Internet of Things (IoT)

Lab 1: KNX

September 2020

Nabil Abdennadher, Marco Emilio Poleggi

Introduction

The goal of this Lab is to develop a python program which monitors and controls a set of blinds (Figures 1.a and 1.b) and radiators (Figures 2.a and Figures 2.b) connected to a KNX [1] network.

Figure 1.a: Blinds at hepia building

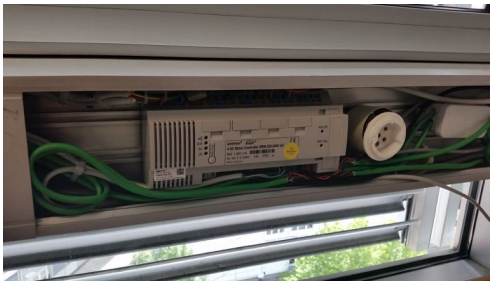

Figure 1.b: Animeo KNX Somfy actuators used to monitor the blinds

Figure 2.a: Radiators at hepia building

Figure 2.b: KNX *radiator valve servo motor* used to monitor the radiators

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 pairs, powerline, RF, infrared or Ethernet links. At the protocol level, the connected devices communicate by exchanging KNX packets called *telegrams*, having the following fields:

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

In this Lab, we will be using KNX on twisted pairs 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 is assigned a logical 3-level *group address* [2] in the form x/y/z whose meaning is arbitrary. In our KNX deployment:

- *x* is the type of command (*action*) to send to the actuator. The command is specified further by the telegram's data field, i.e., the *payload*;
- *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.

The combination of the group address and payload (data) defines the command sent to the controlled device:

- `x = 0`: control of radiator valves. The payload is an integer of 1 or 2 bytes;
- `x = 1, 3, 4`: control of window blinds. The payload is an integer of 1 or 2 bytes;
- `x = 2`: **do not use!**

Please, refer to the lab slides for the full command specifications — the parameter APCI is also needed, as explained later on.

KNXnet/IP

knxnet/IP is a protocol that allows users to bridge the gap between the real world (KNX network in our case) and the digital world (TCP/IP network and virtual objects). This idea has already been explained in the previous lecture (Figure 3).

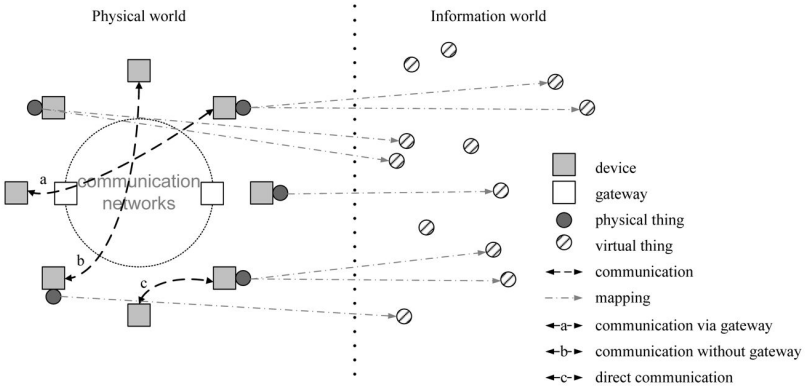

Figure 3: Technical overview of the IoT (ITU recommendation)

The idea behind knxnet/IP is to encapsulate KNX telegrams into UDP socket messages which are sent, through a TCP/IP network, to a dedicated KNX/IP *gateway* (Figure 4). The gateway, on its turn, communicates with the KNX device using KNX telegrams.

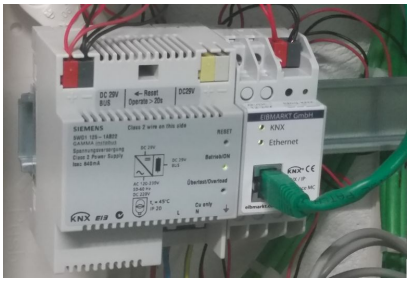

Figure 4: KNX/IP gateway. The RJ45 wire is the TCP/IP network. The red and black wires are the KNX network bus

The communication with the gateway is done according to an appropriate protocol. Figure 5 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: *get* operation) and a “write” (sending command: *set* operation).

