CSE 660: Operating Systems Concepts & Theory

Professor: Dr. Tong Lai Yu

June 16, 2011

Lab 3 Report

Part I Remote Procedure Call (RPC)

Part II Java Remote Method Invocation (RMI)

Part III Android Application Development

By: Rowen Concepcion

Marc Santoro

Purpose:

The purpose of this lab is split up into three different parts:

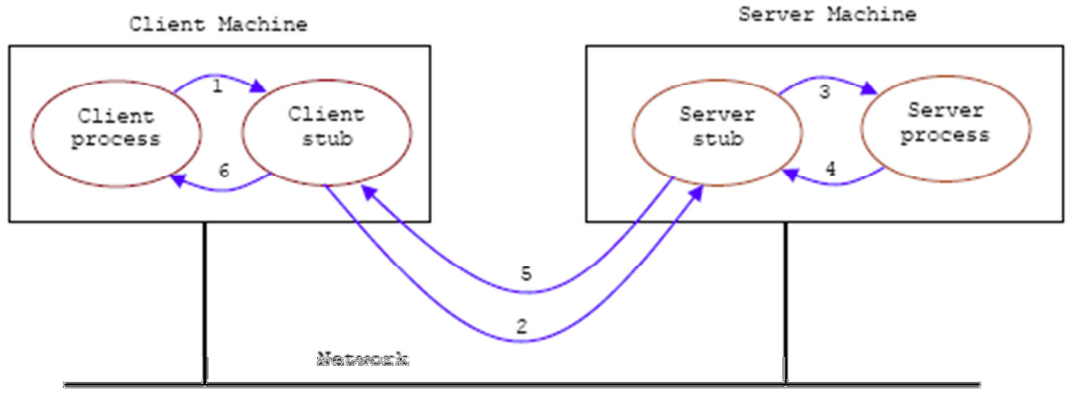
- Remote Procedure Call (RPC)
    - Study RPC
    - Create a RPC application that generates random numbers
- Java Remote Method Invocation (RMI)
    - Create a Remote Java Object that performs the addition of two numbers
    - Create a Remote Java Object that provides an interface to a "random number generator" where it provides two operations:
        i. A method that accepts the number of random numbers N and the input parameter, and return N random numbers
        ii. A method that accepts the number of random numbers N, an upper bound and a lower bound (as constraints), and return N random numbers that are subjected to the constraints
- Android Application Development
    - Install the Android SDK
    - Install Eclipse
    - Create Hello World in Android emulator
    - Do one of the following two options:
        i. Create an animation of the Solar System using OpenGL
        ii. Create a graphical interface of an android client/server simple calculator that can handle integer arithmetic

PART I. Remote Procedure Call:

Definitions:

I. **Remote Procedure Call (RPC)**

a. **Remote Procedure Call** or also known as remote invocation or remote method invocation (in terms of object-oriented design) is defined as an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space. The first popular implementation of RPC is on Unix which is used as a template for Network File System



(1) and (3) are ordinary procedure calls.
(2) and (5) are messages.
(4) and (6) are ordinary procedure returns.

An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process.

b. **XDR** or **External Data Representation** is an Internet Engineering Task Force specification or standard that allows data to be wrapped in an architecture independent manner allowing data to be transferred between heterogeneous computer systems.

c. In Linux system, the command **rpcgen** is a tool that generates C code to implement an RPC protocol.

Implementation of Remote Procedure Call:

For this section of the lab, we were asked to follow the specific instructions in order to understand what RPC is, and how to create and implement one from scratch. It is important to check if the system you are working with has the packages needed to run RPC. To do this, first type the following in the command line:

```
$ man rpcgen
```

If the man page for rpcgen does not show up, most likely the system has the packages missing. Please consult the specific instructions in downloading the rpcgen package of the Linux flavor you are running on. If the system you are using has the ability to run rpcgen, please move on to the next step.

The starting program that is used to construct the program files for this lab is **rand.x**

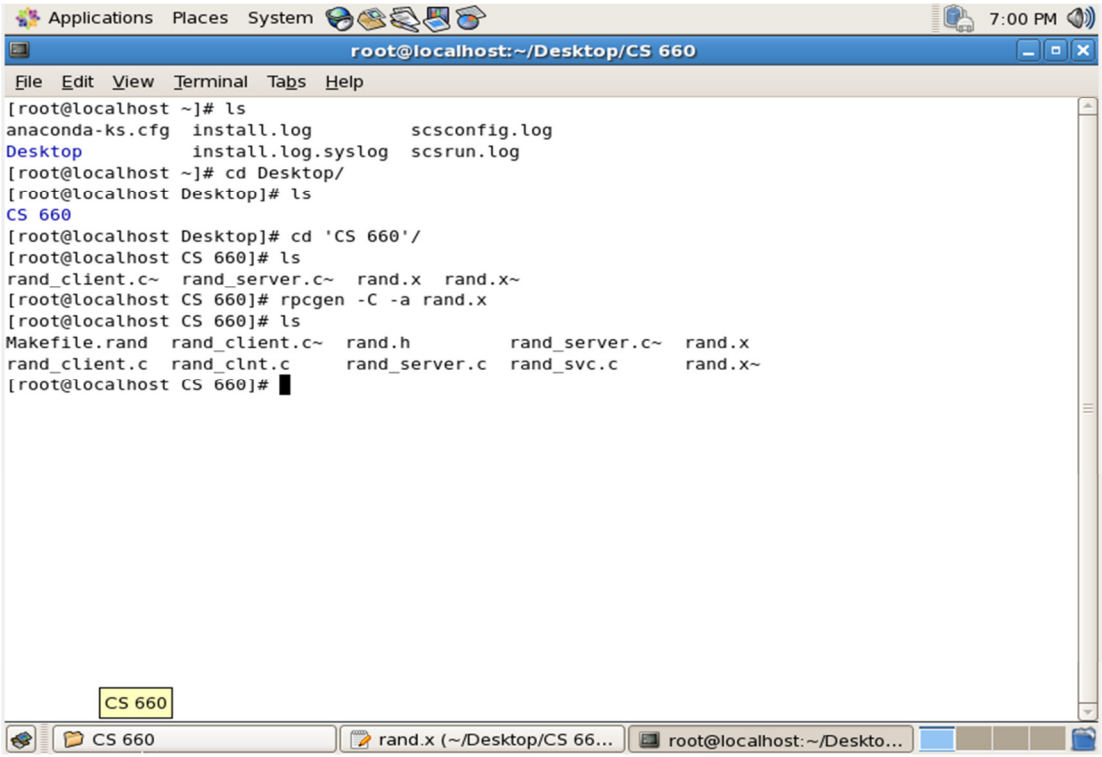
```
/* rand.x */

program RADN_PROG {
  version RAND_VERS {
    void INITIALIZE_RANDOM ( long ) = 1;        /* service #1 */
    double GET_NEXT_RANDOM ( void ) = 2;        /* service #2 */
  } = 1;
} = 0x30000000;          /* program # */
```

We generated the program templates by typing the following command at the terminal:

```
$ rpcgen –C –a rand.x
```

The switch/option **–C** is used to generate code in ANSI C. This option also generates code that could be compiled with the C++ compiler, while the switch/option **–a** is used to generate all the files including sample code for client and server side.



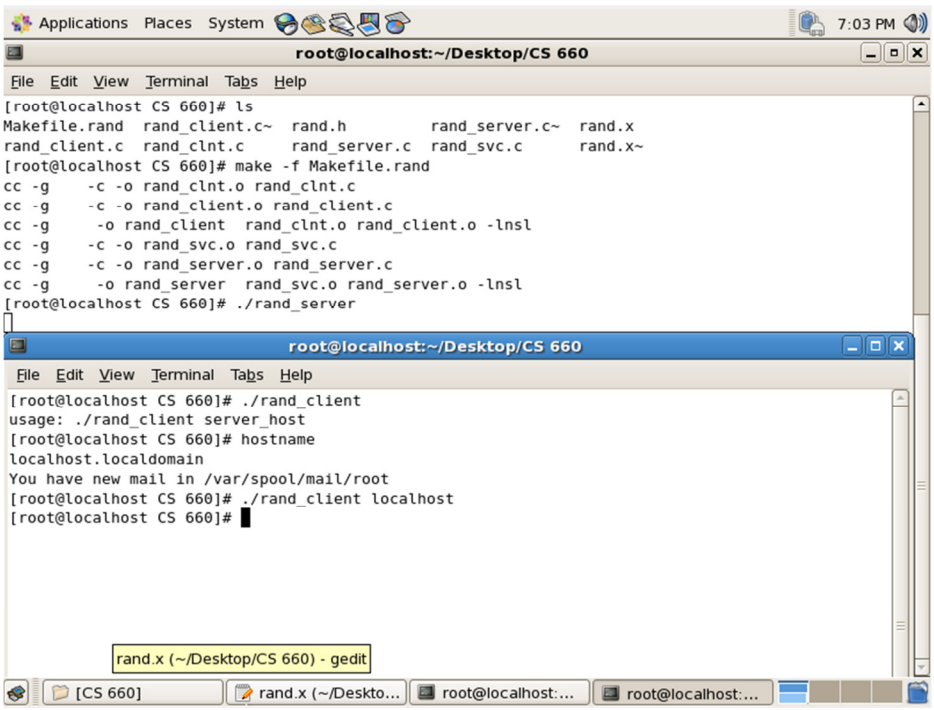
We have now generated the executable files **rand_client** and **rand_server**. We now execute them by opening one terminal and running:

```
$./rand_server
```

Another terminal window is opened, and we run the following command to activate the client:

```
$./rand_client localhost
```

A confirmation of the code working is there in no error message or response to both of the terminals once the two above commands are run. The screenshot below shows that the code compiled and ran as expected.



