Distributed Objects & Remote Invocations

Distributed Software Systems

Motivation

- Sockets API ? send & recv calls ? I/O
- Remote Procedure Calls (RPC)
  - Goal: to provide a procedural interface for distributed (i.e., remote) services
  - To make distributed nature of service transparent to the programmer
    - No longer considered a good thing
- Remote Method Invocation (RMI)
  - RPC + Object Orientation
  - Allows objects living in one process to invoke methods of an object living in another process
Middleware layers

- Applications, services
- RMI and RPC
- request-reply protocol
  marshalling and external data representation
- UDP and TCP

Request-reply communication

Client

- doOperation
- (wait)
- (continuation)

Server

- getRequest
- select object
- execute method
- sendReply

Request message

Reply message
Interfaces in distributed systems

- Programs organized as a set of modules that communicate with one another via procedure calls/method invocations
- Explicit interfaces defined for each module in order to control interactions between modules
- In distributed systems, modules can be in different processes
- A remote interface specifies the methods of an object that are available for invocation by objects in other processes defining the types of the input and output arguments of each of them

CORBA IDL example

```csharp
// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
};

interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p);
    void getPerson(in string name, out Person p);
    long number();
};
```
Object model

- Object references
  - Objects accessed via obj. references
  - Object references can be assigned to variables, passed as arguments and returned as results
- Interfaces
  - Provides a signature of a set of methods (types of arguments, return values and exceptions) without specifying their implementations
- Actions (invocations)
- Exceptions
- Garbage Collection

Distributed Objects

- Remote object references
  - An identifier that can be used throughout a distributed system to refer to a particular remote object
- Remote interfaces
  - CORBA provides an interface definition language (IDL) for specifying a remote interface
  - JAVA RMI: Java interface that extends Remote interface
- Actions: remote invocations
- Distributed Garbage Collection: cooperation between local garbage collectors needed
- Remote Exceptions may arise for reasons such as partial failure or message loss
Remote and local method invocations

A remote object and its remote interface
Design Issues for RMI

**RMI Invocation Semantics**
- Invocation semantics depend upon implementation of Request-Reply protocol used by RMI
- Maybe, At-least-once, At-most-once

**Transparency**
- Should remote invocations be transparent to the programmer?
  - Partial failure, higher latency
- Current consensus: remote invocations should be made transparent in the sense that syntax of a remote invocation is the same as the syntax of local invocation (access transparency) but programmers should be able to distinguish between remote and local objects by looking at their interfaces, e.g. in Java RMI, remote objects implement the Remote interface

---

Request-reply communication

```
Client
  doOperation
  (wait)
  (continuation)

Server
  getRequest
  select object
  execute method
  sendReply

Request message
Reply message
```
Operations of the request-reply protocol

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
    sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.
public byte[] getRequest ();
    acquires a client request via the server port.
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
    sends the reply message reply to the client at its Internet address and port.

Request-reply message structure

<table>
<thead>
<tr>
<th>messageType</th>
<th>int (0=Request, 1=Reply)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestId</td>
<td>int</td>
</tr>
<tr>
<td>objectReference</td>
<td>RemoteObjectRef</td>
</tr>
<tr>
<td>methodId</td>
<td>int or Method</td>
</tr>
<tr>
<td>arguments</td>
<td>array of bytes</td>
</tr>
</tbody>
</table>
Request-Reply protocol

- Issues in marshaling of parameters and results
  - Input, output, Inout parameters
  - Data representation
  - Passing pointers? (e.g., call by reference in C)
- Distributed object references
- Handling failures in request-reply protocol
  - Partial failure
    - Client, Server, Network

CORBA CDR for constructed types

<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 (can also have wide characters)</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 the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>
**CORBA CDR message**

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td>length of string</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>&quot;Smith&quot;</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h___&quot;</td>
<td>length of string</td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td>&quot;Lond&quot;</td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td></td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on__&quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td>unsigned long</td>
</tr>
</tbody>
</table>

The flattened form represents a `Person` struct with value: `{'Smith', 'London', 1934}`

