Enabling Smart Homes Using Web Technologies

PhD Thesis (Draft)

Author: Andreas Kamilaris
kami@cs.ucy.ac.cy

Professor: Andreas Pitsillides

NETworks Research Laboratory (NetRL)
Department of Computer Science
University of Cyprus

November 2012

Appendix A

Application Framework Implementation Details

The application framework for smart homes (see Chapter 4) has been developed in Java using the Eclipse[1] development platform and it follows a modular architecture with three main layers:

1. **Device layer**, which administers embedded home devices.
2. **Control layer**, which initializes, controls and checks the framework.
3. **Presentation layer**, which maintains a Web server following the REST principles.

These three layers are implemented as Java packages and hold the Java classes of the application framework. In other words, each Java class of the framework is categorized, according to its purpose, in one of these three main packages. Below we list the (most important) classes of the application, organized based on the main layer to which they belong.

Device Layer Classes:

- **Devices**. It represents all the devices that have been discovered in the home environment.
- **Device**. It is the thread responsible for managing the communication with some home device.
- **MessageQueue**. It is responsible for implementing the request queue mechanism of each device thread.
- **Request**. It defines the formats and types of the request messages exchanged between the framework and the embedded devices.
- **Response**. It holds the response messages received from physical devices.
- **Resources**. Holds all the services offered by some device.
- **Resource**. A particular service offered by some device.
- **Driver**. Abstract class that defines a standardized interface for physical communication with home devices.
- **TinyosIPv6Driver**. Driver class for enabling communication with IPv6-enabled sensor motes, operating with TinyOS.

[1]http://www.eclipse.org/

- **TinyOSParser**. Parses the request/response messages to/from TinyOS/IPv6-enabled sensor motes.
- **PloggMonitor**. Responsible for the communication and management of Ploggs smart power outlets.

Control Layer Classes:

- **Core**. Acts the main dispatcher of the system, holds the main class for running the framework. Responsible for all the initializations and for maintaining the application's routines.
- **Server**. Maintains HTTP libraries, needed by the Web server.
- **Constants**. This class keeps all the parameters and configurations of the application framework.
- **AsyncToSync**. Helper class for facilitating the transition between synchronous Web operation and asynchronous smart home functionality.
- **WADLparser**. Helper class used to parse the WADL files (see Section 5.2.2), which are transmitted from the home devices during their discovery (see Section 5.2.1).

Presentation Layer Classes:

- **Server**. The Web server of the application framework.
- **RESTApplication**. Routes HTTP requests to the most appropriate Java classes to serve them. It provides REST functionality to the framework.
- **AbstractResource**. Abstract class that generates representations of resources in the standard formats available (HTML, JSON, XML).
- **DevicesResource**. Creates a representation of the physical devices that are available in the home environment.
- **InvokeDeviceResource**. Invokes a request by forwarding it to the appropriate physical device. It sends the response then back to the Web client who created the request.

These are the main classes that compose the application framework. Various smaller classes exist, which only perform some minor tasks. Moreover, the application framework involves a *Simulation* package, which includes classes related to the emulator that is used in the analysis and evaluation procedure of this thesis (see Chapters 6 and 7). An abstract class diagram, displaying the main classes of the framework as well as the relationships between them, is illustrated in Figure A.1.

