Arduino and Odysseus (2)

In the following I will show you how to process sensor data from an Arduino device with the Odysseus data stream management system.

Arduino as a sensor platform

The sensor we use in this Hand-on is a DHT11. The DHT11 is a low cost sensor capable to measure the temperature and the humidity of it's surrounding. The DHT11 has actually four pins but uses only three pins: VCC, GND, and Data. We connect the power pins of the sensor to VCC and GND and connect the Data to pin 7 of the Arduino device.

blocked URL

Next, we start the Arduino IDE to write our program that will read in the values measured by the DHT11 and provide them over a serial connection. To access the sensor and read the temperature and humidity values from within Arduino we use the DHT library from adafruit. Download the library and copy it to your sketch folder so that it is accessible from the Arduino IDE.

blocked URL

In the following, you see the complete listing of the source to read the values and transfer them as a comma separated string over the serial connection:

```
    #include "DHT.h"

#define DHTPIN 7
#define DHTTYPE DHT11

// initialize the sensor:
DHT dht(DHTPIN, DHTTYPE);

void setup() {
Serial.begin(9600);
dht.begin();
}

void loop() {
float temperature = dht.readTemperature();
float humidity = dht.readHumidity();
// print results as CSV:
Serial.print(temperature);
Serial.print(",");
Serial.print(humidity);
Serial.println();
delay(10*1000);
}
```

In the first line we include the DHT library. After that, we create an instance of the DHT class by setting the DHT pin to 7 and tell the library that we are using the DHT sensor version 11. In the setup method we start the serial connection and initialize the DHT. During the loop method we read the temperature and the humidity value and create a comma-separated string that is send over the serial connection. If everything is done, connect the Arduino to an USB port and compile and upload the program. That's all from the hardware part, now we go over to Odysseus to perform the processing.

First Odysseus query

First fetch a fresh nightly build from the Odysseus website. Download the combined Odysseus Server and Studio bundle and unzip it. Before we start with our processing we need an additional feature to read data from the USB port. To do so, run the downloaded Odysseus Studio GUI by calling the "studio" executable. You can download new additional features using the "Install new feature" entry from the menu.

blocked URL

Install the feature "Wrapper Feature" that includes additional libraries to access sensor data from different devices and protocols. After that we can create a new Odysseus Project by issuing the "New" command under the "File" entry or by just pressing "CTRL-N". After creating a new project we create a new Odysseus Script. A script includes the data source definition and the processing query for our data source. Our first script will be written in the CQL language, so select CQL Basic as a template during the Odysseus Script dialog. The template already includes three lines defining our parser, the transformation configuration, and a "RUNQUERY" command:

```
    #PARSER CQL
#TRANSCFG Standard
#RUNQUERY
```

The CQL language is based on the SQL language used in databases like MySQL or PostgreSQL, so it is the perfect starting point when you worked with databases before. Later, we will also take a look at the PQL query language for more advanced stream processing. The transformation configuration sets different parameter for the used data model and metadata. However, the standard configuration should fit our needs right now. The last line tells the parser that we want to run the next lines as a query. Additional Odysseus Script commands can be found here.

To add the Arduino as a source we add the following lines after the “RUNQUERY” command:

```
    DROP STREAM Arduino IF EXISTS

ATTACH STREAM Arduino (temperature DOUBLE, humidity DOUBLE)
WRAPPER 'GenericPush'
PROTOCOL 'CSV'
TRANSPORT 'RS232'
DATAHANDLER 'Tuple'
OPTIONS ( 'port' '/dev/ttyACM3', 'baud' '9600');
```

Here, we first drop the source if it already exists and create a new one with the name “Arduino”. The schema of the stream has two attributes, namely, the temperature and the humidity as provided by our Arduino device. Both attributes are from the type double. The wrapper keyword defines the behavior of the stream, either push or pull. In a pull-based manner, Odysseus requests the data from the sensor; in a push-based manner, the device pushes the data to Odysseus. In our case, the Arduino device pushes the data over the USB connection, so we select “GenericPush”.

The next few lines define the transport handler, the protocol handler, and the data handler. The transport handler is responsible for the communication with our device. Therefore, we select the RS232 transport handler to communicate with our Arduino over the USB connector. To read the data transmitted by the Arduino, we use the CSV protocol handler to parse the data. The data handler converts the parsed data into the internal format. To do so, we use the Tuple data-handler that is the default for most applications.

The last line defines multiple options for the transport and protocol handler. In particular, these options are the device port and the baud rate used for the RS232 handler. Depending on your setup, you may have to change the port.

