Kevin Crane (kevincra)
Prashanth Balasubramaniam (pbalasu1)
Fall 2013
Distributed Systems (15-640/440)
Project 2 – RMI Facility

1. Design/Implementation
   a) In designing this system, I referenced the source code of Java RMI (http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/rmi/) to get an idea of how everything worked originally. The three main components that had to be accounted for were the RMI server, the registry, and each RMI client.
   b) For this project, there were a few main core components that had to be implemented:
      i. RMI Registry
         - This was the most unique interesting part of the system and, as such, the hardest to design. In the end, I decided to implement the registry as a class that could be run as its own thread. This let it maintain logical independence from the RMI server while allowing easy access to the data it contained.
         - There were two main data structures within the registry. The first was a map that linked bound objects to a string key. The server was able to perform local lookups with these keys, providing direct access to the remote objects that were accessible from clients. The other data structure linked remote object references to the same string keys. These were returned to clients whenever the registry received a remote lookup request.
         - In addition, the registry could respond to *list()* requests that would return a list of all of the keys that were found in the registry.
         - The registry is initialized as a new thread from the server, at which point it enters a loop where it listens for socket requests from clients and returns RMI messages.
      ii. RMI Server
         - The server only has a few real responsibilities. First, it is in charge of maintaining a registry that can provide references to each remote object. Second, it has to create new remote objects and bind them with this registry. Finally, it listens for incoming RMI invocation messages from client sockets.
         - The first two responsibilities are trivial and really just amount to instantiating a new registry thread and starting it, then calling the *rebind(Object)* method whenever you want to make an object available remotely.
         - Invoking methods from an RMI message is the trickier part. When the server receives a message, it contains the reference key, name of method, and list of arguments. To get the correct object, just perform a *locallookup()* on the key provided. Next, you'll use this object

and Java reflection to find the appropriate method based off of the method name that was passed in. Finally, this can be invoked on the object found earlier with the arguments that were given.

iii. RMI Client

- Each RMI client can do whatever it wants with a minimum of boilerplate code (makes sense given RMI is supposed to simulate local access). First, you need to locate the registry. This is done by passing the hostname of the server machine, plus a hard-coded registry port number.
- Next, you need to get a remote object reference. This is done by sending an RMI lookup message to the registry, asking for a reference to a given key. This is returned in another RMI message.
- Finally, the client generates a new remote object stub, letting you call methods as if locally. It does this with a proxy object handler and the interface for each remote object.
- Once you've created the proxy object, you can call methods on it normally and get results as if you were calling it with a regular local object. The only difference is that you need to surround these methods with a try/catch block for remote exceptions.

c) There were also a few important pieces that contributed to cover the gaps between the three main components described above:

i. Remote Object Reference

- The remote object references were basically a way to encapsulate the important information about each bound remote object so they can be accessed from a client. Each one contains the hostname and port of the server that holds the object, the key string that corresponds to it in the registry, and the name of the actual class.
- The only non getter method is *localise()* which returns a new instance of the class referenced.

ii. RMI Message

- We used a generic RMI message object to describe any bundle of data that could be serialized and passed between server, client, or registry. Each one had a type and an object payload
- The different types of RMI messages are: Invocation, Return, Exception, Registry list, and Registry lookup. This covers all of the use cases that we had for communicating between client and server/registry.
- These were easily passed around using a helper class we wrote that could take a socket or hostname/port combo and send and receive RMI messages without having to repeat the boilerplate code. This helped simplify our code a lot and let us send and receive data with

one line rather the 8-10 lines that it usually requires.

iii. Remote Stub
- Each remote object on the client side had to be extended as a remote stub. This basically meant that it had to call an *invokeMethod* method that would open a connection with a server machine and give it the information required to invoke a new method and send back its return data.
- This class was only needed for instantiation with the remote stub proxy described next. It helped to have all of the stubs extend this as they could then directly invoke the methods remotely from each generated method.

iv. Remote Stub Proxy
- Because we didn't want to have to rely on having pre-made remote stubs for every class that we wanted to access remotely, we used a proxy handler to generate new stub classes dynamically.
- Each one takes a remote object reference, identifies what type of object is referenced, and creates a new one. However, instead of its normal methods, each method is replaced with code to create a new RMI invocation message, sending it to the server machine to be invoked.
- This was definitely the coolest part of the project for us, as we had never really worked with proxies or dynamically generated classes before, so it was really cool to see them created on the fly.

2. As far as we can tell, everything that we detailed above is working properly. We have ample exception handling, so when things go wrong we print error messages to the console rather than crashing. Because of the breadth of Java's RMI system, there is obviously some stuff that was not implemented. Of the three “bonus” ideas mentioned in the project handout, we implemented the ability to compile and generate stub classes, but did not have time to build the ability to download class files or cache connections. We felt that the ability to compile stubs was the most important piece of functionality to add because while we can reasonably expect developers to have access to the class file of each object they're using, it would be much more burdensome to expect them to write the boilerplate code for each stub class as well.
3. To build the project, you must navigate to the root directory of the project (the one with the Makefile). There, you just type

   ```
   $ make
   ```

   to compile the project. To be safe, be sure you have write-access in the directory as it will create a folder called *classes* that contains all of the compiled classfiles, including any processes that you would want to run. You will need to build the project on each machine that will be running it.

To actually run the project, you will need two different types of clients. On one terminal, first run the RMI server:

```
$ java -cp out/ distsys.RmiServer
```

Next, on any number of other terminals, you can create RMI clients that will work with this server. The two examples that we have developed can be run as follows:

```
$ java -cp out/ distsys.RmiClientMaths server_hostname
```

or

```
$ java -cp out/ distsys.RmiClientSleep server_hostname [sleep_time]
```

As long as you have entered the hostname correctly, that's all you should have to do to run everything. To clean up all of these messy classfiles later, you can type:

```
$ make clean
```

4. There aren't any extra dependencies or libraries required. This was tested with Java 1.6, so it should work fine with that.