Riccardo Pinciroli 795582
Matteo Simoni 796384

PERVASIVE DATA MANAGEMENT PROJECT

PerLa: Motes Application

Professor: Fabio Alberto Schreiber

Dipartimento di Elettronica, Informatica e Biomedica

AA 2013-14

1. INTRODUCTION

PerLa is a declarative language and middleware for pervasive systems. Its target is to allow users to query a wireless sensor network, using a full declarative SQL-like high level language hiding the complexity of handling different technologies through a middleware platform. With PerLa a user can ask the system very general question in a pretty abstract way and those questions (High-Level Queries) will be "translated" in Low-level Queries and operations that components have to do.

At a high level of abstraction, the architecture of the system is divided in three main components:

1. **Motes** with several sensors on board with TinyOS;
2. A **channel** that decouples communication between motes and server. It knows the structure of a message that a mote can understand and wraps them in order to make te communication between the other components simpler;
3. **Server** that makes requests to the WSN and that gathers responses. It allows a very high-level view of the system.

Our work for this project is focused on programming motes in order to be able to answer to a query from the server.

2. TinyOS APPLICATION ARCHITECTURE

A TinyOS application has a particular structure composed by some files that are then compiled and installed on motes.
There are two types of components in nesC: modules and configurations. Modules provide the implementations of one or more interfaces (collection of commands and events). Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. It is important to remind that TinyOS is an operating system with an event-driven architecture.
In our application we have:

- **PerLaSensorAppC.nc**
  This file is the top-level configuration file: it wires together the component that are used in the application.
- **PerLaSensorC.nc**
  This file contains the implementation of commands/events provided/used by the application. It is the biggest part of the code, that specifies the behaviour that motes will assume in different situations (at boot, when a timer fires, when they receive a message and so on).
- **PerLaMessage.h**
  This file specifies the structure that a message must assume in the PerLa system. More details are given in Section 3. It also contains some constants.
- **PerLaSetting.h**
  This file contains some structures, used in the application, that producer can modify as he prefers. This will allow him not to touch the code of the application so deeply by accessing only this file.

Moreover, for testing purposes, we have developed two additional files which simulate temperature and humidity sensors. We done it because in TOSSIM (TinyOS simulator) there are not sensors, so we make two of them that produce random values useful for the debug:

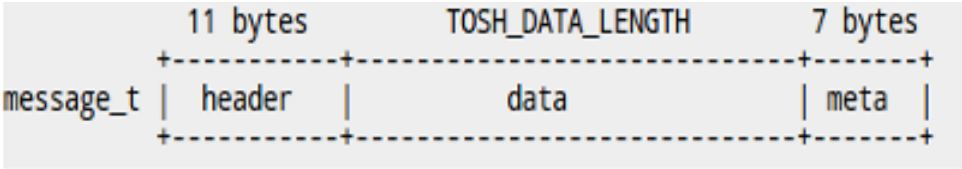
- **MyTempSensorC.nc**
  It generates random values for temperature between 0°C and 40°C.
- **MyHumSensorC.nc**
  It generates random values for humidity between 0% and 100% (it is a percentage value).

In order to test also the serial communciation with the channel, we have implemented two class in Java that are the ones responsible of receiving serial messages and using them:

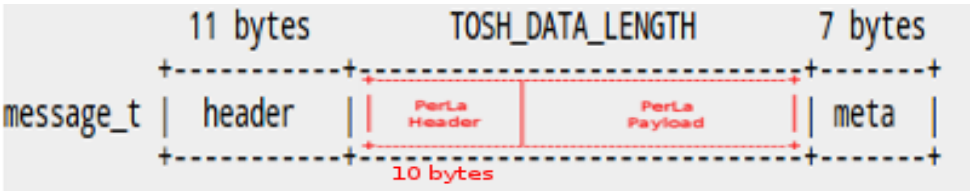
- **TestSerial.java**
  It is the implementation of the class that register a listener for a certain type of packet and that make actions when a packet of the type for which it is listening arrives.
- **TestSerialMsg.java**
  This is the autogenerated class that abastracts the message that is passed through the serial communication and gives the methods to use it.

