Frequently asked questions from the previous class survey

December 5, 2019

CS555: Distributed Systems [Fall 2019]
Dept. Of Computer Science, Colorado State University

L28.

CS555: Distributed Systems [RMI]

Shrideep Pallickara
Computer Science
Colorado State University

December 5, 2019

Topics covered in this lecture

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

L28.

Distributed garbage collection

- Based on reference counting
- Whenever a remote object reference enters a process:
  - A proxy is created and stays there 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

December 5, 2019

The distributed garbage collector works with the local garbage collector [1/2]

- 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

L28.

December 5, 2019

CS555: Distributed Systems [Fall 2019]
Dept. Of Computer Science, Colorado State University
The distributed garbage collector works with the local garbage collector [2/2]

- 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 remote objects

- Object making remote invocation is aware that target is remote
  - Must handle `RemoteException`
- Implementer is aware that it is remote
  - Must implement the `Remote` interface

Programming distributed applications with RMI is easy

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

```java
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;
}
```

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 same 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 RMI Registry 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.addElements(shape);
        return shape;
    }
    public Vector allShapes() throws RemoteException {
        return theList;
    }
    public int getVersion() throws RemoteException {
        return version;
    }
}
```

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`:

- Defined on `UnicastRemoteObject`.
- Makes object available to the RMI runtime.
- Makes it available to receive incoming invocations.
- Using `UnicastRemoteObject` ensures that:
  - The 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 the 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 the 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 ...
1st 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
  - Remote Object
  - RemoteServer
  - Activatable
  - UnicastRemoteObject

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 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

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), short, boolean, OCTET, or any 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](image1)

![Number, type and name of instance variables](image2)

![Values of instance variables](image3)

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 can 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

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

<table>
<thead>
<tr>
<th>Class name, version number</th>
<th>Number, type and name of instance variables</th>
<th>Values of instance variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>int name, String place, int year</td>
<td>Smith, London, 1984</td>
</tr>
</tbody>
</table>

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 can 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

The contents of this slide set are based on the following references

  [Chapter 4 and 5]