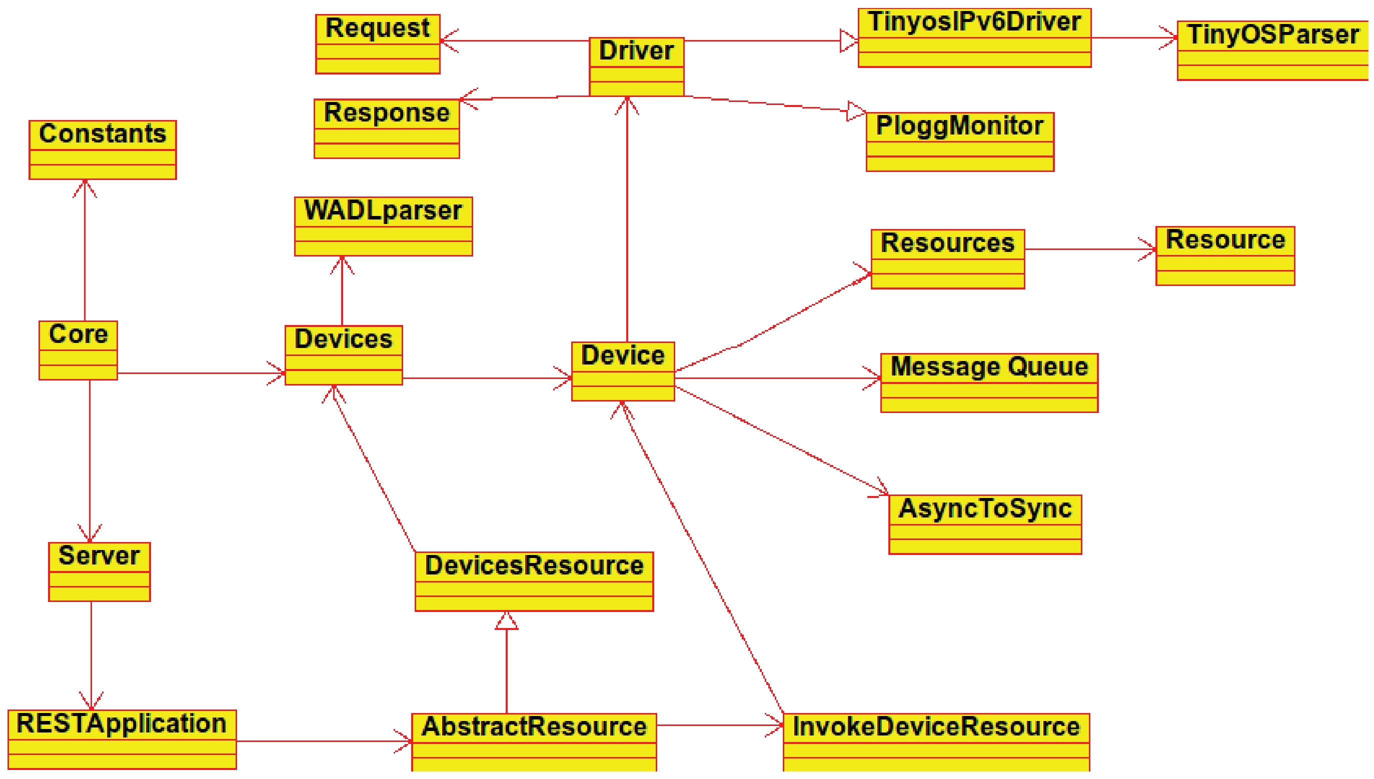

Figure A.1: A class diagram of the application framework.

Appendix B

Application Framework Configuration Parameters

Here we list the configuration parameters that define the operation of the application framework for smart homes (see Chapter 4). These parameters are adjusted inside the *Constants* class, which is located at the package *controlLayer.libraryCode*.

