CPS Humidity Homework

Janos Benjamin Antal
antal.janos.benjamin@gmail.com

January 19, 2018

Homework description

The task in this homework is to design and develop one subsystem of an imaginary "smart" university control system. The subsystem's goal is to help keep the humidity of a classroom in the desired range. The subsystem has to interferes with the control system based on information gathered from the sensors in a classroom and some online services. The subsystem must contains at least a local and an online module communicating each other. Furthermore, the subsystem should grant a minimal level of assurance The subsystem should store the gathered information and controlling decision in the cloud for further alanysis.

- **Sensors**: a humidity sensor inside the classroom
- **Interferer**: a LED on the teacher's desk to indicate the opening/closing of the window: the window should be opened when the LED turns on, and should be closed when the LED turns off
- **Services**: a weather forecast and a schedule information provider service
- **Assurance**: in case of the weather is too hot or the air too polluted the subsystem shouldn't indicate the opening of window. When the classroom is not scheduled, the maximum of humidity should be bigger than it's scheduled.

Requirements

The requirements above are really high level requirements, and in a real situation they should be refined to low level specifications. The requirements are specified as sentences and also most of them as Gherkin test cases:

- *Turn on the only when the outer conditions are right and the humidity reach the upper bound:*

```
Given the system with right humidity value in the classroom
And the LED is turned on
And the temperature is not too high
And the air pollution level is not too high
When the humidity reaches the upper bound
Then after the next refresh the LED turns on
```

- *Turn off the LED when humidity reaches the lower bound:*

```
Given the system with humidity value above the upper boundary
And the LED is turned on
And the temperature is not too high
And the air pollution level is not too high
When the humidity reaches the lower bound
Then after the next refresh the LED turns off
```

- *Turn off the LED when the temperature becomes too high:*

```
Given the system with humidity value above the upper boundary
And the LED is turned on
And the temperature is not too high
And the air pollution level is not too high
When the temperature becomes too high
Then after the next refresh the LED turns off
```

- *Turn off the LED when the air pollution becomes too high:*

```
Given the system with humidity value above the upper boundary
And the LED is turned on
And the temperature is not too high
And the air pollution level is not too high
When the air pollution becomes too high
Then after the next refresh the LED turns off
```

- *Never turn on the LED when the air pollution level is too high:*

```
Given the system with right humidity value in the classroom
And the LED is turned off
And the air pollution is too high
When the humidity raise above the upper boundary
Then after the next refresh the LED is still turned off
```

- *Never turn on the LED when the outer temperature is too high:*

  ```
  Given the system with right humidity value in the classroom
  And the LED is turned off
  And the temperature is too high
  When the humidity raise above the upper boundary
  Then after the next refresh the LED is still turned off
  ```

- *The subsystem must communicate with sensors placed in the classroom:*

  ```
  Given the system with three fully operational sensor
  And the system can get humidity value from each sensor
  ```

  The subsystem can turn on the LED:

  ```
  Given the system with right humidity value in the classroom
  And the LED is turned off
  When the humidity reaches the upper bound
  Then after the next refresh the LED turns on
  ```

- *Use encryption and certification in communication with cloud:*

  ```
  Given the system
  When the system sends status information to the cloud
  Then the cloud server proofs that its the right server
  And the message is encrypted
  ```

- *After the decision made, the actual status must be saved to the cloud.*

- *Parameters (humidity boundaries, maximum air pollution level and maximum temperature) can be configurated remotely.*
- *The subsystem must communicate with weather forecast system.*
- *The subsystem must communicate with schedule information provider.*
- *The subsystem must communicate with external systems by "best practice" protocols.*
- *The subsystem must take into consideration the unavailability of external data sources when deciding on turning on/off the LED.*

Prerequisites

To build the application on Linux Mint 18.3 the following steps are needed:

- Install required packages with apt:
  - `libboost-all-dev`
  - `libcurl4-openssl-dev`
  - `libssl-dev`
  - `uuid-dev`
  - `rapidjson-dev`
  - `build-essential`
  - `cmake`
  - `g++`
  - `git`

  Use the following command:

  ```
  $ sudo apt install libboost-all-dev libcurl4-openssl-dev
      libssl-dev uuid-dev rapidjson-dev build-essential cmake g
      ++ git
  ```

