Università di Pisa

Internet of Things

A.A. 2019/2020

PROJECT DESCRIPTION

Hospital environment control system

Name: Matteo Zini

Student ID: 533197

E-mail: matteozini96@gmail.com

1. Scenario description

The project consists in a simulation of the environment of an hospital.

The available sensor nodes are:

- Patient-monitor: simulates a patient's vital signs monitor. The implemented node has, as an exemplification, a heartbeat sensor and a blood pressure sensor.
- Patient-alarm: simulates an alarm that is triggered when the patient's vital signs deviate from safe values. It has three alarm levels that are simulated with the mote's LEDs and represent three growing severity levels.
- Doctor-dashboard: a node that represents a screen where a doctor can read about the alarms generated by the patients' conditions variation. The cause of the alarm, the specific value of the parameter and the generating node ID are displayed on the screen, which is simulated with the nodes' serial line window.
- Fire-sensor: this node represents a fire-detector, that can be located in every "room" of the hospital.
- Fire-alarm: this node simulates the presence of a fire alarm, that is activated whenever a fire sensor is triggered. The alarm is displayed using the motes' red LED.
- Temperature-sensor: the node simulates the presence of an environmental temperature sensor.
- Air-conditioning: this node simulates the presence of the air conditioning system that can output both hot and cold air. The activation of this node is displayed using the nodes' green and yellow LEDs. Moreover, this node sends a message to the temperature-sensor node that generated its activation, in order to let it emulate the effect of the activation of the healing/cooling system.
- Border-router: the standard border router, provided by Contiki-ng in the examples folder.

These types of sensors have been chosen so as to exploit the widest possible subset of options provided by the cooja-mote interface and the SenML encoding.

2. Sensors interface

In order to have an easy way to test all the functionalities of the system, all the sensors are provided with an interface to modify the data that they send to subscribers.

In detail, it is possible to modify their readings with the following interfaces:

- Patient-monitor: during its operations, the patient monitor readings can be altered sending commands through the serial line. The accepted commands are in the form

  [hb|Mp|mp]=<value>

  This way the user can force the value read by the temperature to *value*, respectively for the heartbeat, maximum blood pressure and minimum blood pressure sensors.

  If not influenced from the external user, the value of the three parameters automatically varies by a little random offset at each sampling.

- Fire-sensor: the fire sensor can be triggered by pushing the mote's button. Pressing again, the sensor can be set back to its normal state.

- Temperature-sensor: the temperature sensor readings can be influenced pressing the mote's button. If the button is pressed once, the temperature is set to 50°C, if the button is pressed twice consecutively, the temperature is set to 0°C.
  If not influenced from the external user, the value of the temperature automatically varies by a little random offset at each sampling.
  The temperature sensor readings can also be altered using a post request. This way the air conditioning system can communicate its state to the sensor, implementing a simulated effect of its behaviour. When the sensor apprehends the air conditioning system is in COLD state, the temperature is decreased by 10°C at each sampling, if the air conditioning system is in HOT state, the temperature is increased by 10°C at each sampling. When the temperature is back into a normal range, the application will deactivate the heating/cooling system as it would do during normal operations.

3. Actuators interface

All the actuators have an interface to display their state to the user, during the simulation:

- Patient-alarm: the patient alarm is activated when the patient's vital signs are not in the standard values range. The alarm has 3 possible level, depending on the gravity of the situation: LVL1, LVL2 and LVL3; the three levels are represented by the green, yellow and red LEDs respectively.
  The alarm accepts a new value from the application if and only if the new level is higher than the current one. In order to go back to a lower level, a manual reset from the CLI is required, thus making sure that the alarm is not neglected even if the patient's conditions go back to normal on their own.

- Fire-alarm: the fire alarm is activated when one of the temperature sensors detects the presence of fire. The alarm is represented by the mote's red LED inside the simulation. The alarm can be switched off only with a manual command from the CLI, thus making sure that the alarm is not deactivated if the sensor does not sense fire anymore.

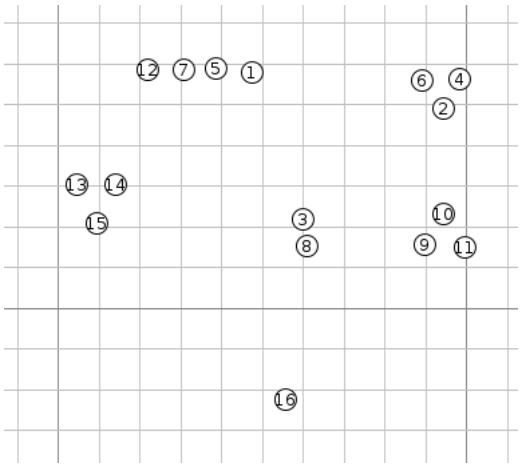
- Doctor-dashboard: each doctor's dashboard in the system receives a message whenever an alarm is raised due a patient's conditions getting out of normal ranges.
  The dashboard displays a description of the problem, the value that caused the alarm and the address of the related patient-monitor.