The next task we needed to do is to modify **rand_server.c**, and the **get_next_random_1_svc()** function in it as follows:

```
double
radn_prog_1(char *host)
{
.....
.....
        return *result_2;
}

int
main (int argc, char *argv[])
{
....
....
 double x;
        int i;
        printf("\n twenty random numbers ");
        for ( i = 0; i < 20; ++i ){
                x = radn_prog_1 (host);
                printf(" %f, ", x );
        }
exit (0);
}
```

```
double *
get_next_random_1_svc(void *argp, struct svc_req *rqstp)
{
        static double  result;

        result += 0.31;
        if ( result >= 1.0 )
          result -= 0.713;

        return &result;
}
```

Afterwards, we modify **rand_client.c** as follows:

Since we have changed both of our code for the RPC client and RPC server, we need to recompile and link all the different files related to our RPC. This is done by typing the following in the terminal:

```
$ gcc -c rand_server.c
$ gcc-c rand_client.c
$ make -f Makefile.rand
```

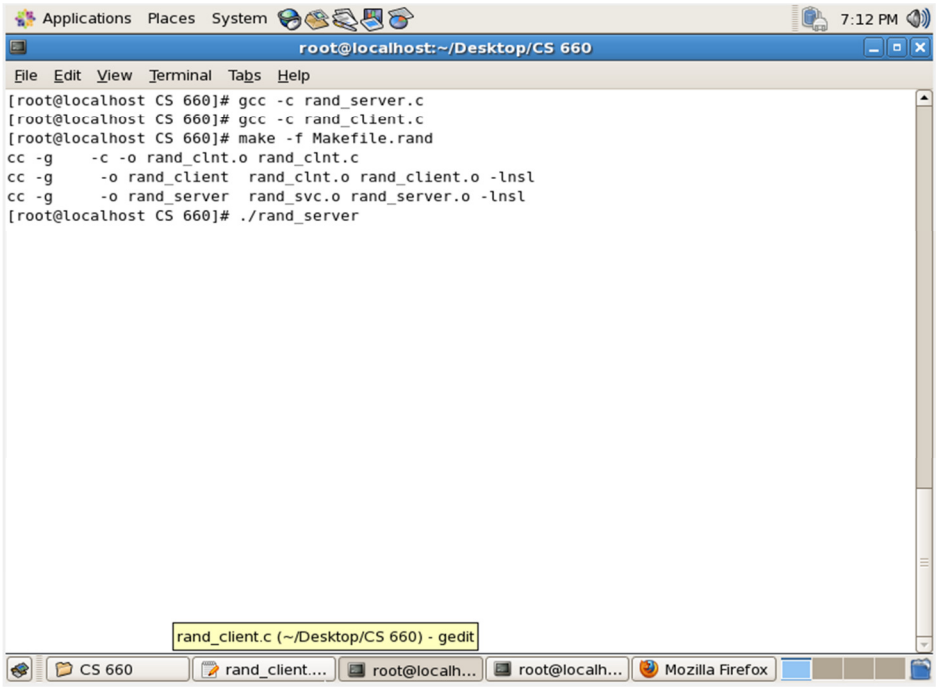
After re- generating the executable files **rand_client** and **rand_server,** we executed them by opening one terminal and running:

```
$./rand_server
```

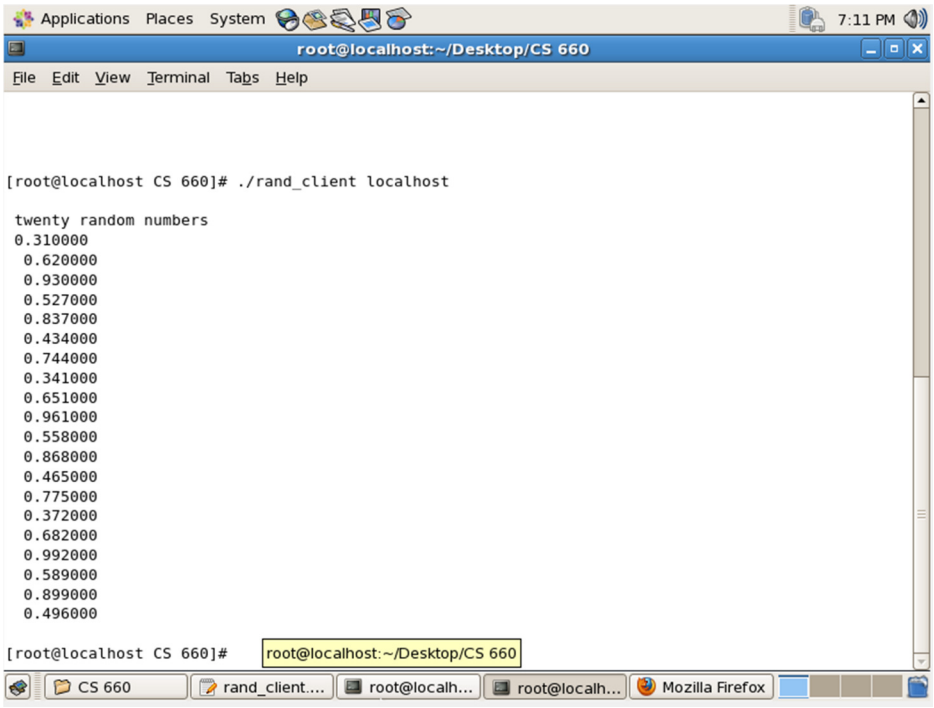
Another terminal window is opened, and we run the following command to activate the client:

```
$./rand_client localhost
```

The following two screenshots show the two terminal widows:



The server terminal is shown to be activated and running.



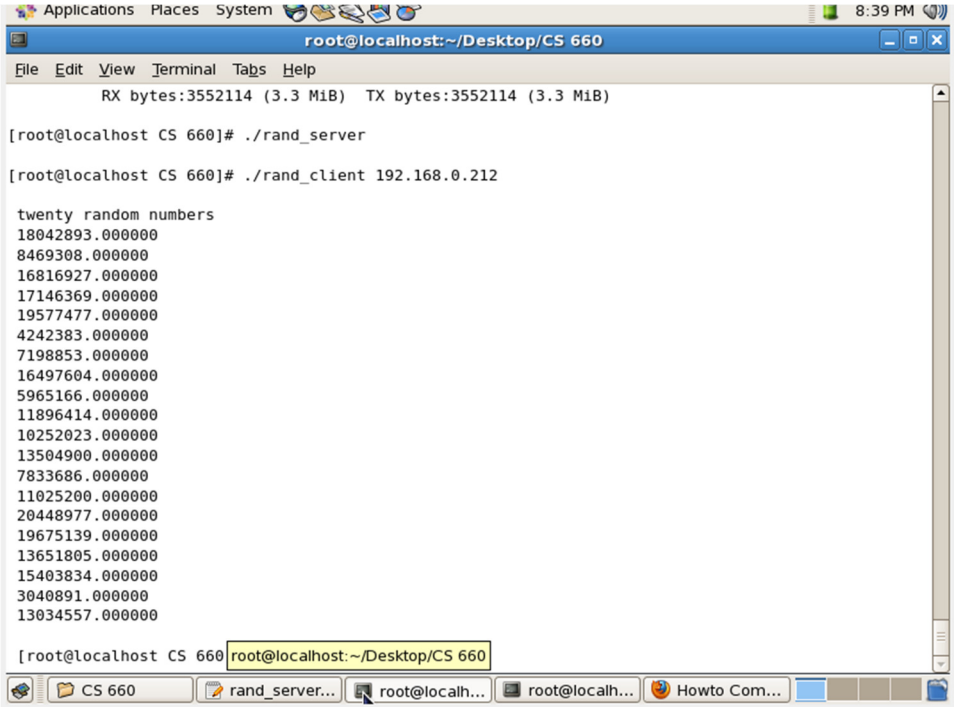
The client terminal is shown to have received the correct response from the server. We were asked to create our own random number generator function in c by modifying the **get_next_random_1_svc()** in **rand_server.c**.

```
double *
get_next_random_1_svc(void *argp, struct svc_req *rqstp)
{
        static  double  result;

        long int n = rand();
        result = n/100;
       return &result;
}
```

The last task for this lab was to run the executable files rand_client and rand_server, but this time connect to a remote server in the lab network. We managed to connect to:

```
$./rand_client 192.168.0.212
```



The screenshot shows the connection from the client to the remote server, and correctly generating twenty random numbers.

