Internet of Things
Snow Monitoring
Project Report

Master’s in
Embedded Computing Systems
Academic year: 2015/2016

Submitted by
Sara Falleni
Elena Lucherini

University of Pisa, June 24, 2016.
Contents

1 Introduction 3
  1.1 Description ................................................................. 3
  1.2 Approach ........................................................................ 3
  1.3 Implementation .............................................................. 4

2 Motes 6
  2.1 Snow level motes ............................................................ 6
  2.2 Avalanche motes .............................................................. 7
  2.3 Simulation .................................................................... 7

3 Proxy 9
  3.1 Receiving data from the motes ......................................... 9
  3.2 Sending data to the client ............................................... 9
  3.3 Border router .................................................................. 10

4 Client 11
  4.1 Classes ........................................................................... 11
  4.2 User’s actions ................................................................. 14
  4.3 Communication .............................................................. 17

5 Conclusions and disclaimers 18
1 Introduction

1.1 Description

The project proposed in this paper is a snow and avalanche monitoring system. The system can be used both in proximity of ski tracks and in zones where the avalanche risk is critical.

Level sensors are employed to detect the amount of snow on a ski track, which will have to stay within a given range. Snow guns nearby are switched on if the snow is below the chosen range; if there is too much snow, maintenance is automatically notified, so that the snow in excess be removed.

As far as avalanche monitoring is concerned, more complex sensors are employed, capable of evaluating the probability of an avalanche; if the probability is higher than a given threshold, the associated user interface alerts the users (ideally, operators at avalanche centers, as well as regular skiers).

A network of motes implements the system. Each mote provides a CoAP server to manage the previously described sensors, and/or the snow guns and communication modules. The motes are connected to a Border Router (BR): when too far from the BR, the motes use the RPL paradigm to exchange data.

1.2 Approach

The implementation is summarized in the following picture.

![Figure 1.1: Overview of the system](image.png)
Whenever the sensor belonging to a mote senses a new value (once every 14 minutes for the snow level motes, and once every hour for the avalanche motes), it sends the new value to a mote acting as a proxy, which collects all the data from the other motes. This part of the communication uses the Simple UDP protocol. The data from the sensors is encoded following the JSON SenML standard.

The motes are connected to a Border Router: when too far from the Border Router, the motes use the RPL paradigm to exchange data.

If the value sensed by a snow level mote is under MIN\_SNOW\_THRESH, the mote automatically activates the associated snow gun. The snow gun takes five minutes to bring the snow level back to its optimal state; it will then be turned off automatically. The proxy is notified of the status of the snow gun when it is activated and, successively, deactivated, through the Simple UDP protocol.

The proxy makes three CoAP resources available to the client: one representing the snow level motes, one for the avalanche motes and one for the snow guns (which are managed by the snow level motes).

Through the GET method, the client can request the status of the motes. The response from the snow level (avalanche) resource will contain all the latest values from the snow level (avalanche) motes. The response from the snow gun resource will contain the status of all the guns, specifying whether they are active. The three resources are also observable.

Through the POST method on the snow level resource, the client can notify the proxy that maintenance has started on a specified mote. When the proxy receives a POST request on the snow level resource, it will send a packet to notify the specified mote. The mote in maintenance shall stop sensing and sending updates until maintenance has been completed. The POST method is allowed on the snow level resource only if the status of at least one snow level mote has requested maintenance – that is, when the snow level detected by the mote is above MAX\_SNOW\_THRESH. After MAINTENANCE\_TIME, the mote shall resume sensing and sending updates to the proxy.

The POST method is available on the snow gun resource, too. This will result in the forced activation (deactivation) of the snow gun related to the specified mote. Thus, the request must contain the ID of the mote to which the snow gun is associated.

The Client is consumed with a Graphical User Interface, through which the user can consult the available information and take the actions described.

### 1.3 Implementation

The proxy and the motes are developed in Contiki, using the C language. Contiki is an operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things devices. The developed systems
The client is developed in Java. A Graphical User Interface is provided, implemented with the Swing Java library.
2 Motes

This chapter describes the 'basic' motes that are used in the network. We can distinguish between two kinds depending on their functions: "snow level” motes and "avalanche” motes.

2.1 Snow level motes

As explained in the previous chapters, the motes belonging to this category are responsible of monitoring the snow level on ski tracks and of managing the snow guns associated to each of them.

The main variables and constants managed by this kind of motes are the following:

- **snow_level**: stores the value sensed by the snow level sensor.
- **maintenance**: specifies whether a maintenance request has been received from the proxy (in this case, it is set to 1, otherwise it is equal to 0).
- **snow_gun**: specifies whether the snow gun associated to the mote is on (1) or off (0).
- **disabled**: specifies whether the snow gun associated to the mote is forced to disabled (1). This happens when the mote receives a request from the proxy to disable its snow gun. When disabled is set to 1, the snow gun cannot be activated until the mote receives an explicit request of resetting disabled to 0 from the proxy.
- **MAX_SNOW_THRESH**: this constant indicates the maximum tolerable snow level on the ski track. Values of snow_level above this threshold are considered critic and maintenance must start.
- **MIN_SNOW_THRESH**: this constant indicates the minimum tolerable snow level on the ski track. Values of snow_level below this threshold are considered critic and the associated snow gun must be switched on.

