Lecture 23
Java Remote Method Invocation
Recall

• Concurrency
  Several operations may be in progress at the same time on the same machine

• Parallelism
  Several operations may be executing simultaneously on the same machine

• “Distributed-ness”
  Several machines may be working at the same time for the same application
So Far We Have Concentrated On:

• Concurrency in Java
  – Threads
  – Locks
  – Etc.

• Parallelism in Java
  – Performance tuning
  – Fork/Join
  – Etc.

• Focus has been on applications running inside a single Java Virtual Machine (JVM)
Remote Method Invocation (RMI)

• Java support for *distributed programming*
  – Applications may use several JVMs
  – JVMs may be on different nodes in a network
  – Key constraint: no shared memory!
• RMI enables methods in objects hosted by on JVM to be called from a different JVM
  – This approach to distributed-system design is often called the *distributed object model*
  – Other distributed object models
    • DCOM
    • Corba
  – Other distributed models
    • Message passing
    • Event-based architectures
Some Distributed System Terminology

• **Host**
  Computer running in a distributed environment

• **Port**
  Communication channel used by hosts to exchange messages

• **Network**
  System consisting of hosts, equipment used to connect hosts

• **IP address**
  Internet Protocol address: number assigned to a host connected to the internet so that other hosts may communicate with it

• **MAC address**
  Media Access Control address: number assigned to a host on a local-area network
RMI Distributed Object Model

- Remotely accessible objects reside on servers (= JVMs)
- Client objects can invoke methods in remote objects
- RMI protocol handles transfer of data to/from servers/clients
Questions

• How does the client object pass arguments to the remote object?
• How does the remote object return information to the client object?
• How do distributed objects find out about each other?
• How does the client object know what argument types to pass, and what return type to expect?
• How does the client object know if the remote object can be trusted (and vice versa)?
Exchanging Information Between Objects via RMI

• RMI uses **TCP / IP** to transfer information between objects
  – TCP = Transmission Control Protocol
  – IP = Internet Protocol

• TCP / IP is a protocol for exchanging data among computers connected to a network
  – TCP (inter-application) is connection-oriented
  – IP (inter-machine) is connection-less, packet-based
  – Data is passive (i.e. sequences of bytes)
RMI and TCP / IP

- In Java one often calls a method with objects as parameters
- TCP / IP only deals with sequences of bytes
- To maintain the illusion of “objects flowing over the network”, distributed-object models (including RMI) use marshaling/unmarshaling
  - Marshaling: translating objects into sequences of bytes
  - Unmarshaling: translating sequences of bytes back into objects
Marshaling / Unmarshaling in RMI

• Primitive types (int, boolean, etc.) can be handled easily
• Remote objects passed by reference (basically, by address)
• Java RMI uses serialization / deserialization to handle marshaling / unmarshaling of local objects, which are passed by value
  – Serialization: converting an object into a sequence of bytes
    • Bytes may stored in a file / sent across network / etc.
    • Entire persistent state of object is stored
  – Deserialization: reconstruction of an object from a sequence of bytes
    • Persistent state of object is rebuilt from bytes
    • Exceptions thrown if bytes contain error, or class is unknown, etc.

• To support serialization, class must implement the java.io.Serializable interface, and all (non-transient) fields must be serializable
  – No methods in interface!
  – Implementing interface is just a signal to compiler that serializability must be checked
  – If you try to serialize / deserialize an object that is not serializable, NotSerializableException is thrown
RMI Distributed Object Model with (Un)Marshaling

In the RMI Distributed Object Model, a client object invokes methods on a remote object through the JVM (server). The process involves (un)marshaling operations: the client object invokes (marshals) the remote object, which then returns (unmarshals) the result. This communication occurs over TCP/IP.
Locating Remote Objects: Object Registries

• Object registry is a name server that relates remotely accessible objects with (unique) names
  – Each server has an object registry on the same host computer
  – The registry associates each remotely accessible object on the server with a name

• A client wishing to access a remote object can do so by giving the object name to a registry

• A server wishing to make an object available for RMI must register it with its object registry
RMI Architecture / Flow

1. Server creates remotely accessible object
2. Server registers object with registry, giving it unique name
3. Client requests remote object by name from registry
4. Registry returns stub to client
5. Client invokes stub method
6. RMI mechanism uses marshaling / unmarshaling to transfer arguments to server, results to client
Stubs

• A stub for a remote object is a proxy that the client uses to initiate remote method invocations
  – When a client queries an object registry for an object, what is returned is a stub
  – The stub matches the same interface (more later) as the remote object
  – The stub handles marshaling of arguments, unmarshaling of results, and communication with runtime environment of remote object

• When a client obtains a stub for a remote object, any method the client invokes on the stub will result in corresponding method in remote object being invoked
RMI Architecture (Refined)
Warning

• The discussion of stubs is with respect to Java 5.0 and later
  – Earlier versions of RMI required the use of a separate compiler, rmic, to produce stubs
  – Disseminating stubs to clients was more complicated
  – In pre-5.0 Java there were also skeletons, which sat on the server side and handled communications with stubs

• The Java 5.0 and later approach is simpler, but you may still encounter legacy code using the older approach
The `java.rmi.Remote` Interface

- Classes of remote objects must implement the `Remote` interface in `java.rmi`
- Here is the interface
  ```java
  interface Remote { }
  ```
- ???
  - `Remote` is an example of a “marker interface” (like `Serializable`)
  - Marker interfaces indicate that classes implementing them are intended for special purposes
  - `Remote` objects will generally implement interfaces that extend `Remote`
    - Requirement on methods in such interfaces: they must throw `RemoteException`
    - This exception is raised when there are problems with the remote invocation (e.g. network disruptions, host problems, etc.)
Example: Test String Printing

• Application contains four files
  – TestString.java
    Remote interface for test-string objects
  – TestStringServer.java
    Remote object class
  – TestStringClient.java
    Client code for accessing remote objects
  – ServerLaunch.java
    Code for creating, registering remote TestStringServer object

• Files must be compiled, then launched
Launching an RMI Application

• Launch registry (on server side)
• Launch server
• Launch client
Launching an RMI Registry

• Two approaches
  – Execute the command `rmiregistry` at the command prompt
    • In Windows: `start rmiregistry`
    • In Linux: `rmiregistry &`
    • The directory holding the `rmiregistry` executable must be in your path!
    • This registry may be shared by multiple servers
  – In your Java (server) program, execute
    `LocateRegistry.createRegistry();`
• In both cases the registry process will listen on port 1099 by default
• You can specify a different port by giving an optional argument to the command / Java method call
• The registry must know what the .class files are!
  – Can start the command in the relevant directory
  – We’ll see other approaches later
Launching Test String Server, Client

- **Execute** `ServerLaunch` for server
  - Can be done from Eclipse
  - Can also be done from command line:
    ```
    java ServerLaunch
    ```
- **Execute** `TestStringClient` for client

*Note:* both applications need to know the .class files