Forschungsprojekt Sommersemester 2014: Autonomes Energiemanagement für Libelium Smart Cities Sensorknoten

Implementierung und Bereitstellung für DA-Sense

TECHNISCHE UNIVERSITÄT DARMSTADT

1 Ziele

Die Ziele des Forschungsprojekts im Sommersemester 2014 sind die Entwicklung und Implementierung eines Energiemanagements und die Bereitstellung der Kommunikation zu DA-Sense auf den Libelium Waspmote Smart Cities Sensorknoten. Weiterführend sollen die Sensorknoten im Bereich Darmstadt und Umgebung zur Überwachung von Umweltcharakteristika eingesetzt werden und die gemessenen Daten evaluiert werden.

1.1 Energiemanagement

Da der Sensorknoten Energie über ein Solarpanel bezieht und kann keine konstante Versorgung des Sensorknoten mit ausreichend Strom garantiert werden. Kritische Phasen für den Sensorknoten stellen damit die Nacht, als auch stark bewölkte Tage dar. Um in diese Phasen ebenfalls aktiv sein zu können wird ein Energiemanagement benötigt. Es ist zu beachten, dass der Sensorknoten von sich aus ab einem bestimmten Grenzwert des Batterielevels das 3G-Modul abschaltet und somit grundlegende Funktionalität nicht mehr verfügbar ist. Dieser Umstand macht die Entwicklung und Implementierung eines Energiemanagements umso essentieller für einen letztendlichen Einsatz der Sensorknoten. Es soll ein Energiemanagement entwickelt werden, welches den Sensorknoten autonom operieren lässt. Dies bedeutet konkret die Gewährleistung der Funktionalität durch ein angemessenes Batterielevel über einen unbegrenzten Zeitraum. Hierzu soll der Sensorknoten dazu im Stande sein, seinen Energiehaushalt zu überwachen und den Energieverbrauch nach Bedarf zu adaptieren. Das Energiemanagement sollte so gestaltet sein, dass ein Sensorknoten in bestimmten Zeitabständen seinen Energieverbrauch überprüft und abhängig von diesem die Zeit der Messzyklen anpasst, bis ein stabiles Batterielevel erreicht ist.

1.2 Überwachung von Klimacharakteristika

In Darmstadt und Umgebung werden bereits im Rahmen des DA-Sense Projekts Klimadaten erhoben und auf einer Karte in Form von Heatmaps visualisiert. Die Libelium Sensorknoten sollen die DA-Sense Plattform weiter mit Daten unterstützen und erfordern daher das Auslesen verschiedener Sensoren. Die Überwachung von Klimacharakteristika beschreibt die periodische Messung von Temperatur, Luftfeuchtigkeit, Luftverschmutzung und Lautstärke. Die Temperatur ist zu messen in °C, die Luftfeuchtigkeit in %, die Luftverschmutzung in ppm (parts-per- million) und die Lautstärke in dB. Neben den Umweltcharakteristika sind aber auch GPS-Informationen relevant und nach Möglichkeit der RSSI (Received Signal Strength Indicator), gemessen in dBm.

1.3 Kommunikation zu DA-Sense

Die Kommunikation zu DA-Sense erfolgt über 3G. Hierbei soll die DA-Sense API zum Upload der gemessenen Klimacharakteristika über POST oder GET-Anfragen angesprochen werden. Die Daten werden intern im Sensorknoten gespeichert und in bestimmten Zeitabständen an den DA-Sense Server übermittelt.

1.4 Einsatz und Evaluation

Nach abgeschlossener Implementierung und Test dieser sollen die Sensorknoten einem Feldtest unterzogen werden. Dieser Feldtest beinhaltet die Platzierung und Aktivierung der Sensorknoten unter Bedingungen, die ihrem späteren Einsatzgebiet möglichst ähnlich sind. Die im Feldtest gesammelten Daten werden für die Evaluation des Energiemanagements und der Zuverlässigkeit der Sensorknotenfunktionalität genutzt. Am Ende der Evaluation sollte eine Aussage darüber getroffen werden können, wie effizient die Datenerhebung und das Energiemanagement der Sensorknoten ist und welche Optimierungen weiter vorgenommen werden können.

