CNT5517/CIS4930 Mobile Computing

Spring 2024

Professor Sumi Helal

Lab 4 – Build Microservices for your Smart Space

Using Atlas Thing Middleware and IoT-DDL

Due Date: 11:55pm Sunday, April 7, 2024

Introduction to Lab 4

The Virtual Smart Space (VSS) you and your group have created in Lab1 is composed of three layers: private cloud, one or two laptops as Edge, and a set of RPi platforms as IoT things. These layers utilize a private network (e.g., via a wireless hub) for connection, enabling the top-bottom flow of instructions (e.g., commands from your IoT applications) and the bottom-top flow of data (e.g., results of running the instructions). In Lab3, you and your group focused on the first two layers of the VSS, the physical layer and the Edge layer. For the physical layer, you designed a circuit with push buttons and LEDs, and programmed your RPi platform with two modes to utilize such a circuit to implement the functionality of a metronome. While, for the Edge layer, you designed and created an HTML-based dashboard as a web interface that runs on your laptop to interact with the RPi platform remotely via RESTful APIs.

The **services** (or functionalities) offered by IoT things like RPi platforms influence the design and implementation of IoT applications. A temperature sensor (as an example of a simple IoT thing) may offer a few services for the users, that include reporting the current temperature value in F (or in C). While a smart lamp (as an example of a more complex IoT thing) usually offers more services, that include on/off switching, controlling the brightness and setting timed operations. Users and developers utilize these available services to compose IoT applications. The more services offered by the IoT thing, the wider the range of IoT applications that can be designed and programmed involving that thing. However, to design meaningful IoT applications, users and developers must identify the **relationship** between such services. **Think of a relationship as a virtual link between two or more services.** While there is a wide range of relationships to consider, in this lab assignment, we will consider only the following two relationships:
***1) Order-based relationships*: Service B may run only after service A runs successfully to completion**, and

2) *Condition-based relationships*: Service B runs conditionally based on a specific return value resulting from running Service A.

Part One — Connecting multiple services to build an IoT application

In this part of Lab 4, **as a stepstone towards the final project**, you and your group will expand your work on Lab 1 and Lab 3 to implement the following scenario:

- **Step1: Prepare an application scenario**. Think of an interesting IoT application that you would like to design using the sensors and actuators available with your group.

- **Step2: Prepare the physical layer**. Following your idea from Step1, prepare RPi-1 platform with a sensor (or sensors) that measures and reports a value of a certain parameter from the environment (e.g., humidity, noise, light, temperature), and RPi-2 platform with an actuator (or actuators) (e.g., servomotor, LEDs, others). You should prepare the hardware aspect (the connections) and the software functionality (C/C++ coding – no Python allowed in this assignment) as you did in Lab 3. Test this step (test that you can get sensor(s) values, and you are able to actuate the actuator(s)).

- **Step3: Prepare the Edge layer**. Your two RPi platforms must connect and report their **available services** to your Edge device. Each service is reported by as a vector **(RPi identifier, service name, the number of required input parameters for the service, and the input parameter names).** For example **(RPi-2, Read_Temp, 1, F_or_C).** The Edge device then displays such reported service information for the user through a web-based dashboard.

- **Step4: Prepare a simple programming environment (IDE).** The user, through the dashboard, can utilize one or both of the relationships defined above to compose a simple IoT application. **You are really designing a very small, very limited IDE at this point.** Think of a **text-based (command line)** or **graphical-based (point, click, drag)** approach to describe (develop) such an IoT application. The former approach is obviously easier, but you are free to choose it or the GUI approach.

- **Step5: Build an IoT application.** Now that you built the IDE, use it! **Compose a simple application and run it.** As you compose the application using your IDE, the Edge translates such an IoT application to a set of service API calls, as defined in the relationship(s), to the corresponding IoT things on the RPi platforms.