A snow level mote takes the following actions:

1. It updates snow_level every 14 minutes with the new value sensed (in cm) and it sends the value to the proxy.
2. When a notification of maintenance has been received from the proxy (maintenance = 1) and snow_level is greater than MAX_SNOW_THRESH, it stops sensing until maintenance has finished. It then resets maintenance to 0 and notifies the proxy.
3. When `snow_level` is below `MIN_SNOW_THRESH`, the associated snow gun must be enabled (if `disabled` is set to 0). Similarly, its snow gun must be enabled when an explicit request of switching on the snow gun comes from the proxy. In both cases, `snow_gun` is set to 1 and the proxy is notified.

4. When `snow_gun` is set to 1, the associated snow gun stays on for 5 minutes. It is then switched off (`snow_gun` is set to 0) and the proxy is notified.

All the communication with the proxy is carried by means of the Simple UDP protocol and is encoded according to the SenML JSON standard, as described in details in Chapter 3.

### 2.2 Avalanche motes

Motes belonging to this category are responsible of monitoring the probability that an avalanche might occur.

In this implementation, avalanche motes update the value of variable `prob` with the probability (in %) computed by the associated sensor every hour. The motes then send the new value of `prob` to the proxy.

As with snow level motes, the communication with the proxy is carried on using the Simple UDP protocol and is encoded according to the SenML JSON standard. See Chapter 3 for more details.

### 2.3 Simulation

To simulate a more realistic evolution of the snow level on a ski track and of the probability of avalanche, we developed an array containing values from a uniform distribution. Figure 2.1 shows the plots of the two arrays.
Figure 2.1: Distribution arrays: avalanche probability (above) and snow level (below)
3 Proxy

As explained in the previous chapters, snow level and avalanche motes send their data to a mote acting as a proxy. The proxy manages the communication between motes and the client.

3.1 Receiving data from the motes

As mentioned, the communication between motes and the proxy is carried on Simple UDP packets using the SenML JSON encoding. The format of the packet is the following:

```
{"e": ["n": type, "v": value], "bn": node_id}
```

where:

- **type**: specifies which resource or mote the packet value refers to.
  - Snow level: 1
  - Avalanche: 2
  - Snow gun: 3

- **node_id**: the id of the mote from which the packet is coming.

- **value**: the value contained in the packet, to be interpreted differently depending on the content of type. If type is equal to 1, value is the new level value sensed by a snow level mote (in mm). Similarly, if type is equal to 2, value is the new probability computed by an avalanche mote (in %). If type equals 3, value specifies whether the snow gun associated to mote node_id is on (1) or off (0).

When a packet is received, the content of its value field is stored in an array. The proxy stores three arrays, one for each value of type: snow level, avalanche and snow gun. The length of each array is equal to the amount of snow level or avalanche motes; the length of snow gun depends on the amount of snow level motes, too, supposing that each snow level mote controls a snow gun.

3.2 Sending data to the client

To make the information available to the client, three CoAP resources are available: one that represents the snow level motes, one representing the avalanche motes and one for the snow guns.
**GET.** The resources provide a GET method, which enables the client to receive information on the state of each resource. The format of the CoAP packet sent to the client is the following:

\[
\{ "e": ["v": value1, "v": value2, ..., "v": valueN], "bn": type \}
\]

where:

- **type**: specifies what kind of values the packet contains, depending on the resource:
  - Snow level: `s`
  - Avalanche: `a`
  - Snow gun: `g`

- **value1, value2, ..., valueN** are the values coming from the first, second, ..., n-th mote (or snow gun, depending on **type**).

The resources are observable, too. Every time the proxy receives an update on the state of a resource, a new notification containing the updated state is sent to the client.

**POST.** The snow level and snow gun resources also support POST requests. The format of the packet is as follows:

\[
[type node_id]
\]

When the proxy receives a POST request on a resource, it sends the content of **type** to the mote specified by **node_id** and then sends a success confirmation back to the client. All POST requests are handled by the same function, so the **type** field distinguishes between requests sent on the snow level resource or snow gun resource as follows:

- **Maintenance**: **type** = 1. The POST method has been requested on the snow level resource. By sending the content of **type** to mote **node_id**, the proxy notifies that maintenance has started on that node. All the operations described in Section 2.1 are then executed.

- **Force snow gun**: **type** = 2. The POST method has been requested on the snow gun resource. By sending the content of **type** to mote **node_id**, the proxy forces the snow gun associated to **node_id** to a state that is the opposite of the one it was previously into. If the snow gun was on, it is disabled; similarly, if the snow gun was off, it is now enabled. If the snow gun was previously on, the **disabled** variable is set to 1; otherwise, it is set to 0. See Section 2.1.