- Install cURLpp with the steps described here:

  ```
  $ sudo apt-get remove libcurlpp0
  $ mkdir curlppbuild
  $ cd curlppbuild
  $ git clone https://github.com/jpbarrette/curlpp.git
  $ cd curlpp
  $ cmake .
  $ sudo make install
  ```

- Install AzureIoT C SDKfrom source. After building the SDK use `sudo make install` to copy the header and the lib files to the system include path.
- Download and install RTI Connext DDS 5.3
- Set the `NDDSHOME` environment variable to the root of RTI Connext DDS, e.g.: `/opt/rti_connext_dds-5.3.0`

Build

Build

1. Clone git repository

   ```
   $ git clone https://github.com/antaljanosbenjamin/
       cps_homework.git
   ```

2. Generate source code from .idl files

   ```
   $ cd cps_homework
   $ $NDDSHOME/bin/rtiddsgen -language C++11 -stl -d DDS/Config
       /common -replace idl_files/Config.idl
   $ $NDDSHOME/bin/rtiddsgen -language C++11 -stl -d DDS/
       Decision/common -replace idl_files/Decision.idl
   $ $NDDSHOME/bin/rtiddsgen -language C++11 -stl -d DDS/
       Schedule/common -replace idl_files/Schedule.idl
   $ $NDDSHOME/bin/rtiddsgen -language C++11 -stl -d DDS/
       Humidity/common -replace idl_files/UvegHaz.idl
   $ $NDDSHOME/bin/rtiddsgen -language C++11 -stl -d DDS/
       Weather/common -replace idl_files/Weather.idl
   ```

3. Build

   ```
   $ mkdir build
   $ cd build
   $ cmake ..
   $ make [-j 8]
   ```

Run demo

1. Start humidity publisher

   ```
   $ ./cps_main h <humidityDataFilePath>
   ```

   As result of this command the application will read the data file and start to publish a humidity value every 4 minutes. The file shall contains a humidity value per line. Each humidity value is a decimal number. See example file.

2. Start IoTEdge

   ```
   $ ./cps_main e <weatherApiKey> <azureConnectionString> <
       scheduleFilePath>
   ```

   The meaning of parameteres are the following:

   - `weatherApiKey`: an API key for http://api.airvisual.com
   - `azureConnectionString`: the connection string of the device used Azure IoT Hub
   - `scheduleFilePath`: path to a CSV file which stores the schedules time intervals. See example file.

   The IoTEdge module is responsible for comunication with Azure IoT Hub, the weather information system and also to send schedule information through DDS topic.

3. Start humidity controller

   ```
   $ ./cps_main c
   ```

   The controller receives the required informations and sensor values and also make decisions based on the collected data. It also sends the decision input and output to the IoTEdge in order to store them in the cloud.

Implementation

The system consists three modules:

- IoTEdge
- Controller
- Publisher

The publisher module exists only for testing purposes, so this documentation doesn't contains it's details.