2 Implementierung

In diesem Abschnitt wird die Entwicklung und Implementierung der 1 angestrebten Ziele vertieft. Zunächst werden in den nachfolgenden Subkapiteln die Ideen erläutert und dann anhand von Code verdeutlicht. Die für Libelium Waspmote genutzte Programmiersprache ist eigenständig, jedoch ähnlich zu C und C++. Grundsätzlich ist bei der Implementierung zu beachten, dass die setup()-Methode, als auch die loop()-Methode, unerlässlich sind. Die setup()-Methode wird genutzt, um benötigte Module zu starten und initiale Aufgaben auszuführen, wie das Erstellen einer Datei auf der SD-Karte. Die loop()-Methode beinhaltet den Code, welcher immer wieder ausgeführt werden soll. Hier ist dies das Energiemanagement, die Kommunikation und die Messungen. Es existiert nur eine Klasse, Forproj14.pde, in welcher die Implementierung erfolgte.

2.1 Energiemanagement

Die Entwicklung des Energiemanagements beinhaltet die Umsetzung der adaptiven Messzyklen, die Überwachung des Batterielevels und die Kontrolle des Energiemodus. Adaptive Messzyklen bedeuten die Adaption des Zeitraums zwischen

zwei Messungen der Umweltcharakteristika (*Samplingrate*). Dazu ist es nötig zu überprüfen, ob in einem bestimmten Zeitraum zu viel oder zu wenig Energie verbraucht wurde, was anhand des Batterielevels möglich ist. Die annehmbaren Energiemodi des Sensorknoten sind der aktive Modus, in dem alle Sensorknotenmodule aktiv sind, der *DeepSleep*-Modus, ein Energiesparmodus, in dem die Sensorknotenmodule in einen Standby-Modus gehen, und der *Hibernate*-Modus, in dem alle Sensorknotenmodule, mit Ausnahme des RTC-Moduls, vom Mainboard abgeschaltet werden.
Um zu überprüfen, ob der Messzyklus adaptiert werden muss, wird bei jedem Erwachen des Knotens für eine potentielle Messung das Batterielevel ausgelesen und in ein vorgesehenes Integerarray gespeichert, da das Batterielevel als % ausgelesen wird und nur geradzahlige Werte annehmen kann. Es werden maximal zwölf vergangene Werte des Batterielevels in dem Integerarray gespeichert, bevor die Messzyklen adaptiert werden. Die Adaption der Messzyklen findet spätestens nach 13 Stunden statt. Über die aufgefassten letzten Batterielevel wird der Durchschnitt gebildet und mit dem aktuellen Batterielevel verglichen. Listing 1 zeigt die Implementierung zur Kontrolle des Energiemodus und

Listing 1: Überwachung des Batterielevels und Kontrolle des Energiemodus

```
1  void loop() {
2          ...
3          batteryLevel = PWR.getBatteryLevel();
4          batteryLevels[batteryCount] = batteryLevel;
5          batteryCount++;
6          if(batteryCount > 11) {
7              batteryCount = 0;
8          }
9          ...
10         //check if an update of the samplingRate is necessary
11         if(timeSinceLastBatterySample >= HalfADayInMilliSeconds) {
12                 //check if there is an increase or decrease in the level of the battery
13                 int avgBattery = 0;
14                 int c = 0;
15                 for(int x = 0; x < 12; x++) {
16                     avgBattery += batteryLevels[x];
17                     if(batteryLevels[x] != 0) {
18                         c++;
19                     }
20                 }
21                 avgBattery = avgBattery / c;
22                 batteryCount = 0;
23                 int diff = batteryLevel - avgBattery;
24                 if(diff > 0) {
25                     enoughEnergy = 0;
26                 } else  if(diff < 0) {
27                     enoughEnergy = 1;
28                 } else {
29                     enoughEnergy = 2;
30                 }
31                 samplingRate = adaptSamplingRate();
32                 deepsleepTime = samplingRateToWaitingTime(samplingRate);
33
34                 timeSinceLastBatterySample = 0;
35                 }
36             timeSinceLastBatterySample += samplingRate;
37             ...
38             PWR.deepSleep(deepsleepTime, RTC_OFFSET, RTC_ALM1_MODE1, ALL_OFF);
39             if(intFlag & RTC_INT) {
40                 intFlag &= ~(RTC_INT);
41                 USB.println(F("Wake up"));
42                 }
43 ...
44 }
```

