MASTER OF SCIENCE
IN ENGINEERING

Internet of Thing: Lab 1

A Smart Building Software based on three home automation communication protocols: KNX, Z-WAVE and BLE (Beacons)

I. Introduction

This lab deals with the development of a smart building software which provides a set of high level functionalities (or end-user functionalities) such as:

1. lower the temperature of a room to a given threshold when it is empty,
2. increase the temperature of a room to a given threshold when it is occupied,
3. close the blinds when the humidity is high,
4. open the blinds at day time, when the luminance is low and the room is occupied,
5. display the status of a given store and/or a given radiator,
6. Manually monitor blinds and radiators of the room where the user is,
7. provide statistics.

The goal of this lab is not to develop all these functionalities, but to design and develop "low services" tools on top of which the functionalities listed above will rely.

During this lab, we assume that we are monitoring a building: blinds, valves (radiators) and lamps according to different inputs such as temperature, light, humidity, physical presence and indoor location of the user. Each room of the building has a sensor, a lamp, a blind, a radiator and a beacon (for indoor location).

Three different network protocols will be used:

1. KNX : to monitor blinds and radiators (Figures 1 and 2).
2. Z-Wave : to collect measures of temperature, light, humidity, presence, battery level. All these measures are provided by the same sensor (Figure 3)
3. Beacon (based on BLE) : to manage indoor location (Figure 3)

The lab will be split into four steps (Figure 4):

1. Step 1: develop the KNX low level services: **Thursday 27 Sept. and Thursday 4 Oct**.

2. Step 2: develop the Z-Wave low level services: **Thursdays 11 and 18 Oct.**
3. Step 3: develop the Beacon low level services: **Thursday 25 Oct.**
4. Step 4: integrate the three low services into one framework (Figure 4) used by the smart building application: **Thursday 1st November**

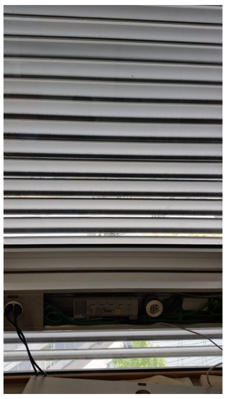	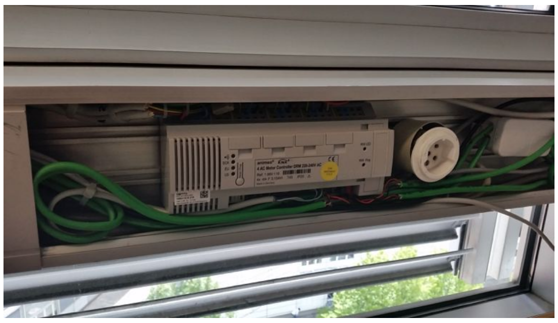
Figure 1.a: Blinds at hepia building	*Figure 1.b: Animeo KNX Somfy actuators used to monitor the blinds*

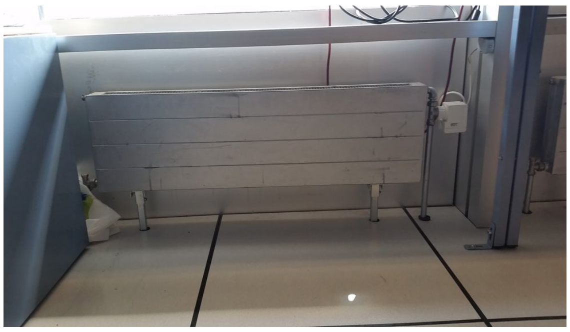	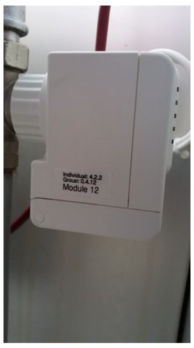
Figure 2.a: Radiators at hepia building	*Figure 2.b:* KNX *radiator valve servo motor* used to monitor the radiators

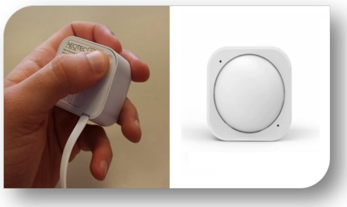	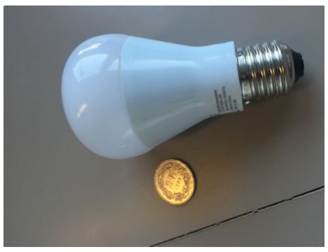
Figure 3.a: IoT Z-Wave device used to measure temperature, humidity, presence and light	*Figure 3.b:* Z-Wave lamp