Problems/Challenges Encountered:

RPC Problems and Challenges

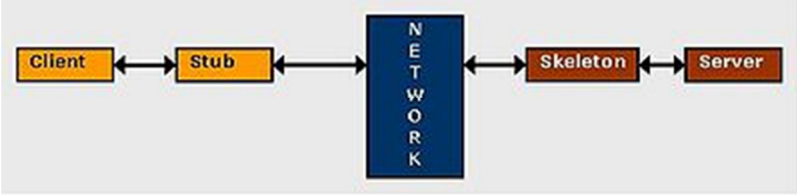
The first challenge encountered during this lab is to completely understand how Remote Procedure Call works, and understanding what rpcgen and it's different options does in terms of creating the necessary files to create an executable file to be used in different Linux Machines. The second issue is the directions on how the lab is being done. While I understand the need for students to learn RPC commands, network and ipconfig commands, and others, the order in which it each code is used, and the code option that needed to be used itself was a bit confusing. Luckily, with the help of fellow classmates who have and knowledge and experience with this subject clarified some confusions I had with the lab instructions. The third problem that I encountered is the compilation of the server.c and client.c files. Compiling them using g++, which is stated in the lab instructions, produced errors, and did not yield any favorable results. Several restarts had to be done with this lab to ensure that no previous step before the compilation was the cause of the error. After a few hours of looking at everything, I realized that gcc is the correct command instruction to use to compile the server and client files. The last challenge I encountered with RPC is connecting to a remote PC that is running the corresponding server program. With the help of a few classmates, after few tweaks to the network configuration of each PC, eventually the client and server were up and running, and the correct results were produced.

PART II. Java Remote Method Invocation (RMI):

Definitions:

d. **Java** is a general purpose, concurrent, class-based, object-oriented, high level programming language that derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. It is specifically designed to have as few implementation dependencies as possible. Java has a number of features that make the language well suited for use on the World Wide Web. Java applications are typically compiled to bytecode (called a class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. It is intended to let application developers "write once, run anywhere".

e. The **Java Remote Method Invocation (RMI)** Application programming interface or system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. RMI provides for remote communication between programs written in the Java programming language.



It is the object-oriented equivalent of remote procedure calls (RPC).

Implementation of Java Remote Method Invocation:

Before beginning to do this lab, it is recommended to go to the following links to learn more about Java and Java RMI.

- http://download.oracle.com/javase/1.4.2/docs/api/index.html
- http://download.oracle.com/javase/tutorial/rmi/index.html
- http://download.oracle.com/javase/1.5.0/docs/guide/rmi/hello/hello-world.html

The examples given in the website “Getting Started Using Java RMI” was used as a guide/template to create the remote object, client object, and server object used to complete this lab. It is also recommended to create the files in the order given in this lab report.

We also need to check if Java is installed in our system. If it is not installed, . Please consult the specific instructions in downloading the Java package of the Linux flavor you are running on. If the system you are using has the ability to run Java, please move on to the next step.

Remote Sum

The first task given in this part of the lab is to create a "remote Java object" that performs addition of two numbers (Remote Sum). A client sends two numbers to the server, and the server performs an addition of two numbers, and returns the result back to the client.

The first step is to create the remote object, which helps define the information that will be exchanged between both the client and the server. A remote object is an instance of a class that implements a remote interface. A remote interface extends the interface java.rmi.Remote and declares a set of remote methods. Each remote method must declare java.rmi.RemoteException (or a superclass of RemoteException) in its throws clause, in addition to any application-specific exceptions.

The remote interface file **RemoteSum.java** is as follows:

```
//RemoteSum.java file

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteSum extends Remote {
    int sum(int num1, int num2) throws RemoteException;
}
```

The next step is to implement the server class. The server class is defined as the class that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java RMI registry. The class that contains this main method could be the implementation class itself or another class entirely.

The **Server.java** file for the Remote Sum is as follows:

```
//Server.java

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class Server implements RemoteSum
{
        public Server() {}

        public int sum(int num1, int num2)
        {
                return num1 + num2;
        }

        public static void main(String args[])
        {
                try
                {
                        Server obj = new Server();
```

```
                        RemoteSum stub = (RemoteSum)
UnicastRemoteObject.exportObject(obj, 0);

                        Registry registry = LocateRegistry.getRegistry();
                        registry.bind("SUMRMI", stub);

                        System.err.println("Server ready");
                }
                catch (Exception ex)
                {
                        System.err.println("Server exception: " +
ex.toString());
                        ex.printStackTrace();
                }
        }
}
```

The third step is to implement the client class. The client program obtains a stub for the registry on the server's host, looks up the remote object's stub by name in the registry, and then invokes the method on the remote object using the stub.

The **Client.java** file for the Remote Sum is as follows:

```
//Client.java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.lang.Integer;

public class Client
{
        public static void main(String[] args)
        {

                String hostname = null;
                int num1 = 0, num2 = 0;

                if(args.length == 3)
                {
                        hostname = args[0];
                        num1 = Integer.valueOf(args[1]).intValue();
                        num2 = Integer.valueOf(args[2]).intValue();
                }
                else
                {
                        System.out.println("Invalid input: Please enter two
numbers to be added together...  " );
                        System.exit(0);
                }



                try
```

```
                {
                        Registry registry =
LocateRegistry.getRegistry(hostname);
                        RemoteSum stub = (RemoteSum) registry.lookup("SUMRMI");
                        int server_response = stub.sum(num1, num2);
                        System.out.println(num1 + " + "  + num2 + " = " +
server_response);
                }
                catch (Exception ex)
                {
                        System.err.println("Client exception: " +
ex.toString());
                        ex.printStackTrace();
                }
        }
}
```

The above code also has built in error checking/data verification, where the input from the client is checked whether or not it is acceptable; we return an error me if the data is invalid, else we send the data to the server for calculation.

Now that the files are completed, we need to compile the java files to create their respective class files. This is done by typing the following command in the terminal:

```
$ javac Client.java Server.java RemoteSum.java
```

The files Client.class, Server.class and RemoteSum.class are now created in the directory. Next is we need to create a **.jar** file to hold our executable files. Type the following in command in the terminal:

```
$ jar cf <name of jar file>.jar Client.java Server.java
RemoteSum.java Client.class Server.class RemoteSum.class
```

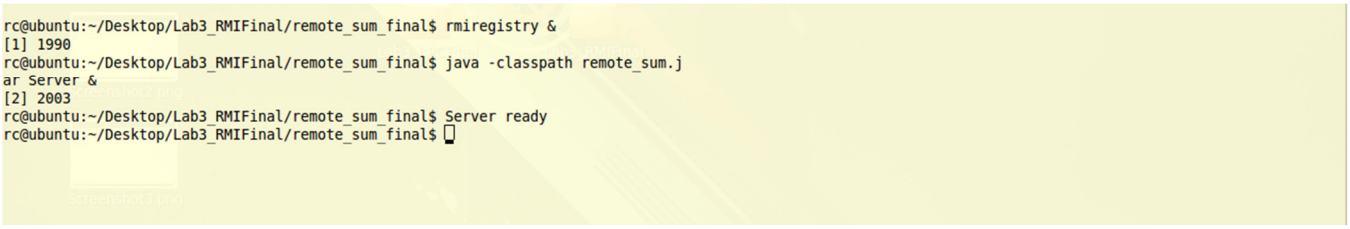
We have now created a **.jar** file that holds our executable file. In order to test it, we first start a remote object registry on the port or host of the server. This is done by typing:

```
$ rmiregistry &
```

In order to run our executable file inside our .jar file, we type this command:

```
$ java –classpath <name of jar file>.jar Server &
```

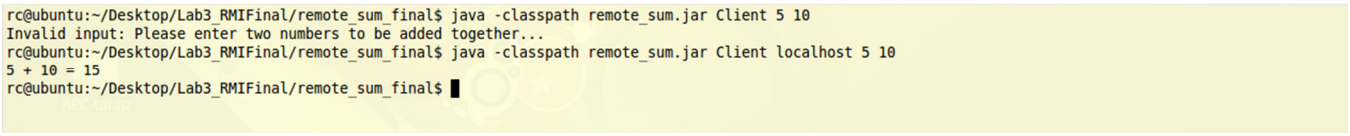
There should be a confirmation message “Server ready” in the terminal screen that the Server class was correctly executed.



We now open another terminal window and run the command to create an instantiation of our client which will speak to our local server.

```
$ java –classpath <name of jar file>.jar Client <local host>
<first number to be added> <second number to be added>
```

The terminal will display the sum of the two numbers that were sent to the server.