3. MESSAGE ARCHITECTURE

A TinyOS message is mainly composed by three parts: header, data and metadata.
Header contains general information about the message for the communication; data contains the values that have to be exchanged; metadata that is an additional communication information.
The structure is the following one:



Because it is strongly recommended not to directly access haeder to change its structure, our design choice was to create a PerLa message with its own header and payload. Then, we incapsulate it in a standard TinyOS message. By doing this, we have a total control of our message (both the header and the payload part) and we lay on a TinyOS standard that will always work, even in case of new motes or of changing in TinyOS structure.
What we have done is here:



The header part contains a structure like the following:

```
typedef nx_struct perlaHeader {
  nx_uint16_t id;
  nx_uint16_t timestamp;
  nx_uint16_t type;
  nx_uint16_t numPckt;
  nx_uint16_t numPcktToReceive = 0;
} perlaHeader_t;
```

where id is the univocal number associated to the mote that sends the message, timestamp contains date and hour in which the message was composed and send, type specifies the kind of the message (something like "0 descriptor, 1 periodic sampling (+ one-shot), 2 event based sampling, 3 end of periodic sampling, 4 end of event based sampling, 5 periodic response, 6 event based response"). numPckt and numPcktToReceive are two fields used when the amount of information to send exceed the maximum size of message_t and data must be splitted over more packets. In particular PerLa payload is splitted and PerLa header is replicated with different values of this field: the first one is used to keep ordered the received packet in order to rebuild the original data, the second one

tells to the receiver how many packets he has to get before the communication of the information ends.

The PerLa payload, instead, has the following structure:

```
typedef nx_struct perlaPayload {
  nx_uint8_t data[TOSH_DATA_LENGTH-sizeof(perlaHeader)];
} perlaPayload_t;
```

and it is simply an array of bytes. We do not need to specifies a more complex structure for this one because we tend to keep it more general as possible. The sequence of bytes is then composed and interpreted both from motes and channel by a **predefined XML descriptor file**. This file is pre-loaded on motes at production time and, when a mote is integrated in a wireless sensor network, it sends its descriptor to the channel at boot. At this point both channel and motes know how to write and read the sequence of bytes, then they can put in or get out information.

4. IMPLEMENTED FEATURES

At the state of art, a mote with installed our TinyOS application is able to perform several operations. In particular it can:

- Sense periodically temperature and/or humidity with a specified time period (in seconds). When it is doing this, each time the timer fires it reads the values, prepares the message and sends it to the channel.

- Sense temperature and/or humidity one-shot. This means that, when a mote receive the request, it sense the chosen parameter once and sends the response.

- Sense temperature and/or humidity by event. Until now the events supported are "temperature greater than" and "humidity greater than", even if it will be pretty simple to extend to different ranges.
  An event reading is, in other words, a periodic reading with a control on the sensed data: if the sensed data satisfies the condition, than it is sent to the channel, otherwise it is discarded.

Until now it is possible to have more sampling operations in parallel with the only constraint that it can be one operation running, per type and per sensor.

- Send a packet with a sensed value or the XML descriptor to the sink (and split it in more packet if the information that has to be sent is too big)

- The sink is able to route the packet that receives from the other motes to the channel (e.g. a PC) via serial communication.

5. DESIGN CHOICES

In the implementation of the code of motes, we have done some design choices:

- In order to guarantee an easier possible migration from TinyOS to some other embedded operating systems or a more comfortable manteinament of the code, we implemented some functions and a task that separate the logic from the applicative code:

    - **splitMessage()**: it is called when a message is too long to be sent in one only packet. It takes the big message, split it into chuncks that, with an header replicated for every partial message, can fit the maxPayloadSize of a TinyOS packet. Moreover it fills the field of the header in which it is specified the total number of packets in the sequence and the number of each packet in the sequence itself. When a perlaMessage is composed, this function puts it into a queue of transmission and starts the task used for sending packets.

    - **sendMessageInQueue()**: this is the task responsible to send packet in queue. It checks if the queue is not empty and sends the first packet. Once it sent the packet, it calls itself recursively until the queue is not empty.