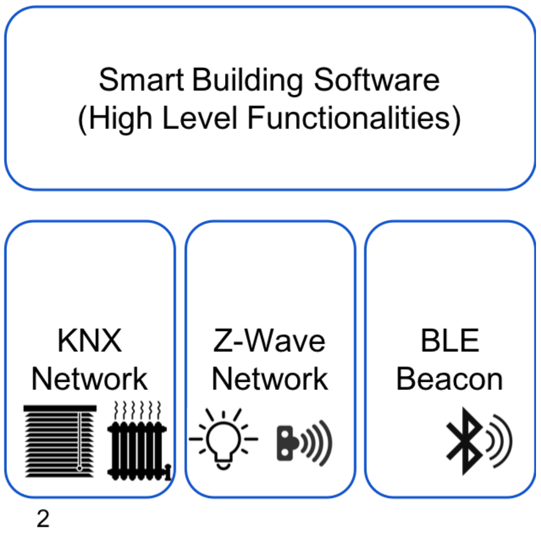

Figure 4: One lab, four steps

This document details step 1 (KNX) of the lab.

II. KNX

KNX is an open standard for domestic building automation. KNX devices can manage lighting, blinds, security systems, energy management, audio video, displays, remote control, etc. KNX can use twisted pair, powerline, RF, infrared or Ethernet links.

We will be using KNX on twisted pair to monitor blinds and radiators installed in the 4th and 5th floors of the hepia building in 4 Rue de Prairie, Geneva. *Animeo KNX Somfy actuators* (resp. KNX *radiator valve servo motor actuators*) are used to monitor the blinds (resp. radiators). Each actuator has its own address, called a group address in the KNX terminology. The structure of the group address will be detailed in the lecture. In our case, a group address has the following form: ***x/y/z*** where:

- *x* is the type of command to send to the actuator (read below)
- *y* is the floor where the blind or the radiator is located
- *z* is the block to which the blind or the radiator belongs. One block is composed of one blind and one radiator.

x can have the following values:

- 0: control of the valves.
- 1: control of the blinds.
- 2: do not use
- 3: control of the blinds.

- 4: Reading the state of the blind

The monitoring of a KNX actuator is done through KNX telegrams. KNX telegrams have the following structure:

- Control Field
- Source Address
- Destination Address (group address)
- Address Type
- Hop Count
- Length
- Payload (data)
- Checksum

The core of a KNX telegram is the payload field.

In our case, the payload value (data) depends on the group address, in particular, the x value

- x= 0: control of the valves. The payload is an integer (2 bytes): 0 (0%) 255 (100%)
- x= 1: control of the blinds. The payload is a bit (1 byte): 0 (100% open) or 1 (100% closed).
- x= 2: do not use
- x= 3: control of the blinds. The payload is an integer (2 bytes): 0 (fully opened) .... 255 (fully closed)
- x= 4: Reading the state of the blind: 0 (fully opened) .... 255 (fully closed).

III. KNXnet/IP

knxnet/IP is an a protocol that allows users to bridge the gap between the real world (KNX network is our case) and the digital world (TCP/IP network and virtual objects). This idea has already been explained in the previous lecture (Figure 5).

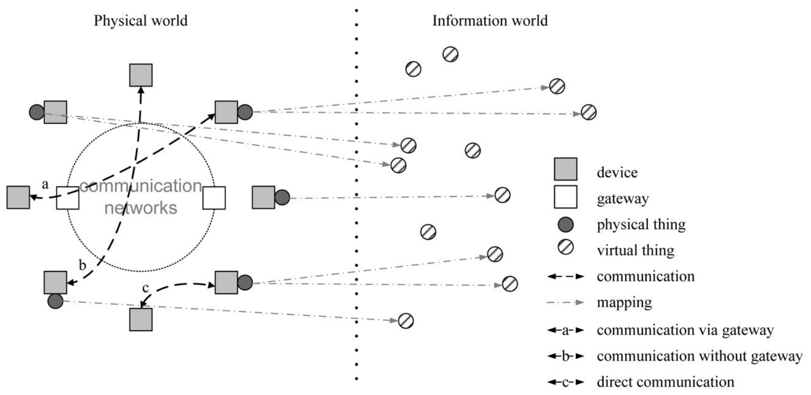

Figure 5: Technical overview of the IoT (ITU recommendation)

The idea behind knxnet/IP is to encapsulate KNX telegrams into UDP socket messages and to send them to a dedicated gateway: KNX/IP gateway (Figure 6). Then, the KNX/IP gateway communicates with the KNX device using KNX telegrams.