- Air-conditioning: the air conditioning system interface consists in the activation of the green and yellow LEDs when the system is in HOT and COLD state. The system also sends a post message to the temperature sensor to let him know about its activation and simulate the variation of the temperature. The system is automatically stopped if the temperature gets back into the "normal" range.

4. Simulation environment

The simulation configurations provided with the project are contained in the two files in the main directory:

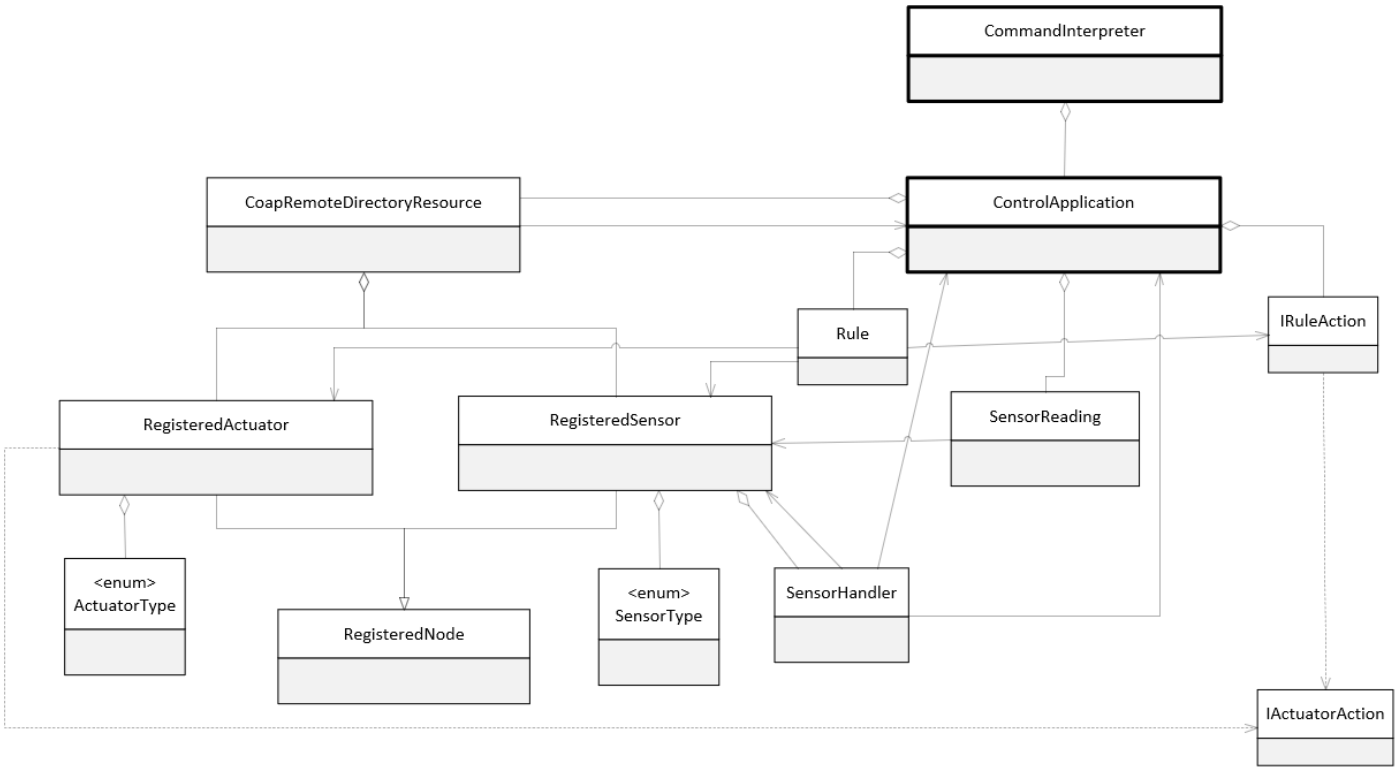
- Simulation.csc: basic simulation with 1 node per type. The sensor-actuators coupling is provided in the server/autoapply.txt file (see sections 5 and 6 for more details).
- Simulation-multiroom.csc: this simulation recreates a more realistic hospital environment. The nodes configuration is the following:



The 1,5,7,12 group of nodes represents the doctors' room. Here we can find the border router (1), the fire alarm (5), the dashboard (7) and a fire-sensor (12).
Each group of 3 nodes represents a patient's room; the nodes are a patient monitor, a patient alarm, and a fire sensor. Node 3 and 8 are the temperature sensor and the air conditioning actuator respectively and node 16 is secondary dashboard.
The sensor-actuators coupling is provided in the server/autoapply2.txt file (see sections 5 and 6 for more details).

5. Server architecture

A very simplified class diagram for the server application is shown below:



The program main function is inside the *CommandInterpreter* class. This class implements the CLI and instantiates the *ControlApplication* class at startup. During normal operation, this class is in charge of interpreting the commands typed by the user or inside a script file and to perform the correct operations on the application object to execute the command.

