Frank Wickström

Getting started with Smart-M3 using Python

Turku Centre *for* Computer Science

TUCS Technical Report
No 1071, March 2013

Getting started with Smart-M3 using Python

Frank Wickström
Åbo Akademi University, Department of Computer Engineering
Joukahainenkatu 3-5 , 20520 Turku, Finland
frwickst@abo.fi

TUCS Technical Report
No 1071, March 2013

Contents

Acronyms

KP Knowledge Processor

KPI Knowledge Processor Interface

SIB Semantic Information Broker

RDF Resource Description Framework

OWL Web Ontology Language

SPARQL **S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage

1 Introduction

This tutorial will guide you through setting up your own Smart-M3 network and creating a small Web Ontology Language (OWL) ontology using Protege. It will include how to install the Redland SIB graph database on an Ubuntu computer, as well as creating a small application using the Python KP. This is strictly a getting started tutorial for creating your first small application using Smart-M3 and the Redland SIB. If you want to learn more about how Smart-M3 came to be, or just want to know how its inner cogs turn, I suggest the following reading materials: [1–3].

It should be noted that at the time of writing the publicly available Python KP (version 0.9.6) does not support loading an entire ontology into the SIB, I will however be using a non-public beta of version 0.9.8 which will include this capability.

1.1 Items not covered by this tutorial

We will not be covering the use of reasoners in this tutorial, as the reasoner bundled with the Redland SIB does not seem to be reasoning the same way as the ontology modeling tool Protégés reasoners does. This makes it harder to predict what actually will happen when a reasoner is being used, and will just end up confusing the programmer at this point. The tutorial will also not explain how Smart-M3 is build, or how you can hack its source code, as this would defeat its purpose of being a getting started guide.

It should be noted that at the time of writing the publicly available Python KP (version 1.0.0) does support subscriptions to be made using SPARQL, we will however not cover these kind of subscriptions in this tutorial as the feature is still in an early stage.

2 What is Smart-M3?

Smart-M3 is an open-source software that aims to make the entities and devices smarter by connecting them together via the sharing of data. Smart-M3 strives to be multi-vendor, multi-device and multi-part compatible, therefore the M3 at the end of the name.

Smart-M3 uses Semantic Information Brokers (SIBs) to distribute the data between devices. A SIB sits on top of a Triplestore (known as a RDF-Store), and acts as handler in between the actual graph database and the Knowledge Processors (KPs), as can be seen in figure 1. Several SIBs can be connected to each other, creating a smart space [1]. The SIB is not tied to any specific triplestore, but as of writing, there are only two publicly available implementations in the Smart-M3 sourceforge repository [4]. In this tutorial we will be using the newest available

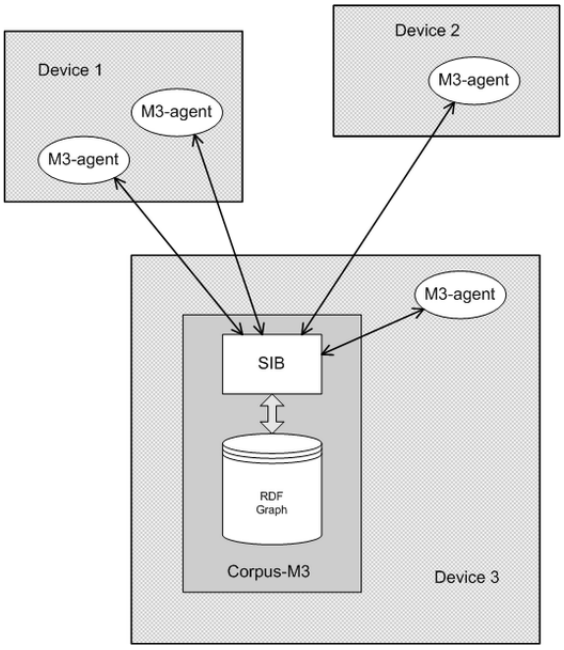


Figure 1: System Decomposition [6].

version of the Redland implementation, as this is the most up-to-date release of any Smart-M3 implementation [5].

What sets Smart-M3 aside from other graph storage solutions is its way to handle how the data flows in the network. Instead of constantly querying for new data on the Triplestore, devices can set up subscriptions to certain queries (Triples) and get a push like behavior from the system.

An example would be that a heater waits for a certain triple containing information about the current temperature in the room. When it gets that information and the temperature is outside a set range it will either turn on or off the heater. This of course requires both the heater and the temperature sensor to have access to the SIB in one way or another, but as stated before, there are no requirements on either make or model of neither the heater nor the sensor.

Knowledge processors are the actual workers or agents of the systems. They are the ones that actually do something with all of the information on the SIBs. It is in the KPs that one inserts, deletes or updates information on the SIB, it is also in the KPs that subscriptions are made. Usually there are three kinds of KP in a smart space; producers, aggregators and consumers. Producers create or update data depending on its inputs, aggregators react on the information inserted into the SIB, by updating some other parts of the SIB and consumers are then used for showing information that exists in the database. As in the example above, the temperature sensor is the producer, the heater is the aggregator, and a display showing the current temperature and information of the heater status could be the consumer.

It should be noted that SIBs and the KPs can run on the same machine, but usually the SIB runs on a computer faster and with more memory then the rest of the nodes in the network/smart-space. And the KPs can run on anything from a mobile phone to a temperature sensor as long as the device supports any of the languages that are supported by the KP interfaces (KPIs). As of right now, KPs can be written in any of the following languages C [7] , C# [8], Python [9], Java [10], JavaScript [11] and PHP [11].

Hopefully you now know a bit about what you can do with Smart-M3. If you are still wondering about what Smart-M3 is all about, I suggest you read the wikipedia article [6] and the introduction paper to Smart-M3 [1].

3 Installation

To create our own smart-space we first need to choose what SIB to use, and then download and install the appropriate software. In this tutorial we will be using a virtual machine, running Ubuntu 12.04 32-bit for installing our software on. The virtual machine runs in VirtualBox and no modifications have been made to Ubuntu after the installation except from running *sudo apt-get update & upgrade*.

Python & Ubuntu 12.10

Please note that from Ubuntu 12.10 onwards, the Python version used will be Python 3.0. You might run in to trouble using Ubuntu 12.10 if you have not installed Python 2.7 along side version 3.0

3.1 Choosing between SIBs

Before we get started with the installation, the different SIBs should be explained a bit. The one big difference between the Redland SIB and the original Nokia one, is that the Redland SIB is capable of receiving SPARQL queries. SPARQL is to a Triplestore what SQL is to a relational database, you can create complex queries with it which would be really tiresome to do with the Nokia SIB. For the Nokia SIB there are two ways of querying the Nokia SIB, either with RDF- or WQL queries. It should be noted that WQL support has been dropped in favor of SPARQL in the Redland SIB. This tutorial will only cover the usage of the Redland SIB and using only SPARQL as the querying language. If you like to know more about SPARQL please visit the SPARQL Wikipedia page [12].

3.2 Downloading, unpacking and installing

3.2.1 SIB

We start of by downloading the Redland SIB from the link below.

Smart-M3 Download

`http://sourceforge.net/projects/smart-m3/files/Smart-M3_B_v0.3.1-alpha`

Please note that we will be using v.0.3.12 in this tutorial and that this URL might have changed when you are reading this.

To download, extract and install the Redland SIB, follow these instructions.

```
1 mkdir smart-m3
2 cd smart-m3
3 wget http://sourceforge.net/projects/smart-m3/files/
     latest/
4 download?source=files
5 tar -zxvf Smart-M3_v0.3.12.tar.gz
6 cd Smart-M3_v.0.3.12
7 ./install.sh
```

3.2.2 Running the SIB

You now have the SIB installed. To start it we open two terminals and run the following commands in order:

```
1 # Terminal 1
2 redsibd
```

```
1 # Terminal 2
2 sib-tcp
```

Changing the port

By default, the *sib-tcp* command starts the server on the internal IP at port 10010. If you want to change the server's port, pass it as a command-line argument. For instance, this command starts the server on port 101010

sib-tcp -p 101010

If you want to run the server on another IP than the internal one, this command starts the server on the IP 192.168.1.54 and port 101010

sib-tcp 192.168.1.54 -p 101010

Now that we have our SIB up and running it is time to install our Python KPI.

3.2.3 KPI

We will be coding in Python in this tutorial, so we need to download the Python KPI from sourceforge.

Python KPI Download

`http://sourceforge.net/projects/smart-m3/files/Smart-M3_B_v0.3.1-alpha`

Please note that this is not a direct link to the KPI, as version 0.9.8 is not publicly available.

To download, extract and install the Python KPI, follow these instructions.

```
mkdir smart-m3 # If you have not created it before
cd smart-m3 # If you are not already in this folder
wget http://sourceforge.net/projects/smart-m3/files/
Smart-M3_B_v0.3.1-alpha/
smart-m3_pythonKP-0.9.6.tar.gz/download
tar -zxvf smart-m3_pythonKP-0.9.6.tar.gz
cd smart-m3_pythonKP-0.9.8.tar.gz
sudo python setup.py install
```

You should now have the Smart-M3 python KPI installed. You can check if the KPI was installed successfully by running *python* in a terminal and writing the following.

```
from smart_m3 import m3_kp # No errors = good
```

Multi-platform KPI

As the KPI does not have any dependencies to programs other than Python, and no 3rd-party Python packages needs to be installed, the KPI can run on any platform that supports Python 2.6 or higher.

3.2.4 Protégé

Protégé is a free open-source ontology creator, which will make our lives much easier when it comes to creating large ontologies. We will however not be using Protégés reasoners in this tutorial for the reason stated in Section 1.1. Protégé will serve mainly as a tool to get a better overview of the ontology and to understand how classes and entities connect. We will be using the beta of version 4.2, as it includes the feature to test SPARQL queries directly on the ontology, without first inserting them into the SIB.