---

**Indication of Java serialized form**

<table>
<thead>
<tr>
<th>Serialized values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 3</td>
<td>8-byte version number</td>
</tr>
<tr>
<td>int year</td>
<td>java.lang.String name:</td>
</tr>
<tr>
<td>1934</td>
<td>java.lang.String place:</td>
</tr>
<tr>
<td>5 Smith</td>
<td>6 London</td>
</tr>
<tr>
<td>h0</td>
<td>h1</td>
</tr>
</tbody>
</table>

The true serialized form contains additional type markers; h0 and h1 are handles.
Representation of a remote object reference

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
<td>interface of remote object</td>
</tr>
</tbody>
</table>

CORBA interoperable object references

IOR format

<table>
<thead>
<tr>
<th>IDL interface type name</th>
<th>Protocol and address details</th>
<th>Object key</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface repository identifier</td>
<td>IIOP</td>
<td>host domain name</td>
</tr>
</tbody>
</table>
**RPC exchange protocols**

<table>
<thead>
<tr>
<th>Name</th>
<th>Messages sent by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client</td>
</tr>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
</tr>
</tbody>
</table>

**Handling failures**

- Types of failure
  - Client unable to locate server
  - Request message lost
  - Reply message lost
  - Server crashes after receiving a request
  - Client crashes after sending a request
Handling failures

- Client cannot locate server
  - Reasons
    - Server has crashed
    - Server has moved
    - (RPC systems) client compiled using old version of service interface
  - System must report error (remote exception) to client
    - Loss of transparency

Handling failures

- Lost request message
  - Retransmit a fixed number of times before throwing an exception
- Lost reply message
  - Client resubmits request
  - Server choices
    - Re-execute procedure service should be idempotent so that it can be repeated safely
    - Filter duplicates server should hold on to results until acknowledged
### Invocation semantics

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-transmit request message</td>
<td>Duplicate filtering</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Handling failures

#### Server crashes

- Client cannot tell difference
Handling failures

- Server crashes
  - At least once (keep trying till server comes up again)
  - At most once (return immediately)
  - Exactly once impossible to achieve

- SUN RPC
  - At least once semantics on successful call and maybe semantics if unsuccessful call

- CORBA, Java RMI
  - at most once semantics

---

Handling failures

- Implementing the request-reply protocol on top of TCP
  - Does it provide applications with different invocation semantics?
    - NO!
      - TCP does not help with server crashes
      - If a connection is broken, the end pts do not have any guarantees about the delivery of messages that may have been in transit
Handling failures

Client crashes
- If client crashes before RPC returns, we have an "orphan" computation at server
  - Wastes resources, could also start other computations

Orphan detection
- Reincarnation (client broadcasts new "epoch" when it comes up again)
- Expiration (RPC has fixed amount of time $T$ to do work)

RMI Implementation

Communication module
- Implements the request-reply protocol

Remote reference module
- Responsible for translating between local and remote object references and for creating remote object references
  - Maintains remote object table that maintains a mapping between local & remote object references
  - E.g. Object Adaptor in CORBA
The role of proxy and skeleton in remote method invocation

RMI Implementation

- RMI software
  - Generated by IDL compiler
  - Proxy
    - Behaves like remote object to clients (invoker)
    - Marshals arguments, forwards message to remote object, unmarshals results, returns results to client
  - Skeleton
    - Server side stub;
    - Unmarshals arguments, invokes method, marshals results and sends to sending proxy's method
  - Dispatcher
    - Receives the request message from communication module, passes on the message to the appropriate method in the skeleton
- Server and Client programs
RMI Implementation

- **Binder**
  - Client programs need a means of obtaining a remote object reference
  - Binder is a service that maintains a mapping from textual names to remote object references
  - Servers need to register the services they are exporting with the binder
  - Java RMI registry, CORBA Naming service

- **Server threads**
  - Several choices: thread per object, thread per invocation
  - Remote method invocations must allow for concurrent execution

---

RMI Implementation

- **Activation of remote objects**
  - Some applications require that information survive for long periods of times
  - However, objects not in use all the time, so keeping them in running processes is a potential waste of resources
  - Object can be activated on demand
    - E.g. standard TCP services such as FTP on UNIX machines are activated by inetd
**RMI Implementation**

- **Active and passive objects**
  - Active object = instantiated in a running process
  - Passive object = not currently active but can be made active
    - Implementation of its methods, and marshalled state stored on disk

