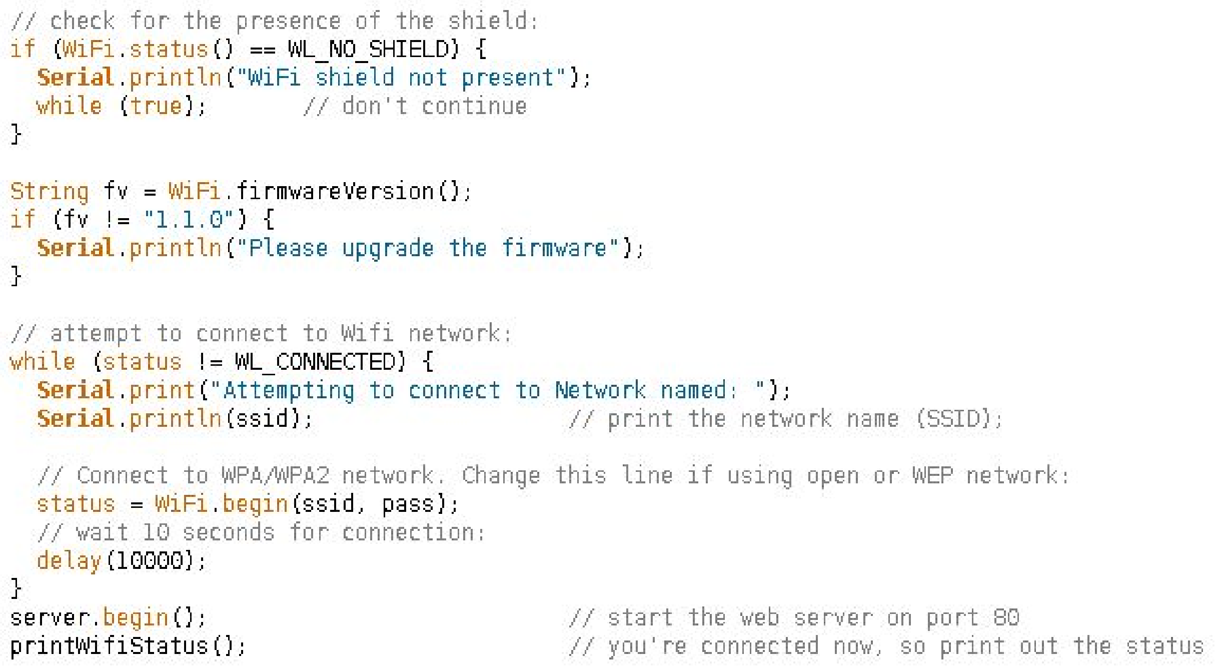
Smart Lamp - Networking Aspect

Network Code - Server Side

Following steps are followed in order to set up the server,

- Checking the connectivity of the WiFi-Shield.
- Checking the firmware version.
- Attempting the connection.
- Start the server.

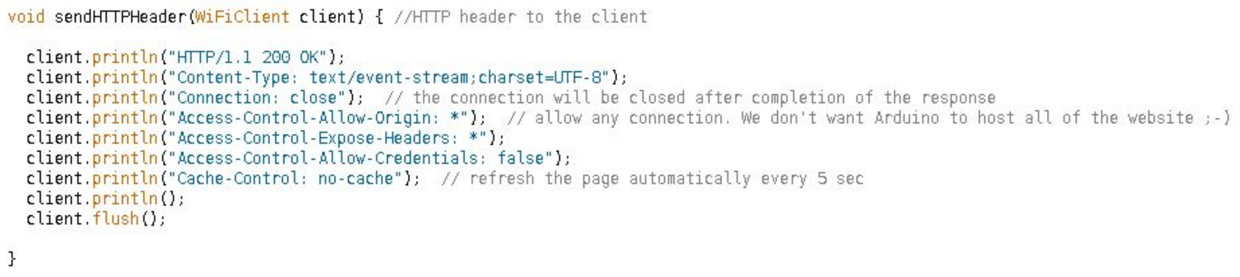


```
// check for the presence of the shield:
if (WiFi.status() == WL_NO_SHIELD) {
  Serial.println("WiFi shield not present");
  while (true);       // don't continue
}

String fv = WiFi.firmwareVersion();
if (fv != "1.1.0") {
  Serial.println("Please upgrade the firmware");
}

// attempt to connect to Wifi network:
while (status != WL_CONNECTED) {
  Serial.print("Attempting to connect to Network named: ");
  Serial.println(ssid);                   // print the network name (SSID);

  // Connect to WPA/WPA2 network. Change this line if using open or WEP network:
  status = WiFi.begin(ssid, pass);
  // wait 10 seconds for connection:
  delay(10000);
}
server.begin();                           // start the web server on port 80
printWifiStatus();                        // you're connected now, so print out the status
```