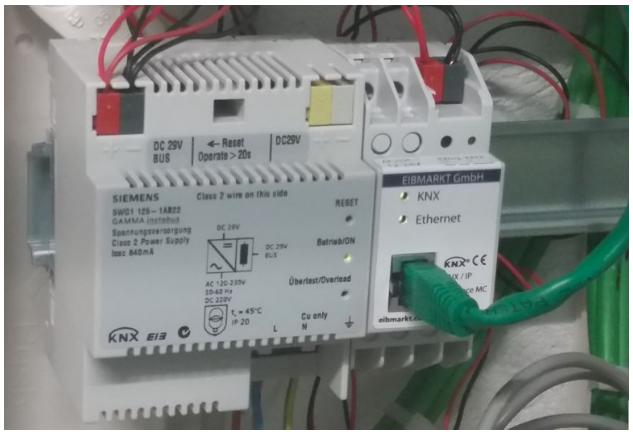

Figure 6: KNX/IP gateway. The RJ45 wire is the TCP/IP network. The red and black wires are the KNX network

The communication with the gateway is done according to an appropriate protocol. Figure 7 shows the messages exchanged (datagrams) between a client (in blue) and the KNX/IP gateway (in green) in the case of a “read” (retrieving data) and a “write” (sending command).

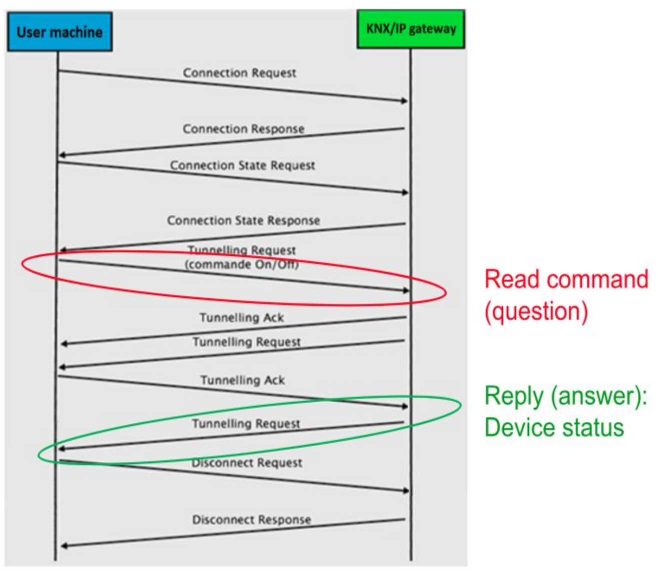

Figure 7.a: KNX/IP protocol: “read” operation: retrieving data

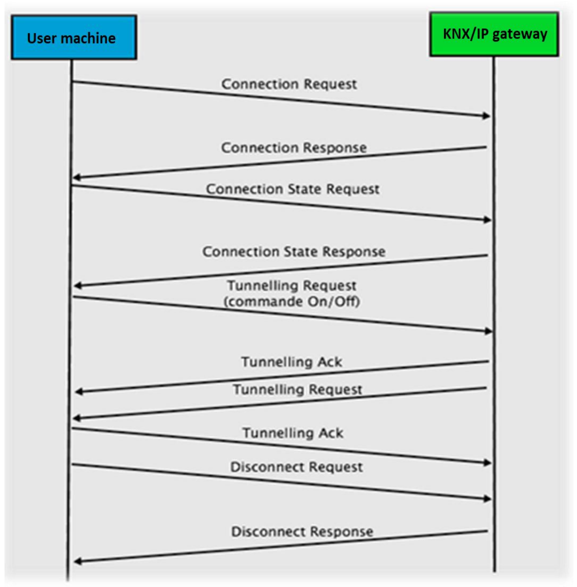

Figure 7.b: KNX/IP protocol used for a “write” operation: sending command

This protocol is the following:

1. The client program sends a "Connection Request" to request a connection. This request contains two Host Protocol Address Information (HPAI) which are two pairs (IP address, Port number). The IP address is the same for both HPAIs but the port numbers may change. The first port is used to send the "Connection_Request", "Connection_State_Request", "Disconnect_Request", "Connection_Response", "Connection_State_Response" and "Disconnect_Response" telegrams. In summary, this port is used to manage the connection with the KNXnet/IP gateway. The second port is used to send and receive the "Tunneling Request" and "Tunneling Ack" telegrams that contain the data (commands) that the KNXnet/IP gateway must transform into KNX telegrams and pass them to the KNX bus. For simplicity, we will use the same port in both HPAI.
2. The gateway returns a "Connection Response" with a "Channel ID" which is a unique identifier of the connection. This channel ID will be used by connections: Connection State Requests, Tunneling Requests, Tunneling Acks and Disconnect Requests.
3. A "Connection State Request" is sent with the "Channel ID" (point 2) to test the connection with this "Channel ID".
4. Reception of a "Connection State Response" with an error code (called "status") "00".
5. At this point, the connection is established. We can now send data necessary for the construction of a "Tunneling Request" telegram (sent from the second port specified in the "Connection Request") with a "Data Service" field set to 0x11 ("Data.request").
6. Upon receipt of the "Tunneling Request", the KNXnet/IP gateway returns an acknowledgment ("Tunneling Ack") with an error code (called "status") "00" if there has been no error.
7. The KNXnet/IP interface checks our "Tunneling Request" and returns it with the "Data Service" field set to 0x2e ("Data.confirmation").
8. The client program compares the Tunneling Request already sent and the Tunneling Request returned by the KNXnet/IP gateway. If there are no transmission errors, the client program sends an acknowledgment ("Tunneling Ack") with an error code (called "status") "00" (zero).
9. If it is a request to read the state of an actuator, the client program receives a "telegram" of the "Tunneling Request" type containing the data.

Concretely, to monitor a KNX device, we must send him the “good” KNX telegrams (through the KNX/IP gateway) at the right moment according to the protocol of KNX. This is a cumbersome task !

We propose to use a dedicated Python API which hide the complexity of the KNXnet/IP datagrams. This Python API is **developed by Adrian Lescourt and maintained by Nizar Bouchedakh**). It allows the user to:

- Send sockets messages to the KNX/IP gateway The socket messages will then be translated by the KNX/IP gateway into a KNX telegram before being sent to the appropriate KNX device

- Receive sockets messages from the KNX/IP gateway, which encapsulate KNX telegram. knxnet will then extract the data wrapped within the KNX telegram.

Python API for KNXnet/IP datagrams

The installation Python API for KNXnet/IP is detailed in Annex 1.

The main objects and methods of this API are:

"ServiceTypeDescriptor": Enumerative object that contains the different types of telegrams possible:

Telegram type	**Attribute**
Connection Request	**CONNECTION_REQUEST**
Connection Response	**CONNECTION_RESPONSE**
Connection State Request	**CONNECTION_STATE_REQUEST**
Connection State Response	**CONNECTION_STATE_RESPONSE**
Disconnect Request	**DISCONNECT_REQUEST**
Disconnect Response	**DISCONNECT_RESPONSE**
Tunnelling Request	**TUNNELLING_REQUEST**
Tunnelling Ack	**TUNNELLING_ACK**

The creation of an object representing a future KNX telegram is done through the 'create_frame' method of the knxnet class. The telegram type is the first argument of this constructor. It is followed by parameters that depend on the type of telegrams to be created:

a) CONNECTION_REQUEST

Attribute	Description

`control_endpoint`	Formatted as (IP address, Port): used to send or receive objects that represent telegrams of the type *Connection Request/Response*, *Connection_State Request/Response* and *Disconnect Request/Response*.
`data_endpoint`	Formatted as (IP address, Port) : used to send and/or receive objects representing telegrams of type Tunnelling Request/Ack

b) CONNECTION_RESPONSE

attribut	description
`channel_id`	Channel identifier allocated by the KNX/IP gateway to communicate with the client program. The gateway may communicate with multiple clients at the same time using the "channel" notion.
`status`	Connection status: 0 (zero): OK. Else: error message (cf. doc).
`data_endpoint`	Cf. (a)

c) CONNECTION_STATE_REQUEST

Attribute	Description
`channel_id`	Cf. (b)
`control_endpoint`	Cf. (a)

d) CONNECTION_STATE_RESPONSE

attribut	description
`channel_id`	C.f. (b)

