CIS 630 - Fall 2004
Distributed Systems

Lecture 4
Distributed Objects and Remote Invocation

University of Oregon
Department of Computer and Information Science
Business and Logistics

☐ Course webpage
  ◆ Updated bibliography

☐ Discussion papers
  ◆ First two papers are selected
  ◆ See logistics and assignments webpage

☐ Term project
  ◆ Send team member preferences
  ◆ Send project interests

☐ Oct. 14 lecture cancelled
  ◆ Extra office hours: Oct. 15, 12:00 - 2:00 pm
Acknowledgements

- Some RPC slides derived from lecture materials by Dr. Paul Krzyzanowski, Rutgers University
- Some XML-RPC slides derived from lecture materials by Dr. Marc Conrad, University of Luton
Lecture Objectives

☐ To understand remote procedure call and its implementation

☐ To study communication between distributed objects and the integration of remote method invocation into a programming language

☐ To be able to use Java RMI to program applications with distributed objects and how to use some aspects of reflection

☐ To study the extension of the event-based programming model to apply to distributed event-based programs
Heterogeneity

- Applies to all of the following:
  - networks
  - computer hardware (e.g., data type representation)
  - operating systems (e.g., system calls)
  - programming languages (e.g., data structures)
  - implementations by different developers
- Middleware provides a programming abstraction and masks the heterogeneity of networks etc.
Middleware Layers

- Higher-level distributed systems programming abstractions built on underlying protocols
- Supports transparency

- Applications
- RMI, RPC and events
- Request reply protocol
- External data representation
- Operating System
Programming Models for Distributed Applications

- Distributed cooperating programs (processes)
- Extend programming models to distributed programs
  - Remote procedure call (RPC) model
    - Client programs call procedures in server programs
  - Object-based remote method invocation (RMI)
    - Extends local method invocation (generic)
    - RPC can be understood in relation to RMI
  - Event-based programming model
    - Notification of registered distributed events
- Middleware application programming interfaces (APIs)
  - Location transparency and independence
Interfaces

- Programming languages provide for modularization
  - Program is composed of modules
- Interaction between modules is specified through the notion of an interface
  - Calling conventions between modules
  - Procedures and variables
- Module implementation and information about modules is hidden behind the interface
- Only the interface is known externally
Interfaces in Distributed Systems

- Modules run in separate processes (think abstractly)
- Cannot specify direct access
  - To procedures or variables
  - Standard module calling conventions not possible
  - Interface has two parts: local and remote
- Need to think of in term of input and output data
  - Need to describe the data and pass local data
- Service interfaces
  - Provide set of remote procedures for clients
- Remote interfaces for distributed objects
So What’s the Problem with Sockets

- Sockets interface is straightforward
  - `connect()`, `read()`, `write()`, `disconnect()`
- Forces read/write mechanism
  - Not how we generally program
  - Usually call a procedure call or object methods call
- Desire to make distributed computing look more like “centralized” computing
  - I/O is not the model to use
- Variables usually defined with respect to service
Remote Procedure Call (RPC)

- Birrell and Nelson, 1984
- Mechanism to call procedures on other machines
- Process on machine A calls procedure on machine B
  - A is suspended
  - Execution continues on B
  - When B returns, control passed back to A
  - Voila! Remote procedure call!
- Goal: it should appear to the programmer that a normal call is taking place
Local Procedure Calls

Procedure call steps:
1. Prepare for call
2. Call
3. Entry to f()
4. Prepare to return
5. Return
Implementing RPC

☐ No architectural support for remote procedure calls
  ☐ “Simulate” it with the tools we have
  ☐ What do we have?
    ➢ local procedure calls

☐ Simulation makes RPC a language-level construct
  ☐ Instead of an operating system construct

☐ The trick:
  ☐ Create stub functions
    ➢ makes it appear to the user that the call is local
  ☐ Stub function contains the function’s interface
Role of Client/Server Stub Procedures in RPC

![Diagram showing the role of client/server stub procedures in RPC communication.]