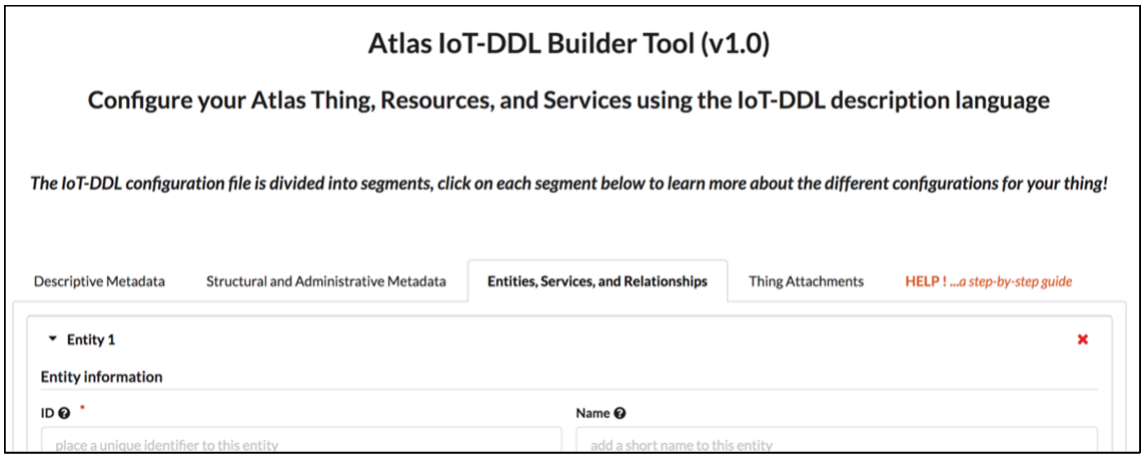
- **Step6: Display the results.** The Edge controls the order of API calls, collects the results from the IoT things, and displays the results to the user.

Part Two — Describe and build your IoT Thing

In this part of the assignment, you will use the **Atlas Thing Middleware** which is the key component of the Atlas Architecture framework (https://github.com/AtlasFramework/Main). Atlas thing middleware is an operating layer (firmware) embedded within an IoT thing that allows for the automatic integration of the IoT thing with the rest of the VSS smart space (integrating with other IoT things). The middleware enables communication between IoT things, with users, and the cloud. The middleware and its **IoT Device Description Language (IoT-DDL)** allow the users (e.g., device owners) to define and use the services provided by these IoT things. They also allow IoT programmers to discover and combine available services to create IoT Applications in a smart space.

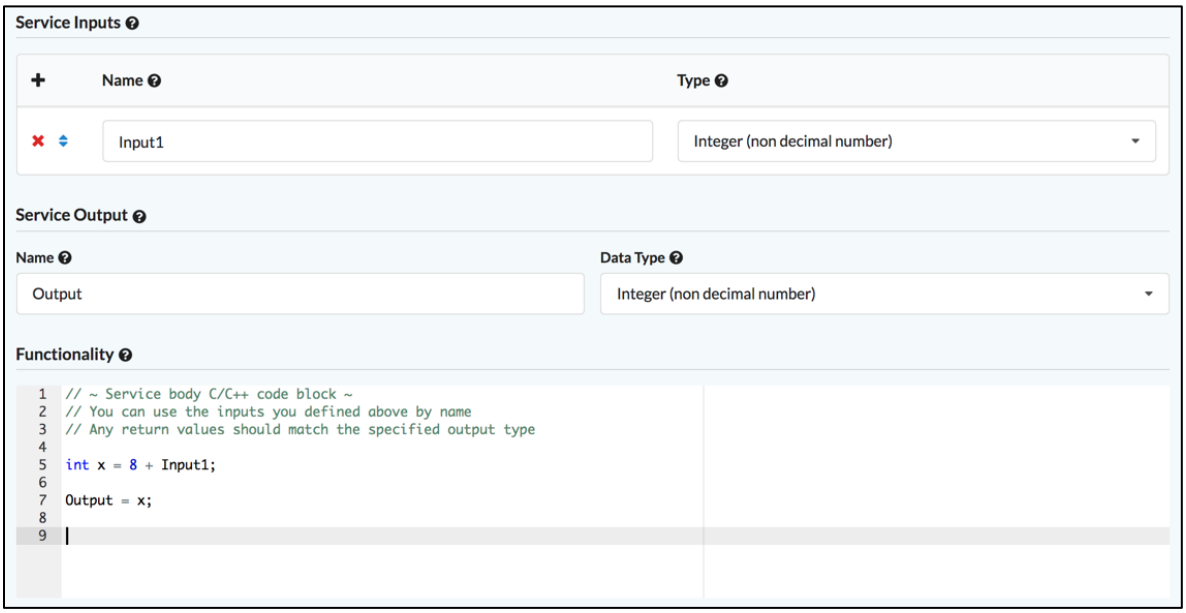
Step1:

The Atlas thing middleware requires the use of the IoT-DDL device description language to configure an Atlas thing and enable it to perform a wide set of functionalities (e.g., advertisement and discovery, dynamically creating microservices, and enabling users and other things to use and invoke such services through their respective APIs). IoT-DDL is an XML-based declarative language that describes a thing in terms of inner components, identity, capabilities, resources, and services (in addition to other elements to be covered later). In this step, you will use the **IoT-DDL Builder web-based tool** (https://github.com/AtlasFramework/IoT-DDL) to build an IoT-DDL specification file that describes your things you developed in Part One.



Consider each Raspberry Pi as a thing with one or more inner entities (called bit-things). Under "*Descriptive Metadata*", you can describe the main attributes and parameters of a thing. Under "*Entities, Services, and Relationships*" you can create inner entities **and include the code of the services you designed in Part One** provided by each entity. Create a separate entity for each bit thing (e.g., entity for each sensor and entity for each actuator).

For each service you create, you may specify the inputs expected for this service (the default is no inputs) as illustrated below, under "*Service Inputs*". Under "*service output*" you specify the expected output (the service returns void by default). Under "*Functionality*", you should develop the C/C++ code of this service (you should bring and "reuse" the code you developed in Part One). As an example, you developed a service -as illustrated below- that takes a single integer input (named Input1) and expected to return output value (named Output). In the functionality window, you can start to develop your C/C++ code, and use "Input1" and "Output" within the code (no need to include the return statement, the middleware automatically includes this).



The IoT-DDL builder tool also provides you with some ready-made APIs and commands that you may use as shown below. Select the desired API, fill the inputs, and click *add* this API to your code.

At this point, you would have created your first IoT-DDL specification for all bit-things that you will attach to your Raspberry Pi things.

An example IoT-DDL for a Servo motor entity (attached to a Raspberry Pi) offering a single service can be viewed using this link:
https://github.com/AtlasFramework/IoT-DDL/blob/master/Resources/Servo_Motor.xml

Step2:

In this step you will import the Atlas thing middleware to your Raspberry Pi and add the IoT-DDL file you developed in Step1 by following the instructions as detailed in this link: https://github.com/AtlasFramework/AtlasThingMiddleware_RPI It is important that you make sure the IoT-DDL filer is indeed placed in the correct directory inside the middleware, as per the instructions given via the link above.

Step3:

When you run the Atlas middleware at end of Step2, it will parse the different parts of the IoT-DDL you included, generate microservices from the services you described, build JSON-based tweet messages summarizing the parameters of the IoT-DDL, and start broadcasting these tweets to the VSS smart space. Other things, or application software deployed in the same smart space can capture these tweets and take advantage of them in anyway meaningful.

In this step, you will create a demo app named “Demo IoT Application” in the VSS that simply invokes the services you described in the IoT-DDL through their appropriate API calls. Design and implement this Demo IoT Application using the programming language you find appropriate (e.g., C/C++, Java, Python) to unicast API calls (to the IP of your thing over port address 6668). The API call is a JSON-based message, structured as follows:

```
{ "Tweet Type"     : Service call,
  "Thing ID"       : the id of your smart thing as declared in the IoT-DDL,
  "Space ID"       : the id of your smart space as declared in the IoT-DDL,
  "Service Name"   : the name of the function you would like to call,
  "Service Inputs" : (list of expected inputs)
};
```

Sending such call to an Atlas thing triggers the appropriate handles and the thing responds with JSON-based results back.