Each block of the above code do the mentioned tasks respectively. If the SSID and password are correct, an HTTP server is started at the port 80. Then the following code line is used to make a client object.

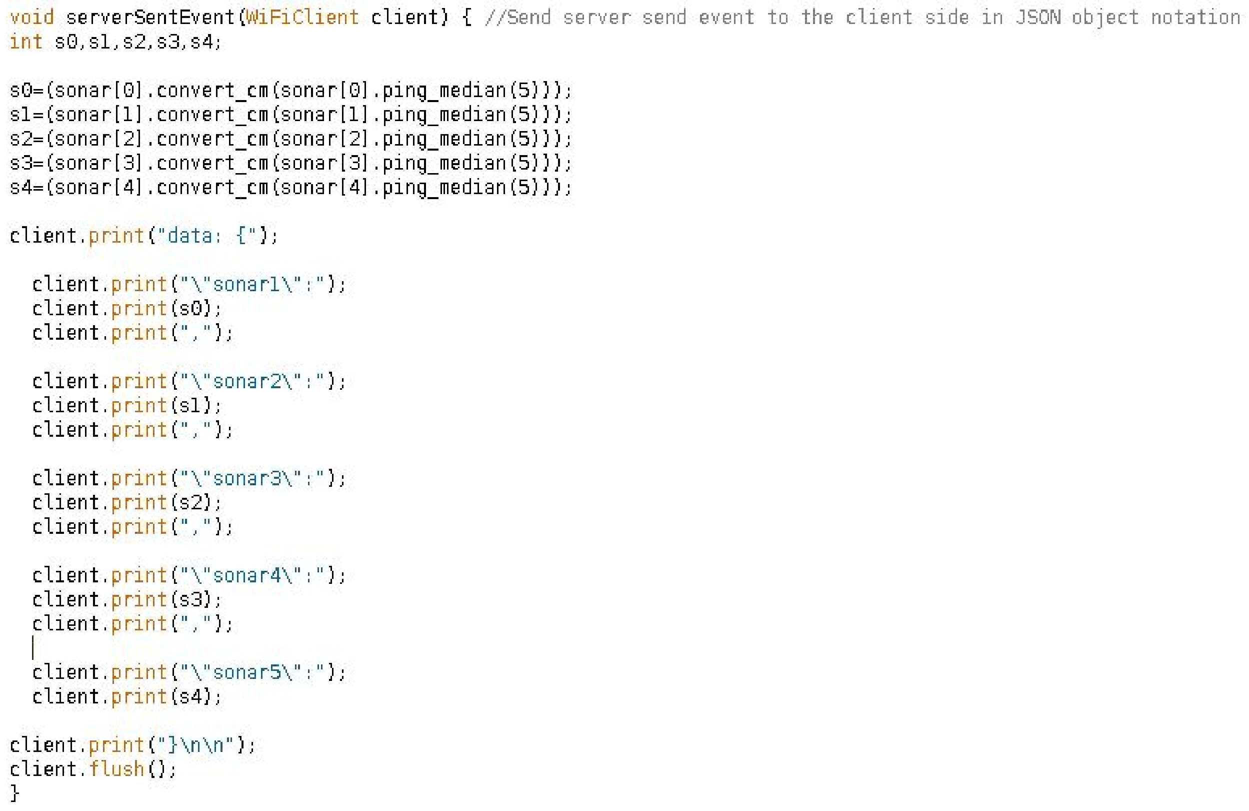
WiFiclient client = server.available();