- **Client Process**: The client stub procedure calls the client program.
- **Communication Module**: Handles the request and reply communication.
- **Server Process**: The server stub procedure calls the service procedure.
- **Request and Reply**: The communication process is facilitated between client and server stub procedures.
Stub Functions

- Client calls stub (parameters on stack)
Stub Functions

- Stub marshals parameters to network message

Diagram:

- Client functions
  - Client stub
  - Network routines

- Server functions
  - Server stub (skeleton)
  - Network routines

Client

Server
Stub Functions

- Network message sent to server
Stub Functions

- Receive message: send to stub

Client functions → Client stub → Network routines

Server functions → Server stub (skeleton) → Network routines

Client → Server
Stub Functions

- Unmarshal parameters and call server function
Stub Functions

- Return from server function
Stub Functions

- Marshal return value and send message
Stub Functions

- Transfer message over network
Stub Functions

- Receive message: direct to stub
Stub Functions

- Unmarshal return and return to client code
Benefits

- Procedure call interface
- Writing applications is simplified
  - RPC hides all network code into stub functions
  - Application programmers do not have to know details
    - sockets
    - port numbers
    - byte ordering
  - All taken care of by RPC
- We regard RPC as existing at the \textit{presentation layer} in OSI model
RPC Has Issues

☐ Pass by value
  ❍ Easy case
  ❍ Just copy data value to network message

☐ Pass by reference
  ❍ Does not make sense without shared memory
  ❍ Need to simulate
    1. copy items referenced to message buffer
    2. ship them over to the server
    3. unmarshal data at server
    4. pass local pointer to server stub function
    5. send new values back (assuming they were modified)
Complex Structures and Data Representation

- Complex structures must be marshalled into pointerless representations
  - Reconstruct with local pointers on other end
- Data representations may be incompatible
  - Different byte ordering
  - Different sizes for integers and other types
  - Different type representations (e.g., floating point)
  - Different character sets
  - Alignment requirements
- Neet standard encoding to enable communication between heterogeneous systems
Where to Bind?

- Need to locate host and correct server process

- Solution 1
  - Maintain centralized DB that can locate a host that provides a particular service (Birrell proposal)
  - Server sends message to central authority stating its willingness to accept certain remote procedure calls
    - provides port number
  - Clients then contact this authority when they need to locate a service
    - retrieve necessary information to bind
Where to Bind?

Solution 2

- Require client to know which host it needs to contact
- Server on that host maintains a DB of locally provided services

Tradeoffs

- Solution 1 is more flexible and adaptable
- Solution 2 may be necessary when services are replicated, but their use is different
  - identical file servers serving different file systems
When Things Go Wrong

- Local procedure calls do not fail
  - Unless the system fails
- More opportunities for failure in distributed systems
  - Server could generate an error
  - Problems in the network
  - Server crash
    - Client might disappear while server executes its code
- Transparency breaks in these situations
- Application should be prepared to deal with RPC failure
Semantics and Failure

- Like a local procedure, remote procedure calls must be executed only once.
- Exactly once may be difficult to know and achieve.
- A remote procedure call may be called:
  - 0 times
    - server crashed or server process died before executing server code.
  - 1 time
  - 1 or more times
    - excess latency or lost reply from server and client transmission results in repeated calls.
RPC Semantics and Issues

- Most RPC systems will offer either:
  - *At least once* semantics
    - allows for the possibility of multiple calls
  - *At most once* semantics
    - if RPC takes place, it only happens once
- Application must understand semantics
  - *Idempotent* functions - may be run any number of times without harm
  - *Non-idempotent* functions - have side-effects
- RPC is slower in performance
- Security concerns arise with authentication
Programming with RPC

- Lack of language support
  - Not supported directly in most programming languages (C, C++, Java, …)
  - Standard compilers will not generate stubs

- RPC compiler
  - Use a separate pre-compiler to generate stubs
  - Link with RPC library
  - Need to describe the interfaces
Interface Definition Language

- Allow programmer to specify remote procedure interfaces
  - Names, parameters, return values
- Pre-compiler uses to generate client and server stubs
  - Marshalling code
  - Unmarshalling code
  - Network transport routines
  - Conform to defined interface
- Similar to function prototypes
RPC Compiler

IDL

RPC compiler

client code (main)

client stub

data conversion

headers

data conversion

server skeleton

server functions

compiler

client

server

compiler
Writing the Program

- Client code has to be modified
  - Initialize RPC-related options
    - Transport type
    - Locate server and service

- Server functions
  - Generally need little or no modification
What kind of services does an RPC system need?

- Name service operations
  - Export/lookup binding information (ports, machines)
  - Support dynamic ports

- Binding operations
  - Establish client/server communications
  - Use appropriate protocols and endpoints

- Endpoint operations
  - Listen for requests
  - Export endpoint to name server
RPC API (continued)

- Security operations
  - Authenticate client and server
- Internationalization operations
- Marshalling and data conversion operations
- Stub memory management
  - Dealing with “reference” data
  - Need temporary buffers
- Program identity operations
  - Allow applications to access IDs of RPC interfaces
XML RPC

- Use XML as a common language to transmit data
- It is a specification and set of implementations
  - Allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet
- It is remote procedure calling
  - Uses HTTP as the transport
  - Uses XML as the encoding
- XML-RPC is designed to be as simple as possible
  - Supports complex data structures
XML-RPC Home Page

“Does distributed computing have to be any harder than this? I don’t think so.” — Byte.