Let us assume, in the IoT-DDL, you made *MySmartThing01* as your smart thing id, and *MySmartSpace* as your smart space id. You also developed two services, *Service1* that accepts no inputs and *Service2* that accepts two integer inputs. The structure of the API calls in this example will therefore be:

```
char Call1[] = "
{ \"Tweet Type\"   : \"Service call\",
  \"Thing ID\"     : \"MySmartThing01\",
  \"Space ID\"     : \"MySmartSpace\",
  \"Service Name\" : \"Service1\",
  \"Service Inputs\" : \"()\"
}";
```

and,

```
char Call2[] = "
{ \"Tweet Type\"   : \"Service call\",
  \"Thing ID\"     : \"MySmartThing01\",
  \"Space ID\"     : \"MySmartSpace\",
  \"Service Name\" : \"Service2\",
  \"Service Inputs\" : \"(3,9)\"
}";
```

You should comment your code adequately to explain what specific demonstration it accomplishes.

Extra-Credit: Virtual Smart Space Across Two Groups (10% extra credit)

You will be able to call and test your group thing services from one of your development machines as long as the machine is on the same network (direct Ethernet connection or WiFi other than UF's network, as described in the previous lab document).

For extra credit, two groups may attempt to create a cross-group "virtual smart space" (VSS) via an external VPN server that will allow the thing services belonging to the two groups to interact with each other (tweet and hear the tweets across the two groups, and of course invoking services across). You may use an OpenVPN server and host it somewhere. If you opt for Open VPN, you should install it on your RPi with **`sudo apt install openvpn`**, and should create an **OpenVPN configuration file** and place it in the **`/etc/openvpn/client`** directory in your RPi's, renaming it to **`vss.conf`**. To connect to the VSS, run the command **`sudo systemctl start openvpn-client@vss`**. After a second, you should see the VPN connection running successfully with **`sudo systemctl status openvpn-client@vss`**. If there was an error, ensure the configuration file has been named correctly and is in the right location.

You will now have an IP address within the cross-group VSS, likely in the `10.254.0.0/24` range. If you run `ip -br a`, you should see the IP under the `tap0` interface. Your device and its services are now visible and available to the other group's devices and services. To disconnect from the VSS, run **`sudo systemctl stop openvpn-client@vss`**.

NOTE: Please follow good IoT security practices and change the password (with `passwd`) for your RPi to something from the default, since it is now visible to others outside your group.

Submission Details

You will submit a *.zip* file containing a folder for each part (folder names: PartOne and PartTwo). For Part One, you should include the C/C++ service implementations (one or more services for each one of your RPi platforms) as a **main(){}** program with the services as functions. For Part Two, use a separate folder to include the IoT-DDL file you have generated using the Builder tool. Also include the code for the demo IoT application you have developed to use the services. Remember, comments (documentations) are very important and do affect your

final grades as per the rubric. Keep track of who did what in the group and include a readme file in the folder for Part One, that includes a short description of the distribution of effort. Be ready to demo and go over your code with the TAs during lab grading. All group members must be available and present during the online grading sessions. Absences will earn their absentees a grade deduction.

Instructions to follow to make sure that Atlas Thing Middleware works on your Raspberry Pi.

Firstly, I am going to copy and paste the exact instructions that are given in the GitHub repo of the Atlas Thing Middleware(https://github.com/AtlasFramework/AtlasThingMiddleware_RPI), for reference.

Prepare your Atlas smart thing on Raspberry Pi through the following steps:

Step1: run the following linux commands through terminal:

- sudo apt-get update
- sudo apt-get upgrade
- sudo apt-get install gcc-6 g++-6 build-essential //usually install the latest gcc and g++
- sudo apt-get install doxygen
- sudo apt-get install cmake cmake-curses-gui
- sudo apt-get install libboost-all-dev
- sudo apt-get install curl libcurl4-openssl-dev
- sudo apt-get install autogen

Step2: Get the latest version of the middleware:

From Github, download the zip version of the middleware on your RaspberryPi, then unzip the folder.