If a client is connected, following HTTP headers are sent to the client.



```
void sendHTTPHeader(WiFiClient client) { //HTTP header to the client

  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/event-stream;charset=UTF-8");
  client.println("Connection: close");  // the connection will be closed after completion of the response
  client.println("Access-Control-Allow-Origin: *");  // allow any connection. We don't want Arduino to host all of the website ;-)
  client.println("Access-Control-Expose-Headers: *");
  client.println("Access-Control-Allow-Credentials: false");
  client.println("Cache-Control: no-cache");  // refresh the page automatically every 5 sec
  client.println();
  client.flush();

}
```

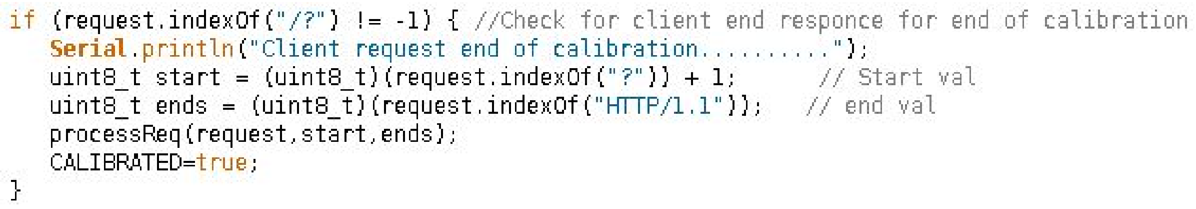
These headers cause the server to generate a server sent event and the client-side code is designed to listen to this event stream and act accordingly. This server sent event is used to send the sensor values to the client in the JSON format which helps the client in calibration.



```
void serverSentEvent(WiFiClient client) { //Send server send event to the client side in JSON object notation
int s0,s1,s2,s3,s4;

s0=(sonar[0].convert_cm(sonar[0].ping_median(5)));
s1=(sonar[1].convert_cm(sonar[1].ping_median(5)));
s2=(sonar[2].convert_cm(sonar[2].ping_median(5)));
s3=(sonar[3].convert_cm(sonar[3].ping_median(5)));
s4=(sonar[4].convert_cm(sonar[4].ping_median(5)));

client.print("data: {");

  client.print("\"sonar1\":");
  client.print(s0);
  client.print(",");

  client.print("\"sonar2\":");
  client.print(s1);
  client.print(",");

  client.print("\"sonar3\":");
  client.print(s2);
  client.print(",");

  client.print("\"sonar4\":");
  client.print(s3);
  client.print(",");

  client.print("\"sonar5\":");
  client.print(s4);

client.print("}\n\n");
client.flush();
}
```

Server continuously listen (even while sending the SSE) for an ‘end of calibration GET request’ sent by the client. It captures the request via following line,

String request = client.readStringUntil(‘\r’);

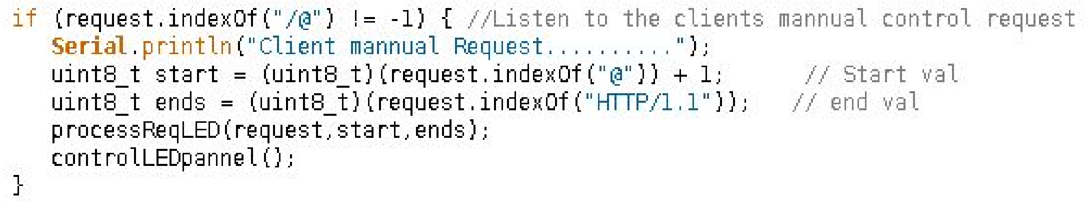


```
if (request.indexOf("/?") != -1) { //Check for client end responce for end of calibration
   Serial.println("Client request end of calibration..........");
   uint8_t start = (uint8_t)(request.indexOf("?")) + 1;       // Start val
   uint8_t ends = (uint8_t)(request.indexOf("HTTP/1.1"));    // end val
   processReq(request,start,ends);
   CALIBRATED=true;
}
```

When the calibration is finished, lamp goes into the real-time positioning mode. But the server is continuously running in the background. A client can again connect with the server in order to manually control the lamp. The data which is needed to manually control the lamp is embedded into the GET request as follows.