What is XML-RPC?

It's a XML and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.

XML-RPC Directory
- Specifications
- Implementations
- Services
- Blogging APIs
- Communities
- Tutorials/Press

The XML-RPC community

The implementations page lists the accomplishments of the community, a set of compatible XML-RPC implementations that span all operating systems, programming languages, dynamic and
Why XML-RPC?

- Specification is small
  - < 1800 words and easy to learn with many examples
  - Lightweight
- Many implementations
  - C/C++, Java, Perl, Python, Lisp, PHP, .NET, Real Basic, Tcl, …
- Uses existing protocols
  - HTTP
- Data structures
  - Integer, boolean, string, double, structs, arrays
  - Base64 binaries
XML-RPC in Java

- Java package to implement XML-RPC
  - `org.apache.xmlrpc`
  - Provides classes to implement
    - XML-RPC client
    - XML-RPC server
A Java Client using XML-RPC

```java
import java.util.*;
import org.apache.xmlrpc.*;

public class JavaClient {
    public static void main (String [] args) {
        try {
            XmlRpcClient server = new XmlRpcClient("http://localhost/RPC2");
            Vector params = new Vector();
            params.addElement(new Integer(17));
            params.addElement(new Integer(13));
            Object result = server.execute("sample.sum", params);
            int sum = ((Integer) result).intValue();
            System.out.println("The sum is: " + sum);
        } catch (Exception exception) {
            System.err.println("JavaClient: " + exception);
        }
    }
}
```
What does the client send to the server?

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
    <methodName>sample.sum</methodName>
    <params>
        <param>
            <value><int>17</int></value>
        </param>
        <param>
            <value><int>13</int></value>
        </param>
    </params>
</methodCall>
```
import org.apache.xmlrpc.*;

public class JavaServer {
    public Integer sum(int x, int y) {
        return new Integer(x+y);
    }
    public static void main (String [] args) {
        try {
            WebServer server = new WebServer(80);
            server.addHandler("sample", new JavaServer());
            server.start();
        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception);
        }
    }
}
Server to Client Transmission

- What does the server send to the client?

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
  <params>
    <param>
      <value><int>30</int></value>
    </param>
  </params>
</methodResponse>
```
SOAP (Simple Object Access Protocol)