Die Methode adaptSamplingRate() verkürzt oder verlängert die Zeiträume zwischen den Messungen, zu sehen in Listing 2. Abhängig von der Differenz zwischen dem Durchschnitt des aktuellen Batterielevels und der Durchschnitt der letzten zwölf gemessenen Batterielevel wird die Samplingrate erhöht oder gesenkt. Ist die Differenz negativ, so werden die Zeiträume zwischen den Messzyklen um den Faktor 1.1 verlängert, ist die Differenz positiv, so werden die Zeiträume zwischen den Messzyklen um den Faktor 0.5 verkürzt. Bei gleichbleibendem Batterielevel wird die Samplingrate nicht verändert.

Listing 2: Adaption der Messzyklen

```
1  long adaptSamplingRate() {
```

```
2         long nSamplingRate;
3         if(enoughEnergy == 0){
4         nSamplingRate = (long) (samplingRate * increasingFactor);
5         } else if(enoughEnergy == 1) {
6         nSamplingRate = (long) (samplingRate * decreasingFactor);
7         } else {
8                 nSamplingRate = samplingRate;
9         }
10        return nSamplingRate;
11  }
```

Zur weiteren Verfolgung des Energiemanagements wird bei jedem Erwachen des Sensorknoten der Zeitstempel des RTC-Moduls, das Batterielevel und die Samplingrate in eine Datei namens "BOT.TXTäuf die SD-Karte geschrieben.

2.2 Kommunikation zu DA-Sense

In Zusammenarbeit mit Ulf Gebhardt wurde die Kommunikation zum DA-Sense Server eingerichtet und die korrekte Struktur der Nachrichten aufgebaut. Die Kommunikation zu dem DA-Sense Server erfolgt über das 3G-Modul des Sensorknotens. Hierfür wird eine SIM-Karte benötigt, welche über eine PIN-Eingabe in der Implementierung freigeschaltet werden muss, um das 3G-Modul für die Kommunikation nutzen zu können. Als Protokoll zwischen DA-Sense Server und dem Sensorknoten wird HTTP verwendet, da der DA-Sense Server eine API dafür bereitstellt. Die API wird mittels POST oder GET angesprochen, um sich gegenüber dem Server als registrierter Nutzer zu authentifizieren und um gemessene Daten zu übertragen. Grundsätzlich erfolgt die Kommunikation mit dem DA-Sense Server in zwei Schritten:

1. Authentifizierung durch Anmeldung mit einem registrierten Benutzernamen und Passwort
2. Upload von Sensordaten als Messreihe

Listing 3 zeigt die Methode *sendSamples()*, welche dafür zuständig ist gemessene Werte in einer Messreihe der Größe 3 an den Server zu schicken. Für jede gemessene Umweltcharakteristik wird eine eigene Nachricht an den Server geschickt und es ist notwendig sich dem Server gegenüber neu zu authentifizieren, da eine Session-ID nicht für mehrere Uploads gültig ist. Die Zeilen 6-9 in Listing 3 sind für die Authentifizierung zuständig, wobei zu beachten ist, dass das angegebene Passwort ein Mal mit SHA1 gehasht und ein anderes Mal mit MD5 gehasht wurde. Die Berechnung mit diesen Hashfunktionen ist auf den Sensorknoten möglich, jedoch wurde dies aufgrund von Speicher und Performanz ausgelagert und daraufhin in Variablen hardcoded implementiert. Nach der Authentifizierung auf dem Server muss der richtige *measurementType* für die Daten angegebene werden, damit der Server erkennt, welche Umweltcharakteristika in der Nachricht enthalten ist. Der *measurementType* ist dabei für eine gesamte Messreihe anzugeben, siehe Zeile 45 Listing 3.

