Ce document décrit brièvement comment utiliser l'API KNX :knxnet.

Le protocole KNX/IP

La figure 1 montre les messages échangés entre un client (en bleu) et la passerelle KNX/IP (en vert) dans le cas d'une écriture (figure 1.a) et d'une lecture (figure 1.b)

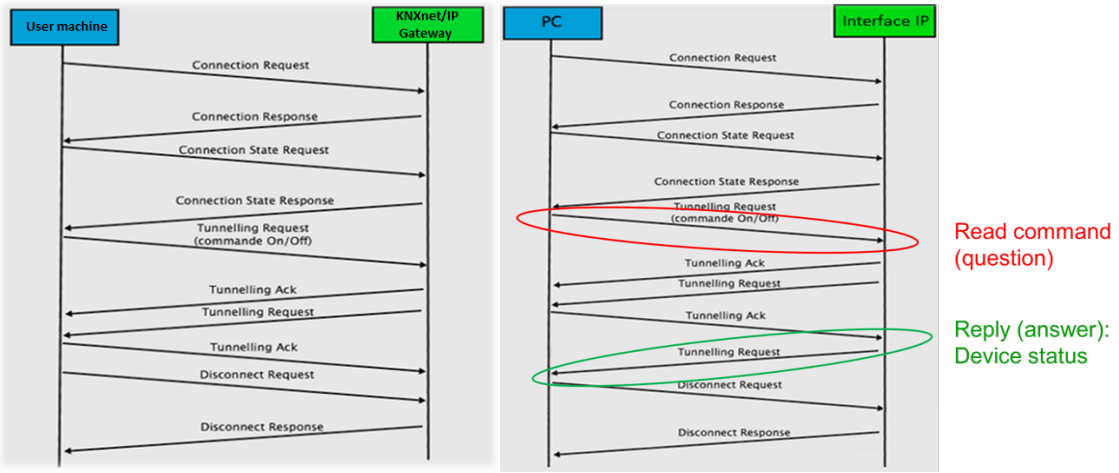

Figure 1 - Utilisation du protocole KNXnet/IP pour l'envoi d'une commande (a) d'écriture, (b) de lecture

1. Le programme client envoie une "Connection Request" pour demander une connexion. Cette requête contient deux Host Protocol Address Information (HPAI) qui sont deux paires (adresse IP, numéro de Port). L'adresse IP est la même pour les deux HPAI mais les numéros de port peuvent changer.

   Le premier port est utilisé pour envoyer les télégrammes *"Connection_Request", "Connection_State_Request", "Disconnect_Request", "Connection_Response", "Connection_State_Response"* et *"Disconnect_Response"*. En résumé, ce port est utilisé pour gérer la connexion avec la passerelle KNXnet/IP.

   Le second post est utilisé pour envoyer et recevoir les télégrammes "Tunnelling Request" et "Tunnelling Ack" qui contiennent les données (les commandes) que la passerelle KNXnet/IP doit transformer en télégrammes KNX et les passer vers le bus KNX. Pour faire simple, on utilisera, le même port dans les deux HPAI.

2. La passerelle renvoie une *"Connection Response"* avec un "Channel ID" qui est un identifiant unique de la connexion. Ce channel ID sera utilisé par les connexions *: Connection State Requests, Tunnelling Requests, Tunnelling Acks et Disconnect Requests*.

3. Une *"Connection State Request"* est envoyée avec le "Channel ID" récupéré au point 2 pour tester la connexion avec ce "Channel ID".
4. Réception d'une *"Connection State Response"* avec un code d'erreur (appelé "status") "00".
5. Arrivé à ce stade, la connexion est établie. Envoie des données nécessaires pour la construction d'un télégramme "Tunnelling Request" (envoyé à partir du deuxième port spécifié dans la "Connection Request") avec un champs "Data Service" mis à 0x11 ("Data.request").
6. Dès réception du "Tunnelling Request", la passerelle KNXnet/IP renvoie un acquittement ("Tunnelling Ack") avec un code d'erreur (appelé "status") "00" s'il n'y a pas eu d'erreur.
7. L'interface KNXnet/IP vérifie notre "Tunnelling Request" et le renvoie avec le champs "Data Service" mis à 0x2e ("Data.confirmation").
8. Le programme client compare le "Tunnelling Request" déjà envoyé et le "Tunnelling Request" renvoyée par la passerelle KNXnet/IP. S'il n'y a pas d'erreurs de transmission, le programme client envoie un acquittement ("Tunnelling Ack") avec un code d'erreur (appelé "status") "00" (zéro).
9. S'il s'agit d'une demande de lecture de l'état d'un actionneur, le programme client reçoit un « télégramme » de type "Tunnelling Request" contenant la donnée

L'API knxnet

On se propose d'utiliser l'API knxnet (***développée par Adrien Lescourt***) pour gérer la communication entre un programme client écrit en python et une passerelle KNXnet/IP.