Protégé Download

`http://protege.stanford.edu/download/protege/4.2/installanywhere/Web_Installers/`

To download, extract and install the Python KPI, follow these instructions:

```
mkdir smart-m3 # If you have not created it before
cd smart-m3 # If you are not already in this folder
wget http://protege.stanford.edu/download/protege/4.2/
```

```
4 installanywhere/Web_Installers/InstData/Linux/VM/
5 install_protege_4.2_beta.bin
6 sh install_protege_4.2_beta.bin
```

The installer will now start and you will be asked where to install Protégé, and whether you want to install the bundled Java VM or not. If you are unsure where to install the program, select the default path choice presented. If you are using a fresh installation of Ubuntu you will need to have Java installed for Protégé to work. Luckily Protégé comes bundled with Java so you can select the *"Use the Java VM installed with this application"* option when the installer asks you for which Java VM you want to use.

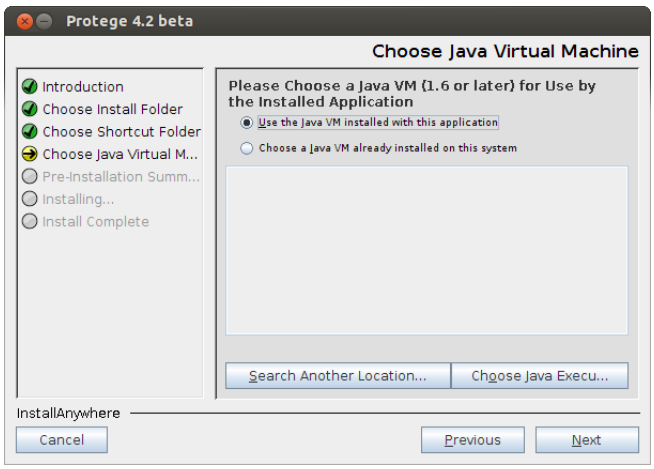


Figure 2: Choose the bundles Java VM if you don't have one installed

4 Creating your first Smart-M3 program

Now that you have both the SIB and the Python KPI installed it will time to create your first program. We will first look at the code, and then we will go through it line-by-line. We start by creating a simple example and then building upon that until we have a finished program.

The program we are creating is a small home automation example starting with just a presence sensor and a lamp, and later extending the system to include a light sensor. This example can easily be extended even further by adding other types of sensors, or even combining the home automation program with other systems such as location based communities such as Foursquare or why not add an event to your Google calendar saying that you need to buy a new sensor when one is faulty.

4.1 Splitting the work

As were noted before, a Smart-M3 program usually consists of several parts, most commonly a producer, an aggregator and a consumer. In our case we will start of by creating simple program which populates the SIB with some data. This is to demonstrate that it is easy to create ontologies in the code if the program is small. Later on, we will explore the usage of Protégé to create a bigger, more complex ontology. After that we move on to creating the main parts of the program the producer, aggregator and the consumer.

4.2 Joining a smart-space

Before we can start pushing data to the SIB we need to join the smart-space that the SIB is located in. We do this with the following code:

```
1  from smart_m3.m3_kp import *
2  import uuid
3
4  class FirstKP(KP):
5      def __init__(self, server_ip, server_port):
6          KP.__init__(self, str(uuid.uuid4())+"_FirstKP")
7          self.ss_handle = ("X",
8                  (TCPConnector,
9                  (server_ip,server_port)
10                 ))
11
12     def join_sib(self):
13         self.join(self.ss_handle)
14
15     def leave_sib(self):
16         self.leave(self.ss_handle)
17
18 pd = FirstKP("127.0.0.1", 10010)
19 pd.join_sib()
20 pd.leave_sib()
```

Lets go through the code and see what happens here.

```
1  from smart_m3.m3_kp import *
2  import uuid
```

These are our imports, we start of by importing the smart-m3 Python KPI, this is something we always need when we are going to communicate with the SIB. On line 2 we import the package *uuid*. This package makes it easy for us to create random alphanumeric string, uuid stands for **U**niq**u**e **ID**. If you want to learn more about the uuid package, I suggest you check out the Python documentation [13].

```
4  class FirstKP(KP):
```

We start of by creating a class that extends the KP class found in the *smart_m3.m3_kp* package. Th KP class includes functions for communicating with the SIB. By extending it we get all these capabilities without the need to explicitly write them.

```
5     def __init__(self, server_ip, server_port):
6         KP.__init__(self, str(uuid.uuid4())+"_FirstKP")
```

Here we initialize our own class with the parameters *server_ip* and *server_port*, as well as call the KPs *__init__* function. The KPs init function takes the *self* variable as well as the name of the node we are creating. The name can be anything we want, it does not even matter if we create a node with the same name later on, the SIB will have an internal name for our node. This name is mainly for our own good, so that we can distinguish between nodes if we need to.

```
7         self.ss_handle = ("X",
8                 (TCPConnector,
9                 (server_ip,server_port)
10                ))
```

Here comes the first real Smart-M3 part, we now create a smart-space handle tuple, *self.ss_handle*, which includes the information needed to connect to the SIB. First we state the name of the smart-space to join, in this case ”X”, then we create another tuple containing a TCPConnector object, as well as the server ip and server port of the SIB. The TCPConnector class is found in the *smart_m3.m3_kp* package and it handles all the connections between the KP and the SIB, making our job much easier.

```
12    def join_sib(self):
13        self.join(self.ss_handle)
```

As we might not want to connect to the SIB at the point we create our KP, we write the method *join_sib* so create the connection. On line 13 we join the SIB by using the *join* function which takes our smart-space handle as an argument.

```
15    def leave_sib(self):
16        self.leave(self.ss_handle)
```

Just as with the join method we just wrote, we also need to have a way to leave the SIB. So instead of using the *join* method we now call the *leave* method.

```
18 pd = FirstKP("127.0.0.1", 10010)
19 pd.join_sib()
20 pd.leave_sib()
```

Now it is time to test what we have created. We start of by creating a *PushData* object with the server IP as a string, and the server port as an integer. The default port of the Redland SIB is 10010, but this can be changed to anything you like, just remember to start the server on a different port if you don’t want to use the default one. On line 16 we then join the SIB with our own *join_sib* method. And

as we don't have anything to say to the SIB yet, we leave, closing all connection to the SIB on line 17.

We have now successfully connected to, and then disconnected from the SIB. This is the basic blocks that are needed for every program that is going to communicate with the SIB.

4.3 Creating the ontology

We begin by creating the ontology we want in Python code, and then do the same in Protégé. The ontology will consist of a motion/presence sensor and a lamp class. The sensor will have a *movement* data property and the lamp will have a *state* data property. They will both belong to a superclass called Device. And as this is an OWL ontology, every class that is created will have a superclass called *Thing*.

4.3.1 Ontology in Python

For inserting our ontology into the SIB we will need to connect to the SIB first. As we have already done that in the previous step, we can use our existing code and build upon that. We will first look at the complete code, and then go through it line-by-line.

```
1  from smart_m3.m3_kp import *
2  from smart_m3.RDFTransactionList import *
3  import uuid
4
5  NS = "http://sm3-tut/Ontology.owl#"
6
7  class PushKP(KP):
8      def __init__(self, server_ip, server_port):
9          KP.__init__(self, str(uuid.uuid4())+"_PushKP")
10         self.ss_handle = ("X",
11                 (TCPConnector,
12                 (server_ip,server_port)
13                 ))
14
15     def join_sib(self):
16         self.join(self.ss_handle)
17
18     def leave_sib(self):
19         self.leave(self.ss_handle)
20
21     def createOntologyStructure(self):
22         t = RDFTransactionList()
23
```

```
24          t.add_Class(NS + "Thing")
25          t.add_subClass(NS + "Device", NS +"Thing")
26          t.add_subClass(NS + "MotionSensor",NS + "Device"
                ,)
27          t.add_subClass(NS + "Lamp", NS + "Device",)
28
29          i1 = self.CreateInsertTransaction(self.ss_handle)
30          i1.send( t.get() )
31
32          self.CloseInsertTransaction(i1)
33
34      def createInstances(self):
35          t = RDFTransactionList()
36
37          ms1 = NS + "MotionSensor_living_room_1"
38          t.setType(ms1, NS + "MotionSensor")
39          t.add_literal(ms1, NS + "hasMovement", "false")
40
41          l1 = NS + "Lamp_living_room_1"
42          t.setType(l1, NS + "Lamp")
43          t.add_literal(l1, NS + "hasState", "false")
44
45          i1 = self.CreateInsertTransaction(self.ss_handle)
46          i1.send( t.get() )
47
48          self.CloseInsertTransaction(i1)
49
50  pd = PushKP("192.168.56.101", 10010)
51  pd.join_sib()
52  pd.createOntologyStructure()
53  pd.createInstances()
54  pd.leave_sib()
```

Remember that you need to have the SIB running for this code to work. So if you get an error about not being able to connect to the SIB, you will need to start it. Let us go through what the code actually does.

```
2  from smart_m3.RDFTransactionList import *
```

To make our work a bit easier, we use the helper functions given by the package *smart_m3.RDFTransactionList*.

```
5  NS = "http://sm3-tut/Ontology.owl#"
```

This constant is the name space we are going to use for your ontology. You can change *sm3-tut* to anything you like, but I recommend you keep it the same during this tutorial. The name space exists so we distinguish between ontologies if we have more than one.

```
21      def createOntologyStructure(self):
```

We will now push the ontology in two parts to the SIB. We start of by creating our ontology structure in the function *createOntologyStructure*.

```
22          t = RDFTransactionList()
```

RDFTransactionList, will keep a list of all the triples we want to insert into the SIB. By instantiating this class we can add several triples to the list and then send them all at once, putting less load on the SIB.

```
24          t.add_Class(NS + "Thing")
25          t.add_subClass(NS + "Device", NS +"Thing")
26          t.add_subClass(NS + "MotionSensor",NS + "Device"
               ,)
27          t.add_subClass(NS + "Lamp", NS + "Device",)
```

Now it is time to create our structure. We first create the main class called *Thing*, everything will then be a subclass of this. Note that we are now using functions from the *RDFTransactionList* package to create the triples. If we were not using these functions we would need to manually create the triples. For instance, the function call *t.add_Class(NS + "Thing")* creates an triple like the one below.

```
Triple(URI(NS+"Thing"),URI("rdf:type"),URI("rdfs:Class"))
```

As we might want to add other devices late on, we also create a class called *Device* and then we create the subclasses *MotionSensor* and *Lamp* under that. Note that we append the name space, *NS*, to the beginning of the class. Lastly we create our *LogEntry* class, to keep things as simple as possible we just make it as a subclass to *Thing*. To better explain how everything is connected have a look at Figure 3.