Listing 3: Authentifizierung und Upload von Messdaten auf den DA-Sense Server

```
1   void sendSamples() {
2       _3G.setPIN("6358");
3       answer = _3G.check(60);
4           if (answer == 1) {
5                 for(int i = 0; i < 3; i++) {
6                   sprintf(args, "call=account&action=login&compatibility=1&username=s&password_sha=%s&
                        password_md5=%s&locale=deDE&deviceinfo={\"deviceType\":2,\"deviceIdent\":\"%lu\",\"
                        deviceManufactor\":\"libelium\",\"deviceModel\":\"plugAndSense\",\"deviceName\":\"
                        SmartCities\",\"sensors\":[{\"measurementType\":1},{\"measurementType\":4},{\"
                        measurementType\":6},{\"measurementType\":8}]}", username, sha1pw, md5pw, Utils.
                        readSerialID());
7                   sprintf(aux_str, "POST %s? HTTP/1.1\r\nHost: %s\r\nContent-Type: application/x-www-form-
                        urlencoded; charset=UTF-8\r\nContent-Length: %d\r\n\r\n%s", url, host, strlen(args),
                         args);
8
9                   answer = _3G.readURL(host, 80, aux_str);
10
11                  phpSessionID = _3G.getSessionID();
12
13                  aux_str[0]='\0';
14                  args[0]='\0';
15                  delay(5000);
16
17                  if(strlen(phpSessionID) > 0) {
18                    int measurementType = 0;
19                    if(i == 0) {
20                      //temperature
```

```
                    measurementType = 4;
                  }
                  if(i == 1) {
                    //humidity
                    measurementType = 6;
                  }
                  if(i == 2) {
                    //noise
                    measurementType = 1;
                  }
                  if(i == 3) {
                    //dust
                    measurementType = 8;
                  }
                  sprintf(args, "call=input&type=data&json={\"deviceIdent\":\"%lu\",\"measurementType\":%
                      d,\"series\":[{\"name\":\"testSensornode\",\"visibility\":1,\"timestamp\":1,\"
                      values\":[{\"timestamp\":1,\"value\":%s,\"longitude\":%s,\"latitude\":%s,\"
                      altitude\":0,\"accuracy\":0,\"provider\":\"GPS\"},{\"timestamp\":1,\"value\":%s,\"
                      longitude\":%s,\"latitude\":%s,\"altitude\":0,\"accuracy\":0,\"provider\":\"GPS
                      \"},{\"timestamp\":1,\"value\":%s,\"longitude\":%s,\"latitude\":%s,\"altitude
                      \":0,\"accuracy\":0,\"provider\":\"GPS\"}]}]}", Utils.readSerialID(),
                      measurementType, values[i], longlat[0], longlat[1], values[i+4], longlat[2],
                      longlat[3], values[i+8], longlat[4], longlat[5]);
                  sprintf(aux_str, "POST %s? HTTP/1.1\r\nHost: %s\r\nContent-Type: application/x-www-form
                      -urlencoded; charset=UTF-8\r\nContent-Length: %d\r\nCookie: PHPSESSID=%s\r\n\r\n%s
                      ", url, host, strlen(args), phpSessionID, args);

                  answer = _3G.readURL(host, 80, aux_str);

                  aux_str[0]='\0';
                  args[0]='\0';

                  delay(1000);
                }
              }
            }
}
```

2.3 Messung der Umweltcharakteristika

Die Messung der Umweltcharakteristika erfolgt mittels der von Libelium bereitgestellten Bibliothek *SensorCities*. Jeder angeschlossene Sensor wird dabei einzeln über die Bibliothek bei dem Board registriert, der Messwert ausgelesen und wieder abgemeldet. Jeder ausgelesene Wert wird in ein Datenarray gespeichert, bis zu einem Maximum von 12 Werten. Sind 12 Werte in dem Datenarray enthalten, so werden diese Werte an den DA-Sense Server verschickt und das Datenarray wird neu beschrieben. In Listing 4 ist dies beispielhaft für den Temperatursensor zu sehen. Sequentiell werden nach danach die anderen angeschlossenen Sensoren ausgelesen.

