Ser321 Principles of Distributed Software Systems

4. Distributed Objects: RMI, JsonRPC, and .NET Remoting

• Section 4.a Distributed Objectives: Outcomes and References
• Section 4.b Introduction to Distributed Objects
• Section 4.c Distributed Objects in Java with RMI
• Section 4.d RMI and Dynamic Class Loading
• Section 4.e Json Remote Procedure Call (Json-RPC)
• Section 4.f .NET Remoting
4.a  Distributed Objectives: Outcomes and References

4.a.1  Outcomes

• What is **Remote Method Call**, and how can I develop distributed applications using **Remoting Object Servers** and their **Clients** in Java, C#, and other languages?

• **Section Outcomes**
  - To understand the motivation for CORBA, .NET Remoting, RMI, and JsonRPC.
  - To be able to use the basic classes provided by **Java RMI** to create and communicate among distributed objects
  - To understand **dynamic class loading** in Java, RMI **overhead**, simple **name registration** and **call-backs** with RMI.
  - To understand **JsonRPC** and its use in distributed applications.
  - To understand the differences between **SingleCall**, **Singleton** and **Client-Activated** remote object servers and the distributed applications best suited for each. To be able to explain how **C# .NET** utilizes each.
  - To understand limitations on and handling of parameter and return types for **.NET remoting** and **Java RMI** applications.
4.a.2 References

- **Readings for Java RMI**

- In [Java Network Programming and Distributed Computing](http://java.sun.com/docs/books/tutorial/rmi/index.html) Text:
  - Chapter 11, Remote Method Invocation.

- .NET Remoting and Windows Communication Foundation Readings
  - C# for Java Developers by Jones and Freeman, Microsoft Press, Chapter 15

- **JsonRPC** (Json Remote Procedure Call)
4.b Introduction to Distributed Objects

4.b.1 What are Distributed Objects?

• **Distributed objects** as realized through .NET Remoting and RMI allow components of distributed applications to communicate at the object level.
  - Components, which may reside in different processes and/or on different machines are viewed as separate executing parts of an application.
  - Example components are **browser** and **http server**, **email client** and **imap server**, as well as their subparts.
  - Components send each other messages in an object-oriented way, consisting of a method to be invoked together with arguments for the method. The result of a message may be a return value.
  - **Arguments**, call **synchronization** and **return values** are all **handled by platform**; not by the application developer.

• Without distributed objects, sending a message to a remote machine:
  - Current **protocols** (http, ftp, telnet) use **stream sockets** with **data protocols**, or RPC
  - Current protocols are data centric, difficult to maintain, enhance, and use.
4.b.2 Vision of Distributed Object Computing

- Applications are **federations** of cooperating distributed objects.
  - Configuration may be **dynamic** - objects can discover each other and communicate to perform some service for each other.
  - Distributed Objects can **survive network outages** and/or follow **mobile paradigms** where connectivity is constantly changing.
  - Distributed Objects may be **service providers** that register themselves with multiple registries. Potential clients may contact the registry and discover information about compatible (and maybe competing) services.

- View the Internet as an **Object Bus**

- Distributed Object technologies remove the underlying **middleware** issues from applications and their developers.
  - How to implement message passing and argument synchronization?
  - What if objects are not “in-process”?
  - Distinction between objects within LAN versus WAN
4.b.3 The Distribute Object Landscape

- **Remote Procedure Call (RPC)**
  - Programmer generates a remote call similar to a local procedure call. Translate argument (and return values) using an external data representation. Doesn’t fit well with object-oriented systems.

- **.NET Remoting** by Microsoft
  - Multi-language, single platform, constrained functionally by not including complete object model

- **Common Object Request Broker** by Object Management Group
  - Multi-language, multi-platform. Sacrifice ease of use and functionality for multi-language capability. Lagging Pass-By-Value implementations (serialization) hinders usage.