- **REQUEST_RETRANSMISSION_INTERVAL**. Specifies the amount of time of the request queue retransmission interval α (see Section 6.1) in milliseconds.
- **TRANSMISSION_FAILURE**. Defines the percentage of transmission failures during the operation of the framework. It is used mostly for analysis/evaluation purposes.
- **DEV_ALIVENESS_CHECK_TIME**. The interval (in minutes) between the aliveness checks performed by the framework to determine if embedded devices are still alive and operate correctly (see Section 4.2.3). Default value is 5 minutes.
- **DEV_MAX_CACHE_DELAY_TIME**. Defines cache freshness period (in seconds), i.e. the amount of time a cached value remains fresh (see Section 4.2.4). The caching mechanism suspends by setting this value equal to 0. Default value is 10 seconds.
- **DEV_REQUEST_MAX_ATTEMPTS**. Maximum number of retransmission attempts, in case of transmission failures, before a home device is considered unavailable (see Section 6.1). A value of 5 is usually suitable for indoor environments and static topologies.
- **FAILURE_MASKING_MECHANISM**. A boolean parameter that decides whether to employ the failure masking mechanism in case of device failures (see Section 4.2.3).
- **PRIORITIES**. A boolean parameter that specifies whether the application framework employs prioritized requests (see Section 6.4.5).
- **HIGH_PRIORITY**. The integer value assigned at a high-priority request, according to the Algorithm 2. Similar parameters exist for normal- and low-priority requests.
- **PROBABILITY_HIGH_PRIORITY**. Percentage of incoming requests having high priority. Used mostly for analysis and evaluation purposes. Similar parameter exists for low-priority requests. Obviously, the rest requests are labeled with normal priority.
- **PLOGGS_STREAMING_DELAY**. Defines the time interval between electricity measurements of the Ploggs smart power outlets (see Section 5.1.2). Default value is 1 minute.
- **PLOGGS_DISCOVERY_INTERVAL**. Specifies the time interval (in seconds) for scanning the house for new Ploggs smart power outlets. Default value is 5 minutes.

Appendix C

Application Framework Installation Instructions

The application framework for smart homes (see Chapter 4) has been developed in Java using the Eclipse[1] development platform. Hence, it is available as an Eclipse project (you can email the author at kami@cs.ucy.ac.cy to obtain the code of the application framework) and may be imported very easily to the Eclipse SDK (File->Import->Existing Projects into Workspace). The framework is very lightweight, requiring only 2.7 Mbytes for its full installation. The development of the framework and all the analysis and evaluation efforts have been performed using Eclipse Galileo (Eclipse SDK version 3.5.2).

To run the Eclipse project of the application framework, a Java Running Environment (JRE) must be installed on the computing device that hosts the application framework. Furthermore, the Java Communications API[2] needs to be installed to support serial-to-USB communication with any embedded device that would act as the base station, enabling the communication with remote wireless devices. A good tutorial for installing the Java Comm API is available at:

`http://www.agaveblue.org/howtos/Comm_How-To.shtml`

The Eclipse project comes along various JAR files, which are needed for its proper operation. These JAR files are:

- tinyos.jar - Used for support of TinyOS 2.x messaging.
- RXTXcomm.jar - Needed for the communication with the sensor mote acting as the base station, from the serial-to-USB port.
- comm.jar - Also needed for the serial-to-USB communication.
- org.simpleframework.jar - Supports HTTP communication with Web entities and XML parsing.
- org.restlet.jar - Supports the Restlet[3] Java-based REST framework.
- org.json-2.0.jar - For JSON encoding/decoding and parsing.
- com.noelios.restlet.jar - Supports a Web server implementation.

[1]http://www.eclipse.org/
[2]http://www.oracle.com/technetwork/java/index-jsp-141752.html
[3]http://www.restlet.org/

Furthermore, the project includes a *wadl* folder, which holds the WADL description files used to describe the services offered by Telosb sensor motes and Ploggs smart power outlets (see Section 5.1).

After the Eclipse project of the application framework is properly imported in Eclipse, the user can run the project as a Java application. The main class for running the Java application is the *Core* class, located in *Control Layer* package. Before running the application, the user needs to set some basic parameters, by navigating to *Run->Run Configurations* and selecting the project. Then, when opening the *Arguments* tab, he must type the following information:

```
<Application Framework Name><Location><Domain Name or IP address><Port>
                    {<Device Type><USB Port>}
```

Device Type specifies an embedded technology to be included in the current run of the framework and *USB Port* defines the port of the device that acts as the base station. More than one embedded technologies may be supported. An example setting of the parameters for running the application framework, using both Telosb sensor motes and Ploggs smart power outlets is the following:

```
ApplicationFramewok UniversityOfCyprus localhost 8080 Telosb USB0 Ploggs USB1
```

Now you should be able to run successfully the application framework and automate your house with reliability and satisfactory performance, with support for all your house members!