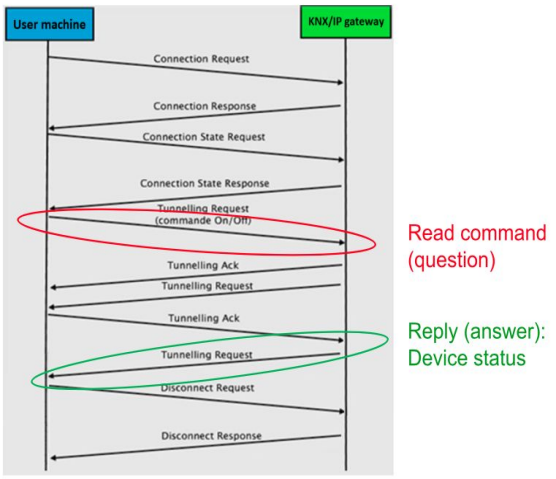

Figure 5.a: KNX/IP protocol: “read” operation: retrieving data

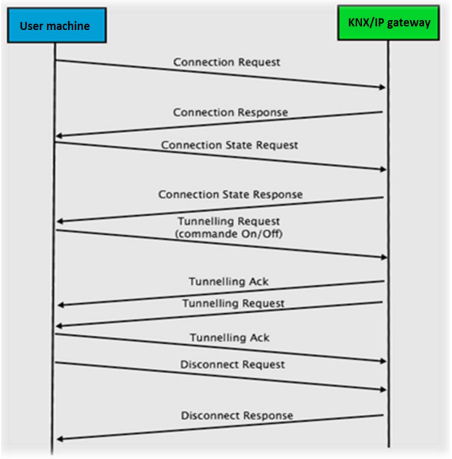

Figure 5.b: KNX/IP protocol used for a “write” operation: sending command

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 along to the KNX bus. For the sake of 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"), normally `0x0`.
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") `0x0` if there is 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") `0x0`.
9. If this is a request to *read* (get) the state of an actuator, the client program shall expect a further telegram of the type "Tunneling Request" containing the read data.

Directly manipulating KNX telegrams is a cumbersome task. Instead, we will use a dedicated Python API to:

- Encode/decode KNX telegrams into/from *frames* conveyed in UDP messages;
- Send/receive the above UDP messages to/from a KNX/IP gateway.

Python API for KNXnet/IP datagrams

Annex 1 describes how to install Python API for KNXnet/IP. This API allows us to manipulate several "Service Type Descriptors", which are enumerative objects abstracting the different types of KNX telegrams. They are listed in the table below.

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 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

Attribute	**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

Attribute	**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

Attribute	**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

Attribute	**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

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

h) DISCONNECT_RESPONSE

Attribute	Description
`channel_id`	Cf. (c)
`status`	It is the status of your disconnection query sent in "Disconnect Request". If it is a 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))
```

Before testing it on real blinds and radiators, you must test your program on a simulator which simulates the blinds, radiators and the Gateway. With the simulator, the IP of the KNX/IP gateway must be set to 127.0.0.1.

Refer to Annex 1 for the installation of the simulator.

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

```
import socket, sys
from knxnet import *
```

```
gateway_ip = '127.0.0.1' # localhost IP is for the simulator. Replace with the
                         # real IP for the physical gateway
gateway_port = 3671      # default for both the simulator and the physical
                         # gateway

# These values are for the simulator -- mind that the port *differs* from
# the gateway's! Replace both with ('0.0.0.0', 0) in a NAT-based VLAN
# (physical deployment)
data_endpoint = ('127.0.0.1', 3672)
control_enpoint = ('127.0.0.1', 3672)

# -> Socket creation
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', gateway_port))

# -> 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
```

Task

You shall:

1. Complete the Python 3 client script **`knx_client_script.py`** available at https://gitedu.hesge.ch/lsds/teaching/master/iot/knx. This script, given a blind/valve device at a given KNX group address, must be capable of:
    a. Open/close partially and totally the device,
    b. Read the status of the device.
2. Test your script with the provided simulator until the tests listed in the Git project's README are OK.
3. Publish your script into a public Git repository of your choice.

Annex 1: installation of the lab's code

These instructions are intended for an Ubuntu Linux instance with Python 3.

```
$ sudo apt-get update
```

```
$ sudo apt-get install git pip python-setuptools python-pyqt5
```

```
$ git clone --recursive
https://gitedu.hesge.ch/lsds/teaching/master/iot/knx.git
```

You should now have the KNXnet/IP library and the simulator in two subdirs.

Python API for KNXnet/IP

```
$ cd knx/knxnet_iot/
```

```
$ pip3 install --user ./
```

```
$ cd ../
```

```
$ knx_client_script.py --man
```

Actuasim

Actuasim

```
$ cd knx/actuasim/
```

```
$ python3 actuasim.py &
```

References

1. KNX - MyKNX. [cited 16 Sep 2020]. Available: https://my.knx.org/en/shop/knx-specifications

2. Group Address & Style. [cited 15 Sep 2020]. Available: http://support.knx.org/hc/en-us/articles/115001825344