The last step is to verify our Remote Sum program works in a remote server. It is important to note that the client’s host file is to add the server’s IP address in /etc/hosts file. To do this, type:

```
$ vim /etc/hosts
```

In this file, you have to comment your localhost IP address and add the current IP address that you’re connected with. The screenshot below is an example on how to change /etc/hosts. Note that the IP address #192.168.0.199 is what our system is assigned to. Uncomment that line of code, and comment out localhost to make the remote server.

It is also important to note that the server can only run one rmiregistry connection at a time. So, when recompiling and running the Remote Sum executable files, we need to kill the rmiregistry running in the background. This is done by:

```
$ ps –aux | grep rmi
$ kill <number of thread that is running rmiregistry>
```

Random Number Generator

For the second part of this we are tasked to provide an interface for a Random Number Generator. The system interface should provide two operations: a method that accepts the number of random numbers N as the input parameter and returns N random numbers, and a method that accepts the number of random numbers N, and an upper and a lower bound for the requested random numbers, and returns N.

Below is the code for the RemoteRandom.java, Server.java, and Client.java for the Random Number Generator. Please note that the names for the Server and Client for the Random Number Generator and Remote Sum are the same. Please save these files in a separate directory.

```
//RemoteRandom.java file

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.ArrayList;

public interface RemoteRandom extends Remote
{
        ArrayList rand1(int number_of_random_values) throws RemoteException;
        ArrayList rand2(int number_of_random_values, int lower_bound, int
upper_bound) throws RemoteException;
}
```

The **RemoteRandom.java** file for the Random Number Generator

```
//Server.java file

import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.Random;

public class Server implements RemoteRandom
{

        public Server() {}

        public ArrayList rand1(int number_of_random_values) throws
RemoteException
        {
                ArrayList server_response = new ArrayList();

                Random rand = new Random();

                while(server_response.size() < number_of_random_values)
                {

        server_response.add((Integer.valueOf(rand.nextInt())));
                }
                return server_response;
        }

        public ArrayList rand2(int number_of_random_values, int lowerbound,
int upperbound) throws RemoteException
        {
                ArrayList server_response = new ArrayList();

                if(lowerbound >= upperbound)
                {
                        return server_response;
                }

                Random rand = new Random();

                while(server_response.size() < number_of_random_values)
                {
                        int random_number =rand.nextInt(upperbound);

                        if(random_number > lowerbound)
                        {

        server_response.add((Integer.valueOf(random_number)));
                        }
                }
                return server_response;
        }


        public static void main(String args[])
```

```
        {
                try
                {
                        Server object = new Server();
                        RemoteRandom stub = (RemoteRandom)
UnicastRemoteObject.exportObject(object, 0);

                        Registry registry = LocateRegistry.getRegistry();
                        registry.bind("RANDRMI", stub);

                        System.err.println("Server ready");
                }
                catch (Exception ex)
                {
                        System.err.println("Server exception: " +
ex.toString());
                        ex.printStackTrace();
                }
        }
}
```

The **Server.java** file for the Random Number Generator

```
//Client.java file
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.ArrayList;
import java.lang.Integer;

public class Client
{
        private static void printList(String message, ArrayList list)
        {
                System.out.println(message);

                for (int i=0; i<list.size(); i++)
                {
                        System.out.println(list.get(i));
                }
        }

        public static void main(String args[])
        {
                String hostname = null;
                int number_of_random_values = 0;
                int lowerbound =0;
                int upperbound =0;

                if(args.length == 4)
                {
                        hostname = args[0];
                        number_of_random_values =
Integer.valueOf(args[1]).intValue();
```

```
                        lowerbound = Integer.valueOf(args[2]).intValue();
                        upperbound = Integer.valueOf(args[3]).intValue();
                }
                else if(args.length == 2)
                {
                        hostname = args[0];
                        number_of_random_values =
Integer.valueOf(args[1]).intValue();
                }
                else
                {
                        System.out.println("Invalid input: Please enter
either the number or random numbers (and optional) upper and lower bounds "
);
                        System.out.println(args.length);
                        for (int i = 0; i < args.length; i++)
                        {
                                System.out.println(args[i]);
                        }
                        System.out.println("Invalid input: Please enter
either the number or random numbers (and optional) upper and lower bounds "
);
                        System.exit(0);
                }

                try
                {
                        if(number_of_random_values == 0)
                        {
                                System.out.println("Error: You have enter zero
random values" );
                                System.exit(0);
                        }
                        ArrayList list;

                        Registry registry =
LocateRegistry.getRegistry(hostname);
                        RemoteRandom stub = (RemoteRandom)
registry.lookup("RANDRMI");

                        if(lowerbound > 0 && upperbound >0)
                        {
                                list = stub.rand2(number_of_random_values,
lowerbound, upperbound);
                                printList("The " + number_of_random_values + "
random numbers between "
                                                + lowerbound + " and " +
upperbound + " are:", list);
                        }
                        else
                        {
                                list = stub.rand1(number_of_random_values);
                                printList("The " + number_of_random_values + "
random numbers are:", list);
                        }
                }
```

```
                catch (Exception ex)
                {
                        System.err.println("Client exception: " +
ex.toString());
                        ex.printStackTrace();
                }
        }
}
```

The **Client.java** file for the Random Number Generator

In order to run our executable files, we follow the same steps that were done when we created the Remote Sum:

- Compile the java files to create their respective class files. This is done by typing the following command in the terminal:

```
$ javac Client.java Server.java RemoteRandom.java
```

- Create a **.jar** file to hold our executable files. Type the following in command in the terminal:

```
$ jar cf <name of jar file>.jar Client.java Server.java
RemoteRandom.java Client.class Server.class RemoteRandom.class
```

- Start a remote object registry on the port or host of the server. This is done by typing:

```
$ rmiregistry &
```

- Run the server inside our .jar file

```
$ java –classpath <name of jar file>.jar Server &
```

```
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$ ls
Client.class  Client.java  META-INF  random_number.jar  random_numbers.jar  RemoteRandom.class  RemoteRandom.java  rn.jar  Server.class  Server.java
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$ rmiregistry &
[1] 2421
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$ java -classpath rn.jar Server &
[2] 2434
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$ Server ready
```

- Create an instantiation of our client which will speak to our local server.

```
$ java –classpath <name of jar file>.jar Client <local host>
<number of random number to be returned>

$ java –classpath <name of jar file>.jar Client <local host>
<number of random number to be returned> <lower bound> <upper
bound>
```

The terminal will display N random numbers or N random numbers with the constraints of a lower bound and upper bound.

```
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$ java -classpath rn.jar Client localhost 5
The 5 random numbers are:
539284681
202033157
-164350825
1099102000
-2060331546
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$ java -classpath rn.jar Client localhost 10 1 100
The 10 random numbers between 1 and 100 are:
52
47
53
47
37
16
73
91
40
81
rc@ubuntu:~/Desktop/Lab3_RMIFinal/random_numbers_final$

rc@ubuntu:~/Desktop/Lab3_RMIFinal/remote_sum_final$ java -classpath remote_sum.jar Client 5 10
Invalid input: Please enter two numbers to be added together...
rc@ubuntu:~/Desktop/Lab3_RMIFinal/remote_sum_final$ java -classpath remote_sum.jar Client localhost 5 10
5 + 10 = 15
rc@ubuntu:~/Desktop/Lab3_RMIFinal/remote_sum_final$
```

Problems/Challenges Encountered:

There were two main issues in the Java RMI Lab. First is the lack of knowledge and experience of this team with Java Programming. Writing a Java program is challenging as is, writing a Client, Server, and a Remote Program to do RMI made it worst. Luckily, the tutorials online regarding RMI really helped a lot, and their sample 'Hello World' programs was used as a template to create both the Remote Sum and the Random Number programs. Also, a fellow classmate who was good at Java Programming was kind enough to help out and give a crash course on how to correctly create a Java Program, and to create the functions necessary for RMI to work. The second issue is for both the Client and the Server to connect to each other in the Network in the JBH 356 Lab. While the experience of doing the RPC part of the lab did give us an idea on how the two PCs to connect to each other, there was an added difficulty due to the /etc/hosts file. This file had to be modified in such a way that both the Client and Server would be able to talk to each other. The lab instructions only briefly mentioned that it may be necessary to edit /etc/hosts, but did not mention why it had to be done, or how it had to be done. The team was stuck in this part for at least one whole week. This issue was resolved only when one of the other teams managed to configure their /etc/hosts correctly, and this knowledge was shared to my team.

Part III. Android Development:

Definitions:

Android: Android is a software stack for mobile devices that includes an operating system, middleware and key applications. (source: http://developer.android.com/guide/basics/what-is-android.html). Android was developed by Google, based on the Java programming language, and is currently the most popular mobile operating system in the world. Android has many releases in use today. The current release is 3.1, and it is code named Honeycomb.