The *ControlApplication* class contains the main logic of the application. It starts the CoAP server, stores all the sensor readings, the sensor-actuator association rules and, through the *CoapRemoteDirectoryResource* class, the data structures describing the registered sensors and actuators, which are represented by the *RegisteredSensor* and *RegisteredActuators* classes, both extending the *RegisteredNode* base class. Every sensor and actuator is also associated to a *Sensor/ActuatorType* from an enumeration containing all the possible sensor/actuators types. The sensors also keep the reference to their event handler, which has itself a reference back to the sensor, in order to know, when a message arrives, which sensor is associated with that reading. Moreover, an handler keeps the reference to the application object, since, when a reading is received, it has to be stored in the main application object.

The *IactuatorAction* interface is an interface implemented by enumerations that contain the possible action for each actuator type. So, for example, if we create a fire-alarm node, we will also create an enumerate called *FireAlarmAction*, implementing that interface and including the possible actions that can be performed by that actuator, such as SWITCH_ON or SWITCH_OFF, in addition to the interface's function getActionCommand(), which returns the String version of the chosen command, with the syntax that is understood by the node (e.g. "cmd=on").

The *IRuleAction* interface must be implemented by all the rule actions that are present in the system. A rule action consists in an association of a sensor type with an actuator type, with an additional function containing the behaviour of the rule action. At runtime, when a sensor and an actuator of the type specified in the rule action are registered on the platform, the user can decide to apply that rule on the specific sensors, creating an object of the class *Rule*, which links the concrete sensor and actuator together with the rule action. From this moment on, every time the handler receives data from the sensor, the *check(SensorReading)* of the rule action is called and, if the function's return value is not null, the action specified by the aforementioned return value, that is an instance of an enumeration implementing the *IactuatorAction* interface, is sent to the actuator.

To clarify this concept, continuing the previous example of the fire alarm, if we want to pair it with a fire sensor, we need to create a class implementing the *IRuleAction* interface inside the appropriate vector in the application class. This class will store information about the fact this rule is going to associate a fire-sensor and a fire-alarm. We will also need to override the *check(SensorReading)* function, writing the code that will decide which will be the action to send to the fire alarm depending on the sensor reading. At runtime, when the real sensors will be registered on the platform, the user, with the "apply" command, will tell the application to apply the rule on the two devices. The system will thus create a Rule object that connects the real devices and the *IRuleAction* object. Every time a reading from the fire sensor arrives, it will be passed as a parameter to the *check* function, which will return null if no action should be performed, *FireAlarmAction.SWITCH_ON* if we want to switch on the alarm or *FireAlarmAction.SWITCH_OFF* if we want to switch off the alarm.

6. Use of the system

From the CLI, the user can perform multiple operations to read data, verify the status of the system and send commands to the actuators.

To activate the system, the user should run the server and start the cooja simulation. The user should then wait for all the sensors to be registered (the list of currently registered devices can be seen with the list command) and then apply the desired rules.

In order to avoid the manual insertion of each rule at every execution, the user can store all the *apply* commands in a script file. Two script files are provided for the simple and the complete simulation files included in the project. With this architecture, the user can add a node at any time during the execution and there is no need to modify the code of the nodes or the server to specify an association between sensors and actuators, thus making possible to use the same code on an indefinite number of motes. The script files shall be executed when all the nodes have registered to the application.

The available commands are displayed in the output of the "help" command. The same information is also printed in case an incorrect command is input.

- **list [sensors | actuators]**: show a list of the registered sensors and actuators. If "sensors" or "actuators" is specified, the system show only the sensors/actuators.
- **read [<sensor_type>] [-t <timestamp>]**: print all the values received by the sensors in chronological order. The readings can be filtered by type, if "*sensor_type*" is specified, or by minimum timestamp, if "-t *timestamp*" is used.
- **set <res_number> <value> [params]**: send a command to the actuator specified by "*res_number*". The number is the one displayed in the *list* command's output. The "*value*" must be chosen among the list that is shown with the *commands* command. If the specific action requires some parameters, they can be specified at the end.
- **rules [applied]**: show a list of the rule actions in the system. If "applied" is specified, the command will show a list of the applied list, with a reference to the involved nodes.
- **apply <rule_num> <sensor_num> <actuator_num>**: this command must be utilized to apply a rule action to two real deviced. The sensor and actuator number to be utilized are the ones displayed in the *list* command's output; the rule action number to be utilized is the one displayed in the *rules* command's output.
- **unapply <applied_rule_num>**: remove a rule from the applied rules. The rule action number to be utilized is the one displayed in the *rules applied* command's output.

- **script <file>**: this command allows to execute some CLI commands contained in a file. This is particularly useful because, since the sensor-actuator pairings can be specified only at runtime, when the devices are actually registered on the system, the user should manually write all the *apply* commands in the command line. To avoid this issue, since the sensors and actuators are always in alphabetical order (ip + path string utilized), a text file containing the pairings can be prepared once for all and utilized every time, when the registration process is complete. The file path must be an absolute path.
- **commands**: display a list of the action that can be performed on each type of actuator that can be utilized by the system.
- **help**: show the help message.
- **exit**: terminate the execution.