- **Activator responsible for**
  - Registering passive objects that are available for activation
  - Starting named server processes and activating remote objects in them
  - Keeping track of locations of servers for remote objects that it has already activated

- **Examples:** CORBA implementation repository, JAVA RMI has one activator on each server computer

---

**RMI Implementation**

- **Persistent object stores**
  - An object that is guaranteed to live between activations of processes is called a persistent object
  - Stores the state of an object in a marshalled (serialized) form on disk

- **Location service**
  - Objects can migrate from one system to another during their lifetime
  - Maintains mapping between object references and the location of an object
RMI Implementation

- **Distributed Garbage Collection**
  - Java approach based on reference counting
  - Each server process maintains a list of remote processes that hold remote object references for its remote objects
  - When a client first removes a remote reference to an object, it makes an addRef() invocation to server before creating a proxy
  - When a client's local garbage collector notices that a proxy is no longer reachable, it makes a removeRef() invocation to the server before deleting the proxy
  - When the local garbage collector on the server notices that the list of client processes that have a remote reference to an object is empty, it will delete the object (unless there are any local objects that have a reference to the object)

- **Other approaches**
  - "Evictor" pattern
  - Leases

RPC/RMI systems

- **RPC systems**
  - SUN RPC
  - DCE RPC

- **RMI systems**
  - CORBA
  - DCOM
  - Java RMI
  - SOAP (Simple Object Access Protocol)
    - HTTP is request-reply protocol
    - XML for data representation
Java RMI

Features

- Integrated with Java language + libraries
  - Security, write once run anywhere, multithreaded
  - Object orientation
- Can pass "behavior"
  - Mobile code
  - Not possible in CORBA, traditional RPC systems
- Distributed Garbage Collection
- Remoteness of objects intentionally not transparent

Remote Interfaces, Objects, and Methods

- Objects become remote by implementing a remote interface
  - A remote interface extends the interface java.rmi.Remote
  - Each method of the interface declares java.rmi.RemoteException in its throws clause in addition to any application-specific clauses
Creating distributed applications using RMI

1. Define the remote interfaces
2. Implement the remote objects
3. Implement the client (can be done anytime after remote interfaces have been defined)
4. Register the remote object in the name server registry
5. Generate the stub and client using rmic
6. Start the registry
7. Start the server
8. Run the client

Java Remote interfaces Shape and ShapeList

```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 g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```
The Naming class of Java RMIregistry

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
This method removes a binding.

Remote lookup(String name)
This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()
This method returns an array of Strings containing the names bound in the registry.

---

Java class ShapeListServer with main method

import java.rmi.*;
public class ShapeListServer{
public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());
    try{
        ShapeList aShapeList = new ShapeListServant();  
        Naming.rebind("Shape List", aShapeList );  
        System.out.println("ShapeList server ready");
    }catch(Exception e) {
        System.out.println("ShapeList server main " + e.getMessage());
    }
}
Java class ShapeListServant
impliments interface ShapeList

```java
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList; // contains the list of Shapes
    private int version;
    public ShapeListServant() throws RemoteException { ... }
    public Shape newShape(GraphicalObject g) throws RemoteException { 
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }
    public Vector allShapes() throws RemoteException { ... }
    public int getVersion() throws RemoteException { ... }
}
```

Java client of ShapeList

```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 aShapeList = null;
        try {
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList");
            Vector sList = aShapeList.allShapes();
            } catch(RemoteException e) { System.out.println(e.getMessage()); }
        } catch(Excepton e) { System.out.println("Client: " + e.getMessage()); }
    }
}
```
Classes supporting Java RMI

RemoteObject
   \-- RemoteServer
      \-- Activatable
      \-- UnicastRemoteObject
         \-- <servant class>

Advanced Techniques

- Passing behavior
  - See Java RMI tutorial track example
- Callbacks
- Activation
The main components of the CORBA architecture

CORBA
IDL interfaces Shape and ShapeList

```idl
struct Rectangle{
    long width;
    long height;
    long x;
    long y;
};

struct GraphicalObject{
    string type;
    Rectangle enclosing;
    boolean isFilled;
};

interface Shape {
    long getVersion();
    GraphicalObject getAllState(); // returns state of the GraphicalObject
};

typedef sequence <Shape, 100> All;

interface ShapeList {
    exception FullException{ }
    Shape newShape(in GraphicalObject g) raises (FullException);
    All allShapes(); // returns sequence of remote object references
    long getVersion();
};
```

Java interface ShapeList generated by idltojava from CORBA interface ShapeList