Java JDK: The Java JDK is the Java Development Kit. It contains everything that a software developer needs to write programs in the Java Programming Language. Aside from the Java Runtime Environment and Java Virtual Machine, it contains a loader, a compiler, a debugger, and various other programming tools required for proper development in Java.

Android SDK: The Android SDK is similar to the Java JDK in that it is a bundle of tools such as a debugger, a phone emulator, software libraries, and other required components for software development on the Android Platform

Eclipse: Eclipse is an extremely popular IDE (Integrated Development Environment) that is used by developers across many different languages including Java, Python, C++, Perl, PHP, Ruby, and many more. It also has a great many plug in features that are available to download to increase the functionality of the system for individual programming languages. Eclipse was written primarily in Java, and it is open source software.

Implementation:

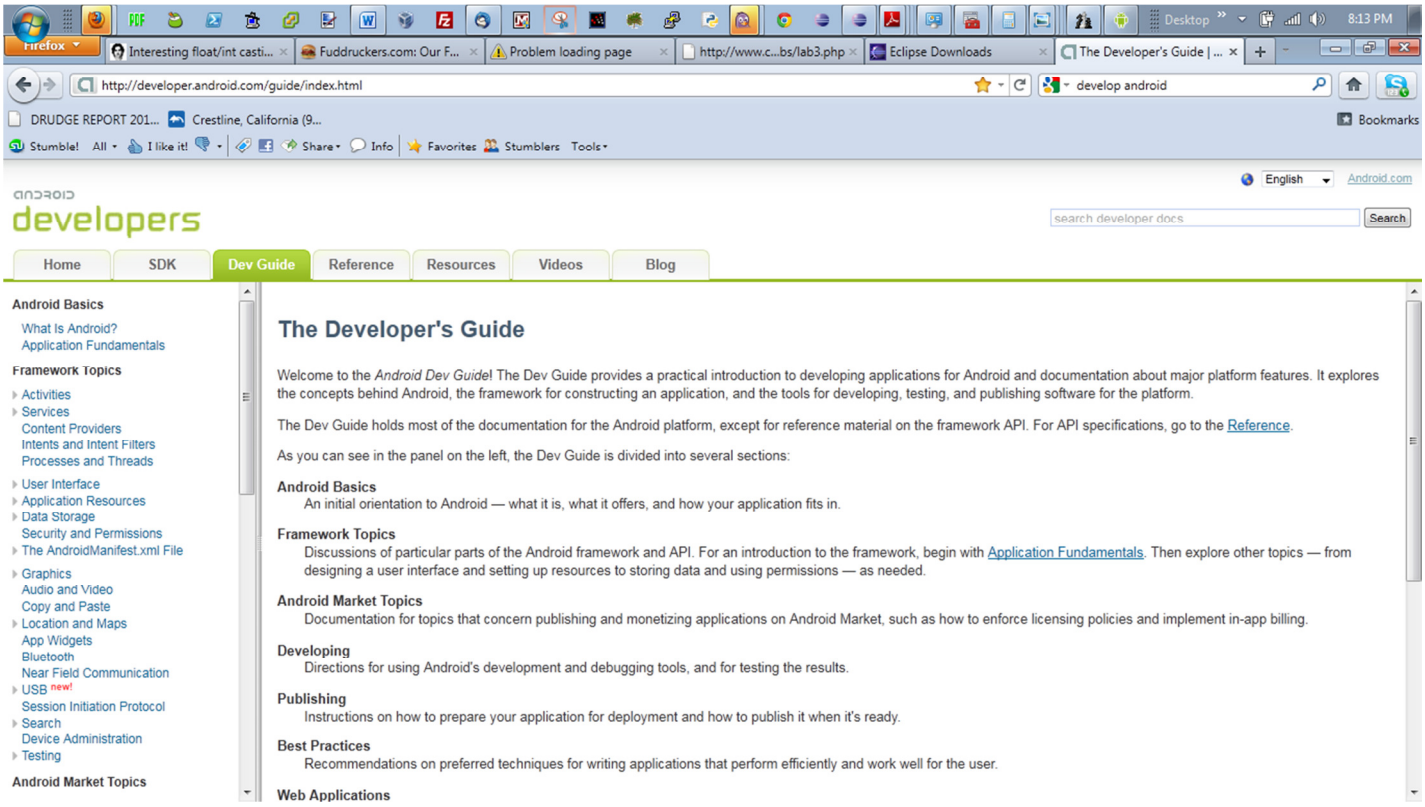
This portion of the lab focused on installing the necessary tool chain to develop Android software, and then the creation of two programs:

1. A hello world style program called “Hello Android”

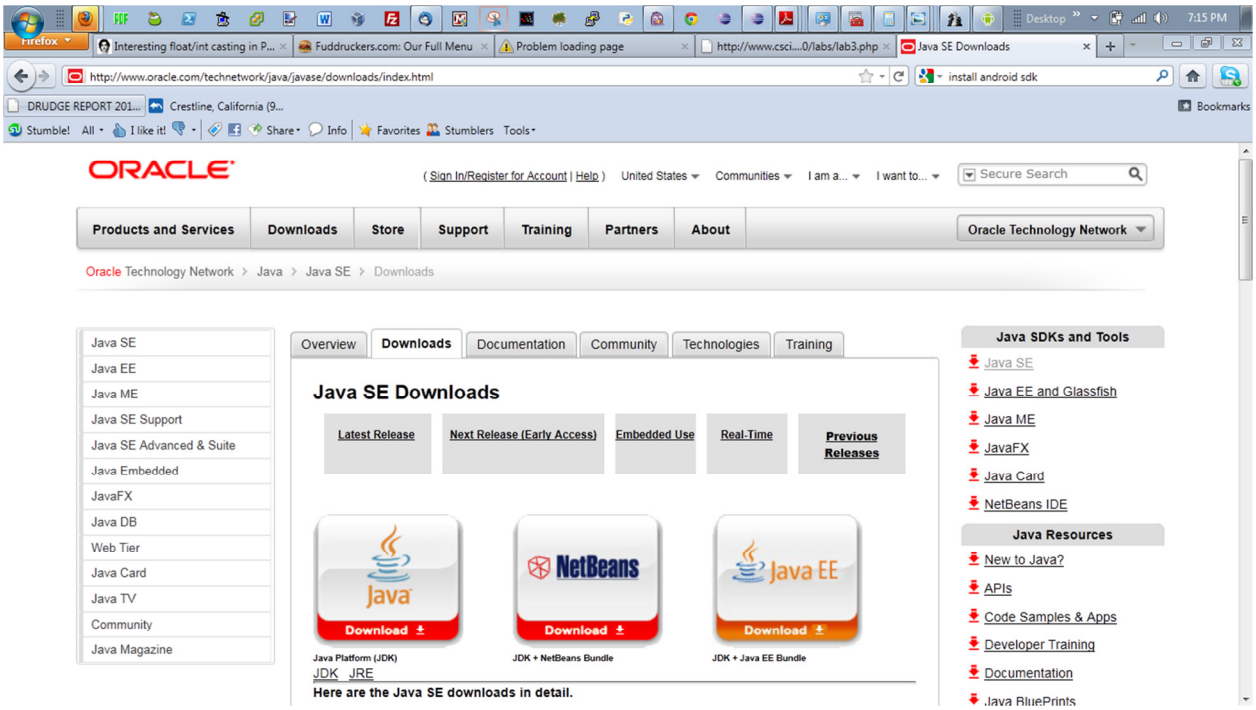
2. A simple calculator that could communicate with a server.

The platform for the team’s development was Windows 7, and the software was developed in the Eclipse IDE.