IoTEdge

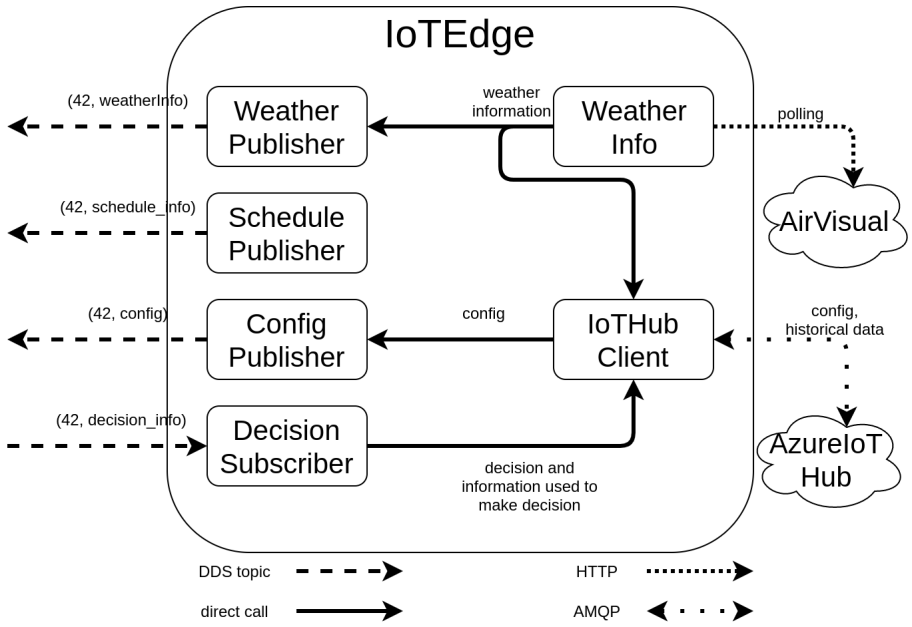

Figure 1: Architecture of IoT Edge

As the figure 1 shows, the IoTEdge translates the messages received from the cloud and the weather forecast API to DDS messages and vica versa. The DDS Interface Description Language exactly describe the contents of the used DDS messages:

```
struct Config {
    double maxTemperature;
    unsigned long maxPollution;
    unsigned long minHumidity;
    unsigned long maxHumidity;
};

struct Schedule {
```

```
    boolean scheduled;
    unsigned long until;
    unsigned long sentTS;
};

struct UvegHaz{
    string ID; //@key
    double Value;
    long TimeStamp;
};

struct Weather {
    double  temperature;
    unsigned long tempTS;
    unsigned long pollution;
    unsigned long pollTS;
};

enum Decision { CLOSE, OPEN };

struct DecisionInfo {
    unsigned long decisionTS;
    Decision decision;
    Config config;
    Weather lastWeather;
    Schedule lastSchedule;
    UvegHaz lastHumidity;
};
```

The most complex part of the IoTEdge module is the IoTHub Client. It is responsible for translating and communicating between the IoTEdge and AzureIoT Hub. It sends the received sensor, weather and decision data to the cloud, and also receives config message from it. A config message contains the maximum outside temperature, the maximum outside pollution and the desired minimum and maximum temperature in the classroom. For example:

```
{
    "maximumTemperature" : 50,
    "maximumPollution" : 55,
    "minimumHumidity" : 42,
    "maximumHumidity" : 46
}
```

- maximumTemperature: a temperature value in Celsius. If the outside temperature reaches this limit, the controller logic mustn't decide to open the window.
- maximumPollution: a pollution value in Air Quality Index. If the outside pollution reaches this limit, the controller logic mustn't decide to open the window.

- minimumHumidity, maximumHumidity: the desired range of humidity expressed as relative humidity. The goal of the system it to keep humidity within this range.

IoTHub Client

The IoTHub Client is a reusable part of the implementation. It's public interface are really simple:

```
class IoTHubClient final {
public:

    IoTHubClient(const std::string &connectionString,
                std::function<void(const rapidjson::Document &)>
                    receivedMessageConsumer,
                bool trace = false,
                uint32_t keepAlive = 240);

    ~IoTHubClient();

    void start();

    void stop();

    void sendMessage(const rapidjson::Document &message);
};
```

The implementation is based on an official Azure IoT C SDK example. Usage of the module also really simple. The constructor takes an Azure IoT Hub decive connection string, a message consumer function and two other parameters. More information about the last two parameters can be found in the referenced example. To start receiving/sending messages with an existing instance of IoTHubClient, the `start()` function has to be called. After the calling, the instance starts to receive messages from IoT Hub and also sends messages to it when `sendMessage(msg)` is called.

Abstract DDS classes

All of the `XXXPublisher` and the `XXXSubscriber` in this project is derived from `AbstractPublisher` and `AbstractSubscriber`. They are generic template classes in order to make easy to create DDS publishers and subscribers. The interface of `AbstractPublisher` is the following:

```
template<typename DataT>
class AbstractPublisher {
public:
    AbstractPublisher(int domain_id,
                      const std::string &topic_name);

    virtual ~AbstractPublisher();

    void publishData(const DataT &data);
```

```
};
```

The `DataT` template parameter should be DDS generated message's type. The `publishData(data)` function can be used to send a message to the given topic.

