LINGI2146
- Mobile and Embedded Computing -
Group Project

Group L
Marcio Antonio De Carvalho Borges
Adrien Widart

May 2019

Github : `https://github.com/decarvalhobo/LINGI2146-Project`
A readme is available to correctly simulate our system on Cooja.

1 General structure

Our system works thanks to two different parts : the motes which are in charge of the data generation and the gateway which is in charge of the communication between the root mote and the MQTT broker. We used Mosquitto[1] for this MQTT broker.

1.1 The motes

1.1.1 The data

The motes have been implemented to generate some data about temperature and humidity. They can send this data in two modes (sending periodically or only on change). The motes have some parameters that you can change at the beginning of the code (see code 1). As shown, you can set the delay (in seconds) between two data sending. By default, all motes send it periodically but you can change the value to false in order to set the "only on change" mode. We have also implemented a function that let you change the sending mode of a mote by pressing the Z1 button. The last defines are about data : you can change the channels name and set the minimum and maximum temperature bounds (since humidity is a percentage, its value obviously varies between 0 and 100).

1.1.2 The processes

The mote code contains 3 different processes. However, all motes but the root actually use two of them. Here are the processes :

- `manage_motes_network` is the process in charge of the organisation and operation of the network. It is always running on every mote (see section 3 for the explainations).
- `data_sender` generates and sends the temperature and humidity data according to some parameters, as explained in the previous subsection (1.1.1).
- `socket_listener` is only executed by the root. It receives messages from the gateway and transfers them through the tree.

[1]`https://mosquitto.org/`

```
#define DELAY_BETWEEN_DATA        5
#define PERIOD_DATA_BY_DEFAULT    true

#define TEMP_CHANNEL_NAME         "temp"
#define MIN_RAND_TEMP             -20
#define MAX_RAND_TEMP             80

#define HUM_CHANNEL_NAME         "hum"
```

Code 1: The mote defines

```
typedef struct {
  uint8_t        type;
} Basic_Msg;

typedef struct {
  uint8_t        type;
  Mote_Status    status;
} Mote_Status_Msg;

typedef struct {
  uint8_t        type;
  Broker_Status  br_status;
} Broker_Status_Msg;

typedef struct {
  uint8_t        type;
  char*          channel_name;
  rimeaddr_t     mote_addr_from;
  int            data_value;
} Data_Msg;
```

```
typedef struct {
  rimeaddr_t     parent_addr;
  uint16_t       parent_rssi;
  int32_t        hops_to_root;
  unsigned long  broker_version;
} Mote_Status;

typedef struct {
  unsigned long  version;
  bool           temp_required;
  bool           hum_required;
} Broker_Status;
```

Code 2: Message format and structures

1.2 The gateway

The gateway is the link between the broker and the sensors network. It has two roles: the main one is to format into MQTT messages the data received from the root before transmitting them to the broker. The second role is an optimization where it notifies the sensors network on whether they can send data for a specific subject. To do so, the gateway tracks the number of subscribers per topic and communicates with the tree root (this will be discussed on section 4.2).

1.2.1 Formatting data into MQTT message

As said before, the data messages are generated by motes and then forwarded to the root. To be able to process them, the gateway listens for the incoming messages from the root through a socket.

When there is an incoming message :

1. The gateway first checks if it is a data message (it must contain an ip address, a topic name and a value following this format : ;IP_ADDR;TOPIC;VALUE)
2. We construct the MQTT message to send it with the extracted data from step 1.
3. We publish it to the broker thanks to the MQTT client command line tool.

2 Message format

We had different communication needs and it's the reason why we created 4 different message formats (as you can see on code 2). All four start with the `uint8_t type` which describes the format message. This is a practical feature : when a mote receives a message, it juste needs to read the first byte in order to know which message type the mote juste received (and it will then read more bytes if needed). Regarding the formats :

- `Basic_Msg` is the simplest one. It is used to send a flag (for example, send a message indicating a disconnection to the motes tree)
- `Mote_Status_Msg` is used to communicate the status of a mote to the others. As written in code 2, a status contains several informations such as the broker version number or the number of hops between the mote and the tree root. This message format is usefull to create and maintain the motes tree, as it will be explained in the next section (3).
- `Broker_Status_Msg` provides some information about the broker : which channel is listened to by at least one subscriber. (See the section 4.2 for more details)
- `Data_Msg` is the main message format. This is the one used to send some data to the root. It contains the channel name as a string, the mote that generated this data and the data value as an integer.