- **Remote Method Invocation** in SUN/Oracle Java
  - Single language, multiple platform. Easy to use and conceptually clean, although with automatic dynamic class loading security must be carefully addressed. Incurs burden of net-wide reference counting and garbage collection. Lots of Intranet application including J2EE and other web server-specific support applications.
4.c Distributed Objects in Java with RMI

4.c.1 Java Remote Method Invocation

- Objects are virtually in the same address space

- **Client side** uses a **proxy** object (**Stub**) with the same interface as the server, which handles the connection.

- **Server side** uses a **skeleton**, which listens for requests and transfers them to the **Server Object**.

- Implementation of the connection between the **Stub** and **Skeleton** is provided with Stream Sockets.
4.c.2 Typical RMI Client Server Application

- **Key Components**

- **Clients** locate remote objects through a **Registry (name server)**,

- Clients send messages to remote objects in the **Object Server**

- Communication of argument and result objects may cause bytecode classes to be downloaded into either **Client** or **Object Server** via a **Web Server**
4.c.3 Steps to Creating an RMI Remote Object and Client

- **Define an interface** that extends `java.rmi.Remote` (no methods to implement in the resulting class).

- Create a class that **implements** the remote interface and **extends**
  - `Activatable` or `UnicastRemoteObject`
  - Can also explicitly export the object rather than through **extends**

- Generate a main program to **construct** an **object** of your remote class
  - A main may create one or more objects and register them with a name service (`rmiregistry`)
  - An object may create remote objects and may or may not register them with a name service

- Create and compile a Client for the remote object.
  - Initially uses a name server to get a reference to a remote object

- Compile the **Remote Object Server** and **Client** using `javac`

- Create **stub** and **skeleton** classes by calling `rmic` on the remote class (these can also be created automatically and dynamically by the runtime).

- Start the naming service, server and client
4.c.4 Remote Java Objects

- Remote classes may extend either
  - `java.rmi.activation.Activatable`
  - `java.rmi.server.UnicastRemoteObject` (URO)

- A remote class may extend another class, but such a class must:
  - implement the `java.lang.Object` methods
  - call one of the `exportObject` methods in `Activatable` or `URO`

- **Activatable** - objects that require persistent access over time and can be activated by the system.

- **URO** - objects whose references are valid only while server is active.

- **Remote objects**
  - can get the hostname of their Client through `Activatable` or `URO`
  - can be created by other remote objects at any time
  - are not thread safe unless explicitly synchronized
  - don’t all need to be registered with a name server
4.c.5  RMI Method Parameters and Return Values

- Valid Parameter and Returns are
  - any Java object that implements serializable (call by value)
  - primitive types (int, boolean and so on)
  - remote Java objects (call by reference)

- Classes, for parameters or return values, that are not available locally are downloaded dynamically by the RMI system, with certain restrictions
  - RMI uses object serialization to transmit data
  - The RMI system annotates the call stream with information so that the class definition files can be loaded at the receiver; codebase, for example.

- Marshalling refers to the process of “streamifying” parameters (or return values) for transport

- Parameters are unmarshalled on the receiving end.
4.c.6 How Remote and Local Objects Differ

- **Clients** of remote objects **use remote interfaces**, not with the Implementation classes for those interfaces.
- Non-remote **Object arguments** to, and results from a method call, **are passed by copy** (serialization) rather than by reference.
- **Remote objects** as arguments **are passed by reference**, not by serializing the object.
- Semantics of **java.language.Object** are specialized for remote objects.
- The additional complexity of remote method calls results in **additional exceptions** that may occur as a result of messages to remote objects.
4.c.7 Example RMI Employee Server

- Here is an example of RMI that uses the RMI Registry to register a Unicast Remote Object (URO) server for Employee information. The server method \( \text{getEmp(int)} \) returns a serializable employee \( \text{Emp} \) object.

