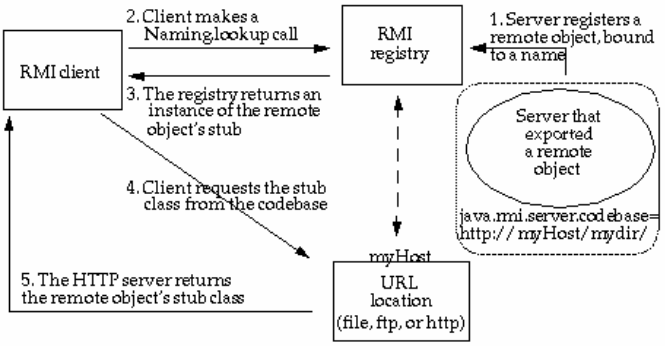
- It can seem limiting to have to deploy stub and skeleton files despite the fact that there is no direct reference to them in the source code
  - same can be said of certain subclasses of method parameter classes
- It is possible to find and download such classes on demand

Automatic Class File Distribution



(from Java RMI Tutorial trail)

Automatic Class File Distribution: Automatically Obtaining Stubs



Automatic Class File Distribution

- RMI class loading is attempted on the following locations:
  - the classpath
  - the location that is encoded by RMI along with class name information
  - additional codebase info specified upon execution:

```
java -Djava.rmi.server.codebase=http://carleton.ca/classes
  MyServerClass
```

Automatic Class File Distribution

- Loading classes dynamically can expose your machine to malicious code!
  - Handling of security again Java version specific
  - New versions: specify appropriate security policy file (see tutorial trail)
  - Old versions: need to load specific security manager, which allows you to load classes without allowing them to access your system resources:

```
System.setSecurityManager(new RMISecurityManager());
RemoteService remoteService = new RemoteService();
Naming.bind("serviceName", remoteService);
```

Code Mobility!

- The ability to download objects along with their class files is a powerful concept
- you can in fact make an object "hop" from host to host and perform various operations
  - can be useful for network management
  - can be used for distributed search
- if the itinerary and/or the behavior is decided by the code autonomously, we are talking about a mobile agent
  - extra support for mobile code is provided by specialized platforms such as Voyager
  - extra support for mobile agents is provided by platforms such as Aglets, Grasshopper...