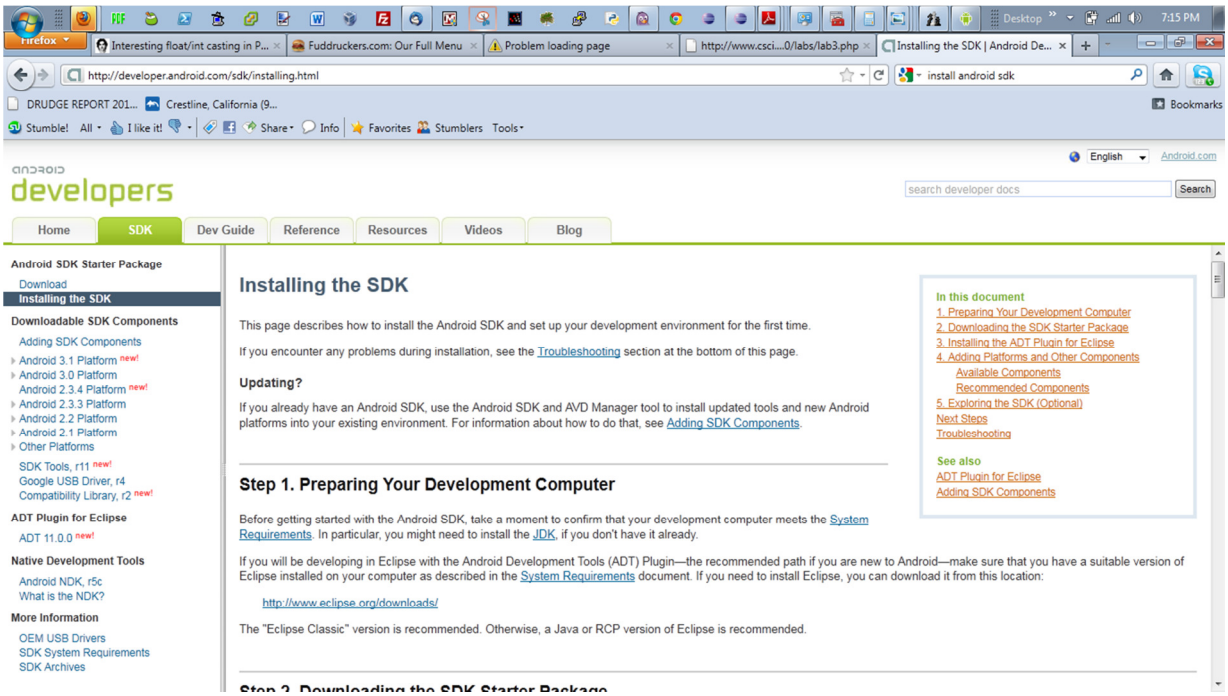
The first thing that was required was to research the steps to properly install the Android Tool Chain on Windows 7. We visited http://developer.android.com/guide/index.html . This website was a very useful guide that provided step by step instructions on everything required to develop software in Android.



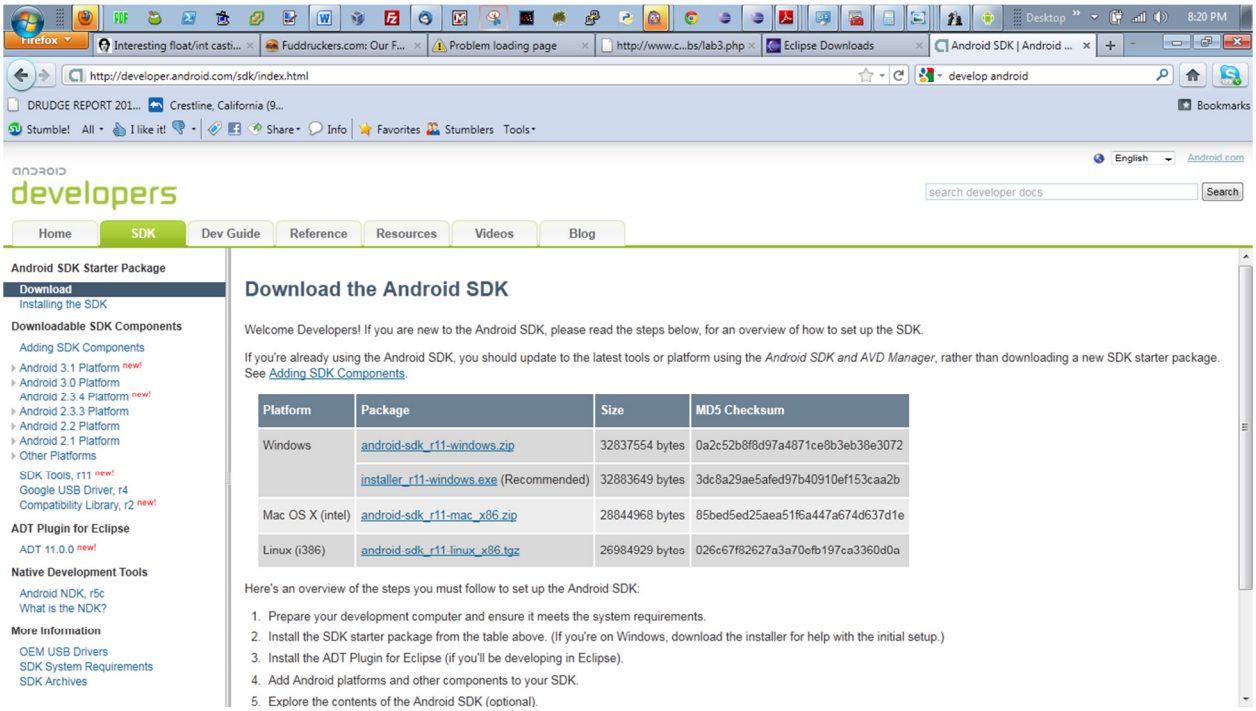
The first piece of software that needed to be installed was the Java JDK, which we found easily on the Oracle website and downloaded:



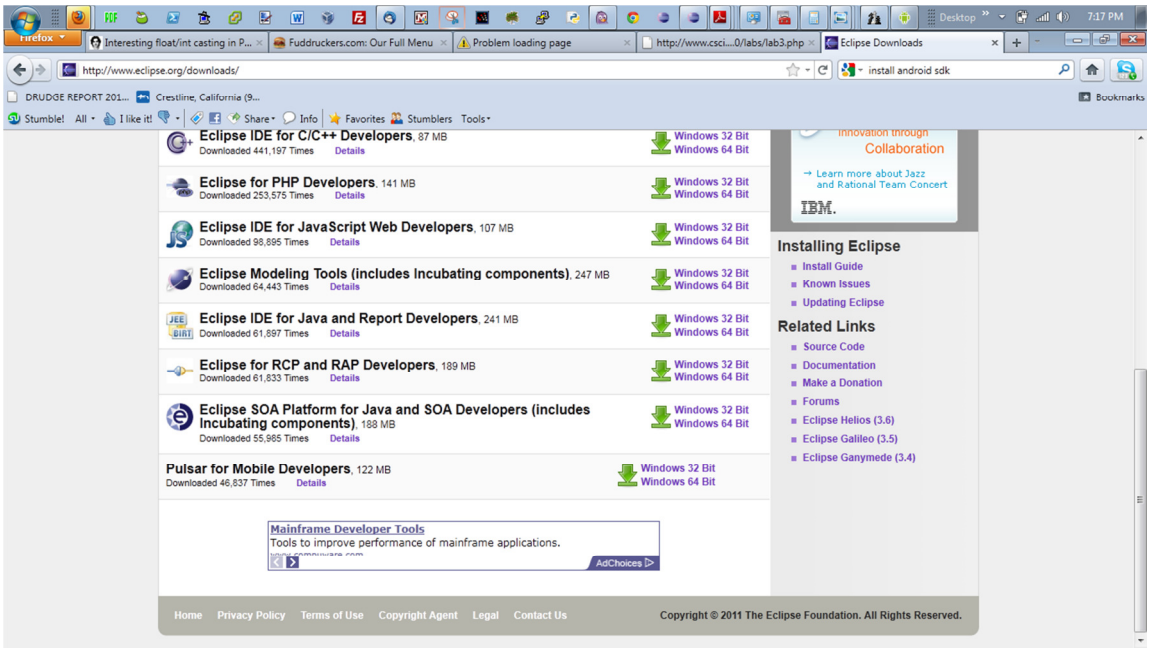
The next software item that was required was the Android Software Development Kit or SDK. We downloaded this from the Android site referenced earlier:



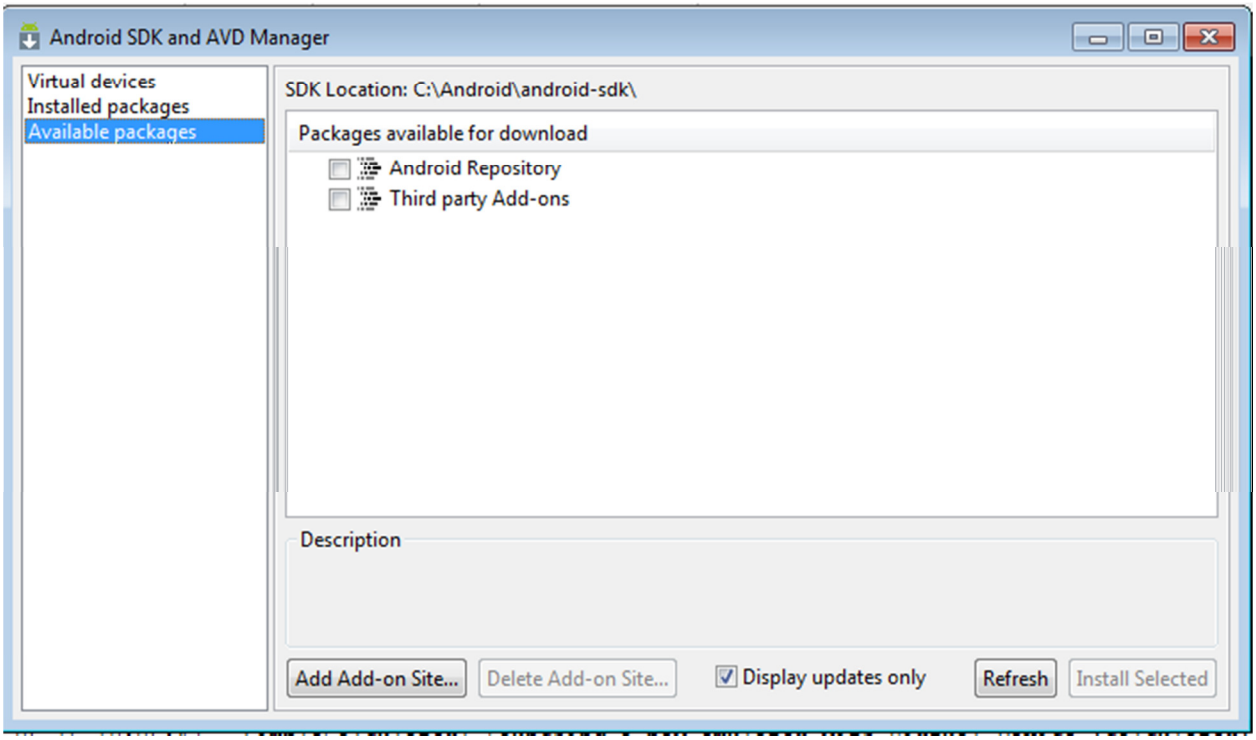
We used the executable installer for Windows.