GET/@1:0:0:1:0 HTTP/1.1

This request is differentiated with others from the '@' sign. '@' sign is followed by the data separated with colons.



```
if (request.indexOf("/@") != -1) { //Listen to the clients mannual control request
   Serial.println("Client mannual Request..........");
   uint8_t start = (uint8_t)(request.indexOf("@")) + 1;       // Start val
   uint8_t ends = (uint8_t)(request.indexOf("HTTP/1.1"));    // end val
   processReqLED(request,start,ends);
   controlLEDpannel();
}
```

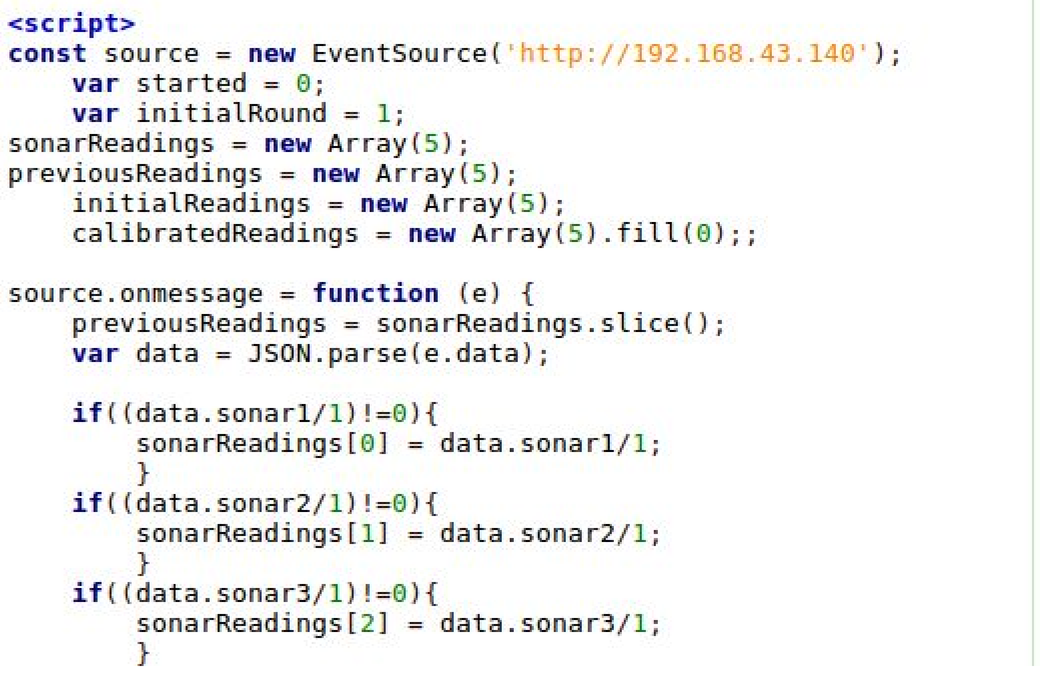
<u>Network Code - Client Side</u>

Client has two interfaces.

- Calibration interface.
- Manual interface.