### 3.3 Border router

To enable RPL, an additional mote acting as a Border Router is added to the network. RPL provides a mechanism whereby multipoint-to-point traffic from devices inside the low power and lossy network (LLN) towards a central control point as well as point-to-multipoint traffic from the central control point to the devices inside the LLN are supported. Support for point-to-point traffic is also available. In this implementation, the border router runs the example code `rpl-border-router.c` available in Contiki.
4 Client

The client side of the project is correlated with a User Interface, realized in Java using the Swing Library. The UI allows the application to have a look and feel unrelated to the underlying platform. Below is a screen capture of the UI.

![User Interface](image)

**Figure 4.1: User Interface**

4.1 Classes

The User Interface is structured in different classes, as shown in the picture below.
4.1.1 AvMote

The AvMote class represents the implementation of an avalanche mote described in Section 2.2. This class stores the probability of an avalanche to happen and the date and time of the last avalanche. The class has a simple constructor and a few public methods.
to read and change its fields, as described in the following.

- **getProb()**: returns the actual probability of an avalanche to happen; when probability is equal to 1 the avalanche is happening
- **getLast()**: returns the date of the last avalanche
- **setProb(double)**: sets a new given probability for the instance, usually after reading it from the server
- **setDate()**: sets the date of the last avalanche with the current date

### 4.1.2 SnowMote

The SnowMote class represents an instance of a snow level mote, described in [2.1]. It stores the level of the snow sensed and a state. It also has an `activeGun` field that is equal to 1 if the snow gun is on. The public functions are the following:

- **getGun()**: returns 1 if the snow gun is on, 0 otherwise
- **getLevel()**: returns the snow level
- **getState()**: returns the state of that mote in a string format `[OPTIMAL, REFILL NEEDED, GUN ACTIVATED, CALL FOR MAINTENANCE]`
- **changeState(String)**: changes the string state of the mote to the specified string
- **setLevel(double)**: changes the snow level of the mote to the specified number. Usually, the value is read from the server side.
- **changeGun(Int)** change the state of the snow gun

### 4.1.3 ImageIconLabel

The ImageIconLabel is the class that manages all the graphics. It maintains the coherence between the motes states and what is displayed on the Interface. It contains a vector of AvMote and a vector of SnowMote.

### 4.1.4 PanelBackground

The PanelBackground is a simple class that gives the possibility to use an image as a background for a section.

### 4.1.5 SnowClient

The SnowClient class is the one that implements the client side of the system and the communication methods with the server side. One of its private members is an instance of the ImageIconLabel class. The most important feature of this class is the Observing function: as a resource on the server changes its state, the updated state is sent to the Client. Some public methods of AvMote and SnowMote are indirectly used to update the instances of the motes in the Client. Here follows a list of methods of this class, with their description:
• obs(): observes the resources. That is, the Client receives new states of the resources every time there is a state change.

• gunNotify(int, int): this function sends a POST request to the snow gun resource, in the format described in Section 3.2, specifying the related snow mote.

• callMaintenance(int): sends a POST request to the snow level resource. This function is called whenever the Maintenance button is clicked on the UI. Thus, the specified mote is the one related to the button that has been clicked.

• blockMaintenance(int): sends a blocking POST request to the given resource.

4.1.6 ManagerMonitoring

It is the manager class which creates three instances of the SnowClient class and contains the main function. The main function creates a SnowClient for each resource.

4.1.7 Constants

Each class has constant variables that match the thresholds defined in the motes: HIGH_SNOW and LOW_SNOW for Snow Motes and HIGH_DANGER and LOW_DANGER for Avalanche Motes.

4.2 User’s actions

In this section, the information the UI provides are described, along with the actions the user can take.

4.2.1 States

The user can monitor the state of each mote by clicking on the relative icon. Snow Motes show the respective snow level values, while Avalanche Motes show the probability for an avalanche to occur and the date of the last avalanche.
4.2.2 Maintenance

If the level of a SnowMote is under the \texttt{LOW\_SNOW} threshold the user can start Maintenance: this switches on the associated snow gun and, after a given interval, the snow level will be brought back to \texttt{OPTIMAL}.
4.2.3 Snow guns

The user can start and stop snow guns associated to each snow level mote by clicking on the start/stop gun button in the dialog window of a Snow Mote. If the snow gun is off, it will be enabled as a consequence of clicking on the button. If the snow gun was already on, it will be disabled.
4.3 Communication

The communication is implemented using the Californium framework that provides a convenient API for RESTful Web services that support all of CoAP’s features. The two CoAP methods used to communicate are the GET and the POST.
5 Conclusions and disclaimers

This paper shows a possible implementation of a system of monitoring of ski tracks and avalanches. As mentioned, the system was tested and simulated in the Cooja environment.

However, the wide distances between motes in a real life implementation of the system would make the communication hard, if not impossible. The goal of this project is to develop and simulate a wireless distributed system in a low power and lossy environment. No study was conducted on the feasibility of the system in real life cases, as it would fall outside of the purpose of the project.