Appendix D

Synchronous/Asynchronous Translation

The following code shows some parts of the synchronous/asynchronous synchronization mechanism (see Section 4.2.2), which was installed on the application framework in order to map synchronous HTTP requests, coming from Web clients (home tenants) at the Web server module of the application framework, to asynchronous operations occurring at the device layer of the framework, involving the physical interaction with home devices for satisfying the client requests.

The code has been developed in Java and it is embedded inside the Device class, which is the class thread responsible for managing embedded home devices.

Listing D.1: Code for supporting synchronous/asynchronous operation at the application framework for smart homes

```java
package deviceLayer;

public class Device extends Observable
    implements XMLRepresentable, Runnable {

  // locking stuff...
  private static Long msgToken = new Long(1);

  /**
   * @return a token for the message id.
   */
  public static synchronized long getToken() {
    synchronized (msgToken)
      return msgToken++;
  }

  /**
   * dispatch a response to the synchronizer.
   * @param r the low level response.
   */
  public static synchronized void dispatchResponse(Response r) {
    AsyncToSync lock = synchronizer.get(r.getRequestID());
    if (null == lock)
      return;

    synchronized (lock) {
      lock.setResponse(r);
      lock.notifyAll();
    }
  }

  /**
   * helper class to synchronize the async communication to tinyos/
      ploggs
   *
   */
  public class AsyncToSync {

    /** my token. */
    private final long token;

    /** the response onto my request. */
    private Response response = null;

    /**
     * constructor.
     */
    public AsyncToSync() {
      token = Device.getToken();
    }

    /**
     * @return my token.
     */
    public long getToken() {
      return token;
    }

    /**
     * @return the response
     */
    public Response getResponse() {
      return response;
    }

    /**
     * @param response the response to set
     */
    public void setResponse(Response response) {
      this.response = response;
    }
  };

  /** a hash map containing the synchronizer objects. */
  private static Map<Long, AsyncToSync> synchronizer =
    new ConcurrentHashMap<Long, AsyncToSync> ();

  // end of synchronizing

  /*  Handles responses from home devices by forwarding them to the
     appropriate Web Client who made the Request */
  public void handleResponse(Response r){
    System.out.println("Handling␣normal␣Response␣for␣service:"+r.
       getServiceName());
    dispatchResponse(r);
  }

  /* adds Request r in Request Message Queue */
  public void addRequest(Request r){
    this.msgQueue.addRequestMessage(r);
  }

  public String handle(org.restlet.data.Response response,
      org.restlet.data.Request request) {

    ... CODE THAT WAS OMITTED ...

    Request re = new Request(deviceID,resourceName,method,params,
       values, false,0);
    Response r = waitSynchronous(re);
    if (null == r) {
      response.setEntity(new StringRepresentation(Constants.NACK));
    } else {
      log.debug(r.getResult());
      response.setEntity(new StringRepresentation(r.getResult().
         toString()));
    }
    return null;
  }


  public Response waitSynchronous(Request request) {
    // handle a request with lock wait...
    AsyncToSync lock = new AsyncToSync();
    synchronizer.put(lock.getToken(), lock);
    try{
      request.setRequestID(lock.getToken());
      addRequest(request);
      synchronized (lock) { lock.wait(); }
    } catch (Exception e) {
      e.printStackTrace();
      synchronizer.remove(lock);
    }
    synchronizer.remove(lock.getToken());
    return lock.getResponse();
  }
}
```

Appendix E

Sensor Programming without REST

We list the code that needs to be written in Java without employing REST, to achieve the same result as the second physical mashup provided in Section 4.2.5, developed in shell scripting.