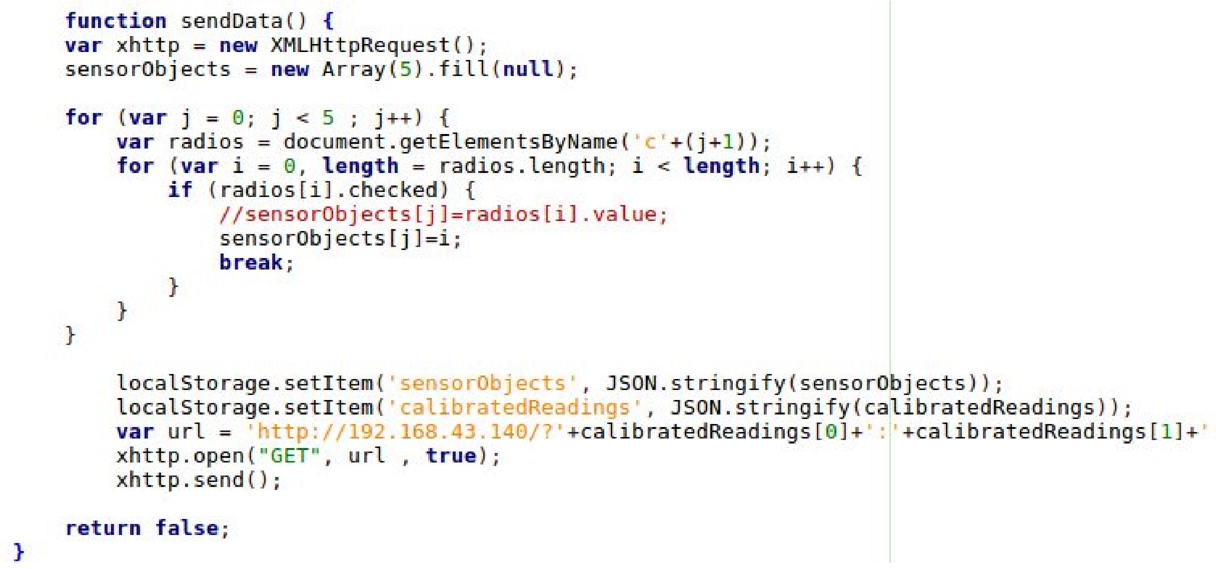
In the calibration mode, initial headers that are sent by the server is captured in order to initiate an event source. Then upon an event, data is captured and parsed into an appropriate arrays to generate a real-time graph. An example JSON object that is received by the client as follows,

"data: {"sonar1" : 10, "sonar2" : 20, "sonar3" : 30, "sonar4" : 40, "sonar5" : 50 }"



```
<script>
const source = new EventSource('http://192.168.43.140');
    var started = 0;
    var initialRound = 1;
sonarReadings = new Array(5);
previousReadings = new Array(5);
    initialReadings = new Array(5);
    calibratedReadings = new Array(5).fill(0);;

source.onmessage = function (e) {
    previousReadings = sonarReadings.slice();
    var data = JSON.parse(e.data);

    if((data.sonar1/1)!=0){
        sonarReadings[0] = data.sonar1/1;
        }
    if((data.sonar2/1)!=0){
        sonarReadings[1] = data.sonar2/1;
        }
    if((data.sonar3/1)!=0){
        sonarReadings[2] = data.sonar3/1;
        }
```

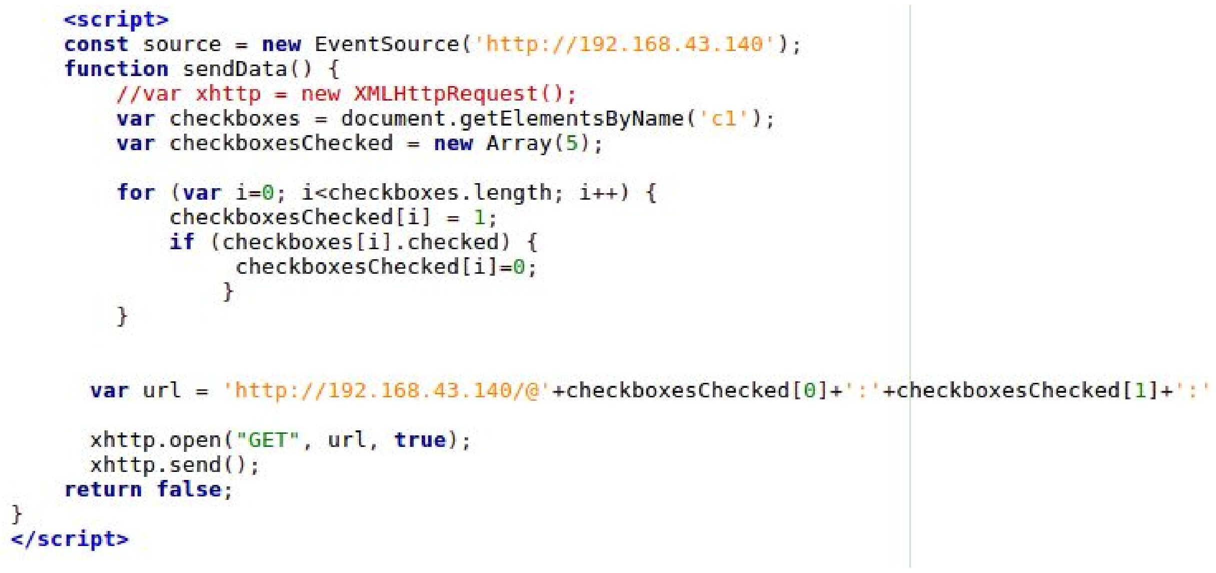
At the end of calibration, for each sensor, maximal movable distance and the ‘physical object’ code is sent back to the server through a GET request.



```
    function sendData() {
    var xhttp = new XMLHttpRequest();
    sensorObjects = new Array(5).fill(null);

    for (var j = 0; j < 5 ; j++) {
        var radios = document.getElementsByName('c'+(j+1));
        for (var i = 0, length = radios.length; i < length; i++) {
            if (radios[i].checked) {
                //sensorObjects[j]=radios[i].value;
                sensorObjects[j]=i;
                break;
            }
        }
    }

        localStorage.setItem('sensorObjects', JSON.stringify(sensorObjects));
        localStorage.setItem('calibratedReadings', JSON.stringify(calibratedReadings));
        var url = 'http://192.168.43.140/?'+calibratedReadings[0]+':'+calibratedReadings[1]+'
        xhttp.open("GET", url , true);
        xhttp.send();

    return false;
}
```