Listing 4: Registrierung, Auslesen und Abmeldung eines Sensors

```
SensorCities.setSensorMode(SENS_ON, SENS_CITIES_TEMPERATURE);
delay(100);
temperature = SensorCities.readValue(SENS_CITIES_TEMPERATURE);
SensorCities.setSensorMode(SENS_OFF, SENS_CITIES_TEMPERATURE);
Utils.float2String(temperature, temp, 2);
strcpy(values[sampleNum], temp);
```

Da für die Datenvisualisierung auf dem DA-Sense Server die GPS-Informationen eine Grundvoraussetzung sind, werden Messungen nur vorgenommen, wenn für den Sensorknoten GPS-Informationen verfügbar sind. Durch das Energiemanagement erwacht der Sensorknoten vor jeder potentiellen Messung aus dem *DeepSleep*-Modus und die Module müssen erst aus ihrem Standby-Modus hochgefahren werden, 3G-Modul nicht direkt Zugriff auf GPS-Informationen hat. Daher wurde eine Schleife implementiert, die in 30 Sekunden Abständen GPS-Informationen abfragt und nach maximal fünf gescheiterten Versuchen oder einem nicht startbaren GPS abbricht und den Sensorknoten für einen Zyklus in den *DeepSleep*-Modus versetzt (siehe Listing 5).

Listing 5: Abruf von GPS-Informationen

```
boolean getGPS() {
  gpsAnswer = _3G.startGPS();
```

```
  char latitude[20];
  char longitude[20];
  if(gpsAnswer == 1 && GPStries < 5) {
    if(_3G.getGPSinfo() == 1) {
    Utils.float2String(_3G.convert2Degrees(_3G.latitude), latitude, 10);
    Utils.float2String(_3G.convert2Degrees(_3G.longitude), longitude, 10);
    strcpy(longlat[GPSnum], longitude);
    GPSnum++;
    strcpy(longlat[GPSnum], latitude);
    GPSnum++;
    GPStries = 0;
    return true;
    } else {
      GPStries++;
      delay(30000);
      getGPS();
    }
  } else {
    GPStries = 0;
    SD.appendln(filename, "GPS module could not be started");
    return false;
  }
}
```

3 Probleme während des Forschungsprojekts

In diesem Abschnitt wird auf die Schwierigkeiten und nicht erreichten Ziele des Forschungsprojekts eingegangen. Neben der Einarbeitungszeit in die API für die Libelium Sensorknoten und der Einarbeitungszeit in die genutzte Programmiersprache sind einige der Probleme durch die hardwarenahe Programmierung, aber auch aufgrund der bereitgestellten Entwicklungsumgebung aufgetreten.

3.1 Typkonvertierung

Da viel mit Zeichenketten bei der Implementierung gearbeitet werden musste, jedoch der Datentyp String aus der Standard-Bibliothek für C++ nicht unterstützt wird, musste auf char-Arrays gearbeitet werden, welche deutlich weniger Funktionen aufweisen. Dies ist auch relevant für die Anpassung der Bibliothek für das 3G-Modul, worauf in einem anderen Kapitel eingegangen wird. Um den Sensorknoten für die Samplingrate in den *DeepSleep*-Modus zu versetzen, welcher aus energietechnischen Gründen angenommen wird, muss noch die Samplingrate in einen String umgewandelt werden. Die Samplingrate ist als Datentyp Long definiert, um Berechnungen mit ihr durchzuführen, und benötigt daher die Konvertierung in ein char-Array. Da das char-Array das Format *dd:hh:mm:ss* aufweisen muss und keine Bibliothek eine solche Konvertierung bereitstellt, musste hierfür eine eigene Methode geschrieben werden, dargestellt in Listing 6.