Listing E.1: Sensor programming in Java to implement a distributed rule on sensor motes.

```
try{
  String deviceID = "sensor8";
  URL url = new URL("http://[" + deviceID + "]/Temperature");
  BufferedReader in = new BufferedReader(new
              InputStreamReader(url.openStream()));
  String str;
  while ((str = in.readLine()) != null) {
    output += str;
  }
  in.close();
}
} catch (Exception e) {

}

char[] checkVal =output.toCharArray();
String temperature = "";

for(int i=0; i < checkVal.length; i++){
  if(Character.isDigit(checkVal[i]))
    temperature += checkVal[i];
      else
            // use only digits for temperature
}

if(temperature < 20){
  String payload ="";
  // Construct payload data
  for(int i=0; i < request.getParameters().size(); i++){
    if(i == 0)
      payload = URLEncoder.encode(request.getParameters().get(i), "UTF
         -8") + "=" + URLEncoder.encode((String)request.getValues().
         get(i), "UTF-8");
    else
      payload += "&" + URLEncoder.encode(request.getParameters().get(i
         ), "UTF-8") + "=" + URLEncoder.encode((String)request.
         getValues().get(i), "UTF-8");
  }
          deviceID = "sensor5";
  URL url = new URL("http://[" + deviceID + "]/Light");
  URLConnection conn = url.openConnection();
  conn.setDoOutput(true);
  OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream
     ());
  wr.write(payload);
  wr.flush();
  BufferedReader rd = new BufferedReader(new
            InputStreamReader(conn.getInputStream()));
  String line;
  while ((line = rd.readLine()) != null) {
    output += line;
  }
           // process output
  wr.close();
  rd.close();
}
```

In the physical mashup in Section 4.2.5, only seven lines of code were needed to implement a distributed rule on sensor motes, for checking the temperature of the room and turning on some LED it this temperature is less than 20 degrees Celsius. However, in Listing E.1, 50 lines of code were used to perform the same task, employing two sensor motes.

In the scenario when the sensor motes are not IPv6-enabled, this distributed rule would need approximately 200 lines to be executed. This happens because the 6LoWPAN stack makes the following command possible:

URL url = new URL("http://[" + deviceID + "]/Temperature");

If not, a Java application would need to include also a *TinyOS driver*, implementing the *MessageListener* interface. It would also need a *TinyOSParser* class, to parse the contents of the message to the appropriate form understood by the sensor mote. Additionally, it would need the *MoteIF* class, for communicating with the mote, through the serial-to-USB port. We attach some of the additional code needed below.

Listing E.2: Additional code in Java in case 6LoWPAN is not supported.

```
public void sendMessage(String nodeid, char message_type, String
   payload) throws IOException{
  private MoteIF mote;
  TinyOSParser parser = new TinyOSParser();
  SmartDeviceMsg smsg = new SmartDeviceMsg();
  smsg.set_nodeid(FRAMEWORK_ID);
  smsg.set_subject((short) message_type);
  String content = parser.parseRequestData(payload);
  switch(message_type){
    case 'R':{  // Service Request message
          try {
            char[] cdata = content.toCharArray();
            short[] sdata = new short[cdata.length];
            for(int i=0; i< cdata.length; i++)
              sdata[i] = (short) cdata[i];
            smsg.set_data(sdata);
            synchronized(mote){
              mote.send(Integer.parseInt(nodeid), smsg);
            }
          } catch (IOException e) {
            System.err.println("Cannot␣send␣Service␣Request␣Message␣to
               ␣mote");
            e.printStackTrace();
          }
          break;
        }
        default:{
          System.err.println("Unknown␣message␣subject.␣Sending␣failed.
             ");
        }
  }
}
```

Appendix F

Programming Telosb Sensor Motes

For people who are not familiar with sensor programming, we list here all the necessary steps needed in order to install IPv6/6LoWPAN support on Telosb sensor devices, transforming them then into embedded Web servers.