As shown in the above code, XMLHttpRequest is used in order to send the calibration data back to the server.

In the manual control mode, same procedure is followed.



```
    <script>
    const source = new EventSource('http://192.168.43.140');
    function sendData() {
        //var xhttp = new XMLHttpRequest();
        var checkboxes = document.getElementsByName('c1');
        var checkboxesChecked = new Array(5);

        for (var i=0; i<checkboxes.length; i++) {
            checkboxesChecked[i] = 1;
            if (checkboxes[i].checked) {
                 checkboxesChecked[i]=0;
                }
        }


      var url = 'http://192.168.43.140/@'+checkboxesChecked[0]+':'+checkboxesChecked[1]+':'

      xhttp.open("GET", url, true);
      xhttp.send();
    return false;
}
</script>
```

Reasons for implement choices

- A SSE is used to send data to client because a continuous stream should be sent to the client for the calibration to be done correctly. A persistent connection like this is better for this application rather than sending HTTP requests for each sensor data.

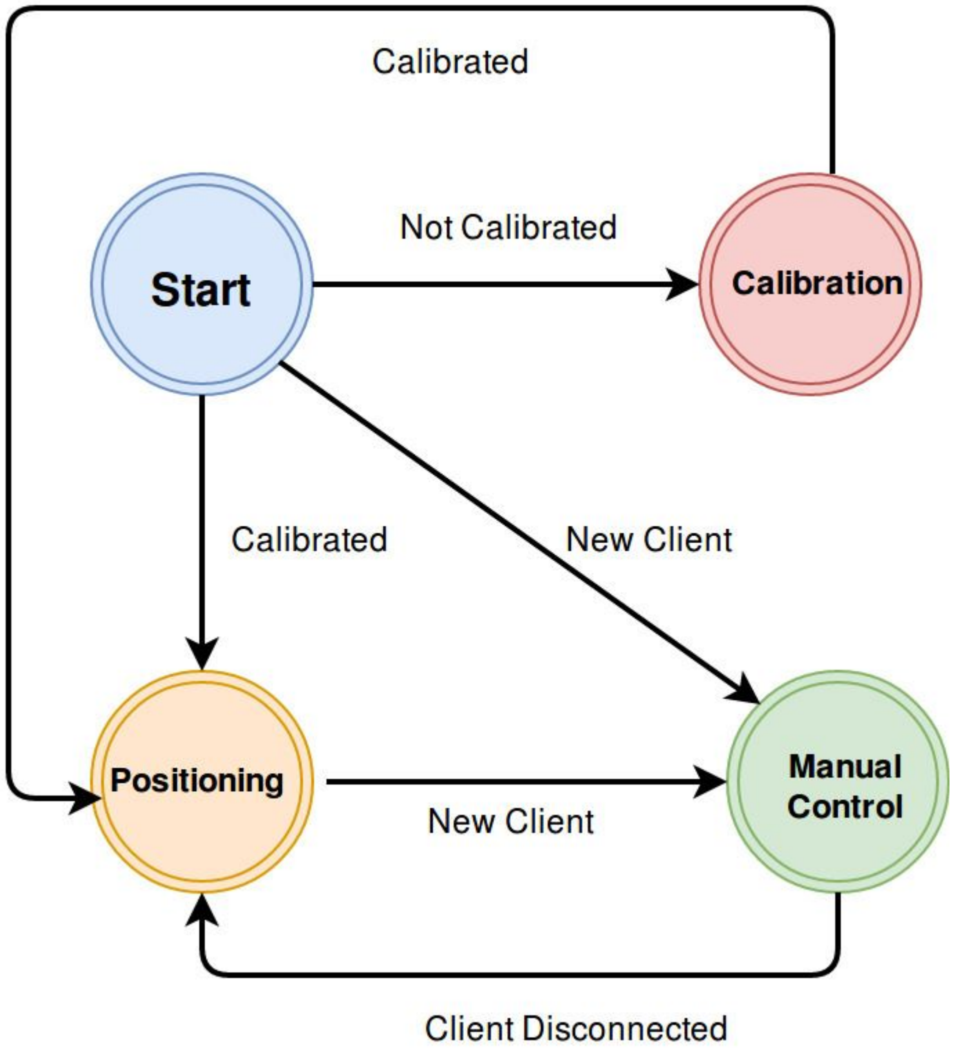
- JSON objects were used to send the data stream because JSON is browser friendly and easier to parse and process. A Javascript library is used in the calibration interface in order to dynamically generate a graph using the sensor values. This Javascript code uses JSON to interpret the data.

