95-702 Distributed Systems
Project 2
Assigned: Sunday, September 12, 2010
Due: Friday, September 24, 11:59:59 PM

Objectives:

First, the student is introduced to TCP sockets, protocol layering, state maintenance, and finite state machines.

Second, the student is asked to build an Ajax web application that executes a protocol that maintains state for multiple users.

Third, the student is introduced to “Web Services”. Web Services provide a callable interface for applications running on distributed systems. The objective is to give students hands-on experience with both SOAP based web services and REST based web services.

The objectives of the fourth and fifth tasks are to introduce the student to interacting with web resources via a mobile platform using a native application. Mobile platforms are increasingly important components of distributed systems and this space is expected to be a place of great innovation.

The objective of the sixth task is to highlight the ease of retrieving and processing of structured data.

And finally, as in all projects this semester, this project requires you to answer a series of reflective questions on Blackboard concerning the solutions you have built. Your answers should clearly and accurately address the non-functional characteristics of your solution, demonstrating a nuanced comprehension of course content and explaining the technical aspects in relation to potential real-world applications. You will be asked to reflect and comment on several important characteristics of distributed systems. For this project, the non-functional characteristics that must be considered include message formatting (REST and SOAP), protocol layering and modeling, performance, reliability and interoperability. After completing the project programming tasks, go to Blackboard and answer the questions in the “Project 2 Reflection” assignment.
For each project task, software documentation is required. The software that you write (HTML files, Java files and so on) must contain comments that describe what each significant piece of code is intended to accomplish. Points will be deducted if code is not well documented.

The graders will be asked to follow the rubric found on the course schedule.

Task 1

Protocol layering and state maintenance are central issues in the study of distributed systems. In project 2, your first task is to experiment with the knock knock joke protocol using Transmission Control Protocol (TCP) sockets and a finite state machine that maintains state.

At the bottom of this handout are three Java classes that make up a small client server system. You should study each of these classes and get them running. The client project will be named KnockKnockSocketClientProject and the server will be named KnockKnockSocketServerProject. The KnockKnockProtocol.java file will reside in the server project and is best viewed as a finite state machine. Your task is trivial. Simply add detailed comments to the code and submit working client and server projects to the assignment section of blackboard.

Task 2

Build a web application project named KnockKnockWebAppProject using Glassfish. This web application will contain the files KnockKnockServlet.java, KnockKnockProtocol.java, and KnockKnock.js for the Ajax Javascript code. You may also include a CSS file if you like, but the application will not contain any separate HTML or JSP files. The servlet will be accessed by a browser using the URL:

http://localhost:8080/KnockKnockWebAppProject/KnockKnockServlet

The KnockKnockProtocol.java file will be the exact same file as the one that appears below. Of course, your comments will be added from Task 1.

On the initial visit, the servlet will create a new KnockKnockProtocol object and add it to an HttpSession object. See the course slides for an HttpSession example.

Once the initial full page is loaded, all subsequent communication between the browser and the web application will be via Ajax (i.e. XMLHttpRequest). Because the browser
need not be doing any additional work upon making requests, you can use a synchronous XMLHttpRequest.

The first response will automatically return a cookie to the browser. You should ensure that this is the case by viewing cookies in your browser.

You do not need to create the cookies in your code. Glassfish will generate cookies for you when it sees that you are using HTTPSessions.

Experiment and try creating new sessions by clearing old cookies. The HTTPSession object will associate a particular KnockKnockProtocol object with a particular cookie.

The servlet will first respond with the prompt "Knock Knock" followed by a text box. The prompt "Knock Knock", being sent to the browser, must be retrieved by the servlet from the KnockKnockProtocol object associated with this session. When the user responds to the initial "Knock Knock" the same servlet is executed again. The KnockKnockProtocol object associated with this user session is accessed from the HttpSession object. This approach allows us to have several knock knock users playing knock knock jokes at the same time. There will be a separate KnockKnockProtocol object for each user. Each of these will track the state of the protocol.

The game should be playable and error messages displayed on the browser as they are generated by the protocol handler. You should experiment with more than one browser using the application at the same time. You will need either two browsers (IE and Opera for example) or you may wish to visit the web application from a separate machines. In the latter case, you need to be connected to the internet and replace the string 'localhost' with the IP address of the web server.