At first, one must install the latest version of TinyOS[1] (currently TinyOS 2.1.1). A Web page with simple instructions for installing TinyOS, as well some additional needed libraries is the following:

```
http://www.tinyos.net/tinyos-2.x/doc/html/install-tinyos.html
```

After the necessary installations of TinyOS, as well as of tools, libraries and drivers for TinyOS support, the software code for transforming sensor motes into embedded Web servers must be installed. There are two different TinyOS programs: one that should be installed on the sensor mote acting as the base station (forwarding requests between the application framework and the remote sensor devices), and another that must be installed on the sensor motes which would be deployed in the smart home environment to sense its environmental context.

The software for an IPv6-enabled base station in TinyOS 2.x, can be found under the directory:

```
{TinyOS Root Directory}/apps/IPBaseStation
```

To install the code on the sensor mote acting as the base station, a user must open a terminal and type the command:

```
make platform blip install.1 bsl,/dev/ttyUSBx
```

where x is the USB port where the mote is plugged in, *platform* is the sensor device type being used (e.g. telosb) and *blip* denotes that Blip[2] libraries will be used. The number after the *install* keyword, in this case *1*, identifies the address of the base station. The assigned USB port for each sensor mote can be found by typing the commad *motelist*. This command lists all the motes connected to the USB ports of the system.

If the following information appear on the terminal screen, this means that the TinyOS code has been successfully installed on the sensor mote.

[1]http://www.tinyos.net/

[2]http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip

```
    installing telosb binary using bsl
tos-bsl --telosb -c /dev/ttyUSB0 -r -e -I -p build/telosb/main.ihex.out-1
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
20886 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-1 build/telosb/main.ihex.out-1
```

Afterwards, the base station needs to start its operation. This can be achieved through the following two successive commands:

```
cd {TinyOS Root Directory}/support/sdk/c/blip/
sudo driver/ip-driver /dev/ttyUSB0 telosb
```

Information similar to the following should appear on your terminal as soon as you run this command.

```
2012-08-16:22:19: INFO: Read config from 'serial_tun.conf'
2012-08-16:22:19: INFO: Using channel 15
2012-08-16:22:19: INFO: Retries: 1
2012-08-16:22:19: INFO: telnet console server running on port 6106
2012-08-16:22:19: INFO: created tun device: tun0
2012-08-16:22:19: INFO: interface device successfully initialized
2012-08-16:22:19: INFO: starting radvd on device tun0
```

Finally, all sensor motes, which will be deployed remotely in the smart home environment, need to install the TinyOS code we developed, for transforming them into embedded IPv6-enabled Web servers. (you can email the author at kami@cs.ucy.ac.cy to obtain the TinyOS code). In order to install the code on a sensor mote, the following command must be executed:

```
make platform blip install.y bsl,/dev/ttyUSBx
```

Remember to use a different *y* number for each sensor mote, avoiding the value *1*, which is reserved for the base station node. As before, *x* is the USB port onto which the mote is temporarily plugged in (for the installation procedure)

This is all! Now you can equip the sensor motes with batteries and deploy them inside the smart home. They will start immediately to scan the environment for an application framework (see the device discovery procedure in Section 5.2.1) and, as soon as they are bound to one, they will begin sensing the environmental conditions that exist in the house area, namely temperature, humidity and illumination.

Appendix G

Example WADL Service Description Files

Two example WADL service description files are presented, one for Telos sensor motes and one for Ploggs smart power outlets. The former case is shown in Listing G.1 and the latter in Listing G.2.