Step3: Install cppMicroservices library

- unzip the folder named CppMicroServices-development under Atlas-IoT_Thing/lib/ of the middleware, and keep in the lib directory
- cmake CppMicroServices-development/
- sudo make
- sudo make install
- LD_LIBRARY_PATH=/usr/local/include
- export LD_LIBRARY_PATH

- sudo ldconfig
- //the new version installs the library in /usr/local/include/cppmicroservices4/ rather than /usr/local/include/
- sudo mv /usr/local/include/cppmicroservices4/ ~/Desktop/
- sudo mv ~/Desktop/cppmicroservices4/cppmicroservices/ /usr/local/include/

Step4: Install WiringPi library and enable the hardware interfaces

- unzip the folder named WiringPi-master under Atlas-IoT_Thing/lib/WiringPi-master/ of the middleware
- cd to the WiringPi-master folder
- ./build
- sudo apt update
- sudo apt upgrade
- sudo apt install rpi.gpio
- sudo raspi-config
- under “Interfacing Options”, enable both I2C and SPI

Step5: Compile and Build Atlas middleware

- Navigate to the directory of Atlas-IoT_thing (use cd command) and Compile as follows:
- cmake Main/
- make

Step6: Add an IoT-DDL

- Use this builder tool to build an IoT-DDL file for your Atlas thing.
- Navigate to the directory of Atlas-IoT_thing and add the genenerated IoT-DDL.xml file to the /ConfigurationFiles directory (replace the default file)

Step7: Run Atlas middleware

- Through therminal, and under the directlory of the middleware, run the following command:
- ./Atlas

Due to differences between the Raspberry Pi version used to develop/test the Atlas Thing Middleware and the versions most students are using, some changes to the above instructions are necessary.

Note: If anyone is using an older version of the Raspberry Pi (Raspberry Pi 3+), the only edit they need to make is the editing of the service.tpl file as mentioned in the Step 6 below. This needs to be done only if they want to use pigpio. If WiringPi works on their Raspberry Pi, no need to make this change.

These are the changes:

1. Follow the above step 1 and step 2 as is. There is no issue in those steps.
2. In step 3, after unzipping the folder and after running the command "cmake CppMicroServices-development/", download the patch that has been provided to you anywhere on your Rapsberry Pi.
3. Now, run the following command in the terminal of your Raspberry Pi "patch -p1 < <absolute_path_of_patch_file>"
    a. Eg: The absolute path of the patch in my Raspberry Pi was '/home/subhash/Downloads/required_patch.patch'. The command that I used was "patch -p1 < /home/subhash/Downloads/required_patch.patch"
4. After this, complete the rest of the Step 3 provided in the GitHub Readme as is.
5. If you are using WiringPi to connect to your Raspberry Pi, follow the exact same steps given in step 4 of the GitHub Readme file.
6. If you are using pigpio, follow these steps:
    a. Make sure to install the pigpio on your Raspberry Pi. To do this, run the following command "sudo apt install libpigpiod-if-dev"
    b. Now, open the service template file that is available at the following path "/AtlasThingMiddleware_RPI-master/Architecture/GeneratedServices/ServiceTemplate/service.tpl" and replace the "#include <wiringpi.h>" with "#include <pigpio.h>". Save the changes.
7. Now, before you execute the step 5 in the GitHub Readme file, make the following changes:

a. Navigate to the following file "/AtlasThingMiddleware_RPI-master/lib/PahoMQTT/Log.h" and go to line number 43 and delete the text "Log_levels". After you do this, your code block starting from line 35 should look like shown below and save the changes:

```
enum LOG_LEVELS {
        TRACE_MAXIMUM = 1,
        TRACE_MEDIUM,
        TRACE_MINIMUM,
        TRACE_PROTOCOL,
        LOG_ERROR,
        LOG_SEVERE,
        LOG_FATAL,
};
```

i.

8. Note: Make sure to save the edited files before you execute any later commands.
9. Now, you can proceed with steps 5, 6 and 7 given in the GitHub Readme file.