  - While running, the Java runtimes and RMI related calls are:

    - See the example jar file:
      - \textbf{employee.jar}
      - Start the server with: \textit{ant server}
      - Start the client with: \textit{ant client}
4.c.8 Points to Note in Reading the Example

- **Emp** is *serializable* and it is returned to the client by the server.
  - The client calls `server.getEmp(int)`, which returns an **Emp** object.
  - In which JVM does the implementation of `getEmp` execute?
  - When the client subsequently calls the **Emp getName** method, in which JVM does it execute?

- The client uses the **EmpServer** interface and not the **EmpServerImpl** class to define a remote object reference.
  - **EmpServer server = (EmpServer)Naming.lookup(“StringRmiurl”);**

- Notes about using Ant and the build.xml to build and execute the project.
  - Both **client** and **server** are loaded from the same codebase. Not typical of a real distributed application. Refined in subsequent examples.
  - See the **server** target: Ant uses `parallel, sequential, sleep` and `exec` tasks to initiate **RmiRegistry** and the server **EmpServerImpl**.

- General organization of an RMI application.
  - Server extends URO and implements a Remote interface, registers itself.
  - Client gets the **remote object reference** through a **name service lookup**.
4.d  RMI and Dynamic Class Loading

4.d.1  Why Would RMI Need to Download a Class File?

• Suppose a client makes a remote call to a method with signature:

  public Person getStudent (String id)  //Person is serializable

  - Although the Person class may be resident in Client VM
  - Server may return a Student object (class Student extends Person)

• The client must have access to the Person class to compile

• Since Person is serializable, any messages the client sends to the object that is returned by the object server will execute on the client.

ClassServer server = (ClassServer) Naming.lookup("/localhost:2222/ClassServer");
Person tim = server.getStudent("Tim");
System.out.println("My schedule is: " + tim.getSchedule());

- ClassServer can return a Student object since it extends Person
  - The Student class may overload getSchedule to providing special semantics depending on the fact that Tim is a student.

- Person is serializable, so getSchedule executes in the client VM

- RMI System must assure that Student.class gets downloaded to client if its not already available there.
4.d.2 Resolving Classes for Received Objects

- **Unmarshalling** attempts to resolve classes by name in its local class loading context

- RMI provides a facility for dynamically loading the class definition of classes not already available in the receiving Virtual Machine
  - remote stub classes corresponding to remote objects
  - any other class whose object is passed by value (serializable object), such as a subclass of a declared parameter or return type

- Every class descriptor written to an RMI marshal stream, includes a string codebase URL path (a space-separated list of URLs) from which the receiver could download the class definition file.

- In unmarshalling if the class isn’t found locally:
  - The **resolveClass** method reads the codebase and class name from the stream.
  - **RMIClassLoader.loadClass** is used with the URL codebase and class name to obtain the class.
4.d.3 The RMI System and Dynamic Loading

- RMI system will **automatically load class files**:
  - When it needs to satisfy return values and parameters for messages to remote objects.
  - When a **client** receives a remote object reference, the stub for the implementing class will be loaded through the rmi codebase.
  - When the rmiregistry receives a **new remote registration**, it must load all associated classes.

  - **New registration classes include**: the remote interface, implementing class stub and any related classes for method parameters or returns.

- Programmer can **explicitly load class** files dynamically

- **Applets** are an example of dynamically loaded class files,
  - applets run in a browser’s virtual machine.
  - applets must extend the awt Applet class
  - applets are otherwise constrained by the browser (focus change)
4.e Json Remote Procedure Call (Json-RPC)

4.e.1 Benefits

• **Json (Java Script Object Notation)**
  - **Lightweight**: Text-based format for externalizing/internalizing (serializing/de-serializing) objects. Generally very compact as compared to other similar formats (SOAP, XML, even some binary language specific formats.)
  - **Multi-Language**: Support for most popular programming languages.
  - **Interoperability**: Supports exchange of objects among programs written in different languages by virtue of multiple implementations to the same specification.