Listing 6: Konvertierung von Millisekunden in das Format *dd:hh:mm:ss*

```
char* millisecondsToDeepSleepTime() {
  char waitingTime[18];
  char h[4];
  char m[4];
  char s[4];
  unsigned long hours =  0;
  hours = (samplingRate / 1000);
  hours = hours / 3600;
  hours = hours % 24;
  unsigned long minutes = 0;
  minutes = samplingRate / 1000;
  minutes = minutes / 60;
  minutes = minutes % 60;
  unsigned long seconds = 0;
  seconds = (samplingRate / 1000);
  seconds = seconds % 60;

  if(hours < 10) {
    snprintf(h, 4, "0%d", hours);
  } else {
    snprintf(h, 4, "%d", hours);
  }
  if(minutes < 10) {
```

```
24      snprintf(m, 4, "0%d", minutes);
25    } else {
26      snprintf(m, 4, "%d", minutes);
27    }
28    if(seconds < 10) {
29      snprintf(s, 4, "0%d", seconds);
30    } else {
31      snprintf(s, 4, "%d", seconds);
32    }
33
34    snprintf(waitingTime, 18, "00:%s:%s:%s", h, m, s);
35    return waitingTime;
36  }
```

3.2 Verfügbare RAM-Größe

Der auf einem Sensorknoten verfügbare RAM-Speicherplatz für ein Programm beträgt 8192Bytes. Aufgrund dieser Einschränkung ist es maximal möglich Messreihen mit einer Größe von 3 an den DA-Sense Server zu kommunizieren. Die Allokierung von Speicherplatz für gesendete Nachrichten belegt dabei den Hauptteil des RAM-Speichers und schränkt auch die Menge an auszuführendem Code ein. Das kompilierte Programm benötigt aktuell 81XX Bytes und lässt damit keinen Platz für weitere Implementierungen. Auch die Berechnung des Passworts für die Authentifizierung mit dem DA-Sense Server mit SHA1 und MD5 wurden daher ausgelagert und hardcoded in Variablen eingesetzt. Wie im vorigen Kapitel 3.1 bereits angesprochen wurde, führt auch die fehlende Unterstützung von Datentypen zu mehr benötigtem Code, da zusätzliche Funktionalität implementiert werden muss. Auch das Formatierungszeichen für Float-Werte wird in der für Libelium genutzten Programmiersprache nicht unterstützt, wodurch Float-Werte mittels der Libelium Utils-Bibliothek zuerst in ein char-Array umgewandelt werden müssen, bevor sie in die Nachrichten an den DA-Sense Server geschrieben werden können. Die Allokation des char-Array für die Float-Werte belegt ebenfalls wertvollen Speicher. Ferner beschränkt die RAM-Größe auch die aktuelle Funktionalität des Programms. Derzeitig ist nicht das Auslesen des RSSI oder die Übertragung dieser an den DA-Sense Server implementiert, da es aufgrund mangelnden Speichers nicht möglich ist.

3.3 Debugging

In der von Libelium bereitgestellten Entwicklungsumgebung Waspmote PRO IDE gibt es keine Möglichkeit den ausgeführten Code sequentiell durchzulaufen und Variablenwerte auszulesen. Die einzige Möglichkeit das Programm zu Debuggen ist die Ausgabe über den USB-Port auf die Konsole der Entwicklungsumgebung. Dies erschwert es ungemein zu Überprüfen, ob das Programm der intentionalen Logik gerecht wird und mögliche Fehler zu finden und zu beheben. Codeabschnitte mussten daher in einem alleinstehenden Programm implementiert und getestet werden, um Fehler vorzubeugen oder zu beheben. Aufgrund von unterschiedlichen Abhängigkeiten der Codeabschnitte zu einander ist dies nur teilweise möglich.

3.4 GPS

Das GPS der Sensorknoten funktioniert über das 3G-Modul und muss aktiv gestartet werden. Da der Sensorknoten in Räumlichkeiten kein GPS-Signal empfangen hat und auch an Fenstern keines empfing musste der Sensorknoten im Freien getestet werden. Aufgrund fehlender mobiler Hardware war dies nur sehr begrenzt möglich und auch im Freien war es dem Sensorknoten nur sporadisch möglich ein GPS-Signal zu erhalten und die GPS-Informationen auszulesen, was dazu führte, dass das fertige Programm nicht vollständig getestet werden konnte.