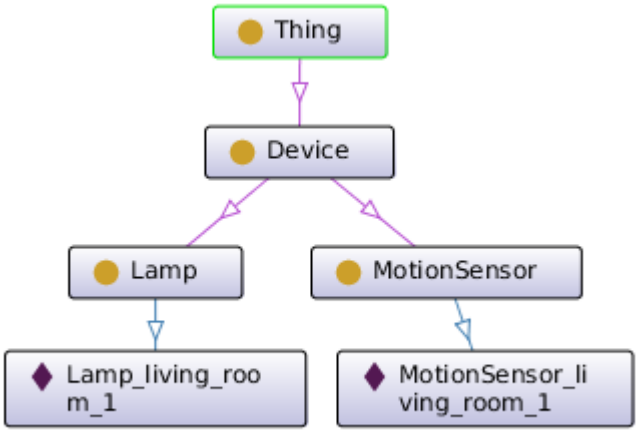


Figure 3: The structure of our ontology.

```
29          i1 = self.CreateInsertTransaction(self.ss_handle)
30          i1.send( t.get() )
31
32          self.CloseInsertTransaction(i1)
```

Our list of triples are now ready to be inserted into the SIB. We start of by creating a insert transaction on line 29, passing it our smart-space handler. This function sits in *smart_m3.m3_kp* and needs to be called before inserting anything into the SIB.

On line 30 we then send our triples to the SIB. Note here that we call *t.get()* to get the list from the *RDFTransactionList* object we created earlier.

We end by closing the transaction to the SIB by calling the *CloseInsertTransaction* function with our insert transaction as an argument.

```
34      def createInstances(self):
35          t = RDFTransactionList()
```

Now that we have our structure ready, it is time to insert some instances into the SIB. For this we create a new function called *createInstance*. And once again we create an transaction list to help to keep track of our triples.

```
37          ms1 = NS + "MotionSensor_living_room_1"
```

First we give our instance its own unique name by appending a string to class name. We do not have to use exactly the same name as the class we created in the last function, but it makes it easier to keep track of everything. Note the *NS* being appended to the beginning of the string!

Unique instance names

As when initializing the KP, we could use *uuid.uuid4()* and let it create our unique string for us. This comes in handy when creating a lot of instances of the same class. Here we just use a ”readable string” to make it a more concrete example.

```
38          t.setType(ms1, NS + "MotionSensor")
```

We also set our instance to be of type *MotionSensor*. Here it is important that you use the same name as you did when creating the class, otherwise you will have a hard time querying for this instance later on.

```
39          t.add_literal(ms1, NS + "hasMovement", "false")
```

Lastly we add a literal to our instance. Literals acts like variables for instances, they can be anything from strings to booleans. This is where we store our sensor state. Note however that literals does not start with *NS*, as that would make them associated with other classes, more on that later in this tutorial.

Naming convetions

A good naming convention to fallow is to start your ”keywords” with *has*, for instance *hasMovement* in the example above. This way you can remember it easier.

```
41          l1 = NS + "Lamp_living_room_1"
42          t.setType(l1, NS + "Lamp")
43          t.add_literal(l1, NS + "hasState", "false")
```

We now do the same thing with the lamp as we did with the motion sensor. Create an instance with a unique id, make it of type *Lamp* and give it a value, *HasState*, or False.

```
44          i1 = self.CreateInsertTransaction(self.ss_handle)
45          i1.send( t.get() )
```

Just like when we created the structure, we now insert everything into the SIB and by creating and insert transaction, sending the data to the SIB and then closing the transaction.

```
52 pd.createOntologyStructure()
53 pd.createInstances()
```

We also add two calls to the newly created functions.

SIB Overview

If you want a good overview over what is stored in the SIB, you can use the *Web SIB Explorer* found at `http://eslab.github.com/Web-SIB-Explorer/`.

We have now created our entire ontology in Python. If you run the code it will insert a total of about 15 triples into the SIB, which we will be used later in this tutorial.

We will now create the same ontology in Protégé to show how we can use its user interface to easily create the same ontology as in the Python code.

4.3.2 Ontology in Protégé

When creating large ontologies it can be easy to make a mistake, or it can just be tiresome to write the code for every class and property. Ontologies usually include several hundreds, if not thousands of triples, which would make the ontology really hard to maintain if we did not have proper tools to handle everything. This is where Protégé comes in. It is a fairly easy to use tool for creating ontologies in a graphic user interface.

For this tutorial we will be using the version 4.2 beta, running on Ubuntu 12.04. And as the version says, this is not a fully stable release, but is has some nice features that are not available in older version.

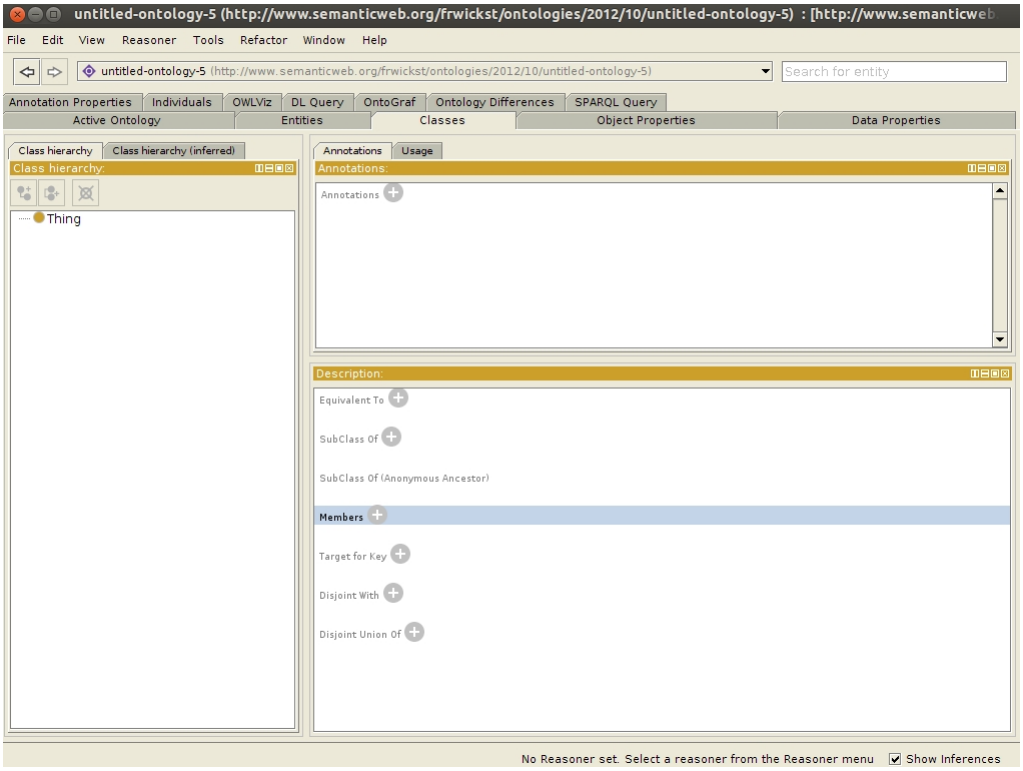


Figure 4: Protege 4.2 Beta

To start Protégé double click *Protege* in the folder where you installed it and then select *Run*. If you like to use the terminal instead, navigate to the folder where you installed Protégé, and run `.\Protege`. Running the program in the terminal will also give you its debug output, so if you run into problems I suggest starting it form a terminal. When the program has started you should see a window resembling figure 4.

If you don't see the exact same thing as in the figure, then click on the *Classes* tab. This will give an overview of your classes.

To start of we need to create our class *Device* and its subclasses *MotionSensor* and *Lamp*. We do this by first selecting the *Thing* class and then clicking the *Add subclass* button  . You will now be asked for a class name, type *Device*. To create the subclasses make sure the *Device* class is selected and then we click *Add subclass* to add *MotionSensor* and *Lamp*. When you are done your class list should look like Figure 5.

Now that we have created our classes we will create the object- and data properties for these classes. This is almost the same thing as we did in the Python examples *createIndividuals* function with the exception that we did not explicitly define any properties, and instead just added literals, which had the properties in them.