• **RPC (Remote Procedure Call)**
  - Been around “forever”
  - **Lightweight**: Does not carry baggage of significant framework overhead, or overhead of managing object life-time or other aspects of object frameworks.
  - **Multi-Language**: Has been implemented for nearly every popular programming language.
4.e.2 JsonRPC Format

• What is it, and how does it work? See: http://www.jsonrpc.org
  - A Json Object is created by the client’s proxy to represent a method call with its arguments. For example the call `proxy.plus(25.3, 2.2)` would be:
    ```json
    { "jsonrpc":"2.0", "method":"plus", "params":[25.3, 2.2], "id":3}
    ``
  - The client’s proxy sends the Json request object to the server’s proxy (sometimes called skeleton), typically via either http, or direct TCP/IP stream-based sockets.
  - The server’s proxy parses (de-serializes) the request, calls the appropriate method with the arguments, and gets the result. With the result, it creates a JsonRPC Response. For example, the request above would result in:
    ```json
    { "jsonrpc":"2.0", "result":27.5, "id":3}
    ``
  - The server sends the response (by responding to the http post, or writing back and completing the stream-based socket connection) to the client who parses (de-serializes) the response object to get the result of the call.

• Client and Server Proxies
  - On the client-side, it implements the interface (methods) of the server.
  - On the server-side, it acts like a threaded server that calls the methods.
4.e.3 JsonRPC Framework Support

- Implementations exist in many languages that isolate the distributed application developer from most of the Json and communication/synchronization overhead.

- JsonRPC Frameworks dynamically or statically create the proxy and manage creating, communicating, and de-serializing request and response objects.

- In C++, for example, one implementation (libjson-rpc-cpp) can be found at: https://github.com/cinemast/libjson-rpc-cpp

- Other implementations may be found from the page: http://json-rpc.org/wiki/implementations

- Or from the JsonRPC Wikipedia page: https://en.wikipedia.org/wiki/JSON-RPC

- Here are some examples of C++ and Java applications using this protocol.
4.e.4 **libjson-rpc-cpp** usage is depicted in the diagram below:
4.f .NET Remoting

4.f.1 ASP.NET Web Services versus .NET Remoting

- .NET Framework includes **two mechanisms for distributed application** development:
  - ASP.NET **Web Services**
  - .NET **Remoting**

- **Remoting** can use different transport mechanisms
  - A native binary TCP based transport protocol is used for LAN
  - **SOAP** and **HTTP** are used to go through a firewall
  - Object references and a proxy object are utilized by the client
  - Remoting is a fast (binary formatted remoting is twice as fast as web services) solution for .NET clients and servers (no interoperability).

- **ASP.NET Web Services** expose web methods using **SOAP** and **HTTP** through an IIS web server.
  - Developed based on open standards (**WSDL**, **SOAP**)
  - Intended to allow heterogeneous development platforms (for example, Java client accessing an ASP.NET Web Service).
4.2.2 Calling a Remote Object’s Methods

- Client method calls to a remote object resolve to the **proxy object**.
  - The proxy object directs the call through the **Remoting layer**, which marshals the parameters and communicates all call information to the receiving application domain through a defined channel.
  - On the object server side, the **Remoting layer** unmarshals the call and arguments, calls the method and then marshals and returns the results.

---

**Diagram:**

```
Application Domain

Client

Proxy

Remoting Layer

Channel

Remote Object Server

Remoting Layer
```

Principles of Distributed Software Systems © T. Lindquist 2017  
January 2017
4.f.3 Types of .NET Remote Objects

- **Server-Activated** remote objects - also called **Well Known Objects**
  - The client must have a **URL naming the object** (well known to client)
  - A server may create and register the remote object, associating it with either a TCP or HTTP channel for communication.

- Two types of **Server-Activated** objects
  - **Singleton object** - one object exists and is used to handle requests from all clients. Used when one client’s information may be displayed in another, such as a chat server. **Hard to scale** across multiple servers.
  