`status`	Connection status requested by "Connection State Request": 0 "Connection State Request" accepted. Else: error message (cf. doc).

e) TUNNELLING_REQUEST

attribut	Description
`dest_addr_group`	Group address of the device to control (read or write). To create a group address from a string, use this method: `dest_addr_group = knxnet.GroupAddress.from_str("x/y/z")`
`channel_id`	Cf. (b)
`data`	The value to send to the KNX/IP gateway (device control) or to receive from the KNX/IP gateway (in the case of a Tunnelling request generated by the gateway). This value is between 0 and 255. In the case of a read operation, this value has no meaning.
`data_size`	Size of the “data” field that has been sent.
`apci`	This field specifies the request type: 0x0 == read query (sent by client); 0x1 == response to previous query (in case of Tunnelling request sent by gateway following a Tunnelling request sent by the client) 0x2 == write query (sent by the client).

`data_service`	Optional field. It may have three possible values: 1. Data.request (0x11) : corresponds to Tunnelling requests sent by the client to the gateway. 2. Data.confirmation (0x2e) : corresponds to a Tunnelling request sent by the gateway in response to a Tunnelling request de type « Data request ». 3. Data.response : corresponds to the second Tunneling request sent by the gateway to respond to a read request (Tunneling request of type Data.request)
`sequence_counter`	Optional field.

f) TUNNELLING_ACK

attribut	description
`channel_id`	Cf. (b)
`status`	Query status sent in "Tunnelling Request". 0 if OK. Other value if error (cf. doc).
`sequence_counter`	The same "sequence_counter" as in the "Tunnelling Request" telegram to be acknowledged.

g) DISCONNECT_REQUEST

attribut	description
`channel_id`	Cf. (b)
`control_endpoint`	Cf. (a)

h) DISCONNECT_RESPONSE

attribut	description

`channel_id`	Cf. (c)
`status`	It is the status of your disconnection query sent in "Disconnect Request". If it is 0 (zero), then everything is OK and your "Disconnect Request" was accepted. If the value is different from 0, then it is an error message (cf. doc).

Example: to create a "Connection Request" telegram, call the create_frame method with the following parameters:

Conn_req_obj = knxnet.create_frame (knxnet.ServiceTypeDescriptor.CONNECTION_REQUEST, ('0.0.0.0',0), ('0.0.0.0',0))

The following code (incomplete) starts the initialization of the connection with a KNX / IP gateway:

```
# -*- coding: utf-8 -*-

import socket,sys

from knxnet import *

gateway_ip = "IP address KNXnet/IP gateway"

gateway_port = Listening Port"

# -> in this example, for sake of simplicity, the two ports are the same

# With the simulator, the gateway_ip must be set to 127.0.0.1 and gateway_port to
3671

data_endpoint = ('0.0.0.0', 3672)

control_enpoint = ('0.0.0.0', 3672)

# -> Socket creation

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock.bind(('',3672))

# -> Sending Connection request

conn_req_object=
knxnet.create_frame(knxnet.ServiceTypeDescriptor.CONNECTION_REQUEST,
control_enpoint, data_endpoint)
```

```
conn_req_dtgrm = conn_req_object.frame # -> Serializing

sock.sendto (conn_req_dtgrm, (gateway_ip, gateway_port))

# <- Receiving Connection response

data_recv, addr = sock.recvfrom(1024)

conn_resp_object = knxnet.decode_frame(data_recv)

# <- Retrieving channel_id from Connection response

conn_channel_id = conn_resp_object.channel_id
```

Finish the rest !

Annex 1: How to install Python API for KNXnet/IP on a Linux ubuntu machine ?

1. sudo apt-get update
2. sudo apt-get install python3-setuptools
3. sudo apt-get install git
4. sudo git clone https://githepia.hesge.ch/adrienma.lescourt/knxnet_iot.git
5. cd knxnet_iot/
6. sudo python3 setup.py install

Annex 2: How to install Actuasim on a Linux ubuntu machine ?

1. sudo apt-get update
2. sudo apt-get install python3-pyqt5
3. sudo apt-get install git
4. git clone https://githepia.hesge.ch/adrienma.lescourt/actuasim_iot.git
5. cd actuasim
6. python3 actuasim.py