In our case we do not have any object properties, so we will skip that step

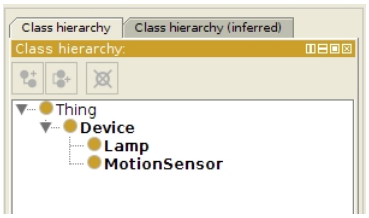


Figure 5: Your class list should look like this

and go straight to the data properties. Start off by clicking the *Data Properties* tab. Here we are going to create or data properties *hasMotion* and *hasState*. First select the *topDataProperty* property, then click the *Add sub property* button  . Name the first property *hasMotion*, then create another property with the name *hasState*. You property list should now look like figure 6.

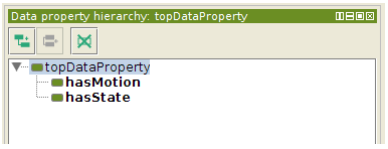


Figure 6: Your data properties list should look like this

We also need to tell Protégé which classes these properties belongs to. Start of by selecting the *hasMotion* property. Then in the lower right corner, in the *Description* section, click the plus sign next to *Domain (intersection)*. You will be presented with a list of the classes that we created earlier. Select *MotionSensor* and click ok. Your description section should now look like Figure 7. Now do the same thing with *hasState* and select *Lamp* as the domain.

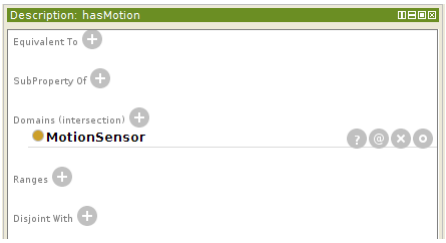


Figure 7: Add a domain to your property to associate it with a class

Now we have to create our actual motion sensor and lamp, click the *Individuals* tab. To create a new individual press the *Add individual* button  . Give it the name *MotionSensor_living_room_1*, then create another individual with the name *Lamp_living_room_1*. Your list of individuals should now look like figure 8.

We now give the individuals a type by, clicking the plus sign next to *Types* in the *Description* section. Like in the previous cases you will be presented with a

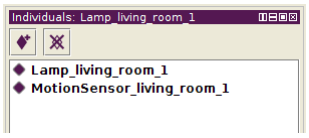


Figure 8: Your individuals list should look like this

view of our classes, for the lamp select *Lamp* as class and *MotionSensor* for the motion sensor individual.

Lastly we give the individuals data properties. First select the individual we want and then click the plus sign next to *Data property assertions* in the *Properties assertions* section. We now see a list of the data properties we have created earlier. For the motions sensor, select *hasMovement* and as the data property and boolean as the type. Now select the lamp and add the data property *hasState* with the type boolean.

Our ontology is now ready to be loaded into the SIB, this can be done by running the following Python script.

```
1  from smart_m3.m3_kp import *
2
3  node = KP("Insert_ontology_node")
4  ss_handle = ("X",
5               (TCPConnector,
6               ('127.0.0.1',10010)
7               ))
8  node.join(ss_handle)
9
10 ins = node.CreateInsertTransaction(ss_handle)
11 ins.send('tutorial_ontology.owl', encoding = "RDF-XML")
12 node.CloseInsertTransaction(ins)
13 node.leave(ss_handle)
```

As you can see, this is basically the same thing as we have done in our previous examples the only new thing is on line 12.

```
11 ins = node.CreateInsertTransaction(ss_handle)
12 ins.send('tutorial_ontology.owl', encoding = "RDF-XML")
```

We first create a normal insert transaction on line 11, and on line 12, instead of sending triples to the SIB, we send an entire file as the first argument. As the second argument we set the encoding to *"RDF-XML"* as that is the only supported format by the Redland KPI at this point.

You have now created the same ontology in both Python and Protégé. It the next section we will go through basic querying and how to access our data in the SIB.

4.4 Transactions and Queries

To get some real use out of the SIB we need to be able to add, remove, update and list the data in it. This latter is done by querying the SIB and adding, removing and updating data is done via transactions. You have actually already done some transactions while creating the ontology in Python, we just need to learn a bit more about them to make them really useful.

4.4.1 Creating the producer

While SPARQL is able to create, insert, delete and update queries we will be using SPARQL mainly for querying and listing data in this tutorial, and use the Python KPIs RDF transactions to actually manipulate the data on the SIB. This is mainly to show the different ways of doing things with the SIB. And as you might need to work with a SIB that does not have SPARQL capabilities, this might be some good training. Later on we will also notice that we will need to know a bit about RDF-triples when we create our first transaction.

We will now create the producer of our program, as you might remember from section 4.1, this is the part of the program that creates data for the rest of the network to work with. In this tutorial, the producer will be the motion sensor and as you might not have an actual motion sensor close by, we will simulate one in this tutorial.

We start of with the code we created in section 4.2 when we learned how to join the smart-space, and build upon that. We will quickly go through the same code one more time, just to refresh your memory. Like in the previous examples, we will first take a look at the entire code, following it up by a line-by-line explanation.

```
1  from smart_m3.m3_kp import *
2  import uuid
3
4  NS = "http://sm3-tut/Ontology.owl#"
5
6  class ProducerKP(KP):
7      def __init__(self, server_ip, server_port):
8          KP.__init__(self, str(uuid.uuid4())+"_ProducerKP"
               )
9          self.ss_handle = ("X",
10                           (TCPConnector,
11                           (server_ip,server_port)
12                           ))
13
14     def join_sib(self):
15         self.join(self.ss_handle)
16
```

```
17      def leave_sib(self):
18          self.leave(self.ss_handle)
19
20      def insert(self, triples):
21          ins = self.CreateInsertTransaction(self.ss_handle
                )
22          ins.send(triples)
23          self.CloseInsertTransaction(ins)
24
25      def remove(self, triples):
26          rem = self.CreateRemoveTransaction(self.ss_handle
                )
27          rem.remove(triples)
28          self.CloseRemoveTransaction(rem)
29
30      def update(self, i_trip, r_trip):
31          # Update = Insert + Remove
32          upd = self.CreateUpdateTransaction(self.ss_handle
                )
33          upd.update(i_trip, "RDF-M3", r_trip, "RDF-M3")
34          self.CloseUpdateTransaction(upd)
35
36
37  pd = ProducerKP("192.168.56.101", 10010)
38  pd.join_sib()
39
40  while True:
41      i = raw_input('\nSet motion sensor state'\
42      '\n0 : No Movement'\
43      '\n1 : Movement'\
44      '\nexit : Exit program\n')
45
46      if i == "1":
47          print "Setting state to True"
48
49          ins_trip = [Triple(
50                      URI(NS+"MotionSensor_living_room_1"),
51                      URI(NS+"hasMovement"),
52                      Literal("true"))]
53
54          rem_trip = [Triple(
55                      URI(NS+"MotionSensor_living_room_1"),
56                      URI(NS+"hasMovement"),
57                      Literal("false"))]
58
59          pd.update(ins_trip, rem_trip)
```

```
60
61
62      elif i == "0":
63          print "Setting state to False"
64
65          ins_trip = [Triple(
66                      URI(NS+"MotionSensor_living_room_1"),
67                      URI(NS+"hasMovement"),
68                      Literal("false"))]
69
70          rem_trip = [Triple(
71                      URI(NS+"MotionSensor_living_room_1"),
72                      URI(NS+"hasMovement"),
73                      Literal("true"))]
74
75          pd.update(ins_trip, rem_trip)
76
77
78      elif i.lower() == "exit":
79          break
80
81  pd.leave_sib()
```

```
1  from smart_m3.m3_kp import *
2  import uuid
```

Here we import the proper packaged to communicate with the SIB and to create unique strings.

```
4  NS = "http://sm3-tut/Ontology.owl#"
```

We will need to use our name space in this program, so we create a constant to help us remember.

```
6  class ProducerKP(KP):
7      def __init__(self, server_ip, server_port):
8          KP.__init__(self, str(uuid.uuid4())+"_ProducerKP"
               )
9          self.ss_handle = ("X",
10                     (TCPConnector,
11                     (server_ip,server_port)
12                     ))
```

We change the name of the KP from the one we created earlier to *ProducerKP*, and also change the *KP.__init__* accordingly. After that we create our handler with the arguments we got from the *__init__* function.

```
14     def join_sib(self):
15         self.join(self.ss_handle)
```

```
16
17      def leave_sib(self):
18          self.leave(self.ss_handle)
```

Next we create to helper function for joining and leaving the SIB.

```
20      def insert(self, triples):
21          ins = self.CreateInsertTransaction(self.ss_handle
                )
22          ins.send(triples)
23          self.CloseInsertTransaction(ins)
24
25      def remove(self, triples):
26          rem = self.CreateRemoveTransaction(self.ss_handle
                )
27          rem.remove(triples)
28          self.CloseRemoveTransaction(rem)
```

On lines 20 to 28 we also create two helper functions for inserting and removing data from the SIB. You will notice later on that we won't actually be using these in the code, but they are good functions to have around if we want to extend the code at later point.

```
30      def update(self, i_trip, r_trip):
31          # Update = Insert + Remove
32          upd = self.CreateUpdateTransaction(self.ss_handle
                )
33          upd.update(i_trip, "RDF-M3", r_trip, "RDF-M3")
34          self.CloseUpdateTransaction(upd)
```

Here is the first new function we create, like the previous two, this is also a helper function. This time for updating triples on the SIB. As there is no way of actually modifying existing data on the SIB at this point, an update actually consist of a insert, and a remove operation. When we send the update to the SIB we also have to specify the encoding to use, at the time of writing, the only supported encoding is *RDF-M3*.

```
37 pd = ProducerKP("192.168.56.101", 10010)
38 pd.join_sib()
```

We now specify the IP and port to use and then we join the SIB.

```
40 while True:
```