3 Organisation and operation of the network

3.1 Tree creation

When a mote A is not connected to the tree, it sends a broadcast every second with a simple flag. This flag means the mote is not connected to the tree and would like to. This allows a receiver (mote B) that is connected to the motes network (the tree) to answer with an unicast message in order to give its status. That way, when the motes A receives this answer, the mote B becomes its parent and A is now connected to the tree (the mote stores its new status in a `Mote_Status` as written in code 2).

Moreover, if the mote A receives multiple answers from the neighbouring devices, it will choose the closest to the root (in hops). If two neighbouring motes have the same number of hops, the mote A will choose the one with the best Received Signal Strength Indicator (RSSI).

Last, each time a mote has a new parent, it broadcasts it. That way, neighbouring devices can adapt their own status if necessary and as quickly as possible.

3.2 Maintaining and optimizing the tree

When a mote is connected to the tree, it periodically sends a broadcast message that provides its status. We decided this feature for several reasons :

- Let's imagine 3 motes A(root) $\leftarrow$ B $\leftarrow$ C where the arrows indicate the parents. It seems correct, but what if C is actually close enough to directly send message to A ? What if, when C asked for the status of its neighbours, the A-mote's answser was lost ? Thanks to the periodic status messages, the mote C will eventually know that the mote A is close enough to become its new parent. So, therefore, this feature always makes it possible to have a tree with the best paths.
- Let's imagine 4 motes A(root) $\leftarrow$ B $\leftarrow$ C $\leftarrow$ D. Then, let's imagine a 5th mote connected to A and close enough to communicate with mote D. Thanks to the periodic status messages sended, the mote D will eventually know this new mote and change its parent for this better one.
- To detect a disconnection with the parent, because this one has moved or does not exist anymore (see section 3.3).
- To detect that a new broker status exists (see section 4.2).
- In the 4 previous cases, the problem is not solved if the message is lost. Again, it is the periodic sending that almost totally increases the probability that each mote will receive the status.

3.3 Loss of the parent mote

Each mote has a counter `no_news_from_parent` initialized at 0. At each cycle of the `manage_mote_network` process, the mote increments this variable. However, when a mote receives a status message from its parent, it resets the variable to zero. That way, when a mote has no news from its parent for two cycles, it sends an unicast message to the parent to get the status. In the next cycle, if the variable has not been reset, the mote considers its parent as lost. Therefore, the mote is no longer connected to the tree and it sends a broadcast message with a flag meaning that it is disconnected.

An other case, more simple : when a mote receives a disconnection flag from its parent (meaning that the parent is no longer connected to the tree), the receiver changes its status and directly send a disconnection flag. That way, the motes belonging to the disconnected branch are informed as soon as possible.

4 Optimization

4.1 Aggregate messages

Unfortunately, we have not been able to implement this optimization.

4.2 Stop sending if there is no subscriber

4.2.1 The gateway side

For this optimization we first added steps to the treatment of incomming message (as explained in in section 1.2.1). When a new topic is encountred by the gateway, we store it and inform the root that we don't want any more data about this topic. The known topics are stored in a dictionary with the topic name as key and the subscriber identifiers list as values.

In the other hand, we have a thread that process the logs from the MQTT broker. Since the gateway lauches the MQTT broker, it has access to its logs. The process detects subscription and unsubscription [2] messages sent to the broker. We treat the logs as follows:

- When we see a subscription in the log :
  - we extract the subscriber identifier and the topic.
  - if we never received data for this topic we add it to the dictionary set of keys and inform the root that we want data for this topic
  - we append the subscriber identifier in the dictionary for this topic.
  - if this topic was stopped we inform the root to restart the transmission for this topic
- When we see an unsubscription :
  - we extract the subscriber identifier.
  - we go through all the keys to withdraw this specific subscriber identifier.
  - if we encounter a topic that doesn't have any more values and if it is not already marked as notified, we inform the root to stop the transmission for this topic and mark it as notified.

4.2.2 The mote side

We have created a structure specific to this optimization : `Broker_Status` (see code 2). This status contains a version number which can only be incremented by the tree root. That way, a larger number always means a newer version and there can be no conflict. So, when the broker sends some new instructions to the root, this one changes the broker status and broadcasts it.

Moreover, when a mote receives a broker status with a version number greater than the one the receiver has stored, it updates its broker status and broadcasts the new status.

Last, when a mote receives a status message, it always checks the broker version number. If the number is greater than its, it asks (with a unicast message) the broker status to the status message sender.

[2] We didn't succeed to generate unsubscription messages, so for the detection we used the disconnection of the suscriber as unsubscription.