Lecture 8: Beyond Java RMI

CS178: Programming Parallel and Distributed Systems

February 21, 2001
Steven P. Reiss

I. Overview

A. Client-server computing is widely used

1. Server generally controls a shared resource

2. Clients want access to that shared resource

B. Resource access is generally done via messages

1. Machines are widely distributed

2. Messages form the communication mechanism

3. Sockets form the basis for sending/receiving messages

C. Message passing is fairly low-level

1. We thus build higher-level abstractions on top of it

a) RPC

b) Remote object invocation (OO RPC)

2. And try to incorporate this into a language

a) Java RMI

b) NIL and messages

D. This time -- other alternatives

II. Java RMI notes

A. Multiple threads

1. Generally each RMI request is handled by a separate thread

2. You have to provide any synchronization for these threads

B. Generating stubs and skeletons

1. Recall how RMI handles remote objects

a) Client makes a call; call translated in messages

b) Arguments are marshalled, etc.

c) Server gets messages, unmarshalls arguments

d) Server calls method on actual object

e) Return value sent back as message

f) Return message translated into value and returned

2. Non-remote objects handled by serialization

a) All passed objects must be serializable

b) Objects read/written -- beware of static/transient fields

3. In order to do this you need to have

a) Stub in the client to translate the calls, handle return

b) Skeleton in server to translate messages, make call

4. Where do these stubs and skeletons come from

a) In java they are dynamically loaded by RMI package

b) This is done invisibly and automatically

c) But they still need to be generated

5. RMIC -- RMI compiler does this

a) First compile the classes

b) Then rmic -d <output> class class ...

III. CORBA

A. Objectives

1. Provide object-based distributed computing

a) Based on a robust object model

b) Language independent

2. Distribution transparency

3. Performance

4. Extensible and dynamic behavior

5. Naming system architecture

6. Concurrency control

7. Transactions

8. Robust and highly available

9. Versioning

10. Event notifications

11. **International and standardized**

B. Architecture

1. CORBA places an ORB between client and server

a) ORB takes care of marshalling, unmarshalling args

b) ORB takes care of finding objects

c) ORB takes care of starting servers

d) ORB takes care of transactions, events, ...

2. Interface defininition language (IDL)

a) Used to describe objects

b) Language independent

c) Used to generate stubs and skeletons

d) Used to generate definitions for use in programs (header files, etc.)

C. Example

1. Basic 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();
};
typedef sequence<Shape,100> All;
interface ShapeList {
   exception FullException { };
   Shape newShape(in GraphicalObject g) raises(FullException);
   All allShapes();
   long getVersion();
};
```

2. Notes

a) Structs correspond to non-remote Java objects in RMI

b) Syntax is not C/C++/...

c) Remote objects again specified by interfaces

3. **Implementation**
   a) Is language dependent
   b) Is dependent on the IDL translator used

```
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;
      // other initializations
   }
   public Shape newShape(GraphicalObject g)
               throws ShapeListPackage.FullExcepiotn {
      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()       { ... }
}
```

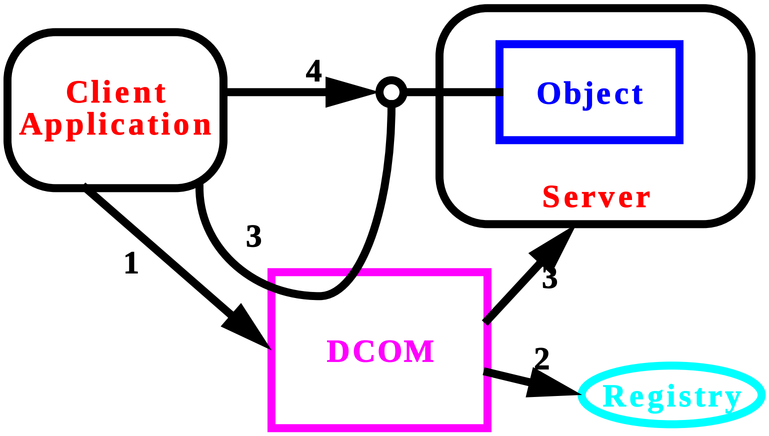
4. **Plus you need a main program for the server and the client**
   a) Use CORBA naming to register the object
   b) Naming is just another remote object
5. **Note similarities to Java RMI**

IV. COM (DCOM, OLE, ACTIVEX)

A. COM views object first, then interfaces

1. **You get a handle to an object**
2. **Then you query what interfaces it supports**
3. **Then you get a handle to one of those interfaces for the object**
4. **Interface = abstract class + object**

B. Using DCOM



1. Client request to create object

a) DCOM looks in registry (all servers must be registered)

b) DCOM locates the implementation

(1) Can be shared library, local/remote server

(2) DCOM starts up server if necessary

c) Factory in server creates object

d) Factory returns interface to DCOM

e) Interface returned to client

2. Interface used to make calls

a) Directly to server

3. Registry

a) Needs to know about interfaces, servers, objects

b) Unique IDs (UIDs) created by each - uuidgen utility

c) Definition file defines the interface

C. Example : Remote calculator

1. Interface Definition