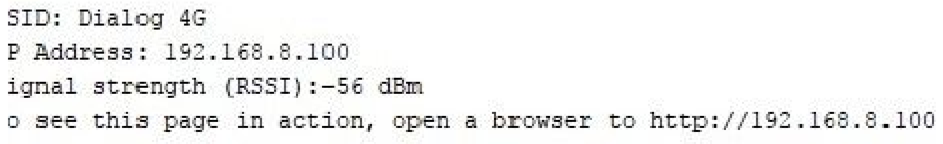
- XMLHttpRequest is used to generate GET request without having to do a full page refresh every time client want to manually control the lamp.

Application layer protocol structure

- Wait 2 minutes for the connection establishment. (Figure 01)
- Client connect to the lamp via calibration interface. (Figure 02)
- Lamp send the client an initial reading.
- Lamp continuously send sensor readings to client.

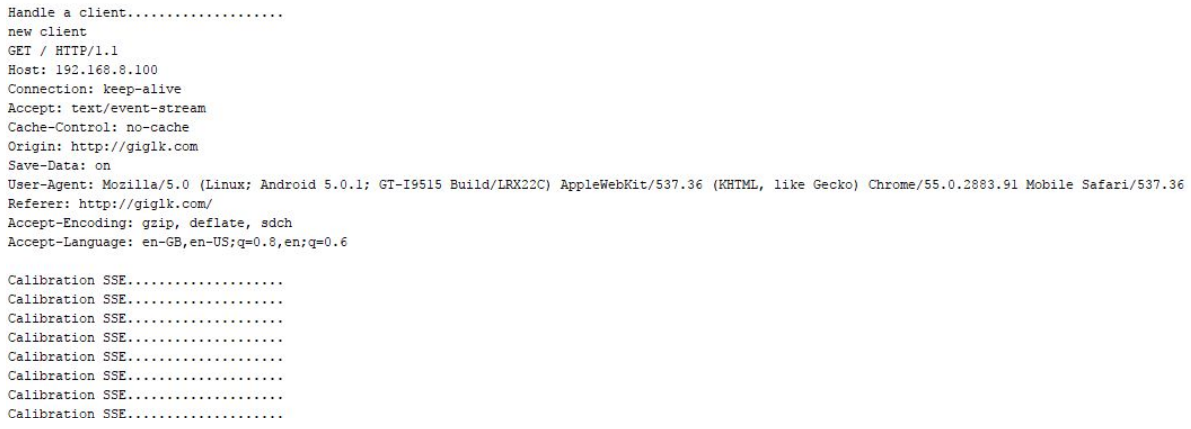
- Client-side plot the data in real-time.
- Client should move through the outlines of the objects in the room.
- Send a confirmation (which includes the calibration data) after calibration is done. (Figure 03)
- Sever captures the calibration data and save into the EEPROM.
- Lamp jump into the real-time positioning state. (Figure 03)
- Keep server available and keep looking for incoming connections.
- If a client connects for manual control, lamp jump into the manual control state.
- Client sends manual control requests. (Figure 04)
- Server captures the requests, process and control the LED panels.
- If client disconnected, jump to the real-time positioning state again.





