CS 555: DISTRIBUTED SYSTEMS
[RPC & DISTRIBUTED OBJECTS]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey
Topics covered in this lecture

- RPC
- Distributed Objects

RPC Mechanism
What makes RPCs tick?

- Blissful ignorance at the clients
- Remote services accessed by making ordinary procedure calls
  - No need for send and receive primitives

Parameter passing

- Stubs *pack* and *unpack* request parameters
- Packing parameters into a message
  - Parameter *marshaling*
Challenges in passing parameters in distributed systems

- Multiple machine *types* are present
- Each has its own representation for
  - Numbers, characters, etc.
- Examples
  - IBM mainframes use EBCDIC character codes
    - PCs use ASCII
  - Integer representations
    - One’s or Two’s complement

The endian issue

- Big endian: SPARC
  - Number their bytes from left to right
- Little endian: INTEL
  - Number their bytes from right to left

An example of two parameters: 32 bits
(a) Integer - 5  (b) 4 character string JILL

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>7</td>
<td>L</td>
<td>6</td>
<td>I</td>
</tr>
</tbody>
</table>

On INTEL: Little endian

How are messages sent?

- Messages are transferred byte-for-byte
- First byte sent is the first to arrive
- Big-endian is typically the convention in data networking (including IPv6)
  - Network byte order
Message sent from INTEL to SPARC

On INTEL:
Little endian

The integer is now $5 \times 2^{24}$
The String is correct “JILL”

On SPARC:
Big endian

Message sent from Intel to Sparc:
How about we invert the bytes of each word

On INTEL:
Little endian

integer is 5
String is “LLIJ”
**Parameters in RPC settings**

**Passing reference parameters**

- Pointers are meaningful only within a process
  - Address 1000 in process A may be start of an array
  - Address 1000 in process B is something else

- Forbid pointers and reference parameters?
  - These are highly important
  - Not having them is not an option
Passing reference parameters: Our read example

- Copy array into message and send to server
  - Server stub calls server with a pointer to the array
- Changes made by server using the pointer?
  - Affects the array
- Send array back to the client
- Call-by-reference replaced by copy/restore

Optimizing the copy/restore

- If the buffer is an input or output parameter?
  - One of the copies can be eliminated
- If it is input to server?
  - No need to copy back to client
- If it is output from the server?
  - No need to send from client in the first place
Parameter specification and stub generation

- Agree on the **format** of the messages
- Similar **actions** for passing complex data structures

Parameter agreements: An example

`foobar(char x; float y; int z[5])`

- **WORD** = 4 bytes
- Character in **rightmost** byte of a word
  - Leave the leftmost 3 bytes empty
- Float as a whole word
- Array as group of words
  - Precede this by word indicating the array length
Parameter agreements: An example

```c
foobar(char x; float y; int z[5])
```

<table>
<thead>
<tr>
<th>foobar's local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>z[0]</td>
</tr>
<tr>
<td>z[1]</td>
</tr>
<tr>
<td>z[2]</td>
</tr>
<tr>
<td>z[3]</td>
</tr>
<tr>
<td>z[4]</td>
</tr>
</tbody>
</table>

Besides message format, there needs to be agreement on representation

- **integers, characters, Boolean, etc**
- **For e.g.**:
  - integers could be 2's complements
  - characters are 16-bit Unicode
  - floats in IEEE #754 standard
  - Store everything in big-endian
Synchronous and Asynchronous RPC

Synchronous

Client

Wait for result

Server

Call local procedure

Time

Asynchronous

Client

Request

Server

Accept request

Call local procedure

Synchronous and Asynchronous RPC

December 3, 2019
CS555: Distributed Systems (Fall 2019)
Dept. Of Computer Science, Colorado State University

Generation of Stubs in RPC/Distributed Objects

December 3, 2019
CS555: Distributed Systems (Fall 2019)
Dept. Of Computer Science, Colorado State University
Programming with interfaces

- Most modern languages provide means to organize program as a set of modules
- Modules can communicate with each other via *invocations* or *direct access* to variables
- To control possible interactions
  - An explicit interface is defined for each module

Interface

- Specifies methods/procedures and variables that can be accessed from other modules
- Modules are implemented so that all information about them are hidden, except for …
  - What was announced via its interface
- Implementation can also change so long as the interface does not
In a distributed setting, interfaces become even more critical

- Programmers are only concerned with abstraction offered by the service and not implementation details
- Natural support for software evolution
  - Implementations can change so long as the interface (external view) does not change
- Programmers also need not be aware of the programming language used by remote object
  - CORBA