Task 3

The traditional web has had a huge impact on society. It is interoperable, allowing diverse systems to interact. It may be secured with protocols such as SSL. It supports concurrency and has been able to scale. With the advent of “Web Service” – not to be confused with “Web Applications”, it has shown itself to be extensible.

In Task 2 we worked with an Ajax web application using HTTP over TCP over IP. In this task we will work with SOAP over HTTP over TCP over IP. Again, we will allow our users to play knock knock jokes with a server.
In preparation, here is a short tutorial on how we might build a simple web service using Netbeans 6.X.X and Glassfish. The resulting web service uses SOAP and WSDL.

1) File/New Project/Java Web/Web Application/PrjName/Location/
   Glassfish V2/Finish
2) Right Click Project Node/New/Web Service/WSName/PkgName/Finish
3) Design/Add Operation/Provide method name, parameter names and types
4) Source/Complete details of method.
5) Right click Project Node/Deploy
6) Expand Web Services node/Right Click Service name/Test Web Service
7) Use the browser to test and examine input and output messages.
8) View WSDL and copy the WSDL URL to the clipboard.

At this point, we can write a web service client. The client might be a stand alone application or a servlet. Here we will write a simple stand alone application.

1) File/New Project/Java/Java Application/PrjName/location/Finish
2) Right click project/New Web Service Client/WSDL URL/paste from clipboard
3) Note the Web Service Reference node in project tree.
4) Expand Web Service Reference down to desired method.
5) Drag and drop into appropriate source location.

On the course schedule, there is a video by Arun Gupta showing this process. His application uses a database and Java’s Persistence API. But most of the steps are the same. You might note that in Gupta’s video he stops just short of writing a client. He simply tests the service with a tester web application. But since the WSDL is available, he could write a client in the language and platform of his choice.

Create a new project named KnockKnockWebServiceProject. Within this project create a web service named KnockKnockWebService. Write a web service method with the signature:

   public synchronized String play(String fromClient, String userID)

We may have many simultaneous users playing knock knock jokes. The play method is marked as synchronized so that only one thread may execute the code at a time.

The first parameter will represent the text input from the player. A typical input would be "Little Old Lady Who?".
The second parameter represents the player's identification. This will be passed on each call from the player. This allows the play method to track each player's progress through the knock knock protocol. In the web application (Task 2), we used cookies to do the same thing. You may assume that each call contains a unique identifier describing the caller. Perhaps this is a screen name or a log in ID.

The response from the play method is a string that represents the protocol handler's response. A typical response would be "Knock Knock!". Behind the scenes, the web service framework will place “Knock Knock” into an XML document (with tags from the SOAP standard) and this document will be placed after the headers in an HTTP response message. This HTTP response message will be written to a TCP socket. The reader is invited to describe a call to the play method in the same manner as done here with the play method’s response.

The play method makes use of the java.util.TreeMap class. A TreeMap object holds a set of name value pairs. The name is the userID and the value is the protocol handler associated with the userID. In this way, the service can track simultaneous users.

Write a console application in Java that allows users to play knock knock jokes with the web service that you developed. This class will have a main routine and will be called KnockKnockWebServiceClient. It will live in a project called KnockKnockWebServiceClientProject.

The console application (not the web service) will be executed with the command "java KnockKnockWebServiceClient userID". The execution will proceed in the same way as the original socket based solution.

In order to establish a command line parameter using Netbeans, right click the project node and select properties. Select run and enter the command line arguments. This is how you can get the userID into your program.

The knock knock socket-level code from Sun is the same as that used in Task 1 and is included below.

Task 4

Note that solutions to Tasks 4 and 5 do not use the browser on Android. They are both native interactive Android applications that make HTTP requests and receive HTTP responses.

On the schedule there is document describing an Android application that visits a web site for palindrome recognition. The file is named AndroidGettingStarted2.text.
Get this system running. You may use your palindrome servlet from project 1. Submit a screen shot showing the Android emulator interacting with your server. You need not submit any code. It is all provided. This is a trivial exercise designed to get you started. The grader will simply look for the screen shot that shows you were successful.

Task 5

Write an Android application that plays knock knock jokes with a web service. Name the Eclipse project KnockKnockAndroidProject and submit it to the assignment section of Blackboard. The server side project will be named MobileAccessToWSProject.

