Topics covered in this lecture

- RMI
- Distributed garbage collection
- Activatable objects
- Serialization and pitfalls

Dynamic Invocation: Alternative to proxies

- Proxy is generated from an interface definition
  - Then compiled *statically*
  - Sometimes this is not enough
- What if a client program receives a remote reference to an object *that was not available at compile time*?
- We need another way to invoke the object
  - Dynamic Invocation

Dynamic invocation

- The remote object reference includes information about the interface of the remote object
- The names of methods and types of arguments are needed to make the invocation
**Server program**
- Contains classes for the dispatchers and skeletons
- Implementations of classes of all server remote objects
- Server remote objects also created in response to requests from clients

**Factory methods**
- Remote object interfaces cannot include constructors
  - Remote server objects cannot be created by remote invocation on constructors
  - Remote server objects are created either in:
    - The initialization section
    - Methods in a remote interface designed for that purpose
    - Also called factory methods

**The binder**
- Client programs need a way to obtain a remote object reference
  - For at least one of the remote objects held by server
- A binder is separate service that maintains a table with a mapping of textual names to remote object references
- Used by:
  - Servers: Register their remote objects by name
  - Clients: Lookup remote objects

**Server threads**
- When an object executes a remote method invocation?
  - Execution may lead to further invocations of methods in other remote objects
- Avoid situations where execution of a remote invocation delays execution of another
- Servers allocate a separate thread for the execution of each remote invocation
- There is also support for the thread per-object model

**Activation of remote objects**
- Often it is not practical for objects to be kept in running processes for unlimited periods
  - Especially if they are being used sporadically
- Servers can be started whenever they are needed by the clients
  - Similar strategy is used in standard TCP services such as FTP
    - Started on demand by a service called Inetd

**Active and passive remote objects**
- Active
  - Available for invocation within a running process
- Passive
  - Not currently active, but can be made so
  - Consists of two components
    - Implementation of its methods
    - State in marshaled form
Activation is the creation of active objects from passive ones
- Register passive objects that are available for activation
- Start named server processes and activate remote objects in them
- Keep track of locations of servers for remote objects

Java RMI includes ability to make some remote objects activatable
- When an activatable object is invoked?
  - If it is not active, it is made so from its marshaled state
  - Uses one activator on each server machine

Distributed garbage collection
- Based on reference counting
  - Whenever a remote object reference enters a process:
    - A proxy is created and stays there for as long as it is needed
    - The process where the remote object lives (its server) should be informed of the new proxy
    - When there is no proxy at client; server should be informed

The distributed garbage collector works with the local garbage collector
- Each server process maintains a set of names of processes that hold remote object references
  - For each of its remote objects
  - B.holders is the set of client processes with proxies for remote object B
  - When client C receives a remote reference to a particular remote object?
    - Makes addRef(B) invocation to server of that remote object and then creates a proxy
    - Server adds C to B.holders

The distributed garbage collector works with the local garbage collector
- When C's garbage collector notices that the proxy for remote object B is no longer reachable
  - Makes a removeRef(B) to the corresponding server
  - Then deletes the proxy
  - Server removes C from B.holders
  - When B.holders is empty
    - Server's local garbage collector will reclaim space occupied by B
    - Unless there are local holders
The distributed garbage collection can tolerate failure of client processes

- Servers lease their objects to clients for a limited period of time
  - Starts when client makes an addRef invocation
  - Ends when time expires or a removeRef invocation is made
- Clients are responsible for requesting server to renew leases before they expire

Java RMI Interfaces

- Single-language system
  - In CORBA, programmer should learn IDL
  - Understand how it maps to the implementation language

Remote interfaces in Java RMI

- Defined by extending java.rmi.Remote
- Methods must throw java.rmi.RemoteException
- Application specific exceptions may also be thrown