Les principaux objets et méthodes de la librairie "knxnet" sont les suivants :

"ServiceTypeDescriptor" : objet énumératif qui contient les différents types de télégrammes possibles:

Type du télégramme	Nom de l'attribut correspondant
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**

La création d'un objet représentant un futur télégramme KNX, se fait via la méthode *'create_frame'* de la classe knxnet. Le type du télégramme est le premier argument de ce constructeur. Il est suivi de paramètres qui dépendent du type de télégrammes à créer :

a) CONNECTION_REQUEST

attribut	description
`control_endpoint`	De la forme (adresse IP, Port) : utilisé pour envoyer ou recevoir les objets qui représentant des télégrammes de type *Connection Request/Response, Connection_State Request/Response et Disconnect Request/Response*.
`data_endpoint`	De la forme (adresse IP, Port) : utilisé pour envoyer et/ou recevoir les objets représentant des télégrammes de type *Tunnelling Request/Ack*

b) CONNECTION_RESPONSE

attribut	description
`channel_id`	Identifiant du canal alloué par la passerelle KNX/IP pour communiquer avec le programme client. La passerelle peut communiquer avec plusieurs clients en même temps grâce à la notion de canal.
`status`	Statut de la connexion : 0 (zéro) : OK. Sinon: message d'erreur (voir doc).
`data_endpoint`	Voir (a)

c) CONNECTION_STATE_REQUEST

attribut	description
`channel_id`	Voir (b)
`control_endpoint`	Voir (a)

d) CONNECTION_STATE_RESPONSE

attribut	description
`channel_id`	Voir (b)
`status`	Statut de la connexion demandée par la "Connection State Request" : 0 "Connection State Request" acceptée, message d'erreur sinon (voir doc).

e) TUNNELLING_REQUEST

attribut	description
`dest_addr_group`	L'adresse de groupe de l'équipement (device) à commander (lecture ou écriture). Pour créer un address group à partir d'une chaîne de caractères utilisez cette méthode : `dest_addr_group = knxnet.GroupAddress.from_str("x/y/z")`
`channel_id`	Voir (b)
`data`	La valeur à envoyer vers la passerelle KNX/IP (commande d'un device) ou à recevoir à partir de la passerelle KNX/IP (dans le cas où Tunnelling request est générée par la passerelle). Cette valeur est comprise entre 0 et 255. Dans le cas d'une lecture, cette valeur n'a pas de signification.
`data_size`	Taille du champ "data" envoyé.
`apci`	Ce champ spécifie la nature de la requête : 0x0 == demande de lecture (envoyée par le client); 0x1 == réponse à une requête précédente (cas d'un Tunnelling request envoyée par la passerelle suite à un Tunnelling request envoyé par le client ; 0x2 == demande d'écriture (envoyée par le client).
`data_service`	Champ optionnel. Il peut avoir trois valeurs possibles : 1. Data.request (0x11) : correspond aux Tunnelling requests envoyées par le client à la passerelle 2. Data.confirmation (0x2e) : correspond à un Tunnelling request envoyée par la passerelle en réponse à un Tunnelling request de type « Data request ».

	3. Data.response : correspond au second un Tunnelling request envoyée par la passerelle pour répondre à une demande de lecture (Tunnelling request de type Data.request)
`sequence_counter`	Champ optionnel.

f) TUNNELLING_ACK

attribut	description
`channel_id`	Voir (b)
`status`	Statut de la requête envoyée dans le "Tunnelling Request". 0 si c'est OK. Valeur différente de 0, si erreur (voir doc).
`sequence_counter`	Le même "sequence_counter" du télégramme "Tunnelling Request" à acquitter.

g) DISCONNECT_REQUEST

attribut	description
`channel_id`	Voir (b)
`control_endpoint`	Voir (a)

h) DISCONNECT_RESPONSE

attribut	description
`channel_id`	Voir (c)
`status`	C'est le status de votre demande de déconnexion envoyé dans le "Disconnect Request". Si c'est 0 (zéro) c'est que tout est bon et votre "Disconnect Request" a été accepté. Si la valeur est différente de 0, alors c'est un message d'erreur (voir doc).

Exemple : pour demander à la passerelle de créer un télégramme de type "Connection Request", il faut appeler la méthode create_frame avec les paramètres suivants :

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

Le code suivant (incomplet) permet de démarrer l'initialisation de la connexion avec une passerelle KNX/IP :

```
# -*- coding: utf-8 -*-
import socket,sys
from knxnet import *

gateway_ip = "adresse IP da la passerelle KNXnet/IP"
gateway_port = Port sur lequel la passerelle KNXnet/IP écoute"

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

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
```

A vous de finir le reste!