Why is this task a little tricky? Well, at this point in time there is little support for SOAP on Android and this task requires you to use a web service. So, what do you do?

One approach that you might consider is to write a web application using JSP's or servlets that interacts with the web service and provides a JSON or XML response to Android. It is very easy to make your Java servlet or JSP page a client of the web service that we already wrote. The conversation between the servlet and the web service will be SOAP based but the conversation between Android and the servlet will use HTTP (passing along parameters as name value pairs.)

Our solution uses a JSP page to talk to the SOAP based web service. The JSP page returns a small XML document to the phone. The phone parses the XML for the server's response and displays it on the screen.

Another approach would be to write a REST Web Service using Netbeans, Glassfish and the Jersey API's. There are pointers on the schedule to information about Jersey. The REST web service would act as a wrapper around the SOAP based web service that we already wrote. We have provided a sample REST web service and a sample client that uses it below. Note that we have not yet worked through a REST solution and that the JAX-RS is less mature than JAX-WS. So, you need to take that into account before trying this approach.

Task 6

In Project 1 Task 4, we used a method called “screen scraping” to access photos from Flickr. This is a notoriously brittle approach. If the HTML that Flickr provides changes, perhaps to provide a different browser experience, our code may break.

For Task 6, you should replicate your application from Project 1 Task 4,
only use Flickr’s web services instead of screen scraping. (You will find this is a much easier than screen scraping. And it will be far less brittle.)

In other words, the requirements for the Task are the same, except:
Instead of reading text from streamed input, you should use the following to directly fetch and parse an XML source:

javax.xml.parsers.DocumentBuilderFactory
javax.xml.parsers.DocumentBuilder
org.xml.sax.InputSource

You will retrieve an XML document and this can be processed using standard XML Document Object Model API’s.
Java provides the following document object classes to search and navigate the XML to find the components (XML Elements and Attributes) that you need:

org.w3c.dom.Document
org.w3c.dom.NodeList
org.w3c.dom.Element

The main Flickr API page can be found at: http://www.flickr.com/services/api/

You will use the flickr.photos.search in a way similar to how you searched in Project 1 Task 4. However note that the arguments are named differently.

It is highly recommended that you use the API Explorer to help build the REST URL for the search.

See: http://www.flickr.com/services/api/explore/?method=flickr.photos.search

You should *not* have to deal with Flickr authentication. You *will*, however, have to have an api_key. You can get one at:
http://www.flickr.com/services/apps/create/apply
Also notice the photo source URL formats at:
http://www.flickr.com/services/api/misc.urls.html
Summary:

There will be several Netbeans projects and one Eclipse project.
The projects will be named as follows:

Task 1
- KnockKnockSocketClientProject
- KnockKnockSocketServerProject

Task 2
- KnockKnockWebAppProject

Task 3
- KnockKnockWebServiceProject
- KnockKnockWebServiceClientProject

Task 4
- Screen shot showing an Android application using the Palindrome servlet

Task 5
- An Eclipse Project named KnockKnockAndroidProject
- MobileAccessToWSPProject

Task 6
- FlickrWebServiceClientProject

The submission should be a single zip file of all six tasks plus screen shots.
KnockKnockClient.java
=====================  
// From Sun Microsystems
import java.io.*;
import java.net.*;

public class KnockKnockClient {
    public static void main(String[] args) throws IOException {

        Socket kkSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            kkSocket = new Socket("localhost", 4444);
            out = new PrintWriter(kkSock
et.getOutputStream(), true);
            in = new BufferedReader(new
    InputStreamReader(kkSocket.getInputStream()));

        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: localhost");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: taranis.");
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(new
    InputStreamReader(System.in));
        String fromServer;


String fromUser;

while ((fromServer = in.readLine()) != null) {
    System.out.println("Server: "+ fromServer);
    if (fromServer.equals("Bye."))
        break;

    fromUser = stdIn.readLine();
    if (fromUser != null) {
        System.out.println("Client: "+ fromUser);
        out.println(fromUser);
    }
}

out.close();
in.close();
stdIn.close();
kkSocket.close();
}
KnockKnockServer.java

====================
// From Sun Microsystems
import java.net.*;
import java.io.*;

public class KnockKnockServer {
    public static void main(String[] args) throws IOException {