Example

```java
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject graphObj) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```
Parameters and result passing

- Parameters of a method are assumed to be input parameters.
- Result of a method is the single output parameter.
- Any object that is serializable can be passed as an argument or result.
  - i.e. Object implements the Serializable interface.

Passing objects

- When the parameter or result value is defined as a remote interface?
  - Corresponding argument or result passed as a remote object reference.
- All serializable non-remote objects:
  - Copied and passed by value.
  - When object is passed by value, new object is created in the receiver’s process.
  - Methods on this object are invoked locally; so state can differ from the original object.

Arguments and return values are serialized to a stream

- When an object that implements the Remote interface is serialized?
  - It is replaced by its remote object reference.
  - Contains name of remote object’s class.
- When any object is serialized?
  - Class information is annotated with the location of class (URL).
  - Allows class to be downloaded by the server.

Downloading classes

- Java is designed to allow classes to be downloaded from one VM to another.
- Relevant for distributed objects that interact via remote invocations.
- Code is downloaded automatically when:
  - Recipient does not possess class of object that is passed by value.
  - If recipient of remote object reference does not possess class for a proxy.

Advantages of this model

- No need for users to keep some set of classes in their working environment.
- Client and server programs make transparent use of instances of new classes when they are added.

RMI Registry

- This is the binder for RMI.
- An instance of RMIRegistry should run on every server computer that hosts remote objects.
- Maintains a table that maps:
  - Textual, URL-style names to references to remote objects.
  - Accessed by methods of the Naming class:
    - Argument includes a URL formatted string.
    - rmi://computerName:port/objectName.
Looking at our remote ShapeList interface

```
import java.rmi.*;
import java.util.Vector;

public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject graphObj) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Implementation of the Remote ShapeList interface

```
import java.rmi.*;
import java.util.Vector;

public class ShapeLister implements ShapeList {
    private Vector theList;   // contains list of shapes
    private int version;

    public ShapeLister() {}

    public Shape newShape(GraphicalObject graphObj) throws RemoteException {
        version++;
        Shape shape = new ShapeRemote(graphObj, version);
        theList.addElement(shape);
        return shape;
    }

    public Vector allShapes() throws RemoteException {...}

    public int getVersion() throws RemoteException;
}
```

The Server for the ShapeList remote object

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeListServer {
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());

        try {
            ShapeList shapeList = new ShapeLister();
            ShapeList exported = 
                UnicastRemoteObject.exportObject(shapeList, port);
            Naming.rebind("rmi://carrot.cs.colostate.edu/ShapeList", exported);
        } catch (...) {
        }
    }
}
```

A closer look at exportObject

```
1. Defined on UnicastRemoteObject
2. Makes object available to the RMI runtime
3. Makes it available to receive incoming invocations
4. Using UnicastRemoteObject ensures ...
   - Object lives only as long as the process in which it was created
```
Installing a Security Manager

- Needs to create a security manager
  - Let Java security apply protection appropriate for RMI server
  - RMISecurityManager is default security manager that's provided
    - Protects local resources to ensure that classes loaded from remote sites cannot have any effect on local resources
      - Such as files
    - Differs from standard Java security manager in allowing program to use its own class loader and to use reflection

If an RMI server sets no security manager?

- Proxies and classes can only be loaded from the local classpath
  - To protect program from code that is downloaded as a result of remote method invocations

Client Program

```java
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient {
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        ShapeList shapeList = null;
        try {
            shapeList = (ShapeList) Naming.lookup("rmi://carrot/ShapeList");
            Vector shapes = shapeList.allShapes();
        } catch (RemoteException e) {...} catch (Exception e) {...}
    }
}
```

Callbacks

- Instead of clients polling the server to see if some event occurred
- Server informs client when the event occurs
- Callback refers to a server's action of notifying clients about an event

Implementing callbacks in RMI

- Clients create a remote object
  - Remote interface contains a method for the server to call
    - This is the callback object
- Server provides operation allowing interested clients to register their callback objects
- When an event occurs
  - Server calls the interested clients
Snippets outlining the callback

```java
public interface WhiteboardCallback implements Remote {
    void callback(int version);
}
```

On the server-side the ShapeList interface must have...
```java
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

The design of RMI
- The original implementation implemented all the components
- In Java 1.2, reflection facilities were used to make a generic dispatcher and avoid need for skeletons
- Prior to J2SE 5.0, client proxies were generated by a compiler rmic
  - No longer necessary with recent versions of J2SE
  - Support for dynamic generation of stub classes at runtime