Now that we have a source, we can perform our processing on that source. To do so, we start a new processing block and write down our processing query. For the sake of simplicity, our first query should just select both attributes so we can use them for, e.g., visualization.

```
    #RUNQUERY
SELECT * FROM Arduino;
```

You might remember this kind of query. It is the easiest way to read all rows from a database. And that’s also what we do here; we simple read all measurements from the Arduino device. To start the query, either press “CTRL+R” or click on the “Execute Script” entry in the context menu. You will see your query in the “Queries” window. To see the measured data from the Arduino just do a right click on the running query to open the context menu and select the “Show Stream Elements” entry.

OK, let’s do some real processing now. To do so, we change our query a little bit to continuously produce the average of the temperature of the last hour. This is done using the AVG keyword and the definition of the window we want to calculate the average on. In this case we use a sliding time window with a size of one hour and an advance of one millisecond:

```
    #RUNQUERY
SELECT AVG(temperature) FROM Arduino [SIZE 1 HOUR ADVANCE 1 TIME];
```

Thus, Odysseus calculates the average of all temperature measurements of the last hour and outputs this average as shown in the next screenshot. Here, you also see one example of visualization using the Line Chart; many more are available using the context menu.

blocked URL

Advanced data stream processing

So fare, we have seen how Odysseus can be used to process measurements from an Arduino device in just a few lines of CQL code. Now we want to make some more advance processing including a pattern matching on the data and the transmission of processing results. To do so, we first have to install the “CEP and Pattern” feature using the “Install new feature” entry from the menu. This feature includes additional operator to perform so-called complex event processing. To use the new operators we will switch the query language to PQL. The PQL language allows connecting operators directly instead of describing a query in a declarative language like CQL. Also, we will now transfer processing results the same way we received the data from our Arduino. First, we create a new Odysseus Script like before, but select the PQL Basic. Thus, our first three lines change to:

```
    #PARSER PQL
#TRANSCFG Standard
#RUNQUERY
```

Next we define our source as before, but now in PQL using the ACCESS operator:

```
    Arduino := ACCESS({ SOURCE = 'Arduino',
TRANSPORT = 'RS232',
DATAHANDLER = 'Tuple',
WRAPPER = 'GenericPush',
PROTOCOL = 'CSV',
SCHEMA = [['temperature', 'DOUBLE'], ['humidity', 'DOUBLE']],
OPTIONS = [['port', '/dev/ttyACM3'], ['baud', '9600']] })
```

You see that there is no big difference in the definition of a source. Now we come to our processing query. This time, we will use the SASE operator to detect a pattern in the stream. The pattern itself should match a situation when the temperature drops more than 20% compared to some value during the last 60 seconds, i.e., someone opened a window in the room.

```
    match = SASE({schema=[['temperature','Double']],
type='Result',
query='PATTERN SEQ(Arduino+ a[]) WHERE skip_till_any_match(a[]){
  a[1].temperature >= 0.8 * a[a.LEN].temperature
 } WITHIN 60 seconds RETURN a[a.LEN].temperature'
}, Arduino)
```

The SASE operator we use here takes three parameters, namely, the output schema, the type name of the output, and the query. The query parameter itself takes a pattern description written in the SASE pattern language.

Talking to the outside world

Right now, our query process the data from the Arduino device and finds situations when the temperature drops by more than 20% in one minute. So, we also want to get warnings when Odysseus detects such a situation. We could now use one of these fancy new social media stuff like Twitter, but I prefer the more old-school email. That’s why I use the SMTP transport handler to directly send the result of the SASE operator to an email address using the following operator in PQL:

```
    out = SENDER({ SINK = 'Mail',
TRANSPORT = 'SMTP',
DATAHANDLER = 'Tuple',
WRAPPER = 'GenericPush',
PROTOCOL = 'CSV',
OPTIONS = [
 ['from','odysseus@example.com'],
 ['to','me@example.com'],
 ['subject','Open window'],
 ['host','smtp.example.com'],
 ['tls','true'],
 ['username','Alice'],
 ['password','***']
] }, match)
```

The configuration of the operator is similar to the access operator used to receive the data from the Arduino device. We use the generic push wrapper, the tuple data handler, and the CSV protocol handler to transform the result back into a comma-separated value. However, now we use the SMTP transport handler to forward the processing results as an email consisting of a comma separated line with the processing result. To do so, we further have to setup some options for the SMTP transport handler including the sender email address, the receiver email address and the SMTP hostname including authentication data. Other protocol handler are also available here.

This Hand-on is based on https://www.kuka.cc/odysseus/2014/02/06/Odysseus-and-Arduino/