```java
public interface ShapeList extends org.omg.CORBA.Object {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[] allShapes();
    int getVersion();
}
```
**ShapeListServant class of the Java server program for CORBA interface ShapeList**

```java
import org.omg.CORBA.*;

class ShapeListServant extends _ShapeListImplBase {
    ORB theOrb;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(ORB orb)
    
    {
        theOrb = orb;
        // initialize the other instance variables
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {
        version++;
        Shape s = new ShapeServant(g, version);
        if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        theOrb.connect(s);
        return s;
    }
    public Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

---

**Java class ShapeListServer**

```java
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ShapeListServer {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);
            ShapeListServant shapeRef = new ShapeListServant(orb);
            org.omg.CORBA.Object objRef =
            org.omg.CORBA.NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, shapeRef);
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) { sync.wait();}
        } catch (Exception e) { ... }
    }
}
```
Java client program for CORBA interfaces
Shape and ShapeList

import org.omg.CosNaming. *;
import org.omg.CosNaming.NamingContextPackage. *;
import org.omg.CORBA. *;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null); 1
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
                ShapeListHelper.narrow(ncRef.resolve(path)); 2
            Shape[] sList = shapeListRef.allShapes(); 3
            GraphicalObject g = sList[0].getAllState(); 4
        } catch(org.omg.CORBA.SystemException e) {...}
    }
}

IDL module Whiteboard

module Whiteboard {
    struct Rectangle{
        ...
    };
    struct GraphicalObject {
        ...
    };
    interface Shape {
        ...
    };
    typedef sequence <Shape, 100> All;
    interface ShapeList {
        ...
    };
}
**IDL constructed types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>typedef sequence &lt;Shape, 100&gt; All; typedef sequence &lt;Shape&gt; All</td>
<td>Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.</td>
</tr>
<tr>
<td>string</td>
<td>String name; typedef string&lt;8&gt; SmallString;</td>
<td>Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.</td>
</tr>
<tr>
<td>array</td>
<td>typedef octet uniqueld[12]; typedef GraphicalObject GO[10][8]</td>
<td>Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.</td>
</tr>
</tbody>
</table>

**IDL constructed types cont’d**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>struct GraphicalObject { string type; Rectangle enclosing; boolean isFilled; };</td>
<td>Defines a type for a record containing a group of related entities. <em>Structs</em> are passed by value in arguments and results.</td>
</tr>
<tr>
<td>enumerated</td>
<td>enum Rand (Exp, Number, Name);</td>
<td>The enumerated type in IDL maps a type name onto a small set of integer values.</td>
</tr>
<tr>
<td>union</td>
<td>union Exp switch (Rand) { case Exp: string vote; case Number: long n; case Name: string s; };</td>
<td>The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by <em>enum</em> which specifies which member is in use.</td>
</tr>
</tbody>
</table>
CORBA interoperable object references

IOR format

<table>
<thead>
<tr>
<th>IDL interface type name</th>
<th>Protocol and address details</th>
<th>Object key</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface repository identifier</td>
<td>IIOP</td>
<td>host domain</td>
</tr>
</tbody>
</table>

Naming graph in CORBA Naming Service

initial naming context

ShapeList

B

C

D

E

P

R

Q

S

T

initial naming context

XX

V

U

T
Part of the CORBA Naming Service NamingContext interface in IDL

```idl
struct NameComponent { string id; string kind; }

typedef sequence <NameComponent> Name;

interface NamingContext {
  void bind (in Name n, in Object obj);
  binds the given name and remote object reference in my context.
  void unbind (in Name n);
  removes an existing binding with the given name.
  void bind_new_context(in Name n);
  creates a new naming context and binds it to a given name in my context.
  Object resolve (in Name n);
  looks up the name in my context and returns its remote object reference.
  void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
  returns the names in the bindings in my context.
};
```

Readings

- Coulouris - Chapters 5, 6, 17
- WWW (see links on class web page)
- Java RMI tutorial on web
- "A Young Persons Guide to SOAP"
- CORBA web documents at OMG web site