As we are going to take user input in this example we create a while loop that is constantly *True*. We will at a later point break out of this loop if the user inputs *"exit"* into the terminal.

```
41      i = raw_input('\nSet motion sensor state'\
42      '\n0 : No Movement'\
43      '\n1 : Movement'\
```

```
44      '\nexit : Exit program\n')
```

Here we take a user input and show a user guide for using the program. *"0"* means no movement, *1* means movement. To exit the program, type *"exit"*.

```
46      if i == "1":
47          print "Setting state to True"
```

We create an if statement saying that *"if the input was 1, then run this*. It also helps to have some output to let the user know what is happening, so we create a print statement on line 47 telling the user that the state of the sensor is about to change.

```
49          ins_trip = [Triple(
50                      URI(NS+"MotionSensor_living_room_1"),
51                      URI(NS+"hasMovement"),
52                      Literal("true"))]
```

Now it is time to actually create something to put on the SIB. We first create our insert triple, which consist of the object we want to modify, the attribute we want to change, and lastly we write what value that attribute should have, in this case *"true"*. Note here that we need to create this as a list, this is because we can actually have several triples in the same insert instruction.

```
51          rem_trip = [Triple(
52                      URI(NS+"MotionSensor_living_room_1"),
53                      URI(NS+"hasMovement"),
54                      Literal("false"))]
```

We also need to create a remove triple, for removing the previous value that the attribute had. This triple looks almost identical to the one before, but we change to literal in the end to *"false"*.

```
59          pd.update(ins_trip, rem_trip)
```

Now we send the triples on their way to the SIB with the update function we wrote on lines 30-34.

```
62      elif i == "0":
63          print "Setting state to False"
64
65          ins_trip = [Triple(
66                      URI(NS+"MotionSensor_living_room_1"),
67                      URI(NS+"hasMovement"),
68                      Literal("false"))]
69
70          rem_trip = [Triple(
71                      URI(NS+"MotionSensor_living_room_1"),
72                      URI(NS+"hasMovement"),
73                      Literal("true"))]
74
```

```
75          pd.update(ins_trip, rem_trip)
```

We also create the inverse case, where we want to set the state to false, instead of true.

```
79      elif i.lower() == "exit":
80          break
```

As we are in a infinite while loop, we also create a way for the user to exit that loop. By typing the work *"exit"* the user will break and exit the loop.

```
81  pd.leave_sib()
```

Lastly we close the connection to the SIB and exit the program.

We have now created our producer, giving us the possibility to update the data of the motion sensor. To test the program, save it as *producer.py* to a place you remember, and run *python producer.py*. Your program should look similar to figure 9. When you have tested your program we can move on to creating a way to control the lamp, depending on our motion sensor.

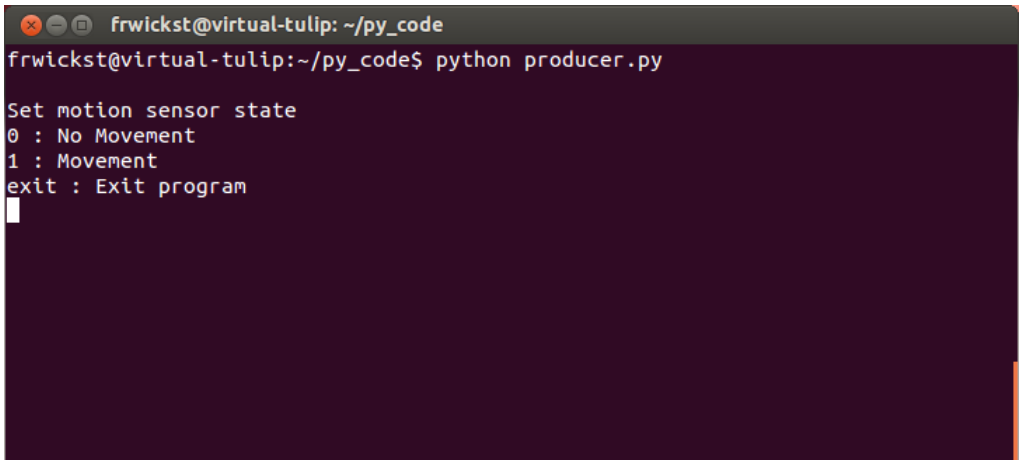


Figure 9: The producer should look similar to this when running in a terminal.

4.4.2 Creating the aggregator

The new feature that Smart-M3 and the SIB brings to the table is subscriptions. The basic idea is simple, when the database is updated with a specific value, then the SIB notifies us of the update. You can think of it as push notifications for your mobile phone, when you get an email, the phone gets a notification that a new email as arrived.

We will now create our aggregator which will benefit from these subscriptions. This part of the program will look for new data inputs into the SIB and act on it. In our case, we will turn on the light, if the there is movement detected by the motion sensor. This can easily be done with subscriptions, which is described below.

```
from smart_m3.m3_kp import *
import uuid

NS = "http://sm3-tut/Ontology.owl#"

class MotionSensorHandler:
    def __init__(self, node):
        self.node = node

    def handle(self, added, removed):
        lamp_on = [Triple(
                        URI(NS+"Lamp_living_room_1"),
                        URI(NS+"hasState"),
                        Literal("true"))]
        lamp_off = [Triple(
                        URI(NS+"Lamp_living_room_1"),
                        URI(NS+"hasState"),
                        Literal("false"))]

        for trip in added:
            if str(trip[2]) == "true":
                #update(TRIPLE_TO_INSERT,TRIPLE_TO_REMOVE
                   )
                self.node.update(lamp_on, lamp_ff)

        for trip in removed:
            if str(trip[2] == "true"):
                self.node.update(lamp_off, lamp_on)

class AggregatorKP(KP):
    def __init__(self, server_ip, server_port):
        KP.__init__(self, str(uuid.uuid4())+"_Aggregator"
           )
        self.ss_handle = ("X",
                (TCPConnector,
                (server_ip,server_port)
                ))

    def join_sib(self):
        self.join(self.ss_handle)

    def leave_sib(self):
        self.CloseSubscribeTransaction(self.st)
        self.leave(self.ss_handle)

    def update(self, i_trip, r_trip):
```

```
45          upd = self.CreateUpdateTransaction(self.ss_handle
                )
46          upd.update(i_trip, "RDF-M3", r_trip, "RDF-M3")
47
48          self.CloseUpdateTransaction(upd)
49
50      def create_subscription(self):
51          trip = [Triple(
52                      URI(NS+"MotionSensor_living_room_1"),
53                      URI(NS+"hasMovement"),
54                      None)]
55          self.st = self.CreateSubscribeTransaction(
56                                              self.ss_handle)
57          initial_results = self.st.subscribe_rdf(trip,
58                                  MotionSensorHandler(self)
                                        )
59          print initial_results
60
61
62  pd = AggregatorKP("192.168.56.101", 10010)
63  pd.join_sib()
64  pd.create_subscription()
65
66  while True:
67      i = raw_input('\nType "exit" to exit the program\n')
68
69      if i.lower() == "exit":
70          break
71
72  pd.leave_sib()
```

We start of with our normal imports as well as defining our name space.

```
1  from smart_m3.m3_kp import *
2  import uuid
3
4  NS = "http://sm3-tut/Ontology.owl#"
```

We now create our subscription handler. The handler will take care of what will happen when a certain subscription gets a hit.

```
6  class MotionSensorHandler:
```

To make calls to the KP that has created an instance of the handler, we create an *init* function that takes a node as an argument and saves that to *self.node* for later usage.

```
7      def __init__(self, node):
8          self.node = node
```

Every handler needs to have a function called *handle*, and it should take two arguments, one for added triples, and one for removed ones. The *added* and *removed* variables are not single triples, but lists of triples. We will need to know this when handling these variables later.

```
10      def handle(self, added, removed):
```

In our aggregator we will need to do two actions, depending on the triple we get from the subscription. We will either turn the lamp on, or off. For this we can use the same triples we used in our producer in section 4.4.1. We set these two triples as variables for now, as we will need to use them more than once.

```
18          lamp_on = [Triple(
19                              URI(NS+"Lamp_living_room_1"),
20                              URI(NS+"hasState"),
21                              Literal("true"))]
22          lamp_off = [Triple(
23                              URI(NS+"Lamp_living_room_1"),
24                              URI(NS+"hasState"),
25                              Literal("false"))]
```

As noted before, the added and removed arguments are lists passed to the handle function. We need to loop over these lists to get the triples returned by the SIB.

```
20          for trip in added:
```

For the triples added, we want to look for when the light has been turned on, in other words we need to look at the literal part of the triple, which is the third part of the triple. If that part is true, meaning that the motion sensor has seen movement, we turn on the lamp.

```
21              if str(trip[2]) == "true":
```

Next we want to update the actual values on the SIB. For this we call the nodes update function. The first value given to the update function is the triple we want to insert, and the second one is the triple we want to remove. In this case, we want to remove the *false* value of the lamp, and set it to *true*.

```
23                  self.node.update(lamp_on, lamp_ff)
```

We now do the same thing when we get removed triples. If we get a removed triple with the literal value *true* that means that we no longer have movement and the lamp should be turned off.

```
25          for trip in removed:
26              if str(trip[2] == "true"):
27                  self.node.update(lamp_off, lamp_on)
```

We are now ready to move over to our actual KP. We start of by creating a new class, *AggregatorKP*, which extends the *KP* class.

```
29 class AggregatorKP(KP):
```

We create our normal *__init__* function, where we take our server ip and port as arguments, and create our smart-space handle. If you are copy and pasting from before, just remember to change the KPs name to *_Aggregator*, so you have a easier time recognizing your KPs.

```
30     def __init__(self, server_ip, server_port):
31         KP.__init__(self, str(uuid.uuid4())+"_Aggregator"
               )
32         self.ss_handle = ("X",
33                 (TCPConnector,
34                 (server_ip,server_port)
35                 ))
```