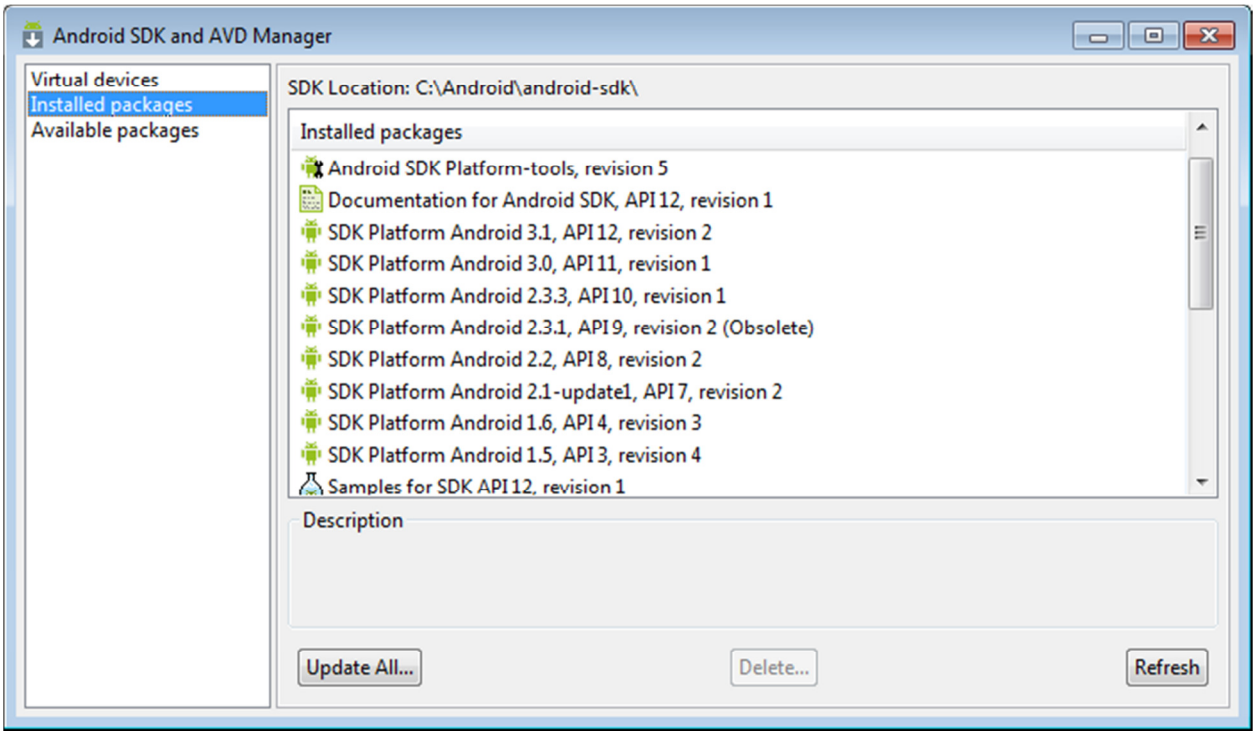
Next we installed Eclipse. We downloaded Eclipse for RCP and RAP Developers from the Eclipse.org website. We chose that version because it came bundled with a lot of the required plug-ins for Android Development, including the ADT plug in.



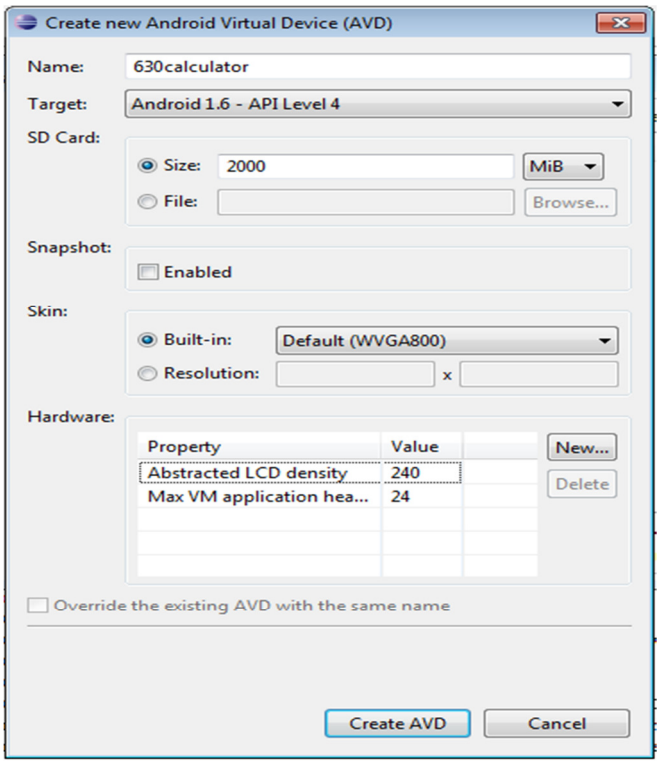
Next we updated all of the Android packages using Eclipse's Android SDK and AVD Manager:



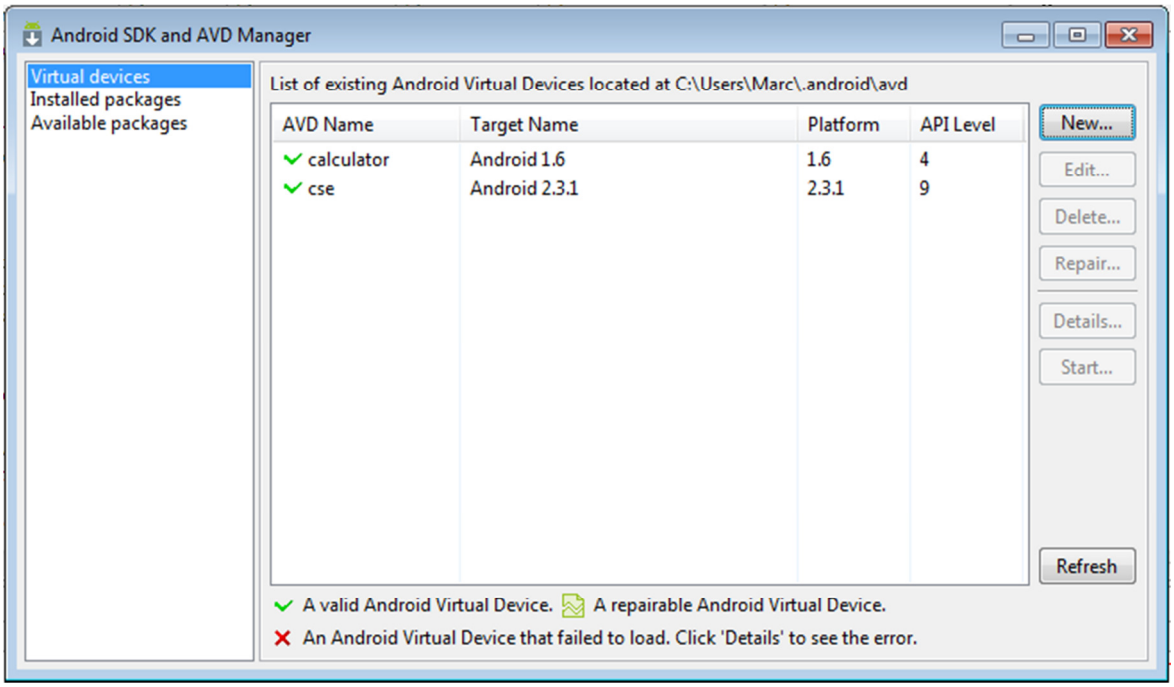
Next we checked to see that all of the necessary packages were installed:



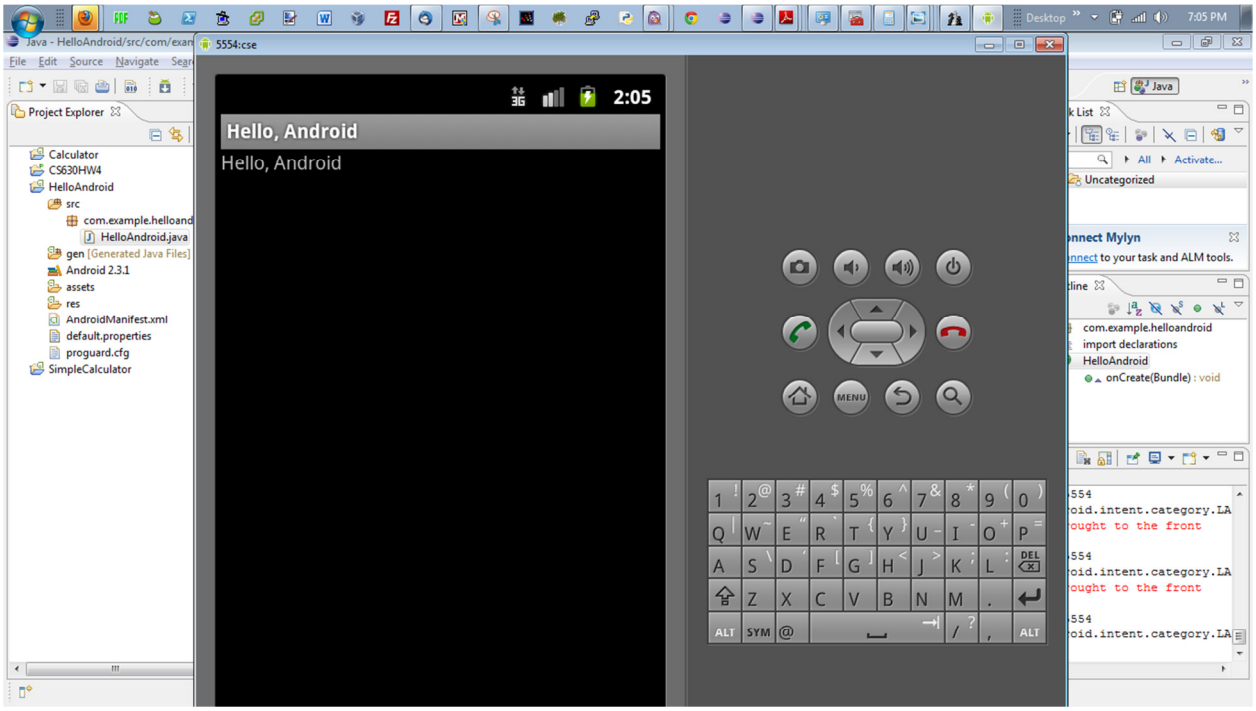
Finally we needed to create an Android Virtual Device or AVD. There are many options to choose from when creating a device.



We ended up creating a several different devices throughout the lab, and mainly developed in 1.6. This was because we found that 1.6 would run more quickly on our laptops.



Now that we had the Android SDK installed and working, we could use Eclipse to start developing Android software. The first program that we wrote was the Android "Hello World" Program which is called Hello Android. When completed it looked this this:



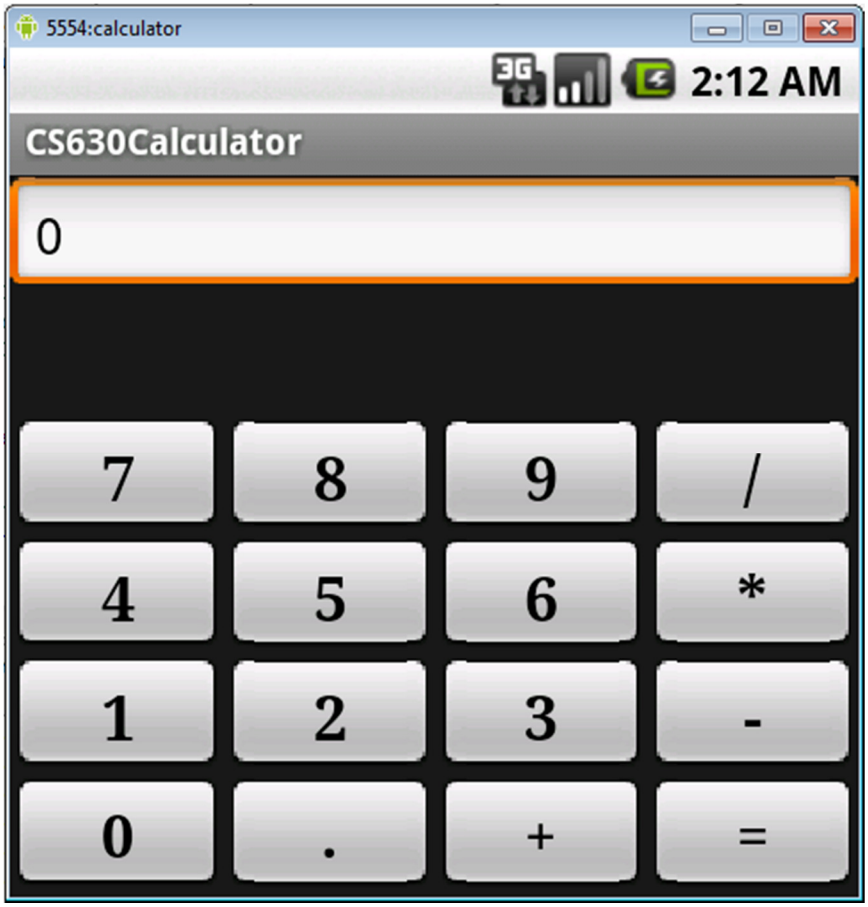
The code for Hello Android is readily available. Here is the copy that we used:

```
package com.example.helloandroid;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello, Android");
        setContentView(tv);
    }
}
```

Now that we had Hello Android working, we started working on the second part of the Android Lab, which was to create a basic calculator and get it to communicate with a server. After a lot of research and help from many different websites, as well as our fellow students, we were able to get a program up and running with all of the basic functions of a calculator. Here is a picture of it:



This is a portion of the code for our basic calculator:

```
import android.app.Activity;
import android.graphics.Color;
import android.os.Bundle;
import android.util.DisplayMetrics;
import android.view.KeyEvent;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnKeyListener;
import android.widget.Button;
import android.widget.EditText;

public class SimpleCalculator extends Activity {

		private final int MENUITEM_CLOSE = 300;
		private EditText txtCalc=null;
		private Button Zero=null;
		private Button One=null;
		private Button Two=null;
		private Button Three=null;
		private Button Four=null;
		private Button Five=null;
		private Button Six=null;
		private Button Seven=null;
		private Button Eight=null;
		private Button Nine=null;
		private Button Plus=null;
```

Now that we had a working calculator the final part of the assignment was to get the calculator to act as a client and communicate equations to a server to be computed on the server, and then returned to the Android calculator. We were able to successfully implement this and demonstrated it in class. Here is a screenshot of the demonstration:

We were able to successfully implement the required programs. We completed all aspects of the task on time and presented them to the professor.

Problems/Challenges Encountered:

There were a large number of difficulties encountered during the course of the lab. First, neither myself, nor my partner had ever used Java before. Also, neither of us had ever worked with Android before. The simple task of installing the complete tool chain to develop software in Android required several class sessions before we had Eclipse up and running with all of the Android SDK plugins and packages that would be needed for the development process. One of the problems we encountered during the installation process was that we were accidentally installing the SDK into Program Files in our Windows system; this created a number of problems with our path when trying to execute certain Android modules. After some research we were able to overcome this problem by uninstalling the software and reinstalling in to C:/ instead. Once we had the development environment running smoothly, we tried to run the Hello Android Program and ran into some problems because we were apparently using the wrong version of the Android Virtual Device. We started with 3.1, and we were having serious latency issues, and our program would not load. We created a new AVD based on version 1.6 and we were able to resolve the problem. Next we tried to develop the calculator. After a fair amount of internet research and help from students in class we were able to figure out how to get a piece of code running and modify it to work properly. The final problem that we encountered was getting the client calculator to communicate with the server. We ran into an initial problem when we tried to get incompatible versions of Android talking to each other. I am not sure that you can even get two device to talk to each other, we could not figure it out. Instead, based on advice from a student, we wrote a simple piece of java code to run on the server and interact with the calculator. After a fair amount of debugging and last minute prayer, we were able to complete the assignment in the last few minutes of the final class.