Generation of client and server stubs

- Interface is usually specified by means of an interface definition language (IDL)
- IDL is compiled into a client and server stub
Distributed Objects:
CORBA early 1990s, RMI mid-late 90s

- RPC based on distributed objects with an **inheritance** mechanism
- **Create**, **invoke** or **destroy** remote objects, and interact with them as if they are local
- Data sent over network:
  - **References**: class, object and method
  - Method **arguments**

Distributed Objects in CORBA defined using the Interface Definition Language

**GIOP/IIOP**
General Inter-ORB Protocol/Internet Inter-Orb Protocol
Web Services in a sense borrowed some of these concepts

- Used XML to describe services: Web Services Description Language (WSDL)
  - Defined methods and arguments to them
- Added another feature/problem
  - Generation of WSDL from actual implementation.

RPC style communications:
Disadvantages

- Access transparency is at the expense of flexibility
- Receiver needs to execute when the sender is sending something
- Communications in RPC are usually synchronous
  - Client is blocked until request is processed
DISTRIBUTED OBJECTS
(Lets try to do this in Java)

The JRE and bytecodes

- The Java runtime environment is based upon a virtual machine that
  - Interprets, verifies, and executes classes in the form of platform-independent bytecode
ClassLoader

- Java API includes mechanisms to:
  - **Load** class definitions in their bytecode form
  - **Integrate** them into the JRE so that instances of classes can be constructed and used

- When your Java files are compiled?
  - A similar mechanism is used when `import` statements are encountered
  - Referenced classes are loaded in bytecode format
    - Using the CLASSPATH variable to locate classes

Distributed Objects vs The ClassLoader

- Distributed Objects (based on CORBA, RMI, etc.)
  - Create object on one host; *allow process on another host* to invoke methods on that object

- ClassLoader
  - Read bytecodes making up a class definition; create an object within its *own* process
Essential requirements for a distributed object system

- Create or invoke objects on remote host/process

- To create a remote object?
  - Reference a class
  - Provide constructor arguments for the class
  - Receive a reference to the created object
    - Used to invoke methods on the object

Creating an object once the name of a class is known

```java
String className;
...
Class targetClass = Class.forName(className);
Object createdObj = targetClass.newInstance();
```
Methods: How to retrieve and invoke

```java
String methodName = "getName";
Class targetClass = Class.forName("a.b.MyClass");

java.lang.reflect.Method targetMethod;
targetMethod =
    targetClass.getDeclaredMethod(methodName, null);

targetMethod.invoke();
```
A truly open system for distributed objects will ...

- Allow clients to access objects regardless of details such as:
  - Hardware platform
  - Implementation Language
- **Java RMI**
  - Java specific but platform-independent
- **CORBA**
  - Language & platform independent

The object model

- Programs are composed of **interacting** objects
  - Each object has data and a set of methods
- Objects communicate with each other by **invoking** their methods
  - Passing arguments
  - Receiving results
- In a distributed object setting, object’s data is accessible **only via its methods**
Objects are accessed via object references

- In Java, a variable that appears to hold an object?
  - Holds a reference to that object

- Object references are first-class values
  - Can be assigned to variables, passed as arguments, and returned as results

Distributed Objects

- Object-based programs are logically partitioned
  - Physical distribution of objects is a natural extension

- Architectural styles
  - Client-Server is the most popular one
  - But there could be other ones
    - Replication for fault tolerance, performance and availability
A remote object and its interface

Remote Object

Data

Implementation of methods

m1
m2
m3

m4
m5
m6

Remote interface

Exceptions in distributed objects

- Remote invocations may fail for several reasons
  - Process containing remote object crashed or is too busy to reply
  - Invocation or result message may be lost

- Remote method invocations should be able to raise exceptions
  - E.g. timeouts, problems during method execution
Remote object references need to be unique over space and time.

<table>
<thead>
<tr>
<th>Internet Address</th>
<th>Port Number</th>
<th>time</th>
<th>Object Number</th>
<th>Interface of Remote Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bits</td>
<td>32-bits</td>
<td>32-bits</td>
<td>32-bits</td>
<td></td>
</tr>
</tbody>
</table>

IMPLEMENTING REMOTE METHOD INVOCATIONS
Remote method invocation

- **Communication Module**
  - **CLIENT**
    - Object A
    - Proxy for B
    - Request
    - Reply
  - **SERVER**
    - Remote object B
    - Skeleton + Dispatcher for B's class

Remote reference module

- **Creates** remote object references
- **Translates** between local and remote object references
The remote reference module in each process maintains a remote object table