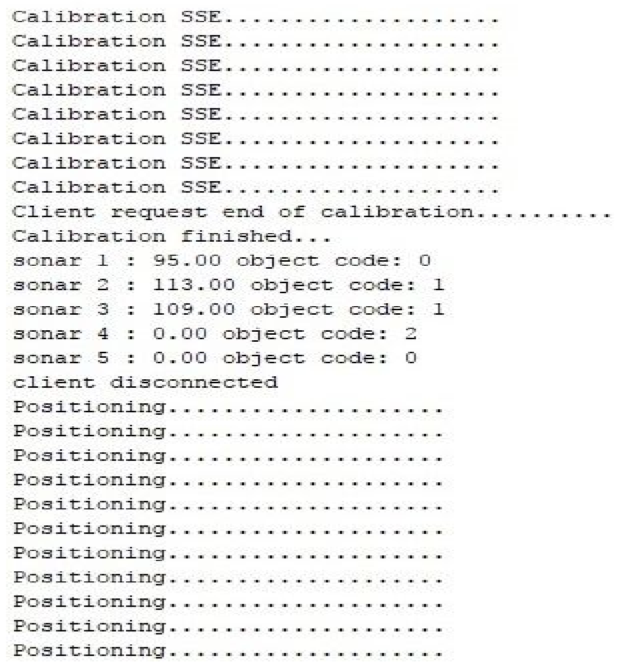
```
SID: Dialog 4G
P Address: 192.168.8.100
ignal strength (RSSI):-56 dBm
o see this page in action, open a browser to http://192.168.8.100
```

Figure 01 : Server waiting for client



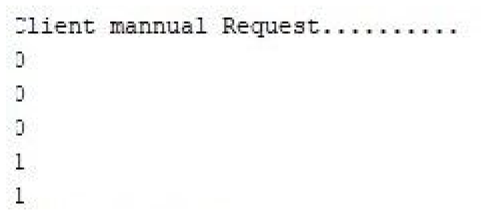
```
Handle a client...................
new client
GET / HTTP/1.1
Host: 192.168.8.100
Connection: keep-alive
Accept: text/event-stream
Cache-Control: no-cache
Origin: http://giglk.com
Save-Data: on
User-Agent: Mozilla/5.0 (Linux; Android 5.0.1; GT-I9515 Build/LRX22C) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.91 Mobile Safari/537.36
Referer: http://giglk.com/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6

Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
```

Figure 02 : Client calibrating via calibration interface



```
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Calibration SSE...................
Client request end of calibration..........
Calibration finished...
sonar 1 : 95.00 object code: 0
sonar 2 : 113.00 object code: 1
sonar 3 : 109.00 object code: 1
sonar 4 : 0.00 object code: 2
sonar 5 : 0.00 object code: 0
client disconnected
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
Positioning...................
```

Figure 03: Calibration data received and switching to real-time positioning mode



```
Client mannual Request..........
0
0
0
1
1
```

Figure 04 : Client manual control request

Further Improvements

- User can be provided with an interface to put in the SSID and password of the network they are connected.
- Server can be configured to have a static private IP or user can be displayed with the dynamic IP of the server each time they change.
- A socket level implementation could have been more efficient.
- A mobile application could be used for calibration and manual control.
- A central server can be used to control a larger network of lights concurrently.
- Since micro-controllers typically have low performance, intelligence of the lamp could be transferred into a separate server in order boost the performance.

Group 09

E/13/073 - Dissanayake T.D.P.T.
E/13/107 - Gamage C.T.N.
E/13/274 - Rajapaksha Y.N.