3.5 Anpassung der 3G-Bibliothek

Der Upload von Messdaten benötigt die Session-ID aus der vorangegangenen Authentifizierung bei dem DA-Sense Server. Da das Auslesen der Session-ID nicht von der Libelium Standard-Bibliothek für das 3G-Modul zur Verfügung gestellt wird, musste diese nachträglich implementiert werden. Da die eingehenden Nachrichten auf dem 3G-Modul außerhalb nicht außerhalb der 3G-Bibliothek zugänglich sind musste die Implementierung innerhalb der 3G-Bibliothek erfolgen.

3.6 Zugriff auf SD-Karte

Der Zugriff auf die SD-Karte erfolgt über eine von Libelium bereitgestellte Bibliothek. Es ist möglich, dass weder das Auslesen der auf der SD-Karte befindlichen Dateien, noch das Erstellen einer Datei, noch das Löschen einer funktionieren. Dies ist bereits öfters aufgetreten und wurde bei mehreren Sensorknoten beobachtet. Es erfolgt keine Ausgabe einer Fehlermeldung und eine Ursache konnte bisher nicht gefunden werden. Dies ist ein Punkt, der weiter zu beobachten ist, falls zukünftige Funktionalität den Zugriff auf die SD-Karte beinhaltet.

3.7 Kommunikation und DA-Sense API

Die Kommunikation zwischen Sensorknoten und DA-Sense Server stellte eines der initial größten Probleme dar. Die von Libelium bereitgestellten Beispiele zur Nutzung des 3G Moduls im Hinblick auf HTTP beinhalten nur die Grundlagen, welche Funktionalität hierzu von der 3G-Bibliothek zur Verfügung gestellt wird. Da keine weiteren ausführlicheren Codebeispiele zu diesem Einsatzzweck in Foren vorhanden waren und die 3G-Bibliothek keine automatische Generierung von HTTP-Paketen unterstützt, mussten Kenntnisse über HTTP-Pakete und deren Struktur erarbeitet und implementiert werden. Da GET-Requests nur einen eingeschränkten Payload beinhalten können eignen sich diese nicht zum Upload von Daten, jedoch funktionierten POST-Requests initial, aufgrund fehlender oder falsch definierter Headereinträge in den HTTP-Paketen, nicht mit der DA-Sense API. Da auch von dem DA-Sense Server keine Fehlermeldung über falsche Headereinträge erhalten werden, konnte das Problem erst in Zusammenarbeit mit Ulf Gebhardt gelöst werden. Jedoch sind noch nicht alle Problematiken bei der Kommunikation mit der DA-Sense API behoben. Die aktuelle Struktur der HTTP-Nachrichten funktioniert nur für einen Sensortyp pro Datenupload, obgleich es laut API möglich ist unterschiedliche Sensortypen in einem Datenupload zu kommunizieren.

3.8 RTC-Modul

Mit dem RTC-Modul wird die interne Uhr des Sensorknoten gesteuert. Sie ist nötig, um den Sensorknoten aus den verschiedenen Schlafmodi erwachen zu lassen und wird von einer gesonderten Batterie versorgt. Die Uhr muss zu Beginn jeder Ausführen eines Programms manuell eingestellt, da sie selbst die Uhrzeit nicht beibehält. Wird die Uhrzeit über das RTC-Modul ohne vorherige manuelle Einstellung abgefragt, so erhält man standardmäßig das Datum SSun, 00/01/01ünd eine inkorrekte Uhrzeit. Bei einem Neustart des Sensorknoten wird die Uhr des RTC-Modul zurückgesetzt. Aus nicht bekannten Gründen kann es vorkommen, dass ein Sensorknoten sich während des Betriebs neu startet und somit einen falschen Zeitstempel generiert. Daher ist es nicht möglich zuverlässig einen korrekten Zeitstempel für die Messreihen zu erhalten.