Listing G.1: An example WADL file for a Telosb sensor mote.

```
<?xml version="1.0"?>
<application xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:html="http://www.w3.org/1999/xhtml"
             xmlns="http://wadl.dev.java.net/2009/02">
 <grammars></grammars>
 <resources base="http://localhost:8080/tinyOS/">
    <resource path="Temperature">
        <doc xml:lang="en" title="tinyOS Temperature Service">
           The <html:i>tinyOS Client Code</html:i> provides Services
              for motes' Temperature sensing capabilities.
        </doc>
        <method name="GET" id="Temperature">
          <doc xml:lang="en" title="Measure Temperature in Celsius"/>
          <request>
          </request>
          <response status="200">
             <representation mediaType="text/plain" type="xsd:string">
          <doc xml:lang="en" title="The Temperature sensed current
             value"/>
               </representation>
          </response>
    </method>
    </resource>
    <resource path="Humidity">
       <doc xml:lang="en" title="tinyOS Humidity Service">
          The <html:i>tinyOS Client Code</html:i> provides Services
             for motes' Humidity sensing capabilities.
       </doc>
       <method name="GET" id="Humidity">
           <doc xml:lang="en" title="Measure Humidity in % value"/>
           <request>
```

```xml
           </request>
           <response status="200">
              <representation mediaType="text/plain" type="xsd:string">
              <doc xml:lang="en" title="The Humidity sensed current value"/>
              </representation>
           </response>
        </method>
    </resource>
    <resource path="Illumination">
       <doc xml:lang="en" title="tinyOS Illumination Service">
        The <html:i>tinyOS Client Code</html:i> provides services for motes' Radiation sensing capabilities.
       </doc>
       <method name="GET" id="Illumination">
          <doc xml:lang="en" title="Measure Illumination in Lux value"/>
          <request>
          </request>
          <response status="200">
             <representation mediaType="text/plain" type="xsd:string">
             <doc xml:lang="en" title="The Illumination sensed current value"/>
             </representation>
          </response>
       </method>
    </resource>
    <resource path="Light">
       <doc xml:lang="en" title="tinyOS Light Service">
        The <html:i>tinyOS Client Code</html:i> provides Services setting motes' Leds.
       </doc>
       <method name="POST" id="Light">
         <doc xml:lang="en" title="Set the RED/GREEN/BLUE Leds that are on the sensor mote ON/OFF"/>
         <request>
              <param name="color" type="xsd:character" required="true" default="" style="query">
              <doc xml:lang="en" title="Leds Color to switch on"/>
            <option value="R"/><option value="G"/><option value="B"/>
              </param>
        </request>
        <response status="200">
             <representation mediaType="text/plain" type="xsd:string">
             <doc xml:lang="en" title="Acknowlegment Value"/>
             </representation>
        </response>
       </method>
    </resource>
 </resources>
</application>
```

Listing G.2: An example WADL file for a Plogg smart power outlet.

```xml
<?xml version="1.0"?>
<application xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:html="http://www.w3.org/1999/xhtml"
             xmlns="http://wadl.dev.java.net/2009/02">
 <grammars></grammars>
 <resources base="http://localhost:8080/plogg/">
    <resource path="Electricity">
      <doc xml:lang="en" title="Plogg Smart Meter Electricity Service">
        The <html:i>Electricity service</html:i> provides energy consumption measurements in Watts and kWh.
      </doc>
      <method name="GET" id="Electricity">
      <doc xml:lang="en" title="Measure electrical consumption in Watts and kWh"/>
      <request>
      </request>
      <response status="200">
       <representation mediaType="application/json" type="xsd:string">
          <doc xml:lang="en" title="The JSON value of the consumption measurement of the connected electrical appliance"/>
        </representation>
      </response>
    </method>
    </resource>
    <resource path="Switch">
      <doc xml:lang="en" title="Plogg Smart Meter Switch Service">
        The <html:i>Switch service</html:i>turns an electrical appliance on/off.
      </doc>
      <method name="POST" id="Switch">
         <doc xml:lang="en" title="Set the appliance ON/OFF"/>
         <request>
            <param name="mode" type="xsd:string" required="true" default="" style="query">
                <doc xml:lang="en" title="Switch on/off the appliance"/>
          <option value="On"/><option value="Off"/>
           </param>
         </request>
         <response status="200">
              <representation mediaType="text/plain" type="xsd:string">
              <doc xml:lang="en" title="Acknowlegment Value"/>
              </representation>
         </response>
      </method>
    </resource>
 </resources>
</application>
```