CPS Humidity Homework

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

January 14, 2018

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

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.

System architecture

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

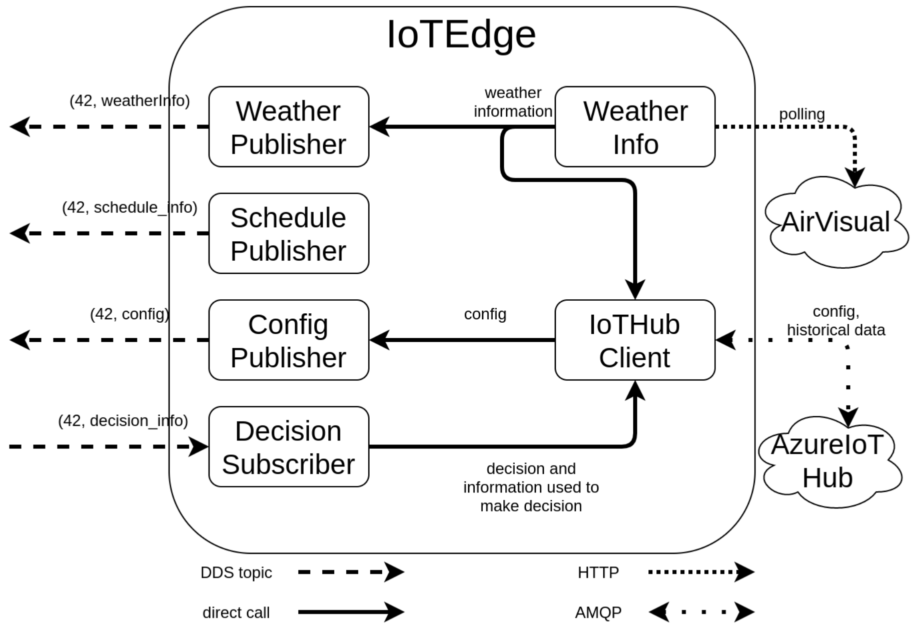

Figure 1: Architecture of IoT Edge

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.

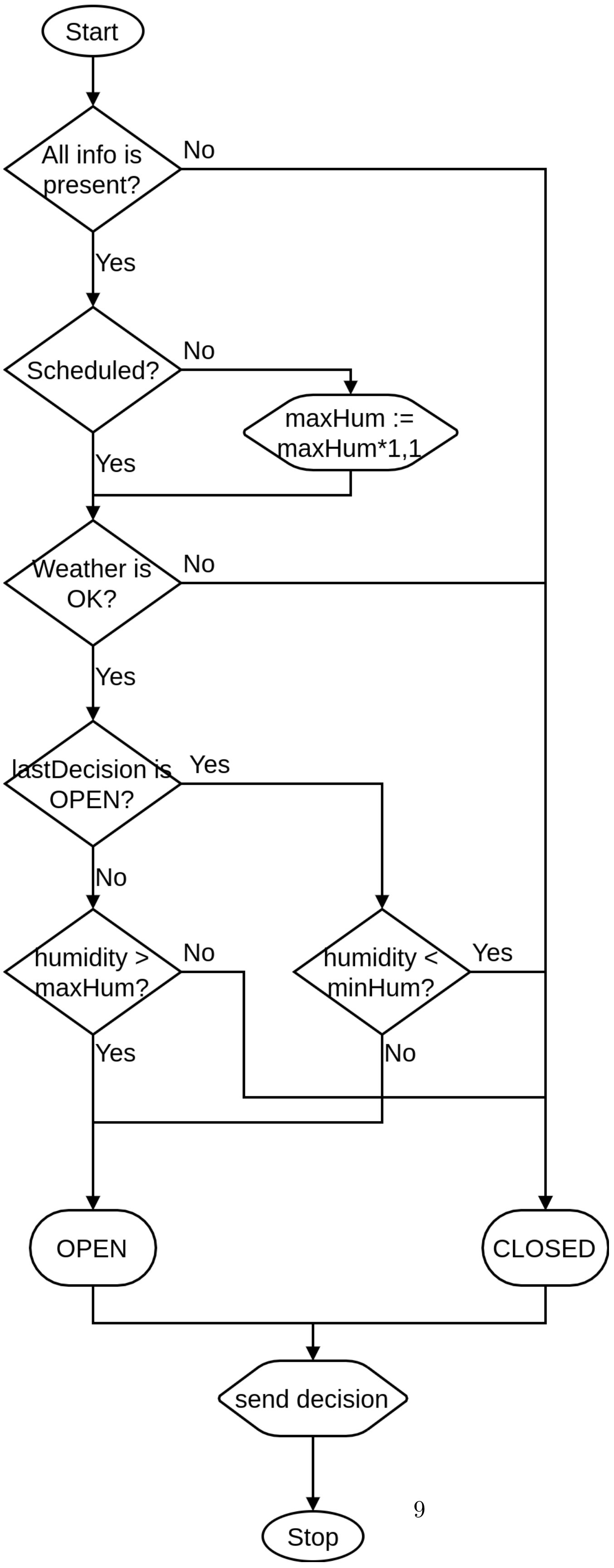

Figure 2: Flow diagram of decision logic