The join, leave and update functions should also look familiar by now, we do however have a new call on line 41 when leaving the SIB. Here we call the *CloseSubscribeTransaction* function, which closes an existing subscription. The subscription here, *self.st*, will create the *create_subscription* function.

```
40     def leave_sib(self):
41         self.CloseSubscribeTransaction(self.st)
```

Next we create a new function, for creating our subscriptions.

```
50     def create_subscription(self):
```

We then write the triples we want our subscription to match. We do this by creating a triple and inserting *None* as a wildcard at the place of the literal. What we mean by this is that *None* can match anything, in our case *true* or *false*.

```
51         trip = [Triple(
52                     URI(NS+"MotionSensor_living_room_1"),
53                     URI(NS+"hasMovement"),
54                     None)]
```

To set up our subscription we first create a subscription transaction by calling, *CreateSubscriptionTransaction*. This function takes only our smart-space handle as an argument and return a *Subscribe* class.

```
55         self.st = self.CreateSubscribeTransaction(
56                                             self.ss_handle)
```

Now for creating the actual subscription, by calling the returned subscribe classes *subscribe_rdf* function from the last step. We pass it the triple we want to match, and the handler we created earlier. We also pass the current instance, *self*, to the handler, so that we can call our instances function from the handler. The subscribe_rdf function returns the current triples that matches our subscription, so we store these as a variable.

```
57         initial_results = self.st.subscribe_rdf(trip,
```

```
58                                      MotionSensorHandler(self)
                                           )
```

On line 59 we print the initial triples matched by the subscription, so that we can see what the current state of the motion sensor is.

```
59          print initial_results
```

As we want the program to start when we run our python file, we also create a aggregator instance, join the SIB and start our subscription.

```
62 pd = AggregatorKP("192.168.56.101", 10010)
63 pd.join_sib()
64 pd.create_subscription()
```

We want the program to run until the user actually want to quit, so we create an infinite loop, that only exists when the user types ”exit”.

```
66 while True:
67     i = raw_input('\nType "exit" to exit the program\n')
68
69     if i.lower() == "exit":
70         break
```

Before quitting the program we want it to close the connection to the SIB as well as our subscription transaction, so we call the *leave_sib* function last.

```
72 pd.leave_sib()
```

While not doing much at this time, we can try our program by first saving it as *aggregator.py*, and then running *python aggregator.py* in a terminal. You should see something similar as in figure 10.

Figure 10: The aggregator should look similar to this when running in a terminal.

4.5 Making SPARQL queries

As we are using the Redland SIB in this tutorial, all queries to fetch data will be written in SPARQL, as that is the main advantage of using the Redland version over the Nokia one. There are currently two versions of SPARQL avaliable, version 1.0 and a draft version of 1.1. We will be using 1.0 as that is the only fully supported version by the Redland SIB. If you, after this tutorial, search for help regarding SPARQL queries, please remember that some of the solutions given might be for version 1.1, and might not work with the current SIB.

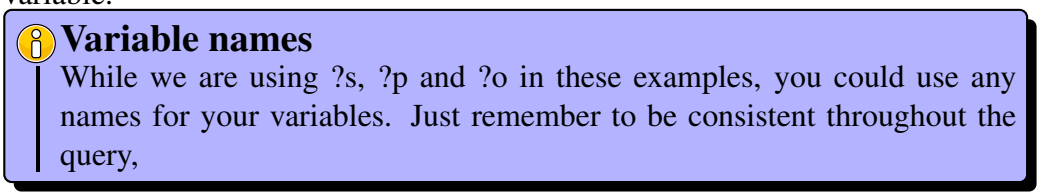
To get a better sense of how queries in SPARQL work, we will start of with a simple example of querying everything in the triple-store. This will also give you a better understanding of why it is called a triple store.

```
1 SELECT ?s ?p ?o
2 WHERE {
3     ?s ?p ?o
4 }
```

Like in SQL, the keywords *SELECT* and *WHERE* are used to create basic queries in SPARQL. But there are however some specific differences from SQL that we need to go through before we start making queries. The first thing you will notice is the lack of the keyword *FROM*. *FROM* exists but is not used as much as in SQL, as there are usually no need to specify the database we are working on. Lets go through the code line by line to get a better sense of what we have done.

```
1 SELECT ?s ?p ?o
```

As everything stored in a triple-store consist of triples, it makes sense that queries are also done on triples. All the triples consists of a subject, a predicate and an object, so what we are saying on line 1 is "**SELECT** every subject **?s** and predicate **?p** and object **?o**". The question mark (?) means that we are using a variable.




```
2 WHERE {
3     ?s ?p ?o
4 }
```

The variables we used in our SELECT line, are then used in the rest of the query. In the *WHERE* part we list all of variables in the same order as in the SELECT, this gives is a query saying: "**SELECT** every subject **?s** and predicate **?p** and object **?o WHERE** we have a triple with a subject **?s**, predicate **?p** and a object **?o**". This gives us everything in the database.

When we query the SIB with the Python KPI we will get back a list of triples back, containing everything in the database.

While this query shows the main building blocks of a SPARQL query, we still do not have much use for it, as we seldom need the entire database. Let us make a query that we actually might make in real life. In this example we will query for all the lamps in the database.

```
1 PREFIX t: <http://sm3-tut/Ontology.owl#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 SELECT ?s
4 WHERE {
5     ?s rdf:type t:Lamp
6 }
```

The first new keyword you come across is *PREFIX* on line 1-2, which is not required, but lets us specify constants for the name spaces we are going to use. For instance on line 1 we specify the constant *t* to be a shortcut to our ontology. Now instead of writing *http://sm3-tut/Ontology.#Lamp* we can just write *t:Lamp* in our query. You migh also be wondering where the name space on line 2 is comming form. This is a name space created by the W3C to have a standardized way of making queries for certain things. In our case we want to ask for all the subjects with the type lamp, that is done by using *http://www.w3.org/1999/02/22-rdf-syntax-ns#type* as the predicate in our query.

```
3 SELECT ?s
```

In this example we are only selecting the subject of all the triples we get, as the other parts of triples will be the same.

```
4 WHERE {
5     ?s rdf:type t:Lamp
6 }
```

Here is the biggest change from where we queried for everything. It might be easier to understand the query if we write it as plain text at this point. "**SELECT** subjects **?s WHERE** the subject **?s** is of **rdf:type t:Lamp**". If we left the out *t:Lamp* part, and inserted *?o*, we would be querying for all triples which includes a type.

We can also have more than one triple in our *WHERE* statement. We do this by adding more triples, and ending all but the last one with a "dot". As a small example, we are now going to select all devices, and their state.

```
1 PREFIX t: <http://sm3-tut/Ontology.owl#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT ?device, ?state
4 WHERE {
5     ?device rdfs:subClassOf t:Device.
6     ?device ?p ?state
7 }
```

You might noticed that we change one of our previous prefixes from *rdf* to *rdfs*. This is to get access to the *subClassOf* property, which gives all the subclasses of another class.

```
1 PREFIX t: <http://sm3-tut/Ontology.owl#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

After that we start of by selecting the *?devices* and their *?state*.

```
3 SELECT ?device, ?state
```

Now in the *WHERE* statement we add a new line, and add a "." to the first triple.

```
4 WHERE {
5     ?s rdfs:subClassOf t:Device.
6     ?s ?p ?state
7 }
```

What our query now says is that "**SELECT** all **device**s and their **state**s **WHERE** the **device** is a **subClassOf t:Device** and those **device**s have the object **state**".

Testing SPARQL queries

If you want to test SPARQL queries while creating your ontology, Protégé has a integrated SPARQL queries. If you want to test queries on the actual SIB however, there is a Web SIB Explorer program you can download at `http://eslab.github.com/Web-SIB-Explorer/`.

This was a basic example of how to query for triples in SPARQL. We will now extend the queries above to write our consumer.

4.5.1 Creating the consumer

Now it is time to create the part that a real user would probably be looking at for getting an overview of their home automation system. The consumer we are going to create will show the status of both the motion sensor and the lamp. You might be wondering why, in the last section, we talked about using SPARQL for the consumer. Why not use subscriptions and get an always up to date value. Well, we are actually going to use both. We will be getting our devices with SPARQL queries, and dynamically creating subscriptions for these devices.

We will first go through the query for getting all the devices and their states, and after that we will look at the actual Python code.

```
4 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
5 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
6 PREFIX ns: <http://sm3-tut/Ontology.owl#>
7
8 SELECT ?device ?device_type ?attribute ?state
9   WHERE {
```

```
10      ?device_type rdfs:subClassOf ns:Device.
11      ?device rdf:type ?device_type.
12      ?device ?attribute ?state FILTER(?attribute != rdf:
           type)}
```

For getting a list of all our device, we will need to know the device name, device type, the device property and its state. This brings us to our first actual line in our SPARQL query.

```
5  SELECT ?device ?device_type ?attribute ?state
```

We have now stated the four things we want to get in our results. Next we need to define where we are going to get these values from. We start of by selecting all available devices by adding the following line to our query.

```
6    WHERE {
7      ?device_type rdfs:subClassOf ns:Device.
```

Remember to add the ”dot” to the end of the line, as we will be adding more lines to our *WHERE* statement. Next we want to select the device types, we do this by selecting all the triples that start with the device just selected, and that has a type.

```
8  ?device rdf:type ?device_type.
```

We now have the device, and its device-type, we still need the property and the property-value. We can get those by selecting all the triples that starts with the device, and has a predicate that is not one of the RDF standards predefined ones. In this case, we do not want any triples including rdf:type as a predicate. We will have to filter out those triples with a *FILTER* statement.