The `AbstractSubscriber`'s interface is very similar:

```
template<typename DataT>
class AbstractSubscriber : public dds::sub::
    NoOpDataReaderListener<DataT> {
public:

    AbstractSubscriber(int domain_id,
                       const std::string &topic_name,
                       int pollSeconds);

    virtual ~AbstractSubscriber();

    void on_data_available(dds::sub::DataReader<DataT> &reader)
        override ;

    void startReceiving(std::function<void(const DataT &data)>
        consumerFunction);

    void stopReceiving();
};
```

The `on_data_available(dds::sub::DataReader<DataT> &reader)` inherited from the base class, and shouldn't be called by the user. To start receiving messages the `startReceiving(consumerFunction)` function has to be called. After that, the subscriber will call the `consumberFunction` when a new message arrives.

Controller

The Controller receives the DDS messages with the required informations (scheduling, weather information, measured sensor values and configuration messsages), and makes decision based on them. Figure 2 shows the exact process of decision making. Decision making runs periodically in each 5 minutes, and decision is also made on receiving every input messages. It's important to make decision independently to message receiving, because the last known informations can be outdated. In case of any information is outdated, the decision output must be `CLOSED` in order to avoid opened window under not safe circumstances.

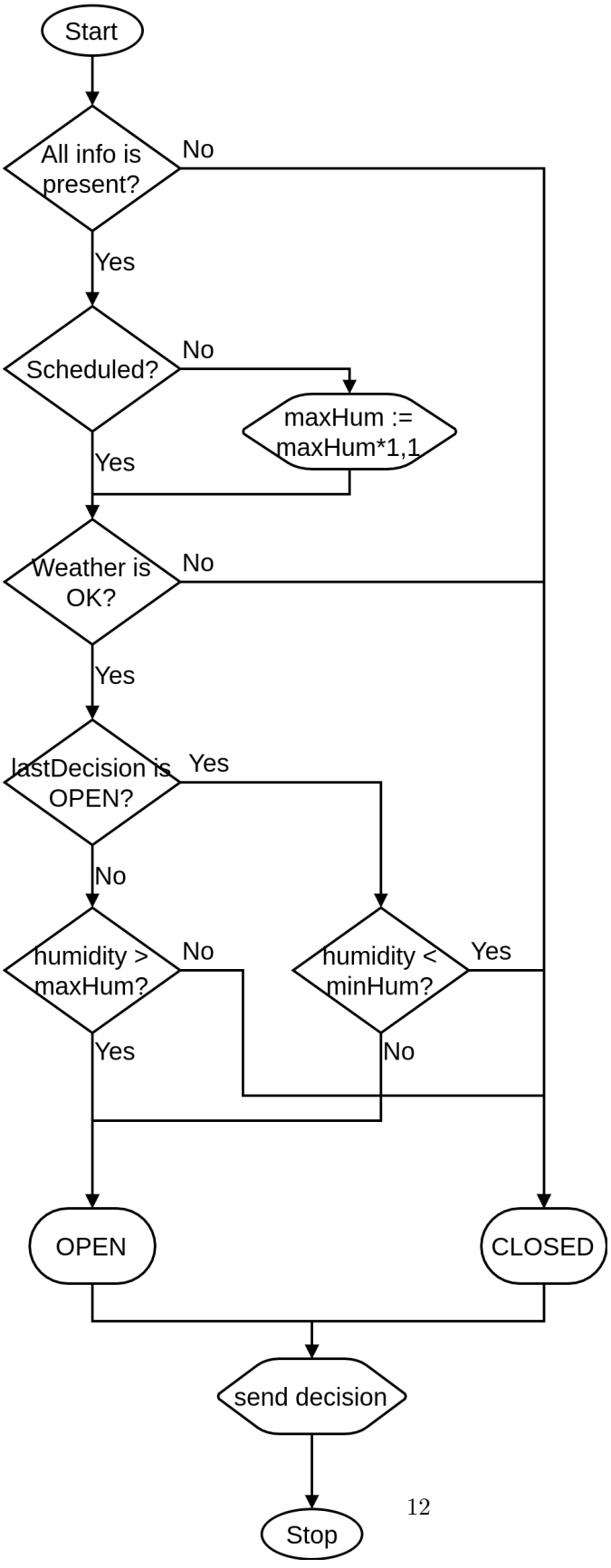

Figure 2: Flow diagram of decision logic