- An entry for all *remote objects* held by process
  - E.g. in our figure, table in server records remote Object B

- An entry for each *local proxy*
  - E.g. proxy for B will be in a table at the client

Actions of the remote reference module

- When a remote object is passed as an argument or result for the first time?
  - The module creates a remote object reference
  - Add this reference to the remote object table

- When a remote object arrives in a *reply* or *request* message?
  - Reference module is asked for the corresponding *local object reference*
    - Refers either to a *proxy* or a *remote object*
The server remote object

- Lives in the server process
- Instance of class that provides the **body** of a remote object
  - Eventually handles remote requests passed by the corresponding skeleton

The Proxy

- Role is to make a remote method invocation transparent to clients
  - Behave like a local object to the invoker
- Instead of executing an invocation, the proxy **forwards invocation** in a message to a remote object
The Proxy

- Hides details of:
  1. Remote object references
  2. Marshaling & unmarshaling
  3. Sending & receiving of messages from the client

- There is one proxy for each remote object for which the process holds a remote object reference

The Proxy

- The class of a proxy implements the methods in the remote interface of the remote object
  - Ensures remote invocations are suitable for the remote object

- Each method of the proxy marshals several things into a request message:
  - A reference to the target object
  - Its operationId
  - Arguments

- After the request the proxy awaits a reply, unmarshals it, and returns results to the invoker
Dispatcher

- A server has one dispatcher and one skeleton for each remote class representing a remote object
- Dispatcher receives request messages from the communications modules
  - Uses the operationId to select the appropriate method in the skeleton
  - Passes on request message
- The dispatcher and the proxy use the same allocation of operationIds to methods of the remote interface

Skeleton

- The class of a remote object has a skeleton
  - Implements methods in the remote interface
- Skeleton method unmarshal arguments in the request message
  - Invokes corresponding method in the server remote object
- Wait for invocation to complete
- Marshals results
  - Include exceptions in a reply to the sender proxy’s method
Generation of classes for proxies, dispatchers, and skeletons

- These are generated automatically by an interface compiler
- Based on the remote interface

Dynamic Invocations

December 3, 2019
Professor: SHRIDEEP PALICKARA
Dept. Of Computer Science, Colorado State University

December 3, 2019
CS555: Distributed Systems (Fall 2019)
Dept. Of Computer Science, Colorado State University

L27.57

CS555: Distributed Systems (Fall 2019)
Dept. Of Computer Science, Colorado State University

L27.58
Dynamic Invocation: Alternative to proxies

- Proxy is generated from an interface definition
  - Then compiled *statically*
    - Sometimes this is not enough

- What if a client program receives a remote reference to an object that was not available at compile time?

- We need another way to invoke the object
  - *Dynamic Invocation*

Dynamic invocation

- The remote object reference includes information about the interface of the remote object

- The names of methods and types of arguments are needed to make the invocation
Server program

- Contains classes for the dispatchers and skeletons
  - Implementations of classes of all server remote objects
- Server remote objects also created in response to requests from clients

Factory methods

- Remote object interfaces cannot include constructors
  - Remote server objects cannot be created by remote invocation on constructors
- Remote server objects are created either in:
  - The initialization section
  - Methods in a remote interface designed for that purpose
    - Also called factory methods
The binder

- Client programs need a way to obtain a remote object reference
  - For at least one of the remote objects held by server
- A binder is separate service that maintains a table with a mapping of textual names to remote object references
- Used by:
  - Servers: Register their remote objects by name
  - Clients: Lookup remote objects

Server threads

- When an object executes a remote method invocation?
  - Execution may lead to further invocations of methods in other remote objects
- Avoid situations where execution of a remote invocation delays execution of another
- Servers generally allocate a separate thread for the execution of each remote invocation
Activation of remote objects

- Often it is not practical for objects to be kept in running processes for unlimited periods
  - Especially if they are being used sporadically

- Servers can be started whenever they are needed by the clients
  - Similar strategy is used in standard TCP services such as FTP
    - Started on demand by a service called Inetd
Active and passive remote objects

- **Active**
  - Available for invocation within a running process

- **Passive**
  - Not currently active, but can be made so
  - Consists of two components
    - Implementation of its methods
    - **State in marshaled form**

Activation is the creation of active objects from passive ones

- **Register** passive objects that are available for activation
- **Start** named server processes and activate remote objects in them
- **Keep track** of locations of servers for remote objects
Java RMI includes the ability to make some remote objects activatable

- When an activatable object is invoked?
  - If it is not active, it is made so from its marshaled state

- Uses one activator on each server machine

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