```
[
   uuid(3e47c00e-6bf6-17e1-8514-0800207ebd7f),
   object,
   pointer_default(unique),
   helpstring("Remove calculator sample application")
]
interface ISimCalc : IUnknown {
   import "unknwn.idl";
   HRESULT clear();
   HRESULT enter([in] float value);
   HRESULT add([in] float value);
```

```
      HRESULT sub([in] float value);
      ...
      HRESULT result([out] float * value);
   }
[
   uuid(3e47c00e-6bf6-17e1-8cb2-0800207ebd7f),
   version(1.0),
   lcid(9),
   helpstring("Simple Calculator Demo")
]
```

a) Methods return HRESULT (S_OK or S_FAIL)

b) Out parameters done via pointers

c) Strings done via OLECHAR * (wide strings)

2. Define the library that implements this

```
library SimCalcLib {
   importlib("stdole32.tlb");
   [
      uuid(3e47c00e-6bf6-17e1-9b42-0800207ebd7f),
      helpstring("Simple calculator demo implementation")
   ]
   coclass CSimCalc {
      interface ISimCalc;
   }
}
```

a) This associates an implementation class with interface

3. Define the server

a) Implement CSimCalc as a standard C++ class

```
class CSimCalc : public ISimCalc {
   private:
      double cur_value;
   public:
      CSimCalc();
      ~CSimCalc();
      HRESULT clear();
      HRESULT enter(float value);
      HRESULT add(float value);
      HRESULT sub(float value);
      ...
      HRESULT result(float * value);
};
... { Implementation of these methods }
```

b) Define the actual server

```
class SimCalcServer : public DcomServer {
private:
   DWORD simcalc_obj;
public:
   SimCalcServer();
   const char * serverName() const { return "SimCalc"; }
   HRESULT registerObjects();
```

```
    HRESULT revokeObjects();
    HRESULT registerClasses();
    HRESULT revokeClasses();
};
```

(1) These are implemented using calls to DcomServer

(2) Effectively keep track of the unique simcalc object

(3) Associate its UID with the object

c) Define the main line for the server

(1) Create a SimCalcServer instance

(2) Call its setup and process methods

4. Define the client object

```
class SimCalcClient : public DcomClient {
private:
    ISimCalc * sc_interface;
public:
    SimCalcClient();
    const char * clientName() const { return "SimCalcTest"; }
    ISimCalc * getInterface();
};
```

a) ISimCalc interface is automatically generated from IDL

b) Code for implementing this:

```
return (ISimCalc *) createObject(CLSID_CSimCalc,IID_ISimCalc);
```

5. Use the client

a) call client.setup() method to indicate its host and register

b) Get the interface you want using getInterface

c) Call methods on that interface

D. Notes

1. The calls are generally handled in separate threads

V. Shared Memory

A. Same machine

1. MMAP/SHM primitives

2. Sync primitives work across processes

3. Much like multithreaded programming

B. Going beyond one machine

1. Apollo -- using file-based sharing

a) This worked because there was no cache, processors were slower

2. **Modern implementations are built on message passing**

C. Granularity options

1. **Page -- typically what is done**
    a) Problems with alignment, multiple items/page, etc.
    b) Hardware support via virtual memory
2. **Object**
    a) Work at the object level
    b) Allows for finer grain control, etc.
    c) But doesn't have hardware support

D. Consistency options

1. **Atomic consistency**
    a) Can view each operation as atomic and can order them linearly based on real time of execution
    b) Too difficult to implement efficiently
2. **Sequential consistency**
    a) Can view each operation as atomic and can order them based on relative time within each process
    b) Typically used in most implementations
    c) Still quite expensive
3. **Coherence**
    a) Each process agrees to order of writes on each location
    b) Processes might differ with different locations
4. **Weaker consistency constaints also used**
    a) Consider

E. Update options

1. **Write-update**
    a) All writes to shared memory are make locally and multi-cast to all other replicas
    b) Problems with multicast performance
    c) Order of multicast affects consistency
2. **Write-invalidate**
    a) Single writer or multiple readers

b) Essentially writer needs to get a lock on the page

F. Practical issues

1. While this is a cleaner model, it is difficult to scale

2. Several research systems exist that implement this

a) In some cases can match performance of message passing

b) But generally not

c) And this works only for limited numbers of processors

3. Can be a simpler way of programming however