Inheritance structure of classes supporting RMI

```
Server
   |__________RemoteObject
   |               
RemoteServer     
   |               
   \            /  
   |Activateable  |  UnicastRemoteObject
```

MARSHALLING AND UNMARSHALLING

- Methods to enable computers to exchange binary data values
  - Values are converted to an agreed external format before transmission
    - And, converted to local form upon receipt
  - Values are transmitted in sender's format
    - With an indication of the format
    - Recipient converts values if necessary

External data representation and marshalling
- Information in running programs represented as **data structures**
  - E.g., a set of interconnected objects
- Information in transmitted data?
  - Sequence of bytes
- Irrespective of the form of communications, data structures must be:
  - Flattened before transmission and rebuilt on arrival
**Marshalling/Unmarshalling**

- **Marshalling**  
  Translation of structured data items and primitive values into an external data representation

- **Unmarshalling**  
  Generation of primitive values from their external data representation
  
  Rebuilding of data structure

---

**An issue with design of marshalling methods**

- Should we include type information?
  
  - **CORBA**  
    Common Data Representation (CDR) includes values of transmitted objects and not the types  
    These can be inferred
  
  - **Java Serialization**  
    Does include type information

---

**CORBA’s CDR**

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order</td>
</tr>
<tr>
<td>array</td>
<td>Array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>In the order of declaration of components</td>
</tr>
</tbody>
</table>

Also includes primitive types: short (16-bit), long (32-bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, boolean (TRUE or FALSE), octet (8-bit) and any

---

**Java Object Serialization**

- **Serialization**  
  Flattening an object or a connected set of objects into serial form  
  For storing on disk or transmissions

- **Deserialization**  
  Restoring the state of the object from the serial form  
  Has no prior knowledge of the type of objects in the serialized form  
  Some information about the class of each object must be included in the serial form

---

**The Serializable interface**

- The Serializable interface has no methods

- Stating that a class implements the Serializable interface:
  
  Has the effect of allowing instances to be serialized

---

**Let’s look at a simple Class**

```java
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;

    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }

    …. Methods for accessing instance variables follow ..
}
```
The Java Serialized form

**Class name, version number**

Person 8-byte version number NO

<table>
<thead>
<tr>
<th>x</th>
<th>int year</th>
<th>java.lang.String name</th>
<th>java.lang.String place</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>S Smith</td>
<td>S London</td>
<td></td>
</tr>
</tbody>
</table>

Values of instance variables

Encoding class information

- The information about a class consists of the:
  - Class name
  - Version
  - Should change with major changes to the class

- **Version assignment**
  - Set by programmer
  - Calculated as a hash of the name of the class and its instance variables, methods and interfaces

- Deserialization checks to see if it has correct version of the class

Java objects contain references to other objects

- When an object is serialized, all the objects that it references are serialized
  - Ensures when an object is reconstructed all its references call be fulfilled at the destination
- References are serialized as **handles**
  - Handle is a reference to an object within the serialized form
  - 1-1 correspondence between object references and handles
  - Each object is written exactly once
  - Subsequent occurrences of an object? The handle is written instead of the object

Serializing an object

- **Class information is written out**
  - Each class is given a **handle** and no class is written more than once
  - Handles are written where necessary
  - This is followed by (types and names of) instance variables

- **Recursive** procedure continues until
  - Class information, types and names of instance variables of all classes are written out

How are things written out?

- Contents of instance variables that are of **primitive** types (e.g., int, char, boolean, etc)?
  - Written in portable binary format using methods of the **ObjectOutputStream** class

- Strings and characters are written using the **writeUTF** method
  - Universal Transfer Format (UTF)
  - ASCII characters are represented unchanged (1 byte), Unicode characters with multiple bytes

Controlling serialization

- Programmers can **modify** the effects of serialization
  - Declare variables that should not be serialized as **transient**
    - E.g., references to local resources such as files and sockets
Reflection makes it possible to do (de)serialization in a generic manner

- Java object serialization uses reflection to find out
  - Class name
  - The names, types, and values of instance variables

- For deserialization
  - Class name in serialized form is used to create class
  - This class is then used to create a new constructor with argument types corresponding to those in serialized form
  - New constructor is used to create a new object with instance variables whose values are read from the serialized form