        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4444.");
            System.exit(1);
        }

        Socket clientSocket = null;
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException e) {
            System.err.println("Accept failed.");
            System.exit(1);
        }

        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                new InputStreamReader(}
clientSocket.getInputStream());
String inputLine, outputLine;
KnockKnockProtocol kkp = new KnockKnockProtocol();

outputLine = kkp.processInput(null);
out.println(outputLine);

while ((inputLine = in.readLine()) != null) {
    outputLine = kkp.processInput(inputLine);
    out.println(outputLine);
    if (outputLine.equals("Bye."))
        break;
}
out.close();
in.close();
clientSocket.close();
serverSocket.close();
}
public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;

    private int state = WAITING;
    private int currentJoke = 0;


    private String[] answers = { "Turnip the heat, it's cold in here!",
                                "I didn't know you could yodel!",
                                "Bless you!",
                                "Is there an owl in here?",
                                "Is there an echo in here?" };

    public String processInput(String theInput) {
        String theOutput = null;
        return theOutput;
    }
}
if (state == WAITING) {
    theOutput = "Knock! Knock!";
    state = SENTKNOCKKNOCK;
} else if (state == SENTKNOCKKNOCK) {
    if (theInput.equalsIgnoreCase("Who's there?")) {
        theOutput = clues[currentJoke];
        state = SENTCLUE;
    } else {
        theOutput = "You're supposed to say \"Who's there?\"! " +
            "Try again. Knock! Knock!";
    }
} else if (state == SENTCLUE) {
    if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
        theOutput = answers[currentJoke] + " Want another? (y/n)";
        state = ANOTHER;
    } else {
        theOutput = "You're supposed to say \"" +
            clues[currentJoke] +
            " who?\"" +
            "! Try again. Knock! Knock!";
        state = SENTKNOCKKNOCK;
    }
} else if (state == ANOTHER) {
    if (theInput.equalsIgnoreCase("y")) {
        theOutput = "Knock! Knock!";
        if (currentJoke == (NUMJOKES - 1))
            currentJoke = 0;
        else

currentJoke++;
state = SENTKNOCKKNOCK;
} else {
    theOutput = "Bye.";
    state = WAITING;
}
} return theOutput;
Sample REST Web Service (Optional)
====================================

/* To create:

    Build a web project.
    Right click the project and select New/Other/Web Services/RESTFull web service from patterns.
*/

package edu.cmu.andrew.mm6;

import com.sun.jersey.spi.resource.Singleton;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.PathParam;
import javax.ws.rs.Consumes;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import java
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

/**
 * REST Web Service
 */

* @author mm6
*/
// The default is a 'singel call' object. That is,
// the object is created and destroyed on each call.
// A singleton has one instance and remains in
// memory.

@Singleton
@Path("/register")
public class Register {

  private static int total = 0;
  private static int max = 10000;
  private static int waitListed = 0;

  /** Creates a new instance of Register */
  public Register() {
  }

  /**
   * Retrieves representation of an instance of edu.cmu.andrew.mm6.Register
   * @return an instance of java.lang.String
   */
  @PUT
  @Produces("text/plain")
  public String registerStudent() {

    System.out.println("registerStudent() method called total = " + total);
    boolean inClass = false;
    if (total < max) {
      total++; inClass = true;
    } else


waitListed++;
if(inClass) return "Registered for class at position " + total;
else return "WaitListed at position " + waitListed;

Sample REST Web Service Client (Optional)
========================================

/* This is a stand alone Java program that
 * is used to visit the service above.
 * *
 * In order to run this on Netbeans 6.8 we must add
 * the Jersey Library:
 * *
 * Right click the project node.
 * Choose properties.
 * Choose libraries.
 * Add the following libraries as Compile time and Run time libraries:
 * Jersey 1.1 (JAX-RS RI)
 * JAX RS 1.1
 * This program should now run.
 * *
 * See the tutorial at:
 * *
http://blogs.sun.com/enterprisetechtips/entry/consuming_restful_web_services_with
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;

/**
 * @author mm6
 */

class Main {

    public static void main(String[] args) {

        Client client = Client.create();
        for(int k = 0; k < 10002; k++) {
            String s = webResource.put(String.class);
            System.out.println("The returned data is \n" + s);
        }
    }
}