Darrin Willis - dswillis
Bohan Li - bohanl

15-440 Writeup
Lab 2: Remote Method Invocation

https://github.com/darrinwillis/distributedSystems

Overview

The overall design of our project mirrors Java's implementation of RMI fairly closely. It consists of a centralized Registry Server which serves as a hashtable server for looking up Remote440 objects and a set of objects which handle client-side and server-side communication. A server which would like to make an object remote simply has to call `Communicate.rebind(String, RemoteObj)` in order to make itself available. Any client which wishes to access that object simply calls `Communicate.lookup(String)`, which returns the object (which is actually remote).

The Communicate object handles any user-initiated interaction with the framework. On the client end, Communicate uses RMIMessage to connect a RemoteStub with a remote object or opens a socket to interface with the registry server. On the Remote Object end, Communicate uses a ProxyDispatcher to listen for client RMIMessages. The Registry Server must be initiated separately.

RegistryServer

The `RegistryServer` serves as a lookup table for clients to find a remote object. All a client must do is access it at a well known port and query based on a lookup string. This is not done concurrently, but it is a fairly basic operation, so it should not be computationally expensive, so the `RegistryServer` should stay quick.

Communicate

Communicate is the object which serves as the single point of entry class for a one to make use of our framework. Communicate consists of a set of static methods which enable a remote object to make itself known and for a client to find it. Specifically, `Communicate.rebind(String key, Remote440 object)` is a static method which contacts the `RegistryServer` and adds object to the `RegistryServer` 's hashtable. It is important to point out that Communicate has a static block which instantiates a new instance of `ProxyDispatcher` (described below), which then takes care of receiving messages and handling the now remote object. `Communicate.lookup(String key)` contacts the server and returns a `Remote440` to the client. This `Remote440` object is a stub which can be treated as the actual object. If a client merely wants to see a list of the objects on the Registry, `Communicate.list()` returns a `Set<String>` of all of the provided keys for the Registry objects. All Communicate methods also have versions which take in a URL and a port number for alternative registry addresses from unix12.andrew.cmu.edu .

RemoteStub

Once the `Remote440` stub is returned, the client can cast it as implementing the

interface of the actual `Remote440` object, and use it as they like. All stubs in this fashion are created by handwriting a SomeClass_stub.java object file which inherits from `RemoteStub` and then compiling it. The `RemoteStub` abstract class has a protected `methodCall` function that takes care of marshalling the method invocation in an `RMIMessage` and sending it to to the `ProxyDispatcher` of the `Remote440` object to handle invocation. The handwritten SomeClass_stub.java file simply intercepts all of the declared methods and instead calls the inherited `methodCall` with the correct arguments.

ProxyDispatcher

When `Communicate.rebind()` is called on a machine for the first time, it creates a single instance of `ProxyDispatcher` that runs on the local machine. This `ProxyDispatcher` exists to receive any messages intended for one of its remote objects. Once it receives the `RMIMessage`, it determines which object the message was destined for and locally calls the method. It then sends the return value back to the client RemoteStub class. Whenever this `Remote440` object throws an exception, `ProxyDispatcher` bundles it as the `cause` of a `Remote440Exception`. This way, whenever a RemoteStub receives a `Remote440Exception` as a return value, instead of simply returning it, the stub throws it as an exception.

Compiling / Running

Simply run `make` to compile all necessary code. `make clean` to clean out all `.class` files.

To run a test:

1. `java RegistryServer [Port]`
   `Starts the Registry server at port Port; Default host is unix12 Default port is 15444`
2. `java testName`
   `Where testName is the desired test from below`

Testing

```
Testing Suite for Basic Functionality, Exception Handling,
     Multiple Objects
1. on a Server
     java RegistryServer [Port]
2. on an object Server
     java TestingServer [Registry URL] [Registry Port]
3. on a client
     java TestingClient
```

```
Multiple Remote Object Concurrent access, other registry hostname
1. on a Server
     java RegistryServer [Port]
2. on an object Server
     java TestingServer [Registry URL] [Registry port]
3. on a client
     java TestConcurrent [Registry URL] [Registry port]
```

Basic functionality testing begins with the server adding two PrintingObjects to the RegistryServer and ProxyDispatcher. Then the client obtains the RemoteObjectReferences from the RegistryServer and localizes them as stubs. Printing_object has an internal counter that keeps track of the number of times printThis is called. We test to see if our rmi facility is working by checking if the counter matches its expected value. We also check exception handling by throwing an exception and checking if it was correctly thrown.

Concurrent testing tests multiple concurrent messages to the ProxyDispatcher. We create 50 threads that call printThis which sends messages to a common ProxyDispatcher. We wait for the threads to finish, then we check if each instance of PrintingObject behaves properly.

Our RegistryServer can run on different URLs and ports. To test this, specify the URL and port to TestingServer. Then run TestConcurrent with the same arguments.

Possible Improvements

Caching

Our framework does not make use of any caching at any level. However, Caching could easily be put in place by keeping a cache of active sockets in ProxyDispatcher and RemoteStub.

Class Download

Our framework could include an http protocol to download/upload any missing .class files to/from the registry server, but we instead chose to spend our time deepening our testing set. In order to implement this, Registry server would have to also accept these HTTP requests, and then return the .class file. Communicate would also need to have a check for missing files, checking with the server.

RMI Compiler

Our framework does not come with an RMI Stub Compiler so these files must be handwritten. This has already been done for all of our provided tests. A compiler would not be terribly difficult to write, as it must simply duplicate the interface of the given class, and replace all the methods with a single method, but as it was pretty far beyond the scope of this project, we did not implement it.