  - **Single Call object** - a new object is created to handle each client call. Used for **stateless** remote objects where an isolated service is to be performed for each client call. For example calculating a function, such as the energy consumed by an air conditioner.

- **Client-Activated objects** - A single object is created and used to service multiple calls from a single client.
  
  - Used to retain client **session** information through multiple calls, such as shopping cart applications.
4.f.4  Publishing a Server-Activated Singleton

• Either (or both) HTTP and TCP can be used to communicate between the client and the server.

  - **Http channels** use **SOAP** and have slower performance. Use Http when clients and server are located on different networks. Http channels have specific support for using proxies. **SOAP** over **HTTP** makes traversing Internet firewalls easier.

  - **Tcp channels** use binary encoding which offers a performance benefit but is less likely to be successfully transmitted across all Internet nodes.

• A single Application Domain may define more than one channel

  - For Example, publishing a server-activated singleton Group Server:

    TcpChannel tcpChan = new TcpChannel(8888);
    ChannelServices.RegisterChannel(tcpChan);
    HttpChannel httpChan = new HttpChannel(8889);
    ChannelServices.RegisterChannel(httpChan);
    RemotingConfiguration.RegisterWellKnownServiceType(typeof(GrpMgr),
    “GrpSrv”,WellKnownObjectMode.Singleton)

    ...

  - A client may obtain a reference to the group manager through **TCP** with:

    GrpMgr gm=(GrpMgr)Activator.GetObject(typeof(GrpMgr),”tcp://localhost:8888/GrpSrv”)
4.f.5 Example Pencil Server

• Suppose we have a pencil server. Clients share pencils by contacting the server to borrow or return some number of pencils.

• Create the server implementation class as a library.
  - See: PencilServerImpl.cs
  - csc /t:library /out:PencilServer.dll PencilServerImpl.cs

• Create an application to register and initiate a single server object
  - See: RegisterPencilServer.cs
  - csc /reference:PencilServer.dll RegisterPencilServer.cs

• If you don’t want to distribute the details of server implementation to all clients, create a Proxy class (PencilServerProxy.cs) using the soapsuds tool
  - soapsuds -ia:PencilServer.dll -gc

• Create a client application, that is compiled with the Proxy
  - See: PencilClient.cs
  - csc PencilClient.cs PencilServer.cs
4.f.6 Executing the Pencil Example

- Download the Ant and Mono based project jar file, extract and execute.
  - The example: `dotNetRemotingPencil.jar`
  - Build all executables: `ant build`
  - `ServerActivatedPencilServer.exe` will take either `singleton` or `single-call` as a command line argument, creating the selected type of server.
  - **Observe:** When you register a singleton, only 1 instance is created and it is shred by all clients. With single-call, a new instance is created for each method call.
  - In separate prompts run `PencilClient.exe` or `PencilClientWinApp.exe`
  - `ClientActivatedPencilServer.exe` registers a `session-based` server.
  - Use the same clients, but giving appropriate command line arguments to demonstrate session clients.
  - The default **life-cycle** for the server is 5 minutes.
### 4.f.7 More About .NET Remoting

- **Client-Activated** and **Server-Activated** objects aren’t active forever, unless explicitly specified this way by the developer.
  - Default idle period is 5 minutes. Released after 5 minutes of inactivity.
  - A new object is instantiated when a client subsequently calls a method.
  - Leases are used to manage lifetime, avoiding network wide reference counting of remote object references, which is done by RMI.

- What are the valid parameter and return types for Remote Object Calls?
  - **Primitive values**, such as int and boolean.
  - **.NET defined Serializable objects** are passed by copy/value, such as **String** or **Array** classes. They are serialized upon **marshalling** and deserialized when **unmarshalled**.
  - **Remote objects**. Classes that extend **System.MarshalByRefObject**
  - **User-Defined Serializable** or **remote classes**. Must provide class information to the client. For serializable, the entire class must be available to the client. For remotes, the client needs access to the proxy.