```
9  ?device ?attribute ?state FILTER(?attribute != rdf:type)}
```

What the filter statement says here is that select the previous triples, but do not include the once that have *rdf:type* as the attribute.

That is it for the query, we can now move on to the Python code. As before, we will first take a look at the entire code, and then go through it in smaller pieces.

```
1  from smart_m3.m3_kp import *
2  import uuid
3  import subprocess
4  import platform
5
6  NS = "http://sm3-tut/Ontology.owl#"
7
8  class OnChangeHandler:
9      def __init__(self, node):
10         self.node = node
11
12     def handle(self, added, removed):
```

```
13          self.node.show_devices()
14
15  class ConsumerKP(KP):
16      def __init__(self, server_ip, server_port):
17          KP.__init__(self, str(uuid.uuid4())+"_Consumer")
18          self.ss_handle = ("X",
19                  (TCPConnector,
20                  (server_ip,server_port)
21                  ))
22          self.subscriptions = []
23
24      def join_sib(self):
25          self.join(self.ss_handle)
26
27      def leave_sib(self):
28          for subscription in self.subscriptions:
29              self.CloseSubscribeTransaction(subscription)
30          self.leave(self.ss_handle)
31
32      def update(self, i_trip, r_trip):
33          upd = self.CreateUpdateTransaction(self.ss_handle
                )
34          upd.update(i_trip, "RDF-M3", r_trip, "RDF-M3")
35          self.CloseUpdateTransaction(upd)
36
37      def create_subscription(self, trip):
38          st = pd.CreateSubscribeTransaction(pd.ss_handle)
39          initial_results = st.subscribe_rdf(trip,
40                                             OnChangeHandler(self)
                                                )
41          self.subscriptions.append(st)
42
43      def sparql_query(self, sparql):
44          PREFIXES = """
45  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
46  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
47  PREFIX ns: <http://sm3-tut/Ontology.owl#>
48          """
49          q = PREFIXES+sparql
50          qt = self.CreateQueryTransaction(self.ss_handle)
51          results = qt.sparql_query(q)
52          self.CloseQueryTransaction(qt)
53
54          return results
55
56      def get_devices(self):
```

```
57          sparql = """
58              SELECT ?device ?device_type ?attribute ?state
59              WHERE {?device_type rdfs:subClassOf ns:Device
                    .
60                  ?device rdf:type ?device_type.
61                  ?device ?attribute ?state
62                  FILTER(?attribute != rdf:type)}
63                  """
64          results = self.sparql_query(sparql)
65
66          devices = []
67          for result_device in results:
68              device = {'attribute':{}}
69              for trip in result_device:
70
71                  if trip[0] == "attribute":
72                      lastAttrib = trip[2]
73
74                  elif trip[0] == "state":
75                      device['attribute'][lastAttrib] =
                           trip[2]
76
77                  else:
78                      device[trip[0]] = trip[2]
79
80              devices.append(device)
81
82          return devices
83
84      def create_device_subscriptions(self,devices):
85          for device in devices:
86              trip = [Triple(URI(device['device']),
87                      None,
88                      None)]
89              self.create_subscription(trip)
90
91      def show_devices(self):
92          subprocess.Popen( "cls"
93                          if platform.system() == "Windows"
94                          else "clear", shell=True)
95          print "Current status (type 'exit' to quit):"
96          devices = self.get_devices()
97          for device in devices:
98              print "\nDevice: "+device['device'].replace(
                    NS,"")
99              for attr, state in device['attribute'].
```

```
                iteritems():
100              print "\t",attr.replace(NS,""),": "\
101                      ,state
102         return devices
103
104 pd = ConsumerKP("192.168.56.101", 10010)
105 pd.join_sib()
106 pd.create_device_subscriptions(pd.show_devices())
107 while True:
108     i = raw_input('\nType "exit" to exit the program\n')
109
110     if i.lower() == "exit":
111         break
112
113 pd.leave_sib()
```

We start of with two new imports, *subprocess* and *platform*. The rest should be known by now, but *subprocess* and *platform* are new. These packages are only going to be used once, and only for esthetic's so if you for some reason don't want to include them, feel free to leave them out, but remember to also change the code where these packages are used. They give us the possibility to run programs and commands outside of our python code. In our case we will use them to clear the screen between sensor updates.

```
1 from smart_m3.m3_kp import *
2 import uuid
3 import subprocess
4 import platform
```

Now that we have the imports done, we also need to include our name-space. As we are still using the same ontology as before, we include the code from our previous programs.

```
6 NS = "http://sm3-tut/Ontology.owl#"
```

Next we will create our subscription handler. What we want our handler to do is that every time our subscription gets a new triple, we update the output displayed for the user of the consumer.

```
8 class OnChangeHandler:
```

As in the aggregator example, we will need the instance calling the handler to be able to call its function so we include the argument *node* to our *__init__* function.

```
9     def __init__(self, node):
10         self.node = node
```

We also need to create the mandatory handle function, which state what will happen when we get a triple matching our subscription. In our case, we don't care if it is an insert or remove, we just want to know if there has been a change.

So every time the handle function is called, all it does it call a *show_devices()* function, which we will create later on.

```
12      def handle(self, added, removed):
13          self.node.show_devices()
```

That is all for our handler, now it is time to create our KP. We will call the class *ConsumerKP*, and also give the KP a unique name ending with the string *_Consumer*. We also initialize the extended KP class on line 18, and create our smart-space handle on line 19-22. We do however also include a new list called *self.subscriptions*. The list will hold all of the subscriptions we are going to create later on.

```
16      def __init__(self, server_ip, server_port):
17          KP.__init__(self, str(uuid.uuid4())+"_Consumer")
18          self.ss_handle = ("X",
19                  (TCPConnector,
20                  (server_ip,server_port)
21                  ))
22          self.subscriptions = []
```

The *join_sib* function stays the same as in all the previous KPs.

```
24      def join_sib(self):
25          self.join(self.ss_handle)
```

The *leave_sib* however has one small change. When we leave the SIB, we also want to close all of our subscriptions, and this is where the *self.subscriptions* list comes in handy. On line 29-30, we traverse over the list, and close every subscription in it. When that is done, we close the connection to the SIB.

```
27      def leave_sib(self):
28          for subscription in self.subscriptions:
29              self.CloseSubscribeTransaction(subscription)
30          self.leave(self.ss_handle)
```

The *update* function stays the same as in the aggregator. We give it two triples, and it performs an update by first inserting the new triple, and then removing the old one.

```
32      def update(self, i_trip, r_trip):
33          upd = self.CreateUpdateTransaction(self.ss_handle
                )
34          upd.update(i_trip, "RDF-M3", r_trip, "RDF-M3")
35          self.CloseUpdateTransaction(upd)
```

Next up we have the *create_subscription* function which we have to change a bit to serve us better when create multiple subscriptions. First we create a subscription transaction and then we create our subscription and pass it the handler *OnChangeHandler* which we created earlier. Then we add the subscription to our subscriptions list to keep track of all our subscriptions.

```
37      def create_subscription(self, trip):
38          st = pd.CreateSubscribeTransaction(pd.ss_handle)
39          initial_results = st.subscribe_rdf(trip,
40                                             OnChangeHandler(self)
                                                   )
41          self.subscriptions.append(st)
```

As we are going to use SPARQL queries in this program, we might as well create a helper function for this so that we don't have to remember all of the function calls and prefixes in every query we make. We call our function *sparql_query* and give it the argument *sparql* which will be the actual query we want to run.

```
43      def sparql_query(self, sparql):
```

The prefixes used in our queries will most likely be the same every time, so we start of by defining those in a constant. We then add these prefixes to our query.

```
44          PREFIXES = """
45  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
46  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
47  PREFIX ns: <http://sm3-tut/Ontology.owl#>
48          """
49          q = PREFIXES+sparql
```

After that we create our query transaction, pass it our query and store the results in the variable *results*. And as with all transaction to the SIB, we also need to close it.

```
50          qt = self.CreateQueryTransaction(self.ss_handle)
51          results = qt.sparql_query(q)
52          self.CloseQueryTransaction(qt)
```

Lastly we return the results to the calling function.

```
54          return results
```

That concludes the helper functions, now we move on to our actual code. We start of by create the function for getting all our devices.

```
56      def get_devices(self):
```

The first thing we do is to specify our SPARQL query which we create in the beginning of this section. We also pass this query over to our *sparql_query* function and get back the results of our query.

```
57          sparql = """
58              SELECT ?device ?device_type ?attribute ?state
59              WHERE {?device_type rdfs:subClassOf ns:Device
                    .
60                  ?device rdf:type ?device_type.
61                  ?device ?attribute ?state
62                  FILTER(?attribute != rdf:type)}
```

```
63                  """
64          results = self.sparql_query(sparql)
```

Now comes the tricky part, parsing the results. We want to store all our triples on device basis instead of just triples. We can do this in several ways, for instance creating custom objects, lists or dictionaries of the devices. We are going to create a list, which will includes dictionaries including the device data.

```
66          devices = []
```

Before the go into more code, we need to know some small things about the structure of the results we are getting from the Python KPI. The results will always contain at least one list, which in turn might contain lists containing the result triples of our query. To get a better view of this we can use Pythons *pprint* (Pretty Print) and print the results. The kind of results we are expecting in our case will look something like the one below.