    - **setHeader(uint16_t type)**: it takes as an input the type of the message (specified by the protocol) and fills the id, timestamp (both automatically got) and type fields of the header of a perlaMessage.

    - **composeAndSend()**: it takes the header and the payload with the field already compiled, puts them into a perlaMessage and sends it. If the size of the perlaMessage is too big with respect to the size of a TinyOS message, it calls the splitMessage() function.

    - **report_error()**, **report_sent()**, **report_received()**: they make action with respect to the specified event. At the state of art, they simply toogle leds.

- As for timers, also sensors, even if they are unique on the board, have been abstracted with more than one at software level. We have done this because more than one action could be run at the same time on the same sensor and the same mote. So we have the PeriodicTemperatureSensor, the EventTemperatureSensor and the OneShotTemperatureSensor. The same thing has been done for Humidity Sensor.
- At the state of art, the code to split a packet, as we have already said, is on motes, but the code to recompose a packet is not in the mote that works as sink, but it is on the channel, at a higher level. We decide to do that because the complexity of this type of operation can be better managed from the Java channel and this make more uniform the behaviour of the sink that only receives a packet and sends it to the channel (indipendently from the type of packet received).

6. HOW TO MAKE THE PROJECT WORKS

In order to make our project works, there are some steps to follow:

A. Copy the "PerLa_Sensor" folder into /tinyos-2.x/apps
B. Cut "MyTempSensorC" and "MyHumSensorC" files from "PerLa_Sensor" folder and paste them in /tinyos-2.x/tos/system
C. Edit "run.py" file by modifying the value of the constants (N_MOTES, SIM_TIME, TOPO_FILE, NOISE_FILE, ..)
D. Open the terminal, go to the PerLa_Sensor folder and type command "make micaz sim-sf"
E. open the serial forwarder on port 9001 typing "java net.tinyos.sf.SerialForwarder -comm sf@localhost:9001&"
F. run the java program that accepts messages from the serial port 9002 "java TestSerial -comm sf@localhost:9002"
G. Run the python simulation typing "./run.py".

In the file "run.py", that contains the python command used in order to run the simulation, some particular file are called:

- linkgain_nonoises5nodes.out: it specifies the topology of the simulated network

- /tinyos-2.x/tos/lib/tossim/noise/meyer-heavy.txt: it specifies the noises on the various links of the network

7. FROM TOSSIM TO REAL MOTES - HOW TO

If you want to deploy our application on a real wireless sensor network, you have to pay attention to some points:

- Because TOSSIM can simulate only one type of node at the time, we had to put together the code of the sink and the code of sensing motes. In a real environment there will be two different applications (one for the sink and the other one for sensing motes). In order to create these two applications, you have to split the two branches that are present in almost every event. The code in the "if(TOS_NODE_ID == 0)" branch is the one for the sink, the code in the "else" branch is the one for other motes.

- You can remove the dbg statement that are used for debugging in TOSSIM, even if in a real application they would not be visualized.

- For our simulation, we created two different fake sensors, one for temperature and the other one for humidity. In a real application, you have to substitute "MyTempSensorC" and "MyHumSensorC" in "PerLaSensorAppC.nc" with the name of the interface of real sensors on your mote.

- In "setHeader()" function, you have to substitute the "sim_time()" function that assigns a value to the timestamp field in the header with the "getTimestamp()"-like function on your mote.

8. OPEN PROBLEMS

Our project works as it is expected to do and as it is described above, but there are of course some points that can be improved or extended. The ones that we have thinked about are:

- Extend the event-based sampling condition. At the state of the art, motes are able to control only one type of condition for event-based sampling that is "measure greater than". It could be useful to have condition like "lower than" or "between .. and ..", maybe being able to choose among them in a dynamic way.

- If the wireless sensor network is very wide, it can be useful to implement some routing protocol in order to cover bigger distances and to save battery on motes (it is not strange that an hop-to-hop communication can be cheaper than a direct communication).

- Battery is one of the weakness of motes and wireless sensor network. If some tricks are run based on battery state of motes, they probably can perform better and for a longer time.

- Packet Resending Management: it could be a good idea to manage also single packet sending with a queue in order not to lose a packet when the channel is busy and a mote is trying to send it.