CS 555: DISTRIBUTED SYSTEMS
[RPC & DISTRIBUTED OBJECTS]

Shrideep Pallickara
Computer Science
Colorado State University

October 26, 2017
CS555: Distributed Systems [Fall 2017]
Dept. Of Computer Science, Colorado State University

Topics covered in this lecture
- RPC
- Distributed Objects

Frequently asked questions from the previous class survey
- XDR
  - Standard serialization format: Uses big-endian, IEEE 754 for floating point, 4-byte words
  - Where does it get implemented?

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

CS555: Distributed Systems [Fall 2017]
Dept. Of Computer Science, Colorado State University

L20.1
L20.3
L20.5
L20.7
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.

Synchronous and Asynchronous RPC:

- Client
  - Wait for result
  - Synchronous
    - Call local procedure
  - Server
    - Accept request
    - Call local procedure
  - Time

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:

- Distributed objects are defined using the Interface Definition Language (IDL).
- Objects are defined in terms of their operations and parameters.
- The IDL code is compiled into client and server stubs.
- 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 (Let's 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 and its 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 unique over space and time

- Remote object references need to be unique over space and time

Implementing Remote Method Invocations

- Remote method invocation
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
  - Behaves like a local object to the invoker
  - Instead of executing an invocation, the proxy forwards invocation in a message to a remote object

- 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 [3/3]
- 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 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 unmarshals 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 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

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