```
1  [ [ [ u'device',
2        u'uri',
3        u'http://sm3-tut/Ontology.owl#Lamp_living_room_1'],
4
5      [ u'device_type',
6        u'uri',
7        u'http://sm3-tut/Ontology.owl#Lamp'],
8
9      [ u'attribute',
10       u'uri',
11       u'http://sm3-tut/Ontology.owl#hasState'],
12
13     [ u'state',
14       u'literal',
15       u'false']
16   ],
17     [ [ u'device',
18         ...
19       ],
20       ...
21     ]
22 ]
```

Lets start traversing the lists and extracting the data parts we want. We will do this with a double for-loop. The first one will go through all the devices, and the second one will go through the triples. We will also need to create a new device to store our information in every time the outer loop runs.

```
67          for result_device in results:
68              device = {'attribute':{}}
69              for trip in result_device:
```

Many ways to code

As you might have noticed we could just use one loop here as we know exactly what we are getting back. We could just say *device['device'] = result_device[0][2]* and so on, to get the values in the first loop. But as we might add more things to our query later on, it is better to keep things a bit more dynamic, and therefore we are using a second loop.

As you can see we also create a key value *attribute* with a new dictionary to store our attribute value in. We want a dictionary here because we are going to store the name of the attribute, as well as its value.

Next we will check if the triple is an attribute. If it is an attribute we set this to an *lastAttrib* variable.

```
71                  if trip[0] == "attribute":
72                      lastAttrib = trip[2]
```

When we get the value for that attribute we save it in the *attribute* dictionary with the attribute as key and the state as its value.

```
74                  elif trip[0] == "state":
75                      device['attribute'][lastAttrib] =
                            trip[2]
```

Dangerous Coding

Please note that what we are doing here is a bit dangerous. We are expecting the attribute to always come before the state. We need to be careful that we always specify the *?attribute* and *?state* value in the correct order in our SPARQL query for this to work. So if you change something in the query, please remember that the order of the variables in our *SELECT* statement is important.

Anything else, that is not a attribute or value, we save to the device with the first part of the triple as the key and the third value as the entries value.

```
77                  else:
78                      device[trip[0]] = trip[2]
```

When we are done traversing over the triples we append the device to our *devices* list.

```
80              devices.append(device)
```

Lastly we return our *devices* list.

```
71          return devices
```

Next we are going to create a function to easily create our subscriptions to the devices we previously got from our query. To do this, we will simple go though all of our devices, and create subscriptions to all of them.

```
84      def create_device_subscriptions(self,devices):
```

We start of with a for-loop, going through all our devices.

```
85          for device in devices:
```

We then create a triple, including only our device URI as a subject, and wild-cards for the predicate and object. What this mean is that our subscription will match any insert or remove action, that includes our device.

```
86              trip = [Triple(URI(device['device']),
87                      None,
88                      None)]
```

Lastly we call our *create_subscription* function, which creates the subscription for us.

```
89              self.create_subscription(trip)
```

Our consumer is almost done now, all we have left to do is to output all of the information to the user. For this we will create a function *show_devices* that we call from our handler in the beginning of the program.

```
91      def show_devices(self):
```

First we clear the screen to get a nice empty shell to show the info in. This is also the part you will need to leave out, if you did not want to import the *subprocess* and *platform* packages.

```
92          subprocess.Popen( "cls"
93                            if platform.system() == "Windows"
94                            else "clear", shell=True)
```

Then we give the user a quick info text of how to exit the program.

```
95          print "Current status (type 'exit' to quit):"
```

After that we need to get all our devices, so we call the *get_devices* function we created before.

```
96          devices = self.get_devices()
```

Now it is time for the actual output. We start of by creating a for-loop, running over all our devices.

```
97          for device in devices:
```

We then print the device name, without the name-space. This is just so that the user is given a clearer view of what is going on. If you would like to include the name-space in the output, then you can remove the *replace(NS,"")* at the end of the print statement.

```
98              print "\nDevice: "+device['device'].replace(
                    NS,"")
```

We also want to view the state of the device, for this we create another for loop, giving us the attribute name, as well as its value. We use *iteritems()* on the dictionary, as this will give back both the key and the value of a dictionary entry. Without using it, we would only get the key back.

```
99              for attr, state in device['attribute'].
                    iteritems():
```

Then we print the attribute, without the name-space, and its value.

```
100                 print "\t",attr.replace(NS,""),": "\
101                         ,state
```

Lastly we return the *devices* list to the calling function.

```
102         return devices
```

Now it is time to run our program. We do this by first creating an instance of our KP and connect it to the SIB.

```
105 pd = ConsumerKP("192.168.56.101", 10010)
106 pd.join_sib()
```

Then it is time to create our subscription and show the device information to the user. We do this by passing the devices return by the *show_devices* function to the *create_device_subscription*.

```
106 pd.create_device_subscriptions(pd.show_devices())
```

We then create an ”infinite” while-loop prompting the user for an exit command.

```
107 while True:
108     i = raw_input('\nType "exit" to exit the program\n')
```

If the input is equal to ”exit” then we break the loop.

```
110     if i.lower() == "exit":
111         break
```

The last thing we do is close all of our subscriptions, as well as the connection to the SIB.

```
113 pd.leave_sib()
```

Congratulations, we are now done! The only thing left to do is saving our consumer, and running it at the same time as the producer and the aggregator. If you want to, you could run any of the programs on any machine with Python and KPI installed that has a network connection to the SIB. Figure 11 is showing the consumer running on Windows 8 for instance.

If everything went well, when we change the value of the motion sensor, in the producer, the aggregator will pick up on the update, and change the value of the lamp in the consumer. If this change happens, the tutorial has been completed successfully. If there is an error, or the values does not change, check

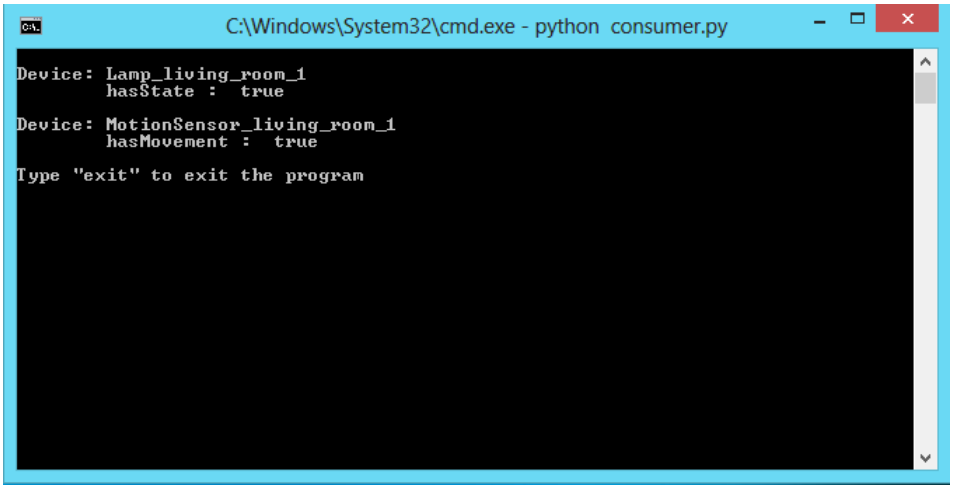


Figure 11: The producer should look similar to this when running in a terminal.

that the code is correct. Remember that we can always look at the data in the SIB with the Web SIB Manager, available at `http://eslab.github.com/Web-SIB-Explorer/`.

5 Conclusion and additions

Our program is now ready, and we now know a lot more about the capabilities of Smart-M3. From this point on, we can build upon the knowledge we have. For instance, what if a device has more than one attribute? What if we had a temperature sensor which can have more values than true and false? These are things that could easily be done with some small modifications to the code we have already written. An example of an consumer, which supports several parameters per device, has been supplied in the code examples included with this tutorial to get you started.

References

[1] J. Honkola, H. Laine, R. Brown, and O. Tyrkko, "Smart-m3 information sharing platform," in *Computers and Communications (ISCC), 2010 IEEE Symposium on*, june 2010, pp. 1041 –1046.

[2] J.-P. Soininen, P. Liuha, A. Lappeteläinen, J. Honkola, K. Främling, and R. Raisamo, "Device interoperability: Emergence of the smart environment ecosystem," December 2010, White Paper. [Online]. Available: http://www.diem.fi/files/DIEM%20White%20Paper.pdf

[3] J. Kiljander, “Reference implementation of interoperable entity for smart environments,” November 2009.

[4] “Smart-m3 - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/smart-m3/files/

[5] ARCES, “Smart-m3 smart-m3 v0.3.1-alpha - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/smart-m3/files/Smart-M3_B_v0.3.1-alpha/

[6] Wikipedia, “Smart-m3 — wikipedia, the free encyclopedia,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Smart-M3&oldid=497914591

[7] “Smart-m3 c kpi - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/smart-m3/files/Smart-M3_B_v0.3.1-alpha/

[8] “Smart-m3 c sharp kpi - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/m3-csharp-kpi/

[9] “Smart-m3 c kpi - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/smart-m3/files/Smart-M3_B_v0.3.1-alpha/

[10] “Smart-m3 c kpi - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/smartm3-javakpi

[11] “Smart-m3 c kpi - sourceforge,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://sourceforge.net/projects/jssmartm3/

[12] Wikipedia, “Sparql — wikipedia, the free encyclopedia,” 2012, [Online; accessed 10-December-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=SPARQL&oldid=527212545

[13] P. S. Foundation, “Smart-m3 — wikipedia the free encyclopedia,” 2012, [Online; accessed 5-November-2012]. [Online]. Available: http://docs.python.org/2/library/uuid.html

[14] “Smart-m3 c kpi - sourceforge,” 2012”, [Online; accessed 5-November-2012]”. [Online]. Available: http://sourceforge.net/projects/sm3-php-kpi-lib

[15] E. Prud’hommeaux and A. Seaborne, “Sparql query language for rdf,” Januari 2008, [Online; accessed 5-November-2012]. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2867-4
ISSN 1239-1891