- SOAP is another protocol for client/server applications
- General principle is similar to XML-RPC
  - Uses XML as a common language
- Promoted as lightweight
  - Specification is >77,000 words
- Defines rules for
  - Encapsulating data being transferred (data encoding)
  - Invoking remote methods
<?xml version='1.0' encoding='UTF-8'?>
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getCandidateKeys xmlns:ns1="urn:examples:keyservice" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <fdSet xsi:type="xsd:string">{{a}-&gt;{b}}</fdSet>
    </ns1:getCandidateKeys>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Example Service Reply

```xml
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
   <ns1:getCandidateKeysResponse xmlns:ns1="urn:examples:keyservice"
      SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <return xsi:type="xsd:string">{{a}}</return>
   </ns1:getCandidateKeysResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
Distributed Objects and Remote Invocation

- Objects that can receive remote method invocations (RMIs) are called *remote objects*.
- They implement remote interfaces:
  - Different semantics from local calls.
  - Code for marshalling and unmarshalling arguments and sending request and reply messages.
  - Generated automatically by an interface compiler.
- RPC is to RMI as procedure call is to object invocation (RPC is a degenerate form of RMI).
- Distributed event-based systems allow objects to subscribe to remote events and be notified.
Object Model

- OO programs consist of a collection of interacting objects, each with set of data and set of methods
- Objects encapsulate their data and method code
- Objects communicate by invoking methods and passing parameters
- Objects accessed via object references
  - 1st class values that can be assigned, passed, returned
- Interfaces provide definition of method signature
  - Argument types, return values, exceptions
  - Does not specify implementation
Object Model (continued)

- Actions are initiated by object method invocation
- Exceptions provide way to deal with error conditions
  - Throw exception on unexpected condition or error
  - Catch exception by handler code
- Garbage collection deals with problem of freeing space when objects are no longer needed
  - Manual or automatic
Distributed Objects

- OO programs are logically partitioned
  - Physical distribution into different processes and onto different computers is a natural extension
- Distributed object systems may adopt client-server architecture or other architectural models
- Objects encapsulate state and state of a distributed computation is captured as the collection of objects
- Objects access only via their methods
  - Location of data is not restricted
  - Heterogeneous systems can use different data representations and storage
Distributed Object Model

- Method invocations can be local or remote
- *Remote objects* can receive remote invocations
- Fundamental concepts
  - *Remote object reference*: methods can be invoked if have access to distribute identifier for unique remote object
  - *Remote interface*: specifies which of its methods can be invoked remotely
  - Actions are initiated by method invocation, which may cause a chain of local/distributed actions to occur
  - Garbage collection and exceptions extended
Distributed Object Model

- Processes contain objects
  - Local and remote objects
- How do objects invoke methods on remote objects?
  - Need to know the *remote object reference*  
  - How do they get it?
- *Remote interface* specifies which methods can be invoked remotely
Interface Definition Languages

- Notation for defining interfaces
- Allows input and output parameters to be mapped onto the language’s normal use of parameters
- Could be added to an OO language (e.g., Java RMI)
  - Distributed applications written in single language
- Need solution for multi-language applications
  - Interface definition languages (IDLs)
  - Allows different languages to interoperate
    - Objects can invoke one another
  - Describe each method parameter as *input* or *output*
  - CORBA IDL, Sun XDR, DCOM IDL, DCE IDL
Remote Object and Its Remote Interface
Remote Method Invocation (RMI) Semantics

☐ Reliability of remote invocation determines semantics

☐ *Maybe invocation semantics*
  - Invoker cannot tell whether a remote method has been executed once or not at all
    - Omission failure if invocation or result message lost
    - Crash failures when remote object server fails
  - Useful only for apps where failures are acceptable

☐ *At-least-once invocation semantics*
  - Invoker receives a result knowing the method was executed at least once
  - Invoker receives an exception that no result received
RMI Semantics (continued)

- Semantics can be achieved by retransmission of request messages
  - Crash failures due to remote server failure
  - Arbitrary failures can cause multiple method invocation

- At-most-once invocation semantics
  - Invoker receives a result knowing that the method was executed exactly once
  - Invoker receives an exception informing that no result was received, in which case the method will have been executed either once or not at all
  - Achieved by using all fault tolerance measures
  - Achieved by using all fault tolerance measures

Java RMI and CORBA invocation semantics AMO
Transparency

- RPC transparency requires all necessary calls to marshalling and message-passing procedures to be hidden.
- Transparency for distributed objects requires also hiding task of locating and contacting a remote object.
- Remote invocations are more vulnerable than local:
  - Object must be able to recover from failures.
  - Latency issues complicate timeout policies.
- Consensus for remote invocation transparency:
  - Syntax is same as for local invocation.
  - Differences express in interfaces and their definitions.
Implementation of RMI

- Two cooperating communication modules carrying out request-reply protocol (request/reply messages)
  - Responsible for specified invocation semantics
- Remote reference module
  - Translate between local and remote object references
  - Creating remote object references
  - Uses a remote object table mapping local object references and remote object references
    - Entry for all the remote objects held by process
    - Entry for each local proxy
  - Called by components of RMI software
Role of Proxy and Skeleton in RMI

- Object A
- Proxy for B
- Object B
- Skeleton & dispatcher for B’s class
- Remote object B
- Remote reference module
- Communication module
- Request
- Reply
RMI Software

- Layer between app-level objects and communication and remote reference modules
  - Proxy in client
    - Make remote method invocation transparent to clients
    - Class of a proxy implements the methods in the remote interface of the remote objects it represents
    - Each method of proxy marshals a reference to target
  - Dispatcher in server
    - Selects appropriate method in skeleton for request
  - Skeleton in server
    - Implements methods in remote interface
Implementation of RMI (continued)

- Unmarshals arguments in request message
- Invokes remote object methods
- Waits for completion and marshals results/exceptions

- Generation of classes for proxies/dispatcher/skeleton
  - Generated automatically by interface compiler
  - Set of methods offered by a remote object for Java RMI defined as a Java interface implemented within the class of a remote object
  - Java RMI compiler generates proxy, dispatcher, and skeleton classes from the class of the remote object
Implementation of RMI (continued)

☐ Server and client programs
  ☐ Server
    ➢ Classes for dispatchers and skeletons
    ➢ Implementations of the classes of the remote objects
    ➢ Remote object initialization
  ☐ Client
    ➢ Classes of the proxies for all remote objects
    ➢ Use binder to look up remote object references
    ➢ Use factories to create remote objects

☐ Binder
  ☐ Separate service to map from names to remote object references
Implementation of RMI (continued)

- Server threads
  - Servers generally allocate a separate thread for the execution of each remote invocation

- Activation of remote objects
  - *Active* object: available for invocation in process
  - *Passive* object: not current active but can be made so
  - Make active by creating a new instance of object class and initializing its instance variables

- Persistent object stores
  - Object guaranteed to live between activations of processes